Analisador estático para reconhecimento de tipos de expressões para a linguagem CQL

Isabella Beatriz da Silva*, graduanda em Engenharia de Computação. Deborah Alves Fernandes†, Professora Associada *EMC/UFG. †INF/UFG. E-mails: isabella_beatriz@discente.ufg.br*, deborah.fernandes@ufg.br†

***Resumo*—A otimização de compiladores e analisadores é importante para melhoria da experiência do profissional que utiliza linguagens de programação. Isso não poderia diferir quando se trata de empresas de tecnologia que oferecem produtos de *software* para seus clientes. No caso da linguagem CQL, a sua facilidade de uso é ainda mais crucial porque tem como seus usuários profissionais não técnicos, como, por exemplo, pessoas de marketing. Este projeto consiste no desenvolvimento de um analisador estático para proporcionar a inferência do tipo do retorno de expressões CQL. Esse utiliza as técnicas de teoria de compiladores. Assim o usuário conseguiria ter a segurança que suas consultas resultam no tipo de dado esperado. Essas regras de inferência foram implementadas para quatro grupos de expressões da linguagem, com uma estrutura a qual permite sua expansão. Este artigo apresenta como foi o processo de desenvolvimento e a teoria por trás da modelagem da aplicação.**

***Palavras-chave*—Compiladores, Análise léxica, Análise sintática, Análise semântica**

***Abstract*—The optimization of compilers and parsers is important for improving the experience of the professional who uses programming languages. This could not be different when it comes to technology companies that offer software products to their customers. In the case of the CQL language, its ease of use is even more crucial because it has non-technical professionals, such as marketing people, as its users. This project consists of developing a static parser to provide type inference of the return of CQL expressions. This uses the techniques of compiler theory. Thus the user could be assured that his or her queries result in the type of data I expect. These inference rules have been implemented for four groups of language expressions with a structure that allows for expansion. This article presents the development process and the theory behind the modeling of the application.**

***Index Terms*—Compilers, Lexycal analyzer, Syntax analyzer, Semantic analyzer**

I. Introdução

Com a evolução do marketing digital, diversos recursos tecnológicos são utilizados para otimizar resultados - como anúncios pagos em mídias como redes sociais e ferramentas de busca, melhoria de usabilidade, testes AB (comparação entre variações de texto e estilo de um mesmo site) - melhorando a experiência do usuário. Outra forma de otimizar resultados é a web personalização. Segundo o livro *The Adaptative Web* [1], a personalização consiste em utilizar o conhecimento de dados do contexto do usuário e criar uma experiência personalizada, a qual entrega uma vivência única para os visitantes daquele site.

O conceito de personalização web pode ser ilustrado por uma situação de um conteúdo promocional em um site de vendas de camisetas. Neste, há a oferta de diversas camisetas com características variadas (estampas, tamanhos, etc.) mas, há também oportunidades de chamar a atenção do usuário que tenha interesse em outros segmentos, tais como cachorros ou filmes. Neste caso pode-se realizar a especificação de conteúdo em partes do site (banners, botões, etc.) inserindo conteúdo promocional nesses dois segmentos visando a personalização para esses interesses.

Outro recurso do marketing digital são os serviços de gerenciamento de conteúdo (do inglês *Content Management System*, CMS), nesse caso um *headless* CMS. Segundo a definição da *JamStack* [2], um *headless* CMS se refere ao gerenciamento de conteúdo sem a necessidade de uma camada de visualização de dados, ao contrário de um *CMS* tradicional, em que é necessário ter o site, e-mail ou aplicativo que o conteúdo será utilizado. O conteúdo pode ser usado onde o desenvolvedor escolher contanto que siga a estrutura definida. Ou seja, no exemplo descrito no parágrafo anterior, que se trata de uma personalização baseada em interesses, o conteúdo mostrado no site de camisetas poderia ser cadastrado no sistema *headless CMS*.

A Croct[3] é uma empresa do ramo de web personalização que disponibiliza para os clientes uma plataforma que integra um *headless CMS* com o gerenciamento de personalizações. O nome dado a esta plataforma é *Personalization Management System* (PMS). Esta apresenta três conceitos:

- Audiência: São os critérios que determinam um conjunto de usuários com características em comum.
- Slots: Definem as estruturas do conteúdo usando atributos tipados.
- Experiência: É a regra de personalização que combina o conteúdo para uma audiência específica.

A *Contextual Query Language* (CQL) é a uma linguagem de consulta de dados de alto nível desenvolvida pela Croct. Sua finalidade é permitir que qualquer cliente da empresa - seja este um técnico em programação, o ou não - consiga ler, escrever, e compreender facilmente as expressões da linguagem. Por esse motivo sua sintaxe é muito semelhante ao inglês.

Uma das aplicações da CQL é para a criação de uma audiência, esta é definida a partir de um critério para agrupar usuários com as mesmas características. Esse critério é escrito na forma de uma expressão na linguagem de programação CQL que deve retornar um valor verdadeiro ou falso. Isto é, se o usuário se enquadra ou não ao critério determinado. Outra aplicação é para a definição de conteúdos dinâmicos, ou seja, um conteúdo que muda em determinadas vezes conforme o contexto do usuário. A única restrição é que o retorno dessa expressão precisa respeitar o tipo do atributo do componente.

No entanto, o fato de permitir que qualquer pessoa com conhecimento técnico ou não de computação escreva expressões permite que estas sejam realizadas erroneamente. Essa situação provoca transtornos fazendo com que as expressões retornem valores diferentes do esperado. Como em uma audiência retornar um valor diferente de verdadeiro ou falso, isso faz com que os usuários nunca vejam a personalização planejada. Outro caso é na definição de conteúdo dinâmico, nos casos em que o conteúdo não é compatível com o tipo do atributo do componente, o que pode interromper o funcionamento esperado do site. Por isso, profissionais de marketing e até mesmo desenvolvedores de software podem se sentir inseguros ao usar o PMS para configurar eles mesmos uma web personalização. Uma das formas de solucionar esse problema é através da oferta de uma análise dos retornos das expressões CQL.

Este projeto final de curso tem por objetivo a implementação de uma solução para o reconhecimento de tipos da linguagem CQL utilizando técnicas de teoria de compiladores. O analisador desenvolvido a ser apresentado consegue identificar o tipo do retorno de expressões. Para este projeto, de todas as possibilidades de constituição de expressões, foram selecionadas algumas. É importante constar que o analisador pode receber atualizações e incrementos passando a analisar mais expressões.

Este trabalho está organizado da seguinte forma: um visão geral da base teórica de compiladores utilizada será apresentada na seção II; na seção III serão apresentados o detalhamento da implementação da solução, a gramática da linguagem, as regras semânticas, um algoritmo de alto nível e as tecnologias usadas para o desenvolvimento do analisador; na seção IV, os resultados e discussões; conclusão e referências ao final.

II. Fundamentação teórica

As técnicas da teoria de compiladores usadas na implementação desse projeto e uma linguagem apresentação da linguagem CQL serão trabalhadas nesta seção.

A. A linguagem CQL

A CQL é uma linguagem desenvolvida pela empresa Croct que permite a recuperação de dados do contexto do usuário em páginas da web tais como nome, cidade, interesses e outros. Com isso a linguagem é utilizada por clientes da empresa que precisam de consultar informações, estabelecer regras de negócio e tomar decisões baseadas nesses dados.

A Tabela I mostra, com o objetivo de comparação, a sintaxe da CQL e exemplos de como retornar o nome de um usuário usando outras linguagens de programação como Python, PHP e Javascript. Apesar das semelhanças com o inglês simples, a CQL é uma linguagem regular em que cada expressão tem sua sintaxe definida. Por isso requer consultas escritas de forma clara, inequívoca e seguindo regras estritas.

A linguagem tem acesso a algumas variáveis para as quais o tipo é conhecido, essas são chamadas de variáveis de contexto. Esse é um conceito interno da Croct e armazena valores do contexto do usuário através do rastreamento de eventos na web, ou, por exemplo, do enriquecimento desses valores pelo usuário através de formulários preenchidos. Alguns exemplos são mostrados na Tabela II.

Tabela I. Comparação simplificada entre sintaxes de linguagens de programação para obtenção do nome de um usuário.

LINGUAGEM	**SINTAXE**
CQL	user's name
Python	user.name
PHP	$utilizador>nome
JavaScript	user.name

Tabela II. Exemplos de variavéis de contexto [4].

VARIÁVEL	**TIPO**	**EXEMPLO**
browser's type	string	"web"
now	string	"2015-08-31T01:02:03"
page's url	string	"https://www.croct.com/plans"
location' stateCode	string	"GO"
campaign's term	string	"web personalização"
user's age	number	21

B. O Compilador

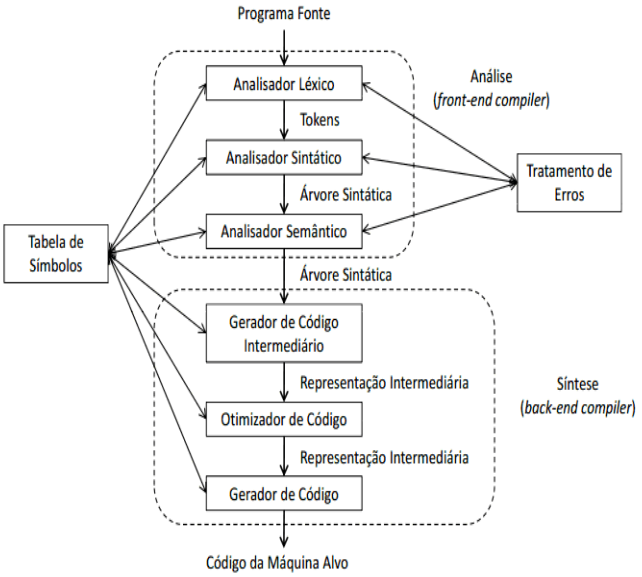
Compiladores são softwares responsáveis por lerem o código-fonte e o traduzem para um programa equivalente em outra linguagem. Para tal há um processo de compilação em que existem duas fases distintas, uma de análise e outra de síntese. A primeira trata o código em questões de palavras, sintaxe e semântica apontando erros cometidos na escrita do programa pelo programador, há a construção de uma árvore sintática que será utilizada na próxima etapa. A segunda fase acontece se não há erros identificados na primeira. Nesta há a tradução do código-fonte para outra linguagem que pode ser uma intermediária, final ou ambas. A Figura 1 apresenta a arquitetura larga de um compilador,

A teoria de compiladores pode ser utilizada em diversas aplicações na totalidade ou em partes, aproveitando-se de alguma técnica para a solução de algum problema. Muitas ferramentas utilizam as técnicas de análise para a implementação de um verificador estático. Este realiza a leitura de um programa e tenta encontrar erros sem executá-lo. Outra a aplicação é a análise de tipos, que como neste trabalho, utiliza das fases de análise para produzir a saída desejada, no caso o tipo da expressão avaliada. Por esse motivo para este projeto o foco será na primeira etapa do compilador, compreendendo todas as fases de análise. Nas próximas subseções serão descritas cada uma delas.

1) Análise léxica: A análise léxica é também conhecida como *scanner*, é a primeira fase do compilador e compreende a leitura de caractere a caractere do programa fonte, agrupando-os com o objetivo de classificá-los. Para o entendimento e implementação da análise léxica existem três conceitos importantes:

- Alfabeto: É um conjunto finito de símbolos ou caracteres

Fig. 1. Arquitetura larga de um compilador [5].



que são aceitos na linguagem. Por exemplo podem ser letras, dígitos ou caracteres especiais como +, @ e !. Esses são os caracteres lidos do programa fonte.

- Lexema: Trata-se de uma sequência finita de símbolos, uma palavra.
- Padrão: Representa as formas que uma cadeia de caracteres pode assumir. São utilizadas as gramáticas regulares ou expressões regulares como formalismos descritores para esses padrões.
- Tokens: É a classe a qual o lexema pertence. Se um lexema é reconhecido por um padrão ele é identificado como pertencente à classe definida por este padrão. Por exemplo, um identificador é a classe a qual os nomes de variáveis, funções e procedimentos pertencem.

Dentro dessa fase é feito o reconhecimento de erros léxicos como, por exemplo, apontar caracteres não aceitos pelo alfabeto da linguagem, uma cadeia de caracteres que não é reconhecida como um token, erros de escrita de palavras-chave, identificadores ou operadores entre outros.

A implementação da fase de análise léxica se dá pela aplicação de conceitos da teoria de linguagens formais e autômatos. Desta são utilizados os formalismos descritores e reconhecedores. Como a linguagem a ser reconhecida na análise léxica é relativamente simples, são utilizadas os formalismos para linguagens do tipo 3 ou também chamadas de regulares. As linguagens regulares podem ser formalmente descritas através de expressões regulares e gramáticas regulares [5]. Para a identificação de padrões da linguagem são adotados os formalismos reconhecedores autômatos finitos. Um autômato finito é uma máquina de estados a qual possui um conjunto de estados, sendo um inicial e alguns deles finais. Para esses autômatos, a medida em que é feita a leitura dos caracteres de entrada, o controle da máquina passa de um estado para o outro, respeitando um conjunto de regras de transição. Se ao final da leitura dos símbolos de entrada o autômato reconhece um estado final, então a cadeia de caracteres de entrada pertence à linguagem, caso contrário não pertence à linguagem. Essas máquinas podem ser divididas em dois tipos:

- Determinísticos (AFD): É um autômato finito em que cada símbolo de entrada possui uma única ou nenhuma saída, isto é, para cada entrada existe um estado para o qual pode ir a partir de seu estado atual [6]. Como é apresentado no exemplo da Figura 2.
- Não determinístico (AFN): É um autômato finito em que um símbolo de entrada tem duas ou mais saídas, ou seja, pode estar em vários estados simultaneamente. Isso possibilita ao algoritmo tentar inferir algo sobre a entrada. Como é apresentado no exemplo da Figura 3.

Fig. 2. Exemplo de um autômato finito determinístico.

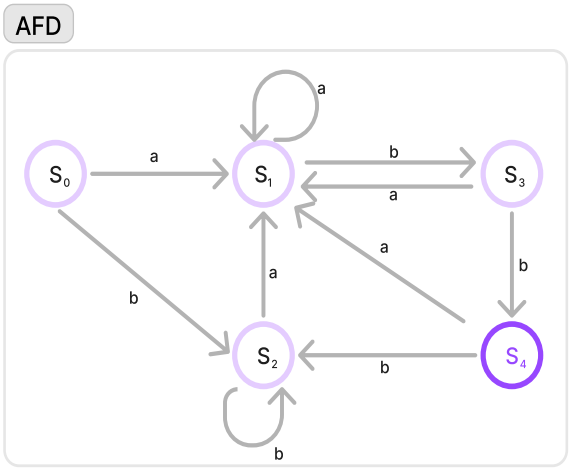
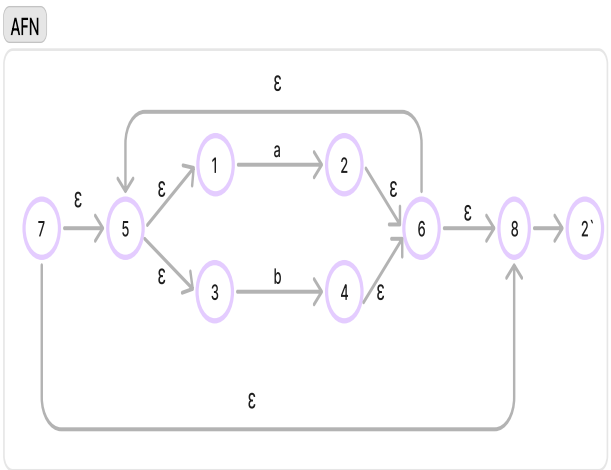


Fig. 3. Exemplo de autômato finito não determinístico.



2) Tabela de símbolos: A tabela de símbolos é responsável por manter informações sobre todos os identificadores do programa fonte com atributos importantes, tais como tipo e

escopo [7]. Essa tabela tem como objetivo tornar o compilador mais eficiente. Para isso, interage com todas as fases de análise e síntese. Na etapa de análise léxica os identificadores podem ser encontrados e adicionados a ela. Durante as fases de análise seguintes a tabela poderá ser atualizada com novas informações descobertas ou consultada.Na fase de síntese, ao gerar o código intermediário, o tipo da variável é utilizado para determinar quais as instruções serão emitidas. Durante a otimização, o escopo de cada variável pode ser colocado na tabela para ajudar na atribuição de registros. Como também, a localização da memória determinada na etapa de geração do código pode ser mantida na tabela de símbolos.

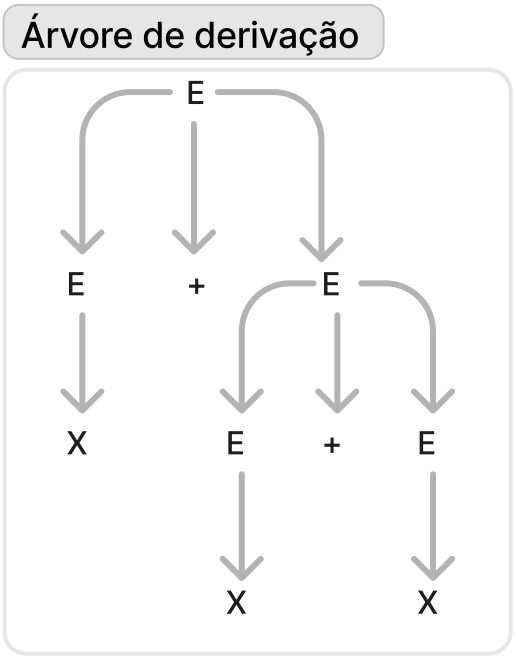
3) Análise sintática: Também conhecida como *parser*, essa fase é responsável pelo agrupamento de tokens em uma representação na forma de árvore sintática. Essa árvore é construída como resultado do reconhecimento do padrão de escrita de sentenças da linguagem. Para isso são adotados outros formalismos da teoria de lingugens formais e autômatos. A linguagem para escrita das sentenças, ou seja, agrupamento de tokens, é mais complexa que a de palavras. Neste caso as linguagens regulares são insuficientes e faz-se necessário utilizar de formalismos mais complexos. As linguagens livres de contexto ou tipo 2 são adequadas para a análise sintática. Para escrever o padrão de sentenças possíveis na linguagem é utilizado o formalismo descritor gramática livre de contexto, para a identicação dos padrões é utilizado o formalismo reconhecedor autômato de pilha. Abaixo algumas definições importantes para o entendimento da fase:

- Gramáticas livre de contexto: Define um conjunto de regras hierárquicas para construções da linguagem de programação. A gramática da Tabela III é um exemplo que representa uma gramática para regras aritméticas. Essa gramática é composta por: (a) Símbolos terminais: são os que não podem gerar produções, como '+', '-' e 'número'; (b) símbolos não terminais: são os capazes de gerar produções, como 'EXPRESSÃO' e 'OPERANDO' na gramática de exemplo da Tabela III; (c) um conjunto de produções compostas pelo lado esquerdo à seta com um único símbolo não terminal, e o lado direito com uma sequência de símbolos que podem ser terminais (tokens) ou não; (d) um símbolo de partida, também chamado de incial pertencente ao conjunto de não terminais.
- Autômatos de pilha (AFP): É um autômato finito com uma memória auxiliar na estrutura de uma pilha. Ele escolhe a transição a partir do símbolo no topo da pilha. Esse autômato funciona como um reconhecedor de linguagens livre de contexto e é um modelo equivalente a gramáticas livres de contexto.
- Derivação: Consiste em substituir os símbolos não terminais utilizando as regras definidas pela gramática por terminais. Isso permite identificar se certas cadeias de caracteres pertencem ou não a linguagem. O resultado desse processo é a estrutura hierárquica que a linguagem assume. Essa derivação é chamada mais à diteira, quando o símbolo substituído é não-terminal mais à direita. Uma derivação é chamada mais à esquerda quando o caractere substituído é o não-terminal mais à direita.
- Árvores sintáticas: Também conhecida como árvore gramatical, esta representa a derivação de uma cadeia de linguagem a partir de um símbolo de partida utilizando as regras gramaticais. Ela é composta por uma raiz(também chamada de símbolo de partida), folhas(tokens ou palavra vazia), nós internos(um não terminal) e a partir de um nó interior não terminal, os filhos daquele nó da esquerda para direita representam uma produção. Como por exemplo, com base na gramática apresentada na Tabela III, é feita uma representação para a expressão de entrada 'x+x*x' em uma árvore sintática demostrada pela Figura 4.

Tabela III. Gramática de exemplo de precedência de operadores.

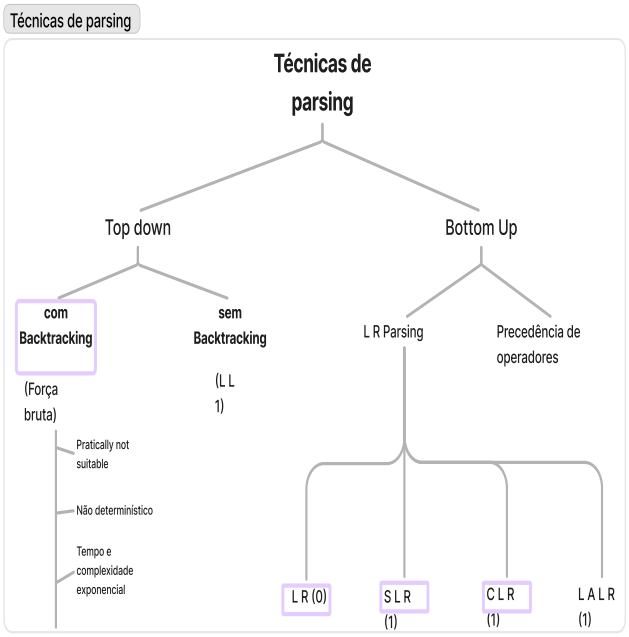
REGRAS
EXPRESSÃO -> OPERANDO + OPERANDO
EXPRESSÃO -> OPERANDO - OPERANDO
EXPRESSÃO -> OPERANDO
OPERANDO -> número

Fig. 4. Árvore de derivação para a expressão x+x*x.



As técnicas de derivação podem ser agrupadas como mostradas na Figura 5. Nela estão dispostos dois grandes grupos divididos em abordagem ascendente e descendente. Na primeira abordagem, também conhecida como *Top-Down* ou preditiva, há a geração de uma derivação mais à esquerda e a formação da árvore sintática de cima para baixo. Na segunda, também chamada de *Bottom-Up*, há a produção de uma derivação mais à direita e a formação de uma árvore das folhas para a raiz. Para esse projeto foi usada a abordagem ascendente que ainda será melhor detalhada.

Fig. 5. Diagrama de técnicas de *parsing*.

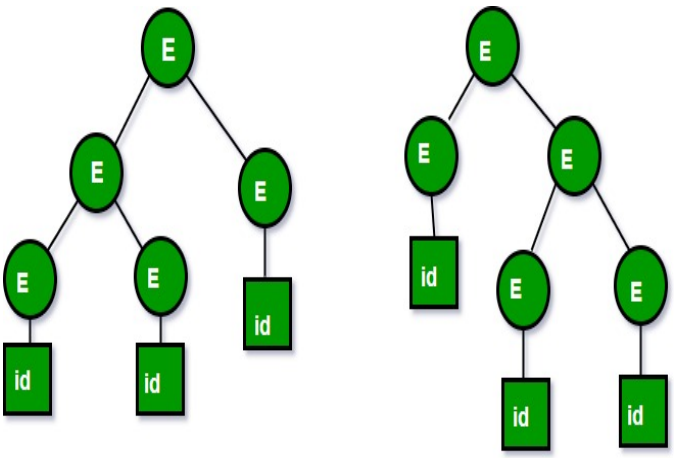


Independente do algoritmo utilizado, a derivação deve sempre produzir o mesmo resultado, ou seja, a mesma árvore de derivação, caso seja possível gerar mais de uma árvore de derivação como resultado há ambiguidade na gramática. Na Tabela IV é apresentada uma gramática ambígua, ao derivar a entrada "id+id+id" é possível obter duas árvores distintas como mostrado na Figura IV.

Tabela IV. Gramática exemplificando regras ambíguas.

REGRAS
E -> E + E
E -> E + id

Fig. 6. Exemplo de duas possíveis derivações de uma gramática ambígua [8].



Uma característica importante com relação à gramática é a precedência, que também pode ser descrita como associação. A precedência dos operadores interfere no agrupamento e na avaliação dos operandos nas expressões da linguagem. Essa prioridade de um operador é significante somente quando outros operadores com precedência maior ou menor estão presentes. As expressões com operadores de maior importância são avaliadas primeiro. Dizem que os operadores com uma precedência mais alta têm uma associação mais apertada. Por exemplo [9], para a linguagem C, as expressões de entrada são avaliadas seguindo as regras de associatividade de operadores como na Tabela V.

Tabela V. Exemplo de associatividade de operadores em C.

EXPRESSÃO	**ASSOCIAÇÃO**
a & b ‖ c	(a & b) ‖ c
a = b ‖ c	a = (b ‖ c)

4) Análise sintática Ascendente: Um analisador *bottom-up*, que constrói uma árvore sintática de baixo para cima, ou seja, incia pelas folhas (tokens recebidos do *scanner*) até alcançar a raiz (símbolo inicial da gramática), aplicando as produções em inverso. Ao longo do caminho, o analisador procura substratos que se igualam ao lado direito de alguma produção. Quando o encontra, ele realiza a ação de redução, ou seja, substitui o lado esquerdo não terminal pelo lado direito correspondente. O objetivo é realizar reduções até encontrar o símbolo de partida e nenhum outro símbolo a ser lido na entrada, relatando assim uma derivação bem sucedida. Esse tipo de análise é denominada *shift-reduce*, em que durante o processo de reconhecimento de uma entrada, a entrada é consumida e deslocada para uma pilha auxiliar, quando é possível realizar uma redução, ou seja, o inverso de uma derivação, há o desempilhamento de elementos da pilha e a substituição do lado direito de uma produção pelo lado esquerdo realizando assim uma redução.

Este analisador requer algumas estruturas de dados, como um *buffer* de entrada para armazenar a cadeia de entrada e uma pilha para o armazenamento e acesso às regras de produção. As operações básicas desse analisador são: *shift* (mover símbolos do *buffer* de entrada para a pilha com o objetivo de reduzir as tentativas do analisador para a construção do *parser*), *reduce* (uma regra de produção é retirada de uma pilha e o resultado de uma regra de produção é empilhado na pilha), aceitar (se apenas o símbolo de início estiver presente na pilha e o símbolo de entrada estiver vazio), erro (quando a construção não é reconhecida pela gramática).

Os modelos de algoritmos para análise *bottom-up* que se encaixam na categoria dos *parsers shift/reduce*, por ordem de complexidade e capacidade de reconhecimento são: LR(0), SLR(1) ou *Simple* LR, LALR(k) e LR(k). "L"significa esquerda (do inglês *left*), ou seja, uma entrada é consumida da esquerda para a direita. "R"de direita (do inglês *right*) informando que será produzida uma derivação mais à direita. O "k"indica quantos elementos serão utilizados à frente da entrada para a tomada de decisão sobre qual regra será empregada para o processo de reconhecimento [5].

Os analisadores *shift-reduce* possuem uma tabela de análise construída a partir de um autômato finito denominado autômato

LR. Esse autômato é constituído de estados que possuem conjuntos canônicos de itens, cada item é uma regra gramatical com um ponto do lado direto que informa quais símbolos já foram consumidos no processo de análise e quais ainda restam para que uma produção seja reduzida.

A diferença entre os tipos de analisadores *shift-reduce* dos modelos citados está forma como essa tabela de análise é gerada. No LR(0) não se leva em consideração os símbolos à frente da entrada para a tomada de decisão sobre qual ação será realizada. No SLR(1) verifica se o símbolo à frente pertence ao conjunto seguinte (conjunto de terminais que podem aparecer à direita de um símbolo não-terminal em uma sentença válida [5]) para realizar uma ação. O LALR e o LR(k) só realizam uma ação quando o símbolo à frente é o esperado para a aplicação de um item gramatical. Esses últimos possuem tabelas de análise maiores e também maior capacidade de reconhecimento. Detalhes desses modelos podem ser verificados em [5].

Neste trabalho, será implementado um analisador sintático ascendente do tipo *Simple*LR(k). O *parser* SLR é considerado o mais simples dos métodos LR. Ele utiliza um AFD com o conjunto de itens LR(0), o qual utiliza a marca seguinte na cadeia de entrada para dirigir suas ações, sendo que garante que existe uma transição no AFD e utiliza o conjunto *FOLLOW* de um não terminal para decidir se a redução deve ser efetuada. As produções reduzidas são escritas apenas no conjunto.*FOLLOW* da variável cuja produção é reduzida. A construção da tabela de análise de SLR segue quatro passos. Por exemplo, se na tabela de análise tivermos múltiplas entradas, então diz-se que se trata de um conflito *shift/reduce*.

Como um exemplo da construção de um analisador SLR [10], partindo da gramática livre de contexto da Tabela VI encontra-se a gramática aumentada. Na qual os símbolos terminais são a,b, os não-terminais são S,A e o símbolo inicial é S'. Depois são encontrados os conjuntos *FOLLOW* da gramática para construção da tabela SLR como mostrada na Figura 7.

Tabela VI. Gramática livre de contexto para gramática aumentada.

GRAMÁTICA	**GRAMÁTICA AUMENTADA**
S–>AA	S'–>.S
A–>aA|b	S–>.AA
	A–>.aA
	A–>.b

5) Análise semântica: Essa fase realiza a verificações relacionadas ao sentido do programa fonte e captura informação de tipos para a fase de síntese. Utiliza como entrada a estrutura hierárquica formada na etapa de análise sintática [5]. Uma das verificações mais importantes dessa fase é a checagem de tipos, esta verifica se os tipos dos operandos são compatíveis com os operadores conforme a gramática da linguagem. Existem conceitos importantes na análise semântica como apresentados a seguir.

- Tradução dirigida pela sintaxe: É um método em que as regras de tradução são guiadas pelas construções sintáticas.

Fig. 7. Tabela SLR para a gramática da Tabela VI.

	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			accept		
2	S3	S4		5	
3	S3	S4		6	
4	R3	R3	R3		
5			R1		
6	R2	R2	R2		

Isto é, o processo e as árvores sintáticas são utilizadas para orientar a análise semântica e a tradução do programa fonte. Pode ser uma fase separada de um compilador ou pode-se utilizar uma gramática de atributos, que é a gramática convencional aumentada com informações para controlar a análise e a tradução. Esses atributos associados podem representar qualquer valor necessário para análise, como tipo, localização na memória ou o que for preciso.

- Definição dirigida pela sintaxe (DDS): É uma generalização de uma gramática livre de contexto na em que símbolo possui um conjunto de atributos associados. Esses atributos podem ser herdados ou sintetizados. As regras semânticas possuem uma ordem de dependência entre os atributos, a qual define a ordem de análise. A partir da derivação de uma cadeia de entrada, as regras semânticas definem valores para os atributos nos nós da árvore gramatical. O resultado dessa análise é uma árvore gramatical anotada e o processo de computar os valores dos atributos é chamado de decoração da árvore ou anotação.
- Atributos sintetizados: Quando os valores dos atributos são computados a partir dos atributos de seus filhos em uma árvore de derivação sintática.
- Atributos herdados: Quando os valores dos atributos são computados através do valor dos atributos de seu pai ou de seus irmãos na árvore de derivação.
- Esquemas de tradução: Se refere a descrição concreta de como a definição dirigida pela sintaxe será implementada. Nela são descritos quais atributos cada símbolo terminal ou não-terminal possui, assim como as ações semânticas de cada produção gramatical.
- S-atribuído: É um tipo de esquema de tradução que utiliza somente atributos sintetizados. Esse esquema é implementado através de analisadores ascendentes e associam as ações semânticas ao lado direito das regras de produção. Nesse caso, os atributos podem ser avaliados conforme a cadeia de entrada é avaliada pelo analisador *bottom-up*, e para cada redução são computados os atributos do símbolo do lado esquerdo utilizando os que estão na pilha

semântica.

III. Solução proposta

Nesta seção serão apresentados a solução proposta para o problema de detecção automática de tipos resultantes de expressões constituídas pelos usuários da plataforma para a personalização de conteúdo web (analisador de expressões) e o detalhamento das etapas do projeto. Para tal será disponibilizada uma figura de desenho de experimento.

A Figura 8 mostra o desenho da solução proposta para o analisador de expressões. Ele recebe como entrada uma expressão escrita em linguagem CQL e emite na saída o tipo associado a ela. A parte responsável por essa análise de expressões implementa as fases de análise léxica, sintática e semântica da arquitetura de um compilador que serão descritas mais adiante.

A. As etapas do desenvolvimento

1) Levantamento das expressões da linguagem CQL: Na etapa 1 mostrada na Figura 9, foi realizada o levantamento e análise da estrutura das expressões mais utilizadas pela plataforma de personalização. Além disso, nessa mesma etapa foi feito o levantamento de dados sobre a linguagem com relação a seus tokens, tipos, regras gramaticais e semânticas. Inicialmente o projeto foi especificado para analisar quatro tipos de expressões da linguagem CQL, mas com possibilidade para expansão. As expressões definidas inicialmente são as aritméticas, lógicas e relacionais em suas formas simples e compostas e as condicionais simples.

Tabela VII. Exemplo de cada tipo de expressões da linguagem CQL escolhidas.

CONJUNTO	**EXPRESSÃO**
Aritméticas	1 + 4 * 3
Aritméticas	cart's total + 10
Relacionais	4 >= 3
Relacionais	cart's total >= 3
Lógicas	true or false
Lógicas	(3 > 1) and (5 < 3)
Condicionais	"Retorno" if (3 < 4)

- Expressões aritméticas: Esse conjunto de expressões compreende as construções de: adição (+), subtração (-), multiplicação (*) e divisão (/). Tanto com números inteiros ou decimais para os operandos quanto com outras expressões aritméticas como exemplificado na Tabela VII.
- Expressões relacionais: Esse conjunto de expressões compreende as comparações entre valores numéricos, decimais ou inteiros. Os operadores desse conjunto são: maior que (>), maior ou igual que (>=), menor (<), menor ou igual que (<=) e igual (==). Como é exemplificado na Tabela VII.
- Expressões lógicas: Esse conjunto de expressões compreende operações lógicas booleanas com o uso dos operadores lógicos *and* (e) e *or* (ou) com os operandos sendo valores booleanos verdadeiro e falso ou expressões que retornam esse tipo de valor.
- Expressões condicionais: Para esse tipo de expressão foi escolhido trabalhar apenas com as condicionais simples, sem o "else". Expressões do formato em que depois do operador "if" encontra-se o teste de tipo booleano e antes do "if" o retorno que pode ser de qualquer tipo aceito pela linguagem como pode ser observado no exemplo da Tabela VII.

Tabela VIII. Alfabeto da linguagem CQL

DEFINIÇÃO	**SÍMBOLOS**
Dígito	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
Letras	{A, B, …, Z, a, b, …, z}
Demais caracteres	{, (vírgula), .(ponto), |, &, >, <, =, ?, ", ', *, +, -, /, (,) }

Tabela IX. Tokens da linguagem CQL definidos no analisador.

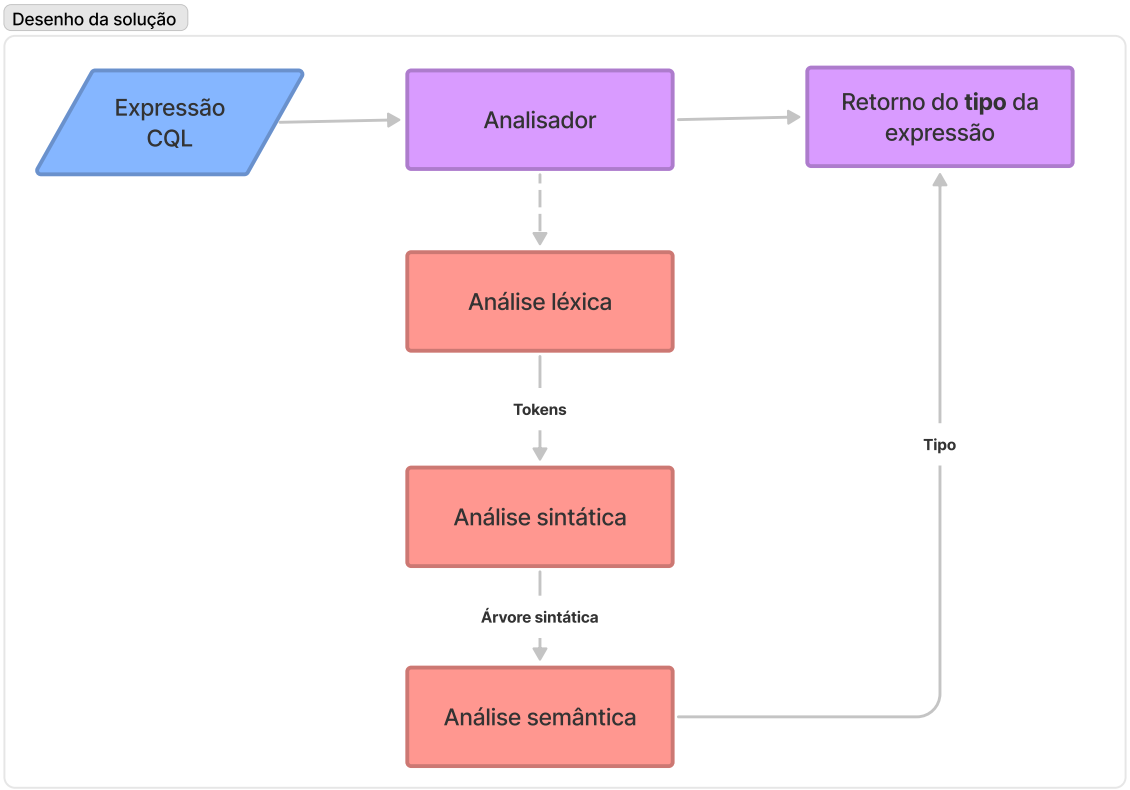
CLASSE	**CARACTERÍSTICA PADRÃO**	**SIGNIFICADO**
Literal	".*"	Constante literal
Num	"D*(/D)?"	Contante numérica real ou decimal
OPM	*, /, +, -	Operadores aritméticos
OPR	>, >=, <, <=, ==	Operadores relacionais
OPL	||, ??	Operadores lógicos
ID	L+(('s)? espaço? L*)*	Identificador

2) Analisador léxico: Na etapa 2 mostrada na Figura 9, foi projetado como seriam os analisadores visando definir quais técnicas seriam utilizadas para cada um e resultados esperados. O alfabeto e os tokens necessários foram definidos para as expressões levantadas. A partir disso o autômato finito determinístico foi modelado e implementado no *scanner* do analisador todos utilizando mapas, esse autômato é apresentado na Figura 10.

Os símbolos que compõem o alfabeto da linguagem definidos para esse projeto são mostrados na Tabela VIII. Os tokens definidos são utilizados pelas expressões escolhidas como mostrados na Tabela IX. No entanto, é importante ressaltar que a linguagem possui outros tokens que não foram contemplados neste projeto.

Para as variáveis de contexto, como são usadas em quase todas as expressões CQL, neste analisador foram adicionadas a tabela de símbolos com um tipo já associado e com a classificação de indicador. Por exemplo, a variável "location" e seu atributo "city" foram adicionados a tabela de símbolos

Fig. 8. Desenho da solução - analisador de expressões para a CQL.



como lexema “location’s city”, classe ID e tipo string. O funcionamento do analisador léxico é ilustrado na Figura 11, que para a entrada “cart’s total + 10”, o lexema “cart’s total” está na tabela de símbolos como um token da classe ID, os tokens retornador pelo analisador são o operador "+"da classe “OPM” e a constante numérica 10 da classe “NUM”.

3) Analisador sintático: Também na etapa 2 da Figura 9, foi realizado o projeto do *parser*. Para a análise sintática a técnica escolhida foi a análise ascendente SLR(1), com isso s regras gramaticais foram modeladas para contemplar as expressões selecionadas para esse projeto. Essas podem ser observadas na gramática especificada para esse analisador é apresentada na Tabela X.

O conjunto de regras 2 à 9 são as referentes a expressões aritméticas com precedência entre os operadores. Sendo que para as expressões aritméticas a gramática definiu a precedência entre os operadores para que fossem associados de forma automática. De forma que operações com multiplicação(*), divisão(/) sejam realizados antes de somas(+) e subtrações(-). As regras 13 e 14 são referentes a expressões relacionais, nas quais o operando pode ser uma expressão aritmética entre parênteses, constantes decimais e inteiras ou identificadores. O grupo de regras 15 à 19 contemplam as expressões lógicas, nas quais os operandos aceitos são descritos pelas regras 18-21 que devem ser um valor do tipo boolean ou uma expressão entre parênteses que retorne esse tipo. As regras 22 - 25 se referem a expressões condicionais simples e reutilizam as produções do não terminal “TEST” que foram definidos para as expressões lógicas e o “Resultado” pode ser de qualquer tipo. Sendo que ε é a palavra vazia e é a regra que define quando uma expressão termina. É importante ressaltar que os parênteses foram adicionados para remover a ambiguidade presente nas expressões compostas explicitamente.

4) Analisador semântico: Na etapa 2 da Figura 9, também foi definido que a análise semântica utilizada o esquema de tradução S-atribuído. Para isso foram definidas regras semânticas cujo objetivo é inferir o tipo associado aos tokens e expressões de entrada.

Para as expressões aritméticas as regras foram baseadas em definir que tipo de número seria retornado em cada operação. Com isso os tipos dos operadores da expressão são comparados e o retorno esperado é da forma mostrada na Tabela XI.

Todos os erros especificados na Tabela XI são de tipos incompatíveis para as operações matemáticas. Essas validações foram implementadas usando uma função para verificar a compatibilidade dos tipos e adicionadas em todas as regras da gramática para reduções de expressões aritméticas como é mostrado no algoritmo de alto nível do Código 1.

Código 1. Função de retorno do tipo de expressões aritméticas

```
function retornaTipoDaOperacaoMatematica(
    oprdEsquerda, oprdDireita)
  if (oprdEsquerda.tipo == 'NULO' ou
      oprdDireita.tipo == 'NULO')
      retorna 'tipos incompativeis'
```

Fig. 9. Etapas do desenvolvimento do projeto.


```
if (oprdEsquerda.tipo == 'inteiro' e
    oprdDireita.tipo == 'inteiro')
     retorna 'inteiro'
else
  retorna 'decimal'
```

Para as expressões relacionais foram especificadas as regras mostradas na Tabela XII para checagem de tipo. Nesse caso, foi definido que para quando os tipos são aceitos o retorno sempre será um valor do tipo booleano.

Isso foi implementado na regra sintática cuja produção é uma expressão relacional como é demonstrado no algoritmo de alto nível no Código 2. Em que mostra como a verificação semântica é adicionada a produção da gramática associada a regra 14.

Código 2. Função de retorno do tipo de expressões relacionais

```
'14':
if (( operando da direita == inteiro |
    decimal) and
(operando da esquerda == inteiro |
    decimal))}
   retorna 'boolean'
else
  retorna 'Tipos conflitantes'
```

Para as expressões lógicas foram especificadas as regras para verificação do tipo como mostrado na Tabela XIII. Para este conjunto foi definido que para quando os tipos são aceitos o retorno sempre será um valor do tipo booleano.

Tabela XIII. Resultado esperado em cada combinação de operandos para as expressões lógicas.

OPERANDO 1	**OPERANDO 2**	**RESULTADO**
Inteiro	Inteiro	Erro
Inteiro	Decimal	Erro
Inteiro	String	Erro
Inteiro	Boolean	Erro para operador *and*, Boolean para operador *or*
Decimal	Decimal	Erro
Decimal	String	Erro
String	Boolean	Erro
Boolean	Boolean	Boolean

Para as expressões condicionais simples a regra sintática associada foi definida como 'COND -> RESULT if TEST'. Com isso foram desenvolvidas duas regras semânticas:

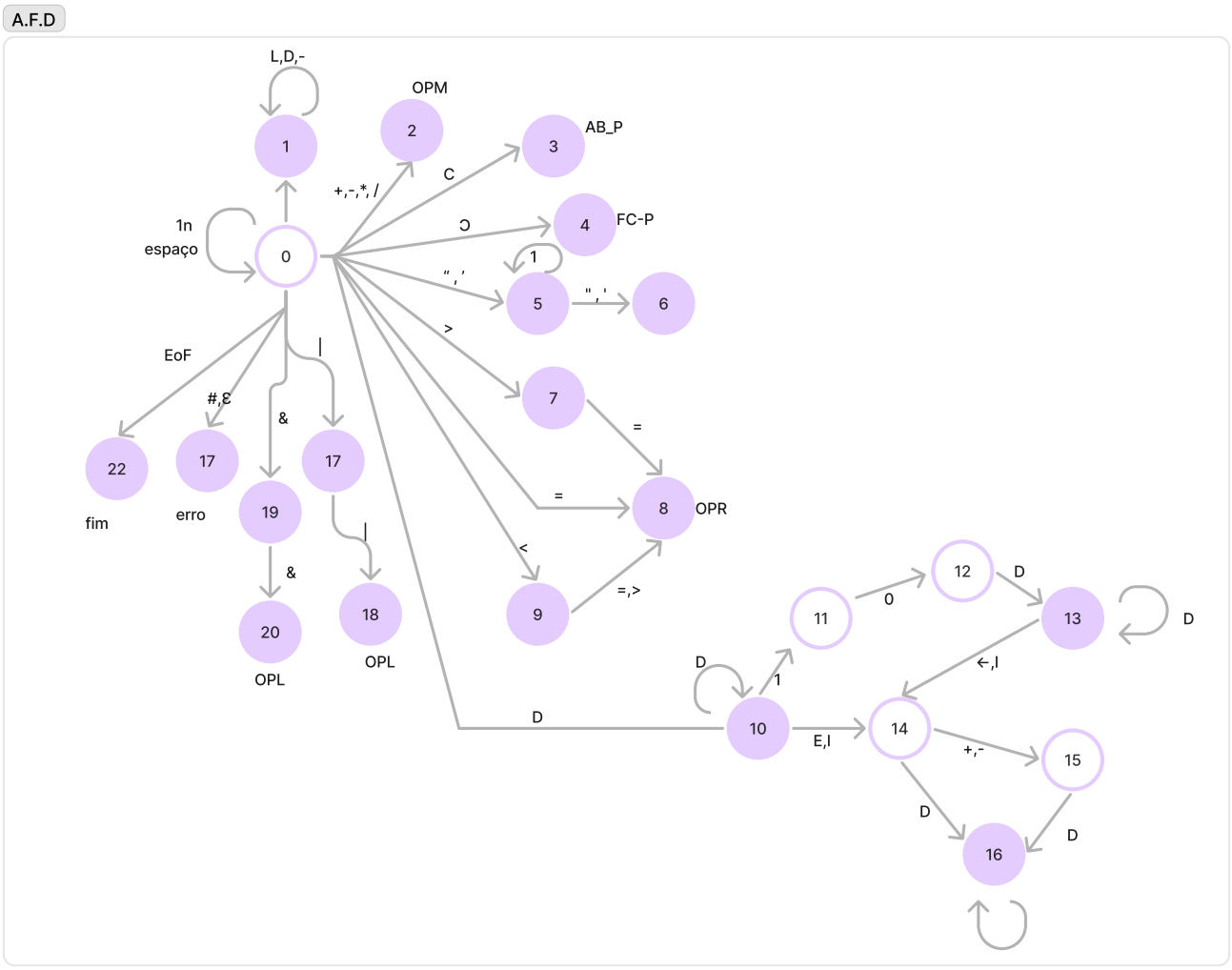
- Verificação se o não terminal 'TEST' é do tipo booleano como mostrado na Tabela XIV.

Tabela XIV. Resultado da regra semântica conforme o tipo do não terminal TEST.

TEST	**RESULTADO**
Boolean	Boolean
Diferente de Boolean	Erro de tipo não compatível

- se o 'TEST' é booleano, então verificar o tipo do não

Fig. 10. Autômato finito determinístico que define os tokens escolhidos.



terminal ‘RETORNO’ e este é o resultado da expressão condicional.

As regras semânticas especificadas são realizadas quando acontecem as reduções na análise sintática. Como é apresentada a lógica de implementação no algoritmo de alto nível no Código 3. Em que, “s” é o estado no topo da pilha, “a” é o próximo símbolo da entrada e “significado” é o mesmo que semântica.

Código 3. Algoritmo de alto nível para análise semântica

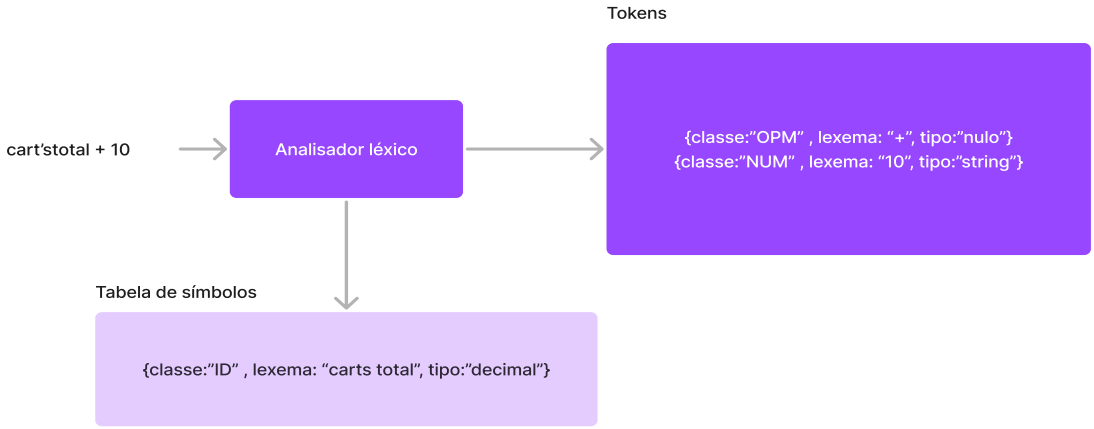
```
// Sendo ‘‘a’’ o topo da pilha
while
    if (ACTION [s,a] = shift t)
        empilha t
    else (ACTION [s,a] = reduce A–> B)
        executa regras de significado
        retorna novo token
        desempilha B carcateres da pilha
    else if (token)
        adiciona token na pilha de
            significado
        //Sendo t o topo da pilha
        empilha GOTO[t,A] na pilha
        imprima A–> B
    else if (ACTION [s,a] = ACCEPT )
    // entrada reconhecida
```

B. *As tecnologias e estruturas de dados utilizadas*

Na etapa 3 da Figura 9, as tecnologias e estruturas usadas para a construção do programa da implementação de cada analisador foram definidas. Para a linguagem de programação foi escolhida o TypeScript [11], sendo uma linguagem de programação fortemente tipada construída em cima do JavaScript. Essa linguagem adiciona tipagem ao JavaScript sem aumentar a complexidade de código. Por ter uma tipagem bem definida foi a linguagem escolhida para desenvolver o analisador desse projeto.

Para a elaboração de testes para confirmar o funcionamento do analisador foi utilizado o *framework* Jest [12]. Com ele é possível passar definir qual é a entrada e a saída esperada para cada expressão, isso é feito de forma paralela testando

Fig. 11. Análise léxica para os tokens do projeto



rapidamente o funcionamento do programa e garantindo que caso o código seja alterado todos os cenários são verificados novamente. Como pode ser observado na Figura'12, na qual é descrito um teste para expressões condicionais com quatro testes.

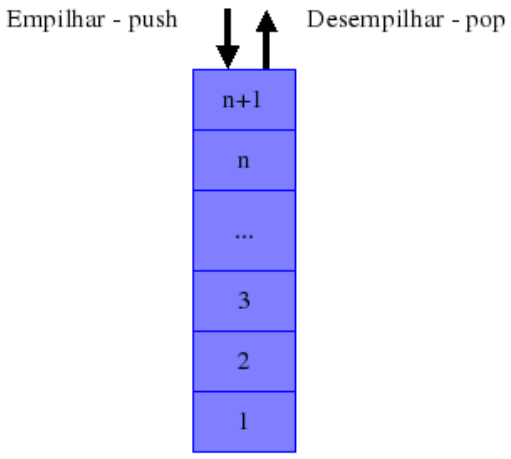
Fig. 12. Exemplo da sintaxe dos testes do analisador em Jest.

```
describe('Testes de expressões condicionais cujo o retorno pode ser nulo', () => {
    test('"30 if false" retorna inteiro', async () => {
        const result = await parser('30 if true')
        expect(result.tipo).toStrictEqual('inteiro')
    });
    test('"cart\'s total if (3>30)" retorna decimal', async () => {
        const result = await parser('cart\'s total if (2>3)')
        expect(result.tipo).toStrictEqual('decimal')
    });
    test('"location\'s city if (true and false)" retorna string', async () => {
        const result = await parser('location\'s city if (true and true)')
        expect(result.tipo).toStrictEqual('string')
    });
    test('"(30>3) if ((2/3)>(5+7))" retorna boolean', async () => {
        const result = await parser('(30>3) if ((2/3)<(5+7))')
        expect(result.tipo).toStrictEqual('boolean')
    });
  }
)
```

Uma das estruturas de dados escolhida foi o mapa, esta é uma estrutura de dados que armazena itens com chave e valor, ela mapeia chaves para valores. Essas chaves e valores não tem tipo definido. As chaves são utilizadas para encontrar valores sendo usada quando é necessário que a pesquisa seja rápida. É comum que um mapa seja usado para implementar uma "Árvore" ou "Tabela Hash" . Por isso essa estrutura foi escolhida para implementar o A.F.D e a tabela de símbolos desse projeto.

Outra estrutura escolhida foi a pilha, também conhecida como *stack* é uma estrutura de dados que admite remoção (*pop*) de elementos e inserção (*push*) de novos. De forma mais detalhada, uma pilha é uma estrutura que quando houver uma remoção o elemento removido é o último adicionado a pilha. Isto é, o primeiro objeto a ser inserido na pilha é o último a ser removido. A isso dá-se a expressão de primeiro a entrar, último a sair (do inglês *Last-In-First-Out* (LIFO)) como é exemplificado na Figura 13. Essa estrutura foi usada para o desenvolvimento do autômato de pilha dos analisadores sintático e semântico.

Fig. 13. Ilustração do funcionamento de uma pilha. [13]



IV. Resultados e Discussões

Esta seção apresenta os resultados obtidos com o analisador para cada conjunto de expressões escolhidas na modelagem desse projeto. Tanto as expressões simples, que são aquelas

Tabela X. Gramática para as expressões escolhidas da linguagem CQL.

0	P' -> P
1	P -> A
2	A -> EXP_M A
3	EXP_M -> EXP_M + T
4	EXP_M -> EXP_M - T
5	EXP_M- > T
6	T -> T * F
7	T -> T / F
8	T -> F
9	F -> OPRD
10	OPRD -> (EXP_M)
11	OPRD -> id
12	OPRD -> num
13	A -> EXP_R A
14	EXP_R -> OPRD opr OPRD
15	A -> EXP_L A
16	EXP_L -> TEST opl TEST
17	EXP_L->TEST
18	TEST -> (EXP_L)
19	TEST -> (EXP_R)
20	TEST -> id
21	TEST -> boolean
22	A -> COND A
23	COND -> RESULT if TEST
24	RESULT -> TEST
25	RESULT -> OPRD
26	A ->ε

que usam constantes ou variáveis de contexto como operandos quanto as compostas que são as que utilizam outras expressões em sua construção. Como as possibilidades de combinações são infinitas foram escolhidos alguns exemplos do resultado, mas o analisador funciona também para outras combinações de expressões dos conjuntos aceitos. Para isso, serão apresentadas as entradas (código fonte) e saídas resultantes (tokens e tipo ou erros).

A. *Expressões aritméticas*

- Soma: A expressão de entrada "3+1"tem como operandos números do tipo inteiro e utiliza o operador soma(+), assim o resultado esperado é um número do tipo inteiro. No caso da entrada "cart's total + 10"um dos operandos é do tipo decimal, enquanto o outro é do tipo inteiro resultando em um número do tipo decimal como retorno como pode ser observado na Figura 14.
- Subtração: A expressão de entrada "5-2"tem como operandos números do tipo inteiro e utiliza o operador subtração(-), assim o resultado esperado é um número do tipo inteiro. No caso da entrada "cart's total - 5"um dos operandos é do tipo decimal, enquanto o outro é do tipo inteiro resultando em um número do tipo decimal como retorno como pode ser observado na Figura 14.
- Divisão: A expressão de entrada "110/2"tem como operandos números do tipo inteiro e utiliza o operador divisão(/), assim o resultado esperado é um número do tipo inteiro. No caso da entrada "cart's shippingPrice / 2"o operando "cart's shippingPrice"é do tipo decimal, enquanto o outro é do tipo inteiro resultando em um número do tipo decimal como retorno como pode ser observado na Figura 14 . Essas expressões usam apenas o tipo da entrada para inferir o retorno, mas em casos como "10/3"o resultado correto levando em conta os valores seria do tipo decimal.
- Multiplicação: A expressão de entrada "3*3"tem como operandos números do tipo inteiro e utiliza o operador multiplicação(*), assim o resultado esperado é um número do tipo inteiro. Considerando a entrada "cart's shippingPrice * user's age"o operando "cart's shippingPrice é do tipo decimal, enquanto o outro é do tipo inteiro resultando em um número decimal como retorno como pode ser

Tabela XI. Resultado esperado para cada combinação de operandos dentre as expressões aritméticas.

OPERANDO 1	**OPERANDO 2**	**RESULTADO**
Inteiro	Inteiro	Inteiro
Inteiro	Decimal	Decimal
Inteiro	String	Erro
Inteiro	Boolean	Erro
Decimal	Decimal	Decimal
Decimal	String	Erro
Decimal	Boolean	Erro
String	Boolean	Erro

Tabela XII. Resultado esperado em cada combinação de operandos para as expressões relacionais.

OPERANDO 1	**OPERANDO 2**	**RESULTADO**
Inteiro	Inteiro	Boolean
Inteiro	Decimal	Boolean
Inteiro	String	Erro
Inteiro	Boolean	Erro
Decimal	Decimal	Boolean
Decimal	String	Erro
String	Boolean	Erro

observado na Figura 14 .

- Compostas: Como exemplificado na Figura 14, a expressão de entrada "3+10*9"tem como operando apenas números inteiros, resultando em um número inteiro. Enquanto a expressão "cart's total - 10/2"que possui o operando "cart's total"do tipo decimal resulta no tipo decimal.

No caso de expressões em que os operando são diferentes de números inteiros ou decimais é possível observar um erro semântico. Por exemplo, para a expressão de entrada "30 + teste", em que "Teste" é um identificador do tipo desconhecido o resultado esperado é que o tipo seja "nulo" e retorne um erro. Outro exemplo é para a entrada "3*quatro", na qual "quatro"é um identificador desconhecido resultando assim um erro semântico como mostrado na Figura 15.

Fig. 14. Resultado dos casos de teste para algumas expressões aritméticas.

```
PASS src/__tests__/expression.test.ts
  Testes de expressões aritméticas
    √ "3+1" retorna inteiro (3 ms)
    √ "cart's total + 10" retorna decimal (1 ms)
    √ "5-2" retorna inteiro
    √ "cart's total - 5" retorna decimal
    √ "110/2" retorna inteiro
    √ "cart's shippingPrice / 2" retorna decimal (1 ms)
    √ "10/3" retorna inteiro
    √ "3*3" retorna inteiro
    √ "cart's shippingPrice * user's age" retorna decimal
    √ "3+10*9" retorna inteiro (1 ms)
    √ "cart's total - 10/2" retorna decimal

Test Suites: 1 passed, 1 total
Tests:       11 passed, 11 total
```

Fig. 15. Resultado dos casos de teste para algumas expressões com operador aritmético que retornam erro.

```
PASS src/__tests__/expression.test.ts
  Testes de expressões com erro
    √ "30 + teste " retorna erro (2 ms)
    √ "3 * quatro" retorna erro

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
```

B. Expressões relacionais

- Maior que: Como é apresentado na Figura 16, para a expressão de entrada "3>1", os operandos são do tipo inteiro e o retorno esperado é boolean.
- Menor que: Para a expressão "2<5"os operandos são constantes numéricas, assim como para "user's age < 18"cujo operando "user's age"é uma variável de contexto o retorno esperado é do tipo boolean.
- Composta: Como é apresentado na Figura 16, para a expressão de entrada "(2+3)<5"na qual os operandos são do tipo inteiro e um deles é uma expressão aritmética o retorno esperado é boolean. Assim como para a entrada "(2/3)<(5+7)", na qual ambos operandos são expressões matemáticas o retorno é do tipo boolean

No caso de expressões em que os operando são diferentes de números inteiros ou decimais é possível observar um erro semântico. Por exemplo, para a expressão de entrada "(30 + 2) < teste", em que "Teste" é um identificador do tipo desconhecido o resultado esperado é que o tipo seja "nulo" e retorne um erro. Outro exemplo é para a entrada "3 < quatro", na qual "quatro"é um identificador desconhecido resultando assim um erro semântico como mostrado na Figura 17.

Fig. 16. Resultado dos casos de teste para algumas expressões relacionais.

```
> type-analyzer@1.0.0 test
> jest

PASS src/__tests__/expression.test.ts
  Testes de expressões relacionais
    √ "3>1" retorna boolean (3 ms)
    √ "2<5" retorna boolean (1 ms)
    √ "user's age < 18" retorna boolean (1 ms)
    √ "(2+3)<5" retorna boolean (1 ms)
    √ "(2/3)<(5+7)" retorna boolean (1 ms)

Test Suites: 1 passed, 1 total
Tests:       5 passed, 5 total
```

Fig. 17. Resultado dos casos de teste para algumas expressões com operador relacional que retornam erro.

```
PASS src/__tests__/expression.test.ts
  Testes de expressões relacionais com erro
    √ "(30 + 2) < teste " retorna erro (2 ms)
    √ "3 < quatro" retorna erro (1 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
```

C. Expressões lógicas

- And: Para a entrada "true and false", que utiliza o operador *And* em uma expressão lógica simples o retorno esperado é do tipo boolean como é mostrado na Figura 18.

- Or: Para a entrada "true or false", que utiliza o operador *Or* em uma expressão lógica o retorno esperado é do tipo boolean como é mostrado na Figura 18.
- Compostas: É apresentado na Figura 18 exemplos de expressões lógicas compostas, nas quais os operando são outras expressões com retorno do tipo boolean. Para as expressões "(userś age < 18) and (cartś total > 100)"que utiliza o operador *And* e "(userś age < 18) or (cartś total > 100)"que utiliza o operador *Or* o retorno é do tipo boolean
- Erro: No caso de uma expressão em que um dos operandos não é do tipo boolean, o retorno esperado é um erro. Como exemplificado para a entrada "(userś age < 18) or locationś city".

Fig. 18. Resultado dos casos de teste para algumas expressões lógicas.

```
> type-analyzer@1.0.0 test
> jest

PASS src/__tests__/expression.test.ts
  Testes de expressões lógicas
    √ "true and false" retorna boolean (2 ms)
    √ "true or false" retorna boolean
    √ "(user's age < 18) and (cart's total > 100)" retorna boolean (1 ms)
    √ "(user's age < 18) or (cart's total > 100)" retorna boolean
    √ "(user's age < 18) or location's city" retorna erro

Test Suites: 1 passed, 1 total
Tests:       5 passed, 5 total
```

D. *Expressões condicionais*

São as expressões condicionais sem *'else'*, ou seja, os tipo de dados de retorno só acontecem caso o teste seja verdadeiro. Essas expressões podem assumir retorno dos tipos inteiro, decimal, string e boolean.

- Inteiro: Para a expressão de entrada "30 if true", o teste é uma constante booleana e tipo do retorno é um inteiro como mostrado na Figura 19.
- Decimal: No caso da entrada "cartś total if (2>3)", o teste é uma expressão relacional com o tipo boolean e o retorno é uma variável de contexto do tipo decimal.
- String: Para a entrada "userś name if (true or false)", o teste é uma expressão lógica e o retorno é uma variável de contexto do tipo string.
- Boolean: Para a entrada "30>3 if ((2/3)<(5+7))", o teste é uma expressão relacional composta com retorno booleano e o retorno é uma relacional simples o que resulta no tipo boolean para a expressão completa.

Fig. 19. Resultado dos casos de teste para algumas expressões condicionais.

```
PASS src/__tests__/expression.test.ts
  Testes de expressões condicionais
    √ "30 if true" retorna inteiro (4 ms)
    √ "cart's total if (2>3)" retorna decimal (1 ms)
    √ "user's name if (true or false)" retorna string (1 ms)
    √ "location's city if (true and true)" retorna string
    √ "(30>3) if ((2/3)<(5+7))" retorna boolean (1 ms)
    √ "3 if location's city" retorna erro

Test Suites: 1 passed, 1 total
Tests:       6 passed, 6 total
```

Para os casos de entradas com o teste diferente do tipo booleano, o retorno esperado é um erro de tipos incompatíveis. Como é o caso da expressão "3 if location's city", na qual o teste "location's city"é do tipo string. Uma falha desse analisador é retornar um único tipo independente, isso porque em alguns casos o retorno pode ser nulo, na Figura 20 são mostradas quatro expressões em que o resultado do teste seria falso, logo o retorno esperado para a expressão seria nulo.

Fig. 20. Resultado dos casos de teste para algumas expressões condicionais que deveriam retornar nulo.

```
PASS src/__tests__/expression.test.ts
  Testes de expressões condicionais cujo o retorno pode ser nulo
    √ "30 if false" retorna inteiro (3 ms)
    √ "cart's total if (3>30)" retorna decimal (1 ms)
    √ "location's city if (true and false)" retorna string (1 ms)
    √ "(30>3) if ((2/3)>(5+7))" retorna boolean (2 ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
```

V. Conclusão

Neste artigo foi apresentado o processo de modelagem para o projeto de um analisador estático que tinha como objetivo determinar o tipo do retorno de uma expressão CQL. Além disso, o processo de desenvolvimento foi detalhado, as tecnologias, os conceitos e a estruturas por trás da implementação da aplicação.

O objetivo geral de desenvolver um analisador de retornos para expressões CQL foi atingido. Para expressões aritméticas, relacionais, lógicas e condicionais simples já é possível utilizar esses retornos para a verificação da validade do código fonte. Além de definir um padrão para que novas regras sejam adicionadas de acordo com a necessidade.

Este projeto deixa disponível para trabalhos futuros o refinamento do analisador, adicionando regras semânticas para erros de lógica booleana, retornos nulos, outras expressões e a integração desse retorno com o serviço de cadastro de audiências e componentes da Croct para comparação entre o tipo esperado e o retorno.

Agradecimentos

Dedico este projeto a todos que me apoiaram durante a graduação.

A minha mãe que sempre me apoiou na escolha do curso e nos estudos. A minha irmã Isadora que me apoiou em todo o processo.

Ao meu falecido pai que me incentivou a cursar computação e ficaria muito orgulhoso por eu chegar nessa etapa do curso.

Ao Thiago que escutou todas as minhas reclamações e momentos em que pensei que não iria conseguir terminar.

A minha orientadora professora Deborah que fez aumentar meu interesse na área de compiladores e que me ajudou a persistir durante o desenvolvimento deste projeto.

Ao meu amigo Heinrych que desde o primeiro período foi um parceiro e amigo em todas as disciplinas que cursamos juntos e até mesmo as que não. A minha amiga Natalie que durante a disciplina de compiladores me ensinou bastante sobre programação, o que me permitiu desenvolver também esse projeto. Também aos meus amigos Iran, Giulianni e o Geovana que me ajudaram na rveisão desse artigo.

REFERÊNCIAS

[1] P. Brusilovsky, A. Kobsa, and W. Nejdl, Eds., *The Adaptive Web*. Springer, 2007.
[2] JAMSTACK. Headless Technology. [Online]. Available: https://jamstack.org/glossary/headless-technology/
[3] CROCT. Croct. [Online]. Available: https://croct.com/
[4] Croct. Materiais educativos - Contexto do usuário. [Online]. Available: https://croct.com/pt-br/materiais-educativos/contexto-do-usuario/
[5] A. V. A. R. S. Jeffrey, Ed., *Compiladores*. Ltc, 1995.
[6] I. L. M. Ricarte. (14/02/2003) Conversão para autômato finito determinístico. [Online]. Available: https://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node48.html
[7] M. Johnson and J. Zelenski. (25/07/2012) Anatomy of a Compiler. [Online]. Available: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/020%20CS143%20Course%20Overview.pdf
[8] GEEKS FOR GEEKS. (01/08/2022) Ambiguous Grammar. [Online]. Available: https://www.geeksforgeeks.org/ambiguous-grammar/?ref=leftbar-rightbar
[9] MICROSOFT. (08/08/2022) Precedência e ordem da avaliação. [Online]. Available: https://docs.microsoft.com/pt-br/cpp/c-language/precedence-and-order-of-evaluation?view=msvc-170
[10] GEEKS FOR GEEKS. (25/02/2022) SLR Parser (with Examples). [Online]. Available: https://www.geeksforgeeks.org/slr-parser-with-examples/
[11] TypeScript. What is TypeScript. [Online]. Available: https://croct.com/
[12] Facebook Open Source. Iniciando. [Online]. Available: https://jestjs.io/pt-BR/docs/getting-started
[13] R. Farias. Estrutura de Dados e Algoritmos. [Online]. Available: https://www.cos.ufrj.br/~rfarias/cos121/pilha1.png

Isabella B. Silva é estudante de Engenharia de Computação pela Universidade Federal de Goiás. Atualmente atua na área de requisitos e testes de software na empresa Croct. Possui grande interesse em Análise de dados e experiência do usuário. Possui ainda experiência em desenvolvimento *web Back-end*.

Deborah S. A. Fernandes trabalha no Instituto de Informática da Universidade Federal de Goiás – Campus Samambaia, Goiânia, Goiás. Além disso, é Cientista da Computação (Pontifícia Universidade Católica de Goiás), mestre em Engenharia Elétrica com ênfase em Visão Computacional (Escola de Engenharia – Universidade de Brasília/DF) e doutora em Engenharia de Sistemas Eletrônicos e Automação (Escola de Engenharia -Universidade de Brasília/DF) com ênfase em análise de dados de redes sociais e apoio à tomada de decisão.