Universidade Federal do Piauí
Centro de Ciências da Natureza
Programa de Pós-Graduação em Ciência da Computação

COGNITE - Um *framework* para construção de geradores de código para arquiteturas multi-core

Eugênio Souza Carvalho

Número de Ordem PPGCC: M001

Teresina-PI, 27 de Agosto de 2018

Eugênio Souza Carvalho

COGNITE - Um *framework* para construção de geradores de código para arquiteturas multi-core

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da UFPI (área de concentração: Sistemas de Computação), como parte dos requisitos necessários para a obtenção do Título de Mestre em Ciência da Computação.

Universidade Federal do Piauí – UFPI

Centro de Ciências da Natureza

Programa de Pós-Graduação em Ciência da Computação

Orientador: Ivan Saraiva Silva

Teresina-PI

27 de Agosto de 2018

Eugênio Souza Carvalho

COGNITE - Um *framework* para construção de geradores de código para arquiteturas multi-core/ Eugênio Souza Carvalho. – Teresina-PI, 27 de Agosto de 2018-

88 p. : il. (algumas color.) ; 30 cm.

Orientador: Ivan Saraiva Silva

Dissertação (Mestrado) – Universidade Federal do Piauí – UFPI
Centro de Ciências da Natureza
Programa de Pós-Graduação em Ciência da Computação, 27 de Agosto de 2018.

1. Palavra-chave1. 2. Palavra-chave2. I. Orientador. II. Universidade Federal do Piauí. III. Título COGNITE - Um *framework* para construção de geradores de código para arquiteturas multi-core

CDU 02:141:005.7

"COGNITE - Um framework para construção de geradores de código para arquiteturas multi-core"

EUGÊNIO SOUZA CARVALHO

Dissertação apresentada ao Programa de Pós–Graduação em Ciência da Computação do Centro de Ciências da Natureza da Universidade Federal do Piauí, como parte integrante dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Aprovada por:

Prof. Ivan Saraiva Silva
(Presidente da Banca Examinadora)

Prof. Sílvio Roberto Fernandes de Araújo
(Examinador Externo)

Prof. Kelson Rômulo Teixeira Aires
(Examinador Interno)

Prof. Raimundo Santos Moura
(Examinador Externo)

Teresina, 27 de agosto de 2018

Aos meus pais Hipólito Macêdo de Carvalho e Raimunda Maria de Souza Carvalho, in memoriam

Agradecimentos

Agradeço primeiramente a Deus, aos meus pais, Hipólito e Raimunda, por todos os ensinamentos em vida.

À minha namorada, Vandressa, por todo carinho e paciência.

À minha irmã, Nívea, mesmo que distante, sempre me apoia e está ao meu lado.

À minha avó, Maria, pelo exemplo dedicação e força.

Agradeço ao meu orientador, Ivan Saraiva, por toda paciência, conselhos ditos e confiança dedicada, desde a graduação.

Finalmente, agradeço a todos que diretamente ou indiretamente contribuíram para o sucesso deste trabalho.

"Aprenda de tudo e utilize o que julgar necessário"
(Minha vó)

Resumo

O processo de desenvolvimento de *hardware* e *software* antes sequencial foi substituído por um modelo de desenvolvimento concorrente. Isso decorre do contínuo crescimento da complexidade dos sistemas e do encurtamento da janela de *time-to-market*. Projetos de *hardware* quase sempre resultam em novos paradigmas, bem como na inserção de instruções não suportados em compiladores convencionais. O objetivo desta dissertação é apresentar o *framework* e compilador Cognite, detalhando os recursos disponibilizados pelas ferramentas. O foco principal deste trabalho é disponibilizar uma infraestrutura que acelere o desenvolvimento de geradores de código para validação de projetos de *hardware*. Um objetivo secundário mas não menos importante consiste na utilização do framework com fins didáticos. Para verificar a capacidade de geração de código foram realizados experimentos comparando o código obtido pelo compilador Cognite e as soluções de compiladores Clang e GCC. Os resultados revelam que apesar da pouca maturidade e com um conjunto mínimo de otimizações Cognite obteve um bom posicionamento quando comparado com soluções consolidadas como Clang e GCC, para o conjunto de aplicações analisado. Os pontos de maior destaque são: a redução da quantidade de operações de *load* e *store*, bem como a melhor utilização dos registradores e memória disponíveis. Acredita-se que esse trabalho possa impulsionar os estudos nas áreas de *hardware*/*software codesign* e construção de compiladores com foco na exploração de paralelismo em arquiteturas não convencionais.

Palavras-chaves: Compilador, Gerador de Código, Simulação, *framework*.

Abstract

The development process hardware and software before sequential has been replaced by a competitor development model. This stems from continued growth the complexity of the systems and the shortening of the time-to-market window. Hardware projects almost always result in new paradigms, as well as insertion of instructions not supported in conventional compilers. The objective of this dissertation is to present the compiler and the Cognite framework, detailing the resources made available by this tool. The main focus of this work is to provide an infrastructure that accelerate the development of code generators for validation of hardware projects. A secondary but not less important objective is to use the framework for didactic purposes. To verify the capacity of code generation were performed experiments comparing the code obtained by the Cognite compiler and the compilers Clang and GCC. The results show that despite the low maturity and with a minimal set of optimizations Cognite has achieved a good positioning when compared to consolidated solutions like Clang and GCC, for the set of applications analyzed. The most important points are: reducing the amount of load and store operations, as well as the best use of available registers and memory. It is believed that this work can boost the studies in the areas of codesign hardware / software and construction of compilers focused on the exploration of parallelism in architectures not conventional.

Keywords: Compiler, Code Generator, Simulation, framework.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	*Application Programming Interface*
AST	*Abstract Syntax Tree*
CFG	*Context-free Grammar*
CPU	*Central Processing Unit*
RI	*Representação intermediária*

Sumário

1 Introdução

Atualmente pode-se observar um grande aumento no uso de computadores nas mais variadas áreas. Sistemas embarcados estão presentes em fábricas, equipamentos médicos, sistemas de fornecimento de energia, aeronaves e carros, em eletrodomésticos, como máquinas de lavar e geladeiras (BEETZ; BÖHM, 2012). A computação está presente em praticamente todos os aspectos da vida humana. Mas isso só foi possível devido aos avanços progressivos das tecnologias de produção de semicondutores. O avanço das tecnologias anteriormente mencionado provém da exploração de técnicas para obter aumento na densidade dos transistores (CATANZARO et al., 2010). Aumento esse previsto por Gordon Moore (MOORE, 2006). A crescente complexidade de *hardware* e *software* tem sido considerado um dos problemas mais sérios nos dias atuais (HALL; PADUA; PINGALI, 2009). A capacidade de produção de processadores munidos de centenas ou mesmo milhares de núcleos heterogêneos implica em desafios para explorar de maneira eficiente os recursos de hardware (CATANZARO et al., 2010). A qualidade do *software* para explorar o paralelismo fornecido pelo *hardware* será essencial para determinar o melhor desempenho das aplicação e sua capacidade em termos de velocidade de execução e consumo de energia.

Um dos grandes desafios na programação paralela consiste de explorar o potencial do hardware sem exigir um esforço indevido do programador (HALL; PADUA; PINGALI, 2009). Para isso são empregados grandes esforços quanto a produção de *frameworks* como: OpenMP (HOME. . . , 2018), CUDA (CUDA. . . , 2018) e OpenCL (OPENCL. . . ,). Os recursos também são explorados em linguagens como: Go (THE. . . , 2018a) e Rust (FREQUENTLY. . . , 2018a) que quando comparadas com a linguagem C apresentam uma sintaxe mais favorável ao desenvolvimento de aplicações concorrentes, além de lidar com os problemas de comportamento indefinido presentes na linguagem (LATTNER, 2011).

Ferramentas que forneçam um bom suporte para compilação são essenciais para plataformas de computação porque elas aumentam a capacidade de programação da plataforma (ADRIAANSEN et al., 2016). Esse aspecto é facilmente notado no caso das arquiteturas reconfiguráveis, onde cada aplicação necessita que uma configuração individual seja gerada e mapeada na arquitetura (PHANI; KRISHNA; SENAPATI, 2017). A grande maioria das ferramentas voltadas para o processo de compilação são direcionadas a linguagem C/C++. Um dos projetos de maior relevância na área é o LLVM (THE. . . , 2018).

O *framework* LLVM é uma excelente ferramenta para a construção e compiladores estáticos. O mesmo não pode ser observado quanto ao suporte a compilação dinâmicas (QINSB. . . , 2011). Como por exemplo a linguagem Go onde os desenvolvedores da

linguagem desencorajam a utilização do LLVM (FREQUENTLY..., 2018b).

Essa dissertação apresenta Cognite. Um solução composta por um *framework* e um compilador que possibilitam a construção de geradores de código. O *framework* possui uma arquitetura modular flexível que favorece a reutilização dos componentes. O compilador implementa um subconjunto da linguagem Go sendo capaz de gerar código para a arquitetura alvo MIPS. Os resultados obtidos nos experimentos mostram melhorias quando comparado com o compilador GCC e Clang, para o conjunto de aplicações analisado. O principal intuito não é concorrer diretamente com o LLVM e sim oferecer uma alternativa mais pratica e com bom resultado para a construção de geradores de código.

Cognite foi criado para sanar uma deficiência do laboratório CESLa (Circuit and Embedded System Lab). No que se refere a geração de código para validação das soluções arquiteturais desenvolvidas internamente. As soluções de arquitetura desenvolvidas apresentam recursos não suportados por compiladores convencionais. O que justifica o desenvolvimento desse trabalho.

1.1 Motivação

O laboratório CESLa desenvolve atividades de pesquisa na área de *hardware* com foco em computação paralela.

Cognite surgiu da necessidade de obter aplicações em um nível de assembly MIPS para a simulação das arquiteturas desenvolvidas no laboratório. Antes as aplicações eram geradas por meio da técnica de compilação cruzada utilizando a linguagem de programação C e o compilador GCC. Mas a metodologia apresentava algumas limitações como:

- o código gerado continha limitações quanto de usos de registradores;
- algumas das arquiteturas possuem instruções adicionais não suportadas pelo GCC;
- a sintaxe da linguagem C não é tão atrativa para programação paralela.

Tais limitações dificultavam uma total exploração dos recursos dos projetos. Para atender a essas demandas foi desenvolvido o *framework* e compilador Cognite. O *framework* é composto por uma estrutura modular de componentes reutilizáveis. O compilador suporta um subconjunto da linguagem Go e é capaz de gerar código para a arquitetura MIPS. A linguagem Go foi adotada por apresentar uma sintaxe que favorece o desenvolvimento de aplicações concorrentes. O intuito do projeto Cognite é prover uma conjunto de ferramentas que possibilitem a exploração dos casos específicos de cada arquitetura desenvolvida no laboratório.

1.2 Justificativa

Como justificava para essa dissertação podemos elencar:

- prover ferramentas que facilitem o estudo e pesquisa na área de compiladores e de síntese de *hardware*;
- permitam ao desenvolvedor obter um maior controle do processo de geração de código. Possibilitando a geração de múltiplas representações de um mesmo código com rearranjo de instruções;
- construção de uma ferramenta flexível capaz de se adaptar a diversos propósitos;
- projetos onde não existe suporte de sistema operacional (NEPOMUCENO, 2016; LUZ, 2016);
- Cognite tem a finalidade de auxiliar projetos menor mas não menos importantes.

1.3 Objetivos

O principal objetivo dessa dissertação é apresentar o *framework* e compilador Cognite. O conjunto de ferramentas proposto por esse trabalho tem proposito tanto didático quanto de pesquisa e tornam o processo de construção de geradores de código mais fácil e ágil com a disponibilização de abstrações para as diversas etapas do processo de compilação.

1.3.1 Objetivos específicos

Para alcançar o objetivo principal, foram realizados os seguintes objetivos específicos:

- definição de uma especificação de divisão do *framework* em módulos visando o reaproveitamento;
- desenvolvimento da especificação da linguagem da representação intermediária contendo as definições de tipos, operadores e conjunto de instruções;
- desenvolvimento de uma versão funcional do *framework*;
- construção de um compilador que implementa um subconjunto da linguagem Go para arquitetura MIPS utilizando o *framework*;
- foram realizados experimentos comparativos entre o código gerado pelo compilador Cognite e as ferramentas de compilação GCC e Clang.

1.4 Contribuições

As principais contribuições dessa dissertação foram:

- o desenvolvimento do *framework* Cognite para construção de gerados de código;
- um compilador que implementa um subconjunto da Go capaz de gerar código para arquitetura alvo MIPS;
- um motor de renderização capaz de interpolar um objeto fonte de dados com expressões contidas em um *string*;
- fornecer suporte quanto a geração de código e ferramentas de análise para outros projetos desenvolvidos no laboratório CESLA.

Espera-se que esse trabalho impulsione pesquisas nas áreas de *hardware/software codesign* e construção de compiladores com foco na exploração de paralelismo em arquiteturas não convencionais.

1.5 Organização da Dissertação

Essa dissertação está organizada da seguinte forma:

- Capítulo 1, **Introdução:** apresentamos a problemática e de maneira resumida a proposta do trabalho;
- Capítulo 2, **Fundamentação Teórica:** são expostos conceitos necessários para o melhor entendimento do trabalho;
- Capítulo 3, **Estado da Arte:** é apresentado o LLVM como principal ferramenta para a construção de compiladores encontrada na literatura. Além de ressaltar alguns argumentos que fornecem suporte a execução desse trabalho;
- Capítulo 4, **O *framework Cognite*:** apresenta, de maneira detalhada, os módulos do *framework*. São descritos os componentes e elementos que possibilitam a construção de geradores de código. Por fim, são abordadas algumas características do compilador Cognite;
- Capítulo 5, **Resultados experimentais:** são discutidos os resultados dos experimentos realizados comparando a solução Cognite com os compiladores Clang e GCC;
- Capítulo 6, **Conclusão e Trabalhos Futuros:** são apresentados os resultados em termo de ferramentas produzidas. Bem como a perspectiva de trabalhos futuros para melhorias e aplicações desse trabalho.

2 Fundamentação Teórica

Esse capítulo fornece informações básicas necessárias para o melhor entendimento da implementação do *framework* Cognite. São apresentados: alguns dos conceitos e ferramentas que possibilitam a construção de compiladores.

2.1 Compiladores

Um compilador consiste de um programa capaz de traduzir um outro programa, escrito em uma linguagem, chamado de código fonte em uma representação equivalente chamada de código objeto (PRICE; TOSCANI; UFRGS., 2000). O processo de construção de um compilador era considerado uma tarefa árdua e extremamente difícil. Mas com o aperfeiçoamento das técnicas e a produção de ambientes e algorítimos o processo se tornou bem mais simplificado (AHO et al., 2011).

Compiladores ocupam uma função muito relevante na computação. Pois tem o papel de traduzir as intenções do programador (geralmente em representação de alto nível, mais próxima do usuário) em uma representação de baixo nível (mais próxima da maquina). O processo de tradução deve garantir que a representação final possua mesmo valor semântico definido pelo programador. Um compilador também é responsável por informar possíveis erros ao programador (PARR, 2013). Esse recurso reforça a garantia e a qualidade do resultado da compilação.

O processo de compilação é composto por várias etapas como mostra a Figura 1. Cada etapa é responsável por uma atividade especifica. As atividades realizadas são: análise léxica, análise sintática, análise semântica, geração da representação intermediária, otimizações e geração de código objeto (AHO et al., 2011). As etapas são agrupas em *front-end*, otimizadores e *back-end* como mostra a Figura 2. A construção de uma representação intermediária é opcional. Embora a adoção dessa técnica favoreça uma maior flexibilidade e reaproveitamento das estruturas geradas pelas etapas anteriores. A representação intermediária permite que um mesmo programa seja traduzido para diferentes destinos(AHO et al., 2011). A etapa de otimização também é opcional, entretanto é essencial para a produção de um código que melhor se adéque aos recursos disponíveis. Como exemplo, deve-se considerar as aplicações desenvolvidas para sistemas embarcados onde a capacidade de processamento, memória e consumo de energia são limitados (LEUPERS, 2013).

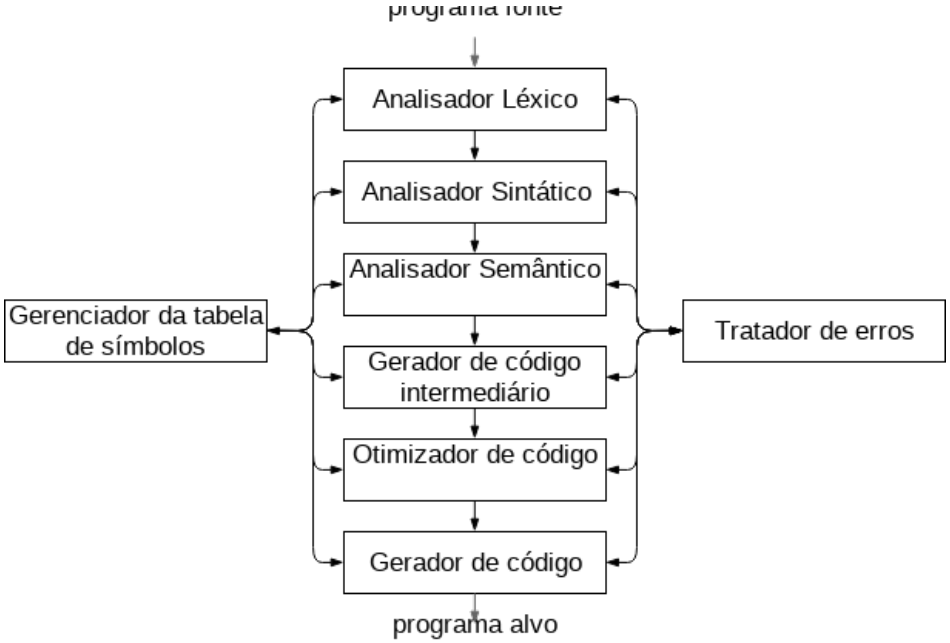

Figura 1 – Representação das fases de compilação.

Fonte – Adaptada de (AHO et al., 2011).

2.1.1 *Front-end*

O *front-end* é responsável por identificar e sinalizar erros sintáticos e semânticos no programa fonte. Ele executa as análises léxica, sintática e semântica. O resultado gerado pelo *front-end* é a representação intermediária.

2.1.1.1 Análise léxica

O processo de análise léxica também chamado de *scanning* compreende a extração dos *tokens* contidos no programa fonte. Um *token* corresponde a uma sequencia de caracteres com significado coletivo (AHO et al., 2011).

2.1.1.2 Análise sintática

O processo de análise sintática consome os *tokens* gerados pela etapa de análise léxica. Os *tokens* são agrupados em frases gramaticais e representados por uma árvore chamada de árvore sintática concreta. A árvore demostra a relação entre os *tokens*. Com isso é possível determinar o grau de precedência de expressões representadas por eles (AHO et al., 2011).

Nessa etapa são identificados os erros sintáticos. Como por exemplo, a ausência de um colchete na definição de um bloco.

2.1.1.3 Árvore Sintática Abstrata

A árvore gerada pela análise sintática também chamada de árvore sintática concreta apresenta todos os *tokens* identificados durante a análise léxica. Uma representação contendo as partes mais relevantes da árvore sintática concreta é chamada de árvore sintática abstrata. Nela são descartados os símbolos não necessários para as próximas etapas do processo de compilação (AHO et al., 2011).

2.1.1.4 Análise semântica

O processo de análise semântica percorre a árvore sintática gerada pela etapa de análise sintática e verifica a validade quando o contexto e considerado (AHO et al., 2011). São realizadas durante a análise semântica:

- Análise de escopo - verifica se um determinado identificador foi definido no escopo em que foi utilizado.
- Checagem de tipo - verifica se os valores atribuídos durantes as operações são validos. Dependendo da linguagem essa verificação pode ser executada em tempo de execução (BURROWS; FREUND; WIENER, 2003).

2.1.2 Representação intermediária (RI)

Os tipos de representação intermediária podem ser: lineares, com notação posfixa, representação de três endereços, com quádruplas, representação de maquina virtual, com o código de maquina de pilha e as representações gráficas árvores e grafos dirigidos acíclicos (AHO et al., 2011). [escrever mais sobre três enderelos]

2.1.3 *Otimizadores*

O fase de otimização representa as tarefas relacionadas a implementação de melhorias na representação intermediária que independem da arquitetura alvo. Isso envolve, entre outras, a remoção de instruções não alcançáveis, propagação de constantes e potencial reorganização de instruções.

2.1.4 *Back-end*

O *back-end* por sua vez é encarregado de traduzir a representação intermediária em código de baixo nível que pode ser executado em algum ambiente especifico. Durante o processo de tradução o *back-end* leva em consideração os aspectos específicos de cada destino podendo realizar otimizações que aproveitem os diferentes recursos de cada alvo. Um aspecto crucial é a atribuição de variáveis aos registradores (AHO et al., 2011).

2.1.4.1 Programa alvo

O resultado da geração de código é o programa-alvo. Um programa-alvo pode assumir várias formas entre elas: linguagem absoluta de máquina, linguagem realocável de maquina e linguagem de montagem (AHO et al., 2011). O processo de geração do programa-alvo implica em uma série de questões determinantes para a qualidade do resultado final.

- Gerenciamento e memória
- Seleção de instruções
- Alocação de registradores

2.1.5 Blocos básicos

Um bloco básico consiste de uma sequência de enunciados consecutivos, na qual o controle entra no início e o deixa no fim, sem uma parada ou possibilidade de ramificação, exceto ao final (AHO et al., 2011). O inicio de um bloco básico é sinalizado pela instrução líder. Em (AHO et al., 2011) um líder corresponde a:

- Primeira instrução;
- Qualquer instrução que seja objeto de um desvio condicional ou incondicional;
- Qualquer instrução que siga imediatamente uma instrução de desvio condicional ou incondicional.

O bloco básico compreende as instruções entre o seu líder e o próximo ou o final do programa.

2.1.6 Tipos de Compiladores

Um compilador pode gerar código para várias arquiteturas e representações diferentes (COMPILERS. . . ,). Quando um compilador gera código para ser executado na mesma maquina que ele é chamado de compilador de código nativo. Quando o código gerado é destinado a uma arquitetura alvo distinta é chamado de compilador cruzado (*cross-compile*) (AHO et al., 2011). Um compilador pode converter uma representação usualmente *bytecode* (JAVA. . . , 2001) em instruções de maquina alvo em tempo de execução. Nesse caso são chamados de compiladores JIT (*Just In Time*). Existem ainda compiladores que realizam a tradução de uma linguagem de alto nível para outra. Esses são chamados de compilador fonte-para-fonte (*source-to-source*) e podem, por exemplo, converter programas aplicando técnicas de paralelização e gerando código utilizando algum *framework* como *OpenMP* (HOME. . . , 2018).

2.1.7 Linguagem Go

Go é uma linguagem de programação de código aberto (LICENSE...,) que facilita a criação de software confiável e eficiente (THE..., 2018a). Go foi desenvolvida pela Google e seu projeto teve inicio em 2009 (GOOGLE'S..., 2009). Um dos aspectos mais relevantes da linguagem é o modelo de concorrência. Go implementa o modelo de concorrência conhecido como *Communicating Sequential Processes* (CSP)(ROSCOE, 1997) que baseia-se na comunicação utilizado canais. A sintaxe da linguagem permite a construção de aplicações concorrentes de maneira fácil quando compara com outras linguagens(POSIX..., 2017; DEFINING..., 2017). Essa subseção apresenta os recursos de programação concorrente oferecidos pela linguagem e que justificam a sua adoção como *front-end* do compilador Cognite.

2.1.7.1 *Goroutines*

Goroutine não são *threads*. *Goroutines* consistem da multiplexação de funções de execução independente (*coroutines*) executadas em conjunto com *threads* (FREQUENTLY..., 2018b). As goroutines são distribuídas entre um conjunto de *threads* e compartilham o mesmo espaço de endereçamento. Quando uma *coroutine* é bloqueada devido uma chamada de sistema. O *run-time* move as outras *coroutines* para outra *thread*. Assim o resto das atividades podem ser executas evitando o bloquei geral. Esse procedimento é abstrato ao programador o que torna a utilização da concorrência bem atrativa. A linguagem Go possui um mecanismo que aumenta e diminue o tamanho da pilha automaticamente. Esse mecanismo permite centenas de milhares de *goroutines* coexistam simultaneamente. Se *goroutines* fossem apenas *threads*, os recursos do sistema operacional se esgotariam em um numero inferior (FREQUENTLY..., 2018b).

Qualquer função pode ser convertida em uma *goroutine*. Basta que a palavra chave *go* seja inseria antes da chamada da função.

O exemplo Código 2.1 demonstra a utilização de uma *goroutine*. A função *main* cria uma *goroutine* da função *f* que é executada de maneira concorrente com a segunda chamada da função *f*. A instrução *fmt.Scanln* evita que o programa encerre sem que a *goroutine* seja concluída. Mecanismos mais avançados de controle são definidos na subseção 2.1.7.2 e na subseção 2.1.7.3.

```
1 package main
2 import "fmt"
3
4 func f(e string){
5     fmt.Println(e)
6 }
7
8 func main() {
9     go f("entrada 1")
10    f("entrada 2")
```

```
11     fmt.Scanln()
12 }
```

Código 2.1 – Exemplo de utilizaçao de uma goroutine na linguagem Go.

2.1.7.2 *Channels*

Os canais representam outro conceito chave para a obtenção da concorrência em Go. Canais são estruturas que permitem a comunicação entre *goroutines*. Eles possibilitam enviar um valor de uma *goroutine* para a outra. Cada canal possui um tipo e um tamanho. Apenas valores com o mesmo tipo do canal podem ser transferidos. Por padrão o processo de envio e recebimento bloqueia a execução das duas *goroutines* envolvidas (THE. . . , 2018a).

A sintaxe de envio e recebimento é definida pela utilização dos operadores <- e <- associados a um canal. Um exemplo de utilização de canais pode ser visualizado na Código 2.2.

```
1 package main
2
3 func main() {
4     mensagens := make(chan string)
5     go func(){mensagens <- "ping"}()
6     mensagem := <- mensagens
7 }
```

Código 2.2 – Exemplo de utilizaçao de canais na linguagem Go.

Para implementar um canal não bloqueante o canal deve ser utilizado em junção com a operação *select*. Durante a execução do programa a operação *select* avalia o estado do canal e verifica se existe a possibilidade de envio ou recebimento. Caso contrário assume o caso padrão. Um exemplo está disponível em Código 2.3.

```
1  package main
2  import "fmt"
3
4  func main() {
5      mensagens := make(chan string)
6
7      go func(){mensagens <- "ping"}()
8
9      select{
10         case mensagem := <- mensagens
11             fmt.Println(mensagem)
12         default:
13             fmt.Println("Sem mensagem")
14     }
15 }
```

Código 2.3 – Exemplo de utilizaçao de canais não bloqueantes na linguagem Go.

Um canal pode ser encerrado utilizando a chamada *close(channel)* onde *channel* é a variável que representa o canal.

2.1.7.3 Sincronização

A linguagem Go possui um pacote nomeado *sync* que prove mecanismos para sincronização entre as *goroutines* (THE..., 2018a). O pacote conta com a implementação de um *Mutex* que prove a exclusão mútua. Além disso, apresenta uma estrutura chamada *WaitGroup* para controle de execução. Os exemplos Código 2.4 e Código 2.5 demonstram a utilização do *Mutex* e do *WaitGroup* respectivamente. No exemplo Código 2.4 as instruções entre a chamada *Lock* e *Unlock* estão protegidas de acessos simultâneos.

```
1 package main
2 import "sync"
3 func main() {
4     var mutex = &sync.Mutex{}
5     /* ... */
6     mutex.Lock()
7     /* ... */
8     mutex.Unlock()
9 }
```

Código 2.4 – Exemplo de utilizaçao da estrutura Mutex na linguagem Go.

O exemplo Código 2.5 demonstra a execução de três *goroutines*. Para cada *goroutine* o tamanho do WaitGroup é incrementado em um. Após o laço a rotina main aguarda a conclusão da execução de todas as *goroutines*. A conclusão de uma *goroutine* é sinalizada por meio da chamada do método *Done* do *WaitGroup* dentro da *goroutine*. O método *Done* decrementa a quantidade do grupo. Quando o valor for igual a zero o fluxo da rotina main é retomado.

```
1 package main
2 import "sync"
3 func f(i int, wg *sync.WaitGroup){
4     /* ... */
5     wg.Done()
6 }
7
8 func main() {
9     var wg sync.WaitGroup
10    for i:= 0; i < 3; i++ {
11        wg.Add(1)
12        go f(1, &wg)
13    }
14    wg.Wait()
15 }
```

Código 2.5 – Exemplo de utilizaçao da estrutura WaitGroup na linguagem Go.

3 Estado da Arte

Esse capítulo introduz a ferramenta de maior relevância da literatura relacionados a construção de compiladores, o LLVM. São apresentados: uma descrição do LLVM, alguns dos trabalhos que adotam LLVM como suporte e os pontos positivos e negativos da ferramenta. São ressaltados alguns argumentos que apoiam a execução desse trabalho. Por fim, uma breve descrição da solução Cognite.

3.1 Trabalhos Relacionados

O nome LLVM era um acrônimo para *low level virtual machine*. Apesar do nome o projeto tem pouco a ver com máquinas virtuais. Com o crescimento do escopo do projeto LLVM deixou de ser um acrônimo. LLVM é um projeto *open-source*(THE. . . , 2018) que começou no ano 2000 na Universidade de Illinois. É construído em C++ e originalmente foi projetados para fornecer um compilador moderno capaz de suportar compilação estática e dinâmica (THE. . . , 2018). A infra-estrutura do compilador LLVM é composta por uma coleção de ferramentas e de módulos reutilizáveis.

O LLVM implementa um projeto de compilador de três fases consistindo de um *front-end*, otimizadores e um *back-end* como mostra a Figura 2. O projeto conta com uma representação intermediaria independente do alvo capaz de representar os diversos recursos das linguagens de alto nível (LLVM, 2018).

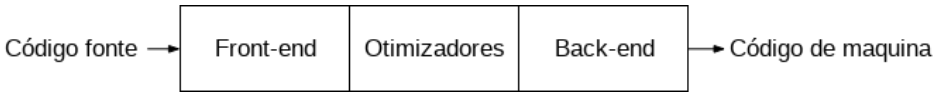

Figura 2 – Representação de um compilador de três fases.

Fonte – Adaptada de (LLVM, 2018).

A representação intermediária é implementada usando *Static Single Assignment form* (SSA)(CYTRON et al., 1991) e viabiliza o modelo de arquitetura *retargetable* como mostra a Figura 3.

A SSA é uma propriedade de uma representação intermediária onde cada variável deve receber uma única atribuição e deve ser definida antes de ser usada. A utilização da SSA simplifica alguns processos de otimização como eliminação de operações redundantes.

LLVM permite a construção de módulos que podem ser inseridos nas fases de compilação. Como por exemplo, quando um novo *back-end* é adicionado ao compilador todas as otimizações e linguagens suportadas são reutilizadas. O modelo reduz o esforço para o desenvolvimento de um novo compilador (LLVM, 2018).

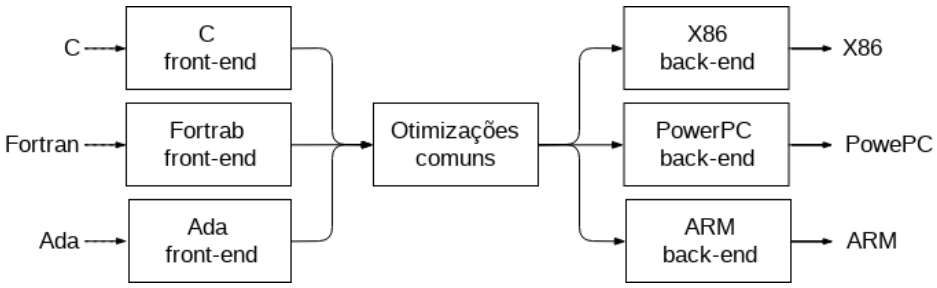

Figura 3 – Representação de uma arquitetura *retargetable*.

Fonte – Adaptada de (KOLEK et al., 2013).

O LLVM possui vários subprojetos. Por exemplo, Clang que corresponde a um implementação do *front-end* para as variações da linguagem C. As variações suportadas são: C, C++, Objective C e Objective C++ (CLANG. . . ,). Assim como o LLVM ele é composto por um conjunto de bibliotecas que permitem manipular e estender as linguagens suportadas. O maiores objetivos do Clang consistem de oferecer um bom diagnostico e recuperação de erros, bem como integração fácil com ambientes de desenvolvimento. Clang foi projetado para ser compatível com o GCC e permite uma substituição imediata (LANGUAGE. . . , 2018).

O LLVM é amplamente utilizado em projetos de pesquisa. Por exemplo, no trabalho de Tian et al. (2016) são inseridas extensões na representação intermediária do LLVM para fornecer suporte a programação paralela explicita. As extensões permitem a transformação no *middle-end* do LLVM para oferecer suporte as APIs do OpenMP, C/C++ e Fortran. Em Nezzari e Bridges (2018) o LLVM foi empregado no desenvolvimento de uma ferramenta de injeção de falhas para validar e avaliar métodos de proteção de *software* em tempo de compilação e de execução para múltiplos erros como corrupção de dados silenciosa.

Khaldi e Chapman (2016) apresentam uma nova ferramenta de análise inserida no etapa de passe do LLVM, chamada *Bandwidth-Critical Data Analysis* (BCDA). A ferramenta avalia quando é benéfico alocar dados em *High-Bandwidth Memory* (HBM) (JUN et al., 2017), visando o aumento do desempenho em sistemas paralelos. Com base na análise da representação intermediaria gerada pelo LLVM é extraída a quantidade e frequência de acessos a memória. Os valores são utilizados para determinar quando chamadas de *mallocs* devem ser convertidas em alocações de HBM.

Existem também vários projetos abertos e comerciais que utilizam LLVM. Como exemplo, nvcc da *Nvidia* (CUDA. . . , 2018) e o compilador de linguagem *Swift* (SWIFT. . . , 2018a).

A prevalência hegemônica do LLVM está relacionada ao fato de não existirem opções semelhantes com a mesma disponibilidade. A grande maioria das soluções alternativas atendem a algum subconjunto especifico de características e quase sempre são destinadas

a compilação da linguagem C. ROSE é uma infraestrutura *open-source* para construção de compiladores *source-to-source* provendo ferramentas para transformação e análise para a linguagem C, C++, OpenMP e Java (ROSE..., 2018). Open64 é uma ferramenta *open-source* que possui um compilador para arquiteturas Intanium e x86-64 com suporte a linguagem Fortran 77/95 e C/C++. O mesmo usa uma representação intermediária chamada WHIRL com múltiplos níveis de representação que funcionam como uma interface entre as diversas fases da compilação (DE et al., 2018). Qbe pretende ser um back-end puramente C. Ele fornece integração trivial e grande flexibilidade (QBE..., 2018).

Uma grande quantidade de linguagens são desenvolvidas sobre a infraestrutura do LLVM. Dentre elas estão inclusas: Haskell (LLVM-HS...,), Julia (THE..., 2018b), Kotlin (KOTLIN/NATIVE..., 2018), Lua (LUA-USERS..., 2018), Rust (FREQUENTLY..., 2018a) e Swift (SWIFT..., 2018b).

Indícios apontam para uma redução da utilização do projeto LLVM em detrimento de outras soluções. Um gráfico disponibilizado na pagina do projeto LLVM e adaptado na Figura 4 indica uma redução na quantidade de publicações referentes a ferramenta.

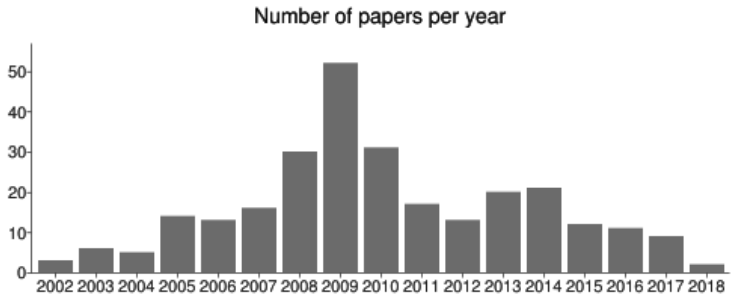

Figura 4 – Número de *papers* por ano.

Fonte – Adaptada de (THE...,).

Uma consulta ao termo LLVM no *Google Trends* (GOOGLE..., 2018) constata uma redução do interesse nas consultas sobre o termo no intervalo que compreende o ano de 2013, onde obteve a maior relevância, para o período atual no ano de 2018. Como exposto na Figura 5.

O LLVM é uma grande compilador para linguagens com tipagem estática, mas sabe-se que ele não funciona igualmente bem no contexto de linguagens dinâmicas (GIVING..., 2016). A Google em sua pagina da linguagem Go descreve o LLVM como uma ferramentas muito grande e lenta para atingir as metas de desempenho que desejavam (FREQUENTLY..., 2018b). Em uma análise da linguagem de programação Julia é relatado que um simples *hello world* usa 18 vezes mais memória que a versão em Python e 92 vezes mais que a versão em C. Quando comparado o tempo de execução um programa em Julia executa 27 vezes mais lento que a versão em Python e 187 vezes mais lento que a mesma versão em C. Os desenvolvedores da linguagem Julia atribuem o motivo a utilização do

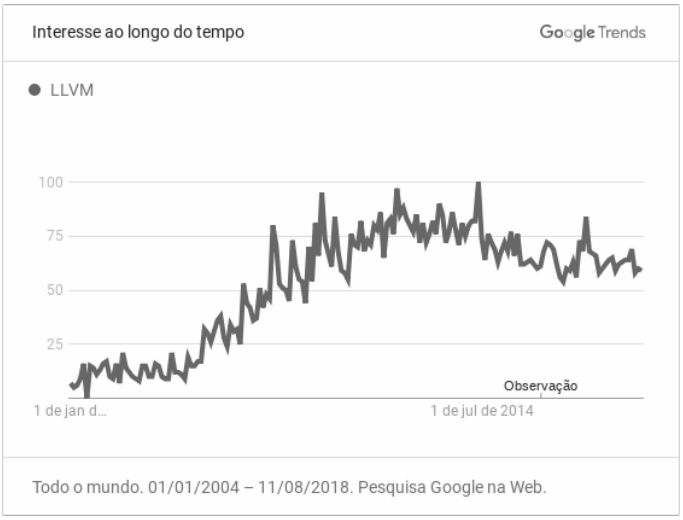

Figura 5 – Índice de interesse do termo LLVM em todo o mundo entre 2004 e 2018.

Fonte – Adaptada de (LLVM...,).

LLVM (GIVING..., 2016). Problemas com elevado tempo de compilação são relatados na linguagem Rust. Onde análises em ferramentas de *debug* e optimização mostram que entre 70 e 80% da quantidade de instruções executadas durante o processo de compilação são referentes ao LLVM (NICHOLAS..., 2018).

O projeto *WebKit's* FTL JIT (*Faster Than Light Just In Time compiler*) abandonou a utilização do *framework* LLVM e passou a utlizar o B3 (*Bare Bone Backend*) como otimizador de baixo nível (INTRODUCING..., 2016). A mudança ocasionou uma aceleração entre 5 e 10 vezes no tempo de compilação (CHANGESET..., 2016).

Nesse contexto está inserido a proposta desse trabalho. A solução Cognite consiste de um *framework* e um compilador. O *framework* é composto por módulos destinado a construção de geradores de código independente da arquitetura alvo. Isso é possível graças a um conjunto de estruturas abstratas de representações intermediaria e código alvo. Existem também mecanismos que controlam o fluxo da compilação e análise. O compilador faz uso de um subconjunto da sintaxe da linguagem Go, favorável o desenvolvimento de aplicações concorrentes, para a geração de código alvo MIPS.

3.1.1 Considerações Finais

Esse capítulo apresentou o estado da arte no que se refere as ferramentas para a construção de compiladores. Foi abordado o LLVM como a principal ferramenta disponível apresentando alguns dos inúmeros trabalhos que utilizam o *framework* como suporte. Também foi introduzindo uma breve descrição da solução Cognite. O Capítulo 4 descreve de maneira detalhada o *framework* Cognite. Enquanto que o compilador é abordado na seção 4.9.

4 O framework Cognite

Esse capítulo expõe uma descrição geral do *framework* e um visão detalhamento dos recursos disponibilizados. São apresentados os principais componentes do *framework* bem como suas interações para a construção de geradores de código.

4.1 Descrição geral

Cognite é um *framework* para construção de geradores de código. Ele fornece aos desenvolvedores mecanismos de controle do processo de geração de código bem como sua representação. Os mecanismos inclusos são:

- APIs de mapeamento de estruturas;
- Árvore Sintática Abstrata;
- Modelo de representação de código (MRC);
- Representação Intermediaria;
- Geradores;
- Motor de renderização;
- Maquina virtual de aplicações.

O principal objetivo do *framework* é disponibilizar uma infraestrutura básica para construção de ferramentas de compilação.

Cognite tem a capacidade de representar tanto compiladores cruzados como fonte-para-fonte definidos na subseção 2.1.6. Para isso ele conta com uma estrutura modular. Um representação dos módulos está disponível na Figura 6. O nome em itálico corresponde ao nome do pacote no *framework* e é utilizado para sinalizar as classes quando referenciadas no texto. Cada módulo desempenha uma função e pode ser reutilizado em diferentes projetos. O desenvolvedor é livre para explorar individualmente um módulo ou compor soluções conjuntas.

O *framework* conta com ferramentas para todas as etapas do processo de compilação, exceto as etapas de análise léxica e sintática. Entretanto o desenvolvedor pode adotar alguma das soluções existentes (PARR, 2013; CUP, 2018; JAVACC..., 2018). Embora recomenda-se a utilização do ATNLR versão 4 por motivos de compatibilidade com o projeto do compilador descrito no seção 4.9.

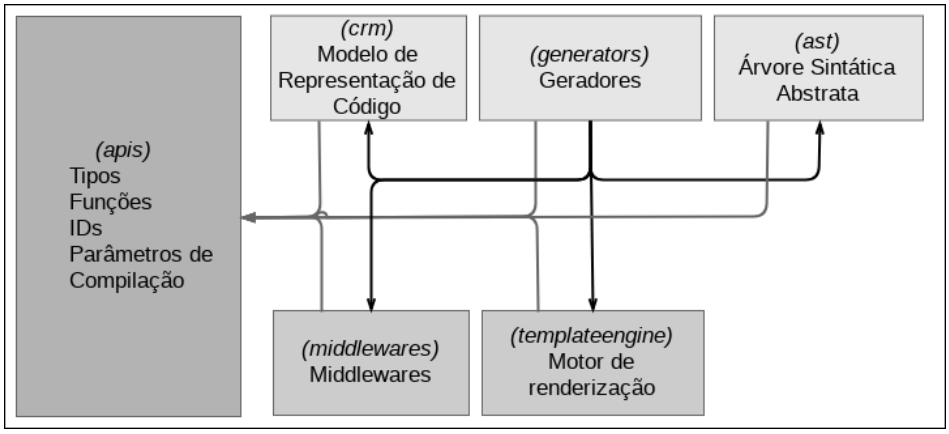

Figura 6 – Organização dos módulos do *framework* Cognite.

Fonte – Elaborada pelo autor.

As seções seguintes descrevem de maneira mais detalhada os recursos disponíveis em cada módulo.

4.2 APIs

O módulo de APIs corresponde ao gerenciador da tabela de símbolos e é responsável por manipular as estruturas básicas do *framework*. As APIs possibilitam acesso fácil aos recursos do *framework* em qualquer etapa do processo de compilação. As subseções seguintes apresentam as estruturas básicas e APIS contida no módulo.

4.2.1 Estruturas básicas

As estruturas gerenciadas pelo módulo API são apresentadas em ordem alfabética e da seguinte forma. Primeiro é exibido o nome do recurso seguido pela definição, uma descrição e os métodos disponíveis. Para encurtar a definição sempre que mencionar um tipo deve-se considerar os seguintes premissas. Um tipo precedido do símbolo '*' indica um ponteiro. A quantidade de '*' deve respeitar as especificações da linguagem que se deseja representar.

ID:

Definição: um ID é uma instância da classe *Cognite.Apis.ID*.

Descrição: A classe define uma estrutura que representa uma variável ou constante ná linguagem de alto nível.

Métodos:

Construtor:

Formato: *new ID*(*String id*, *String type*, *Boolean constant*)

Descrição: O método cria uma instância de um ID. O parâmetro *id* corresponde ao nome do identificador, *type* o tipo e *constant* determina se o identificador representa uma contante. Constantes não podem ser definidas como ponteiros.

GetId:

Formato: *String GetId*()

Descrição: O método retorna o nome do identificador.

GetType:

Formato: *String GetType*()

Descrição: O método retorna o tipo do identificador.

IsPtr:

Formato: *Boolean IsPtr*()

Descrição: O método retorna verdadeiro caso o tipo do identificador for um ponteiro. Caso contrário retorna falso.

SetId:

Formato: *ID SetId*(*String id*)

Descrição: O método define o nome do identificador.

SetType:

Formato: *ID SetType*(*String type*)

Descrição: O método define o tipo do identificador.

Function:

Definição: uma *Function* é uma instância da classe *Cognite.Apis.Function*.

Descrição: a classe define uma estrutura que representa uma função ou um método. O tipo Arg mencionado no formato dos métodos denota uma estrutura composta por um id e um tipo ambos *String*.

Métodos:

Construtor:

Formato: *new Function*(*String id*)

Descrição: O método cria uma instância de um função. O parâmetro id corresponde ao nome da função.

GetArgs:

Formato: *List<Arg> GetArgs*()

Descrição: O método retorna a lista de argumentos de chamada da função.

GetReturns:

Formato: *List<Arg> GetReturns*()

Descrição: O método retorna a lista de argumentos da retorno.

IsMethod:

Formato: *Boolean IsMethod*()

Descrição: O método retorna verdadeiro caso a função possua um contexto.

SetArgs:

Formato: *Function SetArgs*(*List<Arg> args*)

Descrição: O método define a lista de argumentos de chamada da função. O método retorna a instância da própria função.

SetContext:

Formato: *Function SetContext*(Arg arg)

Descrição: O método especifica o tipo e o id do contexto da função. Uma função com contexto representa um método de um tipo especifico. O método retorna a instância da própria função.

SetReturns:

Formato: *Function SetReturns*(*List<Arg> args*)

Descrição: O método define a lista de argumentos da retorno.

Type:

Definição: um *Type* é uma instância da classe *Cognite.Apis.Type*.

Descrição: A classe define uma estrutura que representa um tipo na linguagem de alto nível. Nele são mapeados os atributos e se existe alguma relação de dependência com outro tipo da linguagem.

Métodos:

Construtor:

Formato: *new Type*(*String id*)

Descrição: O método cria uma instância de um tipo. O parâmetro id corresponde ao nome do tipo.

AddAttrib:

Formato: *Type AddAttrib*(*String id, String type, Integer count*)

Descrição: O método adiciona um novo atributo ao tipo. O parâmetro id especifica o nome do atributo, *type* o tipo e *count* a quantidade de elementos. O atributo *count* quando maior que 1 representa um *array*. O atributo id quando vazio define uma extensão do tipo especificado pelo atributo *type*, nesse caso o atributo *count* deve apresentar valor 1.

GetId:

Formato: *String GetId*()

Descrição: O método retorna o identificador do tipo.

GetLayout:

Formato: *List<String> GetLayout*()

Descrição: O método retorna a lista com os atributos do tipo. A ordem padrão dos atributos corresponde a ordem de declaração. O *layout* define a organização dos atributos do tipo na memória.

HasAttrib:

Formato: *Boolean HasAttrib*(*String id*)

Descrição: O método retorna verdadeiro caso o tipo apresente o atributo especificado no parâmetro id. Caso contrário retorna falso.

HasMethod:

Formato: *Boolean HasMethod*(*String id*)

Descrição: O método consulta a API de funções e retorna verdadeiro caso exista uma método equivalente definido. Caso contrário retorna falso. Um método de um tipo deve ser registrado na API de funções. A função registrada deve ter o tipo como contexto.

IsArray:

Formato: *Boolean IsArray*(*String id*)

Descrição: O método retorna verdadeiro caso o atributo especificado no parâmetro id for um *array*. Caso contrário retorna falso.

IsPtr:

Formato: *Boolean IsPtr*(*String id*)

Descrição: O método retorna verdadeiro caso o atributo especificado no parâmetro id seja um ponteiro. Caso contrário retorna falso.

OffsetOf:

Formato: *Integer OffsetOf*(*String id*)

Descrição: O método retorna a posição em bytes do atributo especificado no parâmetro id. Um exceção é lançada quando o atributo não existe.

RemoveAttrib:

Formato: *Type RemoveAttrib*(*String id*)

Descrição: O método remove o atributo especificado no parâmetro id caso exista.

SetLayout:

Formato: *Type SetLayout*(*List<String id> layout*)

Descrição: O método define a posição de cada atributo do tipo. A ordem dos atributos corresponde a posição na lista recebida no parâmetro *layout*. O *layout* define a organização dos atributos do tipo na memória.

Size:

Formato: *Integer Size*()

Descrição: O método retorna o tamanho do tipo em bytes.

SizeOf:

Formato: *Integer SizeOf*(*String id*)

Descrição: O método retorna o tamanho do atributo em bytes caso seja definido. Caso contrário retorna o tamanho 0. O tamanho de um atributo corresponde ao produto do tamanho do tipo do atributo pela quantidade de elementos. Ponteiros tem o tamanho de uma palavra da arquitetura alvo.

4.2.2 Api de parametros de compilção

A api de parâmetros de compilação está definida na classe estática *Cognite.Apis.BuildParams*. Ela tem a função de controlar as definições do usuário, como variáveis de controle, pertinentes ao processo de compilação que o desenvolvedor julgar necessário.

Os parâmetros podem ser definidos e acessados em qualquer etapa da compilação. A api de parâmetros possui o seguintes métodos:

Add:

Definição: *void Add*(*String tag*, *String value*)

Descrição: O método adiciona um novo valor na lista de valores do parâmetro referenciado pelo parâmetro *tag*. Se não existe uma definição anterior o mesmo é criado e em seguida o valor é inserido.

Get:

Definição: *List<String> Get*(*String tag*)

Descrição: O método *Get* retorna uma lista contendo todos os valores atribuídos ao parâmetro referenciado pela parâmetro *tag*. Quando o parâmetro não existe a função retorna uma lista vazia.

GetFirst:

Definição: *String GetFirst*(*String tag*)

Descrição: O método *GetFirst* retorna o primeiro valor da lista de valores do parâmetro *tag*. Quando o parâmetro não existe a função retorna uma string vazia.

Has:

Definição: *Boolean Has(String tag)*

Descrição: O método *Has* retorna o valor verdadeiro caso o parâmetro referenciado pela *tag* tenha sido definido, caso contrário retorna *falso*.

Set:

Definição: *void Set(String tag, ArrayList<String> values)*

Descrição: O método registra um novo parâmetro. Se existe uma definição anterior os valores são sobrescritos.

4.2.3 Api de Tipos

A api de tipos está definida na classe estática *Cognite.Apis.Types*. Ela é responsável por gerenciar as instâncias do tipo *Cognite.Apis.Type*. A api controla as declarações de novos tipos. Conta com mecanismos que permitem a definição de tipos separados por escopo aumentando a sua capacidade de representação.

A seguir são listados os métodos disponíveis na api. Os atributos *id* e *scope* nos formatos representam respectivamente o nome do tipo e o local onde o tipo foi declarado.

Defined:

Formato: *Boolean Defined(String id, String scope)*

Descrição: O método retorna verdadeiro caso exista uma definição do tipo para o escopo determinado. Caso contrário retorna false.

Get:

Formato: *Type Get(String id, String scope)*

Descrição: O método retorna a instância do tipo caso exista uma definição para o escopo determinado. Caso contrario retorna nulo.

GetAllByScope:

Formato: *List<Type> GetAllByScope(String id, String scope)*

Descrição: O método retorna uma lista de instâncias dos tipos definidas no escopo especificado.

Register:

Formato: *Type Register*(*String id*, *String scope*)

Descrição: O método registra um novo tipo no escopo determinado. A api retorna uma instância do tipo *Cognite.Apis.Type* que deve ser utilizada para a inserção dos atributos e métodos.

SizeOf:

Formato: *Integer SizeOf*(*String id*, *String scope*)

Descrição: O método retorna o tamanho de um tipo. Caso o tipo não tenha sido definido o tamanho retornado é 0.

4.2.4 Api de Funções

A api de funções está definida na classe estática *Cognite.Apis.Functions*. Ela é responsável por gerenciar as instâncias do tipo *Cognite.Apis.Function*. A api realiza o controle das declarações e verificações de uma Função. Conta com mecanismos que permitem a definição de funções separadas por escopo aumentando a sua capacidade de representação.

A seguir são listados os métodos disponíveis na api. Os atributos *id* e *scope* nos formatos representam respectivamente o nome da função ou método e o local onde a função foi declarada. O parâmetro *context* contem o valor do tipo. Quando presente indica que a função é um método do tipo referenciado.

Defined:

Formato: *Boolean Defined*(*String id*, *String scope*, *String context*)

Descrição: O método retorna verdadeiro caso exista uma definição de função para o escopo determinado. Caso contrário retorna false. O atributo *context* é opcional e tem a função de diferenciar uma função de um método.

Get:

Formato: *Function Get*(*String id*, *String scope*, *String context*)

Descrição: O método retorna a instância da função caso exista uma definição para o escopo determinado. Caso contrario retorna nulo. O atributo *context* é opcional e tem a função de diferenciar uma função de um método.

GetAllByScope:

Formato: *List<Function> GetAllByScope*(*String id*, *String scope*)

Descrição: O método retorna uma lista de instâncias das funções definidas no escopo especificado.

Register:

Formato: *Function Register*(*String id*, *String scope*, *String context*)

Descrição: O método registra uma nova função ou método no escopo determinado. A api retorna uma instância de *Cognite.Apis.Function* que deve ser utilizada para especificar os argumentos de chamada e retorno. O atributo *context* é opcional e tem a função de diferenciar uma função de um método.

4.2.5 Api de ID

A api de IDs está definida na classe estática *Cognite.Apis.IDs*. Ela é responsável por gerenciar as instâncias do tipo *Cognite.Apis.ID*. A api controla a declaração e verificação de IDs. Conta com mecanismos que permitem a definição de IDs separados por escopo aumentando a sua capacidade de representação.

A seguir são listados os métodos disponíveis na api. Os atributos *id* e *scope* nos formatos representam respectivamente o nome do identificador e o local onde o identificador foi declarado.

Defined:

Formato: *Boolean Defined*(*String id*, *String scope*)

Descrição: O método retorna verdadeiro caso exista uma definição do ID para o escopo determinado. Caso contrário retorna false.

Get:

Formato: *ID Get*(*String id*, *String scope*)

Descrição: O método retorna a instância do identificador caso exista uma definição para o escopo determinado. Caso contrario retorna nulo.

GetAllByScope:

Formato: *List<ID> GetAllByScope*(*String id*, *String scope*)

Descrição: O método retorna uma lista de instâncias de identificadores definidas no escopo especificado.

Register:

Formato: *ID Register*(*String id*, *String scope*,*String type*)

Descrição: O método registra um novo identificador no escopo determinado. A api retorna uma instância do tipo *Cognite.Apis.ID*.

4.3 *Middlewares*

Essa seção aborda com detalhes o conceito de middlewares no *framework* Cognite.

Um *middleware* é um componente flexível e reutilizável que define uma rotina a ser executada antes e/ou depois da execução de um código principal em alguma etapa do ciclo de um gerador. Geradores são definidos na seção 4.4. Um *middleware* corresponde a uma instância da classe *Cognite.Generator.Middleware*.

Um *middleware* manipula o modelo de representação de código (MRC) e possibilita a extensão das funcionalidades de um gerador. Existem dois tipos de *middelware*: os de análise e de transformação. Um *middleware* de análise extrai e armazena informações de um MRC, não realizando qualquer modificação na representação do código. Entretanto o *middleware* de transformação modifica o MRC, como por exemplo, quando instruções são removidas durante o processo de otimização.

Um *middleware* apenas será executado se atender as seguintes restrições:

1. Estiver registrado na api de *middlewares*;
2. Estiver vinculado a um evento em uma instância de um MRC;
3. A instância do MRC sinalizou o evento em que o *middleware* foi vinculado;
4. Nenhum *middleware* anterior a ele gerou erro durante sua execução.

Os *Middlewares* fornecem dados estatísticos que permitem avaliar qual a carga de trabalho e a quantidade de recursos (memória e tempo) foram alocados para sua execução. Esse mecanismo permite avaliar o impacto da implementação de um *middleware*.

4.3.1 Interface de um *Middleware*

Todo *middleware* deve implementar a interface *Cognite.Generator.MiddlewareInterface* que define um único método *Exec* responsável por descrever o comportamento desejado, análise ou transformação. O método *Exec* recebe como parâmetros:

- Code: corresponde a instância do MRC que sinalizou o evento;
- Map<String, Middleware>: equivale a um *map* contendo todos os *middlewares* vinculados ao evento. O mapa possibilita acessar informações geradas por outros *middlewares*. O identificador do *middleware* corresponde a chave no mapa.

4.3.2 Criando um *middleware*

A construção de um *middleware* consiste de 3 etapas:

Implementação - corresponde a definição de uma classe que implementa a interface MiddlewareInterface contendo a tarefa a ser executada. Um exemplo da definição de uma classe está disponível em Código 4.1.

```
1 public class M1Middleware  implements MiddlewareInterface {
2    @Override
3    public void Exec(Code, Map<String,Middleware>) throws Exception {
4       //acao a ser executada
5    }
6 }
```

Código 4.1 – Descrição de um *Middleware*.

Registro - todo *middleware* deve ser registrado na api *Cognite.Generator.Middlewares*. Esse procedimento é necessário para o reconhecimento por parte do gerenciador de eventos. O registro de um *middleware* ocorre por meio da chamada representada no exemplo Código 4.9.

```
1    Middlewares.Add("OP.m1", new M1Middleware());
```

Código 4.2 – Registrando um *Middleware*.

Vinculação - nessa etapa o *middleware* é vinculado a um evento de um determinado MRC. No exemplo Código 4.3, o *middleware* com identificador "OP.m1"foi vinculado ao evento "after.translate"da instância de código "C1".

```
1    Middlewares.On("C1", "after.translate", "OP.m1");
```

Código 4.3 – Vinculação de um *Middleware*.

Um MRC pode executar mais de um *middleware* por evento. Para isso, durante a etapa de vinculação deve ser especificada a lista com os identificados dos *middlewares* separados por virgula. A ordem de execução corresponde a ordem da definição.

4.3.3 Convenções

A implementação de um *middleware* deve seguir as seguintes convenções:

- O nome da classe deve encerrar com a palavra chave "*Middleware*";
- O id de registro deve ser composto por duas partes separadas por o carácter ”.” . A primeira parte deve fazer menção ao identificador do MRC a qual se destina o *middleware*. A pratica aumenta a legibilidade e evita que o *middleware* seja sobrescrito acidentalmente.

4.4 Geradores

Geradores são estruturas responsáveis por controlar o fluxo de construção de um modelo de representação de código (MRC), definindo o conjunto de ações tomadas para tratar cada tipo de entrada. Também são responsáveis por gerenciar o acionamento dos *middlewares* descritos na seção seção 4.3. *Cognite* conta com dois tipos de geradores:

- APM - é o gerador responsável pela conversão de uma árvore sintática abstrata em um modelo de representação de código. São utilizados na construção da representação intermediaria;
- MPM - é o gerador responsável pela conversão de um modelo de representação de código em outro. São utilizados na tradução da representação intermediaria em código alvo.

Devido a natureza distinta das entradas cada gerador apresenta um comportamento diferenciado. As particularidades de cada um dos geradores são descritas nas duas subseções seguintes.

4.4.1 Gerador APM

O gerador APM converte uma árvore sintática abstrata em um modelo de representação de código. O gerador é uma instância da classe *Cognite.Generator.ASTConverter*. O gerador apresenta os seguintes métodos:

Convert

Formato: *Code Convert*(*Node ast*)

Descrição: O método realiza a conversão da árvore sintática abstrata em um modelo de representação de código.

GetIr

Formato: *Code GetIr*()

Descrição: O método retorna a instância atual do modelo de representação de código referente a representação intermediária. O método permite acessar o MRC durante o processo de conversão.

RegisterConvertFunction

Formato: *void RegisterConvertFunction*(*String class*, *ConvertFunction fn*)

Descrição: O método registra um função de conversão para a classe especificada pelo parâmetro *class*. O método sobrescreve qualquer declaração prévia para a mesma classe.

O processo de conversão inicia com a chamada do método *Convert* do gerador. O gerador sinaliza o evento *before_convert* e percorre a árvore sintática abstrata.

Durante o processo de conversão a árvore sintática abstrata é percorrida recursivamente. O gerador busca por nós agregados identificados por classes com valor semântico definido. Quando o gerador visita um nó, ele verifica a existência de uma função de conversão com identificador equivalente ao valor da classe do nó. Caso exista uma definição a função é chamada passando como parâmetro o nó visitado, o gerador e a instância do modelo de representação de código que está sendo construído.

Quando a classe do nó visitado corresponde a *func.decl*, o gerador cria e empilha um novo grupo de instruções e sinaliza o evento *open_group*. Quando todos os nós filhos são visitados o gerador sinaliza o evento *close_group* e desempilha o grupo de instruções. Uma representação dos eventos sinalizados pelo gerador APM está disponível na Figura 7.

O processo de conversão encerra quando todos os nós da árvore sintática abstrata forem visitados ou quando um erro finaliza prematuramente a conversão. O evento *after_convert* apenas é sinalizado quando o processo de tradução encerra sem erros.

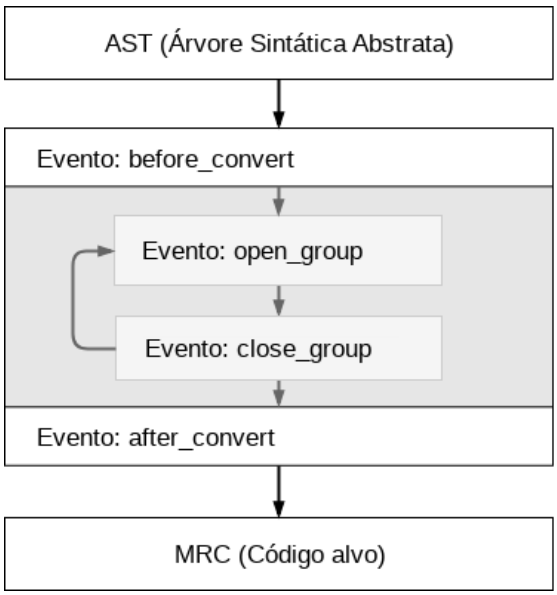

Figura 7 – Mapa de eventos do gerador APM.

Fonte – Elaborada pelo autor.

4.4.2 Gerador MPM

O gerador MPM executa a tradução entre modelos de representação de código. O processo de tradução é dividido em etapas. Uma representação sequencial do fluxo de tradução pode ser visualizado na Figura 8. O processo encerra quando todos as etapas são executadas ou uma falha em algum dos mecanismos acarreta o encerramento prematuro do fluxo de geração. A seguir são descritas as operações realizadas em cada etapa.

Figura 8 – Fluxo de etapas executadas pelo gerador MPM.

Fonte – Elaborada pelo autor.

Inicialização: essa etapa realiza a sinalização do evento *translate_init* onde o desenvolvedor pode definir os *middlewares* e estruturas de controle utilizados nas etapas seguintes do processo de tradução. O gerador acessa o parâmetro de compilação *export* e verifica a definição e existência de instâncias de extração. A não definição de pelo menos um método de extração implica no encerramento prematuro do processo;

Tradução: a etapa de tradução é a que agrega mais elementos do *framework*, nessa etapa são acessadas as descrições de código alvo, *middlewares* e apis que fornecem as informações necessárias para a manipulação e a tradução da representação intermediária. Para cada bloco a ser processado o gerador sinaliza um conjunto de eventos que permitem a execução de *middlewares* registrados pelo desenvolvedor. Um diagrama contendo o fluxo de eventos sinalizados pelo gerador está disponível na Figura 9;

O processo de tradução começa com a localização do grupo de instruções com *label main* dentre os grupos de instruções da representação intermediária, em seguida o gerador sinaliza o evento *before_translate* e inicia a tradução das instruções contidas no grupo. Para cada chamada de função que ainda não foi traduzida, o gerador cria e empilha o novo grupo de instruções, além de sinaliza o evento *before_translate_group*. Quando o grupo é completamente traduzido o gerador sinaliza o evento *after_translate_group*, desempilha o grupo atual e retoma a tradução do grupo anterior caso exista algum na pilha. Quando não houverem mais grupos a serem traduzidos o gerador sinaliza o evento *after_translate* e encerra a etapa de tradução.

Formatação: essa etapa executa a formação do modelo de representação de código alvo. O processo de formação é constituído de duas etapas:

– Seleção das descrições de *template*. Uma representação de modelo de código possui uma descrição de templante padrão vinculada durante sua inicialização.

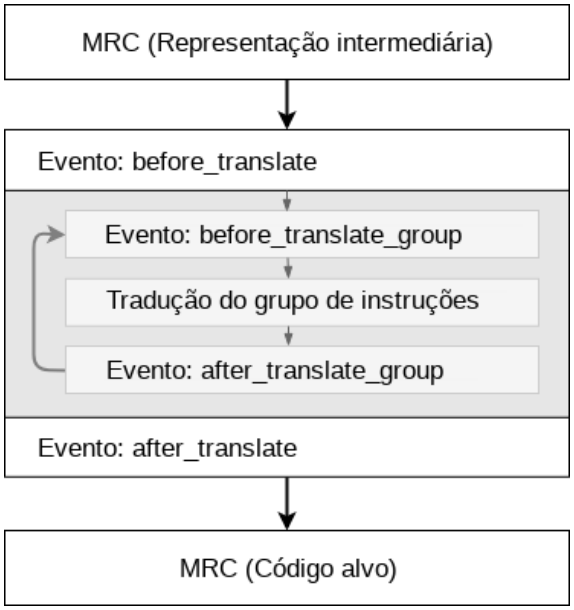

Figura 9 – Mapa de eventos do gerador MPM.

Fonte – Elaborada pelo autor.

O desenvolvedor pode sobrescrever a descrição utilizando o parâmetro de compilação *format* com a lista de descrições de *templates* que deseja utilizar;

– Interpolação da representação de código com a descrição do *template*. Para cada descrição de *template* o gerador acessa o atributo *format* da instrução e realiza uma chamada ao motor de renderização que formata a instrução. O processo de renderização está definido subseção 4.8.3. O resultado da formatação é atribuído à instrução na forma de um atributo e pode ser acessado na etapa de extração. O nome do atributo é composto pelo prefixo *formated_* concatenado com o identificador do *template* e o formato aplicado.

Extração: a etapa de extração é responsável por persistir o modelo de representação de código alvo. O gerador acessa o parâmetro de compilação *export*, verifica a lista de mecanismos de extração e aplica a o modelo de representação de código a cada um dos mecanismo. Um mecanismo de extração implementa a interface *Cognite.Generator.CodeExportInterface*, que contem um único método *Extract* que recebe como parâmetro o modelo de representação de código.

4.4.3 Transformação de Layout de Dados

4.5 Árvore Sintática Abstrata

Muitos compiladores representam o código fonte de um programa na forma de uma árvore sintática abstrata (AST). Uma árvore sintática abstrata é o representação

simplificada de uma *parse tree*, ela reflete o conjunto mínimo de estruturas lógicas essenciais para a validação e geração de código.

As aplicações de uma árvore sintática abstrata são:

- Analise semântica;
- Realizar otimizações;
- Gerar código.

Cognite representa uma árvore sintática abstrata por meio da classe *Cognite.Ast.Node*. Um nó é composto de: um mapa de atributos dinâmicos, um referencia para o nó pai e uma lista de nós filhos. A próxima subseção descreve de maneira detalhada a estrutura nó.

4.5.1 Representação de um Nó

Um nó é a estrutura base de uma AST. Todo símbolo relevante presente na *parser-tree* deve ser representado por um nó na AST. Não existe restrição quanto a quantidade de atributos e de nós filhos.

A representação mínima de um nó contem os seguinte atributos:

id: Valor numérico único que identifica o nó. O valor é gerado automaticamente quando um novo nó é instanciado;

value: Simbolo a ser representado;

class: Identificador de uma grupo de nós de um mesmo tipo;

subclass: Qualificador de um grupo de nós de uma mesma classe;

column: Número do byte da coluna que representa o simbolo;

line: Número da linha onde o simbolo ocorre;

Um nó possui os seguinte métodos:

AddChildren:

Formato: *Node AddChildren(Node n)*

Descrição: O método inseri um nó filho e retorna o Nó atual.

Childrens:

Formato: *List<Node> Childrens(Node n)*

Descrição: O método retorna a lista de nós filhos.

Dump:

Formato: *String Dump*()

Descrição: O método retorna uma representação formatada do nó e todos os nós descendentes.

GetParent

Formato: *Node GetParent*()

Descrição: O método retorna o nó pai. Caso o nó não possua um pai o valore retornado é nulo.

RemoveChildren

Formato: *Node RemoveChildren*(*Node n*)

Descrição: O método remove e retorna o nó especificado pelo parâmetro n.

SetParent

Formato: *Node SetParent*(*Node n*)

Descrição: O método atribui o nó *n* a referencia do pai e retorna o nó atual.

4.5.2 Construindo uma árvore

A construção de uma árvore é realizada utilizando uma instância de *NodeHandler* mapeado na classe *Cognit.ast.NodeHandler*. Um nó é alcançável direta ou indiretamente a partir de qualquer nó conectado a árvore. Isso é possível devido ao manipulador de nós. O manipulador de nós é capaz de:

- Adicionar e remover nós filhos;
- Atualizar os atributos de um nó;
- Navegar por um grupo de nós.

Um exemplo de construção de uma AST utilizando o manipulador de nós está disponível na Figura 10.

O manipulador navega pela AST utilizando a hierarquia dos nós e os métodos *Closest* e *Find*. Como mostra a Figura 11.

O método *Closest* possibilita encontrar um nó por acessos sucessivos a referencia do nó pai. O nó é identificado pelo critério especificado no atributo *class* do nó. Nesse tipo de consulta apenas um nó é retornado.

```
NodeHandler handler  = new NodeHandler();
// Define o primeiro nó com simbolo '+'
handler.SetNode("+")
        .Set("class","expr")
        .Set("subclass", "arith")
        .Set("type", "int32");
// Define um novo nó com simbolo 1
handler.SetNode(1)
        .Set("class","selector")
        .Set("subclass", "constant")
        .Set("type", "int32")
        .Return(); // Desempilha o nó atual

// Define um novo nó com simbolo 3
handler.SetNode(3)
        .Set("class","selector")
        .Set("subclass", "constant")
        .Set("type", "int32")
        .Return(); // Desempilha o nó atual
// Formata o nó inicial
handler.Dump()
```

```
└── + [ expr(arith): int32 (1)]
    ├── 1   [ selector(constant): int32 (2)]
    └── 3   [ selector(constant): int32 (3)]
```

Figura 10 – Exemplo contrução de uma AST. A esquerda o código que define a AST e a direita a representação.

Fonte – Elaborada pelo autor.

O método *Find* realiza a busca nos nós filhos de maneira recursiva. O atributo de identificação do nó e o valor do atributo *class*. Nesse tipo de consulta são retornados todos os nós filhos que correspondem ao critério da busca.

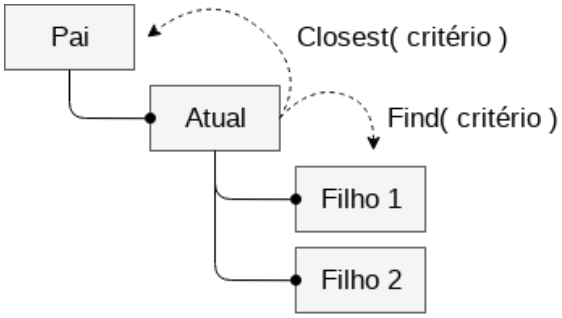

Figura 11 – Hierarquia de nós e métodos de navegação.

Fonte – Elaborada pelo autor.

4.5.3 Nó Agregado

Um nó agregador consiste de uma representação de uma estrutura utilizando uma hierarquia de nós com valor semântico definido. Eles tem mais significado para o projeto do que para o *framework* em si. O desenvolvedor é livre para representar estruturas contanto que tenham valor durante o processo de construção de código. Cognite possui um conjunto inicial de nós agregados, cada representação possui uma função de validação semântica

predefinida. Esse conjunto pode ser estendido, sobrescrito ou tomado como modelo para a construção de outros.

A Figura 12 representa uma agregação de nós que corresponde a uma definição de laço. O nó raiz possui a classe *loop.control* e sub-classes: *for, while* ou *do-while* que determinam o tipo de laço. O nós filhos com classes *initialization, increment* devem ser omitidos no caso de um laço *while*.

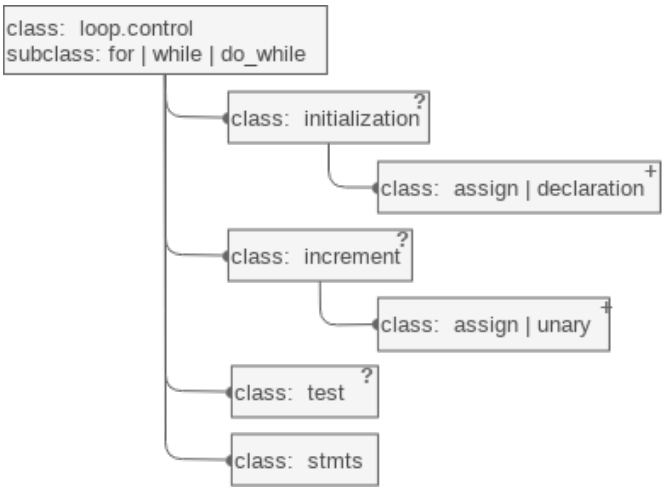

Figura 12 – Representação de um laço com nós agregados.

Fonte – Elaborada pelo autor.

O *framework* possui nós agregados para representar as seguintes estruturas: laço, testes, blocos de declarações, estruturas de controle de fluxo, declaração de funções, declaração de variáveis, declaração de constantes, operações de atribuição, expressões *booleanas*, expressões aritméticas, expressões de *bitwise*, expressões unárias e chamada de função.

4.5.4 Analisador Semântico

O analisador semântico é a estrutura responsável por validar a árvore sintática abstrata e verificar se o programa é semanticamente correto. A etapa de analise semântica envolve:

- Checagem de estruturas de controle de fluxo;
- Checagem de label;
- Checagem de tipo;
- Declaração e uso de identificadores.

O Analisador semântico expõe os seguintes métodos:

AddError:

Formato: *void AddError(Error e)*()

Descrição: O método adiciona um erro a lista de erros do analisador.

Errors:

Formato: *List<Error> Errors*()

Descrição: O método uma lista contendo os erros gerados durante o processo de validação.

RegisterValidator:

Formato: *void RegisterValidator*(*String class*, *SemanticValidatorFunction fn*)

Descrição: O método registra o validador para a classe especificada no parâmetro *class*.

SetMaxErrorCount:

Formato: *void SetMaxErrorCount*(*Integer max*)

Descrição: O método define a quantidade máxima de erros permitidos. Quando o valor é atingido o processo de analise é abortado. O valor 0 invalida a verificação. O valor padrão é 10.

Valid:

Formato: *Boolean Valid*(*Node n*)

Descrição: O método retorna verdadeiro quando a estrutura representada pelo nó é valida. Caso contrario retorna false. Quando uma estrutura não é valida os erros devem ser acessados por meio do método Errors().

Durante o processo de analise semântica o analisador percorre os nós recursivamente. Quando um nó é visitado o analisador lê o valor do atributo *class*. Caso exista uma função de validação registrado com o mesmo valor da classe do nó o analisador semântico executa a função passando o nó como contexto. O processo de validação pode retorna um erro, que é contabilizado pelo analisador. O análise finaliza quando a quantidade máxima de erros for atingida ou todos os nós forem visitados.

Uma função de validação semântica implementa a interface *Cognite.ast.SemanticValidatorFunction* e verifica se a estrutura representada por um nó agregado tem valor semântico válido. Cada classe de nó aceita apenas uma função de validação semântica. Caso mais de uma função seja registrada apenas a ultima será avaliada.

4.6 Modelo de representação de código

Essa seção apresenta o módulo do modelo de representação de código (MRC). O MRC define estruturas para elaboração de uma representação lógica de um código fonte independente da linguagem. Dispõe de quatro estruturas básicas que são vinculadas entre si e obedecem um hierarquia definida na figura Figura 13. A independência de linguagem permite a construção de diferentes representações como a RI e código alvo.

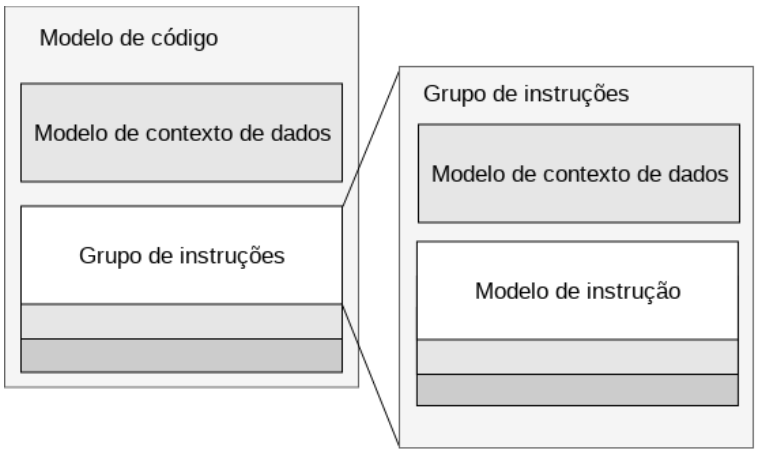

Figura 13 – Hierarquia do modelo de representação de código.

Fonte – Elaborada pelo autor.

4.6.1 Modelo de código

O modelo de código consiste da estrutura raiz de uma representação de código e é mapeado na classe *Cognite.Crm.Code*. Ele é responsável por gerenciar os grupos de instruções. Conta com mecanismos que registram e manipulam as *labels*. As *labels* são rótulos atribuídos a posições dentro de um código.

O modelo de código conta com os seguinte métodos:

Add

Formato: *Code Add(Instruction inst)*

Descrição: O método adiciona a instrução ao grupo de instruções atual. Se não existe um grupo de instruções ativo. Um novo grupo é criado com identificador *main* e em seguida a instrução é inserida.

Groups

Formato: *List<InstructionGroup> Groups()*

Descrição: O método retorna a lista de grupos de instruções existentes na representação de código.

CloseGroup

Formato: *void CloseGroup*()

Descrição: O método sinaliza o evento *close_group*, encerra e desempilha o grupo de instruções atual.

Context

Formato: *DataContext Context*()

Descrição: O método retorna o contexto global da representação de código.

Current

Formato: *InstructionGroup Current*()

Descrição: O método retorna o grupo de instruções ativo. Caso não exista um grupo ativo retorna valor nulo.

Dump

Formato: *String Dump*()

Descrição: O método retorna a representação de código formatada.

GetTemplate

Formato: *String GetTemplate*()

Descrição: O método retorna o identificador da descrição de *template* utilizada para a renderização das instruções.

OpenGroup

Formato: *void OpenGroup*()

Descrição: O método instância um novo grupo de instruções e sinaliza o evento *open_group*. O método pode ser acionado mesmo que exista outro grupo de instruções ativo sem comprometer o resultado. Cada novo grupo de instruções é empilhada e o contexto é salvo. Quando o grupo é encerrado o contexto é restaurado e o grupo do topo da pilha passa a ser o atual.

Remove

Formato: *Boolean Remove*(*Instruction inst*)

Descrição: O método remove a instrução do grupo de instruções ativo, retornando verdadeiro caso a operação seja realizada. Caso contrário retorna falso.

SetTemplate

Formato: *Boolean SetTemplate*(*String templateId*)

Descrição: O método define a descrição de *template* utilizada para a renderização das instruções.

Use

Formato: *Boolean Use*(*String groupId*)

Descrição: O método altera o grupo atual para o especificado no parâmetro *groupId*. O comportamento de empilhamento e desempilhamento é obedecido. O método retorna verdadeiro caso a operação seja realizado. Caso contrario retorna falso.

Update

Formato: *void Update*()

Descrição: O método executa a chamada do método *Update* todos os grupos de instruções. Nessa operação são atualizados os endereços dos *labels*.

4.6.2 Grupo de instruções

O grupo de instruções é uma estrutura mapeada na classe *Cognite.Crm.InstructionGroup*. Ele é responsável por agrupar as instruções referentes a uma rotina ou função dentro de um modelo de código.

O grupo de instruções conta com os seguinte métodos:

Add

Formato: *Code Add*(*Instruction inst*)

Descrição: O método adiciona a instrução ao grupo.

Context

Formato: *DataContext Context*()

Descrição: O método retorna o contexto de dados local.

Dump

Formato: *Boolean Dump*()

Descrição: O método retorna a representação do grupo de instruções formatada.

Remove

Formato: *Boolean Remove*(*Instruction inst*)

Descrição: O método remove a instrução do grupo de instruções, retornando verdadeiro caso a operação seja realizada. Caso contrário retorna falso.

Update

Formato: *void Update*()

Descrição: O método atualiza todas as referencias de *labels* e instruções lideres dos blocos básicos do código. Para isso ele avalia os atributos contidos no modelo de instrução destinados para esse proposito.

4.6.3 Modelo de instrução

O modelo de instrução está mapeado na classe *Cognite.Crm.Instruction*. Ele tem a finalidade de representar uma instrução no modelo de representação de código. O conjunto de atributos de uma instrução é dinâmico é depende do tipo de instrução que se deseja representar.

Para simplificar a descrição a palavra chave 'any' implica em qualquer um dos seguintes tipos: Integer, Boolean, String e Float.

O modelo de instrução apresenta os seguintes métodos:

Eq

Formato: *Boolean Eq*(*String attrib, any value*)

Descrição: O método retorna verdadeiro caso o atributo especificado pelo parâmetro *attrib* possuir valor igual ao valor definido no parâmetro *value*. Caso contrário retorna false. Quando o atributo não for definido a função retorna falso.

Has

Formato: *Boolean Has*(*String attrib*)

Descrição: O método retorna verdadeiro caso a instrução possua o atributo especificado pelo parâmetro *attrib*. Caso contrário a função retorna falso.

In

Formato: *Boolean In*(*String attrib*, List<any> values)

Descrição: O método retorna verdadeiro caso o valor do atributo especificado pelo parâmetro *attrib* esteja contido na lista de valores definida pelo parâmetro *values*. Caso contrário a função retorna falso.

Contains

Formato: *Boolean Contains*(*String attrib*, String search)

Descrição: O método retorna verdadeiro caso o valor do atributo especificado pelo parâmetro *attrib* contenha o valor definida pelo parâmetro *search*. Caso contrário a função retorna falso.

Copy

Formatos:

Instruction Copy()

Instruction Copy(*String attribList*, Instruction src)

Descrição: O método apresenta dois comportamentos. O Primeiro sem parâmetros retorna uma cópia da instrução que executou a chamada *copy*. O segundo o parâmetro *attribList* deve conter uma lista dos atributos separada por virgula que se deseja copiar da instrução definida no parâmetro *src*.

Get

Formatos:

String Get(*String attrib*)

Boolean GetBool(*String attrib*)

Integer GetInteger(*String attrib*)

Long GetLong(*String attrib*)

Descrição: O método retorna o valor do atributo especificado pelo parâmetro *attrib*. Caso o atributo não seja definido é retornado o valor nulo.

IsNumeric

Formato: *Boolean IsValue*(*String attrib*)

Descrição: O método retorna verdadeiro caso o valor do atributo especificado pelo parâmetro *attrib* represente um valor numérico.

Set

Formato: *Instruction Set*(*String attrib*, any value)

Descrição: O método adiciona ou atualiza um atributo. O atributo e o valor são definidos nos parâmetros *attrib* e *value* respectivamente.

Render

Formatos:

String Render()

String Render(*String format*)

Descrição: O método retorna a instrução formatada. Quando o método render é executado ele realiza uma chamada para a api do motor de renderização solicitando a sua formatação. O processo de renderização foi definido na subseção 4.8.3 . Nesse caso o formato pode ter origiem no parâmetro *format* ou no atributo da *format da instrução*. O contexto é a própria instrução.

4.6.4 Modelo de contexto de dados

Um modelo de contexto de dados (MCD) pode ser global ou local e consiste de uma estrutura responsável por mapear as variáveis e constantes declaradas dentro de um determinado contexto. Cada grupo de instruções do MRC possui um contexto próprio chamado de local. O contexto global corresponde a instância do MCD vinculada ao MRC. O MCD fornece mecanismos para alterar a ordem e os endereços das variáveis declaradas. Além de informar o deslocamento até a base da variável.

4.6.5 Api de Código

Descrição: A API de código é responsável por registrar novos destinos para a tradução da representação intermediaria.

Métodos da API:

A classe

Defined:

Formato: *Boolean Defined*(*String targetId*)

Descrição: O método retorna verdadeiro caso a descrição de código alvo especificada pelo parâmetro *targetId* exista. Caso contrário retorna false.

Get:

Formato: *TargetDescription Get*(*String targetId*)

Descrição: O método retorna uma instância do tipo *TargetDescription* especificada pelo parâmetro *targetId*. Caso a descrição não tenha sido registrada o método lança uma exceção.

RegisterTarget:

Formato: *void RegisterTarget*(*String id*, *TargetDescription target*)

Descrição: O método registra uma nova descrição de código alvo.

4.6.5.1 Código alvo

Descrever um alvo consiste de gerar um conjunto de especificações que serão consumidas durante o processo de tradução pelo gerador MPM. Essas especificações

exigem um conhecimento mais profundo da arquitetura a qual se pretende gerar código. Para exemplificar vamos supor a tradução da RI para a arquitetura alvo MIPS. O tradutor precisa conhecer o conjunto de instruções da ISA MIPS, bem como a quantidade de registradores disponíveis e a convenção quanto a utilização dos mesmos. Também é necessário uma descrição do *template* que será empregados na renderização do código.

A definição de um código alvo é formada por três componente:

- Descrição do alvo;
- Descrição do tradutor;
- Descrição de um *template*.

As subseções seguintes ressaltam o proposito e as particularidades de cada componente.

4.6.5.2 Descrição do alvo

O componente de descrição de código alvo estende a classe Cognite.generator.TargetDescription. Nele são representadas as especificações do hardware como conjunto de instruções suportadas, quantidade e a convenção de uso dos registradores e o mapeamento de operações. A classe possui dois atributos que representam os itens descritos.

- *Registers* - o atributo descreve a convenção e quantidade de registradores. Consistes de um *Map<Integer,String>* onde a chave é o índice do registrador e o valor corresponde ao apelido. A quantidade de registradores é inferida pelo tamanho do mapa. Um registrador pode assumir qualquer apelido. Embora exista uma convenção adotada pelo *framework* para a etapa de alocação de registradores. A convenção está disponível na Tabela 1;
- *Instructions* - o atributo é responsável por mapear todas as instruções do código alvo. Consistes de um *Map<Integer, Instruction>* onde a chave é o apelido da instrução e o valor corresponde a representação da instrução;

4.6.5.3 Funções de tradução

A descrição do tradutor alvo (DTA) define como cada instrução da RI (Instrução fonte) deve ser representada no código alvo.

Uma DTA implementa a *interface Cognite.generator.TargetTranslator* que contem um único método *registerTranslateFunctions* que retorna o mapa de funções de tradução. Uma função de tradução implementa a interface

Cognite.generator.TranslatorFunction. Elas são responsáveis por manipular o código alvo traduzindo a instrução fonte em instruções do código alvo preservando o mesmo

Nome	Descrição	Valores
0	Registrador sempre apresenta o valor 0	0
a	Registradores de argumento	a0, a1 ...
s	Registradores salvos	s0, s1 ...
t	Registradores temporários	t0, t1 ...
v	Registradores destinados a retorno de expressões ou funções	v0, v1 ...
k	Registradores reservados para *kernel*	k0, k1 ...
gp	Registrador de apontamento global	gp
fp	Registrador de apontamento do *frame*	fp
sp	Registrador de apontamento da pilha	sp
ra	Registrador de retorno de uma sub-rotina	ra

Tabela 1 – Convenção de apelidos para registradores.

Fonte – Elaborada pelo autor.

valor semântico. Dependendo do código alvo uma instrução fonte pode gerar mais de uma instrução alvo. O exemplo de código Código 4.4 exibe a definição de um tradutor de código alvo.

O desenvolvedor deve implementar uma função de tradução para cada tipo de instrução definida na RI mesmo que não seja realizada nenhuma operação para um determinado tipo de instrução. O não cumprimento acarreta em uma exceção e o processo de tradução é abortado. Essa restrição garante uma tradução completa evitando que alguma instrução fonte não seja traduzida. O gerador realiza o mapeamento da instrução fonte da RI com a função de tradução por meio do atributo *type*.

Além do conjunto de instruções da RI o desenvolvedor deve implementar dois tradutores adicionais, eles são destinadas a descrição das rotinas executadas antes e após a chamada de uma função.

- Prólogo - tem a função de alocar espaço na pilha, persistir o conteúdo dos registradores cujo o valor não deve ser perdido com a troca de contexto. A chave de mapeamento para função de tradução é *prolog*;
- Epílogo - tem a função de desalocar o espaço na pilha, restaurar o contexto dos registradores e retornar o controle para a função anterior. A chave de mapeamento para função de tradução é *epilog*.

```
1 public class MipsTargetTranslator  implements TargetTranslator {
2
3   public HashMap<String, TranslatorFunction> registerTranslateFunctions(){
4     return new HashMap<>(){{
5       put("goto", (inst, target, ir) -> {
6
```

```
7           String label = inst.Get("label");
8
9           target.Add("j")
10              .Set("label", label)
11              .Set("comment", F("jump to %s", label));
12          });
13          /* ... */
14          put("prolog", (inst, target, ir) -> {});
15          put("epilog", (inst, target, ir) -> {});
16      }};
17  }
18 }
```

Código 4.4 – Exemplo da definição de um tradutor de código alvo.

4.7 Representação Intermediaria

A representação intermediária consiste de uma instância do modelo de representação de código. O *framework* possui um conjunto inicial de instruções que foram inseridas levando em consideração as necessidades para a representação das aplicações no laboratório. O conjunto pode ser estendido e novas funcionalidades podem ser inseridas na RI. Essa seção apresenta os tipos primitivos, operadores e o conjunto de instruções suportadas pela RI.

4.7.1 Tipos

A representação intermediária possui o seguintes tipos primitivos. Eles estão registrados na api de tipos do *framework*.

Nome	Descrição
byte	Um tipo inteiro de 8 bits
bool	Um tipo inteiro de 8 bits
int8	Um tipo inteiro de 8 bits
int16	Um tipo inteiro de 16 bits
int32	Um tipo inteiro de 32 bits
int64	Um tipo inteiro de 64 bits

Tabela 2 – Tabela de tipos primitivos suportados pela RI.

Fonte – Elaborada pelo autor.

O desenvolvedor pode expandir o conjunto de tipos primitivos inserir um novo tipo utilizando a api definida na subseção 4.2.3.

4.7.2 Operadores

Os operadores suportados pela representação intermediária estão listados na Tabela 3.

Aritméticos	
+	Soma os dois argumentos
-	Subtrai o segundo argumento do primeiro
*	Multiplica os dois argumentos
/	Divide o primeiro argumento pelo segundo
%	Calcula o resto da divisão do primeiro argumento pelo segundo
Bitwise	
^	Calcula o xor bit a bit dos dois argumentos
|	Calcula o or bit a bit dos dois argumentos
&	Calcula o and bit a bit dos dois argumentos
«	Preenche com zeros a direita do argumento 1, quantos bits forem informados pelo argumento 2
»	Preenche com zeros a esquerda do argumento 1, quantos bits forem informados pelo argumento 2
Comparação	
==	Resulta em verdade quando os dois argumentos são iguais
!=	Resulta em verdade quando os dois argumentos são diferentes
>	Resulta em verdade quando o primeiro argumento for maior que o segundo
>=	Resulta em verdade quando o primeiro argumento for maior ou igual ao segundo
<	Resulta em verdade quando o primeiro argumento for menor que o segundo
<=	Resulta em verdade quando o primeiro argumento for menor ou igual ao segundo

Tabela 3 – Tabela de operadores suportados pela representação intermediária.

Fonte – Elaborada pelo autor.

4.7.3 Representação de uma instrução

Instruções são instâncias de *Cognite.code.Instruction*, e possuem atributos definidos pelo modelo de representação de código (RPC) e pelo desenvolvedor. A Tabela 4 expõe os atributos gerados pelo RPC. Enquanto que a Tabela 5.

Os atributos de uma instrução correspondem a:

4.7.4 Descrição da RI

O conjunto predefinido de instruções da representação intermediaria está descritos em ordem alfabética. Para cada instrução, foram listados os aspectos: formato, proposito, descrição, operação, exemplo e em alguns casos as restrições.

arg1.value	Determina se o argumento tem valor numérico.	
arg2.value	Determina se o argumento tem valor numérico.	
inloop	Determina se a instrução ocorre dentro de um laço.	
local.position	Posição da instrução em relação ao grupo.	
global.position	Posição da instrução em relação ao código.	

Tabela 4 – Tabela de atributos gerados pelo MRC.

Fonte – Elaborada pelo autor.

Nome	Descrição	
label	Referencia para alguma posição do código.	
format	Referencia o formato de renderização da instrução.	
type	Define o tipo da instrução.	
op	Define um operador em instruções que apresentem expressões.	
dst	Destino de uma atribuição.	
arg1	Argumento 1.	
arg2	Argumento 2.	
reg.dst	Registrador de destino.	
reg.dst.indice	Registrador do indice quando a instrução possuir destino indexado.	
reg.arg1	Registrador do argumento 1.	
reg.arg1.indice	Registrador do indice do argumento 1 quando ocorrer.	
reg.arg2	Registrador do argumento 2.	
arg1.type	Tipo do argumento 1.	
arg2.type	Tipo do argumento 2.	
comment	Comentário sobre a operação da instrução.	

Tabela 5 – Tabela de atributos gerados pelo desenvolvedor.

Fonte – Elaborada pelo autor.

ALLOC

Formato: *alloc* (*type*) [, *length*] *id*

Proposito: A instrução *alloc* deve ser empregada para alocar espaço de memoria na pilha de uma chamada de função ou no context global.

Descrição:

- *type* - indica o tipo de variável a ser alocada de onde se extrai o tamanho em bytes;
- *id* - define um identificador para a variável alocada;
- *length* - informa a quantidade sequencial de elementos a ser alocada. O valor maior que 1 indica a alocação de um *array*. O valor padrão é 1.

Os parâmetros *type* e *id* são obrigatórios.

Operação: A ocorrência dessa instrução na RI implica em um registro da variável na estrutura de controle de contexto, que eventualmente será utilizada para obter o deslocamento em relação a base.

Exemplo: O código representa a alocação de duas variáveis. A primeira sendo um inteiro de 4 *bytes* e a segunda um array de 4 bytes apelidadas de _V1 e _V2 respectivamente.

```
1    alloc (int32), 1 _V1
2    alloc (byte), 4 _V2
```

BREAK

Formato: *break <label>*

Proposito: Controlar fluxo de execução.

Descrição: A instrução *break* desvia o fluxo de um laço ou *switch*. O parâmetro *label* é opcional. Quando apresenta um *label* o salto é realizado para o endereço referenciado pelo *label*. Caso contrário o salto é executado para a primeira instrução após o laço ou o *switch*.

Exemplo: O exemplo demonstra a aplicação da instrução *break* em um laço.

```
1 LABEL_1:
2         /* ... */
3         break <OUT>
4         /* ... */
5         LOOP: if _V1 != 2 < LABEL_1 >
6 OUT: /* ... */
```

ASSIGN

Formato: *dst := arg1 op arg2*

Proposito: Realizar operações de atribuições dos resultados das expressões dos tipos:

- Aritméticas (+, -, %, / e *);
- *Booleanas* (==, !=, <, <=, > e >=);
- BitWise (, &, |, «, »);
- Unárias (!, -).

Descrição:

- *dst* corresponde ao endereço de atribuição do resultado da expressão.
- *arg1* primeiro argumento da expressão.
- *op* sinal da operação a ser realizada.
- *arg2* segundo argumento da expressão.

Os parâmetros *dst*, *op* e *arg1* são obrigatórios.

Operação:

A instrução *assign* atribui o valor da expressão no endereço de destino.

CALL

Formato: *call* < *funcId* > [, *length*]

Proposito: A instrução *call* define uma chamada de função.

Descrição:

- *funcId* corresponde ao identificador da função chamada.
- *length* informa a quantidade de parâmetros da chamada. O valor padrão é 0.

Os parâmetros *funcId* e *length* são obrigatórios.

Operação: A ocorrência dessa instrução na RI implica em um salto para o endereço do bloco com identificador igual ao definido na chamada. Os parâmetros de uma chamada devem ser carregados utilizando a instrução *push_param*.

Exemplo: O código representa uma chamada de função com dois parâmetros. O primeiro corresponde a uma variável e o segundo uma constante.

```
1    push_param _V1
2    push_param 1
3    call < FUNC >, 2
```

CONTINUE

Formato: *continue* <*label*>

Proposito: Controlar fluxo de execução de um laço.

Descrição: A instrução *continue* desvia o fluxo de um laço. O parâmetro *label* é opcional. Quando uma instrução *continue* ocorre com *label* o salto é realizado para o endereço apontado pelo *label*. Caso contrário salta para o teste do laço.

Exemplo: O exemplo demonstra a aplicação da instrução *continue*.

```
1 LABEL_1:
2         /* ... */
3         continue <LOOP>
4         /* ... */
5 LOOP:  if _V1 != 2 < LABEL_1 >
```

COPY

Formato: *dst* := *src*

Proposito: A instrução *copy* define uma operação de cópia.

Descrição:

- *dst* define um identificador que recebe o valor.
- *src* define a fonte a ser copiada.

Uma operação de cópia pode ser indexada. Nesse caso apenas o destino ou a fonte deve ser indexado. A restrição limita a quantidade de elementos endereçáveis mantendo a representação de código de três endereços.

Operação: A ocorrência dessa instrução na RI implica em uma copia da fonte para o destino. A fonte deve ser uma constante ou identificador

Exemplo: O exemplo de código representa duas operações de cópia. Na primeira o valor de _T1 é copiado para _V1 e na segunda realiza uma cópia indexada.

```
1    _V1      := _T1
2    _V2[_V1] := 1
```

FALLTHROUGH

Formato: *fallthrough*

Proposito: Controlar fluxo de execução de um *switch*.

Descrição: A instrução *fallthrough* executa um salto para a primeira instrução do caso seguinte.

Exemplo: O exemplo demonstra a aplicação da instrução *fallthrough*. Cada *if* indica o teste de um caso. A instrução *fallthrough* salta para a primeira instrução do segundo caso.

```
1            if _V1 != 1 < LABEL_1 >   // case
2            /* ... */
3            fallthrough
4 LABEL_1:  if _V1 != 2 < LABEL_2 >   // case
5            /* ... */
```

GOTO

Formato: *goto* < *label* >

Proposito: A instrução *goto* define um salto incondicional.

Descrição:

– *label* corresponde ao label alvo do salto.

O parâmetro *label* é obrigatório.

Operação: A ocorrência dessa instrução na RI implica em um salto incondicional para o endereço definido.

Exemplo: O código representa um salto incondicional para o *label* LABEL_1.

```
1    goto < LABEL_1 >
2    _T1 := _V1 + _V2
3    < LABEL_1 > : _T1 := 1
```

LABEL

Formato: < *id* > :

Proposito: A instrução *label* define um rótulo para o endereço de uma instrução ou bloco.

Descrição:

– *id* corresponde ao rótulo do endereço.

O parâmetro *id* é obrigatório.

Operação: A ocorrência dessa instrução na RI implica em uma sinalização do endereço para a estrutura de controle de endereçamento de saltos.

Exemplo: O trecho de código representa a sinalização com identificador LABEL_1 da instrução _T1 = 1.

```
1    < LABEL_1 > : _T1 := 1
```

Restrições: Um rótulo não deve iniciar com números e o uso de caracteres especiais está restrito ao conjunto [+-_]. Todo *label* é único em um código, não podendo referencias mais de uma posição.

Existem dois tipos de rótulos:

– *user*: criados pelo usuário e tem funcionamento limitado ao bloco em que foi definido.

– system: definidos pelo compilador como: inicio e fim de um bloco, endereços de saltos em instruções de laço ou controle de fluxo.

Apenas o rótulo inicial de um bloco pode ser referenciada de qualquer outro bloco por meio da instrução *call*.

LOAD

Formato: *load* (*type*) base [[*offset*]] , *dst*

Proposito: A instrução *load* define uma leitura de um dado da memória.

Descrição:

- *type* indica o tipo de onde se extrai o tamanho em bytes.
- *base* define o identificador do endereço base da escrita.
- *dst* define o identificador destino do valor carregado.
- *offset* indica a quantidade de bytes a ser ignorada a partir da base. O valor padrão é 0.

Os parâmetros *type*, *base* e *dst* são obrigatórios.

Operação:

dst := MEMORIA[base + offset]

A ocorrência dessa instrução na RI implica em uma leitura na memória. Em uma operação de leitura o *offset* é somado ao endereço base, dessa forma é possível endereçar um único *byte* ou estruturas complexas como *structs* e *arrays*. A quantidade de bytes a serem lidos é obtida por meio do parâmetro *type*.

Exemplo: No primeiro exemplo o valor lido do quarto byte após _V4 é atribuído ao identificador destino _T1.

```
1    load (byte) _V4[3], _T1
```

No segundo exemplo é realizada a leitura dos 4 primeiros bytes de _V5 e o valor é atribuído a _T1.

```
1    load (int32) _V5, _T1
```

Restrições: A leitura está condicionada aos tipos primitivos da RI. Mencionados na Tabela 2.

POP PARAM

Formato: *pop_param* (type) *dst*

Proposito: A instrução *pop_param* define o desempilhamento de um parâmetro em uma chamada ou retorno.

Descrição:

- *type* indica o tipo de onde se extrai o tamanho em bytes.

– *dst* informa o identificador onde o valor será atribuído

Os parâmetros *type* e *dst* são obrigatórios.

Operação:

A ocorrência dessa instrução na RI implica no desempilhamento de um parâmetro em uma chamada ou retorno. Os parâmetros devem ter sido previamente carregados com a instrução *push_param*. Dependendo da arquitetura alvo os parâmetros podem estar em registradores ou na própria pilha.

Exemplo:

O código representa leitura de um parâmetro do tipo *int32*. O valor do parâmetro é atribuído ao identificador _V1.

```
1    pop_param (int32) _V1
```

Restrições: A leitura está condicionada aos tipos primitivos da RI. Mencionados na Tabela 2.

PUSH PARAM

Formato: *push_param* (type) *src*

Proposito: A instrução *push_param* define o empilhamento de um parâmetro em uma chamada ou retorno de uma chamada.

Descrição:

– *type* indica o tipo de onde se extrai o tamanho em bytes.

– *src* define a fonte a ser empilhado, que deve ser uma constante ou um identificador.

Os parâmetros *type* e *src* são obrigatórios.

Operação:

A ocorrência dessa instrução na RI implica no empilhamento de um parâmetro. Deve ser inserida antes da instrução *call* ou *return*. O parâmetro deve ser desempilhado usando a instrução *pop_param*. Dependendo da arquitetura alvo os parâmetros devem ser escritos em registradores ou na própria pilha.

Exemplo:

O código representa empilha o valor contido em _V1 com tipo *int32* .

```
1    push_param (int32) _V1
```

Restrições:

A empilhamento está condicionada aos tipos primitivos da RI. Mencionados na Tabela 2.

RETURN

Formato: *return* [q]

Proposito: A instrução *return* define o encerramento da execução em um bloco.

Descrição:

- *q* informa a quantidade de parâmetros de retorno. O valor padrão é 0.

Operação: A ocorrência dessa instrução na RI implica em um salto para a primeira instrução após a instrução que realizou a chamada. Os parâmetros de retorno devem ser carregados utilizando a instrução *push_param*.

Exemplo: O código representa uma definição de retorno com dois parâmetros. O primeiro corresponde a uma variável e o segundo uma constante.

```
1    push_param _V1
2    push_param 1
3    return 2
```

STORE

Formato: *store* (*type*) base [[*offset*]] , *src*

Proposito: A instrução *store* define a escrita de um dado na memória.

Descrição:

- *type* indica o tipo de onde se extrai o tamanho em bytes.
- *base* define o identificador do endereço base da escrita.
- *src* define a fonte do dado a ser escrito, que deve ser uma constante ou um identificador.
- *offset* indica a quantidade de bytes a ser ignorada. O valor padrão é 0.

Os parâmetros *type*, *base* e *value* são obrigatórios.

Operação:

MEMORIA[base + offset] = src

A ocorrência dessa instrução na RI implica em uma escrita na memória. Em uma operação de escrita o *offset* é somado ao endereço base, dessa forma é possível endereçar um único *byte* ou estruturas complexas como *structs* e *arrays*. A quantidade de bytes a serem escritos é obtida por meio do parâmetro *type*.

Exemplo:

No primeiro exemplo é escrito o valor 3 no quarto byte após o endereço _V1.

```
1    store (byte) _V1[3], 3
```

No segundo exemplo é realizada a escrita do valor armazenado em _T4 nos 4 primeiros bytes de _V2.

```
1    store (int32) _V2, _T4
```

Restrições: A escrita está condicionada aos tipos primitivos da RI. Mencionados na Tabela 2.

4.7.5 Convenções

Para manter a padronização e legibilidade no código gerado deve seguir as seguinte convenções:

- As instruções *alloc* devem estar no inicio do bloco.
- Os *labels* de início e fim de um bloco devem conter os sufixos '*-begin*' e '*-end*' respectivamente, não podendo estar presentes em *labels* definidos pelo usuário.
- Os identificadores de variáveis devem usar o prefixo _V (Variável) para as variáveis de pilha e _G (Global) para as globais.
- Os identificadores que armazenam resultado de operações devem usar o prefixo _T (Temporário).

4.7.6 *Middlewares* embarcados

Cognite conta com um grupo de *middlewares* que manipulam a RI. A execução de um *middlewares* pode ser condicionada a prévia execução de outro. Nos casos onde não existe dependência o *middleware* pode ser ou não habilitados sem implicar em erro no resultado final da execução dos manipuladores. Caso contrário o desenvolvedor deve respeitar a ordem de execução.

Os *middlewares* que manipulam a RI são descritos abaixo, em cada descrição foram considerados os seguintes aspectos: propósito, operação realizada e dependências.

ir.basic_blocks

Propósito: Encontrar e sinalizar os blocos básicos de um grupo de instruções de um modelo de representação de código.

Operação: O *middleware* é aplicado sobre um grupo de instruções. Ele percorre a lista de instruções do bloco em busca de instruções lideres. O *middleware* armazena em uma estrutura a referencia para a primeira instrução e a quantidade de instruções de um bloco básico, além de atualizar as referencias para os blocos

básicos que gerem dependência na ordem de execução. Uma instrução lider foi definida na subseção 2.1.5.

Dependências: Não possui dependências.

ir.live_var

Propósito: Definir o intervalo de vida de uma variável dentro de um grupo de instruções.

Operação: O *middleware* é aplicado sobre um grupo de instruções. Para cada bloco básico do grupo são armazenados em uma estrutura o intervalo e as ocorrências de cada variável referenciada. Um intervalo determina a primeira e a ultima ocorrência de uma variável. Uma ocorrência corresponde a cada instrução onde a variável é mencionada.

Uma instrução pode conter até três endereços e a indexação das variáveis ocorre por meio dos atributos: dst, arg1, arg2 e suas variações dst_indice, arg1_indice e arg2_indice.

Dependências: *ir.basic_blocks*.

ir.labels

Propósito:

- Remover *labels* não referenciadas;
- Unificar múltiplas *labels* que referenciam o mesmo endereço;
- Atualizar instruções de salto.

Operação: Durante o processo de geração da RI, instruções podem ser inseridas ou removidas. *Labels* podem deixar de ser referenciadas ou passam a referenciar uma mesma instrução. O *middleware* trata essas ocorrências tornando a gestão de *labels* mais eficiente.

Dependências: Não possui dependências.

ir.inline_function

Propósito: Substituir uma chamada de função pelo conjunto de instruções executadas por ela.

Operação: O *middleware* analisa as ocorrências de instruções do tipo *call*. A substituição de uma chamada pelo corpo da função evita a sobrecarga gerada pela alocação de contexto de uma chamada. A sobrecarga se deve a carga e descarga de parâmetros e valores de retorno, bem como a alocação da pilha para a sua execução. Embora exista um ganho de performance, ela pode implicar em um aumento na quantidade de instruções do código (SERRANO, 1997).

A implementação de *inline* pode ser uma atividade complexa. O *middleware* implementa uma versão simplificada que aborda um caso especifico.

Uma função está apta a substituição quando atende as seguintes restrições:

- Não apresenta chamadas recursivas;
- A quantidade de instruções destinadas a carga e descarga de parâmetros e valores de retorno, bem como as de alocação e desalocação da pilha for superior ao tamanho do corpo da função.

Dependências: Não possui dependências.

ir.optimization.constante_propagation

Propósito: Identificar e avaliar expressões constantes em tempo de compilação, evitando que sejam computadas em tempo de execução.

Operação: O *middleware* avalia as seguintes ocorrências:

- Expressões lógicas e aritméticas. Caso a expressão seja solucionável (não apresenta variáveis), o valor é computador e propagado para as instruções que utilizem o destino da expressão. Um caso particular ocorre em instruções do tipo *branch*, condicionado ao valor da solução. Quando o valor for verdadeiro a instrução *branch* é convertida para um salto incondicional do tipo *goto*. Caso contrário a instrução é removida.
- Variáveis que recebem apenas uma atribuição durante sua vida são consideradas constantes. Nesses casos o valor ou endereço fonte é propagado e a instrução de cópia é removida.

Dependências: Não possui dependências.

ir.optimization.dead_instruction_remove

Propósito: Remover instruções e blocos básicos não atingíveis.

Operação: O processo ocorre em duas etapas. Na primeira o *middleware* percorre, em busca de instruções de retorno no grafo de dependência de blocos básicos construído no *middleware ir.basic_blocks*. Quando uma instrução de retorno é encontrada todos os blocos acessados são marcados como utilizados. Um grupo de instruções pode conter mais de um grafo de retorno. Quando não restarem mais blocos básicos com instruções de retorno, os blocos básicos não marcados são removidos. Na segunda cada bloco básico é analisado de forma independente. As instruções são percorridas em ordem reversa a declaração. As instruções cujo o destino não foi utilizado como argumento de outra instrução são removidas.

Dependências: *ir.basic_blocks*.

ir.optimization.load_store

Propósito: Remover instruções de *load* e *store* desnecessárias.

Operação: O *middleware* avalia as ocorrências desnecessárias de *load* e *store* e as remove. O foco é manter as variáveis do bloco durante o maior tempo em registradores. Operações de *load* e *store* indexados sempre são realiza.

Dependências: ir.basic_blocks.

4.8 Motor de renderização

O módulo de *templateengine* é encarregado de formatar as instruções contidas em uma instância de um modelo de representação de código. O motor de renderização consiste de um conjunto extensível de recursos com a finalidade de:

- Padronização do processo de formatação;
- Formatação otimizada;
- Flexibilidade e extensão;
- Reaproveitamento de código.

O recurso chave do motor de renderização é a capacidade de avaliar expressões contidas em uma *string*. Ele também fornece uma api que permite ao desenvolvedor registrar descrições de templastes para a utilização no processo de formatação de uma representação de código.

A subseção seguinte apresenta o conceito de expressões adotado pelo motor de renderização.

4.8.1 Expressões

Expressões são elementos inseridos no corpo de uma *string* e tem a função de tornar dinâmico seu conteúdo. Durante o processo de renderização cada expressão é substituída pelo valor equivalente a sua avaliação. Expressões tem o formato:

```
E       -> '{' E '[' A (| A)* ']' E '}'

A       -> Literal | Chamada | Id

Chamada -> Id '(' A* ')'

Literal   -> \d+ | "\w*"

Id       -> [a-zA-Z][\w\.]+
```

Uma expressão apresenta dois controles condicionais de renderização.

1. refere-se ao corpo da expressão. O corpo de uma expressão compreende o conteúdo presente entre os caracteres [ ]. O conteúdo é renderizado da esquerda para a direita até que uma valor não nulo seja encontrado;

2. Caso o resultado do corpo não seja nulo. As sub expressões de prefixo e sufixo são avaliadas. O valor resultante das sub expressões são concatenados ao corpo da expressão na ordem que ocorrem.

O exemplo de Código 4.5 mostra a ocorrência de uma expressão inserida em uma *string*.

```
1 String formato = "dst = { arg1 [ op ] arg2 } ";
```

Código 4.5 – Exemplo de expressão em uma *string*.

4.8.2 Interface de uma Função

Uma função deve implementar a interface *Cognite.templateengine.FunctionInterface*. Descrita no exemplo Código 4.6

```
1 public interface FunctionInterface {
2    public String Call(Instruction ctx, ArrayList<String> args) throws Exception;
3 }
```

Código 4.6 – Definição da FunctionInterface

A definição conta com dois parâmetros:

- **ctx**: Objeto fonte dos dados.
- **args**: Lista de argumentos expressas no formato, quando a chamada foi realizada.

4.8.3 Renderização

O processo de renderização resulta em uma *string* formatada.

Renderizar um formato implica em:

1. Selecionar um formato;

2. O formato deve ter sido previamente registrado por meio de uma descrição de *template*;

3. Aplicar o contexto ao formato.

A renderização de um formato ocorre pela chamada demonstrada no exemplo Código 4.7.

```
1 Template.Format(formatId).Render(ctx);
```

Código 4.7 – Renderizando um descrição de *template*

4.8.3.1 Exceções

O *Template Engine* lançará exceções quando:

- Ocorrer uma tentativa de renderizar um formato não definido;
- Uma função chamada não for previamente registrada;
- Uma chamada de função em uma expressão lançar uma exceção.

Nesses casos o processo de renderização é abortado.

4.8.4 Funções internas

A api de *templates* conta com um conjunto funções predefinidas descritas em ordem alfabética. Esse conjunto pode ser ampliado, uma vez que a api permite a inserção de novas funções.

BIN

Definição: *String BIN*(*input* [, *length*])

Descrição: Retorna uma *string* contendo o valor *input* em sua representação binaria. O parâmetro *length* determina a quantidade de bits.

Exemplo:

BIN(3, 5) = 00011

BIN(8, 32) = 00000000000000000000000000001000

DEC

Definição: *String* DEC(*input*, *length* [, *pad*])

Descrição: Retorna uma *string* formatada com preenchimento. O preenchimento se comporta como o descrito na função PAD, com direção fixada em 'L'.

Exemplo:

DEC(128, 4,' ') = ' 80'

HEX(128, 4,0) = 0080

HEX

Definição: *String* HEX(*input*, *length* [, *pad*)

Descrição: Retorna uma *string* contendo o valor *input* em sua representação hexadecimal. Apresenta mesmo comportamento descrito na função DEC.

Exemplo:

HEX(128, 4,' ') = ' 80'

HEX(128, 4,0) = 0080

PAD

Definição: *String* PAD(input, *length* [, *pad*[, *type*]])

Descrição: Retorna a *string input* com o tamanho especificado no parâmetro *length*. Se *input* possuir tamanho inferior ao definido o mesmo é preenchido com o valor do parâmetro *pad*. Se o parâmetro *pad* não for indicado, *input* é preenchido com espaços. O parâmetro *type* determina a direção do preenchimento 'R' (preencher a direita), 'L' (preencher a esquerda).

Exemplo:

PAD(2, 5, 'a', 'R') = '2aaaa'

PAD(2, 5, 0) = 00002

T

Definição: *String* T(*length*)

Descrição: Retorna uma *string* contendo tabulações '\t'. O parâmetro *length* determina a quantidade de tabulações. Valor padrão é 1.

Exemplo:

T(2) = '\t\t'

4.8.5 Api de *Templates*

Uma instância de código pode ser representada por mais de um templante, isso possibilita a geração de vários formatos de saída. Uma descrição de *template* implementa a interface *Congite.templateengine.TemplateDescription* e define um conjunto de formatos e funções que são utilizados pelo motor de rederização para gerar a saída desejada. Um exemplo resumido de como construir uma descrição pode ser visualizado em Código 4.8.

4.8.5.1 Componentes de um *template*

Um *template* é a combinação de um conjunto de formatos e funções.

- Formato: é uma *string* composta por uma ou mais expressões. Expressões são definidas na subseção 4.8.1;
- Função: é uma rotina capaz de transformar o valor de uma entrada. Maiores detalhes sobre podem ser obtidos na subseção 4.8.2.

4.8.5.2 Descrição de um *template*

A interface possui dois métodos:

- *Formats:* Retorna um *map<String, String>* onde o chave é o identificador do formato e o valor é o formato propriamente dito;
- *Functions:* Retorna um *map<String, FunctionInterface>* onde a chave é o identificador da função, o qual deve ser usado em uma expressão para realizar a chamada. O valor corresponde a instância da função.

```
1 public class MipsTemplateDescription implements congite.template.TemplateDescription {
2
3    public HashMap<String, String> Formats() throws Exception {
4       return new HashMap<String, String>() {{
5          put("J", "{[inst]} {[rt],} {[OFFSET(offset,rs)]}");
6          /* ... */
7       }}
8    }
9
10   public HashMap<String, FunctionInterface> Functions() throws Exception {
11      return new HashMap<String, FunctionInterface>() {{
12         put("OFFSET", (ctx, args) -> args.get(0) + "(" + args.get(1) + ")");
13         /* ... */
14      }}
15   }
16 }
```

Código 4.8 – Definição de um descrição de *template*.

Uma descrição deve ser registrada na API de *templates* utilizando o método:

```
1 Template.Register(new MipsTemplateDescription());
```

Código 4.9 – Registrando um descrição de *template*.

4.9 O *Compilador Cognite*:

Essa seção trata da construção do compilador Cognite. A versão atual do compilador apresenta suporte a um subconjunto da linguagem GO e gera código para arquitetura

alvo MIPS. As subseções seguintes apresentam: o compilador, o ciclo de compilação e a definição de uma função de parse.

4.9.1 O Compilador

A construção de um compilador consiste de estender a classe *Cognite.Compiler.Compiler*. A classe estendida possui um método abstrato *Init* que deve ser implementado pelo desenvolvedor para definir os recursos do compilador. O método a ser implementado corresponde a:

Init

Definição: *void Init*()

Operação: o método deve conter todas as definições do compilador. São elas:

- os tipos da linguagem de alto nível;
- os *middlewares* de análise e otimização;
- as funções de validação semântica;
- as funções de conversão de nós da AST adicionais;
- as descrições de *templates*;
- as descrições de código alvo.

As definições devem ser registradas utilizando as apis destinadas para cada recurso.

O desenvolvedor deve ainda registrar um função de parse invocando o método *Parse* durante o processamento da função *Init*. Como mostra o exemplo de Código 4.10. A descrição da função *Parse* segue:

Parse

Definição: *void Parse*(*String extension*, List<String> (String filename , ast) *fn*)

Operação: o métodos registra uma *Cognite.Compiler.ParseFileFunction* para os arquivos com extensão equivalente a especificada no parâmetro *extension*. A função *fn* recebe como parâmetros o caminho do arquivo e uma instância da árvore sintática abstrata (AST). A função deve realizar a analise léxica e sintática de um arquivo e inserir na AST o conjunto de nós agregados que representem o código fonte. Todos os arquivos importados pelo código fonte atual devem ser retornados em uma lista no argumento de retorno para posterior avaliação.

4.9.2 Ciclo de compilação

O processo de compilação inicia com a chamada do método *compile* de uma instância do compilador. O método *compile* recebe como parâmetro o caminho do arquivo principal a ser compilado. O fluxo das etapas de compilação com as representações de entrada e saída está disponível na Figura 14. Na primeira etapa são registradas as definições do desenvolvedor executando o método *Init*. A etapa de parse constrói a árvore sintática abstrata (AST) seção 4.5 executando a análise do arquivo de entrada. Nessa etapa também são realizadas as importações das dependências do programa fonte. Após a conclusão do parse a AST resultante é avaliada pela analisador semântico subseção 4.5.4. Não verificando a ocorrência de erros a AST é percorrida pelo gerador APM subseção 4.4.1 para a construção da RI, realizando a conversão dos nós em instruções da representação intermediária. Por fim a representação intermediária é traduzida para o código alvo pelo gerador MPM subseção 4.4.2. O processo de compilação encerra quando todas as etapas são executadas ou quando algum erro provoca o encerramento prematuro do processo.

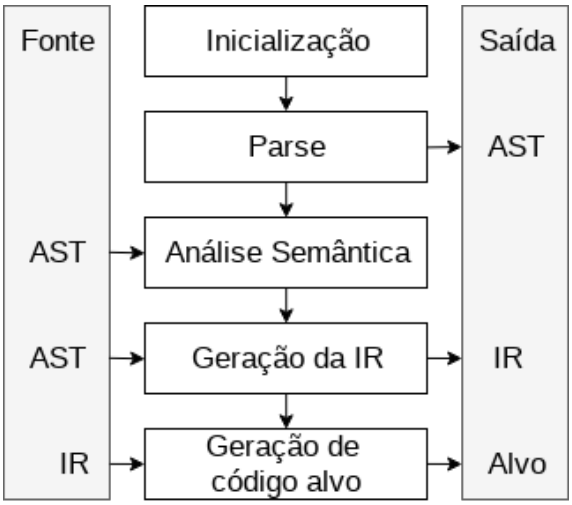

Figura 14 – Ciclo de compilação.

Fonte – Elaborada pelo autor.

4.9.2.1 ParseFileFunction

A função de *parse* é responsável por executar as análises léxica e sintática do arquivo contendo o programa fonte. Cada função é vinculada a uma extensão de arquivo. Um exemplo da definição de uma função de parse pode ser visto no Código 4.10 . Durante o processo de compilação o compilador verifica a existência de uma função de parse para o arquivo de entrada. Caso o arquivo não possua uma função o processo é abortado. Caso contrario a função de parse é chamada passando como parâmetro o nome do arquivo e a instância da árvore sintática abstrata. Cognite não implementa mecanismos para realizar as análises léxica e sintática. Sendo necessário a utilização de ferramentas de terceiros.

Existem diversas opções disponíveis para realizar essa tarefa (PARR, 2013; CUP, 2018; JAVACC..., 2018).

```
1 public class Compiler extends cognite.compiler.Compiler {
2
3    public void Init(){
4
5        Templates.Register("mips", new MipsTemplateDescription());
6
7        Target.Register("mips", new MipsTargetTranslator());
8
9
10       Parse("go", (filename, ast)-> {
11          ANTLRFileStream    input = new ANTLRFileStream(filename, encType);
12          GoGrammarLexer lexer = new GoGrammarLexer(input);
13
14          CommonTokenStream tokens = new CommonTokenStream(lexer);
15          GoGrammarParser parser = new GoGrammarParser(tokens);
16
17          /* Inicia o parser da regra inicial. */
18          tree = parser.init();
19
20          listener = getListener();
21          listener.SetAst(ast);
22
23          walker = new ParseTreeWalker();
24          /* Percore a parse tree */
25          walker.walk(listener, tree);
26       });
27
28   }
29 }
```

Código 4.10 – Definição de uma função de parse para arquivos da extensão go.

4.9.3 *Front-end*

O *front-end* de Cognite conta com suporte inicial para um subconjunto da linguagem Go. Para a implementação do *front-end* as etapas de análise léxica e sintáticas foram realizadas pelo ANTLR (PARR, 2013) versão 4.

O ANTLR é um gerador de *parser*. Um parser organiza os *tokens* em uma estrutura. A estrutura pode vir a ser a árvore sintática abstrata. A utilização do ANTLR implica em três etapas:

- Definição de uma gramática da linguagem;
- Gerar um *lexer* e um *parser* na linguagem que se deseja trabalhar. ANTLR oferece suporte a várias linguagens alvo (PARR, 2013).
- Utilizar as classes geradas para realizar a varredura da árvore sintática concreta (*parse tree*).

Foi implementada a gramatica contendo um subconjunto da linguagem Go. Os resultados da avaliação da gramatica pelo ANTLR são as classes de análise léxica e sintática. Também é gerado *listiner* utilizada no momento em que a *parse tree* é percorrida. Um *listner* é uma classe que contem um conjuntos de funções de entrada e saída para cada regra da gramatica. Essas funções são invocadas quando o nó corresponde da árvore é acessado. O comportamento de cada método de entrada e saída pode ser sobrescrito pelo programador. Durante essa etapa é construída a AST utilizando a api definida na subseção 4.5.2.

4.10 Considerações Finais

Esse capítulo apresentou os componentes do *framework* Cognite. Foram expostos: as apis e tipos básicos, o modelo de representação de código, a RI, os gerados APM e MPM, o motor de renderização e o compilador Cognite. O capítulo seguinte aborda os resultados dos experimentos realizados entre os códigos gerados pelo compilador Cognite e os compiladores Clang e GCC.

5 Resultados Experimentais

Nesse capítulo são explorados os resultados dos experimentos realizados com o compilador Cognite. São apresentados: a metodologia, o conjunto de aplicações utilizado no experimento, os resultados da comporação entre o compilador Cognite e as soluções GCC e Clang e por fim as considerações finais.

5.1 Metodologia

Para avaliar a capacidade de geração de código do compilador Cognite foi adotado uma abordagem comparativa. Foram realizadas duas comparações. A primeira com o GCC e a segunda com o Clang. Ambas soluções de compiladores para a linguagem C maduras e consolidadas. Foram comparados os códigos assembly gerados para quatro aplicações descritas na subseção 5.1.1.

O processo de comparação levou em consideração os seguintes aspectos:

- Quantidade de operações de *load* e *store*;
- Quantidade de registradores utilizados para execução do programa;
- Tamanho da pilha alocada para chamadas de funções;
- Densidade do código gerado;
- Tempo de compilação.

A arquitetura alvo foi a MIPS. As aplicações para os compiladores GCC e Clang foram implementadas utilizando a linguagem de programação C. O código MIPS foi obtido através de cross compilação disponível nas duas ferramentas. O processo de cross compilação permite a geração de código diferente da maquina hospedeira. Enquanto que para o compilador Cognite foram implementadas na linguagem GO suportada pelo compilador.

É importante salientar que não foram utilizados recursos especiais da linguagem Go que de alguma modo viessem a influenciar os resultados comparativos. Todos os recursos empregados estão presentes nas linguagens e compiladores adotados.

GCC e Clang apresentam *flags* que habilitam diferentes tipos de otimização. A flag é especificada durante o processo de compilação por meio do parâmetro -On. Onde n representa um valor correspondente ao nível de otimização. Para cada aplicação foram

compiladas quatro versões com os diferentes tipos de optimização. Cognite ainda não conta com uma subdivisão de recursos de otimização. Por esse motivo apresenta apenas uma representação de código que foi comparada com os códigos gerados pelos compiladores GCC e Clang.

O processo executado para realizar as comparações consiste de:

- Obtenção dos códigos na representação de assembly MIPS;
- Extração das características descritas;
- Análise, representação gráfica e discussão dos resultados.

As duas primeiras etapas foram automatizadas. Um *script* escrito em JavaScript percorre todos os arquivos .c de um determinado diretório. O diretórios é responsável por armazenar os códigos fontes das aplicações na linguagem C e o resultada do processamento. O *script* verifica a existência dos diretórios gcc e clang. Caso algum não existe o mesmo é criado. Para cada aplicação são criados dois novos subdiretórios um para cada compilador. Dentro de cada subdiretório são inseridos os resultados da compilação do código fonte nos diferentes níveis de otimização. O resultado da compilação é um *dump* da linguagem de montagem no formato ELF (STANDARDS, 2001). Após o processo de compilação o *script* acessa o conteúdo do *assembly* MIPS de cada arquivo gerado e executa varreduras utilizando expressões regulares para extrair as características definidas. O resultado da análise é um arquivo contendo uma estrutura no formato JSON de fácil interpretação. O mesmo procedimento é realizado para o código gerado pelo compilador Cognite.

As subseções seguintes abordam: o conjunto de aplicações, ferramentas utilizadas, uma descrição dos gráficos, resultados obtidos nos testes e encerra com as considerações finais.

5.1.1 O Conjunto de Aplicações

Foi implementado um conjunto de quarto aplicações. O conjunto inclui:

- Um *bitcount* (IQBAL; LIANG; GRAHN, 2010) - consiste de uma aplicação que conta a quantidade de *bits* setados em um conjunto randômico de números. O programa realiza operações de *bitwise*, deslocamentos de bits e laços;
- Um filtro laplaciano - aplicada no processamento de imagem;
- Uma decomposição LU (TSENG; JAN; YANG,) - aplicada a álgebra linear;
- Uma multiplicação de matrizes;

As três ultimas aplicações apresentam computação intensiva, com muitos acessos a matrizes e instruções de salto.

Essas aplicações foram selecionadas para suprir as necessidades de outros trabalhos desenvolvidos dentro do mesmo laboratório. Outro ponto relevante é o fato de que elas apresentam características que abrangem uma grande quantidade de recursos de compilação como: utilização de laços, estruturas de controle de fluxo, variáveis globais, chamadas de funções, gerenciamento de memória e otimizações. Permitindo explorar com mais riqueza de detalhes o compilador Cognite.

5.1.2 O *Hardware* e Ferramentas

Para a execução dos testes foi utilizando uma maquina com processador Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz, 8GB de memória RAM, SSD de 256GB e sistema operacional Debian linux 9 x64. Foram instalados os compiladores GCC versão 6.3.0, Clang LLVM 6.0.1 e Java JDK 1.8.0_171. As instalações foram realizadas de forma padrão sem nenhum ajuste especifico em nenhuma das ferramentas. As aplicações adotadas nos testes foram implementadas nas linguagens de programação C e GO.

5.1.3 Leitura dos Gráficos

Todos os gráficos apresentam o mesma configuração. São apresentadas as quatro aplicações em ordem alfabética. Para cada aplicação são demonstrados os resultados obtidos a partir da compilação com diferentes níveis de otimização para os compiladores GCC e Clang. A ultima coluna de cada aplicação descreve o resultado do compilador Cognite. Em todos as figuras, com exceção das Figuras 17 e 23, o menor valor representa o melhor resultado. A legenda no canto superior direito sinaliza as cores de cada versão do código.

5.2 Avaliação Comparativa

Essa seção apresenta os resultados obtidos pela extração e análise dos códigos gerados pelos compiladores GCC, Clang e Cognite. A ordem de apresentação dos resultados segue: quantidade de operações *load*, quantidade de operações *store*, quantidade de registradores utilizados, quantidade de memória alocada na pilha e densidade do código.

Antes de iniciar a apresentação. Alguns conceitos devem ser entendidos para melhor leitura dos resultados. Operações de *load* e *store* tem grande impacto no tempo de execução de um programa devido ao elevado custo (NICOLAESCU; VEIDENBAUM; NICOLAU, 2003). Outro ponto importante tem relação com a quantidade de registradores utilizados. A correta utilização de registradores implica na redução da dependência de dados que por sua

vez aumenta o *instruction level parallelism* (ILP) da aplicação (RAU; FISHER, 1993). Nas aplicações Laplaciano, LU e Mult as matrizes utilizadas na computação foram declaradas de forma global. Por esse motivo foi considerada apenas a quantidade de memória alocada na pilha para a execução de cada aplicação. Por fim, as aplicações Laplaciano e Mult apresentaram anomalias nas versões compiladas com GCC e Clang quando foram adotados diferentes níveis de otimização. A aplicação Mult foi afetada em ambos os compiladores. Enquanto que a aplicação Laplaciano foi penalizada apenas no Clang. As anomalias consistem da ausência de diversas instruções que acarretaram a inválida computação das aplicações.

Os resultados foram divididos em dois grupos. O primeiro grupo apresenta a comparação do compilador Cognite com o compilador GCC. O segundo grupo apresenta os resultados da comparação entre Cognite e Clang.

5.2.1 Comparação entre Cognite e GCC

Essa subseção apresenta o conjunto de resultados obtidos da comparação entre o compilador Cognite e GCC. As Figuras 15 e 16 mostram a relação de operações de *load* e *store*. Cognite apresenta uma quantidade de operações *load* inferior para todas as aplicações. Apesar de GCC apresentar apenas uma operação de *load* para a aplicação Mult. Análises no código gerado pelo GCC com níveis de otimização 1,2 e 3 revelam a ausência de instruções tornando inválida a computação do programa.

Considerando a aplicação LU onde a variações foram maiores. A quantidade de operações de *load* presentes no código gerado pelo compilador Cognite tem uma redução que varia entre 29% e 91% quando comparado com o código gerado pelo compilador GCC com nível de otimização 2 e 0 respectivamente. O nível 2 foi selecionado em detrimento ao nível 3 por apresentar menor quantidade de instruções *load*. Também é observada uma redução nas operações de *store*. Mas uma vez tomando a aplicação LU como exemplo. A quantidade de operações de *store* geradas pelo compilador Cognite tem uma redução entre 84% e 92% quando comparados com a versão do GCC com otimização nível 3 e 0 respectivamente.

Uma das características que favorece o compilador Cognite é o fato de manter as variáveis em registradores pelo maior intervalo possível. Enquanto que as aplicações compiladas pelo GCC, mesmo com diferentes níveis de otimização, preservam o comportamento de *load* e *store* para alguns casos.

O compilador Cognite apresentou melhor exploração de registradores quando comparado com o compilador GCC. Quando comparado os vários níveis de otimização do GCC é possível perceber que o numero de registradores utilizado cresce com o nível de otimização. Entretanto na aplicação Mult ocorre o contrário. Isso está relacionado

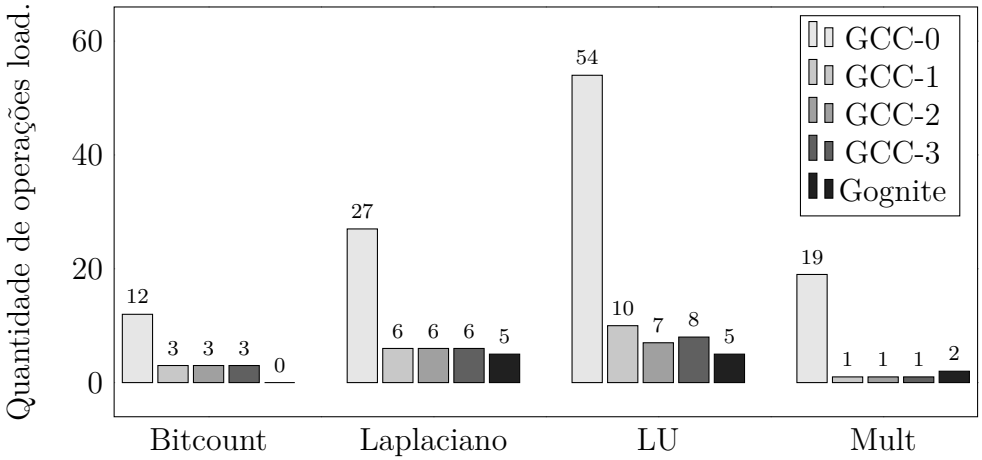

Figura 15 – Gráfico de operações load por aplicação e compilador. Quanto menor melhor.

Fonte – Elaborada pelo autor.

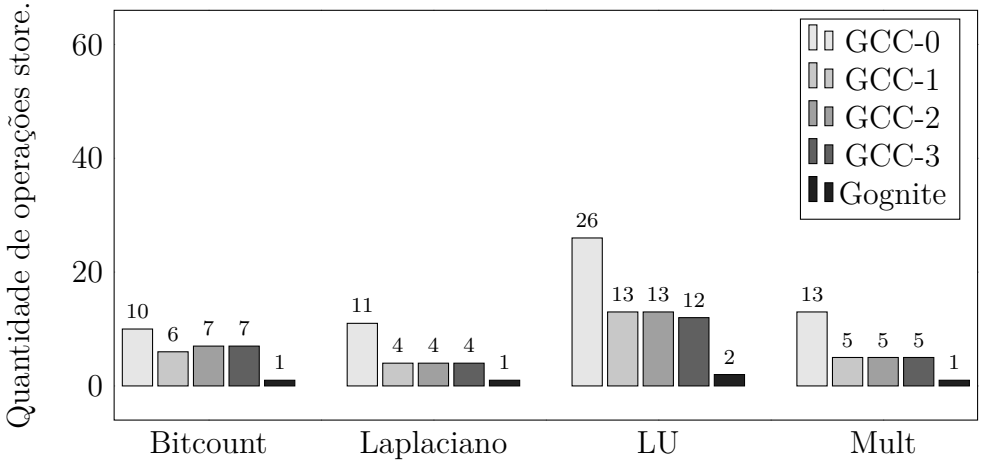

Figura 16 – Gráfico de operações *store* por aplicação e compilador. Quanto menor melhor.

Fonte – Elaborada pelo autor.

a quantidade reduzida de instruções. Os resultados da comparação estão expostos na Figura 17.

Quanto a memória alocada para a pilha apresentada na Figura 18 nota-se a ausência de alocação das variáveis de pilha no código gerado pelo compilador Cognite. Como as variáveis da pilha tem a função de controle dos laços de repetição e acumulação de resultados o compilador preserva os valores em registradores evitado operações de *load* e *store*. Enquanto que GCC apresenta uma baixa redução do uso de memória. Mesmo com nível elevado de otimização ativado. A memória alocada para a pilha além de desnecessária para as aplicações implica em um maior número de operações de *load* e *store* que podem ser visualizadas nas Figuras 15 e 16.

No aspecto densidade de código apresentado na Figura 19, o comportamento variou de acordo com a aplicação. Analisando o código gerado pelo GCC, observa-se a

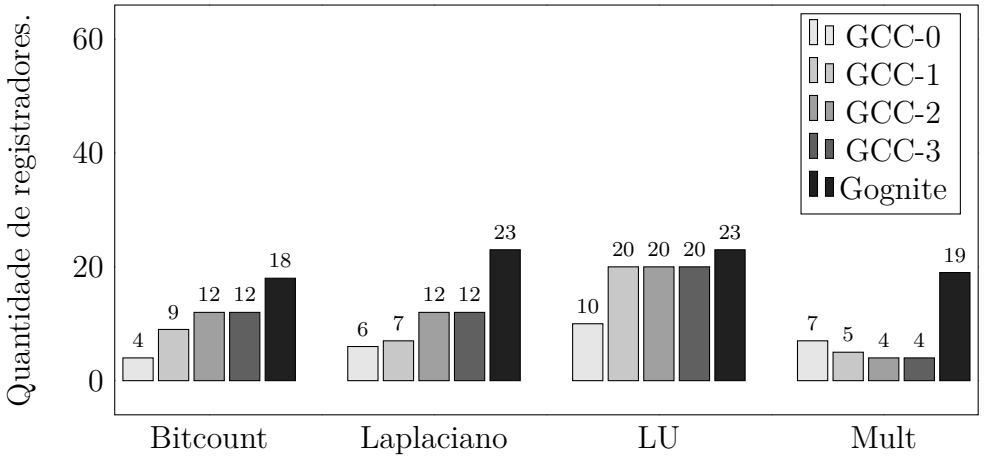

Figura 17 – Gráfico de utilização de registradores por aplicação e compilador. Quanto maior melhor.

Fonte – Elaborada pelo autor.

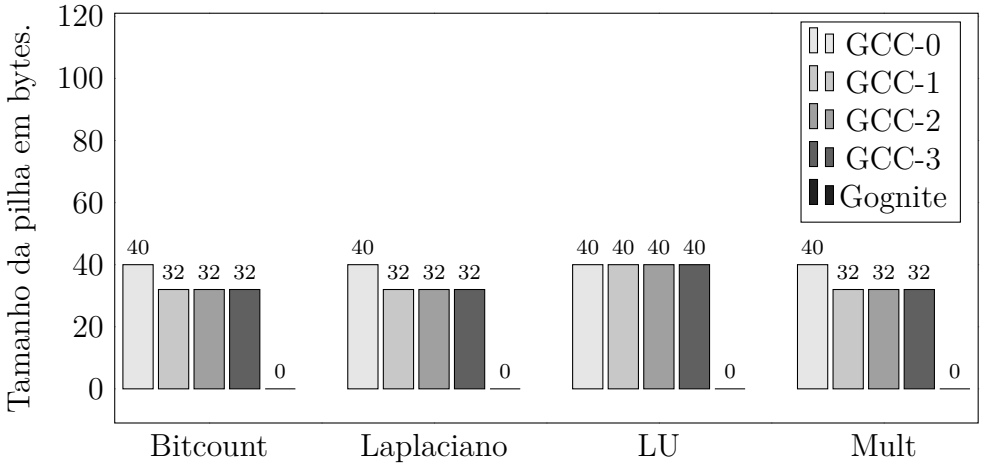

Figura 18 – Gráfico do tamanho da pilha por aplicação e compilador. Quanto menor melhor.

Fonte – Elaborada pelo autor.

existência de um código mais enxuto e otimizado para a aplicação Laplaciano. Muitas das instruções responsáveis por calcular o endereço de acesso a matriz foram substituídas por endereçamento direto. Isso não é uma surpresa considerando que GCC é uma ferramenta madura e conta com um conjunto maior de otimizações. Entretanto para a aplicação Bitcount o código gerado pelo compilador Cognite obteve um melhor resultado. Os dois compiladores apresentaram densidade semelhante para a aplicação LU considerando o maior nível de otimização do GCC. A aplicação Mult não foi abordada pelo fato de apresentar computação inválida.

A Figura 20 demonstra a comparação dos tempos de compilação para Cognite e GCC. É possível verificar que o tempo de compilação do compilador Cognite é superior ao do compilador GCC. O tempo de compilação, no pior caso, equivale a 16 vezes o tempo

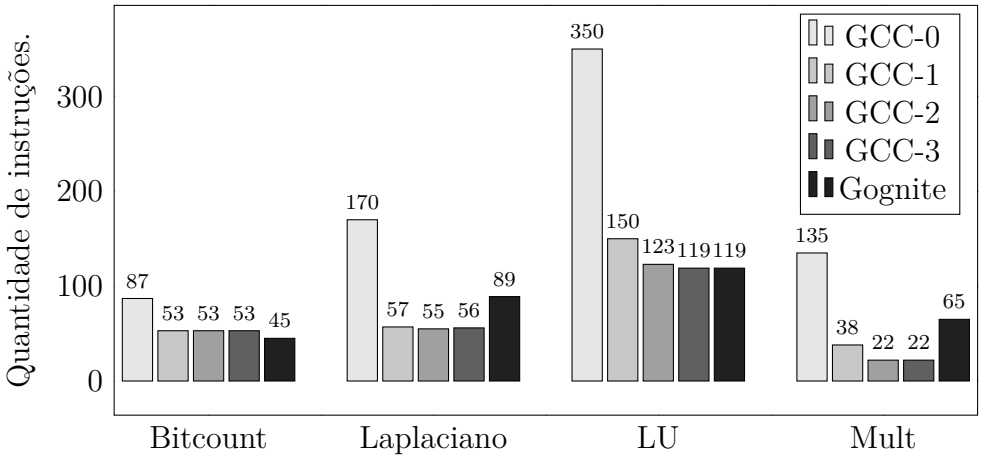

Figura 19 – Gráfico da quantidade de instruções por aplicação e compilador. Quanto menor melhor.

Fonte – Elaborada pelo autor.

gerado pelo compilador GCC. Os possiveis motivos são discutidos na seção 5.3.

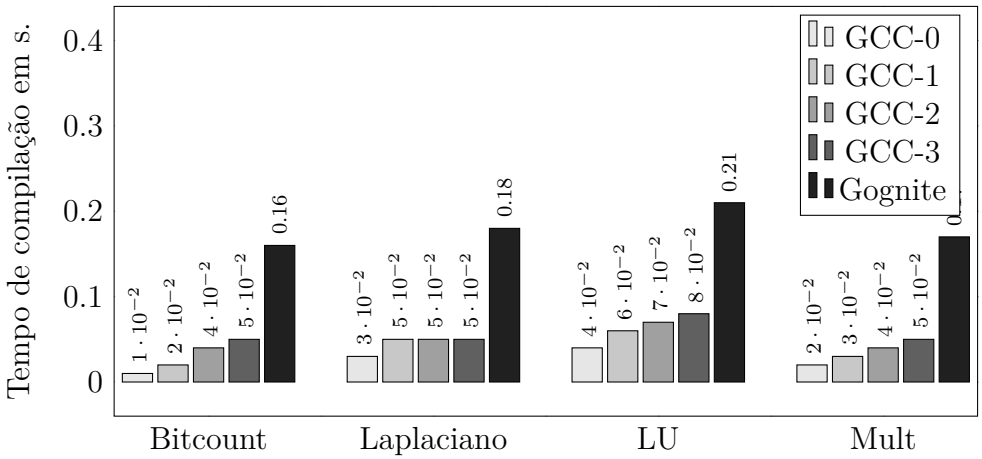

Figura 20 – Gráfico do tempo de compilação para cada aplicação e nível de otimização. Quanto menor melhor.

Fonte – Elaborada pelo autor.

5.2.2 Comparação entre Cognite e Clang

Essa subseção apresenta os resultados da comparação entre o compilador Cognite e Clang. Assim como o resultado da comparação com o Gcc. Cognite apresenta um valor inferior para a quantidade de instruções de *load* e *store* como mostram as Figuras 21 e 22. Tomando como exemplo a aplicação LU a redução na quantidade de operações de *load* varia entre 29% e 90% quando comparados com o resultado da compilação para os níveis de otimização 1 e 0. Ainda considerando a aplicação LU o nível 1 de otimização foi o que mais favoreceu a aplicação. Também é observada uma redução nas operações de *store*. Mas uma vez tomando a aplicação LU como exemplo. A quantidade de operações de *store*

geradas pelo compilador Cognite tem uma redução entre 60% e 96% quando comparados com a versão do Clang com otimização nível 1 e 3 respectivamente.

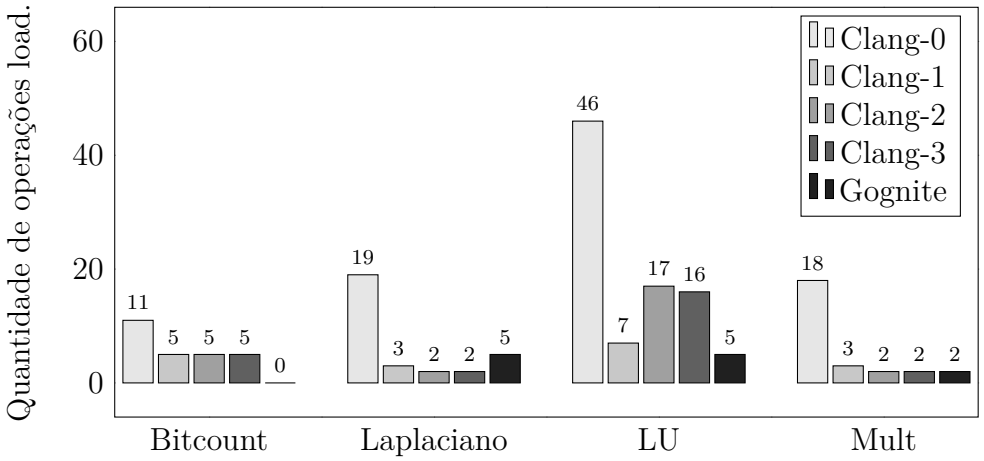

Figura 21 – Gráfico de operações load por aplicação e compilador. Quanto menor melhor.

Fonte – Elaborada pelo autor.

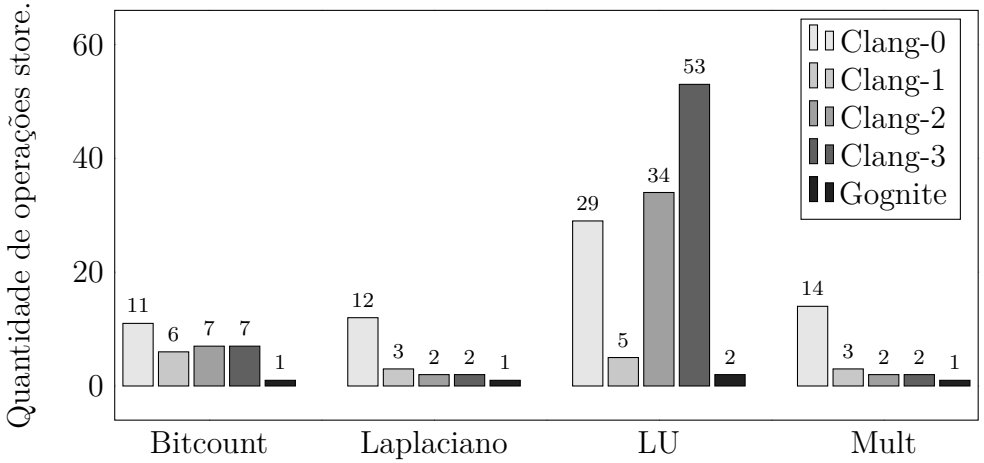

Figura 22 – Gráfico de operações store por aplicação e compilador. Quanto menor melhor.

Fonte – Elaborada pelo autor.

O resultado da comparação quanto a utilização de registradores está representando na Figura 23. Algumas considerações devem ser feitas. As aplicações Laplaciano e Mult apresentam computação inválida para os níveis de otimização 1,2 e 3. Por esse motivo devem ser ignorados os resultados com os níveis de otimização mencionados. A aplicação LU gerada pelo compilador Clang apresentou um maior número de registradores. Embora esteja relacionado a grande quantidade de instruções.

Ao analisar o comportamento da pilha geradas pelo compilador Clang. Observou-se uma maior redução quando comparado ao compilador GCC, exceto pelo caso da aplicação LU onde a flag de otimização -O3 acabou gerando um código com maior quantidade de instruções como mostra a Figura 25. O crescimento da quantidade de instruções também

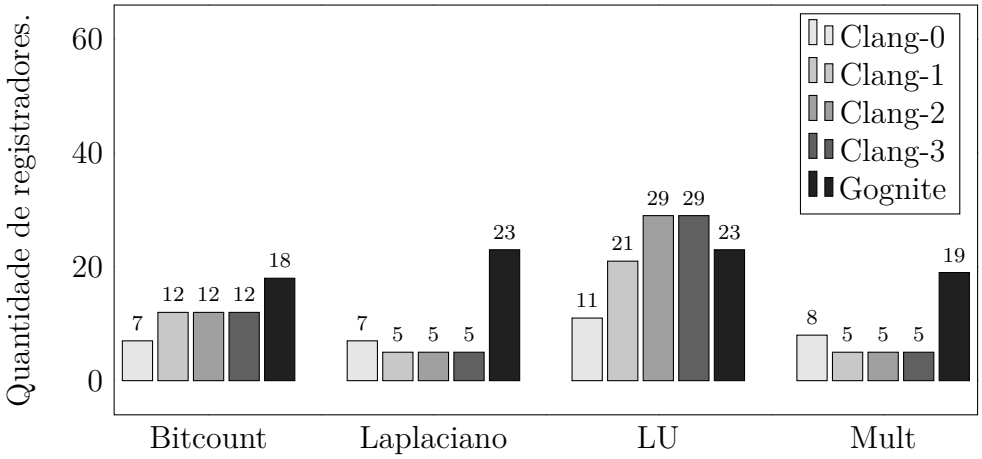

Figura 23 – Gráfico de utilização de registradores por aplicação e compilador. Quanto maior melhor.

Fonte – Elaborada pelo autor.

é observado na Figura 22 onde a quantidade de operações de *store* acaba sendo 82% maior que a versão compilada sem otimização.

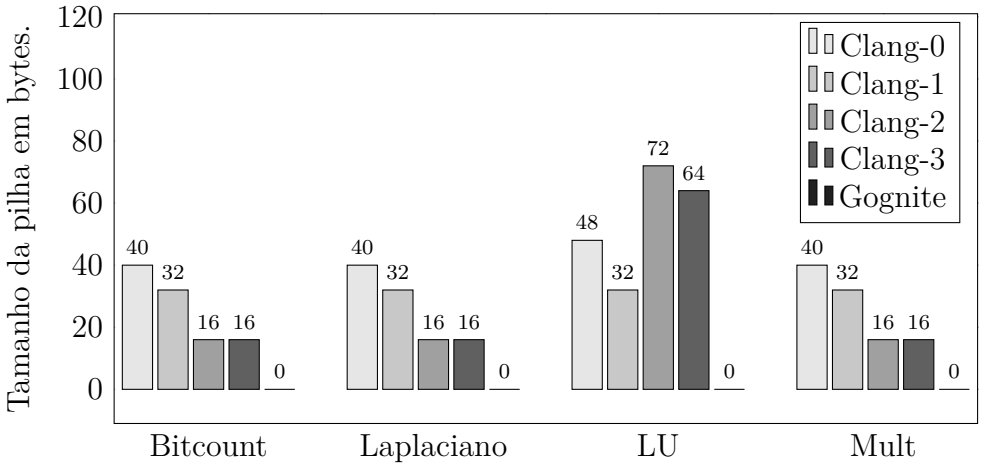

Figura 24 – Gráfico do tamanho da pilha por aplicação e compilador. Quanto menor melhor.

Fonte – Elaborada pelo autor.

No aspecto densidade de código apresentado na Figura 25, assim como na comparação com o GCC o comportamento variou de acordo com a aplicação. Cognite obteve melhores resultados nas aplicações *Bitcount* e *LU*, assim como observado na comparação da Figura 19. O compilador Clang não gerou código válido para as aplicações Laplaciano e Mult quando aplicados diferentes níveis de otimização inviabilizando a comparação.

A Figura 26 apresenta a comparação do tempo de compilação entre Cognite e Clang. É possível observar o mesmo comportamento demonstrado na Figura 20. Cognite apresenta tempo de compilação superior ao do Clang. No pior caso o tempo necessário

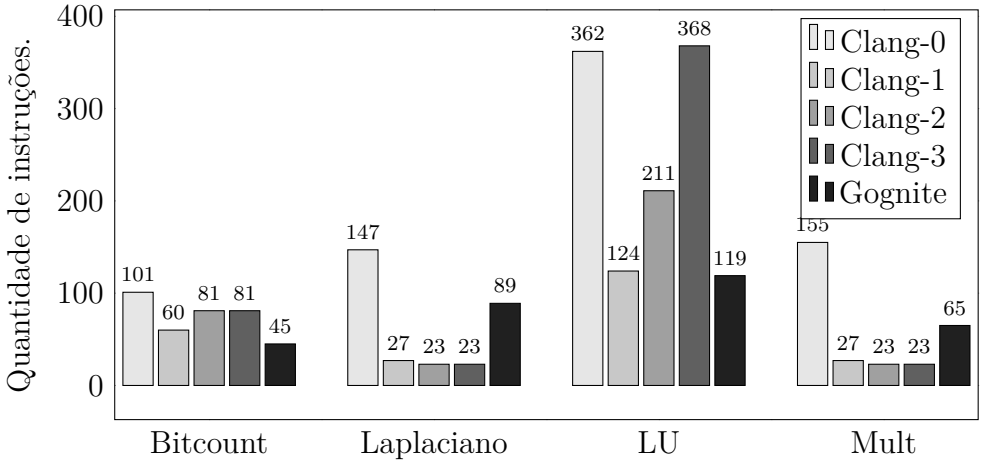

Figura 25 – Gráfico da quantidade de instruções por aplicação e compilador. Quanto menor melhor.

Fonte – Elaborada pelo autor.

correspondeu a 21 vezes o tempo gerado pelo compilador Clang. Outra informação é que para o conjunto de aplicações o compilador Clang também apresentou tempos de compilação menores que os do compilação GCC.

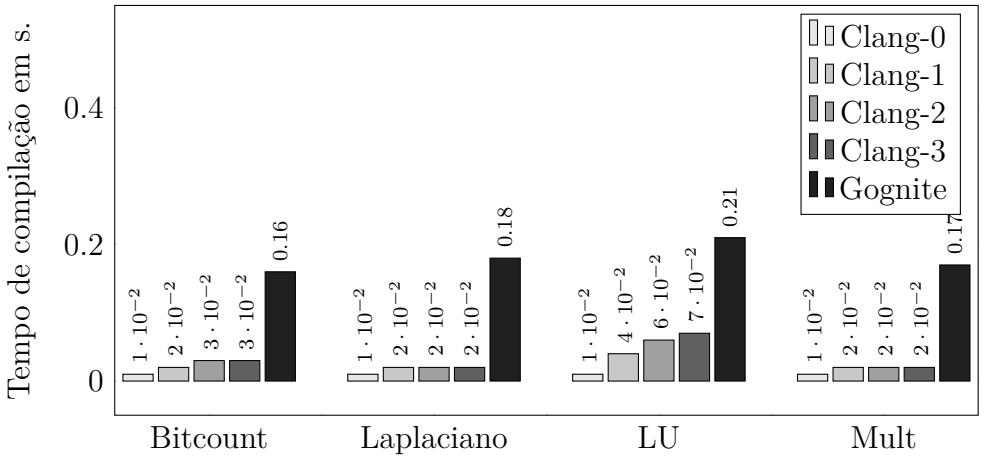

Figura 26 – Gráfico do tempo de compilação para cada aplicação e nível de otimização. Quanto menor melhor.

Fonte – Elaborada pelo autor.

5.3 Considerações Finais

Este capítulo apresentou os resultados da comparação entre o compilador Cognite e duas soluções amplamente utilizadas, GCC e Clang. Foram destacados os métodos e recursos empregados no experimento. Os resultados mostram que o compilador Cognite apesar da baixa maturidade apresentou desempenho igual ou superior para alguns dos

aspectos analisados. Os pontos de maior destaque são: a redução da quantidade de operações de *load* e *store*, bem como a melhor utilização dos registradores e memória disponíveis. Ambos os compiladores, GCC e Clang, pecam quanto a utilização de memória, uma vez que os programas gerados pelo compilador Cognite não fizeram uso da pilha e apresentaram resultados válidos para computação. Os resultados também apontam as fragilidades do compilador Cognite onde devem ser empreendidos esforços para a melhoria da solução. O conjunto de aplicações selecionado permitiu explorar diferentes recursos do compilador. Certamente que os resultados não são conclusivos mas revelam a capacidade de geração de código e exploração dos recursos do *framework* pelo compilador Cognite.

Outro ponto importante é o fator tempo de compilação de Cognite que parece lento. Cognite fornece abstrações de alto nível que são compiladas em código de máquina eficiente. O processo de tradução e verificação dessas estruturas demanda tempo.

O tempo de compilação de Cognite não é tão ruim quanto parece e existem razões para acreditar que ele vai melhorar. Dentre os fatores que afetam o tempo de compilação são ressaltados alguns aspectos da linguagem Java e da própria implementação do compilador. Primeiro, Cognite é implementado em Java que é executado sobre uma maquina virtual. O processo de execução do compilador em si sofre uma sobrecarga da plataforma. Enquanto que Clang e GCC executam diretamente na maquina hospedeira. Em segundo lugar, deve ser considerado o modelo de compilação. Clang e GCC são formados por um conjunto de aplicações compiladas que geram estruturas temporárias para acelerar o processo de compilações futuras de uma mesma aplicação. Enquanto Cognite apesar de compilado apresenta uma sobrecarga na inicialização decorrente da carga das descrições de código alvo e da representação intermediária, um custo gerado pela flexibilidade de representação. E por fim, Cognite possui um sistema de tipos moderadamente complexo e gasta uma quantia não desprezível de tempo de compilação, aplicando as restrições que tornam o código gerado mais seguro em tempo de execução. Uma proposta de melhoria está descrita na seção 6.2.

O capítulo seguinte trata dos resultados desse trabalho. São apresentados os trabalhos futuros e pretensões para a solução Cognite.

6 Conclusões

Nesse capítulo são apresentados: os resultados obtidos até o presente momento, os trabalhos futuros com a perspectiva de aplicação do compilador como suporte para uma solução integrada de memorias cache tolerante a falhas e por fim são relatadas as considerações finais da dissertação.

6.1 Resultados desse trabalho

Dentre os resultados obtidos por esse trabalho são evidenciados:

- Uma versão completa e funcional do *framework* Cognite, contendo os recursos descritos no Capítulo 4;
- Um compilador construído com o *framework*. O compilador é composto por um *front-end* com suporte a linguagem Go e um *back-end* com geração de código para a arquitetura alvo MIPS. A atual versão do compilador implementa todo o *front-end* Go nos aspectos sintáticos e semânticos. O desenvolvimento do compilador foi dirigido pelas necessidades do laboratório. Embora o *front-end* forneça suporte a todos os tipos de dados a arquitetura alvo apenas implementam tipos inteiros. Existe também uma limitação quanto ao tratamento de *overflow* não implementados na arquitetura. A evolução do compilador segue o modelo de codesign com as arquiteturas desenvolvidas no laboratório.
- Um motor de renderização baseado na interpolação de expressões contidas em uma *string* com um objeto de contexto.

6.2 Trabalhos futuros

Enquanto significativos progressos foram realizados até o presente momento da defesa dessa dissertação. Existem atividades remanescentes que precisam ser concluídas. Algumas delas implicam diretamente na construção do *framework*, outras em sua validação.

O projeto Cognite possui as seguintes atividades em andamento, listadas por grau de dificuldade e dependência:

- Migração do *framework* para a linguagem Go - a versão atual foi desenvolvida com a linguagem de programação Java versão 8. Boa parte da justificava de usar Java foi o utilização dos recursos disponibilizados pelo ANTLR, analisadores léxicos e

sintáticos. Entretanto os resultados apresentados no Capítulo 5 mostram que tempo de compilação pode chegar a ser, no pior caso, 21 vezes maior quando comparado com Clang e GCC. Várias etapas do processo de compilação podem executados de forma concorrente sem prejuízo ao resultado e atualmente não são. Visando uma melhoria no tempo de compilação pretende-se:

  - Migrar o *framework* para a linguagem Go. Afim de explorar os recursos de programação concorrente disponibilizados pela linguagem e verificar a sobrecarga gerada pela linguagem Java.
  - Implementar um mecanismo que converta os arquivos de descrição em estruturas do próprio *framework* evitando a sobrecarga gerada durante o processo de inicialização do compilador.

  A migração manterá a utilização do ANTLR que passou a gerar as ferramentas também para a linguagem Go.

- Aprimorar os mecanismos de *profiling* e *debug* - essa atividade implica em melhorias nas ferramentas de análise. Apesar de contar com um conjunto mínimo de ferramentas que permitem obter informações seção 4.3 . O mecanismos ainda é bastante textual o que torna o processo de análise pouco claro. Pretende-se desenvolver uma interface que permita gerenciar a execução das aplicações e facilitar a leitura dos dados coletados durante o processo de *profiling* e *debug*.

- Desenvolvimento de um runtime - o desenvolvimento da biblioteca de runtime possibilitará a exploração dos recursos de concorrência disponibilizados pela linguagem Go. Além de fornecer mecanismos para enriquecer os resultados das simulações. Por meio dele serão possíveis: o gerenciamento de pilhas em ambientes multiprocessados, alocações dinâmicas e a execução de *gorotines*. A linguagem Go implementa um mecanismos chamado *Contiguous stack* (RANDALL, 2018) que deverá ser implementado no *runtime*.

6.3 Conclusão

A simplicidade e a redução da curva de aprendizado são os pontos principais do *framework* Cognite. Os mecanismos apresentados nessa dissertação fornecem aos desenvolve dores um meio ágil, flexível e com grande poder de representação. Tais mecanismos possibilitam a construção de ferramentas de análise e geradores de código para simulação de arquiteturas não convencionais onde a escassez de ferramentas de geração de código e suporte de sistema operacional são conhecidos(FERREIRA, 2016).

Esse trabalho não pretende concorrer diretamente com LLVM. Uma vez que o período de 2 anos não seria suficiente para expressar todos os avanços conquistados durante

os 15 anos do projeto LLVM. Entretanto, almeja-se inserir o *framework* Cognite como uma alternativa onde o foco seja o desenvolvimento controlado de aplicações para a simulação de diferentes situações em projetos de arquiteturas de *hardware*. Uma outra alternativa seria o desenvolvimento de *runtimes* para sistemas embarcados onde os recursos de *hardware* e *software* são limitados (LEUPERS, 2002). Para suprir as deficiências quanto a geração de aplicações executáveis dentro de uma ambiente de sistema operacional o *framework* pode ser integrado a alguma ferramenta para a geração de arquivos ELF (PLAN..., 2018; LLD..., 2018). Ainda é possível a construção de compiladores *source-to-source* que utilizem outras linguagens e compiladores preexistentes com esse proposito.

Os resultados apresentados no Capítulo 5 revelam que apesar da pouca maturidade e com um conjunto mínimo de otimizações Cognite obteve um bom posicionamento quando comparado com soluções consolidadas como Clang e GCC, para o conjunto de aplicações analisado. Atividades em andamento pretendem melhor a performance reduzindo o tempo de compilação. Acredita-se que esse trabalho possa impulsionar os estudos nas áreas de *hardware*/*software codesign* e construção de compiladores com foco na exploração de paralelismo em arquiteturas não convencionais.

Referências

ADRIAANSEN, M. et al. Code generation for reconfigurable explicit datapath architectures with llvm. In: *2016 Euromicro Conference on Digital System Design (DSD)*. [S.l.: s.n.], 2016. p. 30–37. Citado na página 1.

AHO, A. et al. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2011. ISBN 9780133002140. Disponível em: <https://books.google.com.br/books?id=NTIrAAAAQBAJ>. Citado 4 vezes nas páginas 5, 6, 7 e 8.

BEETZ, K.; BÖHM, W. Challenges in engineering for software-intensive embedded systems. In: _____. *Model-Based Engineering of Embedded Systems: The SPES 2020 Methodology*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 3–14. ISBN 978-3-642-34614-9. Disponível em: <https://doi.org/10.1007/978-3-642-34614-9_1>. Citado na página 1.

BURROWS, M.; FREUND, S. N.; WIENER, J. L. Run-time type checking for binary programs. In: HEDIN, G. (Ed.). *Compiler Construction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. p. 90–105. ISBN 978-3-540-36579-2. Citado na página 7.

CATANZARO, B. et al. Ubiquitous parallel computing from berkeley, illinois, and stanford. *IEEE Micro*, v. 30, n. 2, p. 41–55, March 2010. ISSN 0272-1732. Citado na página 1.

CHANGESET 195562 – WebKit. 2016. <https://trac.webkit.org/changeset/195562/webkit>. (Accessed on 08/10/2018). Citado na página 16.

CLANG C Language Family Frontend for LLVM. <https://clang.llvm.org/>. (Accessed on 08/11/2018). Citado na página 14.

COMPILERS.NET > paedia > compiler. <http://www.compilers.net/paedia/compiler/index.htm>. (Accessed on 08/07/2018). Citado na página 8.

CRESSLER, J.; MANTOOTH, H. *Extreme Environment Electronics*. CRC Press, 2012. (Industrial Electronics). ISBN 9781439874318. Disponível em: <https://books.google.com.br/books?id=-gLSBQAAQBAJ>. Nenhuma citação no texto.

CUDA LLVM Compiler | NVIDIA Developer. 2018. <https://developer.nvidia.com/cuda-llvm-compiler>. (Accessed on 08/06/2018). Citado 2 vezes nas páginas 1 e 14.

CUP. 2018. <http://www2.cs.tum.edu/projects/cup/>. (Accessed on 08/07/2018). Citado 2 vezes nas páginas 17 e 65.

CYTRON, R. et al. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 13, n. 4, p. 451–490, out. 1991. ISSN 0164-0925. Disponível em: <http://doi.acm.org/10.1145/115372.115320>. Citado na página 13.

DE, S. K. et al. Development of an efficient dsp compiler based on open64. 08 2018. Citado na página 15.

DEFINING and Starting a Thread (The Java™ Tutorials > Essential Classes > Concurrency). 2017. <https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html>. (Accessed on 08/07/2018). Citado na página 9.

FERREIRA, J. C. dos S. *CLEM & OCEAN: Dois Compiladores OpenCL para as Arquiteturas Manycore METAL e ArachNoC*. Dissertação (Mestrado) — Universidade Federal do Piauí, Piauí, sep. 2016. Citado na página 80.

FREQUENTLY Asked Questions · The Rust Programming Language. 2018. <https://www.rust-lang.org/en-US/faq.html>. (Accessed on 08/11/2018). Citado 2 vezes nas páginas 1 e 15.

FREQUENTLY Asked Questions (FAQ) - The Go Programming Language. 2018. <https://golang.org/doc/faq#csp>. (Accessed on 08/07/2018). Citado 3 vezes nas páginas 2, 9 e 15.

GIELEN, G. et al. Emerging yield and reliability challenges in nanometer cmos technologies. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. New York, NY, USA: ACM, 2008. (DATE '08), p. 1322–1327. ISBN 978-3-9810801-3-1. Disponível em: <http://doi.acm.org/10.1145/1403375.1403694>. Nenhuma citação no texto.

GIVING up on Julia. 2016. <http://www.zverovich.net/2016/05/13/giving-up-on-julia.html>. (Accessed on 08/10/2018). Citado 2 vezes nas páginas 15 e 16.

GOOGLE Trends. 2018. <https://trends.google.com.br/trends/?geo=BR>. (Accessed on 08/11/2018). Citado na página 15.

GOOGLE'S Go: A New Programming Language That's Python Meets C++ | TechCrunch. 2009. <https://techcrunch.com/2009/11/10/google-go-language/>. (Accessed on 08/07/2018). Citado na página 9.

HALL, M.; PADUA, D.; PINGALI, K. Compiler research: The next 50 years. *Commun. ACM*, ACM, New York, NY, USA, v. 52, n. 2, p. 60–67, fev. 2009. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1461928.1461946>. Citado na página 1.

HOME - OpenMP. 2018. <https://www.openmp.org/>. (Accessed on 08/07/2018). Citado 2 vezes nas páginas 1 e 8.

INTRODUCING the B3 JIT Compiler | WebKit. 2016. <https://webkit.org/blog/5852/introducing-the-b3-jit-compiler/>. (Accessed on 08/10/2018). Citado na página 16.

IQBAL, S. M. Z.; LIANG, Y.; GRAHN, H. Parmibench - an open-source benchmark for embedded multiprocessor systems. *IEEE Computer Architecture Letters*, v. 9, n. 2, p. 45–48, Feb 2010. ISSN 1556-6056. Citado na página 68.

JAVA bytecode:. 2001. <https://www.ibm.com/developerworks/library/it-haggar_bytecode/index.html>. (Accessed on 08/07/2018). Citado na página 8.

JAVACC - The Java Parser Generator. 2018. <https://javacc.org/>. (Accessed on 08/07/2018). Citado 2 vezes nas páginas 17 e 65.

JUN, H. et al. Hbm (high bandwidth memory) dram technology and architecture. In: *2017 IEEE International Memory Workshop (IMW)*. [S.l.: s.n.], 2017. p. 1–4. Citado na página 14.

KHALDI, D.; CHAPMAN, B. Towards automatic hbm allocation using llvm: A case study with knights landing. In: *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. [S.l.: s.n.], 2016. p. 12–20. Citado na página 14.

KHUDIA, D. S.; WRIGHT, G.; MAHLKE, S. Efficient soft error protection for commodity embedded microprocessors using profile information. In: *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*. New York, NY, USA: ACM, 2012. (LCTES '12), p. 99–108. ISBN 978-1-4503-1212-7. Disponível em: <http://doi.acm.org/10.1145/2248418.2248433>. Nenhuma citação no texto.

KOLEK, J. et al. Adding micromips backend to the llvm compiler infrastructure. In: *2013 21st Telecommunications Forum Telfor (TELFOR)*. [S.l.: s.n.], 2013. p. 1015–1018. Citado na página 14.

KOREN, I.; KRISHNA, C. M. *Fault-Tolerant Systems*. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN 0120885255, 9780120885251. Nenhuma citação no texto.

KOTLIN/NATIVE - Kotlin Programming Language. 2018. <https://kotlinlang.org/docs/reference/native-overview.html>. (Accessed on 08/11/2018). Citado na página 15.

LANGUAGE Compatibility. 2018. <https://clang.llvm.org/compatibility.html>. (Accessed on 08/08/2018). Citado na página 14.

LATTNER, C. *LLVM Project Blog: What Every C Programmer Should Know About Undefined Behavior #1/3*. 2011. <http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html>. (Accessed on 08/13/2018). Citado na página 1.

LEUPERS, R. Compiler design issues for embedded processors. *IEEE Design Test of Computers*, v. 19, n. 4, p. 51–58, July 2002. ISSN 0740-7475. Citado na página 81.

LEUPERS, R. *Code Optimization Techniques for Embedded Processors: Methods, Algorithms, and Tools*. Springer US, 2013. ISBN 9781475731699. Disponível em: <https://books.google.com.br/books?id=bIvgBwAAQBAJ>. Citado na página 5.

LICENSE - The Go Programming Language. <https://golang.org/LICENSE>. (Accessed on 08/07/2018). Citado na página 9.

LLD - The LLVM Linker — lld 8 documentation. 2018. <http://lld.llvm.org/>. (Accessed on 08/12/2018). Citado na página 81.

LLOYD, J. Electromigration for designers: An introduction for the non-specialist. 2002. Nenhuma citação no texto.

LLVM. 2018. (Accessed on 08/06/2018). Disponível em: <http://aosabook.org/en/llvm.html>. Citado na página 13.

LLVM - Pesquisar - Google Trends. <https://trends.google.com/trends/explore?date=all&q=LLVM>. (Accessed on 08/11/2018). Citado na página 16.

LLVM-HS: General purpose LLVM bindings. <https://hackage.haskell.org/package/llvm-hs>. (Accessed on 08/10/2018). Citado na página 15.

LUA-USERS wiki: Llvm Lua. 2018. <http://lua-users.org/wiki/LlvmLua>. (Accessed on 08/11/2018). Citado na página 15.

LUZ, L. O. *ArachNoc : Um processador manycore com nós de processamento multicore suportando o modelo de programação IPNoSys*. Dissertação (Mestrado), sep 2016. Citado na página 3.

MIURA, Y.; MATUKURA, Y. Investigation of silicon-silicon dioxide interface using mos structure. *Japanese Journal of Applied Physics*, v. 5, n. 2, p. 180, 1966. Disponível em: <http://stacks.iop.org/1347-4065/5/i=2/a=180>. Nenhuma citação no texto.

MOORE, G. E. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, v. 11, n. 5, p. 33–35, Sept 2006. ISSN 1098-4232. Citado na página 1.

NEPOMUCENO, R. S. *METAL: Uma Plataforma Manycore de Propósito Geral Adaptada ao Modelo de Programação OpenCL*. Dissertação (Mestrado), sep 2016. Citado na página 3.

NEZZARI, Y.; BRIDGES, C. P. Modelling processor reliability using llvm compiler fault injection. In: *2018 IEEE Aerospace Conference*. [S.l.: s.n.], 2018. p. 1–10. Citado na página 14.

NICHOLAS Nethercote | Notes on Firefox, MemShrink, JavaScript, and more. 2018. <https://blog.mozilla.org/nnethercote/>. (Accessed on 08/10/2018). Citado na página 16.

NICOLAESCU, D.; VEIDENBAUM, A.; NICOLAU, A. Reducing data cache energy consumption via cached load/store queue. In: *Proceedings of the 2003 International Symposium on Low Power Electronics and Design, 2003. ISLPED '03.* [S.l.: s.n.], 2003. p. 252–257. Citado na página 69.

OPENCL Overview - The Khronos Group Inc. <https://www.khronos.org/opencl/>. (Accessed on 08/13/2018). Citado na página 1.

PARR, T. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013. ISBN 9781680505009. Disponível em: <https://books.google.com.br/books?id=gA9QDwAAQBAJ>. Citado 3 vezes nas páginas 5, 17 e 65.

PHANI, T. S. S.; KRISHNA, B. A.; SENAPATI, R. K. Survey on multigrained reconfigurable architecture using parallel mapping method. *Indian Journal of Science and Technology*, v. 10, n. 6, 2017. ISSN 0974 -5645. Disponível em: <http://www.indjst.org/index.php/indjst/article/view/110837>. Citado na página 1.

PLAN 9 from Bell Labs. 2018. <https://9p.io/plan9/>. (Accessed on 08/12/2018). Citado na página 81.

POSIX Threads Programming. 2017. <https://computing.llnl.gov/tutorials/pthreads/>. (Accessed on 08/07/2018). Citado na página 9.

PRICE, A. de A.; TOSCANI, S.; UFRGS., I. de Informática da. *Implementação de linguagens de programação: compiladores*. Sagra-Luzzatto, 2000. (Série Livros Didáticos). ISBN 9788524106392. Disponível em: <https://books.google.com.br/books?id=OhhOuQAACAAJ>. Citado na página 5.

PSIAKIS, R.; KRITIKAKOU, A.; SENTIEYS, O. Run-time instruction replication for permanent and soft error mitigation in vliw processors. In: *2017 15th IEEE International New Circuits and Systems Conference (NEWCAS)*. [S.l.: s.n.], 2017. p. 321–324. Nenhuma citação no texto.

QBE - Compiler Backend. 2018. <https://c9x.me/compile/>. (Accessed on 08/11/2018). Citado na página 15.

QINSB is not a Software Blog: Unladen Swallow Retrospective. 2011. <http://qinsb.blogspot.com/2011/03/unladen-swallow-retrospective.html>. (Accessed on 08/13/2018). Citado na página 1.

RANDALL, K. *Contiguous stacks*. Google, 2018. Disponível em: <https://docs.google.com/document/d/1wAaf1rYoM4S4gtnPh0zOlGzWtrZFQ5suE8qr2sD8uWQ/pub>. Citado na página 80.

RAU, B. R.; FISHER, J. A. Instruction-level parallel processing: History, overview, and perspective. *J. Supercomput.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 7, n. 1-2, p. 9–50, maio 1993. ISSN 0920-8542. Disponível em: <http://dx.doi.org/10.1007/BF01205181>. Citado na página 70.

ROSCOE, A. W. *The Theory and Practice of Concurrency*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997. ISBN 0136744095. Citado na página 9.

ROSE compiler infrastructure. 2018. <http://rosecompiler.org/>. (Accessed on 08/10/2018). Citado na página 15.

SADI, M. S. et al. An efficient approach towards mitigating soft errors risks. *CoRR*, abs/1110.3969, 2011. Disponível em: <http://arxiv.org/abs/1110.3969>. Nenhuma citação no texto.

SERRANO, M. Inline expansion: When and how? In: GLASER, H.; HARTEL, P.; KUCHEN, H. (Ed.). *Programming Languages: Implementations, Logics, and Programs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997. p. 143–157. ISBN 978-3-540-69537-0. Citado na página 56.

STANDARDS, T. I. Executable and linkable format (elf). *Specification, Unix System Laboratories*, 2001. Disponível em: <http://scholar.google.com/scholar?hl=en{&}btnG=Search{&}q=intitle:Executable+and+Linkable+Format+(+E>. Citado na página 68.

SWIFT - Apple Developer. 2018. <https://developer.apple.com/swift/>. (Accessed on 08/06/2018). Citado na página 14.

SWIFT.ORG - Compiler and Standard Library. 2018. <https://swift.org/compiler-stdlib/>. (Accessed on 08/11/2018). Citado na página 15.

THE Go Programming Language. 2018. <https://golang.org/>. (Accessed on 08/07/2018). Citado 4 vezes nas páginas 1, 9, 10 e 11.

THE Julia Language. 2018. <https://julialang.org/>. (Accessed on 08/10/2018). Citado na página 15.

THE LLVM Compiler Infrastructure Project. <http://llvm.org/pubs/>. (Accessed on 08/11/2018). Citado na página 15.

THE LLVM Compiler Infrastructure Project. LLVM Developer Group, 2018. Disponível em: <http://llvm.org/>. Citado 2 vezes nas páginas 1 e 13.

TIAN, X. et al. Llvm framework and ir extensions for parallelization, simd vectorization and offloading. In: *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. [S.l.: s.n.], 2016. p. 21–31. Citado na página 14.

TSENG, H.-R.; JAN, R.-H.; YANG, W. A bilateral remote user authentication scheme that preserves user anonymity. *Security and Communication Networks*, v. 1, n. 4, p. 301–308. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.26>. Citado na página 68.

TU, H.-Y.; TASNEEM, S. Re-evaluation of fault tolerant cache schemes. In: . [S.l.: s.n.], 2007. Nenhuma citação no texto.

YAHAGI, Y. et al. Threshold energy of neutron-induced single event upset as a critical factor. In: *2004 IEEE International Reliability Physics Symposium. Proceedings.* [S.l.: s.n.], 2004. p. 669–670. Nenhuma citação no texto.