UNIVERSIDADE DE CAXIAS DO SUL
ÁREA DO CONHECIMENTO DE CIÊNCIAS EXATAS E ENGENHARIAS

GABRIEL MÜLLER

AMBIENTE DE APRENDIZADO PARA VISUALIZAÇÃO DE CÓDIGO INTERMEDIÁRIO E SUAS OTIMIZAÇÕES

CAXIAS DO SUL

2024

GABRIEL MÜLLER

AMBIENTE DE APRENDIZADO PARA VISUALIZAÇÃO DE CÓDIGO INTERMEDIÁRIO E SUAS OTIMIZAÇÕES

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Orientador: Prof. Dr. Ricardo Vargas Dorneles

CAXIAS DO SUL

2024

GABRIEL MÜLLER

AMBIENTE DE APRENDIZADO PARA VISUALIZAÇÃO DE CÓDIGO INTERMEDIÁRIO E SUAS OTIMIZAÇÕES

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Aprovado em 25/11/2024

BANCA EXAMINADORA

Prof. Dr. Ricardo Vargas Dorneles
Universidade de Caxias do Sul - UCS

Prof. Me. Alexandre Erasmo Krohn Nascimento
Universidade de Caxias do Sul - UCS

Prof. Dr. Daniel Luis Notari
Universidade de Caxias do Sul - UCS

RESUMO

A abordagem exclusivamente teórica de algumas etapas realizadas pelo compilador acaba por dificultar a visualização prática do conteúdo, além de tornar mais difícil o processo de aprendizado dos alunos da disciplina de compiladores na Universidade de Caxias do Sul (UCS). Identificou-se a necessidade de atividades práticas durante o processo de aprendizagem de otimizações de código independente de máquina e, a partir dessa observação, definiu-se a proposta de desenvolvimento de um ambiente de aprendizado para a visualização de código intermediário e suas otimizações. Com a elaboração do ambiente citado, idealiza-se que fique mais fácil e prático o processo de aprendizagem dos alunos da disciplina de compiladores. Além disso, propôs-se o desenvolvimento de um *hub* de aplicações para agrupar diversos ambientes de aprendizagem que possam ser úteis aos alunos do curso de Ciência da Computação (CC) da UCS.

Palavras-chave: Compiladores. Otimização. Aprendizado.

ABSTRACT

The exclusively theoretical approach to some of the stages carried out by the compiler ends up making it difficult to visualize the content in practice, as well as making the learning process more difficult for students of the compilers course at the University of Caxias do Sul (UCS). The need for practical activities during the process of learning machine-independent code optimizations was identified and, based on this observation, the proposal to develop a learning environment for visualizing intermediate code and its optimizations was defined. The development of this environment is intended to make the learning process easier and more practical for compiler students. In addition, it is proposed to develop an application hub to group together various learning environments that could be useful to students on the Computer Science (CS) course at UCS.

Keywords: Compilers. Optimization. Learning.

LISTA DE FIGURAS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

C3E Código de Três Endereços

CC Ciência da Computação

ComVis Compiler Visualization System

GFC Grafo de Fluxo de Controle

GAD Grafo Acíclico Dirigido

JSON *JavaScript Object Notation* (Notação de Objeto JavaScript)

LISA Language Implementation System based on Attribute grammars

PAVT Parsing Algorithms Visualization Tool

RI Representação Intermediária

SPAs *Single-Page Applications* (Aplicações de página única)

UCS Universidade de Caxias do Sul

SUMÁRIO

1 INTRODUÇÃO

A melhoria da educação no curso de CC da UCS sempre foi e continua sendo alvo de todos os docentes envolvidos no curso. Mais especificamente na disciplina de compiladores, sabe-se que existe amplo espaço para aperfeiçoamento e melhorias no que se refere a propiciar ao discente um ambiente mais acolhedor e favorável ao aprendizado, visto que essa disciplina é considerada por muitos como uma das mais difíceis de todo o curso de CC.

Muitos dos primeiros cursos de compiladores colocavam ênfase na teoria da tradução dirigida por sintaxe. Porém, logo foi percebido que isso não era boa prática, visto que focar apenas na teoria não necessariamente ensina os alunos a construir compiladores úteis. Alunos de CC, ao cursar a disciplina de compiladores, estudam os conceitos básicos de um compilador (análise léxica, análise sintática, análise semântica, geração de código intermediário, ambientes de execução, gerenciamento de recursos e geração de código de máquina), geralmente tendo a tarefa de implementar um pequeno compilador até o final da disciplina (AHO, 2008).

Entretanto, exemplificando o cenário atual da disciplina de compiladores na UCS, algumas etapas realizadas pelo compilador são estudadas quase que exclusivamente de forma teórica, principalmente o que se refere às otimizações de código realizadas sobre a Representação Intermediária (RI). Dessa forma, as tarefas realizadas pelos alunos não contemplam a implementação de algoritmos de otimização de RI, dificultando a visualização prática do conteúdo e o processo de aprendizagem dos alunos.

No que se refere ao ensino de compiladores, alguns *frameworks* foram implementados visando contribuir com as dificuldades mencionadas. Alguns compiladores educacionais, como por exemplo o Compiler Visualization System (ComVis) (STAMENKOVIć; JOVANOVIć, 2024) e o Parsing Algorithms Visualization Tool (PAVT) (SANGAL; KATARIA; TYAGI, 2018) possuem recursos que objetivam ensinar o aluno de compiladores, além de servirem como uma janela para o processo de compilação. Porém, esses *frameworks* não focam na implementação de compiladores em um âmbito acadêmico ou não possuem recursos focados exclusivamente na etapa de otimização de RI.

1.1 OBJETIVOS

A partir da identificação do problema de ensino na disciplina de compiladores, propõe-se o estudo e desenvolvimento de um ambiente de aprendizado que permita a visualização do usuário sobre a geração de código intermediário e suas otimizações.

Com o presente trabalho, idealiza-se a produção de um ambiente capaz de permitir que o usuário:

- Insira um código em linguagem C.
- Consiga visualizar as etapas de geração da RI.
- Possa selecionar se serão e quais serão as otimizações aplicadas sobre a RI.
- Seja capaz de definir a ordem de aplicação das otimizações sobre a RI.
- Consiga exportar o resultado gerado.
- Consiga importar resultados de outros usuários e manipular as configurações mencionadas acima para visualizar novos resultados.

Além do ambiente citado acima, idealizou-se a existência de um *hub* para a centralização de diversas aplicações destinadas a facilitar o processo de aprendizagem dos alunos. Este *hub* atua no presente trabalho como um ambiente adicional, proposto para agrupar o ambiente citado acima com outros ambientes que venham a ser desenvolvidos no futuro e facilitar o processo de localização de ambientes de aprendizagem.

1.2 ESTRUTURA DO TRABALHO

No Capítulo 2, é realizada uma análise inicial sobre a fundamentação teórica utilizada como base para o desenvolvimento do presente trabalho.

No Capítulo 3, discorre-se sobre trabalhos relacionados, no que se refere à existência de ambientes de aprendizagem focados no ensino de compiladores, além de comparativos com a proposta de desenvolvimento, justificando a existência de um novo ambiente.

No Capítulo 4, aborda-se os principais tópicos sobre o desenvolvimento do ambiente e do *hub*, durante a fase de implementação.

No Capítulo 5, elabora-se a descrição do processo de desenvolvimento em relação ao que foi descrito inicialmente no Capítulo 4, citando principais características do desenvolvimento e principais bibliotecas utilizadas.

No Capítulo 6, são apresentadas as considerações sobre o que foi desenvolvido, além dos objetivos atingidos e não atingidos e uma breve descrição dos trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Um compilador é um programa capaz de ler um programa disponibilizado em uma linguagem e traduzí-lo para um programa equivalente em outra linguagem (desta vez, de baixo nível), apresentando erros encontrados durante o processo de tradução para o usuário. Esse processo é dividido em duas etapas: Análise e Síntese (AHO *et al.*, 2007).

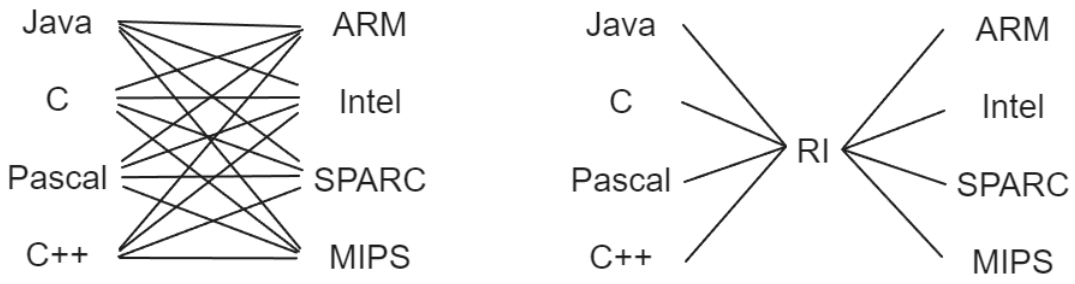
A etapa de análise é responsável por dividir o programa fonte em partes e impor uma estrutura gramatical sobre ele. Esta estrutura é utilizada para criar uma RI do programa fonte. Além disso, esta etapa deve coletar informações sobre o código fonte e armazenar em uma estrutura de dados chamada tabela de símbolos. Assim como mencionado anteriormente, o compilador deve alertar ao usuário em situações de erros sintáticos e/ou semânticos que forem identificados ao decorrer da etapa de análise. Ao final da etapa de análise, são encaminhados para a etapa de síntese a tabela de símbolos e a representação intermediária (AHO *et al.*, 2007).

A etapa de síntese, por sua vez, é responsável por produzir o programa destino, a partir das informações recebidas da etapa anterior. É comum que esta etapa inclua uma sub-etapa para realizar a otimização da RI, mas alguns autores também costumam separar esta etapa em duas: otimização e geração de código. Por mais que aconteça esta separação, a etapa de geração de código ainda pode realizar algumas otimizações em baixo nível de acordo com a arquitetura destinada. Porém, estas otimizações realizadas pela etapa de geração de código não serão abordadas no presente trabalho, restringindo os estudos acerca das otimizações apenas para a etapa de otimização (ROCHA, 2017).

2.1 Representação Intermediária

A utilização de uma RI se faz necessária no momento em que é desejado generalizar um compilador. Com ela, permite-se que um *front end* seja reutilizado por diferentes compiladores, na mesma medida em que novas linguagens possam também se utilizar de *back ends* já implementados. Além disso, é facilitado o processo de otimização de código a partir da utilização de uma RI, visto que a maioria das otimizações são realizadas sobre ela (AHO *et al.*, 2007).

Figura 1 – Comparação entre um compilador que não utiliza RI e um que utiliza RI



Fonte: O Autor (2024).

Conforme o cenário ilustrado na Figura 1, um compilador que não se utiliza de uma RI teria de realizar 16 processos isolados de tradução de uma linguagem A para uma linguagem B de baixo nível, enquanto um compilador que faz utilização de uma RI precisaria realizar apenas 8 processos de tradução (4 para a RI e mais 4 a partir da RI). Na medida em que a quantidade de linguagens aumenta, essa diferença se faz ainda maior. De forma generalista, um compilador que não utiliza RI precisa realizar *i*j* processos de tradução, na medida em que um compilador que utiliza RI realizaria apenas *i+j* processos de tradução para obter os mesmos resultados. No exemplo citado, *i* refere-se à quantidade de linguagens a serem traduzidas, e *j* refere-se à quantiade de arquiteturas/linguagens de baixo nível a que se deseja realizar a tradução.

As RIs podem ser classificadas em três modalidades distintas: RIs hierárquicas, consideradas de mais alto nível em relação às outras modalidades, permitem a utilização de estruturas encadeadas (como fluxos de controle e expressões aritméticas, por exemplo); RIs não hierárquicas (ou *flat IR*), consideradas de mais baixo nível, são compostas de instruções que são executadas sequencialmente e os fluxos de controle são tratados por instruções de desvio como *branch* ou *jump*; e código para máquinas de pilha, enquandrando-se como um meio termo entre as hierárquicas e as não hierárquicas, onde a forma da RI é *flat* (aproximando-se de uma RI não hierárquica), mas a sequência de instruções pode ser facilmente representada internamente em uma estrutura de dados de árvore (aproximando-se de uma RI hierárquica) (CHOW, 2013).

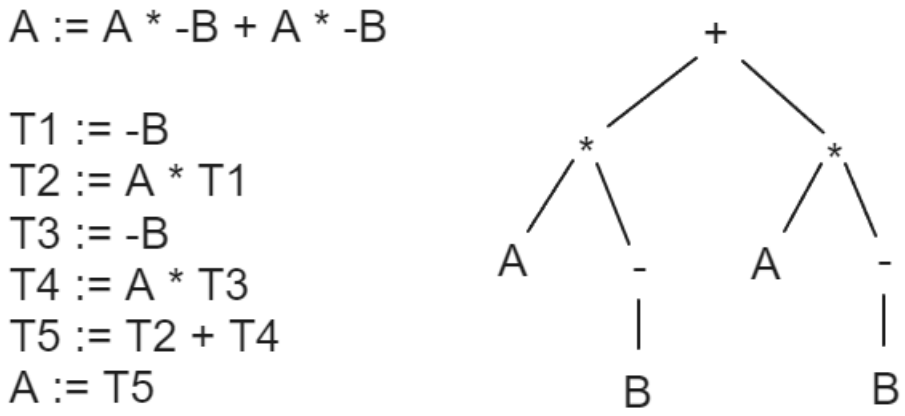
RIs não hierárquicas, por serem mais próximas das linguagens de montagem e também por existir a possibilidade de serem representadas através de Código de Três Endereços (C3E) (CHOW, 2013), são o foco do presente trabalho.

2.2 CÓDIGO DE TRÊS ENDEREÇOS

O C3E é uma forma de representação de código intermediário que consiste em instruções de no máximo 3 operandos, no formato $x = y\ op\ z$, onde x, y e z podem ser nomes, constantes, ou temporários gerados pelo compilador, enquanto *op* é o operador da instrução. As instruções do C3E são executadas em sequência númerica, exceto caso explicitamente forçadas a agirem de outra maneira, através de saltos condicionais ou incondicionais (AHO *et al.*, 2007).

Como o C3E permite a utilização de no máximo 3 operandos, pode-se afirmar que ele é uma representação linear de uma estrutura de dados de árvore. Nessa representação, os nomes são correspondentes aos nodos do interior do grafo, conforme ilustrado na Figura 2.

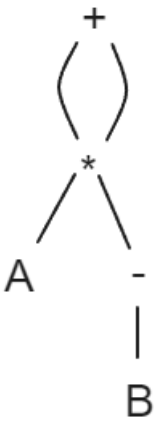
Figura 2 – Equivalência entre C3E e uma árvore



Fonte: O Autor (2024).

Tratando-se da Figura 2, ainda podemos afirmar que ela poderia ser simplificada, visto que tudo que está abaixo do nível do "+", acaba repetindo-se exatamente nos filhos da esquerda e direita. Uma versão simplificada e equivalente à árvore apresentada na Figura 2 pode ser observada na Figura 3. Esse processo de simplificação, conforme mencionado até o momento, é chamado de otimização.

Figura 3 – Árvore otimizada equivalente à apresentada na Figura 2



Fonte: O Autor (2024).

O C3E é construído a partir de endereços e instruções. Endereços podem ser nomes, constantes ou até temporários gerados pelo compilador (AHO *et al.*, 2007). Instruções, por sua vez, podem assumir diferentes formatos, conforme definido por Aho *et al.* (2007) e listado a seguir.

1. Atribuições da forma $x = y\ op\ z$. Define-se que *op* representa um operador aritmético binário ou lógico, enquanto x, y e z são endereços.

2. Atribuições da forma $x = op\ y$. Neste caso, diferenciando-se da forma anterior, *op* é um operador unário.

3. Atribuições da forma $x = y$, onde se atribui a x o valor de y. São conhecidas como instruções de cópia.

4. Desvios incondicionais da forma *goto L*.

5. Desvios condicionais da forma *if x goto L* e *ifFalse x goto L*. Define-se *L* como um rótulo de uma instrução.

6. Desvios condicionais da forma *if x relop y goto L*. Neste item, diferentemente do anterior, ocorre a aplicação de um operador relacional entre x e y.

7. Chamadas de procedimento e retornos nas formas *param x* (para parâmetros), *call p,n* (para chamadas de procedimento), $y = call\ p,n$ (para chamadas de função) e *return y* (para retorno, considerando ainda que y é opcional).

8. Instruções indexadas de cópia da forma $x = y[i]$ e $x[i] = y$.

9. Atribuições de endereço e apontador da forma $x = \&y$, $x = *y$ e $*x = y$.

2.3 OTIMIZAÇÕES

A etapa de otimização de código independente de máquina visa aperfeiçoar o código intermediário de forma em que um código de baixo nível (destino) melhor seja produzido. Nesse caso, a palavra 'melhor' pode referir-se a velocidade, código menor ou até código que consome menos energia (AHO *et al.*, 2007).

Não é desejado que todas as otimizações sejam aplicadas em todas as situações e, ao mesmo tempo, um mesmo conjunto de otimizações pode ser aplicado em diferentes ordens. Sendo assim, diversos compiladores modernos acabam por possuir configurações de otimização, onde são definidas quais e em que ordem as otimizações serão aplicadas sobre a RI.

No presente trabalho, são abordadas as principais otimizações de RI estudadas na disciplina de compiladores na UCS: Movimentações Redundantes, Códigos Inalcançáveis, Otimizações no fluxo de controle, Simplificações Algébricas, Redução de Esforço, Propagação de cópia, Eliminação de código morto e Otimizações sobre laços. Vale ressaltar que, no último item citado, mais de uma otimização é apresentada.

2.3.1 Movimentações Redundantes

Esta otimização visa eliminar instruções onde se faz presente a ocorrência de duas instruções seguidas de movimentação, onde a primeira movimenta de *X* para *Y* e a segunda movimenta de *Y* para *X*. Conforme demonstrado no Algoritmo 1, a segunda instrução poderia ser simplesmente descartada, visto que não realiza nenhuma alteração efetivamente. Vale ressaltar que, em casos onde exista algum *label* entre as duas instruções, não é garantido que a segunda movimentação seja redundante e, portanto, nem sempre pode ser removida, necessitando que análises sejam realizadas antes de realizar a remoção da instrução possivelmente redundante (LOUDEN, 2004).

Algoritmo 1 – Sequência de instruções com a presença de uma Movimentação Redundante

```
1 mov R0, A
2 mov A, R0
```

Fonte: O Autor (2024).

2.3.2 Códigos Inalcançáveis

Acontece quando há a presença de instruções antecedidas de um salto incondicional. É possível observar no Algoritmo 2 que as instruções que vêm após *"goto L1"* podem ser descartadas, visto que não existe um cenário onde as mesmas serão executadas.

Algoritmo 2 – Sequência de instruções com a presença de Código Inalcançável

```
1 mov R0, A
2 goto L1
3 mov A, R0
4 mov B, R0
```

Fonte: O Autor (2024).

2.3.3 Otimizações no fluxo de controle

Pode ocorrer situações onde um salto condicional não é otimizado, conforme apresentado no Algoritmo 3. Quando isso se faz presente, é possível reestruturar a sequência de instruções visando reduzir a quantidade de instruções e *labels* utilizados, conforme visível no Algoritmo 4, onde a otimização do fluxo de controle foi aplicada, removendo uma instrução por completo simplesmente por ter alterado a condição do salto da primeira linha.

Algoritmo 3 – Sequência de instruções com falta de otimização no fluxo de controle (salto condicional)

```
1     if A=1 goto L1
2     goto L2
3 L1: mov R0, A
```

```
4       [...]
5   L2: mov A, R0
```

Fonte: O Autor (2024).

Algoritmo 4 – Otimização de fluxo de controle aplicada sobre o Algoritmo 3

```
1       if A<>1 goto L2
2   L1: mov R0, A
3       [...]
4   L2: mov A, R0
```

Fonte: O Autor (2024).

Além disso, também pode acontecer a situação onde um desvio incondicional para um *label* que possui como primeira instrução um outro desvio, conforme apresentado no Algoritmo 5. Nestes casos, pode-se substituir o primeiro desvio pelo segundo, resultando no que se faz presente no Algoritmo 6 (MOGENSEN, 2024).

Algoritmo 5 – Sequência de instruções com falta de otimização no fluxo de controle (desvio para desvio)

```
1       goto L1
2       [...]
3   L1: goto L2
```

Fonte: O Autor (2024).

Algoritmo 6 – Otimização de fluxo de controle aplicada sobre o Algoritmo 5

```
1       goto L2
2       [...]
3   L1: goto L2
```

Fonte: O Autor (2024).

A partir da substituição realizada pelo Algoritmo 6, deve-se analisar o código para verificar se existem desvios para *L1* e, em caso negativo, tanto o *label* quanto sua instrução podem ser diretamente eliminados (MOGENSEN, 2024). O resultado deste processo aplicado sobre o Algoritmo 6 pode ser visualizado no Algoritmo 7.

Algoritmo 7 – Otimização de fluxo de controle aplicada sobre o Algoritmo 6

```
1       goto L2
2       [...]
```

Fonte: O Autor (2024).

2.3.4 Simplificações Algébricas

Algumas simplificações algébricas podem ser realizadas em instruções de forma a substituir expressões que não precisam ser calculadas ou que possam ser calculadas de forma mais eficaz. Consistem em operações que possam ser simplificadas e são realizadas entre constantes ou entre variáveis e constantes (AHO *et al.*, 2007). Alguns exemplos são:

- X somado ou subtraído de 0 = X
- 0 somado com X = X
- X multiplicado ou dividido por 1 = X
- 1 multiplicado por X = X

Além dos exemplos citados, é importante mencionar a existência das simplificações das expressões entre constantes. Estas podem existir previamente no código (por diversos motivos, como por exemplo a legibilidade do código fonte) ou também podem aparecer no código após a aplicação de outros processos de otimização.

2.3.5 Redução de Esforço

A otimização de Redução de Esforço consiste em substituir operações complexas por operações mais simples que tenham os mesmos resultados (LOUDEN, 2004). Um exemplo clássico dessa otimização é a operação de elevar um número ao quadrado, que pode ser realizada através da chamada de uma rotina de potenciação (considerada, neste exemplo, como a pior solução), ou através de um produto no formato $X * X$ (considerada aqui como a melhor solução para este cenário).

2.3.6 Propagação de cópia

A ideia por trás da otimização de propagação de cópia é eliminar instruções de atribuição que se apresentam no formato $X = Y$, onde Y possa ser substituído por X em expressões futuras. Isso pode acontecer em casos onde a variável Y não é alterada após ser atribuída em X e, ao mesmo tempo, novos valores não são atribuídos em X antes do valor presente em Y ser utilizado (AHO *et al.*, 2007; SU; YAN, 2011).

Um exemplo do cenário descrito pode ser visualizado no Algoritmo 8. Na exemplificação proposta, a otimização de propagação de cópia iria gerar uma única instrução $t1 = y$, eliminando completamente a necessidade da variável x (considerando apenas o trecho de código proposto).

Algoritmo 8 – Exemplo de código onde poderia ser realizada a otimização de propagação de cópia

```
1    x = y
2    t1 = x
```

Fonte: O Autor (2024).

2.3.7 Eliminação de código morto

A otimização que visa eliminar código morto consiste em identificar e remover todo o código que não afeta o resultado final da computação, incluindo expressões não alcançáveis e atribuições a variáveis não utilizadas posteriormente no código (AHO *et al.*, 2007; SU; YAN, 2011).

2.3.8 Otimizações sobre Laços

Visto que programas passam a maior parte de seus respectivos tempos de execução dentro de laços, quaisquer reduções na quantidade de instruções dentro desses laços podem apresentar grandes melhorias na redução do tempo total de execução. Diante disso, diversas otimizações sobre laços são realizadas, sempre visando aprimorar a RI resultante no que se refere ao tempo de execução.

2.3.8.1 Identificação e remoção de variáveis de indução

Uma variável *X* é considerada uma variável de indução básica nos casos em que, para cada atribuição nova de *X*, a alteração de valor seja igual a um acréscimo de uma constante *C* (AHO *et al.*, 2007). Um exemplo de código com a presença de uma variável de indução pode ser observado no Algoritmo 9, onde as variáveis *i* e *t0* são consideradas como variáveis de indução, visto que *i* sempre é acrescida de 1 e que *t0* sempre é acrescida de 2 em cada repetição do laço, indicando que ambas podem ser otimizadas.

Algoritmo 9 – Exemplo de código com presença de variável de indução

```
1    [...]
2    L1: i = i + 1
3        t0 = 2 * i
4        t1 = x[t0]
5        if t1 < y goto L1
6    [...]
```

Fonte: O Autor (2024).

As variáveis de indução podem ser otimizadas ao utilizar-se da otimização apresentada na Seção 2.3.5. Além disso, em diversas situações, é possível que sejam eliminadas todas (exceto uma) de um grupo de variáveis de indução de um laço. Quando uma variável de indução é derivada de outra variável de indução, essa passa a ser considerada como uma variável de indução derivada (AHO *et al.*, 2007).

2.3.8.2 Computações laço-invariantes

Diz-se como uma otimização de computação laço-invariante aquela que permite a movimentação de uma expressão de dentro de um laço para fora dele. Para que aconteça essa otimização, a expressão precisa ter o mesmo valor resultante em cada iteração do laço em que se encontra. É importante ressaltar que essa otimização pode precisar de repetições, visto que ao identificar uma variável como sendo laço-invariante, outras expressões que possam utilizar essa determinada variável possuem a chance de se tornar laço-invariantes também (AHO *et al.*, 2007).

2.3.8.3 Desdobramento de Laços

Com a finalidade de evitar frequentes atualizações e testes do contador em um laço, no caso de laços com corpo pequeno pode ser realizada a otimização de desdobramento de laços. Esta otimização consiste em copiar (algumas vezes) o corpo do laço e colocar as cópias de forma sequencial dentro do próprio laço, tornando assim o código mais eficiente (AHO *et al.*, 2007; THAIN, 2016).

2.4 ANÁLISE DE FLUXO DE DADOS

Todas as otimizações apresentadas na Seção 2.3 dependem diretamente da análise de fluxo de dados, que pode ser vista como um conjunto de técnicas que, ao analisar o fluxo de dados ao longo das possíveis execuções do programa, possuem por objetivo derivar informações, conforme Aho *et al.* (2007):

> "A 'análise de fluxo de dados' refere-se a um conjunto de técnicas que derivam informações sobre o fluxo de dados ao longo dos caminhos de execução do programa."(AHO *et al.*, 2007)

É importante ressaltar que, no presente trabalho, os valores de fluxo de dados existentes antes e depois de cada comando c são denotados por *IN[c]* e *OUT[c]*, respectivamente. Considera-se, nesse contexto, que o problema da análise de dados consiste em encontrar uma solução para as restrições em *IN[c]* e *OUT[c]* para todos os comandos c. As restrições mencionadas podem ser rotuladas como pertencentes a dois grupos distintos: as que são baseadas na semântica do comando, conhecidas como funções de transferência; e as que são baseadas no fluxo de controle (AHO *et al.*, 2007).

2.4.1 Funções de transferência

Conforme mencionado, as funções de transferência são identificadas como restrições baseadas na semântica do comando. Um clássico exemplo de sua funcionalidade consiste em assumir que a análise de fluxo de dados tenha interesse em determinar o valor constante das

variáveis nos diferentes pontos do código. Se uma variável *a* possuir o valor *x* antes do comando *b = a* ser executado, significa que tanto *a* quanto *b* passarão a ter o valor *x* após a execução do comando. A função de transferência é exatamente esse relacionamento entre os valores de fluxo de dados tanto antes quanto depois do comando (AHO *et al.*, 2007).

As funções de transferência podem assumir dois tipos diferentes: de propagação para frente ou para trás. No primeiro tipo, onde o fluxo é para frente, a função de transferência de um comando *c* (denotada como f_c), é $OUT[c] = f_c(IN[c])$ (significando que o valor do fluxo de dados antes do comando é utilizado para produzir um novo valor de fluxo de dados, desta vez após o comando). Já no segundo tipo, onde o fluxo é para trás, a função de transferência é representada por $IN[c] = f_c(OUT[c])$ (significando que o valor do fluxo de dados depois do comando é utilizado para produzir um novo valor de fluxo de dados, desta vez antes do comando) (AHO *et al.*, 2007).

2.4.2 Restrições do fluxo de controle

Dentro de um bloco básico, define-se que o valor de fluxo de controle saindo de um comando é o mesmo que o valor de fluxo de controle entrando no comando seguinte. Porém, isso torna-se mais complexo na medida em que se analisa a relação entre diferentes blocos, considerando que o valor do fluxo de controle do último comando de um bloco não necessariamente é equivalente ao primeiro valor de fluxo de controle de outro bloco com que se relaciona diretamente. Para obter o conjunto de definições que alcança o primeiro comando (ou comando líder) de um bloco básico, é necessário realizar a união das definições presentes no último comando de cada um dos blocos predecessores (AHO *et al.*, 2007).

2.5 IDENTIFICAÇÃO DE LAÇOS

Os laços, dentro de um Grafo de Fluxo de Controle (GFC), são considerados como sendo de extrema importância, visto que os programas gastam a maior parte de seu tempo executando-os. Além disso, afirma-se que os laços afetam o tempo de execução das análises do programa significativamente e todas otimizações que melhoram o desempenho dos laços existentes podem ter impactos significativos. Pelos motivos citados, considera-se que a identificação e tratamento de laços é essencial (AHO *et al.*, 2007).

Dentro de um GFC, laços são considerados como sendo um conjunto de nós (neste momento, representa-se este conjunto por *N*) que possui um cabeçalho (identificado por *h*) e que atende aos seguintes critérios:

- Deve existir um caminho para *h* a partir de qualquer nó existente em *N*.
- Existe um caminho para qualquer nó em *N* a partir de *h*.

- Não deve existir uma aresta de nós fora do conjunto *N* para dentro de um nó do conjunto *N*, com exceção do nó *h*.

Para realizar a identificação de laços, assume-se no contexto do presente trabalho que será utilizada a técnica de análise de dominadores. Esta técnica e seus principais detalhes são detalhados a seguir.

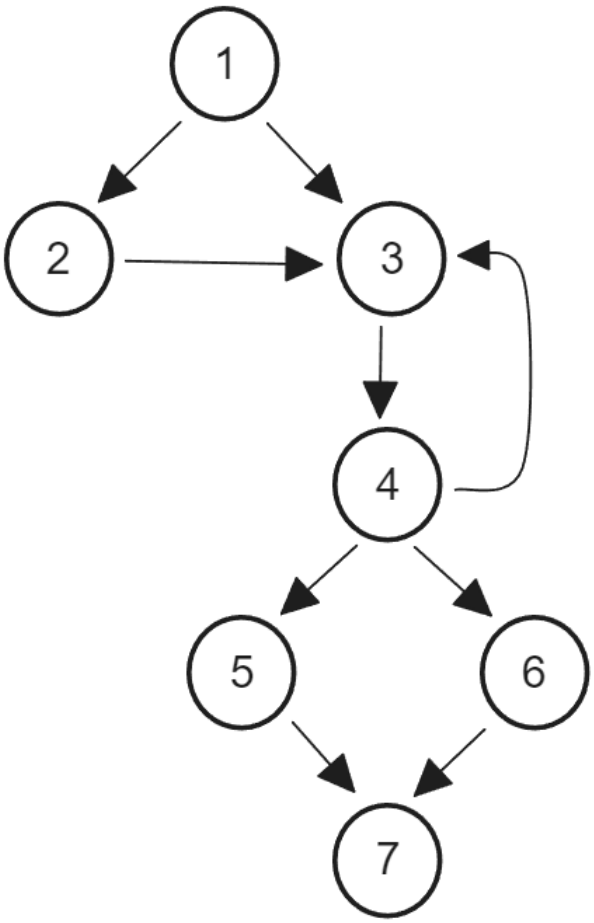
2.5.1 Dominadores

Nomeando o nó inicial de um GFC como sendo n_0, considera-se que um determinado nó *d* domina um outro nó *n* se todos os possíveis caminhos de n_0 até *n* passarem por *d*. Também assume-se que cada nó *n* domina a si mesmo.

Como exemplificação desta definição, é possível visualizar na Figura 4 que:

- O nó 1 domina os nós 1, 2, 3, 4, 5, 6 e 7.
- O nó 2 domina apenas a si mesmo. Mesmo que exista uma aresta entre os nós 2 e 3, este caminho não é obrigatório no GFC, visto que partindo do nó 1 é possível ir diretamente ao nó 3, sem passar pelo nó 2.
- O nó 3 domina os nós 3, 4, 5, 6 e 7.
- O nó 4 domina os nós 4, 5, 6 e 7.
- O nó 5 domina apenas a si mesmo. Observa-se que este nó não domina o nó 7, mesmo possuindo uma aresta direta para o mesmo, porque existe um outro caminho a partir de n_0 até o nó 7 que não passe pelo nó 5.
- O nó 6 domina apenas a si mesmo. Observa-se que este nó não domina o nó 7, mesmo possuindo uma aresta direta para o mesmo, porque existe um outro caminho a partir de n_0 até o nó 7 que não passe pelo nó 6.
- O nó 7 domina apenas a si mesmo.

Figura 4 – Exemplo de um GFC



Fonte: O Autor (2024).

2.5.2 Dominadores Imediatos

Define-se um dominador imediato de um determinado nó *n* como sendo um outro nó, que atende aos seguintes critérios: domina *n*, não é *n* e não domina qualquer outro dominador de *n* existente.

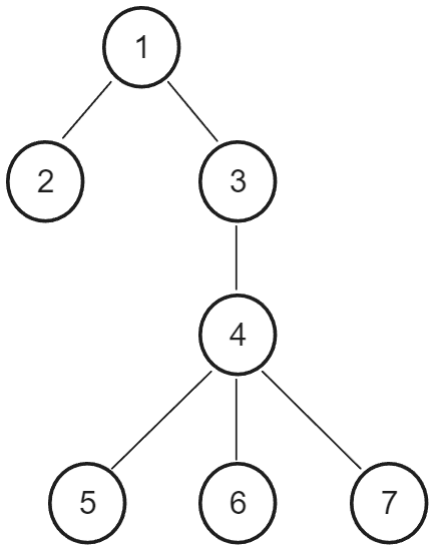
Como forma de representar a relação de dominadores imediatos, utiliza-se de uma estrutura conhecida como árvore de dominadores. Seguindo o exemplo fornecido pela Figura 4, visualiza-se na Figura 5 a árvore de dominadores correspondente. Nela, podemos observar que:

- O nó 1 é dominador imediato dos nós 2 e 3.
- O nó 2 não é dominador imediato de nenhum outro nó.
- O nó 3 é dominador imediato do nó 4.
- O nó 4 é dominador imediato dos nós 5, 6 e 7. Retomando as características citadas anteriormente, observa-se que os dominadores do nó 7 são os nós 1, 3, 4 e 7. O único nó

que não é o próprio nó 7 e que não domina qualquer outro dominador da lista citada é o nó 4 e, portanto, é o dominador imediato do nó 7.

- O nó 5 não é dominador imediato de nenhum outro nó.
- O nó 6 não é dominador imediato de nenhum outro nó.
- O nó 7 não é dominador imediato de nenhum outro nó.

Figura 5 – Árvore de dominadores referente ao exemplo da Figura 4



Fonte: O Autor (2024).

2.5.3 Laços naturais

Laços naturais são caracterizados pela existência de duas propriedades dentro de um subconjunto de um determinado GFC:

1. Existe apenas um nó de entrada, chamado de cabeçalho. O cabeçalho deve dominar todos os nós do laço.
2. Existe uma aresta de retorno, ou seja, uma aresta de um nó do subconjunto que retorne para o cabeçalho. O nó que apresenta a aresta de retorno, no contexto do presente trabalho, será nomeado n_r.

O conjunto de nós que representa um laço natural é definido como sendo o cabeçalho mais o conjunto de nós que podem alcançar n_r sem passar pelo cabeçalho. Na medida em que laços naturais são identificados, algumas análises feitas por compiladores podem realizar suposições sobre algumas condições nas iterações do laço, oportunizando que as propriedades presentes nos laços naturais sejam utilizadas para permitir algumas otimizações.

3 TRABALHOS RELACIONADOS

Diversos alunos da UCS que cursam (ou cursaram) CC consideram a disciplina de compiladores como uma das ou até a mais difícil de todo o curso. Em uma pesquisa realizada com os alunos das duas últimas turmas de compiladores na UCS, 91,7% dos entrevistados responderam sentir falta de um ambiente que permitisse visualizar o que estava sendo estudado na discplina de forma mais eficaz. Nessa mesma pesquisa, 50% dos entrevistados alegaram que consideraram o conteúdo de Otimização de Código como o mais difícil da disciplina.

Considerando-se que algumas etapas realizadas pelo compilador não são estudadas de forma prática ao longo da disciplina, mas ao mesmo tempo a disciplina é popularmente conhecida como uma das mais difíceis de toda a grade curricular atual, assume-se que existe a necessidade de melhorias no que se refere ao ensino de compiladores.

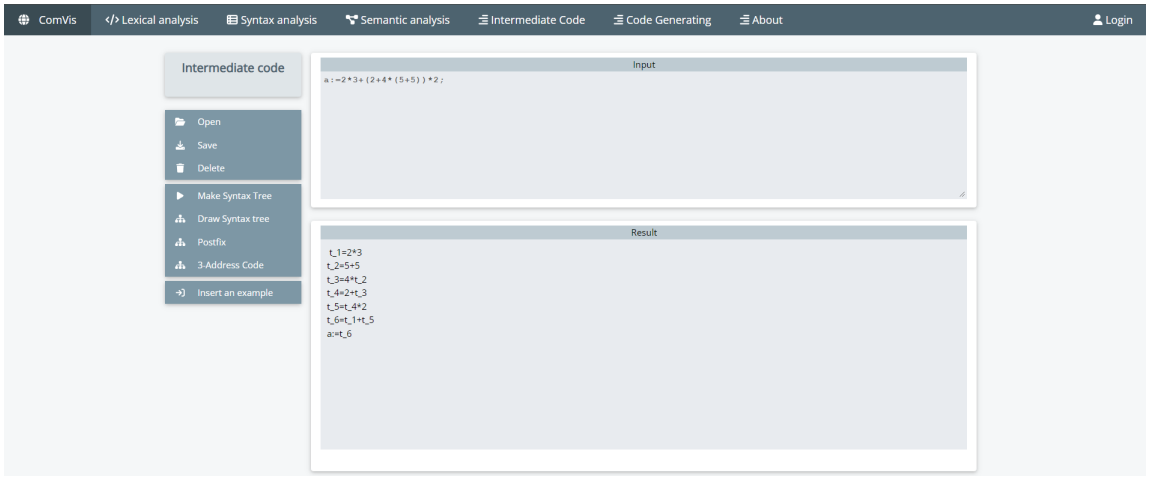
Diversos *frameworks* e ambientes já foram implementados visando solucionar os problemas citados, porém não atendem completamente às necessidades dos alunos de compiladores, ou não contemplam funcionalidades consideradas essenciais para a aprendizagem de otimizações em compiladores. Este capítulo irá discorrer acerca de alguns destes ambientes já desenvolvidos e publicados.

3.1 *COMVIS!: COMPILER VISUALIZATION SYSTEM*

O sistema ComVis foi publicado no ano de 2023 e abrange diversos componentes curriculares importantes para o ensino de compiladores. Possui recursos que auxiliam o aluno com os estudos de análise léxica, sintática e semântica, além de oferecer recursos que auxiliam no aprendizado de geração de código intermediário e geração de código objeto (STAMENKOVIć; JOVANOVIć, 2024).

O ComVis foi o ambiente com maior quantidade de recursos voltados ao ensino de compiladores que foi localizado durante a elaboração do presente trabalho, sendo disponibilizado na *web*, ou seja, sem exigir que quaisquer arquivos sejam baixados e/ou instalados. Porém, esse ambiente não disponibiliza recursos que visam auxiliar o aluno no aprendizado de otimizações, tópico este que é considerado pelo presente trabalho como de suma importância no processo de aprendizagem de compiladores, visto que é um dos componentes curriculares da disciplina que mais carece de atividades práticas. Uma das funcionalidades propostas pelo ComVis pode ser visualizada na Figura 6.

Figura 6 – Exemplo de *interface* do *software ComVis*



Fonte: Adaptado de Stamenković e Jovanović (2024).

Vale ressaltar que o ComVis realiza corretamente tudo a que se propõe, sendo um ótimo sistema de estudos para alunos de compiladores. A análise realizada neste momento restringe-se ao fato de que o ambiente não proporciona nenhum aspecto relacionado ao ensino de otimizações de código intermediário, apenas a geração inicial do mesmo.

3.2 *PAVT: PARSING ALGORITHMS VISUALIZATION TOOL*

O ambiente foi desenvolvido em 2018 e, diferente do ComVis mencionado na Seção 3.1, este possui escopo bem mais restrito. O principal objetivo do PAVT é permitir ao aluno da disciplina de compiladores que possa visualizar, de forma interativa, diversos algoritmos de análise, abrangendo aspectos de análise léxica e sintática (SANGAL; KATARIA; TYAGI, 2018).

No PAVT, o usuário deve preencher (ou importar) a gramática, além de preencher uma entrada de texto a ser analisada pela gramática e qual o algoritmo que será aplicado sobre os dados informados. Após isso, é possível visualizar como o algoritmo iria processar a entrada de texto sobre a gramática fornecida.

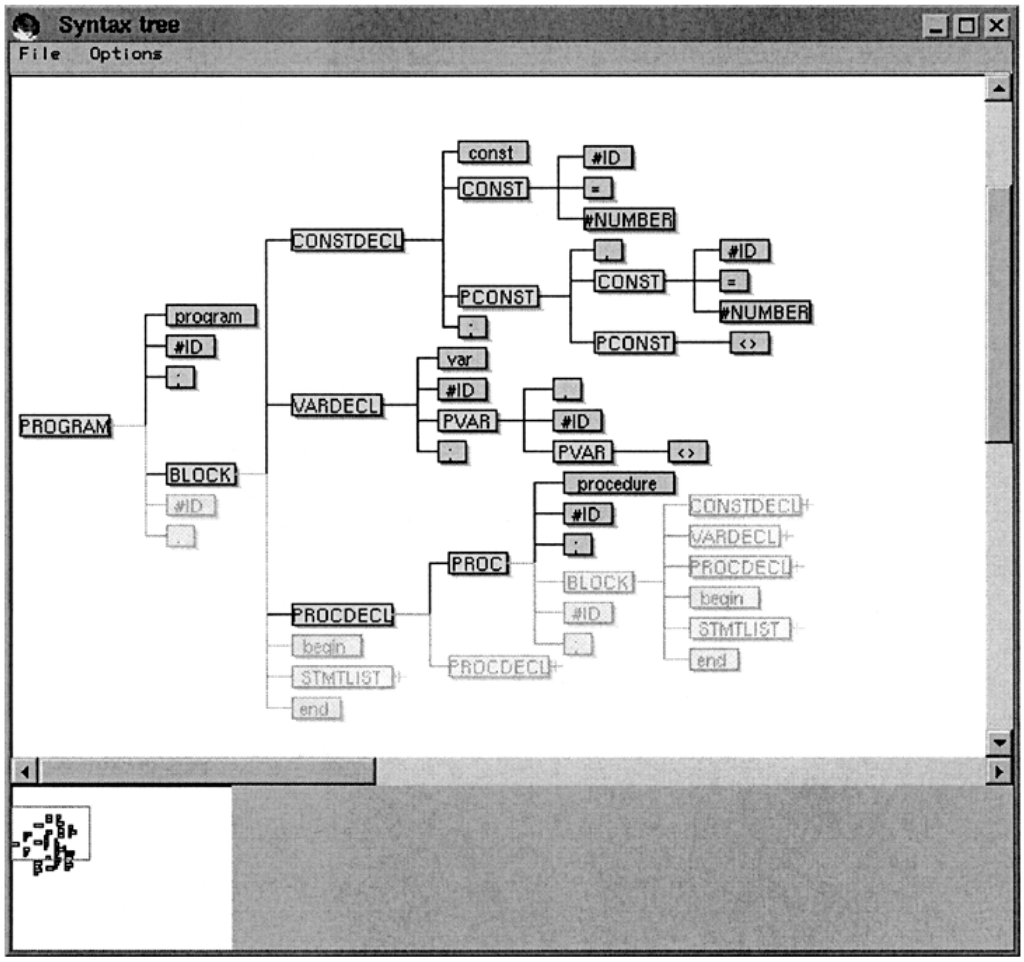
Esse ambiente, assim como o descrito na Seção 3.1, cumpre com o que propõe, mas novamente não apresenta aspectos relevantes para o estudo de otimizações de código, restringindo-se ainda mais em relação ao anterior por apenas abranger conteúdo voltado ao ensino de análise léxica e sintática. No PAVT, não existem recursos direcionados ao ensino de otimizações de código intermediário nem ao ensino de geração do código intermediário (neste último, diferindo-se do ComVis).

3.3 *LISA: LANGUAGE IMPLEMENTATION SYSTEM BASED ON ATTRIBUTE GRAMMARS*

Language Implementation System based on Attribute grammars (LISA) foi publicado em 2003 com o intuito de ser um ambiente de desenvolvimento com foco no ensino de compiladores, mais especificamente no que abrange analisadores léxicos e sintáticos, além de servir como um visualizador de um compilador (ou interpretador) gerado pelo próprio ambiente (MERNIK; ZUMER, 2003).

Esse ambiente é apresentado no presente trabalho como forma de comparação, onde identifica-se o sistema LISA como não ideal para o ensino de compiladores, visto que obriga o usuário/aluno a baixar arquivos e instalar componentes para poder executar o *software* e utilizar-se de seus recursos. Além disso, LISA possui uma *interface* não agradável aos padrões atuais e não é considerado convidativo para novos usuários que desejam aprender mais sobre compiladores, conforme possível visualizar na Figura 7.

Figura 7 – Exemplo de *interface* do *software LISA*



Fonte: Mernik e Zumer (2003)

Com as informações apresentadas, conclui-se que o ambiente LISA apresenta diversas

características a serem evitadas no processo de construção da proposta de solução do presente trabalho.

4 PROPOSTA DE SOLUÇÃO

Com base no que foi apresentado até o momento neste trabalho, propôs-se a elaboração de um ambiente de aprendizado para visualização de código intermediário e suas otimizações. A idealização deste ambiente pode ser visualizada na Seção 4.1. O planejamento de escopo inicial, definido antes do início do desenvolvimento, é listado na Seção 4.3.4.

O ambiente desenvolvido deverá ser capaz de receber como entrada um código em linguagem C e realizar algumas etapas de um compilador: (1) Análise Léxica; (2) Análise Sintática; (3) Análise Semântica; (4) Geração de código intermediário; e (5) Otimização de código. Para cada um dos itens enumerados, são listados seus principais detalhes na Seção 4.4.

4.1 IDEALIZAÇÃO DO AMBIENTE

Visando a criação de um ambiente de aprendizado, idealizou-se o desenvolvimento para plataforma *web*, com a utilização de um *framework*, conforme detalhado posteriormente neste capítulo. Planejou-se a criação de dois projetos, um contendo uma página para servir como *hub* de diversas aplicações e outro sendo o ambiente de aprendizado proposto neste trabalho, que servirá como o primeiro item dentro do *hub*, mas deixando espaço para a inserção de diversos outros ambientes.

4.2 *HUB* DE APLICAÇÕES

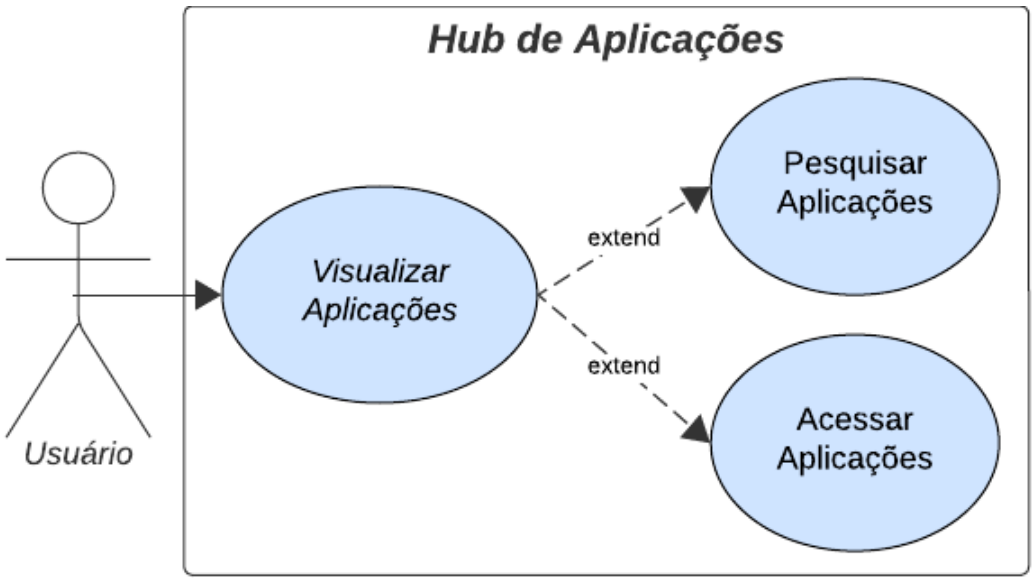
Dentro do *hub* de aplicações proposto, existem diversos *cards*[1], cada um responsável por redirecionar o usuário a um ambiente distinto. O primeiro *card*, conforme mencionado, será responsável por redirecionar o usuário ao ambiente de aprendizado para visualização de código intermediário e suas otimizações, que será descrito na Seção 4.3.

4.2.1 Diagrama de Caso de Uso

Nesta seção, evidencia-se o Diagrama de Caso de Uso proposto para o *hub*, abordando explicações acerca das funcionalidades e do fluxo de interação desejado. Na Figura 8, é possível visualizar o Diagrama de Caso de Uso para o *hub* que está sendo proposto pelo presente trabalho e, a seguir, suas devidas explicações.

[1] Um *card* é um elemento, geralmente retangular, em uma página *web* e que é responsável por apresentar informações sobre um determinado item, por vezes possibilitando interações que evidenciem mais detalhes sobre o item.

Figura 8 – Diagrama de Caso de Uso para o *hub* proposto



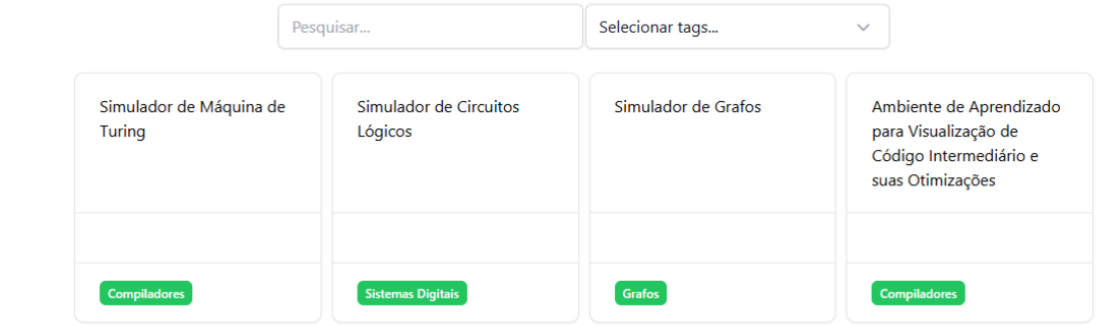
Fonte: O Autor (2024).

4.2.1.1 Caso de Uso 1: Visualizar Aplicações

Para o primeiro caso de uso idealizado para o *hub*, espera-se que o usuário seja capaz de visualizar todas as aplicações inseridas, inicialmente sem filtros aplicados. Essa visualização é composta pela presença de diversos *cards* clicáveis que irão redirecionar o usuário para a aplicação selecionada.

A Figura 9 apresenta a interface proposta para o *hub* de aplicações, evidenciando a presença das aplicações inseridas e sem filtros aplicados.

Figura 9 – Interface do *HUB* de aplicações desenvolvido

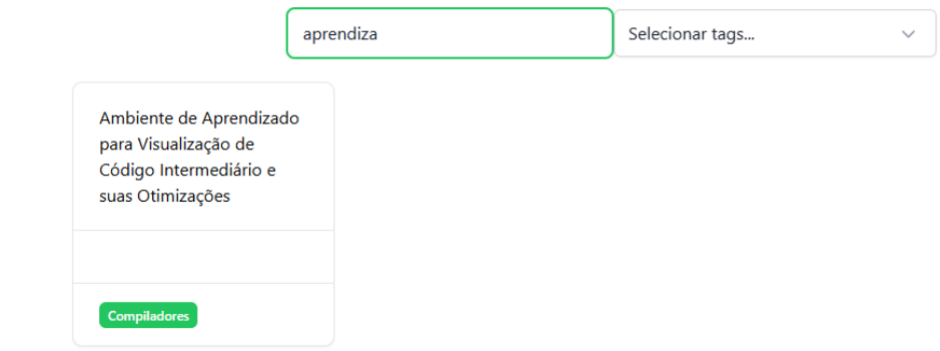


Fonte: O Autor (2024).

4.2.1.2 Caso de Uso 2: Pesquisar Aplicações

A pesquisa de aplicações pode ser realizada através de dois formatos distintos. O primeiro consiste em uma pesquisa direta pelo nome da aplicação através de um campo de texto com edição livre. Nesse primeiro formato, os *cards* apresentados são filtrados na medida em que os caracteres são inseridos no campo de pesquisa. Esse cenário pode ser visualizado na Figura 10, onde é aplicada a pesquisa por texto.

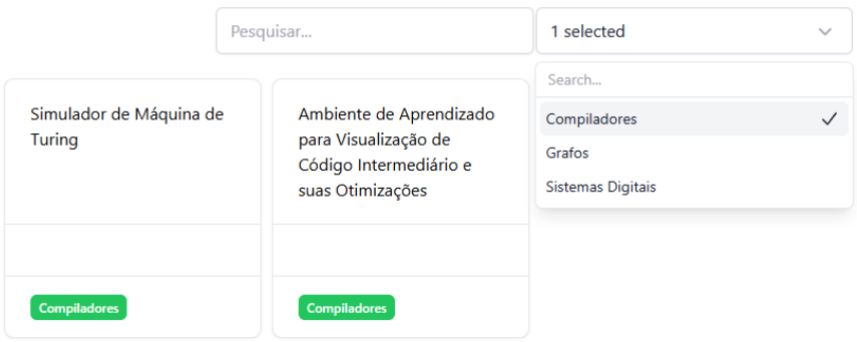
Figura 10 – Interface do *HUB* de aplicações desenvolvido, com pesquisa de texto aplicada



Fonte: O Autor (2024).

O segundo formato de pesquisa consiste na seleção de *tags* para filtrar os resultados de acordo com algumas palavras-chave previamente definidas na aplicação. Nesse modo de pesquisa, múltiplas *tags* podem ser selecionadas ao mesmo tempo, exibindo todos os resultados que tenham em seu cadastro ao menos uma das *tags* selecionadas pelo usuário. Na Figura 11, é possível visualizar os resultados que seriam apresentados em um exemplo de filtro por *tags*.

Figura 11 – Interface do *HUB* de aplicações desenvolvido, com pesquisa por *tags* aplicada



Fonte: O Autor (2024).

Ambos os formatos de pesquisa de aplicações podem ser realizados em paralelo e ambos os filtros serão aplicados ao mesmo tempo. Vale ressaltar que, embora o filtro por *tags* exija

apenas que uma das *tags* esteja vinculada ao *card*, a pesquisa por seleção de *tags* e por pesquisa de texto devem ambas resultar em sucesso para que um *card* seja apresentado. Ou seja, caso ambas as pesquisas sejam aplicadas simultaneamente, mas apenas a pesquisa por texto traga resultados e a pesquisa por *tags* não tenha resultados válidos, nada será apresentado.

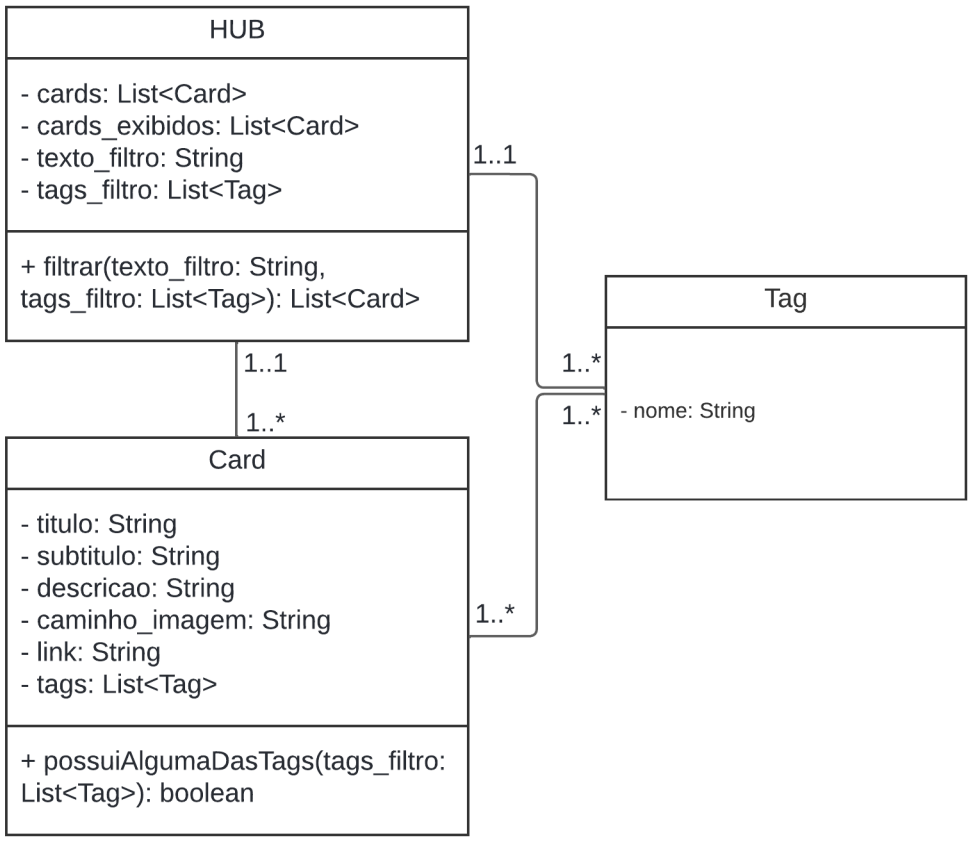
4.2.1.3 Caso de Uso 3: Acessar Aplicações

Esse caso de uso é responsável por permitir que o usuário consiga acessar as aplicações apresentadas no *hub*, através de um simples clique no *card* correspondente à aplicação. Ao clicar no *card* da aplicação que se deseja acessar, uma nova aba será aberta no navegador utilizado, acessando a aplicação enquanto o *hub* continua disponível na aba que já estava aberta.

4.2.2 Diagrama de Classes

Com o objetivo de identificar os relacionamentos e objetos que compõem o sistema proposto, criou-se o diagrama de classes correspondente ao *hub* de aplicações idealizado. O diagrama pode ser visualizado na Figura 12 e suas classes, atributos e métodos são listados abaixo.

Figura 12 – Diagrama de Classes para o *hub* proposto



Fonte: O Autor (2024).

4.2.2.1 *HUB*

A classe *HUB* representa a página completa e tudo que estiver sendo exibido dentro dela depende de sua existência. Partindo desse ponto, justifica-se a relação presente no diagrama da Figura 12, visto que tanto *Card* quanto *Tag* relacionam-se diretamente com *HUB*.

Os atributos listados para essa classe são:

- cards: Atributo do tipo lista contendo objetos do tipo *Card*. Representa a lista completa de todos os *cards* cadastrados no sistema, não tendo relação direta com o que estiver sendo exibido na página.
- cards_exibidos: Atributo do tipo lista contendo objetos do tipo *Card*. Esse atributo, diferenciando-se do atributo cards, possui relação direta com o que está sendo exibido na página. Seu conteúdo é alterado de acordo com o método filtrar, que é explicado abaixo.
- texto_filtro: Atributo do tipo texto (String). Utilizado apenas para armazenar o texto que é inserido pelo usuário no campo de filtro de texto e posteriormente filtrar a lista de *cards*.

- tags_filtro: Atributo do tipo lista contendo objetos do tipo *Tag*. Assim como o atributo texto_filtro, este é utilizado exclusivamente para armazenar as *tags* selecionadas pelo usuário no campo de filtro de *tags* e posteriormente filtrar a lista de *cards*.

O único método abordado pelo diagrama e considerado essencial para o funcionamento do *hub* de aplicações é o método filtrar, que é responsável por receber como entrada dois atributos: texto_filtro e tags_filtro. Após receber esses atributos, deve consultar a lista completa de *cards* (que é armazenada no atributo cards) e filtrar os resultados de acordo com os filtros fornecidos, retornando o resultado e salvando-o no atributo cards_exibidos.

4.2.2.2 *Card*

A classe *Card*, por sua vez, é responsável por abstrair as informações essenciais em todos os *cards* cadastrados no *hub* de aplicações. Conforme mencionado anteriormente, possui relação direta com a classe *HUB*.

Os atributos listados para essa classe são:

- titulo: Atributo do tipo texto (String). Representa o título do *card*.
- subtitulo: Atributo do tipo texto (String). Representa o subtítulo do *card* (opcional).
- descricao: Atributo do tipo texto (String). Representa uma descrição textual do *card* (opcional).
- caminho_imagem: Atributo do tipo texto (String). Representa o caminho relativo da imagem apresentada no *card* (opcional).
- link: Atributo do tipo texto (String). Representa o link de redirecionamento para a aplicação identificada pelo *card*.
- tags: Atributo do tipo lista contendo objetos do tipo *Tag*. Responsável por armazenar todas as *tags* que identificam o *card* de alguma forma. Essa lista é utilizada no momento em que algum filtro for realizado dentro do *hub* de aplicações, como forma de identificação do que deve ser apresentado ou não na página.

No que se refere aos métodos existentes nesta classe, apresenta-se apenas um, entitulado possuiAlgumaDasTags. Esse método é responsável por receber como entrada uma lista de *tags* (chamada de tags_filtro) e verificar se o *card* possui alguma das *tags* informadas como entrada. O retorno desse método consiste apenas em um atributo do tipo booleano identificando se o *card* possui (ou não possui) alguma das *tags* informadas no campo de entrada.

4.2.2.3 *Tag*

A terceira e última classe representada pelo diagrama é a classe *Tag*. Esta classe existe única e exclusivamente para permitir que sejam aplicados filtros predefinidos para exibir uma lista reduzida de *cards* dentro do *hub*, de acordo com as escolhas do usuário.

Apenas um atributo se faz presente nesta classe, entitulado nome. Esse atributo é do tipo texto (String) e apenas identifica o nome da *tag*, valor que é utilizado no filtro de *tags* do *hub* de aplicações e também existe dentro dos *cards* para que o filtro funcione adequadamente. Esta classe não apresenta métodos, sendo responsável apenas por armazenar o nome das *tags*.

4.3 AMBIENTE DE APRENDIZADO

O ambiente de aprendizado proposto no presente trabalho visa, de modo geral, evidenciar ao usuário algumas atividades executadas pelo compilador. Isso será evidenciado através de recursos gráficos e textuais, exibindo ao usuário de forma detalhada o que está acontecendo em cada etapa, facilitando assim o processo de aprendizagem do estudante.

Para o desenvolvimento da aplicação mencionada, decidiu-se pela utilização do *Nuxt*[2], um *framework* construído com base em outro *framework*, este chamado *Vue*[3]. Embora existam argumentos favoráveis e contrários à utilização desse *framework*, a decisão motivou-se exclusivamente por interesse próprio do autor, visto que o mesmo expressa vontade em aprofundar seus conhecimentos em *Nuxt*. Mais detalhes sobre ambos os *frameworks* mencionados são abordados na Seção 4.5.

4.3.1 Gerações do ambiente

O ambiente de aprendizado proposto deve ser responsável por gerar, a partir de um código de origem inserido pelo usuário, alguns recursos visuais que facilitem a compreensão do usuário sobre a conversão do código de origem até o código de destino (em formato de RI). Esses recursos abrangem grafos e diagramas, permitindo a visualização por parte do usuário das diferentes etapas do processo de conversão de código.

4.3.2 Destaque de forma relacionada

O destaque de linhas de código de forma relacionada deve ser apresentado com a utilização do estado de sobreposição do *cursor* do usuário, ou seja, ao passar o *cursor* sobre uma determinada linha do código de origem, os comandos correspondentes deverão ser destacados de forma automática e imediata. Vale ressaltar que o mesmo deve ocorrer no processo inverso,

[2] https://nuxt.com/
[3] https://vuejs.org/

destacando as linhas do código de origem correspondentes ao que estiver sendo sobreposto pelo *cursor* do usuário.

4.3.3 Seleção múltipla e ordenada de otimizações

Outra funcionalidade importante de ser ressaltada é a de seleção múltipla e ordenada das otimizações aplicadas sobre o código intermediário resultante. Através disso, o usuário deve ser capaz de selecionar quais otimizações são aplicadas ou não e também definir a ordem de aplicação das mesmas. Este é um recurso de extrema importância para o projeto, visto que irá resultar em diferentes códigos intermediários resultantes e influencia no processo de aprendizagem do estudante sobre as otimizações estudadas.

4.3.4 Escopo de otimizações

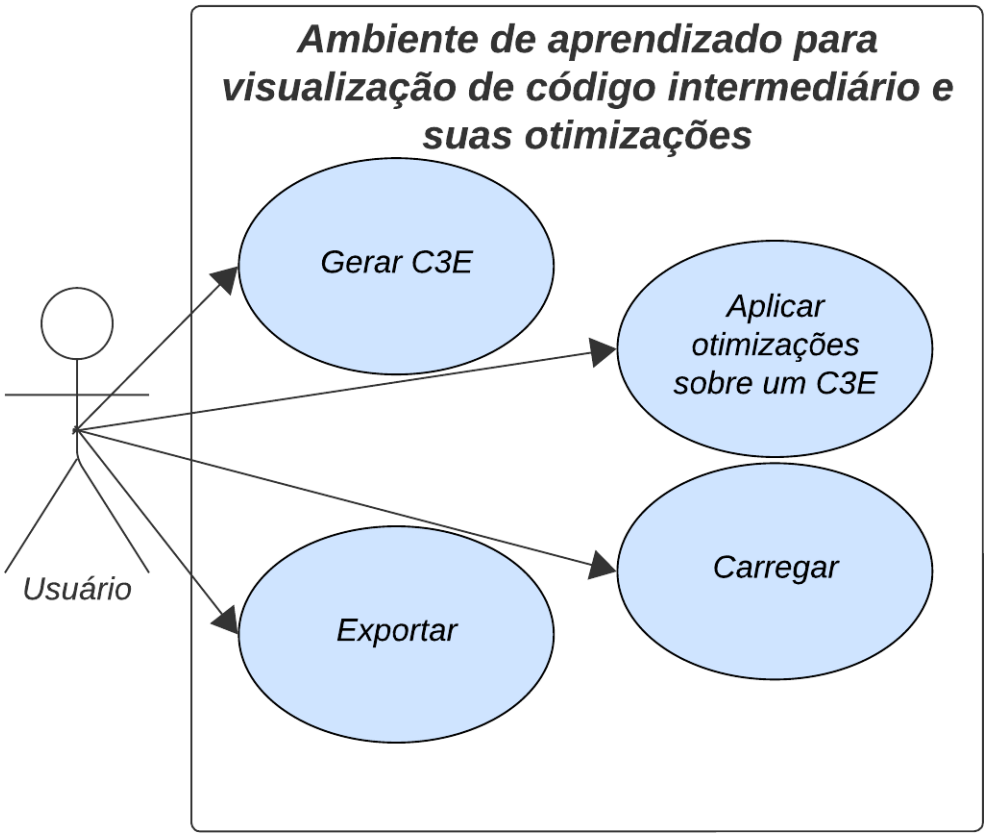
Visto que existem diversas otimizações possíveis em diferentes compiladores e que mesmo as citadas na Seção 2.3 não representam todas as possibilidades existentes, assume-se que algumas restrições são aplicadas ao produto final do presente trabalho. Tendo isso em consideração, realizou-se uma tarefa de definição de prioridades em relação a quais otimizações são consideradas como essenciais para o desenvolvimento deste trabalho. O resultado desta tarefa de priorização, para a primeira versão do ambiente produzido, definiu como essenciais as seguintes otimizações:

- Eliminação de código morto
- Propagação de cópia
- Computações laço-invariantes
- Identificação e remoção de variáveis de indução

4.3.5 Diagrama de Caso de Uso

Nesta seção, é apresentado o Diagrama de Caso de Uso proposto, abordando explicações acerca das funcionalidades e fluxo de interação desejado no ambiente desenvolvido. Na Figura 13, é possível visualizar o Diagrama de Caso de Uso para o ambiente de aprendizado que está sendo proposto pelo presente trabalho e, logo abaixo, as devidas explicações.

Figura 13 – Diagrama de Caso de Uso para o ambiente proposto



Fonte: O Autor (2024).

4.3.5.1 Caso de Uso 1: Gerar C3E

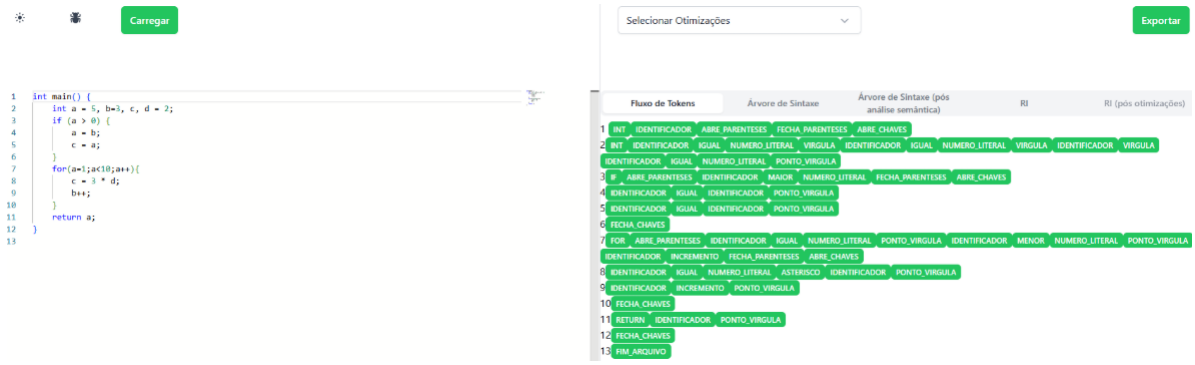
Neste primeiro caso de uso, o principal objetivo é permitir que o usuário insira o código fonte a ser processado pela aplicação e visualize o C3E resultante. Para isso, idealiza-se que exista um campo editável onde seja possível digitar código diretamente, assim como colar trechos de código ou até importar código a partir de um arquivo.

Esta etapa de inserção de código é considerada como fundamental e obrigatória para todos os demais recursos apresentados a seguir, visto que todas as funcionalidades do ambiente partem do pressuposto de que existe um trecho de código a ser processado e a partir dele seja possível gerar resultados.

Após a inserção do código, o ambiente é responsável por realizar o processamento do código inserido e convertê-lo em uma RI (neste trabalho, a RI será sempre um C3E). Todo o processo realizado para converter o código inserido em um C3E é descrito na Seção 4.4.

Todos os itens descritos para este caso de uso podem ser visualizados na Figura 14. Além do que foi descrito, observa-se a presença de diferentes abas para visualizar os resultados das diferentes etapas do processo de conversão.

Figura 14 – Interface do ambiente de aprendizado desenvolvido



Fonte: O Autor (2024).

4.3.5.2 Caso de Uso 2: Aplicar otimizações sobre um C3E

Além das funcionalidades apresentadas na Seção 4.3.5.1, este caso de uso apresenta recursos adicionais no que tange a possibilidade de aplicar diferentes otimizações sobre o C3E, além de definir a ordem de aplicação das mesmas.

Para a seleção de quais otimizações serão aplicadas sobre o código intermediário produzido, é esperado que o usuário seja capaz de selecionar se serão e quais serão as otimizações aplicadas, de forma fácil e intuitiva. Para que isso seja possível, idealiza-se a presença de uma lista predefinida de otimizações, conforme apontado na Seção 4.3.4, permitindo que o usuário selecione quais otimizações deseja aplicar sobre a RI obtida após inserir o código fonte.

Assim como planeja-se possibilitar que o usuário selecione as otimizações que deseja aplicar sobre a RI, é desejável que o mesmo possa definir a ordem com que elas são aplicadas, visto que isso é de extrema importância no momento da geração do código intermediário, onde ordenações diferentes de um mesmo conjunto de otimizações podem gerar diferentes resultados.

Ressalta-se que os resultados serão apresentados mesmo que não ocorra o processo de ordenação de aplicação de otimizações. Mesmo assim, esta opção fica disponível para ser utilizada a qualquer momento, de forma que gere novos resultados assim que a ordem for alterada, sem exigir que o usuário indique a necessidade de reprocessamento do código. Essa abordagem é considerada como ideal para este cenário, visto que procura mitigar possíveis confusões e erros de interpretação que poderiam ocorrer caso o usuário alterasse a ordem das otimizações e novos resultados não fossem automaticamente apresentados.

Na Figura 15, é possível visualizar como será realizado, pelo usuário, o processo de seleção de quais otimizações serão aplicadas sobre a RI, além da ordem de aplicação das mesmas. Idealiza-se que o usuário seja capaz de simplesmente marcar em cada otimização se ela deverá ser aplicada, além de poder mover uma determinada otimização dentro da lista, movendo-a para cima caso deseje que sua aplicação seja realizada antes de outras ou movendo-a para baixo caso deseje que a otimização seja aplicada depois de outras.

Figura 15 – Interface do ambiente de aprendizado desenvolvido, com foco na seleção e ordenação de otimizações

Fonte: O Autor (2024).

4.3.5.3 Caso de Uso 3: Carregar

Para o terceiro caso de uso, objetiva-se permitir que o usuário seja capaz de carregar configurações e resultados previamente obtidos no ambiente, seja pelo próprio usuário ou por outro usuário que disponibilizar um arquivo exportado através da funcionalidade descrita na Seção 4.3.5.4.

Este processo é realizado através de um botão que fica sempre presente na tela, conforme possível observar na Figura 14. O botão que representa a funcionalidade descrita neste caso de uso é o que possui o texto ’Carregar’.

4.3.5.4 Caso de Uso 4: Exportar

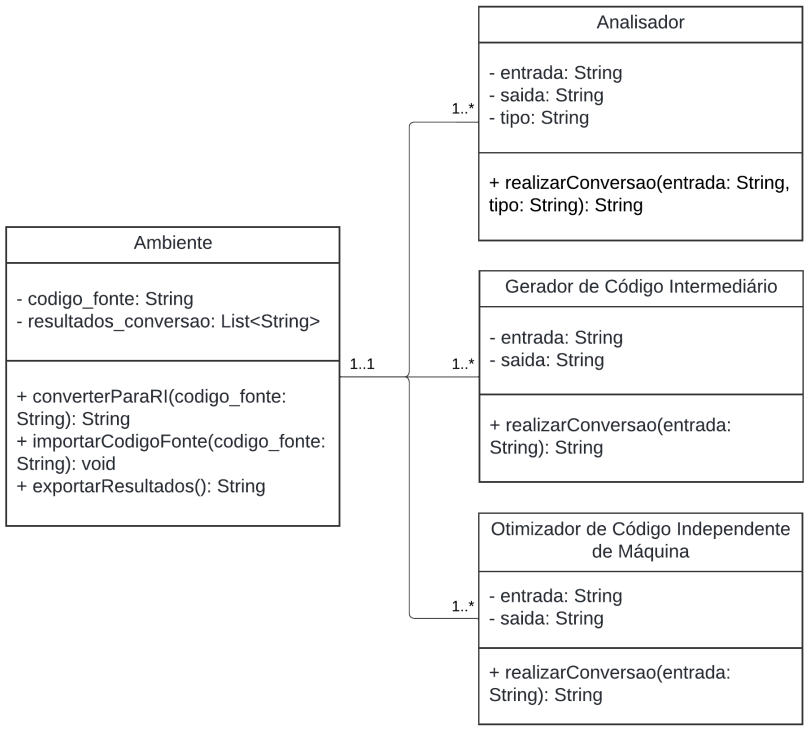
Neste caso de uso, idealiza-se que o usuário possa exportar as configurações e resultados obtidos ao longo de sua atual utilização do ambiente. Todos os dados são exportados em um arquivo único a ser baixado pelo usuário, que posteriormente pode ser carregado pelo próprio ou por outro que possuir o arquivo exportado. O processo de carregar novamente as configurações e resultados presentes no arquivo é descrito na Seção 4.3.5.3.

Este processo é realizado através de um botão que fica sempre presente na tela, conforme possível observar na Figura 14. O botão que representa a funcionalidade descrita neste caso de uso é o que possui o texto ’Exportar’.

4.3.6 Diagrama de Classes

Visando identificar os relacionamentos e objetos que compõem o sistema proposto, criou-se o diagrama de classes correspondente ao ambiente de aprendizado descrito até o momento. O diagrama pode ser visualizado na Figura 16 e suas classes, atributos e métodos são listados abaixo.

Figura 16 – Diagrama de Classes para o ambiente de aprendizado proposto



Fonte: O Autor (2024).

4.3.6.1 Ambiente

A classe Ambiente representa a página completa e tudo que estiver sendo exibido dentro dela depende de sua existência. Partindo desse ponto, justifica-se a relação presente no diagrama da Figura 16, visto que todas as outras classes se relacionam diretamente com a classe Ambiente.

Os atributos listados para essa classe são:

- codigo_fonte: Atributo do tipo texto (String). Representa o código fonte inserido pelo usuário na aplicação.
- resultados_conversao: Atributo do tipo lista contendo objetos do tipo texto (String). Esse atributo é responsável por armazenar os resultados de todos os processos de conversão realizados sob o código conte, conforme descrito na Seção 4.4.

Em relação aos métodos dessa classe, destacam-se os seguintes:

- converterParaRI: Método responsável por executar cada uma das etapas do processo de conversão descrito na Seção 4.4, de acordo com a entrada do usuário. Após realizar o processo de conversão, esse método também é responsável por exibir os resultados obtidos.
- importarCodigoFonte: Este método é utilizado exclusivamente para importar um código fonte dos arquivos locais do usuário para a aplicação, armazenando o código no atributo codigo_fonte.
- exportarResultados: Semelhantemente ao processo descrito no método importarCodigoFonte, o método exportarResultados também interage diretamente com os arquivos locais do usuário. Porém, neste método, os resultados obtidos após o processo de conversão são exportados para arquivos locais, em diferentes formatos, ficando disponibilizados para *download* pelo usuário.

4.3.6.2 Analisador

A classe Analisador representa três das etapas do processo de conversão descrito na Seção 4.4: Analisador Léxico, Analisador Sintático e Analisador Semântico.

Os atributos listados para essa classe são:

- entrada: Atributo do tipo texto (String). Representa a respectiva entrada. Quando o tipo do analisador for Léxico, terá o valor do código fonte inserido pelo usuário. Para os demais tipos, todos terão nesta variável o valor de saída correspondente a etapa anterior do processo de conversão.
- saida: Atributo do tipo texto (String). Representa a saída da classe após a respectiva etapa do processo de conversão que a classe se faz responsável por realizar.
- tipo: Atributo do tipo texto (String). Representa o tipo de analisador que está sendo utilizado, podendo ser Léxico, Sintático ou Semântico.

No que se refere aos métodos existentes nesta classe, apresenta-se apenas um, entitulado realizarConversao. Esse método é responsável por realizar uma etapa do processo de conversão, recebendo uma entrada textual e produzindo uma saída, também textual. Ressalta-se que tanto a entrada como a saída de cada etapa ficam armazenadas em sua respectiva classe.

4.3.6.3 Gerador de Código Intermediário

A classe Gerador de Código Intermediário representa uma das etapas do processo de conversão descrito na Seção 4.4, que possui o mesmo nome da classe. Observa-se que tanto

a classe Analisador quanto a classe Gerador de Código Intermediário são consideradas como obrigatórias no processo de conversão.

Os atributos listados para essa classe são:

- entrada: Atributo do tipo texto (String). Representa a respectiva entrada. Terá, nesta variável, o valor de saída correspondente a etapa anterior do processo de conversão.
- saida: Atributo do tipo texto (String). Representa a saída da classe após a respectiva etapa do processo de conversão que a classe se faz responsável por realizar.

No que se refere aos métodos existentes nesta classe, apresenta-se apenas um, entitulado realizarConversao. Esse método é responsável por realizar uma etapa do processo de conversão, recebendo uma entrada textual e produzindo uma saída, também textual. Assim como para a classe Analisador, tanto a entrada como a saída de cada etapa ficam armazenadas em sua respectiva classe.

4.3.6.4 Otimizador de Código Independente de Máquina

Assim como as classes listadas na Seção 4.3.6.2 e na Seção 4.3.6.3, a classe Otimizador de Código Independente de Máquina também é responsável por realizar uma das etapas do processo de conversão descrito na Seção 4.4. Todavia, esta se diferencia das demais considerando-se que sua presença no processo é facultativa. Caso o usuário opte por realizar o caso de uso da Seção 4.3.5.1, nenhuma otimização será aplicada na RI resultante, tornando obsoleta a etapa de otimização de RI.

Os atributos e métodos da classe Otimizador de Código Independente de Máquina convergem com o que foi descrito na Seção 4.3.6.3, isto é, esta classe possui os mesmos atributos e métodos que a classe Gerador de Código Intermediário.

4.4 CONVERSÃO DO CÓDIGO INSERIDO EM C3E

Para realizar a conversão do código em C3E, um compilador precisa realizar diversas etapas. Nesta seção, são listadas as principais etapas percorridas pelo ambiente proposto visando completar esse processo de conversão.

Inicialmente, o usuário é responsável apenas por fornecer um fluxo de caracteres (assume-se, neste momento, que esse fluxo de caracteres é um código em linguagem C). O ambiente deve, então, processar o código inserido pelo usuário em seu Analisador Léxico, transformando-o em um fluxo de tokens predefinidos na aplicação.

Com o fluxo de tokens definido, a segunda etapa realizada pelo ambiente é a de processamento pelo Analisador Sintático. Nesta etapa, o fluxo de tokens é convertido em uma árvore

de sintaxe. Vale ressaltar que a árvore de sintaxe gerada por esta etapa deve permanecer visível ao usuário, assim como o fluxo de tokens que foi gerado na etapa anterior e também todos os resultados das etapas futuras.

A árvore de sintaxe gerada pelo Analisador Sintático é submetida a outro analisador, desta vez Semântico, onde será convertida em uma nova árvore de sintaxe (conforme mencionado anteriormente, ambas as árvores devem permanecer disponíveis para visualização por parte do usuário). Nesta etapa, são realizadas verificações previamente não realizadas pelo Analisador Sintático, como por exemplo a verificação de tipos entre operações e verificação de escopo dos identificadores.

A quarta etapa, considerada neste momento como a última etapa obrigatória (ou seja, sempre será executada, assim como as anteriores), é a de conversão da árvore de sintaxe para uma RI, através de um Gerador de Código Intermediário. Define-se esta como a última etapa obrigatória em virtude de que a próxima etapa, de otimização de código, é facultativa e depende de uma decisão do usuário para ser aplicada ou não sobre os resultados gerados.

A etapa de otimização da RI é realizada por um Otimizador de Código Independente de Máquina e é responsável por aplicar diversas otimizações (conforme selecionado pelo usuário) na RI existente, resultando em uma nova RI no final do processo. No presente trabalho, sempre que referindo-se a uma RI, assume-se que será implementado um C3E.

4.5 *FRAMEWORK* PARA DESENVOLVIMENTO

Conforme brevemente mencionado na Seção 4.3, optou-se pela utilização do *framework* *Nuxt* para o desenvolvimento do projeto proposto, decisão esta que é motivada pelo interesse próprio do autor. Nesta seção, são abordados alguns tópicos que visam descrever as principais funcionalidades do *framework* utilizado no que tange a presente proposta de solução.

4.5.1 *Vue*

Antes de abordar os principais diferenciais do *Nuxt*, é importante ressaltar que ele é um *framework* voltado ao desenvolvimento de aplicativos *Vue* e, portanto, se faz necessária a contextualização de sua origem e funcionalidades herdadas do *Vue*, que também é amplamente utilizado para a criação de *Single-Page Applications* (Aplicações de página única) (SPAs) (Vue, 2024).

O *Vue* é um *framework* para desenvolvimento *front-end* que, assim como diversos outros, possui o objetivo de facilitar o processo de desenvolvimento de interfaces em plataformas *web*. Este *framework* possui como uma de suas principais características a utilização da estratégia de programação baseada em componentes. Todos os elementos de uma página em *Vue*, que são reutilizados em algum momento, idealmente são abstraídos em componentes para posterior

reutilização com passagem de parâmetros, atuando como alternativa à duplicação de código e facilitando a modularização das SPAs (MACRAE, 2018).

4.5.2 *Nuxt*

O *Nuxt*, conforme mencionado, é um *framework* desenvolvido para criar aplicações *Vue* intuitivas e com bom desempenho (Nuxt, 2024). Possui diversos recursos que o tornam a principal escolha de muitos desenvolvedores, porém apenas os mais relevantes para o presente trabalho serão abordados a seguir.

- Configuração simplificada: *Nuxt* permite que as configurações iniciais de um projeto sejam mínimas, evitando perda de tempo e a complexidade do processo de configuração do projeto em suas fases iniciais (Nuxt, 2024).
- Estrutura de pastas organizada: Existe, dentro do *framework*, uma convenção sobre a configuração, possibilitando uma estrutura de pastas padronizada e organizada, facilitando o processo de desenvolvimento (Nuxt, 2024).
- Componentização com *Vue*: Por ser baseado em *Vue*, o *Nuxt* permite o aproveitamento da modularidade e da reatividade dos componentes do *Vue*, permitindo por sua vez a criação de interfaces dinâmicas (Nuxt, 2024).
- Gerenciamento de estados: Existem recursos internos do *framework* que permitem o gerenciamento de estados centralizado, tornando mais simplificado tanto o controle quanto a manutenção dos estados da aplicação (Nuxt, 2024).

4.6 ESCOPO DO PROJETO

Considerando o tempo previsto para o desenvolvimento do presente trabalho, definiu-se um escopo inicial de implementação. Esse escopo visa abordar o subconjunto da linguagem C que será implementado ao longo do processo de desenvolvimento. Abaixo, é possível visualizar uma lista contendo diversos tópicos que foram definidos como objetivos iniciais de implementação ao longo do presente trabalho.

- Escalares
- Vetores
- Vetores Bidimensionais
- Chamadas de funções (com passagens de parâmetros por valor)
- Tipos numéricos

Vale ressaltar que, embora diversos itens não estejam presentes na lista (como *enums*, *structs* e ponteiros), outros itens podem vir a ser implementados caso torne-se viável durante o processo de desenvolvimento.

No que se refere ao processo de conversão para a RI em formato de C3E, observa-se que há planejamento de implementação dos itens de 1 a 8 definidos na Seção 2.2, ou seja, apenas as instruções de atribuição de endereço e apontador não foram incluídas no escopo de implementação para esta primeira versão do projeto proposto, em virtude do tempo disponível e da complexidade de desenvolvimento deste item em específico.

4.7 PROPOSTA DE VALIDAÇÃO

Visando validar o que foi desenvolvido, idealizou-se que o produto desenvolvido no presente trabalho fosse apresentado à próxima turma de compiladores da UCS. A partir desta apresentação à turma, esperou-se obter opiniões sobre possíveis melhorias e correções não identificadas ao longo do processo de desenvolvimento.

Além disso, planejou-se aplicar uma pesquisa aos alunos para validar a praticidade do ambiente e tentar mensurar quão prático ele será considerado em relação aos métodos hoje aplicados. Visto que, até a data de elaboração do presente trabalho, nenhum ambiente de aprendizado voltado para o ensino de otimizações de RI em compiladores é utilizado, é idealizado que sejam obtidos resultados positivos no que diz respeito à satisfação dos alunos, visando facilitar o processo de aprendizagem e torná-lo mais visual e prático do que teórico.

5 DESENVOLVIMENTO

A partir do que foi descrito ao longo do Capítulo 4, realizou-se o desenvolvimento dos projetos propostos. Este capítulo é responsável por detalhar os principais assuntos que tangem o processo de desenvolvimento realizado ao longo do presente trabalho.

5.1 DESENVOLVIMENTO DO *HUB* DE APLICAÇÕES

Referente ao desenvolvimento do *HUB* de aplicações, ressalta-se que em relação ao que havia sido proposto na Seção 4.2.1, todos os casos de uso propostos foram desenvolvidos.

Dentro do projeto desenvolvido, é possível que sejam visualizados todos os *cards* cadastrados, além de existir a possibilidade de realizar pesquisas por aplicações (via texto ou via seleção de *tags*) e acessar as aplicações listadas (conforme descrito na Seção 4.2.1.3).

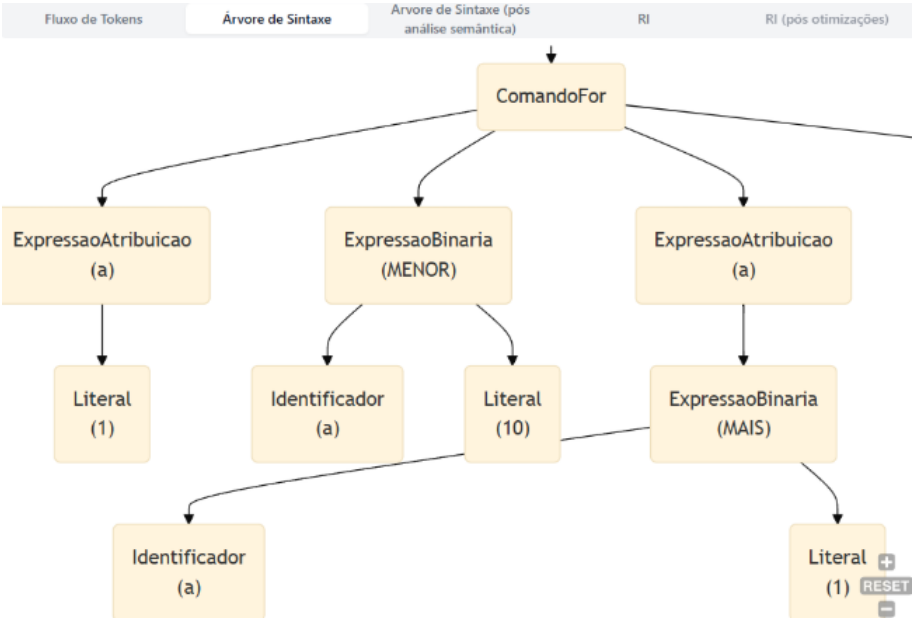
5.2 DESENVOLVIMENTO DO AMBIENTE DE APRENDIZADO

Em relação ao desenvolvimento do ambiente de aprendizado proposto na Seção 4.3, ressaltam-se a seguir os principais tópicos observados ao longo do processo de desenvolvimento.

5.2.1 Gerações do ambiente

No que se refere às gerações realizadas pelo ambiente, destaca-se neste primeiro momento a presença da Árvore de Sintaxe apresentada na Figura 22.

Figura 17 – Interface do ambiente de aprendizado desenvolvido, evidenciando uma Árvore de Sintaxe gerada pelo ambiente



Fonte: O Autor (2024).

Observa-se também que, embora neste momento não existam tantos recursos gráficos que possibilitem ao usuário da aplicação visualizar em detalhes todas as etapas do processo de conversão de código, já é possível visualizar os resultados produzidos em cada etapa, desde a conversão do código em fluxo de *tokens* até a produção do C3E otimizado, conforme descrito na Seção 4.4.

Mais detalhes sobre a criação de novos recursos gráficos para representar as etapas de conversão são mencionados no Capítulo 6.

5.2.2 Destaque de forma relacionada

Este item, em específico, não foi desenvolvido em virtude de sua complexidade não prevista inicialmente. Embora este recurso seja visto como um grande facilitador da utilização do ambiente e como um possibilitador de uma visualização mais fácil dos resultados obtidos no processo de conversão, optou-se por deixar essa funcionalidade de lado nesta primeira versão do ambiente para que outras funcionalidades fossem priorizadas (estas consideradas mais importantes para a primeira versão publicada do ambiente desenvolvido).

5.2.3 Seleção múltipla e ordenada de otimizações

Conforme descrito na Seção 4.3.3, houve o desenvolvimento de uma funcionalidade que permitisse ao usuário selecionar quais otimizações seriam aplicadas sobre a RI e em que ordem isso iria acontecer. Conforme possível visualizar na Figura 15, é possível que o usuário realize

os processos mencionados e decida como as otimizações serão aplicadas sobre a RI obtida após o processo de conversão do código inserido.

5.2.4 Escopo de otimizações

Em relação às otimizações implementadas, destaca-se que o escopo permaneceu muito próximo do previsto. Das quatro otimizações mencionadas na Seção 4.3.4, três foram implementadas ao longo do processo de desenvolvimento, deixando para futura implementação apenas a Identificação e remoção de variáveis de indução. Vale ressaltar que existe a possibilidade de adição de novas otimizações nesta lista e mais detalhes são abordados no Capítulo 6.

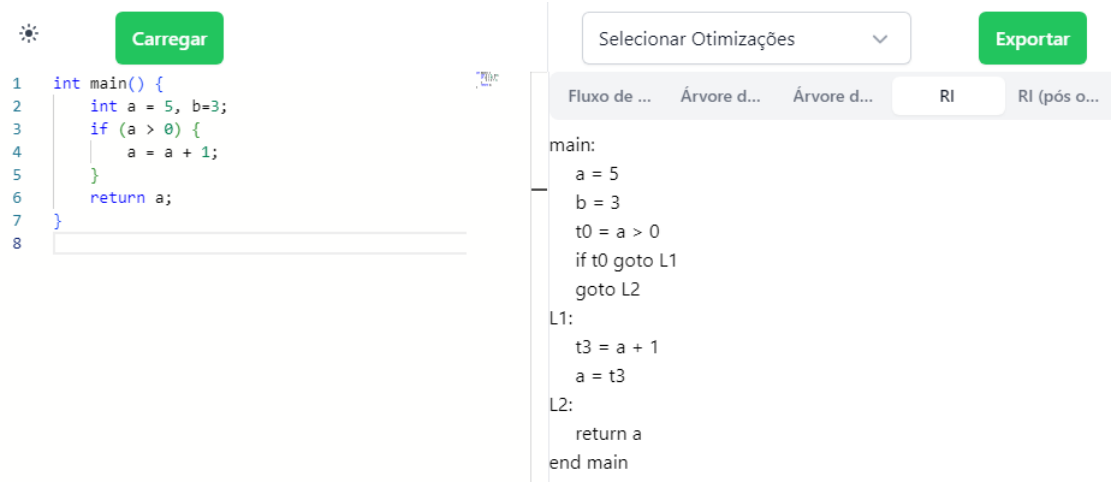
5.2.5 Casos de Uso

Referente aos casos de uso descritos dentro da Seção 4.3.5, evidenciam-se a seguir os principais aspectos de cada um dos casos de uso previstos.

5.2.5.1 Caso de Uso 1: Gerar C3E

Conforme havia sido proposto na Seção 4.3.5.1, o desenvolvimento deste Caso de Uso permitiu que o usuário fosse capaz de, através da utilização do ambiente, gerar C3E a partir de seu código fonte. Uma exemplificação deste processo pode ser visualizada na Figura 18, onde fica evidenciado o código fonte na lateral esquerda da figura e o C3E resultante na lateral direita da figura.

Figura 18 – Interface do ambiente de aprendizado desenvolvido (evidenciando uma RI gerada em formato de C3E)

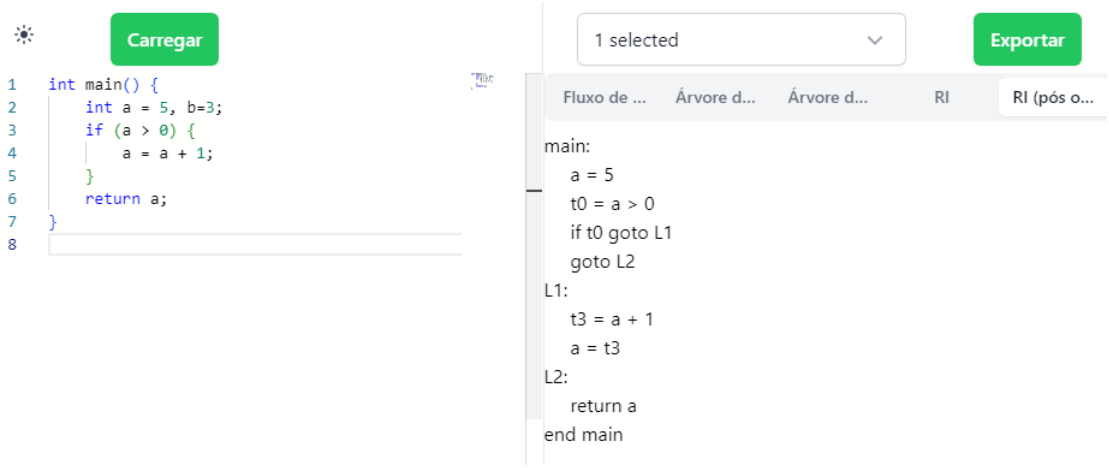


Fonte: O Autor (2024).

5.2.5.2 Caso de Uso 2: Aplicar otimizações sobre um C3E

Conforme havia sido proposto na Seção 4.3.5.2, o desenvolvimento deste Caso de Uso permitiu que o usuário fosse capaz de, através da utilização do ambiente, otimizar uma RI em formato de C3E, gerando assim um outro C3E. Uma exemplificação deste processo pode ser visualizada na Figura 19, onde fica evidenciado o C3E gerado após o processo de otimização. Nesta figura, destaca-se que foi aplicada a otimização de Eliminação de Código Morto. Após a aplicação desta otimização, percebe-se que a operação $b = 3$ é removida (em relação ao que havia sido apresentado na Figura 18).

Figura 19 – Interface do ambiente de aprendizado desenvolvido (evidenciando uma RI gerada em formato de C3E após otimização)



Fonte: O Autor (2024).

5.2.5.3 Casos de Uso 3 e 4: Carregar e Exportar

Conforme havia sido proposto nas Seções 4.3.5.3 e 4.3.5.4, o desenvolvimento destes Casos de Uso permitiu que o usuário fosse capaz de, através da utilização do ambiente, exportar e carregar configurações e resultados do ambiente de aprendizado. Este processo ficou responsável por carregar ou exportar o código fonte e as otimizações selecionadas (assim como sua respectiva ordem). Vale ressaltar que essas informações são sempre carregadas ou exportadas em formato *JavaScript Object Notation* (Notação de Objeto JavaScript) (JSON) dentro de um arquivo de texto, permitindo futuras adições de informações adicionais, caso necessário).

5.2.6 Exemplo de uso

Com o objetivo de exemplificar e ilustrar a usabilidade da aplicação desenvolvida, é descrito nesta seção um exemplo de código que poderia ser inserido e os devidos resultados gerados pela aplicação.

Inicialmente, o usuário é responsável por inserir o código em linguagem C. O código utilizado neste exemplo é evidenciado na Figura 20.

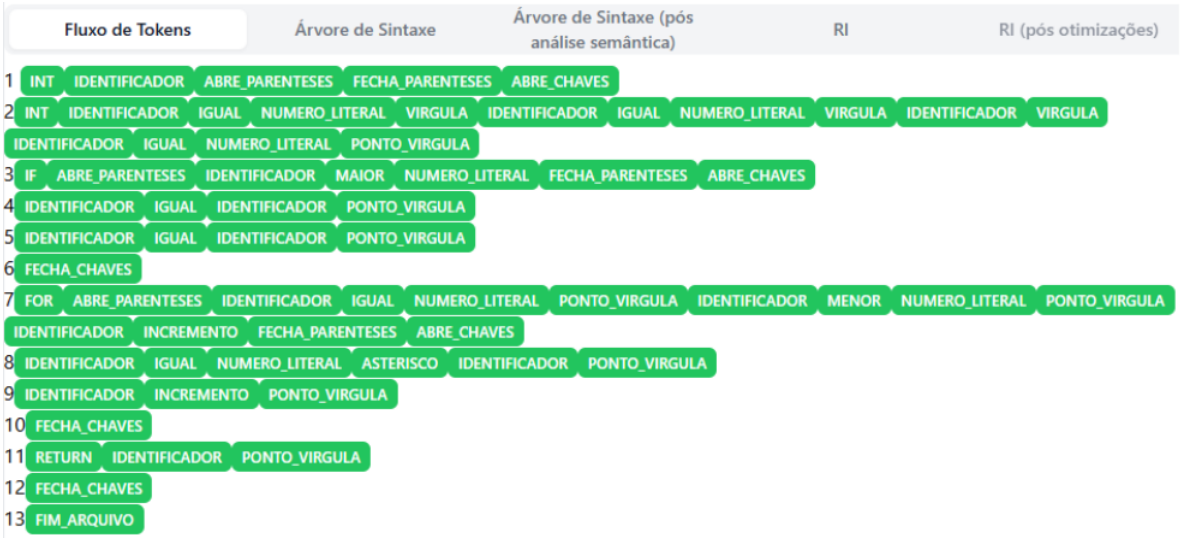
Figura 20 – Interface do ambiente de aprendizado desenvolvido, evidenciando o código inserido pelo usuário

```
1   int main() {
2       int a = 5, b=3, c, d = 2;
3       if (a > 0) {
4           a = b;
5           c = a;
6       }
7       for(a=1;a<10;a++){
8           c = 3 * d;
9           b++;
10      }
11      return a;
12  }
13
```

Fonte: O Autor (2024).

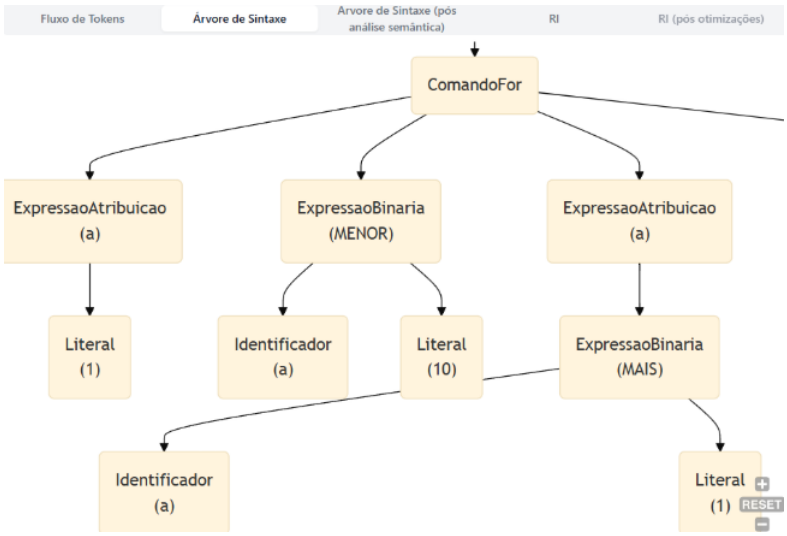
Após a inserção do código por parte do usuário, a aplicação automaticamente irá gerar os resultados que podem ser visualizados nas Figuras 21, 22 e 23. Vale ressaltar que, na Figura 22, apenas parte do resultado está sendo exibido em virtude de que a Árvore de Sintaxe gerada dificultaria a leitura do trabalho caso apresentada por completo.

Figura 21 – Interface do ambiente de aprendizado desenvolvido, evidenciando o fluxo de tokens gerado



Fonte: O Autor (2024).

Figura 22 – Interface do ambiente de aprendizado desenvolvido, evidenciando parte da árvore de sintaxe gerada



Fonte: O Autor (2024).

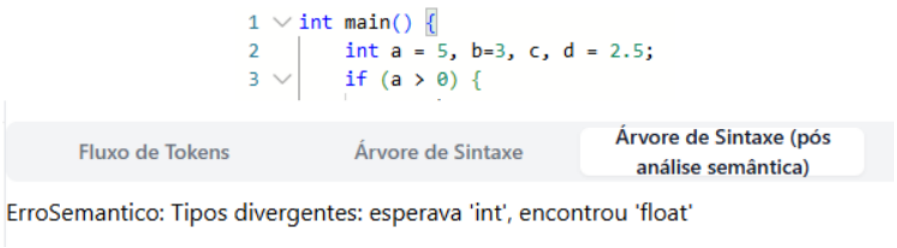
Figura 23 – Interface do ambiente de aprendizado desenvolvido, evidenciando a RI gerada

```
main:
    a = 5
    b = 3
    d = 2
    t0 = a > 0
    if t0 goto L1
    goto L3
L1:
    a = b
    c = a
L3:
    a = 1
L4:
    t7 = a < 10
    if t7 goto L5
    goto L6
L5:
    t8 = 3 * d
    c = t8
    t9 = b + 1
    b = t9
    t10 = a + 1
    a = t10
    goto L4
L6:
    return a
end main
```

Fonte: O Autor (2024).

Caso o usuário realize algum erro semântico dentro do código inserido, a aplicação deve ser responsável por apresentar o erro ao usuário, conforme pode ser visualizado na Figura 24.

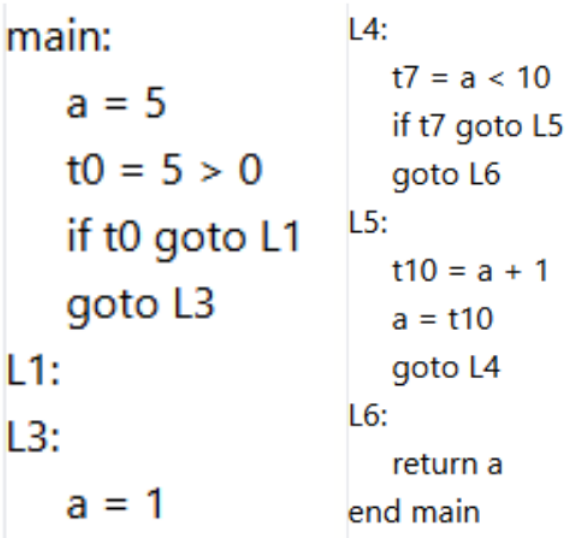
Figura 24 – Interface do ambiente de aprendizado desenvolvido, evidenciando um erro semântico



Fonte: O Autor (2024).

Em relação às otimizações, utiliza-se a Figura 15 como base para o exemplo. Nesse caso, a RI otimizada que a aplicação gera pode ser visualizada na Figura 25.

Figura 25 – Interface do ambiente de aprendizado desenvolvido, evidenciando a RI otimizada gerada



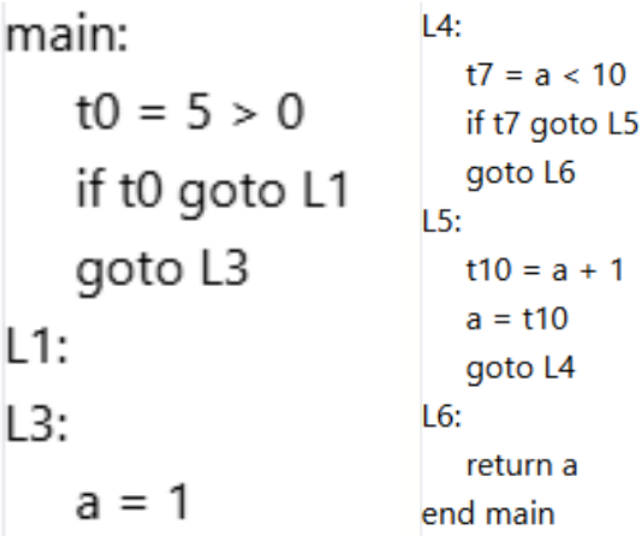
Fonte: O Autor (2024).

Caso o usuário tenha interesse, pode ainda alterar a ordem de aplicação das otimizações, conforme é possível visualizar na Figura 26. O resultado gerado caso essa alteração fosse realizada pode ser visualizado na Figura 27.

Figura 26 – Interface do ambiente de aprendizado desenvolvido, evidenciando a seleção de otimizações em outra ordem

Fonte: O Autor (2024).

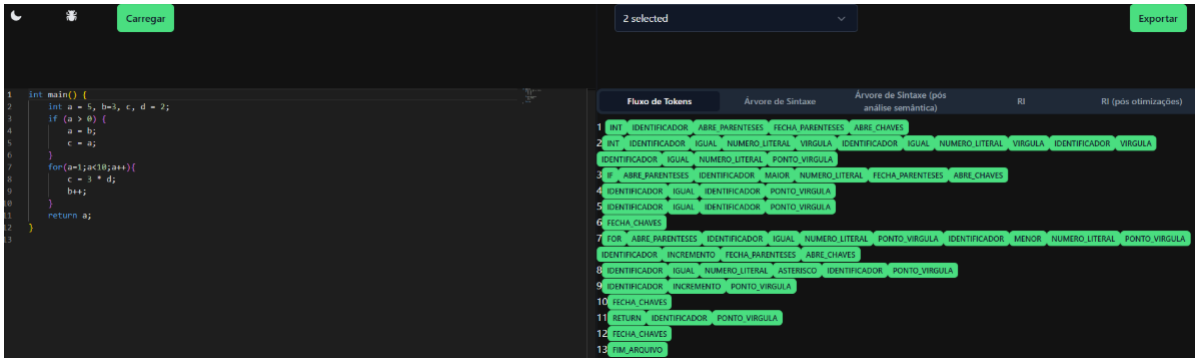
Figura 27 – Interface do ambiente de aprendizado desenvolvido, evidenciando a RI otimizada gerada com outra ordem de otimizações



Fonte: O Autor (2024).

Um detalhe importante de ser mencionado é que também existe, dentro do ambiente desenvolvido, a possibilidade de alteração de tema entre claro e escuro. Todas as imagens apresentadas até então estavam no tema claro, mas um exemplo de como a aplicação seria no tema escuro pode ser visualizada na Figura 28.

Figura 28 – Interface do ambiente de aprendizado desenvolvido no tema escuro



Fonte: O Autor (2024).

5.2.7 Bibliotecas Utilizadas

Mediante obstáculos encontrados durante o processo de desenvolvimento, identificou-se a necessidade de utilização de algumas bibliotecas para auxiliar e agilizar o desenvolvimento do projeto proposto. Esta seção trata de abordar as principais bibliotecas utilizadas e descrever brevemente as principais funcionalidades e objetivos de uso de cada uma.

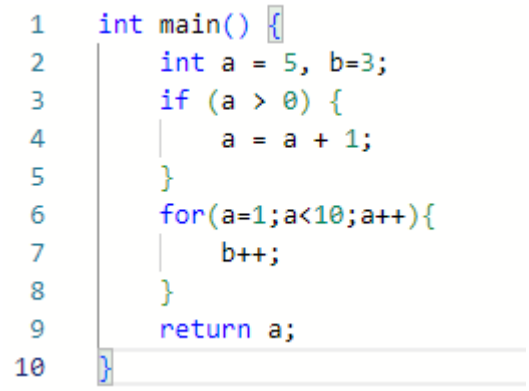
5.2.7.1 *Nuxt Monaco Editor*

A partir da necessidade de utilização de um editor de código dentro do ambiente, optou-se pela utilização da biblioteca *Nuxt Monaco Editor*[1]. Esta biblioteca permite que sejam configurados diversos parâmetros de utilização, destacando-se como principal a configuração de definição de linguagem de programação utilizada para aplicar a estilização de código.

Conforme possível visualizar na Figura 29, a biblioteca fica responsável por destacar o código em cores diferentes de acordo com a linguagem definida na configuração do editor. Neste caso, utilizou-se a linguagem de programação C. Além disso, a biblioteca preencche a numeração de linhas automaticamente.

[1] https://e-chan1007.github.io/nuxt-monaco-editor/

Figura 29 – Exemplo de utilização do *Nuxt Monaco Editor*



Fonte: O Autor (2024).

5.2.7.2 Mermaid

Para a geração de recursos gráficos, com destaque especial na criação da Árvore de Sintaxe (conforme visível na Figura 22), utilizou-se a biblioteca *Mermaid*[2]. Nesta primeira versão do ambiente desenvolvido, optou-se pela utilização exclusivamente das funcionalidades de geração de gráficos no formato *Flowchart*, conforme especificado pela documentação oficial da biblioteca (MERMAID, 2024a).

Vale ressaltar que a biblioteca apresenta diversos outros formatos de geração de recursos gráficos que poderão ser utilizados em futuras adições ao ambiente desenvolvido, como por exemplo os diagramas conhecidos internamente na biblioteca como *State Diagrams* (MERMAID, 2024b), que podem ser utilizados para a geração de gráficos como um Grafo Acíclico Dirigido (GAD) para realizar a representação de blocos básicos no processo de conversão de código.

5.3 VALIDAÇÃO DO DESENVOLVIMENTO

Com o objetivo de validar o que foi desenvolvido, o ambiente foi apresentado à turma atual de compiladores em dois momentos distintos: primeiro em uma versão preliminar e, posteriormente, na primeira versão finalizada do ambiente. Em ambas as apresentações, foi solicitado aos alunos que avaliassem o ambiente de aprendizado e o *hub* por meio de pesquisas. A coleta de opiniões dos alunos permitiu identificar possíveis melhorias e ajustes, bem como validar o alinhamento das funcionalidades com os objetivos pedagógicos propostos.

As perguntas aplicadas na turma atual de compiladores buscavam medir a percepção dos alunos em relação a três diferentes itens, sempre solicitando respostas numéricas entre 1 (pior resultado possível) e 10 (melhor resultado possível). No quesito organização e estilização do conteúdo apresentado, as respostas obtiveram uma média de 9,8. Referente a quantidade e qualidade das funcionalidades apresentadas, 100% das respostas foram nota máxima (10). No

[2] https://mermaid.js.org/

que se refere à percepção dos alunos sobre a possibilidade do ambiente ajudar no processo de aprendizagem da disciplina de compiladores, destaca-se que mais uma vez 100% das respostas foram 10.

Com base nesses valores, afirma-se que os resultados das pesquisas foram amplamente positivos. Todos os alunos indicaram que o ambiente e o *hub* desenvolvidos têm grande potencial para auxiliar no processo de ensino das otimizações de RI e no processo de localização de ambientes de aprendizado, respectivamente. A possibilidade de visualizar o processo de otimização foi considerada como um diferencial significativo em relação aos métodos tradicionais de ensino, destacando a utilidade do ambiente para tornar o aprendizado mais prático e visual. Esses *feedbacks* fornecem uma base sólida para o aprimoramento contínuo do projeto, alinhando-o cada vez mais às necessidades de ensino e aprendizado em disciplinas de compiladores.

6 CONSIDERAÇÕES FINAIS

A abordagem exclusivamente teórica de algumas etapas realizadas pelo compilador acaba por dificultar a visualização prática do conteúdo, além de tornar mais difícil o processo de aprendizagem dos alunos da disciplina de compiladores na UCS. Tendo essa problemática em mente, observou-se que um dos possíveis pontos de melhoria na disciplina em questão refere-se ao ensino das otimizações de código que são realizadas sobre a RI.

Diversos ambientes foram implementados visando tornar mais visual e prático o processo de aprendizagem de compiladores. Porém, mesmo considerando que os ambientes estudados apresentaram diversas funcionalidades interessantes para o processo de aprendizagem dos alunos, destacou-se que tais ambientes possuem ausência de recursos destinados ao aprendizado de otimizações realizadas sobre a RI.

A partir das observações acima, foi desenvolvido um ambiente de aprendizado voltado para a visualização de código intermediário e suas otimizações, com o objetivo de facilitar a compreensão dos processos de conversão de código em compiladores.

Adicionalmente, foi desenvolvido um *hub* de aplicações, visando facilitar o processo de localização de ambientes de aprendizado, na mesma medida que procura instigar os alunos a realizar o desenvolvimento de novas aplicações para serem disponibilizadas no *hub*.

Ao longo do desenvolvimento do presente trabalho, foi possível alcançar resultados promissores, indicando que o ambiente desenvolvido é eficaz para auxiliar estudantes e desenvolvedores a entenderem melhor os passos intermediários do processo de compilação.

Este trabalho representa uma base sólida para futuras expansões e aprimoramentos, como por exemplo a expansão para diferentes linguagens de programação. Dessa forma, o ambiente poderá se tornar uma ferramenta ainda mais completa e útil para o estudo e ensino de compiladores, facilitando a compreensão dos conceitos de código intermediário e otimizações para um público ainda mais amplo.

6.1 OBJETIVOS ATINGIDOS

O desenvolvimento do projeto resultou em um HUB de aplicações completamente implementado, além de uma primeira versão do ambiente pronta para uso. O ambiente permite que os alunos acompanhem o processo de conversão de código C até a RI gerada (em um escopo reduzido, conforme previsto no Capítulo 4), facilitando a análise e compreensão dos processos de compilação.

Adicionalmente, foram implementadas três otimizações de RI (conforme descrito na Seção 5.2.4), permitindo que o ambiente suporte simulações reais de otimização de código,

contribuindo para a compreensão de como tais processos impactam o desempenho e a eficiência dos programas.

O ambiente de aprendizado foi estruturado de forma a ser extensível, permitindo futuras melhorias e implementações adicionais que poderão englobar mais otimizações e maior cobertura de código C. Essas características tornam o projeto promissor para a introdução de novos algoritmos de otimização, possibilitando, no futuro, a comparação de abordagens alternativas e o aprofundamento dos alunos em técnicas de otimização de RI.

6.2 OBJETIVOS NÃO ATINGIDOS

Apesar dos bons resultados obtidos, alguns aspectos ainda não foram completamente explorados ou otimizados, o que limita parcialmente o potencial do ambiente.

Estava previsto o destaque de forma relacionada no ambiente de aprendizado (conforme descrito na Seção 4.3.2), funcionalidade que não pôde ser implementada devido à limitação de tempo e sua complexidade. Esse recurso permitiria uma visualização mais interativa das conexões entre diferentes partes do código com os resultados gerados, facilitando a análise pelos alunos.

Além disso, a implementação de todas as otimizações propostas inicialmente não foi concluída. Embora o ambiente já suporte três otimizações, a otimização de Identificação e remoção de variáveis de indução, que havia tido seu desenvolvimento planejado, teve que ser adiada, deixando espaço para melhorias futuras.

6.3 TRABALHOS FUTUROS

Para aprimorar a experiência de aprendizado e expandir o entendimento sobre a estrutura interna dos programas, futuras implementações do ambiente poderiam incluir a visualização da geração de blocos básicos. Essa funcionalidade possibilitaria que os alunos acompanhassem em tempo real como o compilador organiza o fluxo do código em blocos lógicos, facilitando a compreensão das operações e transições entre as partes do programa.

Outra expansão relevante seria a implementação de uma visualização do GFC correspondente, permitindo uma análise mais detalhada das relações de dependência e das interações de controle entre os blocos.

Além dessas adições, uma possível visualização de diagramas, como a árvore de dominação, traria um recurso importante para o estudo da hierarquia de controle no código. Esse diagrama permitiria que os alunos visualizassem diretamente as relações de dominância entre blocos, auxiliando no entendimento de conceitos avançados de otimização e análise de fluxo. Também foram identificadas algumas melhorias de layout, que, se implementadas, tornarão o

ambiente mais intuitivo e acessível, aprimorando a experiência do usuário e facilitando a navegação entre as diferentes visualizações

REFERÊNCIAS

AHO, A. V. Teaching the compilers course. **ACM SIGCSE Bulletin**, ACM New York, NY, USA, v. 40, n. 4, p. 6–8, 2008.

AHO, A. V. *et al.* **Compilers Principles, Techniques & Tools**. [S.l.]: pearson Education, 2007.

CHOW, F. Intermediate representation. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 56, n. 12, p. 57–62, dec 2013. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/2534706.2534720>.

LOUDEN, K. C. **Compiladores-Princípios e Práticas**. [S.l.]: Cengage Learning Editores, 2004.

MACRAE, C. **Vue. js: up and running: building accessible and performant web apps**. [S.l.]: "O'Reilly Media, Inc.", 2018.

MERMAID. ***Flowcharts - Basic Syntax***. 2024. <https://mermaid.js.org/syntax/flowchart.html>. Acessado em 29 set. 2024.

____. ***State diagrams***. 2024. <https://mermaid.js.org/syntax/stateDiagram.html>. Acessado em 29 set. 2024.

MERNIK, M.; ZUMER, V. An educational tool for teaching compiler construction. **IEEE Transactions on Education**, v. 46, n. 1, p. 61–68, 2003.

MOGENSEN, T. Æ. **Introduction to compiler design**. [S.l.]: Springer Nature, 2024.

Nuxt. ***Introduction***. 2024. <https://nuxt.com/docs/getting-started/introduction>. Acessado em 25 mai. 2024.

ROCHA, R. C. de O. Online iterative compilation guided by work-based profiling. 2017.

SANGAL, S.; KATARIA, S.; TYAGI, T. Pavt: a tool to visualize and teach parsing algorithms. **Education and Information Technologies**, v. 23, n. 6, p. 2737–2764, 2018.

STAMENKOVIć, S.; JOVANOVIć, N. A web-based educational system for teaching compilers. **IEEE Transactions on Learning Technologies**, v. 17, p. 143–156, 2024.

SU, Y.; YAN, S. Y. **Principles of Compilers**. [S.l.]: Springer, 2011.

THAIN, D. **Introduction to compilers and language design**. [S.l.]: Lulu. com, 2016.

Vue. ***Introduction***. 2024. <https://vuejs.org/guide/introduction.html>. Acessado em 23 jul. 2024.