Construção de um Gerador de Código Multipropósito

Sophie Nascimento[1], **Wesley Attrot**[1]

[1]Departamento de Computação – Universidade Estadual de Londrina (UEL)
Caixa Postal 10.011 – CEP 86057-970 – Londrina – PR – Brasil

sophie.nascimento@uel.br, wesley@uel.br

Abstract. *This work proposes the creation of a new code generation tool, capable of doing so with three different algorithms, according to the programmer's needs. Besides that, this tool contains an optional and simple register allocator, so there's no need to do 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.*

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 adeque aos padrões modernos, além de integrar vários métodos de geração em um único programa.*

1. Introdução

Uma etapa custosa [7] na criação de um compilador consiste na criação de um gerador de código para a máquina alvo, principalmente se forem considerados compiladores multiplataforma, que oferecem suporte para diversas arquiteturas e CPUs. Diferentes gerações de CPUs, por exemplo, trazem novas funcionalidades, e para tirar proveito delas, é necessário realizar alterações em como partes do código são geradas. Sendo assim, surge a necessidade de uma ferramenta para auxiliar nesse processo. Geradores de geradores de código, tal como geradores de analisadores léxicos e sintáticos, foram criados para facilitar a tarefa de compilar códigos para diferentes plataformas. Estes geradores tiram das mãos dos programadores grande parte do peso de criar regras de tradução de estruturas intermediárias para comandos *assembly*.

Com o passar do tempo, cada vez mais processadores e arquiteturas são construídas. Mesmo dentro de uma mesma família de processadores ainda há diferenças, desde uma instrução completamente nova e especializada, até coisas menores, como maneiras de lidar com operações de ponto flutuante, endereçamento de memória, dentre outras coisas. Conforme novas técnicas e tecnologias são desenvolvidas para melhorar o desempenho, elas serão implementadas em novas CPUs. Essa implementação particular de um conjunto de instruções em um processador específico é chamada de microprocessador, e varia até mesmo em processadores da mesma família.

A família de processadores *Exynos*, da *Samsung*, por exemplo, lançou o modelo 2100 em janeiro de 2021 que utilizava a versão ARMv8.2-A da arquitetura ARM, e em

janeiro de 2022, lançou o modelo 2200, com a versão ARMv9-A da mesma arquitetura. Para exemplificar pelo menos uma de suas inovações, a versão 8.2-A trouxe suporte para endereços de 52 bits, enquanto a versão 9-A traz coisas como instruções de geração de números aleatórios e identificação de *branch* alvo [1].

Desde do lançamento do primeiro processador *Snapdragon* em 2007, foram utilizadas quatro versões diferentes do conjunto de instruções ARM, além de dezoito microarquiteturas diferentes.

Como um exemplo recente, no ano de 2020, a empresa norte-americana *Apple* lançou seu novo *System-on-a-Chip*, o M1. Diferente dos processadores anteriores, que utilizavam o conjunto de instruções x86, agora é baseado em ARM.

É possível observar a frequência com que novas tecnologias, uma microarquitetura e até mesmo a versão de uma arquitetura é alterada, e para se tomar vantagem dessas mudanças, ou até para que seja possível sequer rodar o código, muitas vezes fazem-se necessários ajustes no gerador de código, para que o código de máquina gerado seja o melhor possível para aquele processador.

Geradores de código, porém, são programas grandes e complexos, o que significa que realizar uma alteração nesses geradores é deveras trabalhoso, consumindo tempo e recursos que poderiam ser alocados para outros trabalhos. Além disso, simplesmente criar um gerador de código para um conjunto não trivial de instruções é uma tarefa extremamente árdua e longa. De todo modo, uma ferramenta para facilitar esse processo é de grande importância.

Ademais, diferentes métodos de geração de código resultam em executáveis diferentes. O *Maximal Munch* [3], por exemplo, é um método que busca sempre as instruções mais complexas, que geralmente abrangem a funcionalidade de duas ou mais instruções mais simples em uma só, o que resulta em um executável menor, e também menos intruções para a CPU executar.

Outros algoritmos de geração de código são o *Minimal Munch* [3] e a programação dinâmica [2], que fazem uma seleção baseada nas instruções mais simples, e em custos definidos pelo programador, respectivamente. O primeiro caso é mais simples de ser analisado e gera instruções mais rapidamente, portanto pode ser melhor para interpretadores ou compilação JIT, enquanto a programação dinâmica é ideal para necessidades personalizadas. Cada programa possui necessidades próprias para melhor funcionar, e a maneira como o código é gerado também faz parte desse aspecto, com métodos diferentes oferecendo vantagens e desvantagens próprias.

Este trabalho propõe um programa para criação automática dessas ferramentas, onde, dadas as regras que descrevem a máquina alvo, a saída é um gerador de código que atende à todas as especificações de entrada. Isso significa que não é mais necessário escrever o gerador inteiro à mão, é preciso apenas dizer quais são os padrões das instruções, e qual código alvo será gerado, assim tornando essa tarefa muito mais fácil de ser realizada e posteriormente mantida. Além disso, também serão ofertados os três métodos acima descritos, para dar uma maior flexibilidade em termos do código gerado, de modo que desempenho e tamanho do código gerado, velocidade de compilação, e outras métricas possam ser considerados.

A Seção 2 apresenta uma breve explicação do processo de compilação, além de uma breve revisão bibliográfica do estado da arte na área. A Seção 3 apresenta os objetivos que esse trabalho visa alcançar. Os métodos que serão utilizados para se atingir os objetivos propostos são apresentados na Seção 4. A Seção 5 é uma visão geral de quando cada atividade será completada, enquanto a Seção 6 resume aquilo que se espera obter ao fim deste trabalho.

2. Fundamentação Teórico-Metodológica e Estado da Arte

Para melhor entender um compilador e como a etapa de geração de código se encaixa com o resto, faz-se necessário compreender sua estrutura como um todo. Este capítulo traz uma revisão sobre os fundamentos da estrutura de compiladores, passando por todas as suas etapas, e logo depois serão apresentados alguns projetos que constroem sobre a promessa de geradores de geradores de código, como o *BEG* [5], *BURG* [6], e *twig* [2].

2.1. Etapas de Compilação

Embora a ideia de traduzir um código de alto nível para linguagem de máquina pareça simples, o processo de compilação está longe de ser algo trivial. O fluxo básico de um compilador pode ser resumido em quatro etapas principais, descritas com maior profundidade em [12]: análise léxica, análise sintática, análise semântica, e finalmente geração do código para máquina alvo. Por simplicidade, as otimizações serão omitidas. A Figura 1 mostra um fluxograma do processo.

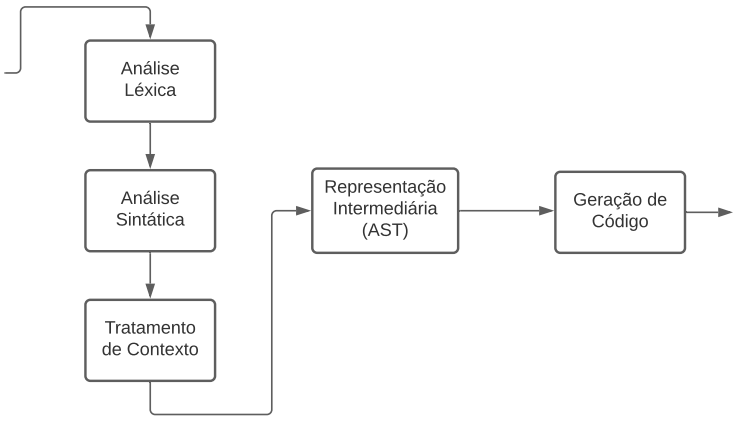

Figura 1. Design geral de um compilador

A análise léxica é a parte mais básica do compilador, tratando de ler e reconhecer todos os símbolos pertencentes à linguagem (palavras reservadas, símbolos matemáticos, identificadores, strings, etc); quando um símbolo é reconhecido, lhe é atribuído um *token*, que identifica o que aquele símbolo representa. Algumas ferramentas utilizadas para esse processo são *lex* [10] e sua versão aprimorada, o *flex* [11].

Em seguida, há a análise sintática, que verifica a corretude da estrutura dos programas ao ver se todas elas estão de acordo com a especificação (como a ausência de pontos e vírgulas, operações sem operandos, parênteses que não foram fechados, etc); caso erros sintáticos não tenham sido encontrados, uma árvore sintática abstrata (*AST*) é gerada. Ela

é uma estrutura de dados que representa o código e retém apenas informações essenciais. Exemplos de softwares que realizam essa análise incluem o *yacc* [9] e sua popular versão aprimorada, o *GNU Bison* [4], e também o mais moderno *ANTLR* [13].

O próximo passo é a análise semântica, cuja função é checar por inconsistências de contexto. Operações com tipos conflitantes (incremento de uma string, subtração entre dois ponteiros, indexação de um inteiro, etc.), inexistência da variável no escopo atual, função ser chamada com parâmetros incompatíveis, dentre outros, são exemplos de erros que podem ser detectados nesta etapa; quando concluída, um código intermediário é gerado, sobre o qual é possível aplicar otimizações. Graças às variadas especificações de diferentes linguagens, uma ferramenta para tal análise seria deveras limitada, e desconhecida pela autora desse trabalho. Ao invés disso, a análise da *AST* é feita logo após sua criação, embora seja possível fazer validações diretamente nas regras dos analisadores sintáticos.

Para finalizar, a quarta e última parte do compilador será apresentada. Essa etapa é responsável por casar os padrões gerados no código intermediário e transformá-los em código para a máquina alvo. Esse casamento é feito com base em algoritmos como *Maximal Munch*, *Minimal Munch* [3], e através de programação dinâmica [2], de acordo com a necessidade do programador. Embora seja possível programar diretamente o gerador de código, tal processo é custoso por si só, e é necessário que um novo seja construído para cada arquitetura diferente em que se deseja compilar.

2.2. Geradores de Geradores de Código

Como foi visto, a criação de um gerador de código é uma tarefa custosa, e que pode ser facilitada por automatizar sua criação, deixando assim para o programador apenas o processo de descrição das regras, e modelá-las no gerador de geradores. Aqui serão brevemente apresentadas algumas dessas ferramentas de automação.

O *twig* foi desenvolvido quando as tecnologias de geração de código ainda eram experimentais e diferentes métodos estavam a ser aplicados para se descobrir qual a melhor solução para o problema. Baseando-se em trabalhos prévios, o *twig* foi um dos primeiros a utilizar uma notação de árvore para implementar seu gerador, além de um sistema de casamento de padrões também baseado em árvores, sendo que o código é gerado através de regras para reescrita da árvore. Essas regras têm o formato $replacement \longleftarrow template\{cost\} = \{action\}$, onde *template* é a sub-árvore que foi casada com um padrão, *replacement* é o nó que substituirá a sub-árvore, *cost* e *action* são pedaços de códigos, onde o primeiro retorna o custo daquele *template* ser casado, e o segundo contém as ações a serem aplicadas quando esse casamento ocorre.

Graças ao seu sistema orientado a árvores, o *twig* é capaz de simplificar e juntar regras quando necessário. Por conta dessas otimizações, essa ferramenta é capaz de, com base nos custos, encontrar soluções ótimas para os subproblemas e, consequentemente, para o problema como um todo. É importante notar que, apesar disso, ele não faz otimizações sobre o código em si, como *common subexpression elimination*, por exemplo, o que traz a necessidade de outras ferramentas para a realização dessas atividades.

O *BEG* (que possui inspiração no *twig* e no *ASCOT* [8]) é outra ferramenta que utiliza uma técnica geral de casamento de padrões para seleção de instruções, e programação dinâmica já imbuída no código, sem a necessidade de utilizar tabelas. Um grande diferen-

cial do *BEG* é que ele traz consigo um alocador de registradores, que pode ser utilizado de duas maneiras: alocador *on the fly*, que funciona para conjuntos de registradores considerados regulares; e um método geral, que permite a alocação para diversos processadores. Uma evolução em relação ao *twig*, é a possibilidade de alteração da representação intermediária antes do casamento de padrões, embora os próprios autores reconheçam que, no momento, esse ainda era um recurso rudimentar, que poderia ser trabalhado no futuro.

Tomando como base conceitos mais complexos, o *BURG* utiliza BURS (Bottom-Up Rewrite System) para criar um *tree parser*. O gerador resultante, chamado de *BURM* é capaz de descobrir, em tempo linear, um *parser* ótimo. Duas travessias são feitas na árvore de entrada do *BURM*, a primeira delas visita todos os nós da árvore uma vez, de baixo pra cima e da esquerda para a direita, e atribui a cada um um estado, que representa os casamentos possíveis naquele nó; essa etapa é chamada de *labeller*. Já a segunda etapa, nomeada *reducer*, percorre a árvore de cima para baixo, e ao combinar o rótulo de cada estado com um dado não-terminal, define qual regra deve ser aplicada. Como podem haver vários casamentos possíveis em cada nó, e também dada a variabilidade do não-terminal, este passo pode ser repetido até que o código ótimo seja encontrado.

2.3. Alocação de Registradores

A alocação de registradores é um conjunto de técnicas utilizadas para que os registradores sejam escolhidos automaticamente, sem que o programador tenha necessidade de mexer no código *assembly* resultante e manualmente inserir os registradores.

Este não é um problema simples, pois uma solução trivial pode debilitar o código gerado, com acessos muito mais frequentes à memória por exemplo. Por outro lado, um algoritmo muito complexo aumenta o tempo de compilação, que pode não ser interessante para alguns casos. Sendo assim, a escolha de qual algoritmo usar é importante.

Nas ferramentas analisadas, apenas o *BEG* implementava um alocador de registradores, algo que é incomum para esses programas de geradores de geradores de código fazerem. Não só há um aumento na dificuldade de implementação, mas também por ser uma escolha do usuário final, que é quem sabe qual o melhor algoritmo a ser utilizado para seu caso específico.

3. Objetivos

Este trabalho visa fazer uma revisão bibliográfica sobre outros projetos similares com o intuito de entender seu funcionamento e seus padrões e, com base no conhecimento levantado, produzir uma ferramenta para automatizar a etapa de geração de código, além de prover algumas funcionalidades extras que não são comumente encontradas em projetos similares.

É comum que cada gerador utilize apenas uma das técnicas de casamento de padrões acima mencionadas, usualmente o método de programação dinâmica. Isso se deve à versatilidade desse procedimento, porém tanto *Maximal Munch* e o *Minimal Munch* possuem casos de uso específicos. Tendo isso em vista, este trabalho pretende gerar uma ferramenta que implemente as três técnicas, oferecendo assim maior controle para o usuário sobre como gerar seu código.

Além disso, a ferramenta contará também com um alocador de registradores opcional, para que o código gerado não precise mais ser tratado. Esse alocador é um passo

opcional, de modo que o usuário poderá desabilitá-lo caso deseje utilizar outro algoritmo.

Além disso, para tirar maior proveito de padrões modernos de código, e ainda manter a liberdade sobre otimizações de baixo nível, a linguagem C++ será utilizada, em contraste com outras ferramentas similares, que geralmente são escritas na linguagem C.

4. Procedimentos metodológicos/Métodos e técnicas

O passo inicial é revisar a literatura sobre seletores de instrução e geradores de código. A ferramente será criada com base nas melhores práticas encontradas nesta revisão, dando sempre preferências para técnicas de maior efetividade, e que gerem um melhor código com custo mínimo. Para a leitura e interpretação das regras do gerador, que serão fornecidas em um arquivo pelo usuário, serão utilizadas as ferramentas *Flex* e *Bison*. Além disso, fica a cargo do programador a maneira como a *AST* será lida, para que possa escolher o método que melhor lhe sirva. Assume-se que essa árvore represente um código válido na linguagem fonte, portanto a ferramenta a ser desenvolvida não checará por erros, voltando seu foco apenas para a geração de código.

5. Cronograma de Execução

Atividades:

1. Revisão bibliográfica;
2. Definição do formarto de entrada;
3. Implementação;
4. Testes;
5. Escrita do TCC.

Tabela 1. Cronograma de Execução

	ago/22	set/22	out/22	nov/22	dez/22	jan/23	fev/23	mar/23	abr/23	mai/23	jun/23
1	x	x									
2		x	x								
3				x	x	x	x	x	x		
4								x	x	x	
5		x	x	x	x	x	x	x	x	x	x

6. Contribuições e/ou Resultados esperados

Espera-se que este trabalho possa aumentar a compreensão dos leitores sobre uma das etapas da compilação, a geração de código, através de uma linguagem mais explicativa e simples de se entender. Além disso, a produção de uma nova, mais moderna, e bem documentada ferramenta, visa trazer a teoria aqui abordada para o mundo real.

7. Espaço para assinaturas

Londrina, *21 de Agosto de 2022*.

Sophie

Aluno

Orientador

Referências

[1] Features of arm 8.x and 9.x. https://developer.arm.com/documentation/102378/0201/Armv8-x-and-Armv9-x-extensions-and-features. Acessado em: 2022-08-21.

[2] Alfred V Aho, Mahadevan Ganapathi, and Steven WK Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):491–516, 1989.

[3] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, & Tools*. Pearson/Addison Wesley, 2007.

[4] Robert P Corbett. Static semantics and compiler error recovery. Technical report, CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, 1985.

[5] Helmut Emmelmann, F-W Schröer, and Rudolf Landwehr. Beg: a generator for efficient back ends. *ACM Sigplan Notices*, 24(7):227–237, 1989.

[6] Christopher W Fraser, Robert R Henry, and Todd A Proebsting. Burg: fast optimal instruction selection and tree parsing. *ACM Sigplan Notices*, 27(4):68–76, 1992.

[7] Dick Grune, Kees Van Reeuwijk, Henri E Bal, Ceriel JH Jacobs, and Koen Langendoen. *Modern compiler design*. Springer Science & Business Media, 2012.

[8] Hans-Stephan Jansohn. *Automated generation of optimized code*. Number 154. R. Oldenbourg, 1985.

[9] Stephen C Johnson et al. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

[10] Michael E Lesk and Eric Schmidt. Lex: A lexical analyzer generator, 1975.

[11] John Levine. *Flex & Bison: Text Processing Tools*. "O'Reilly Media, Inc.", 2009.

[12] Torben Ægidius Mogensen. *Basics of compiler design*. Torben Ægidius Mogensen, 2009.

[13] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.