SOPHIE NASCIMENTO

YAMG - *YET ANOTHER MACHINE CODE GENERATOR*: CONSTRUÇÃO DE UM GERADOR DE CÓDIGO MULTIPROPÓSITO

LONDRINA
2023

SOPHIE NASCIMENTO

YAMG - *YET ANOTHER MACHINE CODE GENERATOR*: CONSTRUÇÃO DE UM GERADOR DE CÓDIGO MULTIPROPÓSITO

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação do Depertamento de Computação da Universidade Estadual de Londrina, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Wesley Attrot

LONDRINA
2023

Ficha de identificação da obra elaborada pelo autor, através do Programa de Geração
Automática do Sistema de Bibliotecas da UEL

N244y Nascimento, Sophie.
YAMG - Yet Another Machine code Generator: Construção de um Gerador de Código Multipropósito / Sophie Nascimento. - Londrina, 2023.
117 f. : il.

Orientador: Wesley Attrot.
Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) - Universidade Estadual de Londrina, Centro de Ciências Exatas, Graduação em Ciência da Computação, 2023.
Inclui bibliografia.

1. Geração de Código - TCC. 2. Compiladores - TCC. I. Attrot, Wesley. II. Universidade Estadual de Londrina. Centro de Ciências Exatas. Graduação em Ciência da Computação. III. Título.

CDU 519

SOPHIE NASCIMENTO

YAMG - *YET ANOTHER MACHINE CODE GENERATOR*: CONSTRUÇÃO DE UM GERADOR DE CÓDIGO MULTIPROPÓSITO

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação do Depertamento de Computação da Universidade Estadual de Londrina, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA

Orientador: Wesley Attrot
Universidade Estadual de Londrina

Prof. Dr. Segundo Membro da Banca
Universidade/Instituição do Segundo Membro da Banca – Sigla instituição

---

Prof. Dr. Terceiro Membro da Banca
Universidade/Instituição do Terceiro Membro da Banca – Sigla instituição

---

Prof. Ms. Quarto Membro da Banca
Universidade/Instituição do Quarto Membro da Banca – Sigla instituição

Londrina, 03 de maio de 2023.

AGRADECIMENTOS

Agradeço ao Prof Drº Wesley Attrot por toda a ajdua e orientação durante esse trabalho, que foram essenciais para sua conclusão.

À meu namorado, Iury Pereira de Souza, que esteve ao meu lado durante tantos momentos de dificuldade nesse trabalho, e também sempre esteve comigo durante minha jornada.

Aos meus pais, Angela Maria Gallo e Marcos Roberto do Nascimento, que sempre me deram apoio incondicional para seguir meus sonhos, e respeitaram meu tempo e todas as minhas escolhas.

NASCIMENTO, S.. **YAMG - *Yet Another Machine code Generator*: Construção de um Gerador de Código Multipropósito**. 2023. 131f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2023.

RESUMO

Este trabalho propõe a criação de um novo programa para geração de código, capaz de executar essa tarefa a partir de três algoritmos diferentes, à escolha do usuário dependendo de sua necessidade. Além disso, encontra-se no programa um simples alocador de registradores opcional, para que essa etapa não precise ser feita posteriormente. Com a evolução das linguagens de programação e paradigmas de computação, faz-se bem-vinda uma nova ferramenta, que se adéque aos padrões modernos, além de integrar vários métodos de geração em um único programa.

Palavras-chave: Gerador de Código. Compiladores.

NASCIMENTO, S.. **YAMG - Yet Another Machine code Generator: Construction of a Multipurpose Code Generator**. 2023. 131p. Final Project (Bachelor of Science in Computer Science) – State University of Londrina, Londrina, 2023.

ABSTRACT

This work proposes the creation of a new code generation tool, capable of doing so with three different algorithms, according to the developer's needs. Besides that, this tool contains an optional and simple register allocator, so there's no need worry about this task afterwards. With the evolution of computing paradigms and programming languages, a new system that conforms to modern standards is a welcome addition, aside from integrating various methods of code generation in a single program.

Keywords: Code Generator. Compilers.

LISTA DE ILUSTRAÇÕES

Figura 1 – *Front-end* e *back-end* e suas etapas 19
Figura 2 – Árvore e notação para instruções ADD e LOAD 25
Figura 3 – Árvore de Entrada 25
Figura 4 – Autômato de duas instruções 26
Figura 5 – Exemplo de notação préfixa 28
Figura 6 – Exemplos de diferentes regras de cadeia. Os números perto de cada nó indicam quais regras foram aplicadas 31
Figura 7 – Dois possíveis ladrilhamentos de uma árvore. Exemplo retirado de [1] . 32
Figura 8 – Árvore Exemplo 33
Figura 9 – Árvore Ladrilhada por Maximal Munch 34
Figura 10 – Ladrilhamento por Minimal Munch 37
Figura 11 – Casamentos dos Nós 39
Figura 12 – Instruções Geradas Junto de Suas Regras e Custos 40
Figura 13 – Ladrilhamento por Programação Dinâmica 40
Figura 14 – Ladrilhamento por Programação Dinâmica com os Custos Alterados . . 42
Figura 15 – Exemplo de lista com todos os *live ranges* de um programa 61
Figura 16 – Estrutura do Código de Entrada 67
Figura 17 – Exemplo de Definições de Regra com Custos e Ações 70
Figura 18 – Árvores Equivalentes às Regras de Exemplo 70
Figura 19 – Árvore mostrando a dependência entre operações 71
Figura 20 – Autômato da Regra `ADD(reg,CONST)` 73
Figura 21 – Autômato com a Regra `ADD(reg,reg)` Adicionada 73
Figura 22 – Autômato de Reconhecimento de Três Regras `ADD` 75
Figura 23 – Criação de uma Instância da Classe `Instruction` 78
Figura 24 – Regra que Utiliza o Alocador de Registradores 79
Figura 25 – Código de Entrada no Iburg 85
Figura 26 – Código Gerado pelo Iburg 86
Figura 27 – Parte do Arquivo de Entrada do Olive 88
Figura 28 – Função *burm_label* Gerada pelo Olive 89
Figura 29 – Arquivo de Entrada do YAMG 91
Figura 30 – *Header* gerado pelo YAMG 92
Figura 31 – Código MIPS Gerado no YAMG Pelos Diferentes Algoritmos 93
Figura 32 – Código C para Calcular Fatorial 94
Figura 33 – Código MIPS para Calcular Fatorial, Gerado Automaticamente pelo YAMG 95

LISTA DE TABELAS

Tabela 1 – Casamento de padrão da árvore exemplo 26
Tabela 2 – Padrões de Árvore 33
Tabela 3 – Pilha de Derivação Maximal Munch 34
Tabela 4 – Pilha de Derivação Minimal Munch 36
Tabela 5 – Padrões de Árvores com Custos 41
Tabela 6 – Comparação entre os programas estudados 55
Tabela 7 – Comparação entre os programas 65

SUMÁRIO

1 INTRODUÇÃO . . . 19
1.1 Objetivos . . . 20

2 GERAÇÃO DE CÓDIGO . . . 23
2.1 Casamento de Padrões . . . 24
2.2 Métodos de Análise . . . 27
2.2.1 Análise *Bottom-Up* . . . 27
2.2.2 Análise *Top-Down* . . . 28
2.2.3 Análise e Seleção em Tempo Linear . . . 29
2.3 Maximal Munch . . . 31
2.4 Minimal Munch . . . 35
2.5 Programação Dinâmica . . . 38

3 GERADORES DE GERADORES DE CÓDIGO . . . 43

3.1 Burg . . . 43
3.1.1 Geração do Seletor de Instruções . . . 43
3.1.2 Arquivo de Saída . . . 44
3.1.3 BURS . . . 45
3.2 Iburg . . . 46
3.2.1 Casamento de Padrões e Implementação . . . 46
3.2.2 Otimizações . . . 48
3.3 Olive . . . 50
3.3.1 Melhorias . . . 51
3.3.2 Especificação da Linguagem . . . 51
3.3.3 Gramática . . . 51
3.3.4 Custos . . . 52
3.3.5 Ações . . . 52
3.3.6 Declarações . . . 53
3.3.7 Interfaceamento . . . 53
3.4 Diferenças Principais Entre as Ferramentas Apresentadas . . . 54
3.4.1 Principais Características . . . 54
3.4.2 Metodologia . . . 54
3.4.3 Pontos Fracos . . . 55

4 ALOCAÇÃO DE REGISTRADORES . . . 57

4.1 **Métodos de Alocação** **57**
4.2 ***Linear Scan*** **58**
4.2.1 Metodologia 58
4.2.2 Demonstração Prática 60

5 **YAMG** **63**
5.1 **Dois Programas** **63**

6 **YAMG: DEFINIÇÃO DA DESCRIÇÃO DE MÁQUINA** . . . **67**
6.1 **Linguagem do YAMG** **67**
6.1.1 Definições do Casador de Padrões 67
6.1.2 Regras 69
6.1.3 Definições do Alocador de Registradores 70
6.1.4 Código Auxiliar C++ 71
6.2 **YAMG: Algoritmo de Casamento de Padrões** **71**
6.3 **Interface do Programa** **72**
6.4 **Análise do Código de Entrada** **73**

7 **YAMG: ALOCADOR DE REGISTRADORES** **77**
7.1 **Representações das Instruções** **77**
7.1.1 Instruções 77
7.1.2 Interface do Alocador 78

8 **RESULTADOS EXPERIMENTAIS** **81**

8.1 **Burg** **81**
8.2 **Iburg** **84**
8.3 **Olive** **87**
8.4 **YAMG** **90**

9 **CONCLUSÃO** **97**
9.1 **YAMG** **97**
9.2 **Trabalhos Futuros** **98**

REFERÊNCIAS **99**

APÊNDICES **101**

APÊNDICE A – EXEMPLO DE ARQUIVO DE ENTRADA PARA O YAMG **103**

APÊNDICE B – EXEMPLO MIPS - INVERSÃO DE VETOR 107

APÊNDICE C – EXEMPLO MIPS - MULTIPLICAÇÃO DE VETOR 111

APÊNDICE D – EXEMPLO MIPS - FIBONACCI 113

APÊNDICE E – BACK-END PARA GERAÇÃO DE CÓDIGO MIPS 117

ANEXOS 127

ANEXO A – REPOSITÓRIO DO ANALISADOR SEMÂNTICO 129

ANEXO B – REPOSITÓRIO DO YAMG NO GITHUB . . . 131

1 INTRODUÇÃO

Ao falar de compiladores e geradores de código, uma distinção útil que pode ser feita é a de *front-end* e *back-end* [2]. O *front-end* pode ser definido como a parte do compilador que analisa e transforma o código de entrada em uma representação intermediária, enquanto o *back-end* é aquele que traduz essa representação para a linguagem da máquina alvo. Em um compilador que aceita código C e gera código assembly x86, por exemplo, o *front-end* é responsável por processar o programa em C, e o *back-end* por gerar o x86.

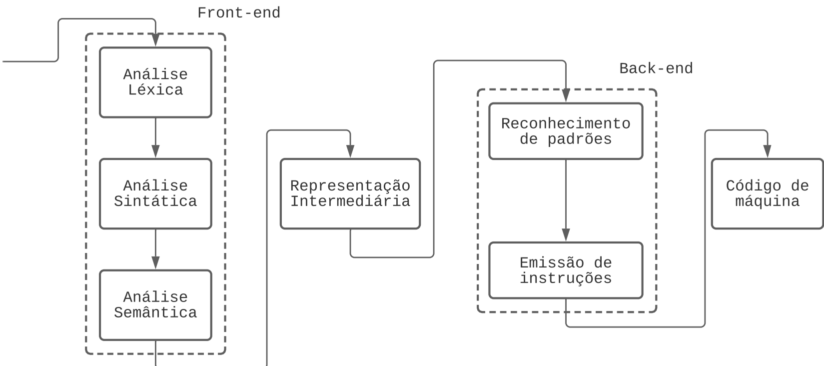

Figura 1 – *Front-end* e *back-end* e suas etapas

Essa distinção é importante porque este trabalho tratará exclusivamente do *back-end*. Assume-se que as análises léxica, sintática, semântica, e possíveis otimizações já foram feitas sobre o código de entrada, e o *front-end* gerou a Representação Intermediária (RI) que será usada no gerador. Uma Representação Intermediária consiste, usualmente, em uma árvore, onde cada nó desta árvore representa uma constante, um registrador da máquina, uma operação, ou um acesso à memória. Muitas máquinas são capazes de executar mais de uma operação por instrução, por exemplo ao invés de realizar uma adição e depois um acesso à memória com esse resultado, com uma única instrução é possível pegar os valores que seriam somados e indexá-los diretamente, tornando o processo mais eficiente.

Com várias instruções que realizam funções similares porém com diferentes eficácias, o problema deixa de ser somente uma maneira de como traduzir um código de alto nível para linguagem de máquina, e se torna como fazer isso da maneira mais eficiente.

A busca por eficiência data desde o próprio problema de tradução em si, ambos vindo com o advento das linguagens de alto nível e a necessidade de transformá-las em código de máquina. No decorrer da história da computação, foram propostos vários métodos para realizar esse objetivo [3] [4], e é uma área que, embora tenha uma sólida fundação, continua a ver novas tecnologias e métodos.

1.1 Objetivos

Diferentes arquiteturas de conjuntos de instruções, como x86 e ARM, não são compatíveis entre si, e até mesmo diferentes versões de uma mesma arquitetura possuem novas instruções que não poderiam ser executadas em versões anteriores. Portanto, para que um compilador possa gerar código ótimo para novas arquiteturas, ou para extensões de uma arquitetura já existente, é necessário que o gerador de código seja adaptado para as mudanças.

Para exemplificar, a empresa estadunidense *Apple*, em 2020, lançou um novo processador, o M1. Deixando de lado a arquitetura x86, que os dispositivos utilizaram por anos, o chip é baseado na ARM, mudança essa que faz com que geradores de código projetados para os *macintosh* precisem agora gerar para esse novo conjunto.

A corporação *Samsung* lançou, em janeiro de 2021, o modelo 2100 de sua família de processadores *Exynos*, que utilizava a versão 8.2-A da arquitetura ARM. No mesmo mês do ano seguinte, veio o modelo 2200, da mesma família, desta vez trazendo a ARMv9-A.

Dentre várias outras mudanças, a extensão ARMv9-A possui instruções para geração de números aleatórios e também identificação de *branch* alvo, que não estavam presentes na versão 8.2-A. Essa, por sua vez, trazia suporte para endereços de 52 bits, cuja extensões anteriores não continham.

Embora não seja necessário dar suporte para novas versões de uma arquitetura, uma vez que o código gerado poderá ser executado sem as novas instruções, o programa não poderá tomar vantagem das mesmas quando elas estiverem disponíveis. Por essa razão, é importante que geradores de código estejam sempre atualizados, para que os fontes gerados possam ser os mais eficientes possíveis.

Tanto a tarefa de criar um gerador para uma nova arquitetura, quanto a de atualizá-lo para uma nova versão, são tarefas que poderiam ser facilitadas através da automação de vários passos do processo. Sabendo-se somente a descrição da máquina alvo, tal como as regras de tradução, é possível criar um gerador de código para essa máquina. Portanto, é possível criar um programa que receba como entrada somente essas especificações, e a partir delas, criar um gerador que siga as especificações, assim diminuindo a carga de trabalho do programador.

Este trabalho busca criar uma ferramenta capaz de gerar código automaticamente através das especificações previamente mencionadas. Muitos dos programas já existentes nessa área são escritos na linguagem C, porém a ferramenta aqui proposta será criada utilizando a linguagem C++, para poder se favorecer de paradigmas mais modernos de programação sem perda de eficiência do programa.

Ademais, existem vários métodos para geração de código, três dos quais serão

explicados detalhadamente no Capítulo 2. Outras ferramentas focam em apenas um destes métodos, mesmo que hajam vantagens em cada um deles. Assim sendo, este trabalho almeja criar um gerador de código que seja capaz de utilizar as três técnicas, para que o programador possa escolher a melhor opção para seu caso específico.

O trabalho se encontra na seguinte disposição: o Capítulo 2 é uma extensiva apresentação de técnicas para geração de código, tal como uma evolução cronológica das mesmas. Algumas ferramentas propostas para a geração ótima de código, tal como suas diferenças e deficiências, são mostradas no Capítulo 3. No capítulo 4 encontra-se uma explicação sobre o processo de alocação de registradores, tal como o detalhamento de uma técnica de alocação. A descrição da ferramenta aqui proposta, chamada YAMG, encontra-se no Capítulo 5, seguido da descrição de seu arquivo de entrada no Capítulo 6, e a interface do alocador de registradores no Capítulo 7. O Capítulo 8 mostra os resultados experimentais e compara o código gerado por algumas ferramentas com o código do YAMG. Por fim, no Capítulo 9 encontram-se as considerações finais do trabalho, tal como planos futuros.

2 GERAÇÃO DE CÓDIGO

Durante a introdução, foi feita uma distinção entre *front-end* e *back-end*. Embora esse trabalho lide exclusivamente com o *back-end*, uma visão geral do compilador é necessária, para poder melhor entender seu funcionamento e as limitações da etapa aqui tratada.

Inicialmente, o compilador recebe como entrada arquivos de código fonte, e realiza uma análise léxica sobre eles. Essa análise verifica cada caractere desses arquivos, e quando um padrão é reconhecido, como palavras chave ("if", "else", "while"), caracteres especias ("{", "}", "->", "%"), ou identificadores ("variavel_1", "soma"), um token correspondente é retornado. Se por exemplo esse analisador reconhece, em ordem, os caracteres "e", "l", "s", "e", um valor `ELSE_SYM` poderia ser retornado, enquanto os caracteres "e", "l", " ", "s", "e" poderiam retornar dois tokens `IDENTIFIER_SYM`, em duas chamadas consecutivas. É possível realizar essa etapa manualmente, porém é uma tarefa que se torna mais complexa conforme o tamanho da linguagem a ser analisada. Uma das primeiras ferramentas mais difundidas que automatizavam esse processo foi o Lex (*Lexical Analysis Generator*) [5], capaz de criar uma analisador léxico com apenas um arquivo de especificação dos tokens. Eventualmente surgiram outras ferramentas, de código aberto, que o substituíram, como por exemplo o Flex (*Fast Lexical Analysis Generator*) [6], ANTLR (*ANother Tool for Language Recognition*) [7], dentre outras.

Logo depois, tomando como entrada os tokens recebidos do analisador léxico, é realizada uma análise sintática. Essa fase lida com a sintaxe da linguagem, garantindo que nenhum token esteja em um lugar indevido, como por exemplo um identificador onde um símbolo de operação seria esperado. Aqui são definidas as regras que definem a ordem correta dos tokens, tal como o que uma determinada ordem significa. Para exemplificar, uma regra do tipo `sum: NUMBER PLUS NUMBER` diz que uma sequência de um número ("12", "-3", "0.75"), um sinal de soma ("+"), e outro número representa uma soma. Se o programa recebesse os tokens em alguma outra ordem, como `PLUS NUMBER NUMBER` por exemplo, a análise terminaria com uma falha, já que essa sequência não é esperada. De maneira complementar ao Lex, foi criada também uma ferramenta para realizar a análise sintática, o Yacc (*Yet Another Compiler-Compiler* [8], que era capaz de ler regras como as mostradas acima, e a partir delas criar um autômato capaz de verificar a corretude da entrada. Tal como foi o caso com o Lex, outras ferramentas também sucederam o Yacc [9], como foi o caso do Byacc [8] (*Berkley Yacc*), do GNU Bison [10], e também do próprio ANTLR [7] que realiza tanto a análise léxica quanto sintática.

É importante notar que meramente verificar se uma entrada é válida não é tão útil por si só. Cada uma dessas regras pode possuir código atrelado à elas, de tal forma que,

quando um padrão é encontrado, um pedaço de código seja executado. Isso faz com que seja possível relacionar informações de tokens, e adicionar metadados sobre eles, além de tornar possível a criação de uma AST (*Abstract Syntax Tree*), que é uma representação do programa na forma de uma árvore. Metadados seriam informações sobre um token que não são visíveis ao observar apenas ao token, mas se tornam disponíveis analisando outros tokens aos quais ele se relaciona. Para exemplificar, uma variável `var_1` por si só não diz seu tipo nem seu valor, porém a linha `int var_1 = 37;` revela que ela é um inteiro, e também que possui valor 37. Um exemplo de regra para identificar essa relação poderia ser `declaration: TYPE IDENTIFIER EQUALS EXPRESSION SEMICOLON`, onde `IDENTIFIER` por si só não revela nada sobre seus metadados, porém, ao casar a regra, seria possível atribuir o tipo e valor à ela.

A última tarefa do *front-end* é a análise semântica, que recebe como entrada a AST criada no passo anterior. O objetivo dessa etapa é analisar os metadados de cada expressão para garantir que nenhuma regra seja violada. Essa é a fase que cuida das inconsistências de contexto como operações em tipos inválidos, problemas de escopo, identificadores não declarados ou não iniciados, identificadores sendo usado de maneira errônea como uma variável sendo chamada como função, dentre outras coisas. Como a semântica é algo muito específico de cada linguagem, além de ser possível construir as ASTs de várias maneiras diferentes, criar uma ferramenta geral para automatizar a construção de um analisador semântico seria uma tarefa praticamente impossível, não havendo nenhuma ferramenta conhecida pela autora desse texto. Assim sendo, é necessário que essa etapa seja implementada manualmente, logo após o término da análise sintática, embora também seja possível realizar algumas dessas verificações nas regras da etapa anterior.

Após passar por toda a árvore, analisando os tipos e em busca de erros, caso nenhum seja encontrado, uma Representação Intermediária é gerada. Como as três análises já foram feitas sobre o código de entrada, a RI resultante deve estar sempre correta, de tal maneira que não é necessário realizar verificações extras nos passos seguintes. Quaisquer possíveis otimizações no código são feitas sobre a RI, entre a análise semântica e a geração de código.

2.1 Casamento de Padrões

Para que seja possível transformar uma RI em um código de máquina, é necessário conhecer as instruções da máquina alvo, e também como traduzir um nó da RI para essa instrução. Cada instrução da máquina alvo pode ser representada por uma árvore, e para saber se uma subárvore da representação corresponde à de uma instrução, é necessário que as duas sejam iguais e, se for o caso, a instrução será emitida. Caso não haja nenhuma instrução que corresponda à subárvore da RI, então não é possível gerar um código para aquele nó, e um erro deve ser emitido.

Uma comparação entre duas árvores pode ser feita de diversas maneiras, porém este trabalho considerará autômatos para realizar o casamento, ao invés de uma comparação nó a nó. Cada instrução será representada por um estado final em um autômato, sendo o caminho (ou o subcaminho) até esse estado uma representação do padrão de árvore da instrução em notação préfixa. Uma subárvore da RI percorre esse autômato até que chegue à um estado final e emita uma instrução.

Para um exemplo, considere os padrões de árvore da Figura 2, seguidos por sua notação prefixa. O padrão de cima representa a operação binária ADD entre um registrador e uma constante, enquanto o de baixo representa a operação LOAD, que lê o conteúdo na posição de memória resultante da soma entre a constante e o registrador.

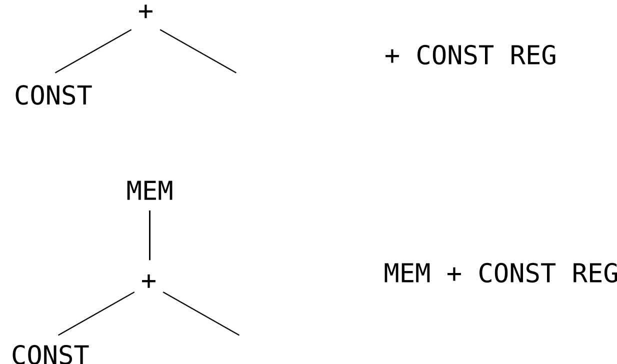

Figura 2 – Árvore e notação para instruções ADD e LOAD

Considere a árvore de entrada da Figura 3 e sua representação em notação prefixa. O casamento de padrões é feito através da leitura de cada nó de entrada e navegando o autômato (Figura 2) de acordo com essa entrada.

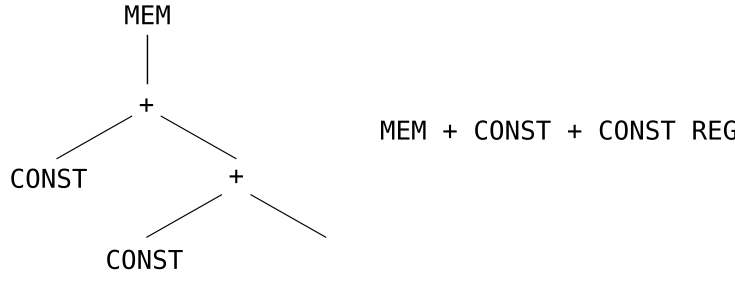

Figura 3 – Árvore de Entrada

Uma instrução somente será emitida quando o autômato chegar em um estado final e houver um *reduce*, onde a cadeia que casa o padrão será consumida. Quando isso acontece uma instrução é gerada, e, possivelmente, um novo token é inserido na pilha. No passo 6 da Tabela 1, é feito um reduce pela regra `[R -> + const reg]`, e portanto a subexpressão da direita (o padrão casado) é substituído pelo símbolo à esquerda, o que permite que a expressão acima seja combinada, e reduzida no passo 8 pela regra `[R -> mem + const + R]`.

Passo	Pilha	Input	Ação
0		mem + const + const reg	Shift(1)
1	mem	+ const + const reg	Shift(3)
2	mem +	const + const reg	Shift(4)
3	mem + const	+ const reg	Shift(6)
4	mem + const +	const reg	Shift(7)
5	mem + const + const	reg	Shift(8)
6	mem + const + const reg		Reduce
7	mem + const	R	Goto(4)
8	mem + const	R	Shift(5)
9	mem + const R		Reduce
10	R		Accept

Tabela 1 – Casamento de padrão da árvore exemplo

As instruções geradas por esse exemplo são, em ordem, `ADD reg2 reg1 const`, onde `reg2` é o registrador destino, e `reg1` e `const` são os valores passados como parâmetro para serem somados, e `LOAD reg3 reg2 const`, onde `reg3` é o registrador de destino, `reg2` é o resultado da instrução `ADD`, e `const` é o outro parâmetro passado para o operador. Em uma linguagem de programação como C, essas instruções poderiam corresponder ao seguinte trecho de código:

`varA[varB + 7]`

Nesse caso, `varB + 7` corresponderiam à instrução `ADD`, seu resultado seria guardado em um registrador, que então seria somado ao registrador base (`varA`, correspondendo à `reg2`) para realizar a leitura de uma parte da memória ou, mais especificamente, a indexação de um vetor.

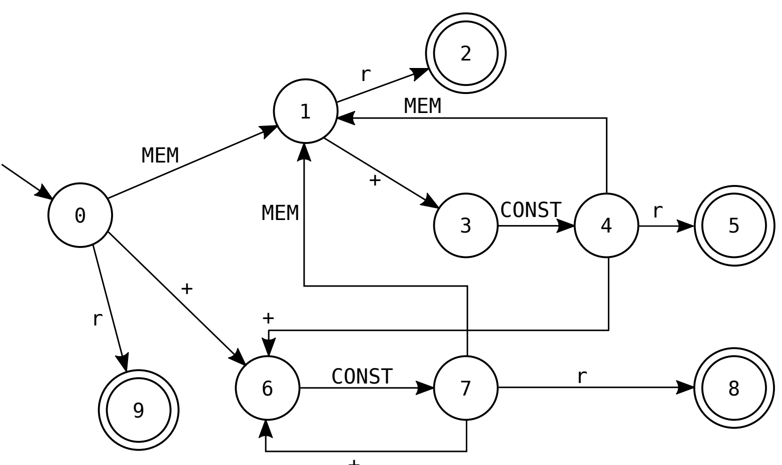

Figura 4 – Autômato de duas instruções

Embora neste exemplo os estados finais não possam executar nenhum *shift*, e possam realizar um *reduce* por apenas uma regra, esse não é sempre o caso. As regras de linguagens de máquinas tendem a ser extensas e inerentemente ambíguas, portanto

existirão situações onde um único estado final pode reduzir por várias regras diferentes, e ainda executar *shifts* para outros estados.

Há três métodos principais para resolução das ambiguidades do tipo *shift-reduce*, e cada uma gera resultados diferentes, portanto é necessário que o programador do gerador escolha com cautela a estratégia utilizada considerando o contexto em que o gerador será utilizado. Este capítulo apresentará cada uma delas, explicará suas diferenças e alguns casos de uso.

2.2 Métodos de Análise

Não só é importante conhecer e definir as técnicas de resolução de conflitos, mas também é vital entender as diferentes técnicas de análise das árvores, afinal elas são a base do casamento de padrões.

2.2.1 Análise *Bottom-Up*

A técnica descrita por Glanville e Graham [4] recebe a descrição da máquina alvo, armazenando detalhes como as regras da máquina, e os padrões correspondentes à essas regras. Essas informações são utilizadas para criar uma tabela de regras e estados que reconhecem essa linguagem, essencialmente um autômato reconhecedor da gramática.

A RI de um programa pode ser descrita por uma gramática livre de contexto, que consiste de um conjunto de símbolos terminais e não-terminais, onde os primeiros fazem parte da linguagem que está sendo reconhecida e serão representados por letras minúsculas, e os últimos são símbolos que produzem outras cadeias de símbolos e serão representados por letras maiúsculas.

Uma produção é do tipo `S → a B c...`, onde o lado esquerdo é sempre um único não-terminal que gera a cadeia do lado direito, composta por quaisquer combinações de símbolos. Embora seja possível representar uma RI de diversas maneiras, a mais conveniente para análise sintática é a notação préfixa, pois esta representação lida com precedência de operadores sem necessidade de parênteses ou quaisquer símbolos extras. A expressão `a * b + c`, por exemplo, seria representada como `+ c * a b`, onde a Figura 5 representa a árvore desta expressão.

As regras e especificações da máquina por si só são suficientes para realizar o *parsing* da árvore, porém não lidam com os problemas de combinação de padrões, o problema de quais conjuntos de padrões podem ser formados, tampouco com a seleção de padrões, ou seja, dos possíveis conjuntos, qual será escolhido.

Para tratar desses problemas é utilizado um *left-to-right, right-most derivation parser*, ou *LR parser*. Esse analisador lê a *string* de entrada, que é uma representação

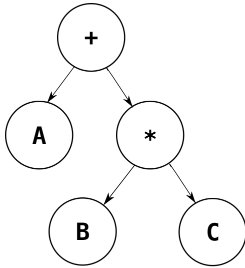

Figura 5 – Exemplo de notação préfixa

em *string* da RI, da esquerda para a direita, e executa as regras de redução conforme os símbolos mais à esquerda. Os símbolos lidos são colocados em uma pilha e o programa realiza um *shift*, ou seja, muda de estado com base no estado atual, e no símbolo lido. Ao alcançar um estado final, isso é, um estado de reconhecimento da entrada, onde os tokens no topo da pilha correspondem exatamente à um determinado padrão da linguagem alvo, o programa consome esses tokens e empilha o não-terminal à esquerda da produção. Isso garante que as subárvores mais abaixo sejam derivadas primeiro, uma vez que elas devem ser emitidas antes das instruções acima, que dependem delas.

Embora o programa gerado seja similar a um analisador sintático, ele não é um reconhecedor puro e simples. Primeiramente, devido a natureza das linguagens de máquina, existem várias maneiras de expressar as diferentes regras, isso faz com que haja várias maneiras de se construir a mesma árvore de derivação para uma única instrução, o que torna a linguagem inerentemente ambígua. Isso implica que é necessário aplicar um método para resolução dos conflitos gerados pelas ambiguidades.

O segundo ponto onde os dois se diferenciam é na etapa de redução. Quando um padrão é reconhecido, este programa tem que escolher entre uma das instruções que podem ser emitidas naquele estado. Essa escolha é feita considerando informações sintáticas e semânticas presentes para o gerador, como por exemplo o valor de uma constante. Após a escolha da instrução, ela é emitida e o gerador continuará a ler o resto da entrada, se houver. Caso não hajam falhas ou *bugs* na etapa de compilação, a RI gerada é sempre correta, assim não é necessário que aconteça validação no gerador. Erros nessa etapa indicam um *bug* com o gerador.

2.2.2 Análise *Top-Down*

O método descrito na seção anterior utiliza como base uma técnica *bottom-up* para análise, começando esse processo sempre pelas folhas, e iterando sobre os nós até a raiz da árvore, ao mesmo tempo que casa e seleciona padrões pelo caminho. A técnica oposta a essa seria a análise *top-down*, onde, em contraste, o seletor de instruções começa da raiz da árvore e desce até as folhas enquanto casa padrões.

Essa alteração na direção da análise traz algumas outras mudanças também, uma

vez que um seletor que começa das folhas pode simplesmente casar um padrão e passar informações para cima, por exemplo, em qual registrador o resultado de uma instrução foi alocado, um analisador *top-down* deve ter essa informação com antecedência e repassá-la para baixo. Esse tipo de *parser* tende a ser recursivo, uma vez que há restrições extras, além de permitir *backtracking*, que pode ser necessário, visto como algumas seleções de padrão podem fazer partes inferiores da árvore não casarem com nenhum padrão.

A técnica exposta por Cattell [11] utiliza este método *top-down*. O programa busca todos os padrões que podem ser casados a partir da raiz da árvore, e então segue de maneira recursiva para as subárvores de cada casamento encontrado, e cada padrão casado é armazenado como um conjunto de *templates* em uma árvore. Esse processo cria várias sequências de *templates* que cobrem a RI.

A partir da descrição de uma linguagem de máquina são gerados esses *templates*, e cada instrução da linguagem é modelada como um conjunto de operações de máquina que descrevem os efeitos de cada instrução, como por exemplo se há efeitos na memória, movimentação entre registradores, dentre outros. Como uma etapa de pré-processamento, o programa realiza uma análise conhecida como *means-end*, onde há um objetivo, e são calculadas as ações necessárias para se chegar nesse objetivo. Cada ação diminui a diferença entre o estado atual e o estado desejado, nesse caso, como a árvore de *templates* é nesse estado, e como ela deveria ser.

Há vantagens e desvantagens no método descrito, quando comparado com técnicas baseadas em análise sintática. Graças à análise *means-end* [11] e a possibilidade de *backtracking*, há um menor risco do seletor ser capaz de gerar código para a RI de entrada. Por outro lado, essas características também tornam o código mais complexo e demorado. Enquanto um gerador LR possui complexidade de tempo linear, esse método possui complexidade quadrática no pior caso.

2.2.3 Análise e Seleção em Tempo Linear

Com avanços nas técnicas de casamento de padrões tornou-se possível encontrar todas as combinações de uma RI em tempo linear. O próximo passo para tornar essas ferramentas mais eficientes foi encontrar uma maneira de realizar a seleção ótima de padrões com essa mesma complexidade de tempo. Para atingir esse objetivo foi utilizada a técnica de programação dinâmica [12].

Embora propostas para tais seletores ótimos tenham sido feitas anteriormente, a primeira implementação prática foi descrita por Aho et al. [13], onde foi usado um algoritmo de três etapas. Enquanto outras ferramentas similares fazem uma análise puramente *bottom-up* ou *top-down*, esse sistema utiliza um misto de ambas as técnicas.

A partir da descrição da máquina, é gerado um programa que passa três vezes pela

entrada dada, começando com uma análise de cada nó da árvore. Essa primeira etapa é feita utilizando uma versão generalizada de um algoritmo para casamento de palavras, previamente proposto também por Aho e Corasick [14], que encontra todos os possíveis padrões a partir de um determinado nó, para cada nó da árvore.

O segundo passo é uma passagem *bottom-up* pela árvore, que computa o custo de cada padrão possível naquele nó. Esse cálculo considera o custo não só da regra de redução do nó em si, mas também o custo de redução dos não-terminais que aparecem na regra. Após calcular todas as regras dos padrões possíveis, as regras de cadeia aplicáveis são checadas, uma vez que elas podem fazer o nó atual ser reduzido para mais não-terminais, ou não-terminais mais baratos. O menor custo para a redução daquele nó é armazenado em um vetor, que é utilizado pela última etapa do algoritmo.

Regras de cadeia são regras que consistem de um único não-terminal, e seu nome vem do fato de que reduções feitas através dessas regras podem ser encadeadas uma atrás da outra. Considere por exemplo a seguinte gramática, e a árvore mostrada na Figura 6.

1	$\mathtt{r_A} \rightarrow \mathtt{INT}$
2	$\mathtt{r_B} \rightarrow \mathtt{r_A}$
3	$\mathtt{r_B} \rightarrow \mathtt{r_B}\ \mathtt{r_A}$

Considerando um analisador que utiliza apenas comparação de nós, ele não seria capaz de casar um padrão para a árvore da Figura6(b), uma vez que não existe nenhuma regra `regA → intA + intB` e o analisador não pode aplicar regras implicitamente. Há duas maneiras razoáveis de se resolver esse problema. A primeira, chamada de *fecho transitivo* [15], é considerar as regras de cadeia durante a etapa de casamento de padrões, essencialmente combinando as regras de cadeia com regras normais, processo que é mostrado na Figura 6(c). Isso permite que regras de cadeia possam ser aplicadas em qualquer combinação de qualquer tamanho, porém complicam a tarefa de casamento e seleção de padrões, uma vez que cada nó é mais complexo.

O segundo método é inserir nós auxiliares na árvore, cada um representando a aplicação de uma dessas regras, o que acaba aumentando o tamanho da árvore, e pode ser visto na Figura 6(d). Embora essa técnica amplie a árvore, ela não requer alterações nos algoritmos de casamento e seleção.

De volta ao gerador de código, após a computação dos custos, o programa realiza uma terceira e última passagem pela árvore. Com o custo da raiz da árvore calculado, é possível encontrar uma cobertura de custo mínimo que reduz a árvore para o não-terminal alvo consultando o vetor de custos. A partir da raiz, seleciona-se a regra de redução mais barata, e o mesmo é feito recursivamente para os nós abaixo que formam a regra selecionada. Essa última passagem também executa as ações associadas a cada padrão selecionado, o que significa emitir o código *assembly* correspondente.

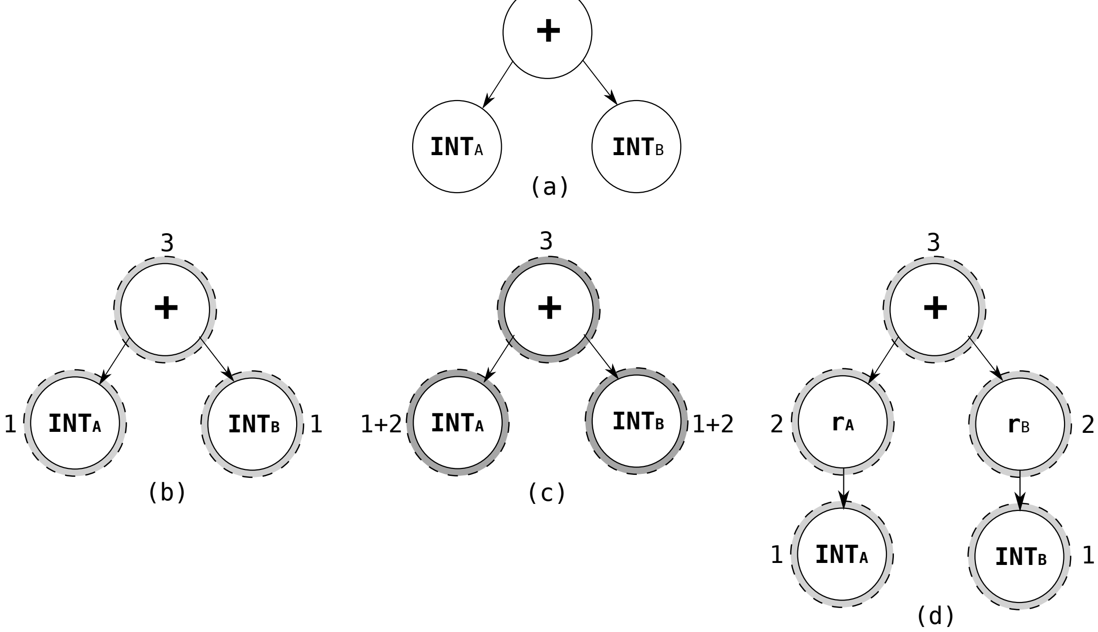

Figura 6 – Exemplos de diferentes regras de cadeia. Os números perto de cada nó indicam quais regras foram aplicadas

2.3 Maximal Munch

Chama-se de *cobertura* ou *ladrilhamento* de uma árvore um casamento entre sub-árvores e padrões de instrução, tal que todos os nós façam partes de uma sub-árvore que foi casada, e todas as sub-árvores correspondam a um padrão.

Uma única árvore pode ser ladrilhada de múltiplas maneiras, de acordo com a arquitetura alvo. A Figura 7 mostra dois possíveis ladrilhamentos para uma mesma árvore. A diferença se encontra nos ladrilhos 8 e 9, onde na Figura 7(a) o ladrilho 8 inclui o nó `MEM`, enquanto na Figura 7(b) este nó está presente no ladrilho 9. Os padrões de instrução para estes ladrilhamentos encontra-se na Tabela 2.

As instruções geradas pelos ladrilhos 8 e 9 também são diferentes. Na Figura 7(a) a instrução gerada em 8 é um `LOAD`, seguido de um `STORE` em 9, enquanto na Figura 7(b) é realizada uma adição seguida de um `MOVEM`. Cada opção possui características diferentes, e mesmo com essa pequena mudança, é possível observar como métricas diferentes funcionariam. Um algoritmo que prioriza menor número de acessos à memória não escolheria a opção da Figura 7(a), por exemplo, pois ela geraria uma instrução LOAD extra.

Um algoritmo de *Maximal Munch* [16] pode ser categorizado como um algoritmo guloso pois seu objetivo é englobar o maior número de nós em um único ladrilho, de tal modo que cada um seja o maior possível. Isso faz com que a árvore tenha grandes ladrilhos, e também a menor quantidade deles.

O objetivo de um algoritmo de *Maximal Munch* é gerar sempre o código com a menor quantidade de instruções possíveis. Usualmente, cada ladrilho corresponde à

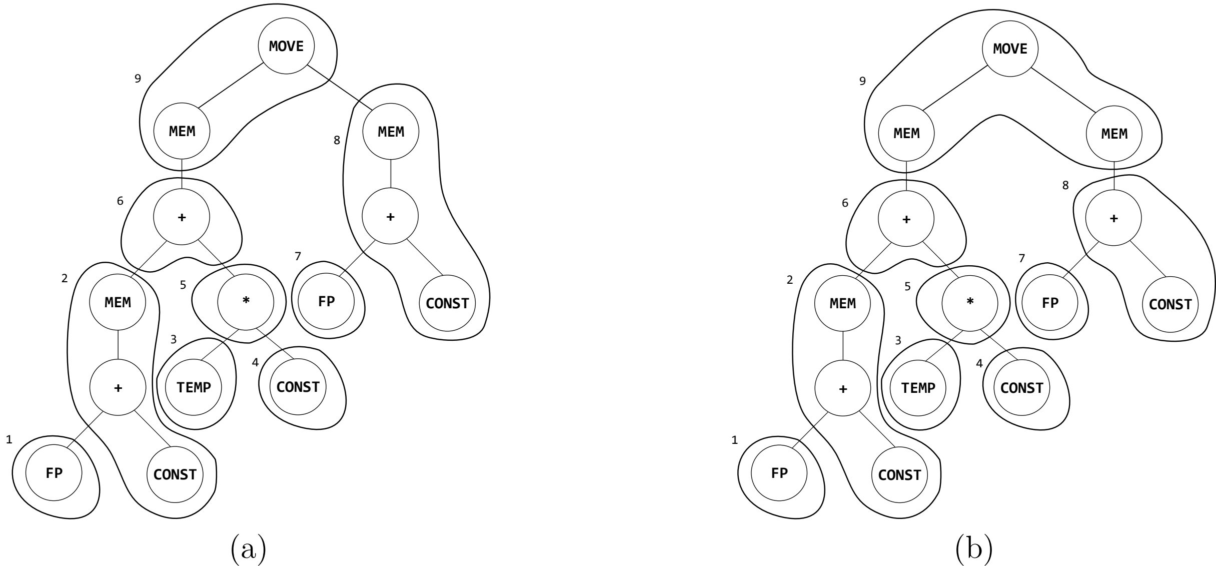

Figura 7 – Dois possíveis ladrilhamentos de uma árvore. Exemplo retirado de [1]

uma única instrução, embora hajam exceções, como o caso de registradores, que não geram instrução alguma (ladrilhos 1, 3, e 7 na Tabela 2). Um código com menos linhas possui vários benefícios, o mais visível é o espaço que o código ocupa no disco e na memória, porém existem vantagens menos óbvias, como o fato de instruções *jump* e *branch* precisarem de código extra para serem utilizadas, caso a *label* alvo esteja demasiado longe. Um código menor pode evitar esse problema, caso seu tamanho final seja menor que o limite da instrução.

Para se obter o menor código, este algoritmo casa sempre os maiores padrões disponíveis. Quando encontra um conflito *shift-reduce*, opta por realizar uma transição de estado, consumindo assim a maior quantidade de nós possíveis antes de criar um ladrilho. Isso permite que a maior parte possível da árvore vire um comando único, reduzindo assim o tamanho do código final ao máximo. Caso o autômato encontre um estado final, ele continuará a leitura se possível, porém caso se depare com um erro à frente, ele retorna ao último estado final encontrado, e reduz por alguma das regras nesse estado.

Na Tabela 2, é possível observar que cada regra possui uma produção (efeito), um nome (que também representa a ação associada à essa regra), e um padrão de árvore. A regra número sete, por exemplo, possui a produção `r ← - r c`, onde a parte esquerda (à esquerda da seta) representa o símbolo que será empilhado quando a parte direita for reconhecida, e a parte direita é o padrão de árvore em notação préfixa à ser casado, e emitirá a instrução `addi r, c`.

Para melhor exemplificar este processo, será utilizado esse algoritmo na árvore da Figura 8. A Tabela 3 contém a pilha de derivação desta árvore sob o método *Maximal Munch*. As ações s denotam uma ação de *shift* para outro estado, enquanto um r_i representa um *reduce* pela regra i, e a ação a serve para dizer para o programa que a geração de

Nº	Nome	Efeito	Produção	Padrões de Árvore
1	-	r_i		TEMP
2	ADD	$r_i \leftarrow r_j + r_k$	r ← + r r	+
3	MUL	$r_i \leftarrow r_j * r_k$	r ← * r r	*
4	SUB	$r_i \leftarrow r_j - r_k$	r ← - r r	-
5	DIV	$r_i \leftarrow r_j / r_k$	r ← / r r	/
6	ADDI	$r_i \leftarrow r_j + c$	r ← + r c r ← + c r r ← c	+ CONST; + CONST; CONST
7	SUBI	$r_i \leftarrow r_j - c$	r ←- r c	- CONST
8	LOAD	$r_i \leftarrow M[r_j + c]$	r ← m + r c r ← m + c r r ← m c r ← m r	MEM + CONST; MEM + CONST; MEM CONST; MEM
9	STORE	$M[r_j + c] \leftarrow r_i$	λ ← mv m + r c r λ ← mv m + c r r λ ← mv m c r λ ← mv m r	MOVE MEM + CONST; MOVE MEM + CONST; MOVE MEM CONST; MOVE MEM
10	MOVEM	$M[r_j] \leftarrow M[r_i]$	λ ← mv m m	MOVE MEM MEM

Tabela 2 – Padrões de Árvore

código foi realizada com sucesso. A Figura 9 representa a árvore completamente ladrilhada por esse algoritmo.

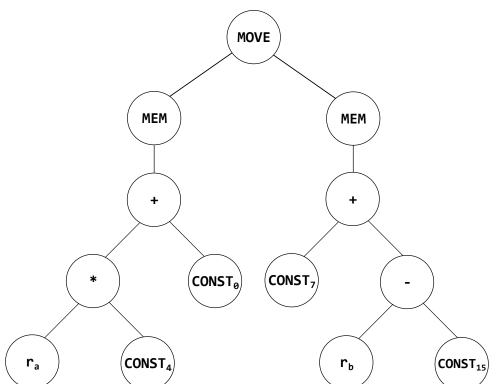

Figura 8 – Árvore Exemplo

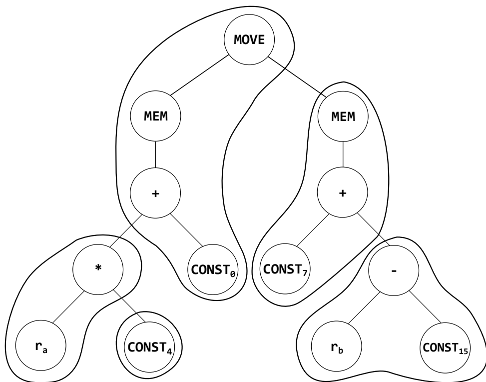

Figura 9 – Árvore Ladrilhada por Maximal Munch

Passo	Pilha	Entrada	Ação
1		mv m + * r_a c_4 c_0 m + c_7 - r_b c_{15}	s
2	mv	m + * r_a c_4 c_0 m + c_7 - r_b c_{15}	s
3	mv m	+ * r_a c_4 c_0 m + c_7 - r_b c_{15}	s
4	mv m +	* r_a c_4 c_0 m + c_7 - r_b c_{15}	s
5	mv m + *	r_a c_4 c_0 m + c_7 - r_b c_{15}	s
6	mv m + * r_a	c_4 c_0 m + c_7 - r_b c_{15}	s
7	mv m + * r_a (c_4)	c_0 m + c_7 - r_b c_{15}	r_6
8	mv m + * r_a	r_1 c_0 m + c_7 - r_b c_{15}	s
9	mv m + * (r_a r_1)	c_0 m + c_7 - r_b c_{15}	r_3
10	mv m +	r_2 c_0 m + c_7 - r_b c_{15}	s
11	mv m + r_2	c_0 m + c_7 - r_b c_{15}	s
12	mv m + r_2 c_0	m + c_7 - r_b c_{15}	s
13	mv m + r_2 c_0 m	+ c_7 - r_b c_{15}	s
14	mv m + r_2 c_0 m +	c_7 - r_b c_{15}	s
15	mv m + r_2 c_0 m + c_7	- r_b c_{15}	s
16	mv m + r_2 c_0 m + c_7 -	r_b c_{15}	s
17	mv m + r_2 c_0 m + c_7 - r_b	c_{15}	s
18	mv m + r_2 c_0 m + c_7 (- r_b c_{15})		r_7
19	mv m + r_2 c_0 m + c_7	r_3	s
20	mv m + r_2 c_0 (m + c_7 r_3)		r_8
21	mv m + r_2 c_0	r_4	s
22	mv m + r_2 c_0 r_4		r_9
23			a

Tabela 3 – Pilha de Derivação Maximal Munch

```
ADDI r1, 6
MUL r2, rA, r1
SUBI r3, rB, 15
ADDI r4, 7, r3
STORE r4, 0(r2)
```

A cada passo do algoritmo, um símbolo é retirado da entrada e colocado na pilha.

Quando não há como fazer um *shift*, os símbolos no topo da pilha que correspondem a um padrão são reduzidos pela regra desse padrão e, quando aplicável, um símbolo é inserido na entrada. No passo nove, por exemplo, o topo da pilha corresponde ao padrão da regra 3, $r_2 \leftarrow * \ r_a \ r_1$, então o lado direito da produção é reduzido para o não-terminal do lado esquerdo que é inserido na entrada para ser empilhado depois, e utilizado em uma próxima redução.

Neste exemplo, é possível observar que o *parsing* da árvore é feito de maneira *top-down*, onde o nó raiz da árvore é colocado primeiro na pilha, seguido do nó esquerdo, e assim recursivamente, até que se encontre uma folha, então o filho direito do nó anterior é empilhado, continuando esse processo para toda a árvore. Esse comportamento descreve um *Left to Right parser*, que passa pela árvore de cima para baixo, e da esquerda para a direita. O Algoritmo 1 demonstra este funcionamento.

Algorithm 1 Algoritmo de Empilhamento

```
empilha(pilha, no)
if no.esquerda then
    empilha_arvore(no.esquerda, pilha)
end if
if no.direita then
    empilha_arvore(no.direita, pilha)
end if
```

É possível observar que, embora o processo se inicie pelo topo da árvore, as primeiras sub-árvores a serem ladrilhadas são as sub-árvores mais profundas, uma vez que o casamento de padrões é feito com os itens mais ao topo da pilha, e também que esses ladrilhos são dependências dos ladrilhos acima. Uma instrução gerada por um nó que não seja uma folha não pode ser executada até que as instruções de seus nós filhos já tenham sido realizadas.

2.4 Minimal Munch

O Minimal Munch [16] é um método trivial para solucionar o problema de casamento de padrões. Muitos nós da RI possuem uma instrução de máquina correspondente, e aqueles que não possuem, como é o caso do nó `MOVE` na Tabela 2, tem apenas dois nós, sendo que seu maior padrão possui quatro.

A solução é chamada trivial, pois utiliza a solução simples para cada nó, que é justamente reduzi-los pelo menor padrão possível, e também porque gera, para muitos casos, um resultado não ótimo. Quando esse algoritmo encontra um conflito *shift-reduce*, ele opta sempre pela redução, reconhecendo já o padrão atual sem sequer considerar cadeias maiores. Esse comportmento tem o efeito de gerar o maior código possível, uma vez que cada nó corresponderá a uma instrução de máquina, ou o mais próximo disso.

Embora isso possa não parecer desejável, uma vez que ocupa mais espaço em memória e em disco, ainda há possíveis aplicações. Compiladores JIT (Just-In-Time) e interpretadores, por exemplo, podem se beneficiar ao pegar a primeira instrução encontrada para executar, ao invés de esperar todo o processo de análise e casamento de uma instrução complexa.

Para facilitar comparações, o exemplo dado nessa seção e na próxima utilizará a mesma árvore e padrões de árvore mostrados na seção anterior, e a pilha de derivação será também no mesmo formato.

Passo	Pilha	Entrada	Ação
1		mv m + * r_a c_4 c_0 m + c_7 - r_b c_{15}	s
2	mv	m + * r_a c_4 c_0 m + c_7 - r_b c_{15}	s
3	mv m	+ * r_a c_4 c_0 m + c_7 - r_b c_{15}	s
4	mv m +	* r_a c_4 c_0 m + c_7 - r_b c_{15}	s
5	mv m + *	r_a c_4 c_0 m + c_7 - r_b c_{15}	s
6	mv m + * r_a	c_4 c_0 m + c_7 - r_b c_{15}	s
7	mv m + * r_a (c_4)	c_0 m + c_7 - r_b c_{15}	r_6
8	mv m + * r_a	r_1 c_0 m + c_7 - r_b c_{15}	s
9	mv m + (* r_a r_1)	c_0 m + c_7 - r_b c_{15}	r_3
10	mv m +	r_2 c_0 m + c_7 - r_b c_{15}	s
11	mv m + r_2	c_0 m + c_7 - r_b c_{15}	s
12	mv m + r_2 (c_0)	m + c_7 - r_b c_{15}	r_6
13	mv m + r_2	r_3 m + c_7 - r_b c_{15}	s
14	mv m (+ r_2 r_3)	m + c_7 - r_b c_{15}	r_2
15	mv m	r_4 m + c_7 - r_b c_{15}	s
16	mv m r_4	m + c_7 - r_b c_{15}	s
17	mv m r_4 m	+ c_7 - r_b c_{15}	s
18	mv m r_4 m +	c_7 - r_b c_{15}	s
19	mv m r_4 m + (c_7)	- r_b c_{15}	r_6
20	mv m r_4 m +	r_5 - r_b c_{15}	s
21	mv m r_4 m + r_5	- r_b c_{15}	s
22	mv m r_4 m + r_5 -	r_b c_{15}	s
23	mv m r_4 m + r_5 - r_b	c_{15}	s
24	mv m r_4 m + r_5 - r_b (c_{15})		r_6
25	mv m r_4 m + r_5 - r_b	r_6	s
26	mv m r_4 m + r_5 (- r_b r_6)		r_4
27	mv m r_4 m + r_5	r_7	s
28	mv m r_4 m (+ r_5 r_7)		r_2
29	mv m r_4 m	r_8	s
30	mv m r_4 (m r_8)		r_8
31	mv m r_4	r_9	s
32	(mv m r_4 r_9)		r_9
33			a

Tabela 4 – Pilha de Derivação Minimal Munch

Uma clara diferença na pilha em relação ao Maximal Munch é que, mesmo para

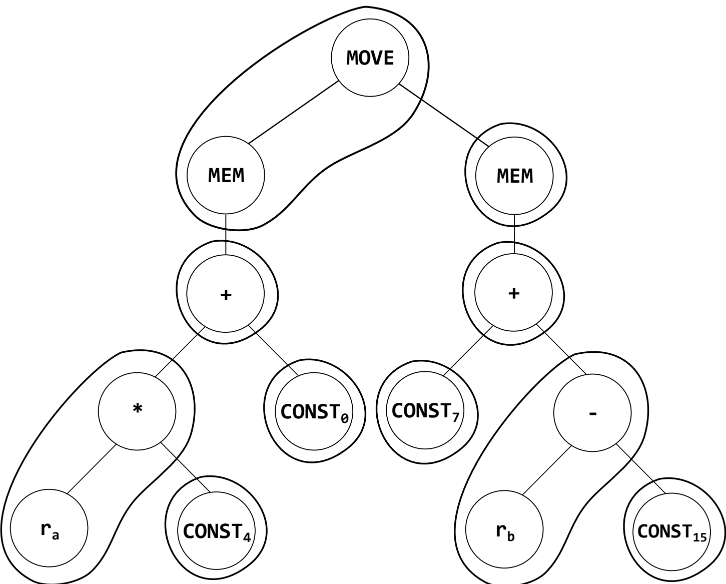

Figura 10 – Ladrilhamento por Minimal Munch

uma árvore pequena, ela é significativamente maior, totalizando dez passos extras. Note também que, pela própria natureza do método, de executar ações de *reduce* sempre que possível, elas são muito mais comuns, e para esse exemplo, são o dobro das reduções geradas pelo método anterior. É simples observar como os padrões casados são pequenos, muitas vezes casando um único nó, ou no máximo dois nessa pequena linguagem de máquina, o que leva à geração de muito mais instruções simples como `ADD`, `ADDI`, `SUB`, dentre outras.

É importante citar que essas técnicas apenas resolvem conflitos *shift-reduce*, enquanto conflitos *reduce-reduce* são tratados de outras maneiras. Uma possibilidade é optar pelo não-determinismo, ou seja, não ter um método de decisão para escolher qual instrução gerar, seja escolhendo ao acaso, ou sempre a primeira da lista de possibilidades, por exemplo. Outra opção é ter as instruções ordenadas em uma lista de "melhor" pra "pior", baseado em alguma especificação ou decisão do programador, e escolher sempre o "melhor" padrão que se encaixa. Ainda outra possibilidade é designar custos para cada instrução, seja ele um custo de ciclos da CPU, o custo de energia daquela instrução, ou alguma outra métrica designada pelo programador. O Minimal Munch, por exemplo, sempre escolhe o menor padrão dentro dos possíveis naquele estado, e caso haja mais de um padrão com menor tamanho, a escolha pode ser feita de alguma das maneiras citadas, seja ela arbitrária, pré-definida, ou qualquer outra.

```
ADDI r1, 4
MUL r2, ra, r1
ADDI r3, 0
ADD r4, r2, r3
```

```
ADDI r5, 7
ADDI r6, 15
SUB r7, rB, r6
ADD r8, r5, r7
LOAD r9, 0(r8)
STORE r4, 0(r9)
```

2.5 Programação Dinâmica

Essa seção entrará em mais detalhes sobre a abordagem da programação dinâmica na geraçao de código, como implementada por Aho et al. [13], e também exemplificará este método, previamente descrito na Seção 2.2.3. Esse algoritmo gera resultados ótimos para arquiteturas com registradores uniformes, ou seja, um conjunto de registradores substituíveis entre si, além de instruções da forma $R_i := E$, onde E é uma expressão qualquer que contenha registradores, operadores, ou acessos à memória.

A ideia de um algoritmo de programação dinâmica é particionar recursivamente o problema em subproblemas menores, encontrando sua solução ótima e utilizando-as para pegar a melhor solução. Dividindo o problema E em E_1 *op* E_2, onde *op* é uma operação qualquer, a melhor maneira de resolver E é avaliar E_1 e E_2 em qualquer ordem, que serão por sua vez avaliados recursivamente da mesma maneira, e então calcular o custo de *op*.

Uma avaliação é dita contígua quando, para uma árvore T que representa um programa, primeiramente são analisadas as subárvores de T que podem ser computadas na memória, e depois avalia o restante de T, calculando as expressões filhas T_1 e T_2 em qualquer ordem e depois a raiz, usando os valores previamente computados na memória quando necessário. Uma avaliação é não-contígua quando o programa calcula T_1 e T_2 de maneira alternada, ou então armazena o valor de uma avaliação em um registrador, ao invés da memória. A propriedade de análise contígua dita que, para toda árvore de expressões T, há um programa ótimo constituído pelas subárvores ótimas de T e uma avaliação da raiz. Justamente essa propriedade que permite o uso de programação dinâmica para a geração de um programa ótimo para T.

A Tabela 14 representa os mesmos padrões da Tabela 2, porém com os custos fictícios de cada padrão adicionados. A mesma árvore dos exemplos anteriores será utilizada aqui para demonstrar o algoritmo.

O primeiro passo é uma passagem *bottom-up* para a computação dos custos. O primeiro nó a ser visitado é r_a, cuja única regra possível é a *1*, portanto a solução ótima para ele tem custo zero. O próximo nó a ser avaliado é o nó $CONST_4$, que só pode ser casado pela regra *8* com custo um, portanto esse é o conjunto regra/valor colocado no vetor de custos. Os próximos nós são, nessa ordem *** e $CONST_0$, que só possuem uma

```
rA        [0|1]
CONST 4   [1|8]
*         [3|3]
CONST 0   [1|8]
+         [4|6, 5|2]
MEM       [5|10, 6|13]
CONST 7   [1|8]
rB        [0|1]
CONST 15  [1|8]
-         [1|9, 2|4]
+         [3|2, 3|7]
MEM       [3|11, 5|13]
MOVE      [8|14, 10|17]
```

Figura 11 – Casamentos dos Nós

regra aplicável cada, *3* e *8* respectivamente, de custos três (dois da operação, zero do operando esquerdo, e um do operando direito) e um.

Um caso mais interessante é o do nó seguinte, de adição, que pode ser casado por duas regras, tanto a *2* quanto a *6*. A regra *2* possui custo um por padrão, somada do custo de r_j que é três como já determinado, e de r_k, custo um, totalizando cinco. Por outro lado, a regra *6* inclui a constante sem precisar armazená-la em registrador antes, portanto seu custo é apenas o da operação, somado ao de r_j, um mais três para um custo total quatro. Os possíveis casamentos de cada nó, tal como seus vetores de custos e regras associadas estão listados na Figura 11. O vetor está em ordem crescente com base nos custos, e cada item está na forma custo|regra. O menor custo no vetor da raiz, nesse caso *MOVE*, é o custo mínimo para avaliar a árvore, ou seja, o menor custo para esse programa é oito pela regra *14*.

Depois dessa primeira etapa, o programa passa mais uma vez pela árvore, utilizando os vetores de custo para determinar quais subárvores serão computadas em memória. Como o vetor está em ordem crescente, as subárvores escolhidas serão sempre aquelas na posição zero do vetor.

Por último, o algoritmo realiza uma passagem *top-down* na árvore do programa para realizar as ações associadas à cada regra selecionada, e o código das subárvores computadas na memória são gerados primeiro.

Para esse exemplo, essa última etapa pegaria a raiz *MOVE*, e executaria recursivamente as ações dos seus nós filhos antes da sua própria. O primeiro a ser executado seria o nó ***, que por sua vez precisaria executar as ações de seus próprios nós filhos, r_a e depois $CONST_4$. A ordem de instruções geradas, tal como o nó que as gerou e a regra estão na Figura 12 (na forma nó|regra), considerando registradores arbitrários.

```
ADDI  r1, 4        [CONST 4|1]
MUL   r2, rA, r1   [*|3]
SUBI  r3, rB, 15   [-|9]
LOAD  r4, 7(r3)    [MEM|11]
STORE r4, 0(r2)    [MOVE|14]
```

Figura 12 – Instruções Geradas Junto de Suas Regras e Custos

É interessante observar também que, embora o resultado tenha sido o mesmo do Maximal Munch, isso se deve aos custos atribuídos aos padrões, pois o método é diferente. Caso, por exemplo, o padrão *14* tivesse custo quatro, o padrão selecionado poderia ter sido tanto o *14* quanto o *17*, e se o custo fosse cinco ou mais, certamente o *17* seria escolhido. O mesmo acontece com outros nós, como na subtração, se o padrão *9* tivesse custo maior, ele não necessariamente seria escolhido em detrimento de *4*.

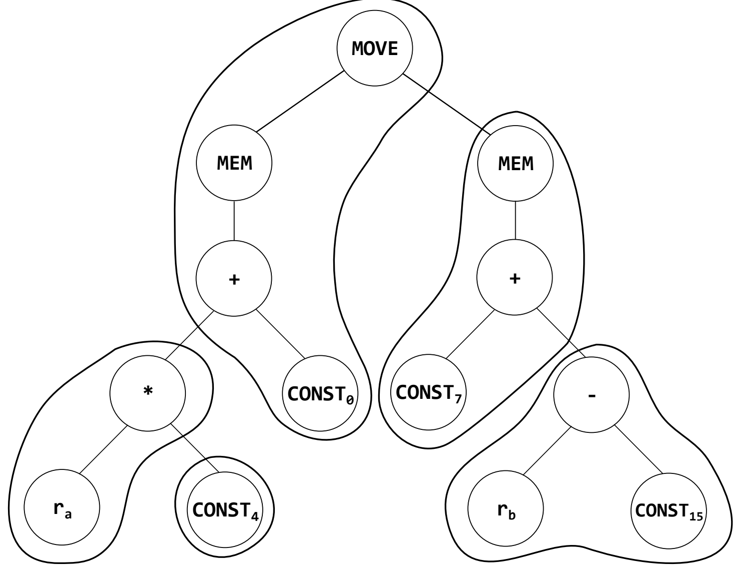

Figura 13 – Ladrilhamento por Programação Dinâmica

Nome	Nº	Produção	Custo	Padrão
-	1	-	0	TEMP
ADD	2	$r_i \leftarrow$ + r_j r_k	1 + r_j + r_k	+
MUL	3	$r_i \leftarrow$ * r_j r_k	2 + r_j + r_k	-
SUB	4	$r_i \leftarrow$ - r_j r_k	1 + r_j + r_k	-
DIV	5	$r_i \leftarrow$ / r_j r_k	2 + r_j + r_k	/
ADDI	6	$r_i \leftarrow$ + r_j c	1 + r_j	+ CONST
	7	$r_i \leftarrow$ + c r_j	2 + r_j	+ CONST
	8	$r_i \leftarrow$ c	1	CONST
SUBI	9	$r_i \leftarrow$ - r_j c	1 + r_j	- CONST
LOAD	10	$r_i \leftarrow$ m + r_j c	2 + r_j	MEM + CONST
	11	$r_i \leftarrow$ m + c r_j	2 + r_j	MEM + CONST
	12	$r_i \leftarrow$ m c	2	MEM CONST
	13	$r_i \leftarrow$ m r_j	2 + r_j	MEM
STORE	14	$\lambda \leftarrow$ mv m + r_j c r_k	2 + r_j + r_k	MOVE MEM + CONST
	15	$\lambda \leftarrow$ mv m + c r_j r_k	2 + r_j + r_k	MOVE MEM + CONST
	16	$\lambda \leftarrow$ mv m c r_j	2 + r_j	MOVE MEM CONST
	17	$\lambda \leftarrow$ mv m r_j r_k	2 + r_j + r_k	MOVE MEM
MOVEM	18	$\lambda \leftarrow$ mv m r_j m r_k	2 + r_j + r_k	MOVE MEM MEM

Tabela 5 – Padrões de Árvores com Custos

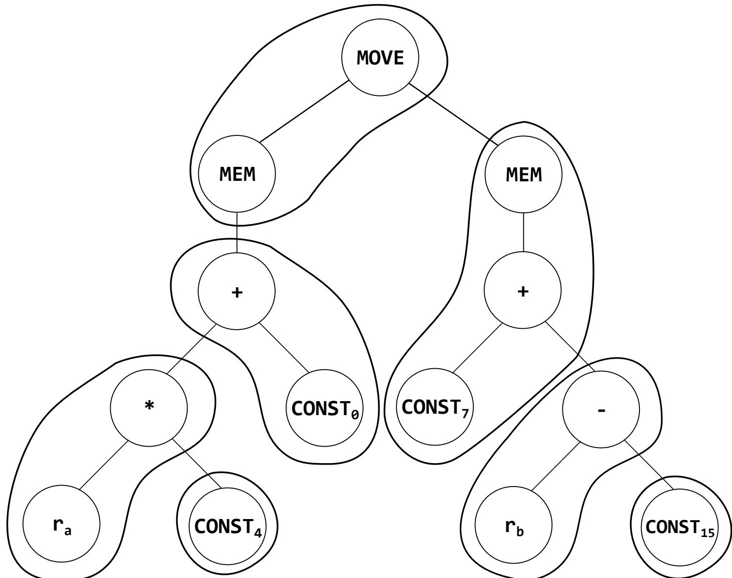

Figura 14 – Ladrilhamento por Programação Dinâmica com os Custos Alterados

3 GERADORES DE GERADORES DE CÓDIGO

A geração de código é um tópico que data das primeiras linguagens de alto nível, desde a saída da programação diretamente em linguagem de máquina, foram necessárias maneiras de se traduzir linhas mais legíveis para humanos em *assembly*. Sendo assim, não é surpreendente que hajam tantos trabalhos que tratam desse tema, visando tornar esse processo cada vez mais eficiente.

Esse capítulo abordará alguns desses trabalhos, e falará sobre suas ideias e metodologias, método de funcionamento, além de inovações que trouxeram para em relação à outras ferramentas. Ao fim do capítulo se encontra um comparativo entre todos os algoritmos abordados.

3.1 Burg

O burg, descrito por Fraser et al. [17], cujos autores se referem a ela como uma rápida seletora de instruções ótimas, utiliza a técnica BURS (*Bottom-Up Rewrite System*) [18] para realizar uma análise ótima de uma linguagem descrita em tempo linear.

3.1.1 Geração do Seletor de Instruções

O Burg recebe como entrada um arquivo similar ao do YACC [9], onde é descrita uma gramática em forma de árvore, cada regra possui um custo associado. Sua saída é um código C, chamado Burm, que realiza a seleção ótima de instruções. O Burg resolve muitos dos problemas evidenciados pelo Twig [13], tendo como consequência uma maior velocidade de execução, ao custo de ser uma ferramenta menos flexível.

Tal como o YACC, Flex, e Bison, por exemplo, o arquivo de entrada do Burg é composto por três seções, divididas pelo símbolo "%%". A primeira seção serve para configurar o programa, com diretivas como `%term` e `%start` para definir símbolos terminais e o símbolo inicial da gramática, por exemplo, além de um trecho de código com `#define` obrigatórios de macros, que dizem para o programa como andar e interagir com a árvore. Definições como `NODEPTR_TYPE`, que dita o tipo de ponteiro da árvore, `OP_LABEL(p)`, `LEFT_CHILD(p)` e `RIGHT_CHILD(p)`, respectivamente o operador do nó, tal como seus filhos esquerdo e direito, uma macro `PANIC` para caso encontre algum erro na execução e, por fim, um `STATE_LABEL(p)`, que pega o estado para qual o nó aponta.

A segunda seção do programa define as regras da gramática, divididas em dois lados, e tem a seguinte aparência `addr: Plus(con,Mul(Four,reg)) = 5 (0);`. O lado esquerdo representa o símbolo não-terminal que aquela regra, enquanto o lado direito é

um padrão, uma árvore parentizada em notação polonesa. Essa árvore pode consistir em um único terminal, ou de um conjunto de nós onde o pai é um operador, e os filhos são operandos, onde cada filho pode também ser uma árvore. Esse último caso é chamado de *chain rule*.

Depois da árvore vem um inteiro positivo precedido por um símbolo de igual, que identifica a regra, e idealmente formam uma enumeração exata. O usuário pode definir funções para realizar ações semânticas nas árvores, e a ferramenta utiliza esses números de identificação para associar aos procedimentos do usuário. Por fim, há um vetor opcional de números envoltos por parênteses, que representam o custo associado àquela regra, que é assumido como zero caso seja omitido. Como é de se esperar, o custo de uma derivação é a soma dos custos de todas as derivações anteriores, número que só aumenta conforme mais derivações são realizadas. Embora alternativas possam ser usadas, por padrão o Burg só considera o "custo principal", que é o primeiro elemento do vetor de custos.

A última seção do arquivo de entrada nada mais é do que código C, que será copiado para o fim do arquivo do Burm.

3.1.2 Arquivo de Saída

O arquivo gerado pelo Burg, denominado Burm, é o seletor de instruções. Por utilizar a metodologia do BURS, ele realiza duas passagens na árvore de entrada. A primeira, chamada de *rotulador*, visita os nós de baixo para cima e da esquerda para a direita, rotulando cada um com um número inteiro que representa um estado, um conjunto de todas as seleções ótimas naquele nó. Essa visita é feita exatamente uma vez para cada nó.

A segunda passada, denominada *redutora*, é uma visita *top-down* de redução da árvore. A função de redução toma como parâmetros o rótulo do estado de um nó da árvore, além de um não-terminal como objetivo, que inicialmente são a raiz da árvore, e o símbolo inicial da gramática, que juntos determinam qual regra deverá ser esolhida para ser casada naquele nó. Desta vez, quando um nó é visitado, as ações definidas pelo usuário (designadas pelo "=" na regra) são executadas, podendo controlar a ordem em que subárvores são visitadas, além de gerar efeitos colaterais, como a emissão de código, dentre outras coisas que o usuário defina.

O Burm possui as seguintes funções que são geradas pelo Burg:

```
extern int burm_labeI(NODEPTR_TYPE p);
```

Essa função é a utilizada pelo rotulador, a primeira etapa do programa, e rotula toda a árvore *p*, retornando o seu estado. Um valor zero significa que a árvore não pode, por qualquer razão, ser casada.

A função a seguir é similar, porém mais simples, do que a `burm_label`. Ela não lê

nem escreve nos campos dos nós passados, e portanto não viaja pela árvore também, ela meramente rotula os nós passados, e retorna o estado a ser associado com o nó *op*. Caso todos os operadores da gramática sejam unários, a segunda versão da função é gerada.

```
extern int burm_state(int op, int leftstate, int rightstate);
```

```
extern int burm_state(int op, int leftstate);
```

O Burm, por si só, não possui procedimento nativo para realizar a tradução da árvore para linguagem de máquina, mas ele gera algumas funções e estruturas para auxiliar o usuário nesse processo.

A função a seguir recebe o rótulo do estado de uma árvore, e também um não-terminal que será o objetivo. Caso os dois parâmetros não tenham associação, a função retorna zero, senão ela devolve o número de uma regra definida pelo usuário. Para as chamadas recursivas da função, o Burm possui um vetor, indexado pelos números das regras, onde cada elemento é um vetor que engloba os não-terminais para o padrão daquela regra.

```
extern short *burm_nts[ ]  ... ;
```

Embora apenas esses dois elementos sejam suficientes para que o usuário defina a redução, o Burm oferece uma função extra que visa facilitar o trabalho:

```
extern NODEPTR_TYPE *burm_kids(NODEPTR_TYPE p, int eruleno, NODEPTR_TYPE
kids []);
```

Ela recebe uma árvore *p*, o número de uma regra, e também um vetor vazio de árvores. É assumido que a árvore passada tenha associação com a regra. A função então retorna o vetor *kids* preenchido com as subárvores de *p* que devem ser reduzidos de maneira recursiva.

3.1.3 BURS

O BURS [18] é uma técnica para geração de geradores de código que não requer que o cálculo dos custos seja feito no gerador. Ao invés disso, o custo ótimo é colocado nos estados do autômato, então esses estados deixam de ser apenas um conjunto de regras, e se tornam um conjunto de pares regra-custo.

Pela natureza *bottom-up* da estratégia, é possível computar o conjunto de pares para cada nó da árvore. O menor custo é encontrado subtraindo-se o custo base de cada regra pelo menor custo das regras associadas à essa.

A técnica é uma alternativa para outras, propostas por outros pesquisadores, mais notavelmente a de análise LR(1) [4], a de programação dinâmica [13], e uma evolução da estratégia *bottom-up* de [19]. Todas elas têm suas vantagens e desvantagens: a primeira provê facilidade na descrição da gramática, além de corretude e simplicidade do código

gerado, porém seu gerador não lida com ambiguidades adequadamente, além de ser demorado e extenso; a segunda lida corretamente com ambiguidades, e garante um código sempre ótimo, mas leva muito tempo para calcular os custos durante a geração de código; e a última garante uma tradução ótima sem a necessidade de calcular o custo durante a geração de código, contudo ela requer alterações na especificação da máquina alvo.

Desta maneira, o BURS é uma evolução da técnica de Hatcher e Christopher [19] no sentido de que ela não necessita de alterações na especificação da máquina alvo, além de empregar uma extensão de uma técnica para compressão de tabelas, e assim melhorar o tempo de processamento.

3.2 Iburg

O Iburg [15] é outro programa para gerar analisadores de árvores, utilizando a mesma especificação de entrada utilizada pelo Burg. Ele se diferencia ao utilizar programação dinâmica em tempo de compilação, ou seja, enquanto o programa está sendo executado, em contraste com o Burg, que utiliza programação dinâmica na ferramenta gerada. Isso permite que o Iburg possa utilizar ponteiros para referenciar suas estruturas, enquanto o Burg está limitado a utilizar números que referenciam funções de custo.

Além disso, o Iburg utiliza um casador de padrões estaticamente codificado, uma técnica que se provou eficiente em outros geradores de código. A ferramenta também é mais simples de ser utilizada, uma vez que permite a utilização de uma classe maior de gramáticas, e também por permitir uma computação dinâmica dos custos.

3.2.1 Casamento de Padrões e Implementação

Tal como o Burg, esse programa tem sua função `label(p)`, que realiza uma visita *bottom-up, left-to-right* na árvore *p*, porém aqui, ao invés de cada nó receber um número de estado, eles recebem um ponteiro para um par *(M, C)*, que indica que o padrão associado com a regra M possui custo C. O custo é a soma de cada não-terminal da árvore, e essa anotação só é dada para os nós se C for menor que todos os custos de casamento anteriores. A seguir está a implementação dessa função, que recebe a raiz e avalia recursivamente o resto da árvore (`NODEPTR_TYPE`, `LEFT_CHILD`, e `RIGHT_CHILD` representam as mesmas macros do Burg, enquanto `OP_LABEL` e `STATE_LABEL` são funções que retorna o número do símbolo, e acessa o número do estado):

```
int label(NODEPTR_TYPE p) {
    if (p) {
        int l = label(LEFT_CHILD(p));
        int r = label(RIGHT.CHILD(p));
        return STATE_LABEL(p) = state(OP_LABEL(p), 1, r);
```

```
    } else
    return 0;
}
```

A função `state( int symbol_number, NODEPTR_TYPE left_child, NODEPTR_TYPE right_child)` retorna o número do estado a ser designado para aquele nó. Ela é implementada com código bruto, e seus números de estado são ponteiros para um vetor dos pares *(M, C)*. Um exemplo desse estado seria o seguinte:

```
struct state {
    int op;
    struct state * left, * right;
    short cost[6];
    short rule[6];
};
```

Ao serem criados, os estados estão vazios, e possuem somente um número inteiro positivo associado à eles, que é o número da regra. Sendo assim, um valor não-zero para `p->rule[ X ]` mostra que o nó p casou com uma regra que é definida pelo símbolo X. A função `state` cria e inicializa um novo estado, inicializando também o número do símbolo para que ele possa ser casado. Todos os nós possuem uma quantidade de *if's* igual ao seu número de filhos, que realizam o casamento consultando os estados de seus nós filhos. Para nós folha, um *switch* cuida do casamento.

A ferramenta define também uma função `record` (definida a seguir), que é chamada quando um casamento é bem sucedido. Seus parâmetros são um ponteiro para um estado, o número do não-terminal casado, o custo, e o número externo da regra. O novo estado só é registrado se o custo for menor que casamentos anteriores, e como os custos são iniciados com o valor "infinito" (o maior valor possível de um inteiro com sinal de 16 bits, 32.767), o primeiro casamento sempre será registrado.

```
void record(struct state *p, int nt, int cost, int eruleno) {
    if (cost < p->cost[nt]) {
        p->cost[nt] = cost;
        p->rule[nt] = eruleno;
    }
}
```

3.2.2 Otimizações

Este analisador é simples e prático para vários geradores de código, além de criar um programa pequeno. Ainda sim há pequenas melhorias que podem ser feitas para otimizar o código.

O vetor `rule` nos estados podem armazenar qualquer número de regra, porém vários não-terminais são definidos por apenas algumas regras. Por isso, suas definições podem ser mapeadas em pequenos intervalos de um número inteiro, o que visa economizar espaço e tempo em especificações maiores. O vetor previamente definido como `short rule[6];`, agora é da seguinte forma:

```
struct {
    unsigned int stmt:2;
    unsigned int disp:2;
    unsigned int rc  :2;
    unsigned int reg :3;
    unsigned int com :2;
} rule;
```

Como essa nova struct não pode ser indexada da mesma maneira de um vetor, a função `record` deve ser alterada, e uma maneira de pegar esse dado deve ser adicionada. Para acessar essa nova definição da regra, uma função `rule` é implementada, que utiliza tabelas para mapear a representação compacta para números inteiros:

```
short decode_disp[] = {0, 10, 11};
short decode_rc[]   = {0, 12, 13};
short decode_stmt[] = {0, 4, 5};
short decode_reg[]  = {0, 6, 7, 8, 9};
short decode_ con[] = {0, 14, 15};

int rule(int state, int goalnt) {
  struct state *p = (struct state *)state;
  switch (goalnt) {
    case disp_NT: return decode_disp[p->rule.disp];
    case rc_NT:   return decode_rc[p->rule.rc];
    case stmt_NT: return decode_stmt[p->rule.stint];
    case reg_NT:  return decode_reg[p->rule.reg];
    case con_NT:  return decode_con[p->rule.con];
  }
}
```

Embora juntar os números de regra dessa maneira tenha um impacto pela maior dificuldade de codificar e decodificar, isso é compensado pelo fato da estrutura menor poder ser inicializada de maneira mais rápida, copiando uma única estrutura. Essencialmente, como ela é definida com campos de bits, o exemplo acima é meramente um *int*.

É possível economizar ainda mais recursos na inicialização. Embora todos os custos devam ser definidos, o campo `rule` só precisa ser inicializado para o símbolo inicial da gramática. Além disso, as regras são lidas em apenas dois lugares: a função `rule`, e nas comparações para casamento. No primeiro caso, a função é chamada recursivamente a partir da análise *top-down*, que começa com o símbolo inicial como não-terminal objetivo. Na segunda ocasião, nenhuma inicialização da regra é necessária, já que são lidos os campos dos nós filhos. Assim, a inicialização praticamente não é mais necessária, salvando tempo, e com o campo `rule` ocupando um único *int*, espaço também é economizado.

Outra otimização que pode ser feita se encontra na função `record`. Originalmente, quando um custo falhava, as regras de cadeia ainda eram testadas. Essa otimização faz com que esses testes não ocorram mais caso a checagem de custo falhe, economizando assim tempo que seria desnecessariamente gasto.

Há também uma melhoria que visa compactar a manipulação de regras de cadeia. Quando um não-terminal X é alcançado através dessas regras, e caso esse casamento tenha um custo menor do que uma regra encontrada previamente, a ferramenta cria uma função `closure_X` para esse não-terminal e, se aplicável, chama outra função de fecho.

```
void closure_X(struct state *p, int c) {
    if (c + 1 < p->cost[reg_NT]) { / * reg X * \
        p->cost[reg_NT] = c + 1;
        p->rule.reg = 4;
        closure_reg(p, c + 1);
    }
}
```

Por fim, é proposta uma mudança para economizar recursos na computação das folhas, uma vez que são tão abundantes. Diferentemente dos nós intermediários, que exigem um processo custoso para sua computação, as folhas sempre casam com alguma regra, e seu campo de estado é facilmente computado simulando atribuições e os conjuntos de fecho. A ideia aqui é economizar o tempo de casamento das folhas.

```
case x: { /* x is a terminal */
    static struct state z = { 295, 0, 0,
        { 0,
```

```
            1, /* stmt: reg */
            0, /* disp: x */
            1, /* rc: reg */
            1, /* reg: X */ /* X is a non-terminal*/
            32767,
        },{2, /* stint: reg */
            2,/* X: x */
            2, /* rc: reg */
            4,/* reg: X */
            0,
            }
        };
    return (int) &z;
}
```

3.3 Olive

O Olive Twig [20] é uma versão mais nova do Twig [13], com base nas melhorias trazidas pelo Iburg. Embora o artigo se refira à ferramenta como Novo Twig, este trabalho o chamará de Olive, por simplicidade. Seu predecessor, embora intencionado para ser utilizado em geração de código, graças à sua natureza flexível, também era utilizável em outros contextos, como por exemplo sistemas de síntese lógica, como foi citado no próprio artigo do Olive. Uma grande característica que influenciava isso, era a maneira como ele calcula os custos. Ao invés de cada regra receber um número fixo, como ocorre nos outros programas vistos até então, o usuário fornece um bloco de código em C que calcula esse custo.

A técnica de programação dinâmica empregada no Twig, combinado com os custos inseridos pelos usuários, fazem dele uma ferramenta poderosa. Porém, visto que os custos não são visíveis ao programa, seu cálculo precisa ser feito durante a execução do casador de padrões. Técnicas como a do BURS, onde os custos são armazenados nos estados dos autômatos, e a programação dinâmica que é feita previamente para criá-lo, não podem ser aplicadas graças à esse fato. Sendo assim, toda a programação dinâmica do Twig é feita no tempo de execução do casador de padrões, tornando essa etapa mais lenta.

Ferramentas mais novas, como o Burg e Iburg, por exemplo, se beneficiam das deficiências de estratégias anteriores, ao aprender sobre elas, e melhorá-las. O Burg resolveu esse problema do Twig ao aplicar a teoria BURS, trazendo a programação dinâmica para cálculo dos custos durante a etapa de criação do analisador. Ele porém tinha um problema de difícil legibilidade. Como seus detalhes ficavam em tabelas difícies de ler,

realizar o *debugging* do código não era fácil. O Iburg resolveu isso ao implementar seus estados com *if's* e *switch's*, além de outras melhorias já discutidas.

3.3.1 Melhorias

Tendo em vista esses problemas, um novo Twig é proposto, para se tornar uma ferramenta especializada em geração de código. As mudanças que o Olive propõe são:

1. Uma linguagem de especificação mais rica: A proposta é facilitar o fluxo de informações entre as regras, coisa que não podia ser feita no Twig, nem em ferramentas como o Burg ou Iburg, que não ofereciam especificações tão poderosas, pois eram supostas a serem *back-ends* para outros sistemas que podiam ter linguagens mais ricas. Enquanto no Twig a comunicação entre regras só poderia ser feita através de variáveis globais, o Olive propõe tratar as regras como funções que invocam outras regras, o que facilitará a troca de informações entre elas;

2. Casamento de padrões mais rápido: Baseando-se no Iburg, o Olive implementa as tabelas como séries de *if's* e *switch's*, técnica provada a ser mais rápida, e de mais fácil visualização;

3. Custos generalizados: O Olive mantém os custos generalizados do Twig, embora proponha diminuir o impacto desse método, se aproximando à complexidade de tempo alcançada pelo Iburg.

3.3.2 Especificação da Linguagem

Identificadores são definidos por cadeis de letras, números, e *underline*s, sendo precedidos por uma letra ou '%', que é utilizado para identificar palavras chave do Twig. As pontuações são dadas pelos símbolos ' ; : ( ) , = '.

Além disso, as regras do Olive podem conter expressões em C, envoltas por parênteses, ou blocos de código inteiros, encapsulados em chaves. Além disso, trechos iniciados por '$' têm um significado especial, e são tratados pelo Olive.

3.3.3 Gramática

O Olive possui a gramática a seguir:

```
    rule -> nonterm : tree [cost] = action;
    tree -> term(tree_list)
         | term
         | nonterm
tree_list -> tree_list, child
```

```
         | child
   child -> tree
         | _
    cost -> C-code
         | C-expr
  action -> C-code
```

Aqui, *nonterm* e *term* correspondem, respectivamente aos símbolos não-terminais: os padrões de regras; e aos símbolos terminais da gramática, os nós da árvore. O caractere '_' é especial, e quer dizer que o Olive passará por ele sem executar nenhum ação. Para uma regra casar com uma árvore, são necessárias duas condições:

1. A árvore deve casar com o padrão. Para que uma árvore case com um padrão $x(t_1, t_2, ..., t_n)$, sua raiz tem um símbolo x, possui n subárvores, e a mais a esquerda casa com t_1, a seguinte com t_2, e assim sucessivamente até t_n;
2. O custo da regra deve ser um número finito.

3.3.4 Custos

O custo é calculado quando uma regra é casada, e pode assumir uma dentre três formas, um bloco de código C, um expressão em C, ou um valor fixo. No último caso, pode-se atribuir valor zero para o custo para automaticamente casar uma regra, ou infinito para descartá-la de imediato.

3.3.5 Ações

O Olive trata cada não-terminal como se estivesse associado a um conjunto de funções idênticas em sua declaração, ou seja, todas possuem mesmo tipo de retorno, e mesma quantidade e tipo de argumentos. Não-terminais podem ser declarados com a diretiva `%declare` da ferramenta.

As partes `action` e `code` de uma regra podem se comunicar com o casador de padrões a partir de algumas diretivas:

- *$n*: Acessa o n-ésimo nó da árvore. Os nós são numerados com a ordem que aparecem na regra, e *$0* se refere ao não-terminal do lado esquerdo da regra.
- *$cost[n]*: Retorna o custo do n-ésimo nó da regra. Atribui-se um custo à regra com *$cost[0]*.
- *$action[n](a, b, ...)*: Executa a ação associada ao n-ésimo nó, passando como parâmetro *a*, *b*, etc. A ação retorna o tipo especificado na declaração, e pode ser executada múltiplas vezes.

- *$immed[n, lth](a, b, ...)*: Executa a ação associada ao *n*-ésimo nó, passando como parâmetro *a*, *b*, etc. A diferença é que o nó em questão deve ser '_', e a ação associada à ele deve ter o mesmo protótipo que o não-terminal `lhs` do lado esquerdo de sua regra.
- *$match[n, lth]*: Retorna verdadeiro se o *n*-ésimo nó casa com o não-terminal *lth*. O nó em questão deve ser '_'.

3.3.6 Declarações

Símbolos terminais e não-terminais devem ser declarados antes de poderem ser utilizados. Terminais, ou uma lista deles, são declarados com a diretiva `%term`, e o Olive cria um `#define` para que ele possa ser referenciado. Não-terminais utilizam a diretiva `%declare`, e devem ser associados a um protótipo, como mostrado a seguir:

```
%declare<return_type> nonterm<arguments>
```

3.3.7 Interfaceamento

Para que o Olive funcione corretamente, o usuário deve fornecer dois tipos de dados: um para a representação dos nós da árvore, e outro para os custos. Isso provê maior flexibilidade em como representar esses dados.

As representações para custos são:

- `COST`: Um *data type* em C, definido por `#define` ou `typedef`.
- `INFINITY`: O maior valor possível para um custo, utilizado para garantir que um padrão não seja casado.
- `DEFAULT_COST`: Define um custo padrão para a regras que não são definidas com um.
- `COST_LESS(x, y)`: Função que compara dois custos, e retorna verdadeiro caso o custo x seja menor que o custo y.

Para árvores, são disponibilizadas as seguintes representações:

- `NODEPTR`: O tipo do ponteiro para o nó;
- `NULL`: Representa um nó nulo.
- `GET_KIDS(r)`: Função que retorna um vetor de `NODEPTR` para os filhos de r.
- `OP_LABEL(r)`: Função que retorna o rótulo do nó r.

- `SET_STATE(r, s)`: Atribui o rótulo do estado s para o nó r.
- `STATE_LABEL(r)`: Retorna o rótulo previamente atribuído ao nó r.

O Olive gera duas funções: uma chamada `burm_label`, que recebe o nó de uma árvore e realiza o casamento de padrões nela, retornando zero em caso de falha; a segunda é a `X_action`, sendo X o não-terminal que casou na raiz, ela executa as funções associadas à cada regra.

3.4 Diferenças Principais Entre as Ferramentas Apresentadas

Cada nova ferramenta aprende com as experiências das anteriores, e é possível tomar decisões mais conscientes. Embora o Burg não tenha sido o primeiro gerador de geradores de código, ele foi criado ainda no início desse campo, portanto era uma ferramenta menos moderna, embora ainda muito inovadora e veloz.

3.4.1 Principais Características

O Burg foi capaz de criar um analisador muito mais rápido que o Twig, vendo que ele gastava muito tempo para calcular os custos, uma vez que eles não eram visíveis ao programa. O Iburg tentou trazer otimizações e uma maneira diferente de representar suas tabelas, colocando-as em código bruto, porém ainda sim não foi capaz de criar analisadores tão rápidos quanto os do Burg. Apesar disso, conseguiu criar um código mais amigável ao usuário final, não deixando seus detalhes escondidos em tabelas.

Apesar do Burg e do Iburg serem ferramentas rápidas, elas foram criadas com o propósito de servir de *back-end* para outras aplicações. O Olive surgiu com o propósito de suportar linguagens de especificação mais robustas, trazendo aprimoramentos como custo dinâmico, e uma interface mais robusta com o programa. Apesar do possível impacto na velocidade do programa em comparação aos outros dois, ele propunha ser mais rápido que o Twig, mantendo ainda seu poder de expressão. Isso seria obtido com técnicas empregadas pelas outras ferramentas, como as tabelas representadas em código bruto do Iburg, por exemplo.

3.4.2 Metodologia

Embora todas as ferramentas tenham a mesma proposta, elas realizam seu objetivo de maneiras diferentes. O Burg utiliza programação dinâmica durante a criação das tabelas, calculando os custos estáticos de cada regra e inserindo-os na tabela gerada. Isso permite que o Burm (programa gerado) possa apenas casar os padrões, sem o peso adicional de calcular os custos de cada um.

	Estrutura Interna	Programação Dinâmica	Custo Dinâmico
Burg	Tabela	Programa Gerado	Não
Iburg	Código Bruto	Próprio Programa	Não
Olive	Código Bruto	Programa Gerado	Sim

Tabela 6 – Comparação entre os programas estudados

O Iburg, por outro lado, não gera tabelas em si, mas sim funções para os não-terminais, que são chamadas quando aquele não-terminal é encontrado, e então a análise de seus filhos é feita recursivamente. Além disso, diferentemente do Burg, o Iburg não inclui os custos nessas funções, ao invés disso, essa computação é feita no programa gerado, onde também fica a parte da programação dinâmica.

O Olive, por sua vez, utiliza a mesma metodologia do Iburg para sua geração de funções de não-terminais, e também de programação dinâmica para os custos, sendo essa realizada no programa gerado, e não no analisador de gramática.

3.4.3 Pontos Fracos

Como visto anteriormente, cada uma das ferramentas traz inovações na maneira de gerar o código. Apesar disso, todas elas possuem algum ponto fraco, seja graças ao desconhecimento de alguma técnica, seja por decisões de design que impossibilitam uma maior eficácia em um dado ponto.

Para exemplificar uma decisão de design que representa um sacrifício em eficácia, é possível observar o cálculo de custo dinâmico. Como esse cálculo é uma seção de código inserida pelo programador, ao invés de um número fixo, não é possível definir o custo na geração do programa, ao invés disso é necessário que ele seja calculado em cada padrão encontrado. Optar por custo dinâmico torna o programa gerado mais lento, porém permite um maior controle sobre o custo de cada padrão.

Por utilizarem a metodologia BURS, tanto o Burg como o Iburg são ferramentas mais limitadas em seu poder de expressão, de modo que usuários só possam definir valores fixos para os custos.

O Olive, embora prometa ser uma ferramenta mais completa, porém na realidade ele é um programa ainda incompleto, de modo que a autora desse texto teve dificuldades para se rodar o programa.

A ferramenta proposta neste trabalho visa lidar com esses pontos fracos através de um algoritmo de cálculo dinâmico para os custos, o que afetará a performance do programa em comparação à metodologia BURS, porém foi optado por uma maior expressão e liberdade para o usuário. Além disso, por ser programado em C++, o programa será naturalmente mais legível, e também haverá uma maior facilidade de realizar manuten-

ção no mesmo, seja para correção de erros, quanto para implementação de novos recursos. Ademais, instruções claras, tal como uma implementação exemplo serão disponibilizadas, para que não hajam dúvidas na utilização.

4 ALOCAÇÃO DE REGISTRADORES

Registradores são circuitos digitais para armazenamento de dados. Eles são extremamente rápidos, ocupando o topo da hierarquia de memória, porém extremamente custosos [21]. Essas características os tornam ideias para a comunicação direta com a CPU, porém também são muito escassos: um *Intel Core i7* por exemplo possui apenas 16 registradores de propósito geral quando em modo *64 bits* [22]. É possível observar assim que registradores são um recurso importante e escasso e, uma vez que geralmente há várias variáveis em um programa, aloca-los para armazenar as variáveis de um programa é uma tarefa delegada ao *back-end* de um compilador.

Apesar disso, as ferramentas aqui estudadas não lidam com alocação de registradores, com foco unicamente no casamento de padrões, deixando esse papel a cargo do desenvolvedor do compilador. O programa proposto nesse trabalho irá incorporar um alocador de registradores opcional, para que seja mais fácil criar geradores de código e testá-los se funcionam corretamente.

4.1 Métodos de Alocação

É impensável cogitar que, com a criação e popularização de linguagens de alto nível, o trabalho de escolher quais registradores seriam usados, e quais iriam para a memória continuaria a ser feita manualmente pelos programadores, o que é provado pelo fato do *Fortran H*, compilador criado pela IBM para a linguagem *Fortran*, já na década de 60 incluía um alocador de registradores próprio [16].

Atualmente há alguns algoritmos principais [23] para resolver o problema de quais variáveis são mapeadas para quais registradores, e também o momento que uma variável que ainda será utilizada no futuro precisa ser mandada temporariamente para a memória.

Primeiramente há a solução trivial, que consiste simplesmente em ignorar esse problema e manter sempre todas as variáveis em memória, apenas colocando-as em registradores quando elas são utilizadas. Esse método é extremamente fácil de implementar, porém deixa completamente de lado a vantagem de se usar registradores, além da enorme quantidade de escritas e leituras da memória principal, que são operações custosas.

Outro método, desenvolvido mais recentemente, é o *Linear Scan* (Varredura Linear) [24]. Ele é um algoritmo guloso que calcula o *live range* das variáveis, ou seja, o intervalo que se inicia quando uma variável é criada e vai até sua última utilização, e atribui registradores a essas variáveis de maneira cronológica. Quando, em um determinado momento, há mais variáveis vivas do que registradores disponíveis, ocorre um processo

chamado *spilling*, onde uma variável é armazenada em memória principal, ao invés de um registrador. Devido à sua natureza gulosa, o algoritmo não necessariamente gera um resultado ótimo, porém isso também implica em uma execução muito rápida, sem necessidade de realizar muitos cálculos para encontrar a melhor solução.

Por fim há o método de *Graph Coloring* (Coloração de Grafo) [25] que, embora antigo, ainda hoje é o algoritmo predominante para solucionar o problema de alocação de registradores, o que se deve a ele encontrar uma solução muito boa. As variáveis são representadas como nós em um grafo, e quando duas variáveis estão vivas em um mesmo momento, uma aresta é inserida entre seus respectivos nós, criando assim um grafo de interferência. O problema então é de como colorir esse grafo, de maneira que nenhum par de nós conectados por uma aresta tenham a mesma cor, sendo a quantidade de cores disponíveis igual à quantidade máxima de registradores. Caso não seja possível encontrar uma coloração, tenta-se diminuir a quantidade de interferências de um dado nó através do *spilling*, processo que é repetido até que seja possível colorir o grafo completamente. Geralmente aplica-se algum tipo de heurística para decidir qual nó sera selecionado para o *spilling*. O problema de coloração de grafos é NP completo, ou seja, não foi encontrado um algoritmo que possa solucioná-lo em tempo polinomial. Apesar disso, a heurística usada [16] foi encontrado a solucionar esses problemas de maneira consideravelmente rápida na prática.

O alocador de registradores criado nesse trabalho utilizará a técnica de Varredura Linear. Em contraste com a complexidade quadrática do algoritmo de Coloração de Grafo [26], sua rápida velocidade de execução é ideal para prototipar a linguagem a ser criada, enquanto ainda fornecendo um resultado aceitável no código gerado.

4.2 *Linear Scan*

Com o objetivo de criar um algoritmo de alocação de registradores menos computacionalmente custoso foi criado o *Linear Scan*. Métodos mais agressivos, que se utilizam da coloração de grafos, são $O(n^2)$ no pior caso, enquanto esse método de Varredura Linear possui complexidade $O(n)$.

4.2.1 Metodologia

O primeiro passo na tarefa de alocação é saber em quais pontos uma varável existe ou não, um conceito conhecido como *live range*. Seja uma variável v e duas instruções i e j, o *live range* da variável é o intervalo *[i, j]*, tal que não haja nenhuma instrução antes de i que utilize v, e também que essa variável não seja acessada em nenhum ponto depois de j. Podem haver subintervalos dentro de *[i, j]* onde v não está viva, porém, para o propósito do método, e simplicidade da análise, esses subintervalos são ignorados, tudo que importa

é o momento de nascimento e de morte das variáveis.

É importante também definir um método de ordenação das instruções, para que seja possível contá-las e saber quais vem primeiro. A métrica utilizada é importante pois, embora a corretude do código gerado não será afetada, a qualidade da alocação pode variar. Duas possíveis métricas são a ordem em que as instruções aparecem no código intermediário, ou uma ordenação por profundidade sobre a AST. Vale notar que a ordem escolhida não tem impacto no funcionamento do algoritmo, apenas na saída. Por simplicidade, será utilizada a ordem em que as instruções aparecem na representação intermediária.

Algorithm 2 Algoritmo Principal da Varredura Linear

```
procedure LinearScan(intervalos)
    ativos ← []
    for i em intervalos do                          ▷ Ordenado por data de nascimentos
        MatarIntervalos(i)
        if tamanho(ativos) = limite_registradores then
            Spill(i)
        else
            registradores[i] ← registradores_livres[0]
            insere(ativos, i)                       ▷ Ordenado por data de morte
        end if
    end for
end procedure
```

Algorithm 3 Algoritmo de Decisão Para Qual Variável Irá Para Memória

```
procedure Spill(i)
    spill ← ativos[tamanho(ativos)]
    if morte(spill) > morte(i) then
        registrador[i] ← registrador[spill]
        insereNaMemoria(spill)
        remove(ativos, spill)
        ativos(i)
    else
        insereNaMemoria(i)
    end if
end procedure
```

O cálculo do intervalo de vida das variáveis é facilmente realizado ao se analisar a representação intermediária do programa. O *live range* de duas variáveis possuem interferência quando ao menos parte de seus intervalos se sobrepõe. Caso, em um dado momento, a quantidade de variáveis cujo *live range* se sobrepões seja maior que a quantidade de registradores disponíveis, as variáveis excedentes devem ir para a memória.

Esses intervalos são mantidos em uma lista, ordenada com base no momento de instanciação de cada variável, portanto, identificar a quantidade de interferências é sim-

Algorithm 4 Algoritmo de Liberação de Registradores

```
procedure MatarIntervalos(i)
    for j em ativos do                                   ▷ Ordenado por data de morte
        if morte[j] ≥ nascimento[i] then
            return
        else
            remove(ativos, i)
            insere(registradores_livres, registrador[j])
        end if
    end for
end procedure
```

ples, sendo necessário apenas olhar o próximo elemento da lista e verificar se os anteriores ainda estão vivos. Ademais, o algoritmo também mantém uma lista de *live ranges* ativos, ou seja, aqueles que, no ponto atual sendo analisado, ainda estão vivos e também que foram atribuídos para um registrador. Essa lista é ordenada a partir do momento de morte das variáveis que estão nela, e a cada iteração, se algum intervalo não está mais vivo, ele é retirado da lista, e o registrador que lhe foi alocado fica livre novamente.

Idealmente a lista de *ativos* sempre terá, no máximo, tamanho R, sendo R a quantidade máxima de registradores. O pior caso acontece quando essa lista cheia, e no momento que um novo intervalo será inserido, não haja nenhum outro que possa ser retirado. Nesse caso, o processo de *spilling* deve ser feito sobre algum desses *live ranges*, seja algum da lista de *ativos*, ou o que está prestes a ser inserido. Há vários métodos para se resolver esse problema, desde métodos cegos, como escolher sempre o primeiro intervalo da lista, até métodos heurísticos, como por exemplo sempre mandar para memória a variável com *live range* mais longínquo, que minimiza o número de variáveis vivendo em memória.

4.2.2 Demonstração Prática

A Figura 15 demonstra uma lista de *live ranges* ordenada a partir dos pontos de instanciação de cada uma, sendo que o eixo Y representa o número das instruções, de tal modo que a variável **B** nasce na instrução *2* e morre na *12*. Seja $R = 2$, ou seja, existem dois registradores disponíveis para se distribuir as variáveis, é evidente que as variáveis **B** e **C** podem ser alocadas sem problemas.

Na instrução *8*, a lista *ativos* é <***B**, **C***>, e nenhuma delas está morta. Quando **A** nasce, duas variáveis já ocupam os registradores existentes, portanto é necessário escolher qual irá para a memória. Utilizando a metodologia de realizar o *spilling* com a variável que viverá mais, a própria **A** é escolhida.

A variável **E** é declarada em *9*, onde os *ativos* são ainda <***B**, **C***>. Dessa vez porém, **C** morre no mesmo ponto onde **E** nasce, então a primeira é removida da lista, permitindo que a nova variável seja inserida sem problemas. Note que, como a lista *ativos* é ordenada

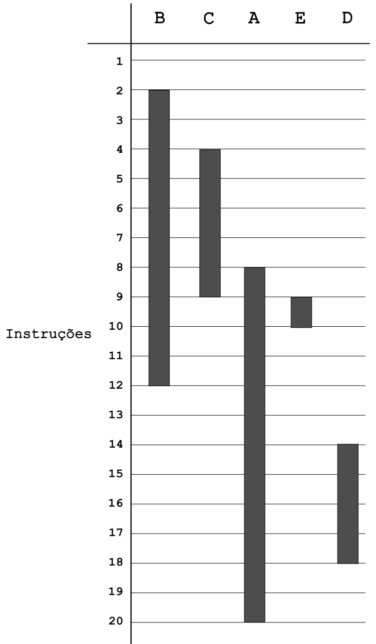

Figura 15 – Exemplo de lista com todos os *live ranges* de um programa

por data de morte, **E** entrará na primeira posição.

Por último, o intervalo **D** é criado na instrução *14*, e a lista *ativos* é <***E***, ***B***>. Porém, como ambas as variáveis na lista já morreram, elas podem ser retiradas para que um registrador seja atribuído a **D**.

É importante citar que, mesmo que o *live range* de **A** ainda não tenha morrido, ele não interfere na entrada de **D**, pois foi decidido que essa variável viverá em memória. O algoritmo de *linear scan* não permite que uma variável com a qual foi feita o *spilling* seja designada um registrador no futuro, mesmo que haja algum disponíveis. Isso permite que o método seja o mais simples possível, ainda sendo efetivo.

É possível constatar, mesmo de maneira textual, a simplicidade do algoritmo. Sem análises ou heurísticas complexas, ele é capaz de encontrar uma alocação adequada para os programas em um curto espaço de tempo.

Uma variação desse algoritmo chamada *Second Chance Binpacking* [27] propõe a possibilidade de dar uma segunda chance às variáveis que foram enviadas à memória. *Live intervals* são utilizados para considerar se uma variável vai ou não para a memória, ao invés de *live ranges*. Isso implica que possíveis buracos nos intervalos dessas variáveis são considerados, então é possível que uma variável que foi mandada para a memória possa ser alocada à um registrador posteriormente.

Esse método é capaz de gerar um melhor código, porém aumenta também a complexidade, embora ainda não se compare à um algoritmo de coloração de grafos. Apesar disso, como foi dito anteriormente, esse trabalho utilizará a versão original e simples do

linear scan, uma vez que o objetivo é apenas proporcionar ao desenvolvedor uma maneira simples de testar seu código.

5 YAMG

A ferramenta proposta nesse trabalho, denominada YAMG (*Yet Another Machine code Generator*), é um gerador de analisadores de árvores, tal como outros apresentados no Capítulo 3. Seu objetivo é criar um programa que, a partir de uma entrada do usuário, gere um programa capaz de ler uma árvore de entrada, e realizar um casamento de padrões ótimo nela, além de, opcionalmente, realizar a alocação de registradores.

O objetivo principal é que essa ferramenta seja usada na etapa final da compilação. Ela fará o casamento de padrões sobre a representação intermediária passada, encontrando os casamentos com menor custo para então realizar as ações associadas às regras encontradas.

É possível criar tais analisadores de maneira manual, mas tal processo é longo e árduo, além de que o programa gerado só seria capaz de realizar análises para um tipo de arquitetura de CPU. Uma alternativa mais rápida é utilizar geradores automáticos de tais analisadores, onde é necessário que o usuário supra apenas a gramática da linguagem alvo, e o gerador faz todo o trabalho pesado de compreender essa gramática e criar uma ferramenta de análise para ela. Além de ser mais fácil, é possível criar analisadores para outras arquiteturas alvo sem a necessidade de escrever um novo analisador ou fazer grandes alterações em um já existente.

5.1 Dois Programas

Este trabalho visa criar uma única ferramenta, embora o resultado final sejam dois programas. Para cumprir o objetivo completo de geração de geração de código, duas etapas são necessárias.

Primeiramente, é necessário a definição de uma linguagem alvo, completa com todos os detalhes da arquitetura. Essa definição, apresentada na seção seguinte, serve como entrada para o primeiro programa, que irá interpretar a linguagem alvo, e criar um tradutor de uma AST para instruções dessa arquitetura.

O segundo programa, gerado a partir do primeiro, é o gerador de instruções. Ele recebe como entrada um arquivo que é uma árvore sintática abstrata da linguagem a ser gerada, casar os padrões, considerando o custo de cada um, dessa árvore com os daqueles definidos no passo anterior. Por fim, após o casamento, tem-se então a emissão das instruções alvo para um arquivo de saída.

Como visto no Capítulo 3, há vários outros programas que cumprem esse mesmo propósito, embora cada uma tenha suas peculiaridades. A ferramenta aqui proposta é

uma evolução das anteriores no sentido de utilizar uma linguagem de programação mais moderna, tal como prover opções adicionais para uma maior comodidade do usuário, como outros algoritmos de casamento de padrão e um alocador de registradores .

Todas as ferramentas vistas, assim como muitas outras, são escritas em C, e geram seu código na mesma linguagem. Sendo assim, as duas etapas da ferramenta sofrem de certa forma com legibilidade, e a falta de algumas características presentes em outras linguagens. Com isso em mente, a ferramenta aqui descrita será feita utilizando C++, e o código que ela gerará será também nessa linguagem.

A presença de classes e construtores facilitam a legibilidade e a visualização da relação entre uma variável e suas funções. Além disso, tipos da biblioteca padrão permitem funcionalidade extra sem a necessidade de escrever código a mais, como por exemplo `std::vector` que permite aos nós terem *n* filhos sem aumentar o tamanho do código, `std::optional` e `std::unique_ptr` possibilitam o manuseamento de valores sem a necessidade de se preocupar com o gerenciamento de memória, `std::string` é um *container* que facilita o uso de strings e abstrai funções pouco legíveis como `strcpy` e `strcmp`, dentre outros tipos prontos que, em C, precisariam ser escritos manualmente.

Não só a legibilidade é melhorada em C++, mas também é mais difícil cometer erros que causariam alguma corrupção na memória, e fica mais fácil dar manutenção ao código para quaisquer mudanças que precisem ser feitas. Esses benefícios são válidos tanto para o programa aqui proposto quanto para as partes de código que serão escritas pelo usuário.

Esta ferramenta se baseia nas técnicas já descritas, de modo que a linguagem de programação é a principal diferença. Primeiramente, é realizada uma análise *bottom-up* da árvore, para realizar os casamentos dos possíveis padrões em cada nó, junto de seu custo. Depois, faz-se uma passagem *top-down* para escolher quais casamentos serão realizados. Tabelas são usadas no lugar de código bruto para representar o autômato, pela simplicidade de sua construção, e o cálculo dos custos é feito com programação dinâmica no código gerado, uma vez que os custos não são estáticos, mas sim providenciados pelo usuário. A Tabela 7 faz uma comparação entre as ferramentas do Capítulo 3 com o YAMG.

Outro aspecto onde o YAMG se diferencia é nas funcionalidades disponibilizadas, além do casador de padrões com custo dinâmico, que é a proposta principal, são fornecidos os três algoritmos de casamento já discutidos: Maximal Munch, Minimal Munch, e Programação Dinâmica. Naturalmente os dois primeiros utilizam o tamanho de cada padrão como heurística, ignorando a função de custo fornecida pelo usuário, enquanto o terceiro método utiliza essa função.

A presença desses outros procedimentos possibilita uma maior flexibilidade na geração de código, sendo possível alterar o algoritmo de casamento simplesmente mudando

	Estrutura Interna	**Programação Dinâmica**	**Custo Dinâmico**	**Linguagem**
Burg	Tabela	Programa Gerado	Não	C
Iburg	Código Bruto	Próprio Programa	Não	C
Olive	Código Bruto	Programa Gerado	Sim	C
Yamg	Tabela	Programa Gerado	Sim	C++

Tabela 7 – Comparação entre os programas

uma chamada de função.

Por último, o YAMG fornece um alocador de registradores, de modo que o código gerado possa ser imediatamente testado, sem a necessidade de passar a saída do programa por outro alocador, ou até mesmo designar os registradores de maneira manual.

É possível observar que, embora os métodos de casamento de padrões utilizados sejam os mesmos de outras ferramentas, o YAMG traz inovações que o tornam mais fácil e seguro de se trabalhar, tal como outras características que não estão presentes em outros programas.

6 YAMG: DEFINIÇÃO DA DESCRIÇÃO DE MÁQUINA

A descrição da máquina é um arquivo com extensão **.md** (*machine description)*, dividido em seções da mesma maneira que ferramentas como *flex*, *bison*, dentre outras, e seu formato será discutido nesse capítulo.

6.1 Linguagem do YAMG

O arquivo de entrada do gerador é dividido em quatro partes (três, se o alocador de registradores não for utilizado), separadas pelo símbolos `%%`. Elas são as seguintes:

- *Definições do Casador*: contém todas as definições pertinentes ao casador de padrões, incluindo terminais e não-terminais, os algoritmos de casamento desejados, e código C++ com implementações obrigatórias;
- *Regras*: seção que descreve todas as regras da descrição de máquina, desde seu padrão de árvore até os custos e ações;
- *Definições do Alocador*: seção opcional que contém identificadores usados pelo alocador, como os nomes dos registradores, nomes de regitradores de *spill*, instruções de acesso à memória;
- *Código Auxiliar*: código C++ opcional possivelmente com definições de funções auxiliares.

Na Figura 16 é possível observar a estutra geral do arquivo de entrada.

```
Definições do Casador de Padrões [padrão]
%%
Regras {custo} {ação} [padrão]
%%
Definições do Alocador de Registradores [seção opcional]
%%
Código Auxiliar C++ [opcional]
```

Figura 16 – Estrutura do Código de Entrada

6.1.1 Definições do Casador de Padrões

A primeira seção descreve os símbolos que fazem parte da linguagem a ser analisada, e também os nós internos (nós que não são folhas). Para o primeiro caso, usa-se

a diretiva %term, seguida de um identificador, já para o segundo, utiliza-se %nonterm seguida de um identificador, como demonstrado abaixo.

```
%term ADD
%term CONST
%nonterm reg
```

Podem-se definir também quais os algoritmos de casamento serão utilizados através da diretiva %arg. São disponibilizados três algoritmos para casamento de padrões: o *Minimal Munch*, *Maximal Munch*, e Programação Dinâmica. O código gerado cria uma função para cada algoritmo selecionado, à escolha do desenvolvedor de qual será utilizada. Caso nenhum algoritmo seja especificado, os três serão gerados. Naturalmente, caso *Minimal* ou *Maximal Munch* sejam utilizados, apenas o tamanho dos padrões será considerado, e não os custos definidos na regra. Os nomes dos algoritmos estão apresentados abaixo.

```
%alg MAXIMAL_MUNCH
%alg MINIMAL_MUNCH
%alg DYNAMIC_PROGRAMMING
```

Além disso, essa seção contém um trecho de código C++, que pode ser composto por *includes*, declarações de funções ou variáveis, ou quaisquer outras estruturas desejadas, mas ele deve conter, obrigatoriamente, as seguintes definições:

- Classe Tree: ela deve estender a classe YamgTree<Tree, ReadType, HelperType= std::string> e implementar os respectivos construtores;
  - Método readTree(readType): responsável pela leitura da árvore de entrada, e atrubuí-la à instância atual da classe;
  - Método readNodeType(HelperType): método responsável pela leitura do tipo de nó;
  - Método readUserSymbol(HelperType): retorna o símbolo daquele nó com base no parâmetro passado.

ReadType e HelperType são tipos quaisquer, sendo HelperType uma std::string por padrão, podendo assim ser omitido. Essa classe é uma interface entre o gerador e o usuário, e nela podem ser definidos atributos extras que podem ser usados nas ações dos nós.

```
//Assume-se que UserAst é um tipo definido pelo usuário, que
// representa a AST do programa.
//O tipo HelperType foi omitido, portanto ele é considerado uma std::string
```

```
class Tree: public YamgTree<Tree, UserAst>{
    public:
        Tree(){}
        void readTree(UserAst) override { ... }
        Node_type readNodeType(std::string) override { ... }
        Yamg::User_Symbols readUserSybol(std::string) override { ... }

        int register_number; //atributo definido pelo usuário;
}
```

6.1.2 Regras

A segunda parte do arquivo é composta pelas regras da linguagem, elas expressam desde o padrão de árvore a ser casado, até a instrução da linguagem alvo que será emitida.

```
        rule -> rule_name <- non-terminal : tree cost = action ;
        tree -> terminal ( tree_list )
             |  terminal
             |  non-terminal
   tree_list -> action tree_list , tree
             |  tree_list , tree
             |  tree
      action -> { CPP-Code }
        cost -> { CPP-Code }
```

`rule_name`: String representando o nome da regra;

`action`: Trecho de código C++ que será executado sempre que o nó for visitado;

`non-terminal`: Símbolo não-terminal para o qual aquela árvore é reduzida.

`tree`: Árvore que representa o padrão, os filhos nós filhos devem aparecer em um conjunto de parênteses, separados por vírgulas, e à direita do nó pai. Nós internos devem ter sido declarados pela diretiva `%nonterm`;

`cost`: Expressão C++ que calcula o custo do casamento por aquele padrão, que deve retornar um valor inteiro, e pode conter diretivas $cost[n], onde n é um número inteiro, começando do zero, e representa o n-ésimo nó da árvore. Para uma árvore `A(B(C,D),E)`, por exemplo, A seria o nó zero, B o nó um, C o nó dois, D o nó três, e E o nó quatro. A diretiva retorna o menor custo para se casar o nó n.

Na Figra 17 encontram-se alguns exemplos de regras, cujos padrões de árvore correspondem àqueles da Figura 18.

```
addReg <- reg: ADD(reg, reg)
        { return $cost[1] + $cost[2] + 1; }
        = {
            std::cout << "add $ri, $rj, $rk\n";
        };

mul <- reg: MUL( reg, reg )
        { return $cost[1] + $cost[2] + 1; }
        = {
            std::cout << "mul $ri, $rj\n";
            std::cout << "mfhi $rk\n";
        };
```

Figura 17 – Exemplo de Definições de Regra com Custos e Ações

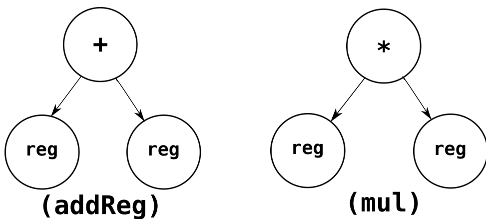

Figura 18 – Árvores Equivalentes às Regras de Exemplo

6.1.3 Definições do Alocador de Registradores

Para poder utilizar o alocador de registradores, é necessário incluir uma seção extra após a definição das regras. Ela ditará algumas características que o alocador utilizará. Atualmente, são quatro diretivas disponíveis:

- `%register IDENTIFIER`: Denota que `IDENTIFIER` é um registrador que será utilizado para alocação. Podem haver n registradores.

- `%spill_reg IDENTIFIER`: Denota que `IDENTIFIER` será um registrador utilizado para *spilling*. A quantidade de registradores de *spilling* deve, obrigatoriamente, ser igual à maior aridade presente nas regras.

- `%set_read STRING`: A *string* é uma instrução na linguagem alvo para leitura de uma variável da memória. Será chamada quando uma variável que foi mandada para a memória for utilizada.

- `%set_write STRING`: Mesmo que `%set_read`, exceto que deve ser uma instrução de escrita na memória, e só será utilizada quando uma variável que foi mandada para a memória for alterada.

Caso o alocador de registradores não seja utilizado, essa seção deve obrigatoriamente ser omitida, de modo que o arquivo ficará com apenas três seções.

6.1.4 Código Auxiliar C++

A terceira (ou quarta) e última parte do arquivo é um código C++ definido pelo usuário, que pode conter quaisquer funções ou estruturas, possivelmente até mesmo uma função `main`.

6.2 YAMG: Algoritmo de Casamento de Padrões

O programa gera algumas funções, variáveis, e *enums*, dos quais vários são internos ao gerador, dentre eles, um `enum` com todas as regras, e outro com os símbolos definidos pelo usuário. As funções principais dependem de quais algoritmos foram especificados no arquivo .md, podendo ser pelo menos uma dentre `matchMaximalMunch(Tree&)`, `matchMinimalMunch(Tree&)`, e `matchDynamicProgramming(Tree&)`, e se encontram dentro do `namespace Yamg`. Essas três funções executam dois passos: primeiramente percorrer a árvore em uma maneira *bottom-up*, invocando, para cada nó, uma função interna chamada `matchTree`, que encontra todos os possíveis casamentos daquela árvore, e os coloca em um vetor naquele nó, junto do custo do casamento.

A segunda etapa é uma passagem *top-down* da árvore, que a percorre realizando os casamentos de menor custo a partir da raiz. Naturalmente, os padrões mais abaixo são reduzidos primeiro, afinal a operação denotada por uma regra depende do resultado das regras filhas. Considere, por exemplo, a árvore da Figura 19, que representa uma adição entre uma variável e o resultado de uma multiplicação. Fica claro aqui que é necessário realizar a operação mais abaixo (a multiplicação) primeiro, uma vez que a operação da raiz depende dela.

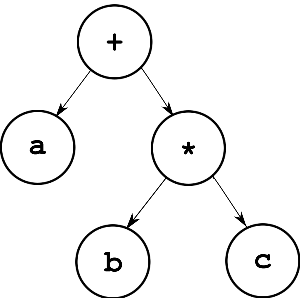

Figura 19 – Árvore mostrando a dependência entre operações

O programa gera também outras estruturas para conseguir realizar o casamento de padrões. São elas:

- A tabela de estados, `recognition_table[][]`, que diz, dado o estado atual e o símbolo do nó filho, para qual estado deve-se seguir;

- As variáveis de controle `TERMINALS_START`, `TERMINALS_END`, `NON_TERMINALS_START`, e `NON_TERMINALS_END`, que dizem onde, no `enum` de símbolos, começam e terminam os símbolos terminais e não-terminais;

- A função `isFinalState(int state)`, que retorna o par (regra, custo) ao qual aquele casamento está relacionado;

- Alguns *aliases* para melhorar a legibilidade do código gerado.

6.3 Interface do Programa

Como dito anteriormente, o usuário deve implementar a classe `Tree`, que estende `YamgTree`, e implementar o método `readTree(Tree&)`. Mas, além disso, é possível adicionar outros métodos e membros na classe, que podem ser utilizados nas ações definidas nas regras, além de poder acessar todos os métodos para criação e manipulação dos nós.

Todas as funções e variáveis descritas anteriormente também estão disponíveis para utilização do usuário, embora seja necessário apenas invocar a função de casamento sobre a raiz da árvore.

Há ainda mais uma ferramenta na interface com o programa, que seriam as diretivas `$cost[n]`, `$[n]`, `$node[n]`, que podem ser utilizadas tanto nas ações das regras, quanto na definição de custo de uma regra. n pode assumir qualquer valor inteiro entre 0 e o número de nós em uma árvore, sendo 0 a raiz da árvore, e os números seguintes cada nó da árvore na ordem em que eles aparecem em sua representação pré-fixa.

Seja, por exemplo, uma regra da forma `reg: ADD(reg,CONST)`, onde `reg` é um não-terminal, e os outros símbolos são terminais. Neste caso, `$cost[0]` se refere ao símbolo ADD, enquanto `$cost[1]`, e `$cost[2]` se referem, respectivamente, ao `reg` na árvore, e `CONST`. Assim, definir o custo dessa regra como `return $cost[1] + 1`, retornaria para a regra um custo 1 somado ao custo do nó `reg`.

A diretiva `$cost[n]` retorna o custo do respectivo nó. O valor dessa diretiva depende do algoritmo de casamento selecionado. Caso a Programação Dinâmica esteja sendo usada, ele retorna a função de custo definida pelo usuário, caso o *Maximal Munch* ou *Minimal Munch* estejam sendo usados, o tamanho da árvore é retornado.

A diretiva `$[n]` é uma maneira simples de pegar o nome do nó selecionado, facilitando o trabalho de obter o nome de uma variável, ou outras informações significativas que podem ser armazenadas nesse campo. Ela expande para o código `t.getChild(n).getName()`. Por fim, a diretiva `$node[n]` retorna uma referência para o nó em si, sendo possível utilizar

```
                     ADD  CONST  reg
/* Estado 0 */  |    1      0     0   |
/* Estado 1 */  |    0      0     2   |
/* Estado 2 */  |    0      3     0   |
/* Estado 3 */  |    0      0     0   |
```

Figura 20 – Autômato da Regra `ADD(reg,CONST)`

```
                     ADD  CONST  reg
/* Estado 2 */  |    0      3     4   |
/* Estado 4 */  |    0      0     0   |
```

Figura 21 – Autômato com a Regra `ADD(reg,reg)` Adicionada

seus atributos ou chamar funções da árvore, como por exemplo `$node[0].setName("$t0")`, ou quaisquer outras funções.

6.4 Análise do Código de Entrada

O primeiro passo para a construção do Yamg é inserir cada um dos terminais e não-terminais definidos em um vetor, para poder verificar se os símbolos nas árvores estão corretos, e posteriormente, criar um `enum` de símbolos no arquivo gerado.

O passo seguinte é transformar cada uma das árvores das regras em estruturas que o programa compreenda, para poder manipulá-las e, com base nelas, criar o autômato. A regra `ADD(reg,CONST)`, por exemplo, geraria um nó do tipo `ADD`, com um filho do tipo `reg` na posição zero, e um filho do tipo `CONST` na posição um. Um autômato para essa regra ficaria da maneira apresentada na Figura 20, com o estado três sendo o estado final que casa a regra.

Com a adição da regra add com registrador `ADD(reg,reg)`, o autômato seria similar, com a adição de um estado de aceitação e uma alteração no estado 2, como mostrado na Figura 21, onde a leitura de um `reg` levaria ao estado 4, que também é um estado final, mas casa essa nova regra.

Esse processo é feito para cada uma das regras, obtendo-se ao final um autômato completo, que casa cada regra a partir do nó raiz. Para saber se um estado é final, o programa possui duas funções. A primeira `isFinalState(int state)`, que passa o estado por um *switch* e retorna verdadeiro caso ele seja final. A segunda função, chamada `finalState(int state, Tree& t)`, também passa o estado por um *switch*, mas retorna um par (regra, custo), onde a regra é o valor da regra no `enum`, como por exemplo `Rules::addReg`, e o custo é o retorno da função `cost(Tree& t)`, que realiza o cálculo

com base na função definida pelo usuário.

A função de casamento de padrões é a `label(Tree& t)`, que realiza uma análise *bottom-up* da árvore, casando todos os padrões possíveis para o nó mais abaixo e à esquerda, seus nós irmãos, seu nó pai, e assim por diante. Quando um casamento é encontrado, o estado final referente à regra é inserido em um vetor, e assim que todos os padrões foram casados, a regra de menor custo desse vetor é selecionada para representar o nó.

`matchTree(Tree&, StateArray& states, StateArray& final_states)` é a função auxiliar usada para realizar o casamento dos padrões. Nela, todos os possíveis símbolos de um nó são considerados, ou seja, ele pode admitir o próprio valor, como por exemplo `CONST`, tanto quanto um não-terminal pelo qual ele pode ser reduzido, como `reg`. O autômato é então percorrido, para cada estado atual, procurando um caminho possível para cada símbolo.

Para exemplificar, considere uma chamada para a função `label` com a árvore `ADD(CONST,CONST)`, e o autômato da Figura 22. O primeiro nó a ser casado seria o `CONST` esquerdo com o padrão de registrador, realizando o mesmo processo para o outro nó folha, enquanto para o `ADD`, a função `matchTree` realiza uma chamada recursiva, passando o estado 1 e o filho esquerdo como parâmetros. Nessa segunda chamada, a análise será feita a partir do estado 1, e os possíveis símbolos serão `CONST` e `reg`, ou seja, o valor original do nó, tal como o que ele foi casado. Essa chamada então retorna os estados 2 e 4, e, a partir da primeira chamada, outra recursão é feita para o nó esquerdo, desta vez com os estados retornados como parâmetro.

Nesta nova chamada, os símbolos a serem analisados são `CONST` e `reg` novamente. Para o estado 2, `CONST` leva ao estado 3, e `reg` leva ao estado 6, enquanto para o estado 4, `reg` leva a 5. Eles são então retornados para a chamada inicial, com os estados 3, 5, e 6, que são todos finais e não levam a nenhum outro. Desta maneira, a análise em `ADD` foi realizada, casando os padrões `addConstRight`, `addConstLeft`, e `addReg`, onde as duas primeiras regras possuem custo 2, e a terceira possui custo 3, portanto a escolha de um casamento ótimo é arbitrária entre os `add`s de custo 2. Na execução do programa, o padrão `addConstRight` foi escolhido.

As funções descritas nessa seção são internas ao gerador, e não devem ser chamadas explicitamente. Elas foram apresentadas aqui por propósitos explicativos, para compreensão de seu funcionamento.

```
              ADD  CONST  reg
/* 0 */  |   1     -     -   |
/* 1 */  |   -     4     2   |
/* 2 */  |   -     3     6   |
/* 3 */  |   -     -     -   | /* addConstRight */
/* 4 */  |   -     -     5   |
/* 5 */  |   -     -     -   | /* addConstLeft */
/* 6 */  |   -     -     -   | /* addReg */
```

Figura 22 – Autômato de Reconhecimento de Três Regras `ADD`

7 YAMG: ALOCADOR DE REGISTRADORES

Utilizando-se apenas a ferramenta principal é possível realizar geração de código, porém, ao se gerar código de máquina, quais registradores serão utilizados? É possível que em um programa pequeno, cada variável seja designada à um registrador durante a etapa de análise da AST, mas uma ferramenta flexível deveria ser capaz de oferecer uma maneira de fazer isso automaticamente.

Para esse fim, o programa possui um alocador de registradores embutido, que utiliza o algoritmo de *Linear Scan* [24] visto no Capítulo 4. Esse alocador é uma implementação simples e concisa desse algoritmo, completamente integrada com o gerador de código, perfeita para testar e prototipar a linguagem sendo gerada. Ele não é eficaz como outras implementações especializadas e bem desenvolvidas, mas sua facilidade de uso e integração com o programa sem a necessidade de configurações adicionais o torna perfeito para um ambiente de desenvolvimento, e mesmo para *back-ends* onde o foco não seja desempenho.

7.1 Representações das Instruções

A interface do alocador possui duas classes: `Instruction`, que corresponde à representação de uma instrução, e `RegAlloc`, responsável pela alocação em si.

7.1.1 Instruções

A classe `Instruction` é composta pelo *template* de uma instrução, ou seja, uma instrução da linguagem alvo com os operadores incompletos, para serem substituídos posteriormente. Existem dois *templates* que podem ser usados em uma instrução:

- `%o`: representa uma variável que será substituída por um registrador durante a alocação;
- `%c`: representa uma constante, que será substituída por uma respectiva constante armazenada em uma lista;

A classe possui duas listas, a primeira de `OperandTypes`, onde um `OperandType` é o nome de uma variável junto com uma *flag* que indica se ela está sendo usada para escrita ou apenas para leitura. Essa informação é utilizada no caso da variável ter sido enviada à memória, para que a instrução de escrita na memória possa ser chamada. A segunda lista contém as constantes que substituirão os `%c`.

```
Instruction{
    "addi %o, %o, %c",
    {
        {"var1", YAMG_WRITEABLE_OPERAND},
        {"var2"}
    },
    { 7 }
}
```

Figura 23 – Criação de uma Instância da Classe `Instruction`

Para exemplificar, a linha de código `var1 = var2 + 7` seria representada, em *mips*, pela *template_string* `"addi %o, %o, %c"`. A lista de `OperandType`s possuiria os elementos `[{name: "var1", write_operand: true}, {name: "var2", write_operand: false}]`, e a lista de constantes conteria o valor 7. No código não é necessário explicitar quando o atributo `write_operand` é falso, uma vez que esse é seu valor padrão. A Figura 23 demonstra o código para criar uma instância da classe `Instruction` que representa esse exemplo.

Adicionalmente, é implementado um método chamado `printInstruction()` que substitui os *templates* da *string* pelos operadores reais. A instrução acima, passada por esse método, retornaria, possivelmente, a *string* `"addi $s0, $s1, 7`, sendo que os registradores podem ser diferentes com base na decisão do alocador.

7.1.2 Interface do Alocador

A classe `RegAlloc` encapsula uma lista de instruções que pode ser manipulada através da interface da classe. Seu principal propósito é ser utilizada dentro das regras, para que a cada regra casada, uma instrução seja adicionada à lista, de modo a gerar o código posteriormente.

A principal função para isso é `newInstruction(Instruction)` que recebe uma instrução e insere-a na classe. A Figura 24 demonstra um exemplo de uso dentro de uma regra:

Como foi dito no Capítulo 5, pela natureza de um *left-to-right bottom-up parser*, as instruções mais abaixo da AST serão as primeiras a serem inseridas na lista, de tal modo que as instruções do código gerado estarão na ordem correta da análise.

Além de inserir instruções, são disponibilizados outros três métodos para a alocação, que lidam especificamente com a pilha. O primeiro é o *clearStack()*, que inserirá na lista uma instrução de limpeza da *stack* que será transformada na respectiva instrução, e com o tamanho correto a ser limpo durante a etapa de alocação. Em *mips*, essa função será transformada na instrução `addi $sp, $sp, x`, onde x é a quantidade de registra-

```
addConst -> reg: ADD(reg,CONST) { return $cost[1] + 1; } = {
    RegAlloc::newInstruction(Instruction{
        {"addi %o, %o, %c"},
        {
            { $[0], YAMG_WRITEABLE_OPERAND }, //Assume-se que na análise da
                //AST, um registrador temporário foi colocado no nome do nó
            { $[1] } //Pega o nome da variável
        },
        { $node[2].value } //Assume-se que `value` é um campo definido pelo
                //usuário, e devidamente inserido na nálise da AST.
    });
};
```

Figura 24 – Regra que Utiliza o Alocador de Registradores

dores mandados para a memória (que estão armazenados na pilha) multiplicado por 4, o tamanho de um registrador em *mips*. Esse método é útil ao se retornar de uma função, quando é necessário limpar a pilha para que o `$sp` possa voltar ao valor que era na função anterior.

As outras duas funções são *storeStack()* e *retrieveStack()*, que respectivamente armazenam e restauram todos os registradores que não estão na memória. Da mesma maneira que a função anterior, elas são inseridas na lista de instruções e substituídas no momento da alocação. Sua utilização mais comum é antes e depois de realizar uma chamada de função, para preservar e posteriormente restaurar o estado da pilha.

No momento de escrita da descrição de máquina não é possível saber quantas variáveis existirão no código analisado, muito menos quantas terão sido mandadas para memória, essas informações requerem uma análise de *live range*, discutida no Capítulo 4, onde seria conhecido quais variáveis estariam vivas em cada ponto do código. Essa análise pode ser feita tanto sobre a AST quanto sobre a lista de instruções gerada, porém a primeira opção faria com que a análise devesse ser implementada pelo programador, enquanto a segunda pode ser feita de maneira automática pelo alocador.

O propósito dessa ferramenta é justamente auxiliar a criação de um gerador de código, portanto esses três métodos (*clearStack()*, *storeStack()*, e *retrieveStack()*) são fornecidos para facilitar essa tarefa, de modo que o usuário só precise utilizá-los de maneira correta, e chamar a função de alocação no momento apropriado.

8 RESULTADOS EXPERIMENTAIS

Este capítulo realiza uma comparação do código gerado pelas ferramentas já estudadas, sendo elas Burg, Iburg, Olive e, claro, o YAMG. Para que a comparação feita seja a melhor e mais justa possível, a gramática utilizada será a mesma, tal como a árvore de entrada. A gramática é um conjunto reduzido de instruções da linguagem MIPS, e o arquivo de entrada que a descreve, na ferramenta Yamg, encontra-se no Apêndice A. Para demonstrar também algumas diferenças do código gerado, e tornar visível as diferenças de legibilidade, a função `label` que foi gerada por cada programa será apresentada na seção respectiva.

8.1 Burg

O primeiro gerador de código, em ordem cronológica, aqui estudado é simples de se escrever e gerar gramáticas, embora compreender seu código gerado seja uma tarefa mais complexa se comparada à saída de geradores futuros. Apesar disso, ainda é uma ferramenta sólida que cumpre bem sua proposta.

Entrada

O arquivo de entrada de todos os programas possui o mesmo formato geral, sendo divididos em três seções com a primeira possuindo o código de usuário que implementa funções e estruturas necessárias e também funções opcionais, além de diretivas declaratórias de terminais e não-terminais, tal como outras declarações específicas de cada ferramenta. A segunda seção possui a definição das regras, tais como suas ações associadas e seus custos. Por fim, a última seção leva código de usuário que implementa quaisquer funções definidas na primeira parte, uma função *main*, ou quaisquer outros pedaços de código que o usuário julgue necessário.

Para o Burg, o arquivo de entrada possui a extensão **.md**, e requer que algumas funções sejam definidas pelo usuário, e até mesmo a própria `struct tree`, `#defines`, além de algumas funções que não são obrigatórias, mas que podem ser úteis, como `_trace(struct tree *t, int ruleno, int cost, int bestcost)`, ou `walk(struct tree *p)`. Essas funções possuem implementações padrões que são providenciadas nas descrições de máquina exemplo, porém elas ainda precisam ser escritas em cada arquivo **.md**.

As únicas diretivas declaratórias da primeira seção são `%term` e `%start`, definindo respectivamente símbolos terminais e o símbolo alvo do casador. As regras são compostas apenas pelo não-terminal alvo, o *template* da árvore, uma *string* que representa a ação daquela regra, seguida de um número inteiro que representa o custo da regra. A seguir há

um exemplo de um arquivo de entrada, excluído o código C. As regras aqui apresentadas serão as mesmas para todos os programas, feitos os devidos ajustes em cada caso.

```
%{
    ... código C
%}
%term ADD   =  1
%term MUL   =  2
%term SUB   =  3
%term DIV   =  4
%term ADDI  =  5
%term SUBI  =  6
%term MEM   =  7
%term STORE =  8
%term MOVE  =  9
%term CONST = 10
%term FP    = 11
%term REG   = 12
%term VAR   = 13
%start stmt

%%

stmt:   reg                            ""
reg:    ADD(reg,reg)                   "ADD ri = rj + rk"      1
reg:    MUL(reg,reg)                   "MUL ri = rj * rk"      1
reg:    SUB(reg,reg)                   "SUB ri = rj - rk"      1
reg:    DIV(reg,reg)                   "DIV ri = rj / rk"      1
reg:    ADD(reg,CONST)                 "ADDI ri = rj + c"      1
reg:    SUB(reg,CONST)                 "SUBI ri = rj - c"      1
reg:    MEM(ADD(reg,CONST))            "LOAD ri = M[rj + c]"   1
reg:    MEM(ADD(CONST,reg))            "LOAD ri = M[rj + c]"   1
reg:    MEM(reg)                       "LOAD ri = M[reg + 0]"  1
stmt:   MOVE(MEM(ADD(reg,CONST)),reg)  "STORE M[rj + c] = ri"  2
stmt:   MOVE(MEM(ADD(CONST,reg)),reg)  "STORE M[rj + c] = ri"  2
stmt:   MOVE(MEM(reg),reg)             "STORE M[rj + 0] = ri"  2
stmt:   MOVE(MEM(reg),MEM(reg))        "MOVEM M[rj] = M[ri]"   3
reg:    FP                             ""
reg:    VAR                            ""
reg:    CONST                          "ADDI ri = r0 + c"      1

%%
```

```
... código C
```

Código Gerado

O Burg gerou um arquivo pequeno, com apenas 601 linhas, com vários comentários que identificam a relação entre vetores e regras, e também para mostrar a qual não-terminal se refere um trecho de código, que facilita analisar o código, sem precisar olhar apenas para números sem significado. Apesar disso, graças à natureza da linguagem e da geração automática de código, tal como escolhas dos próprios autores, o código é, por vezes de difícil leitura, utilizando números brutos, que poderiam ser trocados por um *enum* que lhes daria mais significado, `if's` complexos, e uma função demasiadamente longa. A função *_label* gerada com essa gramática possui 233 linhas, mais do que um terço do programa. O trecho de código abaixo mostra o código gerado ao se encontrar o terminal `ADD`.

```
switch (NODE_OP(t)) {
case 1: /* ADD */
    assert(l && r);
    _label(l);
    _label(r);
    /* 6. reg: ADD(reg, CONST) */
    if (
        NODE_OP(r) == 10 /* CONST */
    ) {
        c = ((struct _state *)(NODE_STATE(l)))->costs[_reg_NT] + 1;
        if (c + 0 < p->costs[_reg_NT]) {
            p->costs[_reg_NT] = c + 0;
            p->rule.reg = 5;
            _closure_reg(t, c + 0);
        }
    }
    /* 2. reg: ADD(reg, reg) */
    c = ((struct _state *)(NODE_STATE(l)))->costs[_reg_NT] + ((struct _state *)(NODE_STAT
    if (c + 0 < p->costs[_reg_NT]) {
        p->costs[_reg_NT] = c + 0;
        p->rule.reg = 1;
        _closure_reg(t, c + 0);
    }
    break;
...
case 10: /* CONST */
```

```
    /* 17. reg: CONST */
    if (1 + 0 < p->costs[_reg_NT]) {
        p->costs[_reg_NT] = 1 + 0;
        p->rule.reg = 12;
        _closure_reg(t, 1 + 0);
    }
    break;
...
}
```

Apesar dos problemas de legibilidade, ele é um programa eficiente. Ao longo de cem execuções com a gramática dada, a média de execução foi de 16,49 milissegundos. Esse tempo considera apenas o tempo que o programa ficou ativo na CPU, deixando de lado o período em que o processo não fora escalonado. O tempo de execução aqui analisado se refere apenas à criação do analisador, e não às análises das árvores em si. Por simplicidade, foram criadas poucas regras para que pudessem ser facilmente portadas entre as ferramentas, desse modo o tempo de execução dos casadores de padrões em si é negligível, portanto sua análise não será feita.

8.2 Iburg

O Iburg é uma evolução do programa anterior, ele mantém uma sintaxe extremamente similar, mas permite maior expressividade de árvores gramáticas, além de implementar várias otimizações que não estavam presentes em seu predecessor.

Entrada

Tal como seu predecessor, o Iburg também necessita de definições prévias para poder funcionar corretamente, dentre elas a `struct tree` que é a representaçção da árvore do programa, as funções `LEFT_CHILD(struct tree)`, `RIGHT_CHILD(struct tree)`, `STATE_LABEL(struct tree)`, e `PANIC`, sendo as três primeiras análogas à *getters*, e a última para *loggar* e possivelmente limpar quaisquer possíveis estados do programa em caso de erro.

Na primeira seção, a única outra diferença para o Burg é que a diretiva `%start` não é mais necessária, o não-terminal alvo é determinado agora pelo não-terminal da primeira regra da lista.

Na descrição das regras, a ação não aparece explicitamente, ao invés disso é inserido, após o padrão de árvore, um símbolo de igual seguido de um número inteiro. Esse número representa o *número externo da regra*, e é utilizado para identificar o padrão ca-

```
...
%%
stmt:   reg                         =  1    ;
...
reg:    ADD(reg,CONST)              =  6 (1);
%%
...
```

Figura 25 – Código de Entrada no Iburg

sado, sendo trabalho do usuário tratar esse número e executar a ação correspondente à regra.

Após esse número vêm, opcionalmente, um vetor de inteiros não-negativos envoltos por parênteses. Esse vetor representa o custo de se casar aquela regra (zero caso o vetor seja omitido), e na grande maioria dos casos apenas o primeiro elemento é utilizado.

Na Figura 25 se encontra o mesmo código de entrada apresentado na seção do Burg, mostrando apenas duas regras para ressaltar a diferença entre as sintaxes. O restante da entrada foi adaptada de acordo.

Código Gerado

O código gerado pelo Iburg foi marginalmente maior do que o de seu predecessor, com 653 linhas, e também possui vários comentários em cada vetor para mostrar o significado de cada índice, além de comentários na função de rotulamento para identificar qual símbolo foi encontrado, tal como as possíveis regras que aquele símbolo casa. Do mesmo modo que o Burg porém, todas as relações entre regras e significados são feitas puramente através de números sem identificadores, de modo que dificulta a leitura do código, principalmente caso seja necessário depurá-lo. Porém, depois de se familiarizar com o código gerado isso deixa de ser um problema, e essa familiarização não é um processo longo.

Na Figura 26 encontra-se o mesmo código mostrado na seção do Burg, a ação tomada pela função *burm_state* quando um símbolo `ADD` é encontrado. É possível observar que, embora nomes diferentes sejam usados, e o código gerado pelo Burg tenha duas linhas a mais pois foram inseridos `\n`s nos comentários das regras, ambos os códigos são exatamente os mesmos, realizando as mesmas comparações e atribuições.

A função da qual o trecho da Figura 26 foi retirado, chamada *burm_state*, possui 255 linhas, maior do que a função *_label* do Burm. Isso acontece pois os programas lidam de maneira diferente com padrões com um único nó. O Burg simplesmente sai do *switch*, enquanto o Iburg cria e retorna uma *struct* interna do programa. Quanto mais regras de nó único estiverem presentes, mais essa disparidade aumenta.

```
switch (op) {
case 1: /* ADD */
    assert(l && r);
    if ( /* reg: ADD(reg,CONST) */
        r->op == 10 /* CONST */
    ) {
        c = l->cost[burm_reg_NT] + 1;
        if (c + 0 < p->cost[burm_reg_NT]) {
            p->cost[burm_reg_NT] = c + 0;
            p->rule.burm_reg = 5;
            burm_closure_reg(p, c + 0);
        }
    }
    { /* reg: ADD(reg,reg) */
        c = l->cost[burm_reg_NT] + r->cost[burm_reg_NT] + 1;
        if (c + 0 < p->cost[burm_reg_NT]) {
            p->cost[burm_reg_NT] = c + 0;
            p->rule.burm_reg = 1;
            burm_closure_reg(p, c + 0);
        }
    }
    break;
...
case 10: /* CONST */
    {
        static struct burm_state z = { 10, 0, 0,
            { 0,
                1,/* stmt: reg */
                1,/* reg: CONST */
            },{
                1,/* stmt: reg */
                13,/* reg: CONST */
            }
        };
        return (STATE_TYPE)&z;
    }
...
}
```

Figura 26 – Código Gerado pelo Iburg

Por todos os benefícios que o Iburg traz, o custo deles vêm no tempo de compilação. Como os próprios autores constataram [15], o Iburg é mais lento do que o Burg, e isso pode ser visto com o tempo que levou para executar o programa cem vezes, que foram 66,85 milissegundos, mais de quatro vezes o tempo do Burg. Essa diferença está relacionada ao tamanho da gramática e a diferença de *overhead* entre os dois programas, sendo que em descrições maiores essa discrepância deve diminuir. Ademais, o tempo de compilação não é um grande problema considerando que as ferramentas não possuem complexidades exponenciais.

8.3 Olive

O Olive traz uma proposta diferente dos dois outros programas apresentados, embora seu objetivo principal ainda seja o mesmo. Todas as ferramentas aqui estudadas têm como propósito principal servirem de geradores de geradores código, porém a flexibilidade do Olive o permite ser utilizado para propósitos mais gerais. Para exemplificar, o Twig, *software* no qual o Olive se baseia, foi utilizado para criar não só geradores de código, mas também um sistema de síntese lógica [20].

Entrada

Embora ele seja um descendente do Twig, a linguagem do Olive muito se assemelha à do Iburg, desde seu arquivo de entrada que também possui extensão **.brg** até o código gerado, que mantém a mesma nomenclatura do Iburg, com funções e estruturas sendo precedidas por *burm_*.

Além das definições que os outros programas esperam, no Olive também é necessário definir uma *struct* chamada *COST* com um membro que representa o tipo do custo, além de uma função de comparação entre custos, e também o custo zero e infinito. Essas definições serão mostradas no trecho de código de entrada.

Junto das diretivas `%term` e `%start`, esse programa também requer a diretiva `%declare<TIPO> NÃO_TERMINAL`, onde TIPO é o tipo de dado que aquele não-terminal possui.

A última diferença no arquivo de entrada são as regras, onde custos não são mais números inteiros, mas sim um bloco de código C que calcula o custo e o atribui para uma variável, além de possuir outro bloco de código C dedicado para as ações.

Nesses dois blocos de código há variáveis especiais que podem e devem ser usadas, sendo as duas principais são `$cost[n]`, e `$action[n]`. A primeira representa o custo do n-ésimo nó da árvore, e na seção de custos, deve-se atribuir o valor para a variável especial `$cost[0]`. A segunda variável especial é uma função que chama a ação do n-ésimo nó, e é necessário chamá-las para cada sub-nó da árvore onde é desejado que a ação seja

```
%{
...
    typedef struct COST {
        int cost;
    } COST;
    #define COST_LESS(a,b) ((a).cost < (b).cost)

    static COST COST_INFINITY = { 32767 };
    static COST COST_ZERO     = { 0 };
%}
...
%declare<void> stmt;
%declare<void> reg;

%%

stmt:   reg { $cost[0].cost = $cost[1].cost; }
    = {;
        $action[1]();
     };
 ...
 reg:    ADD(reg,CONST) { $cost[0].cost = $cost[2].cost + 1; }
    = {;
        $action[2]();
        printf("ADDI ri = rj + c\n");
    };
...

%%
...
```

Figura 27 – Parte do Arquivo de Entrada do Olive

realizada. Essas são as diretivas mais comuns, há outras três com outros propósitos que já foram discutidas na Seção 3.3 do Capítulo 3.

Na Figura 27 encontra-se uma parte do arquivo **.brg** utilizado para entrada no Olive.

Código Gerado

Com o maior código gerado até então, o Olive gerou um arquivo de 1142 linhas. Essa diferença se dá, em parte, graças à maior complexidade das ações. O programa gera uma função chamada *reg_action* que aplica um *switch* sobre o número da regra passada para a função, de modo que quanto mais ações houverem e maiores elas forem, maior será o código final. Para a gramática exemplo, a função *reg_action* gerada possui 124 linhas, pois também são adicionadas diretivas `#line` que ajudam a apontar erros no código de

entrada, uma vez que eles são reportados no arquivo de entrada **.brg**.

A função de rotulamento *burm_label* é marginalmente maior do que a gerada pelo Iburg, com 277 linhas. É possível ver no código da Figura 28 que essa diferença se dá pois cada *case* da função é um pouco maior que os do Iburg, mas ainda sim os códigos fazem a mesma coisa, com a notável diferença de que no Olive a recursão é feita na própria função *burm_label*, enquanto no Iburg a função *burm_state* apenas retorna o estado do nó, e a recursão é feita em outra função.

```
  switch(op){
case 0: /* ADD */
    s=burm_alloc_state(u,op,arity);
    SET_STATE(u,s);
    k=s->kids;
    children=GET_KIDS(u);
    for(i=0;i<arity;i++)
      k[i]=burm_label1(children[i]);
    if (   /* reg: ADD(reg,CONST) */
      k[0]->rule.burm_reg &&
      k[1]->op == 9 /* CONST */
    ) {
      if(burm_cost_code(&c,5,s) && COST_LESS(c,s->cost[burm_reg_NT])) {
        s->cost[burm_reg_NT] = c ;
        s->rule.burm_reg = 5;
        burm_closure_reg(s, c );
      }
    }
    if (   /* reg: ADD(reg,reg) */
      k[0]->rule.burm_reg &&
      k[1]->rule.burm_reg
    ) {
      if(burm_cost_code(&c,1,s) && COST_LESS(c,s->cost[burm_reg_NT])) {
        s->cost[burm_reg_NT] = c ;
        s->rule.burm_reg = 1;
        burm_closure_reg(s, c );
      }
    }
    break;
...
case 9: /* CONST */
#ifdef LEAF_TRAP
    if(s=LEAF_TRAP(u,op))
      return s;
#endif
    s=burm_alloc_state(u,op,arity);
    SET_STATE(u,s);
    k=0;
    {   /* reg: CONST */
      if(burm_cost_code(&c,16,s) && COST_LESS(c,s->cost[burm_reg_NT])) {
        s->cost[burm_reg_NT] = c ;
        s->rule.burm_reg = 12;
        burm_closure_reg(s, c );
      }
    }
    break;
...
}
```

Figura 28 – Função *burm_label* Gerada pelo Olive

Regras que possuem um único nó também são um pouco diferentes, com uma verificação da definição de um *LEAF_TRAP*, cuja presença causa um retorno prematuro da função. Infelizmente, essa função não está presente no manual nem nos exemplos,

portanto não é possível dizer com certeza qual seria seu propósito.

O tempo médio de compilação do programa ao longo de cem execuções foi de 81,47 milissegundos, acima do Iburg, embora os próprios autores constatem [20], que a principal diferença de performance entre as ferramentas se dê na análise de árvores, uma vez que o Olive realiza a programação dinâmica durante a execução do casador de padrões. Isso se dá pela utilização de custo dinâmico, que torna impossível aplicação da teoria BURS, apresentada no Capítulo 3.

8.4 YAMG

Embora C e C++ sejam duas linguagens diferentes, coloquialmente é possível dizer que C++ é um superconjunto de C, de modo que praticamente qualquer código escrito na primeira linguagem pode ser compilado pela segunda, necessitando em alguns casos de pequenas alterações. Desse modo é possível escrever código em C++ da mesma maneira que se escreve em C. O YAMG poderia ter sido escrito dessa maneira, porém assim não seria possível tirar vantagem de tantas melhorias que foram adicionadas à linguagem ao longo dos anos, que permitem maior segurança e legibilidade com menos esforço.

Entrada

O arquivo de entrada foi desenvolvido à imagem do arquivo do Olive, de modo que seja fácil portar de um programa para o outro. Apesar disso, o código C++ que o usuário precisa implementar é completamente diferente. É necessário apenas implementar uma classe chamada *Tree* que estende uma classe própria da ferramenta, e alguns métodos dessa classe.

Tal como os outros programas, é necessário declarar todos os terminais através da diretiva `%term`, e, tal como no Olive, é necessário definir os não-terminais, exceto que no YAMG a diretiva para isso é `%nonterm`. Ademais é possível especificar quais algoritmos serão utilizados com `%alg`.

As regras se assemelham muito às do Olive, com as diferenças das variáveis especiais, e que no YAMG cada regra possui um nome. Há três variáveis aqui disponíveis, são elas: `$cost[n]`, retorna o menor custo de casar o n-ésimo nó; `$[n]`, retorna o nome do n-ésimo nó; e `$node[n]`, que retorna o próprio nó n, sendo possível manipulá-lo diretamente, tal como acessar quaisquer métodos nele implementados.

Como foi extensivamente descrito no Capítulo 7, há uma terceira seção opcional que deve ser declarada quando se utiliza o alocador de registradores. O exemplo aqui mostrado omitirá essa seção, porém o arquivo no Apêndice E contém uma demonstração completa dessa parte.

```
{
    class Tree: public YamgTree<Tree, UserAst> {
        public:

        Tree(){ ... }
        Yamg::User_Symbols
            readUserSymbol(std::string str) override { ... }
        Node_type readNodeType(std::string str) override { ... }
        void readTree(UserAst) override { ... }
        ...
    }
    ...
}
...
%term ADD
%nonterm reg

%%

...
statement <- stmt: reg { return $cost[0]; } = {
    std::cout << "Statement\n";
};
...
addConstRight <- reg: ADD(reg, CONST) { return $cost[1] + 1; } = {
    std::cout << "addi $ri, $rj, c\n";
};
...

%%

...
```

Figura 29 – Arquivo de Entrada do YAMG

Código Gerado

Em comparação com as outras ferramentas, o YAMG gera um código maior do que o Burg e Iburg, porém ainda menor que o código emitido pelo Olive. Com 749 linhas, uma das principais razões pela qual essa ferramenta tem quase 400 linhas a menos que o Olive é o uso de tabelas para codificação de estados. Enquanto nos outros programas o autômato de reconhecimento é representado por código bruto, com *if's* e *switch's* como demonstrado nas Figuras 26 e 28, enquanto o YAMG utiliza uma matriz onde linhas representam o estado, e colunas codificam o símbolo lido. O elemento em uma dada posição indica qual o próximo estado.

Para a gramática já apresentada, foi gerada uma matriz de 39 estados por 11 símbolos, sendo alguns deles símbolos usados internamente pelo gerador. As duas funções

principais utilizadas no casamento de padrões são *_label* e *matchTree*, e elas são geradas estaticamente, portanto elas não alteram o tamanho do código.

```
    /* Classe do usuário */
class Tree;

namespace Yamg{
        /* Enums */
    enum class Rules {
        null = -1,
        statement = 0,
        addReg = 1,
        ...
        variable = 15,
        constant = 16,
    };

    enum class User_Symbols {
        null = -1,
        ADD = 0,
        ...
        VAR = 8,
        reg = 9,
        stmt = 10,
    };

        /* Aliases */
    using cost_t = int;
    using rule_number_t = Rules;
    using MyPair = std::pair<rule_number_t, cost_t>;
    using SymbolArray = std::vector<int>;
    using StateArray  = std::vector<int>;
    using RuleLimit_t = std::pair<int, User_Symbols>;

        /* Variáveis */
    extern const int infinity;
    const int TERMINALS_START     = 0;
    const int TERMINALS_END       = 8;
    const int NON_TERMINALS_START = 9;
    const int NON_TERMINALS_END   = 10;
    const std::map<Rules, int> Rules_Non_Terminals{
        { Rules::statement, 10 },
        ...
        { Rules::constant, 9 },
    };

        /* Funções */
    void matchDynamicProgramming(Tree&);
    void matchMaximalMunch(Tree&);
    void matchMinimalMunch(Tree&);
}
```

Figura 30 – *Header* gerado pelo YAMG

Ao longo de cem execuções, o tempo médio de compilação foi muito maior do que as outras ferramentas, alcançando um total de 1,049 segundos. É necessário que mais testes sejam feitos, porém acredita-se que o amplo uso de *templates* na estrutura de árvore seja a principal causa para essa diferença. Apesar disso, é importante lembrar que um grande aspecto está sendo deixado de lado graças à simplicidade das descrições de máquina aqui criadas para exemplo, e esse aspecto é a execução do código em si, cuja medição de performance não foi feita pelas razões já explicadas.

Uma vez que o código emitido pelo YAMG é muito diferente daquele dos outros

geradores, tornando assim muito difícil qualquer comparação, o arquivo **.cpp** não será mostrado. Ao invés disso, será inserido aqui o **.hpp** que representa a interface pública que o programa gerado disponibiliza, que já foi descrita no Capítulo 5.

Resultados do Código Gerado

Como foi apresentado, o YAMG disponibiliza três algoritmos para casamento de padrões, tal como um alocador de registradores automático. Para demonstrar a diferença dos algoritmos foi implementado um exemplo com a linguagem que vêm sendo mostrada nesse capítulo, e a árvore utilizada foi uma já discutida extensivamente para demonstrar os algoritmos no Capítulo 2, a árvore da Figura 8.

Para simplificar, apenas imprimiu-se um pseudo-código MIPS com registradores genéricos, de modo que foco esteja em quais padrões foram casados. O programa foi rodado três vezes, cada uma utilizando um algoritmo diferente. Os blocos abaixo representam a saída dos algoritmos do Maximal Munch, Minimal Munch, e Programação Dinâmica, respectivamente.

É possível observar que os dois primeiros resultados correspondem às Figuras 9 e 10 do Capítulo 3. Essa é uma prova do funcionamento do programa, uma vez que a saída bate com os resultados obtidos durante o estudo teórico de cada um dos algoritmos.

```
addi $ri, $zero, c
mul $ri, $rj, $rk
sub $ri, $rj, c
lw $ri, c($rj)
sw, $ri, c($rj)
```

(a) Maximal Munch

```
addi $ri, $zero, c
mul $ri, $rj, $rk
addi $ri, $zero, c
add $ri, $rj, $rk
addi $ri, $zero, c
sub $ri, $rj, c
add $ri, $rj, $rk
lw, $ri, 0($rj)
sw, $ri, $rj
```

(b) Minimal Munch

```
addi $ri, $zero, c
mul $ri, $rj, $rk
addi $ri, $zero, c
add $ri, $rj, $rk
sub $ri, $rj, c
lw $ri, c($rj)
sw, $ri, $rj
```

(c) Programação Dinâmica

Figura 31 – Código MIPS Gerado no YAMG Pelos Diferentes Algoritmos

É importante observar que para a geração por programação dinâmica, os custos foram ajustados de acordo com a alteração feita ao fim do Capítulo 3, de modo que o casamento do padrão `MEM(ADD(reg,CONST))` fosse custoso demais, forçando assim o algoritmo a casar o nó `ADD` separadamente.

Essa alteração foi feita para que a saída desse algoritmo se diferenciasse dos outros, tornando possível distinguí-la das outras com maior facilidade. Além disso, tal como foi dito no parágrafo anterior, essa alteração prova a corretude do código, se igualando à saída esperada na Figura 14.

Quanto ao alocador de registradores, foi criado um *back-end* para gerar um conjunto mínimo de instruções MIPS. Junto com um analisador semântico criado pela autora na matéria de Compiladores, fez-se um programa mais complexo do que os exemplos aqui apresentados, capaz de ler um código C simplificado, representá-lo na forma de árvore sintática abstrata, e passá-lo pelo gerador de código e finalmente pelo alocador de registradores. O repositório git do YAMG, presente no Anexo B, contém esse programa.

Foram escritos quatro códigos C para serem processados por esse programa. Graças à limitações na implementação do analisador semântico, junto com o grande trabalho que seria fazer *back-end* mais completo, os exemplos criados não foram de alta complexidade, mas o que suficiente para demonstrar a utilização do gerador de código em um ambiente mais realista, além de expor a funcionalidade do alocador de registradores com programas não triviais.

Na Figura 32 encontra-se o programa em C que recebe um número do usuário, calcula seu fatorial, e o imprime na tela. Logo em seguida, na Figura 33, encontra-se esse mesmo programa em MIPS depois de ser processado pelo código descrito nos parágrafos acima.

```
int fat(int n);
int main() {
    int n = 0;
    int f = 0;
    printf("Insira o número cujo fatorial será calculado: ");
    scanf("%d", &n);
    f = fat(n);
    printf("fatorial: ");
    printf("%d", f);
    return 0;
}
int fat(int n) {
    int n1 = n - 1;
    if(n == 1) {
        return 1;
    }
    return n * fat(n1);
}
```

Figura 32 – Código C para Calcular Fatorial

Esse código possui pseudo-instruções MIPS, portanto é necessário executá-lo em um emulador que as aceite. Ele foi executado no programa *MARS - MIPS Assembler and Runtime Simulator* [28] e, assim como os outros, gerou resultado correto.

Os outros exemplos, junto de seu respectivo código C, encontram-se nos Apêndices B, C, e D, e os repositórios do analisador semântico e do YAMG se encontram nos Anexos A e B respectivamente.

```
.data:
__literal_0: .asciiz "Insira o número cujo fatorial será calculado: "
__literal_1: .asciiz "fatorial: "
.text:
main:
    li $s7, 0
    li $s6, 0
    subi $sp, $sp, 4
    sw $a0, 0($sp)
    li $v0, 4
    la $a0, __literal_0
    syscall
    lw $a0, 0($sp)
    addi $sp, $sp, 4
    li $v0, 5
    syscall
    move $s7, $v0
    subi $sp, $sp, 8  #store stack
    sw $s7, 0($sp)  #store stack
    sw $s6, 4($sp)  #store stack
    subi $sp, $sp, 4
    sw $a0, 0($sp)
    move $a0, $s7
    jal fat
    lw $a0, 0($sp)
    addi $sp, $sp, 4
    lw $s6, 4($sp)  #retrieve stack
    lw $s7, 0($sp)  #retrieve stack
    addi $sp, $sp, 8  #retrieve stack
    move $s6, $v0
    subi $sp, $sp, 4
    sw $a0, 0($sp)
    li $v0, 4
    la $a0, __literal_1
    syscall
    lw $a0, 0($sp)
    addi $sp, $sp, 4
    subi $sp, $sp, 4
    sw $a0, 0($sp)
    li $v0, 1
    la $a0, ($s6)
    syscall
    lw $a0, 0($sp)
    addi $sp, $sp, 4
    li $v0, 10
    li $a0, 0
    syscall
```

```
fat:
    subi $sp, $sp, 4
    sw $ra, 0($sp)
    subi $t5, $a0, 1
    move $s5, $t5
    seqi $t6, $a0, 1
    beq $t6, $zero, if_end_1
    li $v0, 1
    # clear stack
    lw $ra, 0($sp)
    addi $sp, $sp, 4
    jr $ra
    if_end_1:
    subi $sp, $sp, 12  #store stack
    sw $s5, 0($sp)  #store stack
    sw $s7, 4($sp)  #store stack
    sw $s6, 8($sp)  #store stack
    subi $sp, $sp, 4
    sw $a0, 0($sp)
    move $a0, $s5
    jal fat
    lw $a0, 0($sp)
    addi $sp, $sp, 4
    lw $s6, 8($sp)  #retrieve stack
    lw $s7, 4($sp)  #retrieve stack
    lw $s5, 0($sp)  #retrieve stack
    addi $sp, $sp, 12  #retrieve stack
    mult $a0, $v0
    mflo $t6
    move $v0, $t6
    # clear stack
    lw $ra, 0($sp)
    addi $sp, $sp, 4
    jr $ra
```

Figura 33 – Código MIPS para Calcular Fatorial, Gerado Automaticamente pelo YAMG

9 CONCLUSÃO

A tradução de código de alto nível para código de máquina é uma tarefa importante e também complexa. O código de máquina de cada arquitetura de CPU é muito diferente, e até mesmo diferentes versões de uma mesma arquitetura trazem mudanças que tornam necessárias a alteração desse tradutor.

Para facilitar a tarefa de criar um *back-end* de um compilador que traduza a AST para o código de máquina foi proposta uma ferramenta que realiza casamento de padrões sobre uma árvore, dada uma descrição de regras dessa linguagem, tal como ações a serem tomadas quando uma regra é encontrada. Dessa maneira, não há a necessidade de criar um tradutor completo, apenas definir a descrição da máquina. Essas regras podem ser criadas, removidas, ou até alteradas com facilidade, tal como as ações.

Quando há uma nova versão de uma arquitetura de processadores só é necessário alterar a descrição da máquina e executar o programa novamente, que o casador de padrões atualizado será gerado automaticamente. Da mesma maneira, se for necessário dar suporte para uma nova linguagem de máquina, é possível deixar as regras intactas e apenas alterar as ações para gerar instruções para a nova arquitetura.

9.1 YAMG

O programa que foi proposto nesse trabalho, *Yet Another Machine code Generator* (YAMG), foi criado utilizando a linguagem de programação C++, aproveitando-se de várias características modernas da linguagem para garantir a maior segurança e facilidade de uso possíveis.

O YAMG utiliza as mesmas técnicas de casamento que se provaram muito efetivas há décadas atrás, tal como uma análise *bottom-up* para rotular os nós com o melhor casamento possível, seguida de uma passagem *top-down* para encontrar a cobertura de custo mínimo, e realizar as ações dos nós selecionados. O programa possui suporte à custos dinâmicos, de modo que o usuário pode especificar funções para o cálculo de custo, ao invés de depender apenas de números fixos.

Outros diferenciais do YAMG, além da linguagem e seus benefícios, são a possibilidade de escolher qual algoritmo será utilizado, dentre os três que foram apresentados e detalhados no Capítulo 2, além de um alocador de registradores fácil de ser utilizado.

Desde a simplicidade de escrita da descrição das regras e ações, a facilidade de trocar o casador de padrões utilizado, até a possibilidade de utilizar o alocador para gerar código *assembly* pronto para ser executado, o programa aqui desenvolvido não só cumpre

sua proposta principal, como também proporciona ao programador utilidades extras para facilitar o desenvolvimento.

9.2 Trabalhos Futuros

Apesar da ferramenta estar completamente funcional, ainda há algumas melhorias que podem ser feitas. Primeiramente, a transição de estados foi feita com tabelas por simplicidade, mas, como foi discutido no Capítulo 3, utilizar código bruto para essa tarefa não resulta em uma queda de performance, e melhora significativamente a legibilidade do código, tornando-o mais fácil de compreender e depurar.

Outra melhoria que poderia ser feita é relacionada com os algoritmos de *Minimal* e *Maximal Munch*. Uma vez que a análise de seu custo é estática (considera-se apenas o tamanho dos padrões), seria possível inserir os custos diretamente nas regras, tomando vantagem da técnica BURS.

Seria possível também implementar outros alocadores de registradores, como um algoritmo de Coloração de Grafos, ou até mesmo um *Linear Scan* com *Second Chance Binpacking*, e permitir que o usuário selecione qual algoritmo ele queira usar, dependendo de sua necessidade.

Nesse trabalho foi desenvolvida uma ferramenta completa capaz de reconhecer e casar padrões com bases em custos definidos pelo usuário, com a possibilidade de associar-se ações e sub-ações com cada padrão. Com ela é possível criar com facilidade *back-ends* para um compilador e gerar código para uma linguagem de máquina desejada. O YAMG pode também ser utilizado de maneira geral para outros propósitos que envolvam casamento de padrões em árvores.

REFERÊNCIAS

[1] APPEL, A. W. *Modern compiler implementation in C*. [S.l.]: Cambridge university press, 2004.

[2] MOGENSEN, T. Æ. *Basics of compiler design*. [S.l.]: Torben Ægidius Mogensen, 2009.

[3] AHO, A. V.; JOHNSON, S. C. Optimal code generation for expression trees. *Journal of the ACM (JACM)*, ACM New York, NY, USA, v. 23, n. 3, p. 488–501, 1976.

[4] GLANVILLE, R. S.; GRAHAM, S. L. A new method for compiler code generation. In: *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. [S.l.: s.n.], 1978. p. 231–254.

[5] LESK, M. E.; SCHMIDT, E. *Lex: A lexical analyzer generator*. [S.l.]: Bell Laboratories Murray Hill, NJ, 1975.

[6] HEURING, V. P. The automatic generation of fast lexical analysers. *Software: Practice and Experience*, Wiley Online Library, v. 16, n. 9, p. 801–808, 1986.

[7] PARR, T. J.; QUONG, R. W. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, Wiley Online Library, v. 25, n. 7, p. 789–810, 1995.

[8] LEVINE, J. R. et al. *Lex & yacc*. [S.l.]: " O'Reilly Media, Inc.", 1992.

[9] JOHNSON, S. C. et al. *Yacc: Yet another compiler-compiler*. [S.l.]: Bell Laboratories Murray Hill, NJ, 1975. v. 32.

[10] CORBETT, R. P. *Static semantics and compiler error recovery*. [S.l.], 1985.

[11] CATTELL, R. Automatic derivation of code generators from machine descriptions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM New York, NY, USA, v. 2, n. 2, p. 173–190, 1980.

[12] BELLMAN, R. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, v. 60, n. 6, p. 503–515, 1954.

[13] AHO, A. V.; GANAPATHI, M.; TJIANG, S. W. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM New York, NY, USA, v. 11, n. 4, p. 491–516, 1989.

[14] AHO, A. V.; CORASICK, M. J. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, ACM New York, NY, USA, v. 18, n. 6, p. 333–340, 1975.

[15] FRASER, C. W.; HANSON, D. R.; PROEBSTING, T. A. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems (LOPLAS)*, ACM New York, NY, USA, v. 1, n. 3, p. 213–226, 1992.

[16] AHO, A. et al. *Compilers: Principles, Techniques, & Tools*. [S.l.]: Pearson/Addison Wesley, 2007. ISBN 9780321486813.

[17] FRASER, C. W.; HENRY, R. R.; PROEBSTING, T. A. Burg: fast optimal instruction selection and tree parsing. *ACM Sigplan Notices*, ACM New York, NY, USA, v. 27, n. 4, p. 68–76, 1992.

[18] PELEGRI-LLOPART, E.; GRAHAM, S. L. *Optimal code generation for expression trees: An application of burs (bottom-up rewrite systems) theory*. [S.l.], 1988.

[19] HATCHER, P. J.; CHRISTOPHER, T. W. High-quality code generation via bottom-up tree pattern matching. In: *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. [S.l.: s.n.], 1986. p. 119–130.

[20] TJIANG, S. W. An olive twig. *Technical report, Synopsys Inc*, 1993.

[21] EL-REWINI, H.; ABD-EL-BARR, M. *Fundamentals of computer organization and architecture*. [S.l.]: John Wiley & Sons, 2005.

[22] GUIDE, P. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, v. 2, n. 11, 2011.

[23] PEREIRA, F. M. Q.; PALSBERG, J. Register allocation by puzzle solving. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. [S.l.: s.n.], 2008. p. 216–226.

[24] POLETTO, M.; SARKAR, V. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM New York, NY, USA, v. 21, n. 5, p. 895–913, 1999.

[25] CHAITIN, G. J. et al. Register allocation via coloring. *Computer languages*, Elsevier, v. 6, n. 1, p. 47–57, 1981.

[26] JOHANSSON, E.; SAGONAS, K. Linear scan register allocation in a high-performance erlang compiler. In: SPRINGER. *International Symposium on Practical Aspects of Declarative Languages*. [S.l.], 2002. p. 101–119.

[27] TRAUB, O.; HOLLOWAY, G.; SMITH, M. D. Quality and speed in linear-scan register allocation. *ACM SIGPLAN Notices*, ACM New York, NY, USA, v. 33, n. 5, p. 142–151, 1998.

[28] VOLLMAR, K.; SANDERSON, P. Mars: an education-oriented mips assembly language simulator. In: *Proceedings of the 37th SIGCSE technical symposium on Computer science education*. [S.l.: s.n.], 2006. p. 239–243.

Apêndices

APÊNDICE A – EXEMPLO DE ARQUIVO DE ENTRADA PARA O YAMG

```
{
#include<iostream>

class Tree: public YamgTree<Tree, std::string>{
    public:

        Tree(){}
        Tree(const std::string& name, const Yamg::User_Symbols user_symbol, const Node_type& type)
            : YamgTree{ name, user_symbol, type }
            {}

        Yamg::User_Symbols readUserSymbol(std::string str) override {
            std::string s = this->name;
            if(s == "ADD")  return Yamg::User_Symbols::ADD;
            if(s == "SUB")  return Yamg::User_Symbols::SUB;
            if(s == "MUL")  return Yamg::User_Symbols::MUL;
            if(s == "MEM")  return Yamg::User_Symbols::MEM;
            if(s == "CONST")return Yamg::User_Symbols::CONST;
            if(s == "MOVE") return Yamg::User_Symbols::MOVE;
            if(s == "FP")   return Yamg::User_Symbols::FP;
            if(s == "VAR")  return Yamg::User_Symbols::VAR;
            if(s == "reg")  return Yamg::User_Symbols::reg;
                            return Yamg::User_Symbols::stmt;
        }
        Node_type readNodeType(std::string str) override {
            std::string s = this->name;
            if(s == "c") return Node_type::constante;
                         return Node_type::constante;
        }

        void readTree(std::string) override {
            Tree root{"MOVE", Yamg::User_Symbols::MOVE, Node_type::constante};
            {
                Tree mem{"MEM", Yamg::User_Symbols::MEM, Node_type::constante};
                Tree add{"ADD", Yamg::User_Symbols::ADD, Node_type::constante};
                Tree mul{"MUL", Yamg::User_Symbols::MUL, Node_type::constante};
                Tree reg{"a", Yamg::User_Symbols::VAR, Node_type::constante};
                Tree const1{"4", Yamg::User_Symbols::CONST, Node_type::constante};
                Tree const2{"0", Yamg::User_Symbols::CONST, Node_type::constante};
                mul.insertChild(reg);
                mul.insertChild(const1);
                add.insertChild(mul);
                add.insertChild(const2);
                mem.insertChild(add);
                root.insertChild(mem);
            }
            {
                Tree mem{"MEM", Yamg::User_Symbols::MEM, Node_type::constante};
                Tree add{"ADD", Yamg::User_Symbols::ADD, Node_type::constante};
                Tree const1{"7", Yamg::User_Symbols::CONST, Node_type::constante};
                Tree sub{"SUB", Yamg::User_Symbols::SUB, Node_type::constante};
                Tree reg{"b", Yamg::User_Symbols::VAR, Node_type::constante};
                Tree const2{"15", Yamg::User_Symbols::CONST, Node_type::constante};
```

```
                sub.insertChild(reg);
                sub.insertChild(const2);
                add.insertChild(const2);
                add.insertChild(sub);
                mem.insertChild(add);
                root.insertChild(mem);
            }
            *this = root;
        }

        int coisa1;
        int coisa2;
};

}

%term ADD
%term SUB
%term MUL
%term DIV
%term MEM
%term CONST
%term MOVE
%term FP
%term VAR
%nonterm reg
%nonterm stmt

%%

statement           <- stmt: reg                              { return $cost[0]; }              = {
↪  std::cout << "Statement\n"; };
addReg              <- reg: ADD(reg, reg)                     { return $cost[1] + $cost[2] + 1; } = {
↪  std::cout << "add $ri, $rj, $rk\n"; };
mul                 <- reg: MUL( reg, reg )                   { return $cost[1] + $cost[2] + 1; } = {
↪  std::cout << "mul $ri, $rj, $rk\n"; };
sub                 <- reg: SUB( reg, reg )                   { return $cost[1] + $cost[2] + 1; } = {
↪  std::cout << "sub $ri, $rj, $rk\n"; };
div                 <- reg: DIV( reg, reg )                   { return $cost[1] + $cost[2] + 1; } = {
↪  std::cout << "div $ri, $rj, $rk\n"; };
addConstRight       <- reg: ADD(reg, CONST)                   { return $cost[1] + 1; }           = {
↪  std::cout << "addi $ri, $rj, c\n"; };
subConst            <- reg: SUB( reg, CONST )                 { return $cost[1] + 1; }           = {
↪  std::cout << "sub $ri, $rj, c\n"; };
loadAddConstRight   <- reg: MEM( ADD( reg, CONST ) )          { return $cost[2] + 1; }           = {
↪  std::cout << "lw $ri, c($rj)\n"; };
loadAddConstLeft    <- reg: MEM( ADD( CONST, reg ) )          { return $cost[3] + 1; }           = {
↪  std::cout << "lw $ri, c($rj)\n"; };
loadReg             <- reg: MEM( reg )                        { return $cost[1] + 1; }           = {
↪  std::cout << "lw, $ri, 0($rj)\n"; };
storeMemAddConstRight <- stmt: MOVE( MEM( ADD(reg,CONST) ), reg ) { return $cost[3] + $cost[5] + 5; } = {
↪  std::cout << "sw, $ri, c($rj)\n"; };
storeMemAddConstLeft  <- stmt: MOVE( MEM( ADD(CONST,reg) ), reg ) { return $cost[4] + $cost[5] + 2; } = {
↪  std::cout << "sw, $ri, c($rj)\n"; };
storeMemReg         <- stmt: MOVE( MEM( reg ), reg )          { return $cost[3] + $cost[2] + 2; } = {
↪  std::cout << "sw, $ri, $rj\n"; };
moveMemMem          <- stmt: MOVE( MEM( reg ), MEM( reg ) )   { return $cost[2] + $cost[4] + 3; } = {
↪  std::cout << "move, $ri, $rj\n"; };
framePointer        <- reg: FP                                { return 0; }                      = { ;
↪  };
```

```
variable              <- reg: VAR                         { return 0; }                    = { ;
↪  };
constant              <- reg: CONST                       { return 1; }                    = {
↪  std::cout << "addi $ri, $zero, c\n"; };

%%

%register $s0
%register $s1
%register $s2
%register $s3
%register $s4
%register $s5
%register $s6
%register $s7

%spill_register $t0
%spill_register $t1
%spill_register $t2

%set_read "lw %o, %o($sp)"
%set_write "sw %o, %o($sp)"

%%

{

using namespace Yamg;

int main(){
    {
        Tree t{};
        t.readTree("");
        std::cout << "DP:\n";
        matchDynamicProgramming(t);
    }
    {
        Tree t{};
        t.readTree("");
        std::cout << "\nMax Munch:\n";
        matchMaximalMunch(t);
    }
    {
        Tree t{};
        t.readTree("");
        std::cout << "\nMin Munch:\n";
        matchMinimalMunch(t);
    }
}

}
```

APÊNDICE B – EXEMPLO MIPS - INVERSÃO DE VETOR

```
int main() {
    int v[8];
    int i=0;
    int aux=0;
    int aux2 = 0;
    int max = 8;

    v[0] =  2;
    v[1] =  9;
    v[2] = 15;
    v[3] = 22;
    v[4] = 30;
    v[5] = 37;
    v[6] = 41;
    v[7] = 90;

    for(i=0;i<4;i++){
        aux = v[i];
        aux2 = v[max - i - 1];
        v[i] = aux2;
        v[max - i - 1] = aux;
    }

    for(i=0; i<max; i++) {
        aux = v[i];
        if(aux%3) {
            printf("%d", aux);
            printf("\n");
        } else {
            printf("MD3\n");
        }
    }

    return 0;
}

.data:
__literal_1: .asciiz "MD3\n"
__literal_0: .asciiz "\n"

.text:

main:
subi $sp, $sp, 32
move $s7, $sp
li $s6, 0
li $s5, 0
li $s4, 0
li $s3, 8
lw $t5, 0($s7)
li $t6, 2
li $t5, 0
addi $t5, $s7, 0
```

```
sw $t6, ($t5)
lw $t4, 4($s7)
li $t5, 9
li $t4, 4
addi $t4, $s7, 4
sw $t5, ($t4)
lw $t3, 8($s7)
li $t4, 15
li $t3, 8
addi $t3, $s7, 8
sw $t4, ($t3)
lw $t7, 12($s7)
li $t3, 22
li $t7, 12
addi $t7, $s7, 12
sw $t3, ($t7)
lw $t6, 16($s7)
li $t7, 30
li $t6, 16
addi $t6, $s7, 16
sw $t7, ($t6)
lw $t5, 20($s7)
li $t6, 37
li $t5, 20
addi $t5, $s7, 20
sw $t6, ($t5)
lw $t4, 24($s7)
li $t5, 41
li $t4, 24
addi $t4, $s7, 24
sw $t5, ($t4)
lw $t3, 28($s7)
li $t4, 90
li $t3, 28
addi $t3, $s7, 28
sw $t4, ($t3)
for_init_1:
li $s6, 0
for_check_1:
slti $t5, $s6, 4
beq $t5, $zero, for_end_1
j for_body_1
for_update_1:
addi $s6, $s6, 1
j for_check_1
for_body_1:
sll $t6, $s6, 2
add $t6, $s7, $t6
lw $t6, ($t6)
move $s5, $t6
sub $t5, $s3, $s6
li $t3, 1
sub $t7, $t5, $t3
sll $t5, $t7, 2
add  $t5, $s7, $t5
lw $t5, ($t5)
move $s4, $t5
sll $t4, $s6, 2
add $t4, $s7, $t4
lw $t4, ($t4)
```

```
sll $t4, $s6, 2
add $t4, $s7, $t4
sw $s4, ($t4)
sub $t3, $s3, $s6
li $t7, 1
sub $t5, $t3, $t7
sll $t3, $t5, 2
add  $t3, $s7, $t3
lw $t3, ($t3)
sll $t3, $t5, 2
add $t3, $s7, $t3
sw $s5, ($t3)
j for_update_1
for_end_1:
for_init_2:
li $s6, 0
for_check_2:
slt $t4, $s6, $s3
beq $t4, $zero, for_end_2
j for_body_2
for_update_2:
addi $s6, $s6, 1
j for_check_2
for_body_2:
sll $t5, $s6, 2
add $t5, $s7, $t5
lw $t5, ($t5)
move $s5, $t5
li $t6, 3
div $t6, $s5, $t6
mfhi $t6
beq $t6, $zero, if_else_3
subi $sp, $sp, 4
sw $a0, 0($sp)
li $v0, 1
la $a0, ($s5)
syscall
lw $a0, 0($sp)
addi $sp, $sp, 4
subi $sp, $sp, 4
sw $a0, 0($sp)
li $v0, 4
la $a0, __literal_0
syscall
lw $a0, 0($sp)
addi $sp, $sp, 4
j if_end_3
if_else_3:
subi $sp, $sp, 4
sw $a0, 0($sp)
li $v0, 4
la $a0, __literal_1
syscall
lw $a0, 0($sp)
addi $sp, $sp, 4
if_end_3:
j for_update_2
for_end_2:
li $v0, 10
li $a0, 0
```

```
syscall
```

APÊNDICE C – EXEMPLO MIPS - MULTIPLICAÇÃO DE VETOR

```
int main() {
    int v[5];
    int resultado = 1;
    int i=0;
    int n=5;

    v[0] = 7;
    v[1] = 5;
    v[2] = 33;
    v[3] = 10;
    v[4] = 2;

    for(i=0; i<n; i++){
        resultado = resultado * v[i];
    }
    printf("soma: ");
    printf("%d", resultado);
    return 0;
}

.data:
__literal_0: .asciiz "soma: "

.text:

main:
subi $sp, $sp, 20
move $s7, $sp
li $s6, 1
li $s5, 0
li $s4, 5
lw $t3, 0($s7)
li $t4, 7
li $t3, 0
addi $t3, $s7, 0
sw $t4, ($t3)
lw $t7, 4($s7)
li $t3, 5
li $t7, 4
addi $t7, $s7, 4
sw $t3, ($t7)
lw $t6, 8($s7)
li $t7, 33
li $t6, 8
addi $t6, $s7, 8
sw $t7, ($t6)
lw $t5, 12($s7)
li $t6, 10
li $t5, 12
addi $t5, $s7, 12
sw $t6, ($t5)
lw $t4, 16($s7)
li $t5, 2
```

```
li $t4, 16
addi $t4, $s7, 16
sw $t5, ($t4)
for_init_1:
li $s5, 0
for_check_1:
slt $t6, $s5, $s4
beq $t6, $zero, for_end_1
j for_body_1
for_update_1:
addi $s5, $s5, 1
j for_check_1
for_body_1:
sll $t4, $s5, 2
add $t4, $s7, $t4
lw $t4, ($t4)
mult $s6, $t4
mflo $t7
move $s6, $t7
j for_update_1
for_end_1:
subi $sp, $sp, 4
sw $a0, 0($sp)
li $v0, 4
la $a0, __literal_0
syscall
lw $a0, 0($sp)
addi $sp, $sp, 4
subi $sp, $sp, 4
sw $a0, 0($sp)
li $v0, 1
la $a0, ($s6)
syscall
lw $a0, 0($sp)
addi $sp, $sp, 4
li $v0, 10
li $a0, 0
syscall
```

APÊNDICE D – EXEMPLO MIPS - FIBONACCI

```
int fib(int n);
int main() {
    int n = 9;
    int resultado = 0;
    printf("Insira o número: ");
    scanf("%d", &n);
    resultado = fib(n);
    printf("fibonacci: ");
    printf("%d", resultado);
    return 0;
}

int fib(int n) {
    int n1 = n - 1;
    int n2 = n - 2;
    int fib1 = 0;
    int fib2 = 0;
    if(n<=1) { return n; }
    fib1 = fib(n1);
    fib2 = fib(n2);
    return fib1 + fib2;
}
```

```
.data:
__literal_0: .asciiz "Insira o número: "
__literal_1: .asciiz "fibonacci: "

.text:

main:
li $s7, 9
li $s6, 0
subi $sp, $sp, 4
sw $a0, 0($sp)
li $v0, 4
la $a0, __literal_0
syscall
lw $a0, 0($sp)
addi $sp, $sp, 4
li $v0, 5
syscall
move $s7, $v0
subi $sp, $sp, 8  #store stack
sw $s7, 0($sp)  #store stack
sw $s6, 4($sp)  #store stack
subi $sp, $sp, 4
sw $a0, 0($sp)
move $a0, $s7
jal fib
lw $a0, 0($sp)
addi $sp, $sp, 4
lw $s6, 4($sp)  #retrieve stack
lw $s7, 0($sp)  #retrieve stack
addi $sp, $sp, 8  #retrieve stack
```

```
move $s6, $v0
subi $sp, $sp, 4
sw $a0, 0($sp)
li $v0, 4
la $a0, __literal_1
syscall
lw $a0, 0($sp)
addi $sp, $sp, 4
subi $sp, $sp, 4
sw $a0, 0($sp)
li $v0, 1
la $a0, ($s6)
syscall
lw $a0, 0($sp)
addi $sp, $sp, 4
li $v0, 10
li $a0, 0
syscall

fib:
subi $sp, $sp, 4
sw $ra, 0($sp)
subi $t5, $a0, 1
move $s6, $t5
subi $t4, $a0, 2
move $s4, $t4
li $s3, 0
li $s2, 0
sle $t4, $a0, 1
beq $t4, $zero, if_end_1
move $v0, $a0
# clear stack
lw $ra, 0($sp)
addi $sp, $sp, 4
jr $ra
if_end_1:
subi $sp, $sp, 24  #store stack
sw $s7, 0($sp)  #store stack
sw $s6, 4($sp)  #store stack
sw $s4, 8($sp)  #store stack
sw $s5, 12($sp)  #store stack
sw $s3, 16($sp)  #store stack
sw $s2, 20($sp)  #store stack
subi $sp, $sp, 4
sw $a0, 0($sp)
move $a0, $s6
jal fib
lw $a0, 0($sp)
addi $sp, $sp, 4
lw $s2, 20($sp)  #retrieve stack
lw $s3, 16($sp)  #retrieve stack
lw $s5, 12($sp)  #retrieve stack
lw $s4, 8($sp)  #retrieve stack
lw $s6, 4($sp)  #retrieve stack
lw $s7, 0($sp)  #retrieve stack
addi $sp, $sp, 24  #retrieve stack
move $s3, $v0
subi $sp, $sp, 24  #store stack
sw $s7, 0($sp)  #store stack
sw $s6, 4($sp)  #store stack
```

```
sw $s4, 8($sp)  #store stack
sw $s5, 12($sp)  #store stack
sw $s3, 16($sp)  #store stack
sw $s2, 20($sp)  #store stack
subi $sp, $sp, 4
sw $a0, 0($sp)
move $a0, $s4
jal fib
lw $a0, 0($sp)
addi $sp, $sp, 4
lw $s2, 20($sp)  #retrieve stack
lw $s3, 16($sp)  #retrieve stack
lw $s5, 12($sp)  #retrieve stack
lw $s4, 8($sp)  #retrieve stack
lw $s6, 4($sp)  #retrieve stack
lw $s7, 0($sp)  #retrieve stack
addi $sp, $sp, 24  #retrieve stack
move $s2, $v0
add $t3, $s3, $s2
move $v0, $t3
# clear stack
lw $ra, 0($sp)
addi $sp, $sp, 4
jr $ra
```

APÊNDICE E – BACK-END PARA GERAÇÃO DE CÓDIGO MIPS

```
{
    #include"../src/Tree.h"
    #include<iostream>

    bool isNumber(std::string);
    bool isLiteral(std::string);
    bool isRegister(std::string);
    bool isTempReg(std::string);
    bool isVariableRegister(std::string);
    bool isArgumentReg(std::string);
    bool isNonAllocable(std::string);
    void doubleRegisterInstruction(std::string instruction, std::string root, std::string reg1, std::string
    ↪  reg2, bool isRootWriteable=false);
    void regConstInstruction(std::string instruction, std::string root, std::string reg, std::string cnst,
    ↪  bool isRootWriteable=false);
    void returnInstruction(std::string operand);
    void functionInstruction(std::string function, std::string var1="", std::string var2="", std::string
    ↪  var3="", std::string var4="");
    void printInstruction(std::string literal, std::string integer="");
}

%term COMMAND

%term RETURN
%term WHILE
%term FOR
%term IF

%term CONST
%term STRING
%term VARIABLE
%term FUNCTION_CALL

%term EXIT
%term SCANF
%term PRINTF
%term SWITCH

%term ASSIGN

%term ADD
%term SUB
%term DIV
%term MUL
%term MOD

%term RSHIFT
%term LSHIFT

%term INC
%term DEC
%term UNR_PLUS
%term UNR_MINUS
```

```
%term BIT_OR
%term BIT_AND
%term BIT_XOR
%term SUBSCRIPT

%term LOG_AND
%term LOG_OR
%term LOG_NOT
%term EQUALS
%term NOT_EQUALS
%term LESS
%term GREAT
%term LEQ
%term GEQ

%nonterm reg
%nonterm stmt

%%

_var <- reg: VARIABLE { return 0; } = {
    Logger::Log("Var", $[1]);
};
_const <- reg: CONST { return 1; } = {
    std::string reg = newTempRegister();
    RegAlloc::newInstruction({"li " + reg + ", " + $[0]}, {});
    $node[0].setName(reg);
    Logger::Log("Const", reg + " = " + $[0]);
};
_statement <- stmt: reg { return $cost[0]; } = {
    Logger::Log("Statement", $[1]);
};

_addConst <- reg: ADD(reg,CONST) { return $cost[1] + 1; } = {
    regConstInstruction("addi", $[0], $[1], $[2]);
    Logger::Log("Add Const", $[1] + " + " + $[2]);
};
_addReg <- reg: ADD(reg,reg) { return $cost[1] + $cost[2] + 1; } = {
    doubleRegisterInstruction("add", $[0], $[1], $[2]);
    Logger::Log("Add Reg", $[1] + " + " + $[2]);
};
_subConst <- reg: SUB(reg,CONST) { return $cost[1] + 1; } = {
    regConstInstruction("subi", $[0], $[1], $[2]);
    Logger::Log("Sub Const", $[1] + " - " + $[2]);
};
_subReg <- reg: SUB(reg,reg) { return $cost[1] + $cost[2] + 1; } = {
    doubleRegisterInstruction("sub", $[0], $[1], $[2]);
    Logger::Log("Sub Reg", $[1] + " - " + $[2]);
};
_mod <- reg: MOD(reg, reg) { return $cost[1] + $cost[2] + 3; } = {
    doubleRegisterInstruction("div", $[0], $[1], $[2]);
    RegAlloc::newInstruction({"mfhi " + $[0]}, {});
    Logger::Log("Mod");
};
_mul <- reg: MUL(reg,reg) { return $cost[1] + $cost[2] + 2; } = {
    // doubleRegisterInstruction("mul", $[0], $[1], $[2]);
    std::string templ{ "mult " };
    std::vector<Instruction::OperandType> operands{};
    if(isNonAllocable($[1])){
        templ += $[1] + ", ";
```

```
    } else {
        templ += "%o, ";
        operands.emplace_back(Instruction::OperandType{$[1]});
    }
    if(isNonAllocable($[2])) {
        templ += $[2];
    } else {
        templ += "%o";
        operands.emplace_back(Instruction::OperandType{$[2]});
    }
    RegAlloc::newInstruction(templ, operands);
    RegAlloc::newInstruction({"mflo " + $[0]}, {});
    Logger::Log("Mul", $[1] + " * " + $[2]);
};

_increment <- reg: INC(reg) { return $cost[1] + 1; } = {
    std::string templ{"addi "};
    std::vector<Instruction::OperandType> operands;
    if(isNonAllocable($[1])) {
        templ += $[1] + ", " + $[1] + ", 1";
    } else {
        templ += "%o, %o, 1";
        operands.emplace_back(Instruction::OperandType{$[1]});
        operands.emplace_back(Instruction::OperandType{$[1]});
    }
    RegAlloc::newInstruction(templ, operands);
    Logger::Log("Increment", $[1] + "++");
};
_subscriptConst <- reg: SUBSCRIPT(reg, CONST) { return $cost[1] + 1; } = {
    RegAlloc::newInstruction({"lw " + $[0] + ", " + std::to_string(std::stoi($[2])*4) + "(%o)"}, {
    ↪  Instruction::OperandType{$[1]} } );
    Logger::Log("Subscript Const", $[1] + "[" + $[2] + "])");
};
_subscript <- reg: SUBSCRIPT(reg, reg) { return $cost[1] + $cost[2] + 1; } = {
    std::string templ{"lw " + $[0] + ", "};
    if(isNonAllocable($[2])) {
        std::string subscriptor = $[2];
        if(isNumber(subscriptor)) {
            subscriptor = std::to_string(4 * std::stoi(subscriptor));
            RegAlloc::newInstruction({"li " + $[0] + ", " + subscriptor}, {});
        } else {
            RegAlloc::newInstruction({"sll " + $[0] + ", " + subscriptor + ", 2"}, {});
        }
        std::string addi = (isNumber(subscriptor) ? "i " : " ");
        std::string addiReg = (isNumber(subscriptor) ? $[2] : $[0]);
        RegAlloc::newInstruction({"add " + addi + $[0] + ", %o, " + addiReg}, { {$[1]} });
        templ += "(" + $[0] + ")";
    } else {
        RegAlloc::newInstruction({"sll " + $[0] + ", %o, 2"}, { {$[2]} });
        RegAlloc::newInstruction({"add " + $[0] + ", %o, " + $[0]}, { {$[1]} });
        templ += "(" + $[0] + ")";
    }
    RegAlloc::newInstruction(templ, {});
    Logger::Log("Subscript", $[1] + "[" + $[2] + "])");
};


_equalsConst <- reg: EQUALS(reg,CONST) { return $cost[1] + 1; } = {
    regConstInstruction("seqi", $[0], $[1], $[2]);
    Logger::Log("Equals Const", $[1] + " == " + $[2]);
```

```
};
_equalsReg <- reg: EQUALS(reg,reg) { return $cost[1] + $cost[2] + 1; } = {
    doubleRegisterInstruction("seq", $[0], $[1], $[2]);
    Logger::Log("Equals", $[1] + " == " + $[2]);
};
_lessThanConst <- reg: LESS(reg, CONST) { return $cost[1] + 1; } = {
    regConstInstruction("slti", $[0], $[1], $[2]);
    Logger::Log("Less Than Const", $[1] + " < " + $[2]);
};
_lessThan <- reg: LESS(reg, reg) { return $cost[1] + $cost[2] + 1; } = {
    doubleRegisterInstruction("slt", $[0], $[1], $[2]);
    Logger::Log("Less Than", $[1] + " < " + $[2]);
};
_leqConst <- reg: LEQ(reg, CONST) { return $cost[1] + 1; } = {
    doubleRegisterInstruction("sle", $[0], $[1], $[2]);
    Logger::Log("Leq Const", $[1] + " <= " + $[2]);
};
_leq <- reg: LEQ(reg, reg) { return $cost[1] + $cost[2] + 1; } = {
    regConstInstruction("sle", $[0], $[1], $[2]);
    Logger::Log("Leq", $[1] + " <= " + $[2]);
};

_if <- stmt: IF(reg, {
                         std::string index = std::to_string(labelIndex(true));
                         $node[0].label = index;
                         RegAlloc::newInstruction({"beq "+$[1]+", $zero, if_end_"+index}, {});
                     } stmt) { return $cost[1] + 1; } = {
    std::string index = $node[0].label;
    RegAlloc::newInstruction({"if_end_"+index+":"}, {});
    Logger::Log("If");
};
_ifElse <- stmt: IF(reg, {
                        std::string index = std::to_string(labelIndex(true));
                        $node[0].label = index;
                        RegAlloc::newInstruction({"beq "+$[1]+", $zero, if_else_"+index}, {});
                    } stmt, {
                        std::string index = $node[0].label;
                        RegAlloc::newInstruction({"j if_end_"+index}, {});
                        RegAlloc::newInstruction({"if_else_"+index+":"}, {});
                    }stmt) { return $cost[1] + 1; } = {
    std::string index = $node[0].label;
    RegAlloc::newInstruction({"if_end_"+index+":"}, {});
    Logger::Log("_ifElse");
};

_for <- stmt: FOR({
                      $node[0].label = std::to_string(labelIndex(true));
                      RegAlloc::newInstruction({"for_init_" + $node[0].label + ":"}, {});
                  } stmt, {
                      std::string index = $node[0].label;
                      RegAlloc::newInstruction({"for_check_" + index + ":"}, {});
                  } reg, {
                      std::string index = $node[0].label;
                      std::string templ{"beq "};
                      std::vector<Instruction::OperandType> operands{};
                      if(isNonAllocable($[2])) templ += $[2] + ", $zero, for_end_" + index;
                      else {
                          templ += "%o, $zero, for_end_" + index;
                          operands.emplace_back(Instruction::OperandType{$[2]});
                      }
```

```
                    RegAlloc::newInstruction(templ, operands);
                    RegAlloc::newInstruction({"j for_body_" + index}, {});
                    RegAlloc::newInstruction({"for_update_" + index + ":"}, {});
                } reg, {
                    std::string index = $node[0].label;
                    RegAlloc::newInstruction({"j for_check_" + index}, {});
                    RegAlloc::newInstruction({"for_body_" + index + ":"}, {});
                } stmt) { return $cost[2] + $cost[3] + 1; } =
{
    std::string index = $node[0].label;
    RegAlloc::newInstruction({"j for_update_" + index}, {});
    RegAlloc::newInstruction({"for_end_" + index + ":"}, {});
    Logger::Log("For");
};

_returnConst <- stmt: RETURN(CONST) { return 1; } = {
    returnInstruction($[1]);
    Logger::Log("Return Const", $[1]);
};
_return <- stmt: RETURN(reg) { return 2; } = {
    returnInstruction($[1]);
    Logger::Log("Return", $[1]);
};

_print <- stmt: PRINTF(STRING) { return 1; } = {
    printInstruction($[1]);
    Logger::Log("Printf", AstSymbols::Programa::getLiteralVar($[1]).value() + ": " + $[1]);
};
_printVar <- stmt: PRINTF(STRING, VARIABLE) { return 1; } = {
    printInstruction($[1], $[2]);
    Logger::Log("Printf", $[2]);
};
_scan <- stmt: SCANF(VARIABLE) { return 1; } = {
    RegAlloc::newInstruction({"li $v0, 5"}, {});
    RegAlloc::newInstruction({"syscall"}, {});
    RegAlloc::newInstruction({"move %o, $v0"}, { {$[1]} });
    Logger::Log("Scanf", $[1]);
};
_exit <- stmt: EXIT(CONST) { return 1; } = {
    RegAlloc::newInstruction("li $v0, 10", {});
    RegAlloc::newInstruction("li $a0, %c", {}, {$[1]});
    RegAlloc::newInstruction("syscall", {});
    Logger::Log("Exit", $[1]);
};

_assignConst <- stmt: ASSIGN(reg,CONST) { return $cost[1] + 1; } = {
    if($node[1].getSymbol() == (int)Yamg::User_Symbols::SUBSCRIPT) {
        std::string subscripted = t.getImmediateChild(1).getName();
        std::string subscriptor = t.getImmediateChild(2).getName();
        std::string tempReg = $[1];
        if(isNonAllocable(subscriptor)) {
            if(isNumber(subscriptor)){
                subscriptor = std::to_string(4 * std::stoi(subscriptor));
                RegAlloc::newInstruction({"li " + tempReg + ", " + subscriptor}, {});
            } else {
                RegAlloc::newInstruction({"sll " + tempReg + ", " + subscriptor + ", 2"}, {});
            }
            std::string addi = (isNumber(subscriptor) ? "i " : " ");
            std::string addiReg = (isNumber(subscriptor) ? subscriptor : tempReg);
```

```
            RegAlloc::newInstruction({"add" + addi + tempReg + ", %o, " + addiReg}, {
            ↪  Instruction::OperandType{subscripted} });
    } else {
        RegAlloc::newInstruction({"sll " + tempReg + ", %o, 2"}, { Instruction::OperandType{subscriptor}
        ↪  });
        RegAlloc::newInstruction({"add " + tempReg + ", %o, %o"}, { Instruction::OperandType{subscriptor},
        ↪  Instruction::OperandType{subscripted} });
    }
        if(isNonAllocable($[2])) {
            RegAlloc::newInstruction({"sw " + $[2] + ", (" + tempReg + ")"}, {});
        } else {
            RegAlloc::newInstruction({"sw %o, (" + tempReg + ")"}, { Instruction::OperandType{$[2]} });
        }
    } else {
        if(isNonAllocable($[1])) RegAlloc::newInstruction({"li " + $[1] +", %c"}, {}, {$[2]});
        else RegAlloc::newInstruction({"li %o, %c"}, { {$[1], YAMG_WRITEABLE_OPERAND} }, {$[2]});
    }
    Logger::Log("Assign Const", $[1] + " = " + $[2]);
};
_assign <- stmt: ASSIGN(reg,reg) { return $cost[1] + $cost[2] + 2; } = {
    if($node[1].getSymbol() == (int)Yamg::User_Symbols::SUBSCRIPT) {
        std::string subscripted = $node[1].getImmediateChild(1).getName();
        std::string subscriptor = $node[1].getImmediateChild(2).getName();
        std::string tempReg = $[1];
        if(isNonAllocable(subscriptor)) {
            if(isNumber(subscriptor)){
                subscriptor = std::to_string(4 * std::stoi(subscriptor));
                RegAlloc::newInstruction({"li " + tempReg + ", " + subscriptor}, {});
            } else {
                RegAlloc::newInstruction({"sll " + tempReg + ", " + subscriptor + ", 2"}, {});
            }
            std::string addi = (isNumber(subscriptor) ? "i " : " ");
            std::string addiReg = (isNumber(subscriptor) ? subscriptor : tempReg);
            RegAlloc::newInstruction({"add" + addi + tempReg + ", %o, " + addiReg}, {
            ↪  Instruction::OperandType{subscripted} });
        } else {
            RegAlloc::newInstruction({"sll " + tempReg + ", %o, 2"}, {
            ↪  Instruction::OperandType{subscriptor} });
            RegAlloc::newInstruction({"add " + tempReg + ", %o, " + tempReg}, {
            ↪  Instruction::OperandType{subscripted} });
        }
        if(isNonAllocable($[2])) {
            RegAlloc::newInstruction({"sw " + $[2] + ", (" + tempReg + ")"}, {});
        } else {
            RegAlloc::newInstruction({"sw %o, (" + tempReg + ")"}, { Instruction::OperandType{$[2]} });
        }
    } else {
        std::string templ{"move "};
        std::vector<Instruction::OperandType> operands{};
        if(isNonAllocable($[1])) templ += $[1] + ", ";
        else {
            templ += "%o, ";
            operands.emplace_back(Instruction::OperandType{$[1], YAMG_WRITEABLE_OPERAND});
        }
        if(isNonAllocable($[2])) templ += $[2];
        else {
            templ += "%o";
            operands.emplace_back(Instruction::OperandType{$[2]});
        }
        RegAlloc::newInstruction(templ, operands);
```

```
    }
    Logger::Log("Assign", $[1] + " = " + $[2]);
};
_functionZero <- reg: FUNCTION_CALL { return 1; } = {
    functionInstruction($[0]);
    $node[0].setName("$v0");
    Logger::Log("Function Call", "no parameters");
};
_functionOne <- reg: FUNCTION_CALL(reg) { return $cost[1] + 1; } = {
    functionInstruction($[0], $[1]);
    $node[0].setName("$v0");
    Logger::Log("Function Call", "1 parameter");
};
_functionTwo <- reg: FUNCTION_CALL(reg,reg) { return $cost[1] + $cost[2] + 1; } = {
    functionInstruction($[0], $[1], $[2]);
    $node[0].setName("$v0");
    Logger::Log("Function Call", "2 parameters");
};
_functionThree <- reg: FUNCTION_CALL(reg,reg,reg) { return $cost[1] + $cost[2] + $cost[3] + 1; } = {
    functionInstruction($[0], $[1], $[2], $[3]);
    $node[0].setName("$v0");
    Logger::Log("Function Call", "3 parameters");
};
_functionFour <- reg: FUNCTION_CALL(reg,reg,reg,reg) { return $cost[1] + $cost[2] + $cost[3] + $cost[4] +
↪   1; } = {
    functionInstruction($[0], $[1], $[2], $[3], $[4]);
    $node[0].setName("$v0");
    Logger::Log("Function Call", "4 parameters");
};

_command <- stmt: COMMAND(stmt,stmt) { return 0; } = {
    Logger::Log("Commands");
};
_lastCommand <- stmt: COMMAND(stmt) { return 0; } = {
    Logger::Log("Last Command");
};

%%

%register $s0
%register $s1
%register $s2
%register $s3
%register $s4
%register $s5
%register $s6
%register $s7

%spill_register $t0
%spill_register $t1
%spill_register $t2

%set_read "lw %o, %o($sp)"
%set_write "sw %o, %o($sp)"

%%

{
bool isNumber(std::string reg)           { return (std::isdigit(reg[0])); }
```

```
bool isLiteral(std::string reg)          { return (reg[0] == '\"' || reg[0] == '\'' ||
↪  std::isdigit(reg[0])); }
bool isTempReg(std::string reg)          { return (isRegister(reg) && (reg[1] == 't')); }
bool isRegister(std::string reg)         { return (reg[0] == '$'); }
bool isArgumentReg(std::string reg)      { return (isRegister(reg) && (reg[1] == 'a')); }
bool isVariableRegister(std::string reg) { return (isRegister(reg) && reg[1] == 's'); }
bool isReturnReg(std::string reg)        { return (isRegister(reg) && reg[1] == 'v'); }
bool isNonAllocable(std::string reg) {
    return isNumber(reg)    ||
           isLiteral(reg)   ||
           isTempReg(reg)   ||
           isReturnReg(reg) ||
           isArgumentReg(reg);
}

void doubleRegisterInstruction(std::string instruction, std::string root, std::string reg1, std::string
↪  reg2, bool isRootWriteable){
    std::string templ{ instruction + " " + root + ", " };
    std::vector<Instruction::OperandType> operands{};
    if(!isNonAllocable(root)){
        operands.emplace_back(Instruction::OperandType{root, isRootWriteable});
    }
    if(isNonAllocable(reg1)) {
        templ += reg1 + ", ";
    } else {
        templ += "%o, ";
        operands.emplace_back(Instruction::OperandType{reg1});
    }
    if(isNonAllocable(reg2)) {
        templ += reg2;
    } else {
        templ += "%o";
        operands.emplace_back(Instruction::OperandType{reg2});
    }
    RegAlloc::newInstruction(templ, operands);
}

void regConstInstruction(std::string instruction, std::string root, std::string reg, std::string cnst, bool
↪  isRootWriteable){
    std::string templ{ instruction + " " + root + ", " };
    std::vector<Instruction::OperandType> operands{};
    if(!isNonAllocable(root)){
        operands.emplace_back(Instruction::OperandType{root, isRootWriteable});
    }
    if(isNonAllocable(reg)) {
        templ += reg + ", ";
    } else {
        templ += "%o, ";
        operands.emplace_back(Instruction::OperandType{reg});
    }
    templ += "%c";
    RegAlloc::newInstruction(templ, operands, {cnst});
}

void returnInstruction(std::string operand){
    std::string mvTempl{};
    std::vector<Instruction::OperandType> mvOperands{};
    std::vector<std::string> mvConst{};
    if(isRegister(operand)){
        mvTempl = "move $v0, ";
```

```
        if(isNonAllocable(operand)) {
            mvTempl += operand;
        } else {
            mvTempl += "%o";
            mvOperands.emplace_back(Instruction::OperandType{operand});
        }
    } else {
        mvTempl = "li $v0, %c";
        mvConst.emplace_back(operand);
    }
    RegAlloc::newInstruction(mvTempl, mvOperands, mvConst);
    RegAlloc::clearStack();
    RegAlloc::newInstruction({"lw $ra, 0($sp)"}, {});
    RegAlloc::newInstruction({"addi $sp, $sp, 4"}, {});
    RegAlloc::newInstruction({"jr $ra"}, {});
}

void functionInstruction(std::string function, std::string var1, std::string var2, std::string var3,
↪  std::string var4){
    std::vector<std::string> vars{};
    if(!var1.empty()) vars.emplace_back(var1);
    if(!var2.empty()) vars.emplace_back(var2);
    if(!var3.empty()) vars.emplace_back(var3);
    if(!var4.empty()) vars.emplace_back(var4);

    RegAlloc::storeStack();

    RegAlloc::newInstruction({"subi $sp, $sp, " + std::to_string(vars.size() * 4)}, {});
    for(int i=0; i<vars.size(); i++) {
        std::string store{"sw $a" + std::to_string(i) + ", " + std::to_string(i*4) + "($sp)"};
        RegAlloc::newInstruction(store, {});
        if(isLiteral(vars[i])) {
            if(isNumber(vars[i])) {
                RegAlloc::newInstruction({"li $a" + std::to_string(i) + ", %c"}, {}, { vars[i] });
            } else {
                throw std::runtime_error(std::to_string(i) + "º argumento da função \"" + function + "\"
                ↪  não é um inteiro nem uma variável");
                // RegAlloc::newInstruction({"move $a"+std::to_string(i)+", " + vars[i]}, {});
            }
        } else {
            RegAlloc::newInstruction({"move $a"+std::to_string(i)+", %o"}, { {vars[i] } });
        }
    }

    RegAlloc::newInstruction({"jal " + function}, {});

    for(int i=vars.size()-1; i>=0; i--) {
        std::string store{"lw $a" + std::to_string(i) + ", " + std::to_string(i*4) + "($sp)"};
        RegAlloc::newInstruction(store, {});
    }
    RegAlloc::newInstruction({"addi $sp, $sp, " + std::to_string(vars.size()*4)}, {});
    RegAlloc::retrieveStack();
}

void printInstruction(std::string literal, std::string integer) {
    RegAlloc::newInstruction({"subi $sp, $sp, 4"}, {});
    RegAlloc::newInstruction({"sw $a0, 0($sp)"}, {});
    if(integer.empty()) {
        std::string var = AstSymbols::Programa::getLiteralVar(literal).value();
        RegAlloc::newInstruction({"li $v0, 4"}, {});
```

```
        RegAlloc::newInstruction({"la $a0, " + var}, {});
    } else {
        RegAlloc::newInstruction({"li $v0, 1"}, {});
        std::string templ{"la $a0, ("};
        std::vector<Instruction::OperandType> operands{};
        if(isNonAllocable(integer)) {
            templ += integer;
        } else {
            templ += "%o";
            operands.emplace_back(Instruction::OperandType{integer});
        }
        RegAlloc::newInstruction(templ + ")", operands);
    }
    RegAlloc::newInstruction({"syscall"}, {});
    RegAlloc::newInstruction({"lw $a0, 0($sp)"}, {});
    RegAlloc::newInstruction({"addi $sp, $sp, 4"}, {});
}

}
```

Anexos

ANEXO A – REPOSITÓRIO DO ANALISADOR SEMÂNTICO

https://github.com/SophieNyah/Compiladores

O repositório possui vários programas escritos ao longo da disciplina de Compiladores. O analisador semântico se encontra na pasta `2_semestre/Analisador_Semantico_C`.

ANEXO B – REPOSITÓRIO DO YAMG NO GITHUB

https://github.com/SophieNyah/TCC

O repositório possui tanto o YAMG quanto o programa citado no Capítulo 8, sobre a pasta `user/`. No repositório há um manual detalhado de como utilizar o YAMG, e também um manual básico para o programa de usuário.