Representações Internas e Geração de Códigos no Compilador Redirecionável Xingó

Cristiano Lino Felício

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Representações Internas e Geração de Códigos no Compilador Redirecionável Xingó

Cristiano Lino Felício[1]

Fevereiro de 2005

Banca Examinadora:

- Paulo Cesar Centoducatte
  IC – UNICAMP (Orientador)
- Ricardo Luís de Freitas
  PUC – CAMPINAS
- Sandro Rigo
  PUC – CAMPINAS
- Rodolfo Jardim de Azevedo
  IC – UNICAMP

[1]**Apoio Financeiro Parcial : Coordenadoria de Aperfeiçoamento de Pessoal de Nível Superior (CAPES)**

Representações Internas e Geração de Códigos no Compilador Redirecionável Xingó

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Cristiano Lino Felício e aprovada pela Banca Examinadora.

Campinas, 28 de fevereiro de 2005.

Paulo Cesar Centoducatte
IC – UNICAMP (Orientador)

Guido Araújo
IC – UNICAMP (Co-orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Resumo

Devido ao aumento da complexidade dos novos processadores, especialmente processadores DSPs, a capacidade dos compiladores de gerar um código altamente otimizado para as novas arquiteturas de computadores é cada vez mais desafiador. O compilador Xingó tem como objetivos possibilitar pesquisas em otimização e geração de código para novas arquiteturas. Para permitir tais pesquisas, é necessário que o compilador possua uma infra-estrutura capaz de representar, de uma maneira simples e correta, o código nas diversas fases de compilação.

Este trabalho apresenta as principais representações de programa do compilador Xingó, bem como os módulos que fazem a tradução de um programa, em uma representação origem, para um programa em uma representação destino (intermediária ou final). Estes módulos compõem o *front-end* do compilador, que leva à Representação Intermediária Xingó, e parte do *back-end*, com destaque para a Infra-Estrutura de Geração de Código do Xingó.

A Representação Intermediária do Xingó é de fácil manipulação, independente de máquina e apresenta uma sintaxe muito próxima à da linguagem C, por isso tem permitido otimizações independente de máquina e pode ser traduzida em código C compilável. Já a Infra-Estrutura de Geração de Código apresenta facilidades no desenvolvimento de novos geradores de código, principalmente porque os detalhes da máquina alvo são externos ao compilador, sendo incluídos através de parâmetros configuráveis e módulos bem definidos.

Os testes realizados até o presente momento validam parte da Infra-Estrutura de Geração de Código e têm demonstrado uma boa qualidade do Código Intermediário Xingó. Os testes mostram que o Código Intermediário está sendo corretamente representado para todos os programas do *benchmark NullStone* (6611 programas) e para uma quantidade razoável de programas dos *benchmarks MediaBench* e *SPEC*. Os resultados alcançados

até o momento trazem novas oportunidades em pesquisas na área de compiladores, especialmente otimização e geração de código.

Abstract

Due to the increasing complexity of the new processors, mainly DSPs processors, the capacity of generating highly optimized code for the new computer architectures by the compilers is increasingly motivating. The Xingó compiler has as goal allow research in code optimization and code generation for new architectures. In order to enable such researches, is necessary to the compiler to provide an infrastructure capable of representing, in an easy and correct form, the code in the several compiling phases.

This work presents the main program representations of the Xingó compiler, and the modules that perform the conversion of a program, in a base representation, to a program in a destination representation (intermediate or ultimate). These modules constitute the *front-end* of the compiler, that generates the Xingó Intermediate Representation, and share of the *back-end*, with highlight to the Xingó Code Generation Infrastructure.

The Xingó Intermediate Representation is easy to use, it is machine independent and has a very approximate syntax of the C language, thereby it has enabled to perform machine independent optimizations and is able to be converted to C Code. Upon the Code Generation Infrastructure, it provides facilities to development of new code generators, mainly because the details of the target machine are outside to the compiler, been included across configurable parameters and well-defined modules.

The tests completed up to now validate piece of the Code Generation Infrastructure and reveal good quality of the Xingó Intermediate Code. The testes show that the Intermediate Code is been correctly generated to any programs of the *NullStone benchmark* (6611 programs) and also to a reasonable quantity of programs of the *MediaBench and SPEC benchmarks*. The test results acquired up to now conduce to new opportunities for researches on areas such as optimization and code generation.

Agradecimentos

Gostaria de agradecer primeiramente a Deus, por Ele ser o meu melhor amigo, por ter me dado o dom da vida, a oportunidade de estar aqui e por jamais ter me desamparado, permitindo que eu realizasse este sonho.

Quero também agradecer ao meu orientador, o Prof. Dr. Paulo Cesar Centoducatte, que sempre me deu a oportunidade de realizar este trabalho, mesmo quando as dificuldades surgiam. Os seus ensinamentos, tenho certeza, não ajudaram apenas na minha formação acadêmica, mas também na minha formação pessoal. Obrigado, Paulo, pelo incentivo e sugestões.

Não posso deixar de agradecer também ao Prof. Dr. Guido Araújo, que tanto auxiliou no desenvolvimento deste trabalho, principalmente apontando o próximo passo a ser tomado.

Ao meu companheiro de Projeto Xingó, o grande Wesley Attrot, que tanto contribuiu para a realização deste trabalho. Lembro-me das nossas conversas para decidir as estratégias do compilador Xingó. Não esqueço da nossa compilação, execução e depuração de códigos no papel. O *QuickSort* foi também um grande companheiro neste sentido, o Wesley sabe bem disso. Wesley, você é um amigo muito especial que eu ganhei durante este tempo de mestrado.

Ao professor na graduação e amigo, Alexandre Ribeiro, que sempre acreditou no meu potencial e que tanto me incentivou a vir para a UNICAMP realizar este trabalho. Ele foi fundamental nesta minha decisão.

Um agradecimento especial aos meus pais. Eles simplesmente fazem tudo que podem para ver os meus sonhos realizados. Papai, muito obrigado por tudo o que fez e que faz por mim. Sua dedicação pelo trabalho, pela família e, principalmente, sua honestidade são exemplos que levarei comigo para sempre. Muito obrigado a vocês, meus amados pais!

Aos meus grandes amigos Flávio, Hiata e Stela, pela amizade verdadeira desde os tempos de faculdade. O destino nos fez estarmos sempre próximos, e o apoio de vocês é fundamental para a minha vida. Flávio, você não é só um colega com quem divido a casa, você é mais que um irmão. Aliás, acho que nós dois deveríamos ter nascido irmãos. Dentre as muitas pessoas que conheci nestes "poucos anos de vida", você é uma das mais incríveis.

À todos os amigos de célula: Bete, Cleison, Eduardo (Du), Filipe, Grace, Judy, Juliana, Lélia, Ricardo, Ted e Thelen por serem a minha família em Campinas. Muito obrigado pelas orações e pelos momentos especiais que passamos juntos com Jesus!

À minha noiva, e se Deus quiser, futura esposa, meus agradecimentos pelo apoio e compreensão. O tempo que não pudemos estar juntos foram tão difícieis para você quanto para mim. Mas os grandes ideais é que fazem a vida ter valor, e estaremos sempre lutando, juntos, para vermos nossos sonhos realizados. Meu amor, muito obrigado pela paciência e carinho!

À CAPES pelo apoio financeiro a esta pesquisa. Ao CNPq pelo incentivo dado aos trabalhos de pesquisa do LSC (Laboratório de Sistemas de Computação), atavés do processo PDI&TI 55.2117/2002-1. À FAPESP pelo apoio financeiro dado ao Projeto Xingó, com o processo 2000/15083-9.

À Unicamp e ao Instituto de Computação por terem me acolhido e me proporcionado uma infra-estrutura de estudo e pesquisa de primeira grandeza.

À todos vocês, que direta ou indiretamente participaram deste sonho, obrigado!

C.L.F.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Os novos experimentos em otimização e geração de código precisam de uma infra-estrutura de compilação que seja simples de usar e que possa ser rapidamente adaptada para uma nova situação ou um novo experimento. O compilador Xingó tem evoluído no sentido de proporcionar uma infra-estrutura com tais características.

Como parte da infra-estrutura do Xingó, o presente trabalho apresenta as principais representações de programa utilizadas no compilador. Durante o processo de compilação, é comum o código de um programa estar representado de formas diferentes, dependendo da fase de compilação e das técnicas utilizadas [10]. Por isso, foram projetadas e implementadas representações de programa para melhor atender às necessidades impostas pelas técnicas empregadas na fase de compilação na qual a representação é utilizada. Adicionalmente, foram implementados módulos que fazem a tradução de um programa, em uma representação origem, para um programa em uma representação destino.

A principal representação de programa do compilador Xingó é a Representação Intermediária Xingó (XIR). Essa representação é independente de máquina, é de leitura simples e possui uma sintaxe muito próxima à da linguagem C. Uma característica importante dessa representação é que ela pode ser convertida em um código fonte C e este compilado com um compilador C para uma arquitetura existente, permitindo, de uma forma ágil, verificar a corretude, por exemplo, de uma nova otimização independente de máquina. Essa conversão foi realizada no trabalho de mestrado de W. Attrot [7].

Uma outra representação de programa importante dentro do compilador é a Representação Olive Xingó (XOR). Tal representação faz parte da Infra-Estrutura de Geração

de Código do Xingó. Esta infra-estrutura, além de modular, possui mecanismos que facilitam a geração de código redirecionável. A representação XOR é passada como entrada para o gerador de código sintetizado por Olive [45], que foi integrado à Infra-Estrutura de Geração de Código do Xingó, permitindo um desenvolvimento mais rápido de um novo gerador de código. A representação XOR é baseada em árvores e também é independente de máquina.

De modo a facilitar a alocação de registradores e otimizações dependentes de máquina, o Xingó mantém uma representação em memória do programa *assembly* que será gerado. Esta representação é chamada de Representação Assembly Xingó (XAR). Ela é baseada em lista de operações que correspondem às instruções *assembly* reconhecidas pelo montador da arquitetura alvo. Essa representação, portanto, é dependente da máquina alvo.

As representações XIR e XOR não precisam ser modificas para uma nova máquina alvo, mas a representação XAR precisa. Mas essa modificação é facilitada pelo fato de que as operações em XAR são implementadas através de classes genéricas em C++, que podem ser reusadas por diversos geradores de código.

A estrutura do compilador Xingó apresentada na seção seguinte ajuda a compreender melhor as representações de programa utilizadas dentro do compilador, bem como os módulos que fazem a tradução entre elas.

1.1 O Projeto Xingó

O trabalho apresentado nesta dissertação é parte do *Projeto Xingó* [9], que visa desenvolver a infra-estrutura de um compilador redirecionável (do inglês, *retargetable compiler*), tendo como objetivos permitir pesquisas em otimização e geração de código. A estrutura do compilador Xingó está ilustrada na figura 1.1 e descrita abaixo. Os módulos em destaque (bordas mais largas) na figura são aqueles que foram implementados durante a realização deste trabalho.

O Xingó é um compilador otimizante para a linguagem C, e foi desenvolvido na linguagem C++. O *front-end* do compilador *Xingó* é implementado pelo compilador LCC [14]. O *front-end* tem como entrada um arquivo escrito na linguagem ANSI C e o traduz para uma representação intermediária do LCC (LIR).

O Xingó implementa um módulo (*XIR Generator*) que converte o programa em LIR

para um programa em XIR. Um otimizador de código independente de máquina (*Optimizer*) aplica várias transformações sobre a representação XIR, no intuito de tentar melhorar a qualidade do código gerado pelo compilador.

Depois que a representação XIR é otimizada, a compilação pode seguir um dos caminhos: geração de código *assembly* ou geração de código C. No primeiro caminho, o módulo *XOR Generator* faz o mapeamento do código em XIR para um código em XOR. XIR mantém as operações do programa sob a forma de quádruplas[1], e XOR mantém as operações sob a forma de ASTs[2] (Abstract Syntax Tree). No segundo caminho, o gerador de código C (*C Code Generator*) percorre as estruturas de dados da representação XIR para gerar código C, que é então compilado por um compilador C (por exemplo, GCC) para produzir um código executável.

Como parte deste trabalho, foi incluído no compilador Xingó, o gerador de geradores de código Olive [45]. Olive tem como entrada uma gramática contendo uma descrição da máquina alvo (*Machine Description*) e produz um gerador de código para esta máquina.

O gerador de código (*Code Generator*) produzido por Olive toma como entrada um conjunto de árvores representadas em XOR, que tiveram os seus padrões descritos em *Machine Description*, seleciona código que casa cada uma das árvores, e gera um código *assembly* preliminar (XAR). A representação XAR mantém o programa numa sintaxe muito próxima à da linguagem *assembly* destino, mas ainda em estruturas de dados na memória, o que permite alocação de registradores e otimizações de código dependentes de máquina.

O programa mantido em XAR é entrada para um gerador de código alvo (*Assembly Code Generator*), que produz um arquivo *assembly*. Finalmente, este arquivo *assembly* pode ser traduzido em um código executável, por meio de um montador.

Para efeitos de validação do compilador, são comparados o código C e o código *assembly* gerados. Dado que o código C gerado esteja funcionando corretamente, o código *assembly* deve produzir o mesmo resultado.

[1] quádruplas serão vistas com mais detalhes no capítulo 2.
[2] ASTs serão vistas com mais detalhes no capítulo 2.

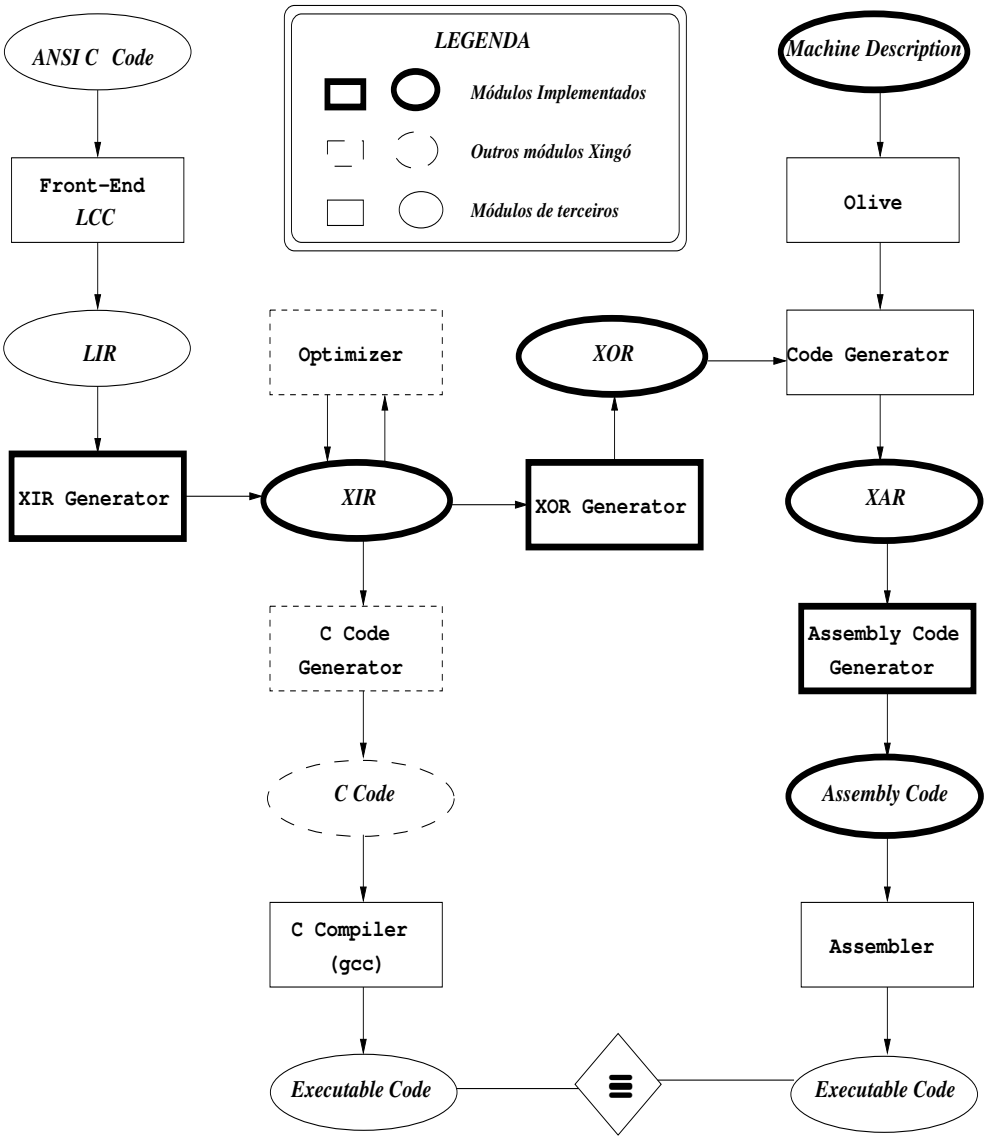

Figura 1.1: O diagrama do compilador Xingó

1.2 Organização do Texto

O capítulo 2 apresenta um conjunto de trabalhos relacionados ao tema desta dissertação. Estes trabalhos estão focados em representações intermediárias e geração de código. Dentre as abordagens que são descritas nestes trabalhos, destacam-se as técnicas, necessidades, infra-estruturas e linguagens intermediárias e de descrição de máquina utilizadas para construir um compilador otimizante e redirecionável.

No capítulo 3 é descrita a Representação Intermediária do Xingó. Esta representação é independente de máquina e usa um código baseado em quádruplas para representar internamente as operações de um programa. A Representação Intermediária Xingó é o repositório principal das informações internas referentes ao programa que está sendo compilado pelo Xingó.

O capítulo 4 aborda a geração de código para a Representação Intermediária do Xingó. Este capítulo mostra como é feita a interface entre o LCC e o Xingó, já que o LCC é quem implementa o *front-end* do compilador Xingó. A principal função do Gerador de Código Intermediário do Xingó é converter a Representação Intermediária do LCC, baseada em DAGs, para a Representação Intermediária do Xingó, baseada em quádruplas.

No capítulo 5 está descrita a Infra-Estrutura de Geração de Código do Xingó. Esta infra-estrutura inclui várias facilidades no desenvolvimento de novos geradores de código, principalmente porque ela integra o Olive, um gerador de geradores de código. Além disso, muitos dos detalhes específicos da máquina alvo são definidos através de parâmetros configuráveis. Essa infra-estrutura também permite o desenvolvimento mais rápido de um novo gerador de código pelo fato das representações internas dependentes de máquina poderem ser reutilizadas para diversas visões de arquiteturas alvo.

O capítulo 6 auxilia, através de um estudo de caso — Geração de Código MIPS R3000, o entendimento de como se deve utilizar a Infra-Estrutura de Geração de Código para construir um novo gerador de código no compilador Xingó.

Finalmente, no capítulo 7, conclui-se o trabalho, apresentando os resultados experimentais relativos à corretude do Código Intermediário Xingó e também sobre a síntese automática de um gerador de código. Também são sugeridos alguns trabalhos futuros que podem ser desenvolvidos no compilador Xingó.

Capítulo 2

Trabalhos Relacionados

Compiladores, normalmente, realizam várias passagens sobre o código do programa em compilação, no intuito de melhorar a qualidade do código gerado. Compiladores com essa característica são denominados compiladores otimizantes. Para o programador, o programa é um arquivo na forma de um texto, sintaticamente representado por uma determinada linguagem de programação. Internamente, para possibilitar múltiplas passagens sobre o código, o compilador utiliza uma linguagem própria para representar o programa. Essa linguagem é denominada linguagem intermediária, e o código por ela expressado é conhecido como representação intermediária (IR - do inglês, *Intermediate Representation*). A representação intermediária, como será vista neste capítulo, é definida por estruturas de dados bem conhecidas, tais como listas, árvores, pilhas e outras.

Como desfecho da compilação correta de um programa, deve ser gerado um código alvo a partir da representação intermediária. O código alvo, geralmente, é um programa executável, um código em linguagem *assembly*, ou, ainda, um programa em uma determinada linguagem de programação.

Compiladores, em geral, são projetados para gerar código para mais do que uma máquina alvo, daí surge a necessidade de criar mecanismos que facilitem o processo de geração de código. Compiladores que exigem o mínimo de adaptações para gerar código para uma nova máquina alvo têm sido denominados compiladores redirecionáveis.

Neste capítulo, serão analisados alguns trabalhos relacionados ao tema desta dissertação, com enfoque especial em representações intermediárias e geração de código. Estes trabalhos apresentam técnicas, necessidades, infra-estruturas e linguagens inter-

mediárias e de descrição de máquina utilizadas para construir um compilador otimizante e redirecionável.

2.1 Estrutura de um Compilador Otimizante

Compiladores que realizam transformações no código de entrada para tentar melhorar a qualidade do código gerado são conhecidos como compiladores otimizantes. Para realizar tais transformações, compiladores fazem uso de otimizações, tais como *dead code elimination, copy propagation e common subexpressions elimination* etc [2].

Na maioria dos casos, otimizar significa fazer o programa gerado ser capaz de executar com maior velocidade. Em outros casos, otimizar pode significar reduzir o espaço de memória requerido, a dissipação de energia, dentre outros.

Independente do objetivo a ser alcançado pela otimização, uma representação intermediária para o programa em compilação se faz necessária, uma vez que o processo de otimização requer múltiplas passagens sobre o código do programa [10].

A figura 2.1 ilustra a estrutura de um compilador otimizante. O *front-end* faz análise léxica, sintática e semântica sobre o código de entrada e traduz para uma representação intermediária, ainda sem nenhum tipo de otimização. Para gerar uma IR otimizada, um primeiro otimizador (*OPTIMIZER*) realiza múltiplas passagens sobre a IR não otimizada, utilizando técnicas de otimização independentes da máquina alvo [2]. O gerador de código produz, a partir da IR otimizada, uma representação do código alvo, ainda interna ao compilador. O objetivo de gerar uma representação para o código alvo é o fato de poder ser incluido um módulo otimizador (*OPTIMIZER**) que realiza otimizações dependentes de máquina, podendo melhor capturar as especificidades da máquina alvo.

As principais técnicas de otimização independente de máquina tentam melhorar o desempenho[1] do programa gerado, enquanto as otimizações de código dependente de máquina são especializadas para capturar aspectos muito particulares de cada arquitetura, por isso nem sempre o objeto é exclusivamente desempenho do código, mas pode ser, por exemplo, redução do tamanho do código alvo gerado ou menor dissipação de energia.

As otimizações de código podem ser, adicionalmente, classificadas como sendo locais, globais, ou interprocedurais [2][10]. Otimização local realiza análise dentro de um único

[1] execução mais rápida.

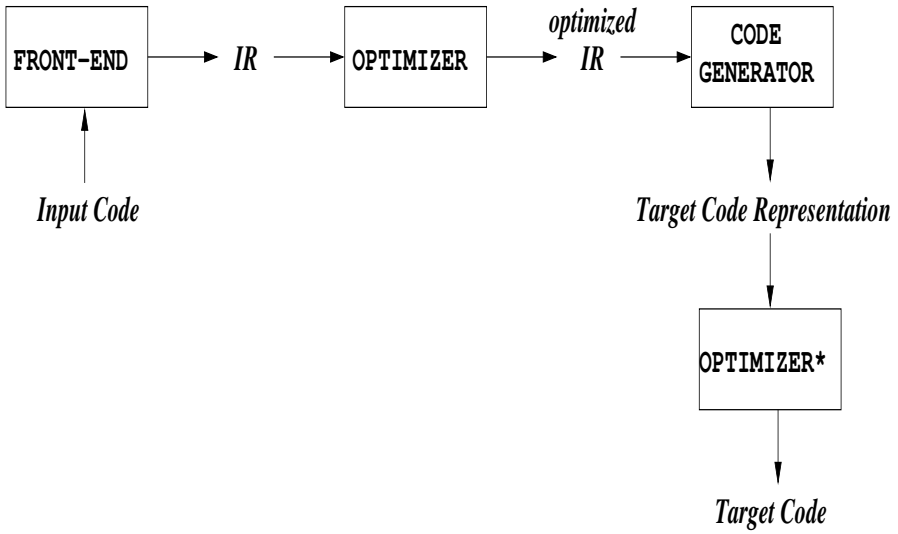

Figura 2.1: Estrutura de um compilador otimizante

bloco básico. Um bloco básico é uma seqüência de código que tem um único ponto de entrada e também um único ponto de saída, ou seja, não existe nenhuma instrução que possa desviar o fluxo seqüencial dentro do bloco. Otimização global realiza análise sobre um conjunto de blocos básicos, de modo que a seqüência de instruções de um, afeta na análise da seqüência do outro. Otimização interprocedural faz análise considerando todos os procedimentos do programa, ou seja, todos os blocos básicos.

2.2 Representação Intermediária

Implementar um compilador otimizante (compilador multi-passos), que realiza várias passagens sobre o código do programa, requer uma representação intermediária (IR) do código que está sendo compilado. As informações geradas em um passo de otimização são usadas para tomar decisões em passos posteriores [10]. A representação intermediária deve ser capaz de registrar todas as informações utilizadas entre as fases do compilador. Quando se fala em representação intermediária, não se deve referir somente às operações (instruções) de um programa, mas também às informações simbólicas (tais como variáveis e tipos) que devem ser mantidas em tabelas de símbolos.

Uma representação intermediária é um tipo de linguagem de máquina abstrata que

pode expressar as operações da máquina alvo sem os detalhes específicos dessa máquina [2]. Uma representação intermediária como esta é chamada de representação intermediária independente de máquina. Segundo Aho et al [2], uma representação intermediária independente de máquina é ponto-chave para:

- *Facilidade de redirecionamento.* Um compilador para uma máquina diferente pode ser criado acoplando um gerador de código para essa nova máquina ao *front-end* já existente.

- *Otimizações independentes de máquina.* A representação intermediária não agrega nenhum detalhe específico da máquina alvo, por isso as técnicas de otimizações independentes de máquina podem ser aplicadas.

Embora seja possível traduzir diretamente código fonte em código de máquina real, esta não é uma boa abordagem no projeto de um compilador [6]. Por exemplo, se for desejado gerar código para n diferentes linguagens fonte, direcionado para m máquinas diferentes, isto deve levar à implementação de nxm "versões" do compilador. Além da difícil tarefa de implementação, tal abordagem restringe a portabilidade e a modularidade. Em um modelo mais modular de compilador, o *front-end* faz análise léxica, sintática e semântica sobre um programa na linguagem fonte, e traduz para uma representação intermediária. O *back-end* faz otimizações sobre a IR e traduz para um programa na linguagem de máquina [2].

Escolher um modelo de representação intermediária para o projeto de um compilador requer um entendimento de como transformar o programa da linguagem fonte para um programa na linguagem intermediária. Também é necessário entender as propriedades dos programas que serão compilados, e os pontos fortes e fracos da linguagem na qual o compilador será implementado [6]. No projeto de uma IR, é preciso considerar as operações a serem realizadas e seus custos, além das otimizações de código visadas pelo compilador.

Grande parte do esforço no projeto e implementação de uma representação intermediária recai sobre a forma estrutural de representar as operações de um programa. Internamente, depois que o *front-end* faz a análise do programa de entrada, e antes que o *back-end* faça a tradução, as operações do programa estão descritas numa linguagem de máquina abstrata.

Apesar de cada projeto de compilador ter suas especificidades, existem três tipos principais de organização estrutural das operações na representação intermediária [10]:

- *Representação em grafos.* Os algoritmos são expressados em termos de nodos e arestas, em termos de listas e árvores. Exemplos incluem árvores sintáticas abstratas (ASTs) e grafos de fluxo de controle (CFGs).
- *Representação linear.* As operações são mapeadas para uma máquina abstrata e os algoritmos trabalham sobre uma simples seqüência linear de operações. Exemplos incluem *bytecodes* e código de três endereços.
- *Representação híbrida.* Combina elementos de representação em grafos e linear. Uma IR híbrida comum usa um código linear de baixo nível para representar blocos e um grafo para representar o fluxo de controle entre estes blocos.

A escolha da organização estrutural das operações de uma representação intermediária tem um impacto forte em como o projetista de compiladores pensa sobre a análise, transformação e tradução do código. Essa escolha implica diretamente no nível de abstração usado para representar as operações. Aspectos como o custo de gerar e manipular a IR, e o espaço de memória por ela requerido, têm efeito direto na velocidade do compilador, sendo, portanto, também relevantes para o projetista de compiladores.

2.2.1 Representação em Grafo

Muitas representações intermediárias organizam as operações do programa como sendo traduzidas através de um grafo. A estrutura dessas IRs é muito parecida, a diferença está nas restrições colocadas na forma do grafo [10]. A seguir, são apresentadas as estruturas mais utilizadas para representar uma IR em forma de um grafo.

Árvore Sintática

Dada uma gramática que define um conjunto de operações de uma linguagem, uma árvore sintática [2][10][36] é uma representação em grafo para a derivação das operações de um programa de entrada, segundo esta gramática.

A gramática mostrada na figura 2.2 define as operações binárias $+$, $-$, $\times$ e $\div$, sobre os terminais `num` e `id`. A figura 2.3 ilustra a árvore sintática resultante da análise sintática

da expressão $a \times 2 + a \times 2 \times b$ sobre a gramática mostrada na figura 2.2. Esta árvore representa a derivação completa, com um nodo para cada símbolo da gramática (terminal e não-terminal).

$$
\begin{aligned}
&s \rightarrow e \\
&e \rightarrow e + t \mid e - t \mid t \\
&t \rightarrow t \times f \mid t \div f \mid f \\
&f \rightarrow \texttt{num} \mid \texttt{id}
\end{aligned}
$$

Figura 2.2: gramática para operações matemáticas simples

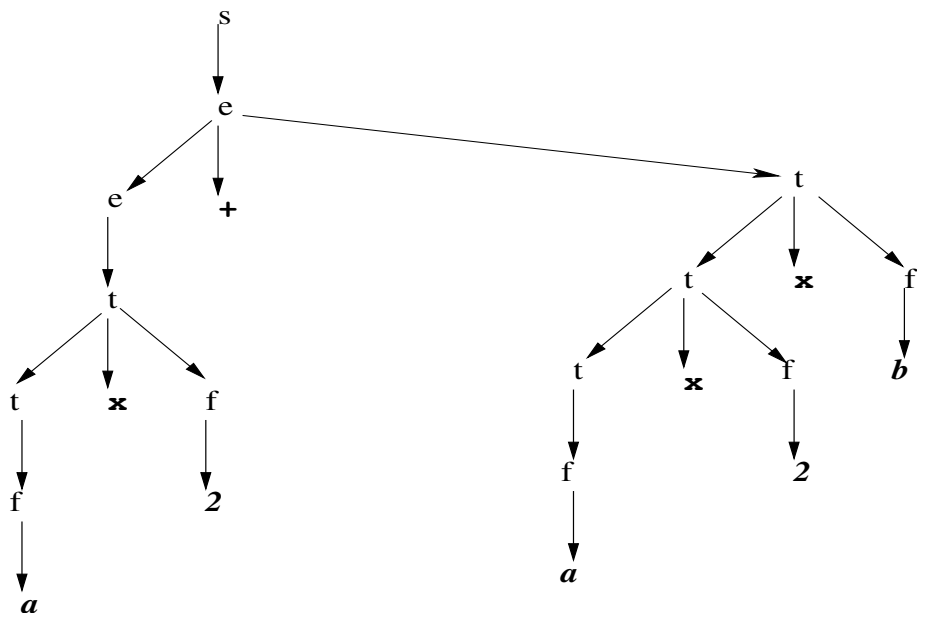

Figura 2.3: Árvore sintática para a expressão $a \times 2 + a \times 2 \times b$

Árvore Sintática Abstrata

Uma árvore sintática abstrata (AST) [2][10][36] apresenta a mesma estrutura de uma árvore sintática, mas elimina nodos não-terminais e parênteses. Com isso, a árvore fica mais simples e sua representação ocupa menos espaço na memória. A precedência e

o significado da expressão são preservados [2]. A figura 2.4 ilustra a árvore sintática abstrata para a mesma expressão ($a \times 2 + a \times 2 \times b$).

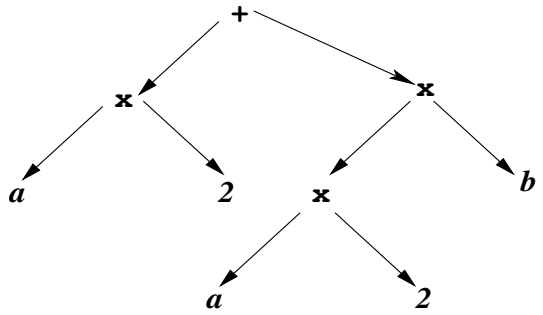

Figura 2.4: AST para a expressão $a \times 2 + a \times 2 \times b$

Grafo Direcionado Acíclico

Pelo fato de o compilador ter que atravessar diversas vezes uma árvore de expressão, é importante evitar a geração e preservação de nodos e arestas que não são realmente necessários. Um grafo direcionado acíclico [2][10][36] (DAG - do inglês, *Directed Acyclic Graph*) é uma compactação de uma AST, evitando duplicação desnecessária de nodos.

Em um DAG, nodos podem ter múltiplos pais. Portanto, as duas ocorrências de $a \times 2$ na figura 2.4 podem ser representadas por uma única subárvore, como mostrado na figura 2.5. O compilador pode gerar código que avalia a subárvore $a \times 2$ uma única vez e usar o resultado duas vezes.

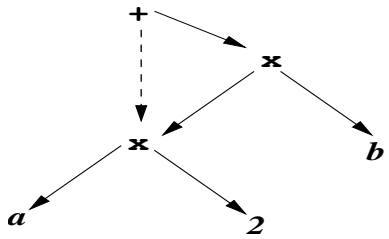

Figura 2.5: DAG para a expressão $a \times 2 + a \times 2 \times b$

Grafo de fluxo de controle

Um grafo de fluxo de controle (do inglês, *Control Flow Graph* - CFG) modela a forma como o controle de execução é transferido entre os blocos de um procedimento. O CFG tem nodos, que correspondem aos blocos básicos, e arestas, que correspondem aos possíveis fluxos de controle entre estes blocos [2][10][36].

Compiladores, normalmente, usam um CFG em conjunto com outra organização estrutural de IR. O CFG representa o relacionamento entre os blocos, enquanto as operações dentro do bloco são representadas por uma AST, um DAG, ou um código linear.

Além das representações intermediárias em grafo que foram relacionadas, outras também são muito usadas dentro de um compilador [2][10]. Grafos de precedência e *static single assignment* (SSA) são algumas delas. Porém, estas representações não serão descritas aqui, pois elas são representações intermediárias auxiliares, usadas para transformação de código, e tipicamente o compilador mantém uma outra representação, conhecida como representação intermediária principal, que é o alvo deste trabalho.

2.2.2 Representação Linear

Uma outra alternativa para a organização estrutural das operações de um programa é utilizar uma forma linear de IR.

A lógica por trás de uma forma linear é simples. Basicamente, o compilador deve emitir uma seqüência de instruções para a expressão que está sendo analisada. Normalmente, uma instrução corresponde a uma das operações da expressão em análise, sendo as instruções emitidas na ordem com que devem ser avaliadas as operações. Uma IR linear, portanto, tem a vantagem de impor uma ordenação direta da seqüência de operações [10].

Ao contrário das representações em grafo, uma forma linear não possui arestas indicando os possíveis fluxos de controle de execução. Por isso, quando uma forma linear é usada como representação intermediária, ela deve incluir um mecanismo para codificar a transferência de controle entre diferentes pontos no programa. Isto é usualmente feito com operações de desvio condicional ou incondicional.

Código de um endereço

Código de um endereço [2][10], também chamado de código de máquina baseado em pilha, assume a presença de uma pilha de operandos. Por exemplo, uma operação de subtração poderia remover do topo da pilha os dois operandos e inserir de volta no topo da pilha, o resultado dessa diferença (push(pop() - pop()).

A representação do código de um endereço é compacta. O código de máquina baseado em pilha é simples de ser gerado e executado. A natureza compacta da representação do código de um endereço é uma forma atrativa de codificar programas em ambientes onde o espaço de memória é muito limitado.

A figura 2.6 mostra a seqüência de instruções geradas pela expressão $a \times 2 + a \times 2 \times b$, utilizando código de um, de dois, e de três endereços.

`PUSH 2`
`PUSH a`
`MUL`
`PUSH b`
`MUL`
`PUSH 2`
`PUSH a`
`MUL`
`ADD`
	`LOADI t1,2`
	`LOAD t2,a`
	`MUL t1,t2`
	`LOAD t3,b`
	`MUL t3,t1`
	`ADD t3,t1`
		`LOADI t1,2`
		`LOAD t2,a`
		`MUL t3,t1,t2`
		`LOAD t4,b`
		`MUL t5,t3,t4`
		`ADD t6,t3,t5`

código de um endereço	*código de dois endereços*	*código de três endereços*

Figura 2.6: Representações lineares para a expressão $a \times 2 + a \times 2 \times b$

Código de dois endereços

Código de dois endereços [2][10] permite representar expressões da forma $x \leftarrow x\ op\ y$, com um único operador (op) e, no máximo, dois operandos (x e y).

Código de três endereços

Código de três endereços [2][10] permite representar expressões da forma $z \leftarrow x\ op\ y$ com um único operador (op) e, no máximo, três operandos (x, y e z).

O código de três endereços é atrativo por diversas razões:

- O código é compacto, se comparado com a maioria das representações em grafo.
- introduz um novo conjunto de nomes, em comparação com código de um e dois endereços. Muitas vezes, isso revela oportunidades para geração de código melhor.
- apresenta uma similaridade muito grande com o cojunto de instruções de computadores reais, especialmente microprocessadores RISC.

O código de três endereços pode ser descrito com diferentes representações, tais como quádruplas, triplas e triplas indiretas. Essas representações são descritas a seguir.

Quádruplas

Uma quádrupla [2][10] é um registro com quatro campos (um operador, dois operandos fonte e um operando destino). O conjunto de quádruplas que representa um programa pode ser mantido em um vetor kx4 de inteiros, onde k é o número de quádruplas [10]. Em alguns casos, esse vetor deve ser dinâmico, ou seja, ter um valor de k variável, já que várias transformações poderão ser realizadas sobre o código do programa. A principais vantagens de quádruplas sobre são simplicidade e facilidade de manipular.

A figura 2.7 ilustra como ficaria a representação da expressão $a \times 2 + a \times 2 \times b$ usando quádruplas. Para cada operação foi criado um temporário (t1, t2, ..., t6) para guardar o resultado computado.

LOADI	t1		2
LOAD	t2		a
MUL	t3	t1	t2
LOAD	t4		b
MUL	t5	t3	t4
ADD	t6	t3	t5

Figura 2.7: Quádruplas para a expressão $a \times 2 + a \times 2 \times b$

Triplas

Triplas [2][10] são uma forma de código de três endereços mais compacta do que quádruplas. Em representações de tripla, o índice da operação no vetor linear de instruções é usado como um nome implícito. Isto elimina 25% do espaço requerido por quádruplas. O ponto forte deste modelo é a introdução de um único e implícito espaço de nomes para valores criados por operações. Infelizmente, o espaço de nomes é diretamente associado ao número da instrução, dificultando a reordenação de instruções, muito comum em transformações no código intermediário.

A figura 2.8 ilustra como ficaria a representação da expressão $a \times 2 + a \times 2 \times b$ usando triplas. O índice da instrução no vetor é usado como um nome implícito.

(1)	LOADI	2	
(2)	LOAD	a	
(3)	MUL	(1)	(2)
(4)	LOAD	b	
(5)	MUL	(3)	(4)
(6)	ADD	(3)	(5)

Figura 2.8: Triplas para a expressão $a \times 2 + a \times 2 \times b$

Triplas Indiretas

Triplas indiretas [10] são uma solução simples para o maior problema apresentado pela representação por triplas, a dificuldade de reordenação de instruções. A figura 2.9 ilustra como ficaria a representação da expressão $a \times 2 + a \times 2 \times b$ usando triplas indiretas.

Triplas indiretas representam as operações através de tabelas de triplas, mas adicionalmente o compilador mantém uma lista de apontadores que especificam a ordem na qual as triplas ocorrem. Para reordenar as operações, o compilador simplesmente move os ponteiros da lista.

A grande vantagem do uso de triplas indiretas em relação a quádruplas é que se

a mesma instrução ocorre múltiplas vezes, ela pode ser representada somente uma vez e referenciada por mais de um ponteiro da lista. A maior desvantagem é que triplas indiretas são muito mais difíceis de serem manipuladas do que quádruplas. Se memória não for um elemento extremamente escasso, é preferível utilizar quádruplas a utilizar triplas indiretas.

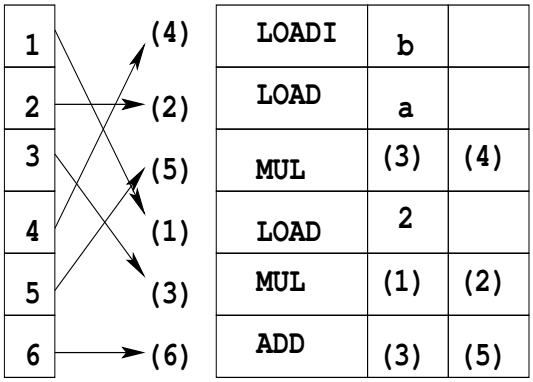

Figura 2.9: Triplas indiretas para a expressão $a \times 2 + a \times 2 \times b$

2.2.3 IR Linear *versus* IR em Grafo

A representação intermediária escolhida para ser utilizada no Xingó foi a linear. A escolha por uma IR linear foi devido as suas vantagens em relação a uma IR em grafos. Abaixo, seguem as vantagens mais relevantes:

- é mais fácil de ser gerada e manipulada;
- apresenta grande semelhança com instruções de processadores reais, especialmente RISCs;
- impõe uma ordenação direta da seqüência de operações.

Por outro lado, a IR em grafos oferece um fluxo de controle implícito nas próprias arestas, ao passo que a IR linear deve utilizar instruções explícitas para garantir esse fluxo. No entanto, tais instruções acontecem em processadores reais e são fáceis de serem introduzidas e manipuladas no código intermediário do compilador.

Dentre as formas de IR linear, o código de três endereços é o que gera menos instruções ou operações, por isso ele foi o escolhido no Xingó. Quádruplas, dentre as possíveis formas de código de três endereços, é aquela que apresenta maior facilidade de manipulação.

Pelos motivos relacionados, optou-se por uma IR linear baseada em quádruplas para implementar a representação intermediária do Xingó.

2.2.4 Tabelas de Símbolos

Um compilador usa uma tabela de símbolos [2][10][36] para guardar as informações sobre os símbolos de um programa. Os símbolos compreendem variáveis (globais, locais e temporárias), procedimentos e funções, parâmetros, *labels*, constantes, tipos, dentre outras informações simbólicas necessárias para traduzir um programa na linguagem fonte para um programa na linguagem alvo. Todo símbolo na representação intermediária fica armazenado em alguma tabela de símbolos, de modo que esta tabela é parte fundamental da representação intermediária do compilador.

Alguns símbolos precisam ter associado o tipo do dado que eles manipulam, seu local de armazenamento (registrador, por exemplo), e o endereço base e *offset* na memória. Para procedimentos e funções, o compilador precisa saber o número de parâmetros e seus tipos, e também o tipo do ítem retornado. Registros ou estruturas precisam de uma lista com seus campos, além das informações relevantes sobre cada campo.

Apesar de estarmos falando em alguns momentos "a tabela de símbolos", dentro do compilador podem existir diversas tabelas de símbolos, para as diferentes funcionalidades. Por exemplo, tabela para constantes, tabela para símbolos globais, e tabela de procedimentos [10]. As tabelas de símbolos devem ser muito bem planejadas, pois são responsáveis por grande parte da memória utilizada pelo compilador.

Implementação de uma Tabela de Símbolos

A implementação de uma tabela de símbolos requer um cuidado especial, que é a quantidade de símbolos que ela deverá ser capaz de armazenar. Símbolos são consultados, alterados, inseridos e removidos na maioria das transformações feitas no código. Estaticamente, o compilador não consegue prever a quantidade de símbolos que vai ser armazenada na tabela. A implementação de uma tabela de símbolos deve levar em consideração esses fatores.

Os símbolos numa tabela podem ser armazenados em um vetor ou em uma lista encadeada, por exemplo. No entanto, tais alternativas não parecem muito boas. Um vetor tem tamanho limitado, a quantidade de símbolos é imprevisível. Uma lista encadeada tem manutenção lenta.

Uma alternativa para implementar uma tabela de símbolos é utilizar as chamadas tabelas *hash*. Uma tabela *hash* [10] possui um conjunto limitado de entradas, chamadas de "*buckets*". Na figura 2.10 é ilustrada uma tabela *hash* com 100 entradas.

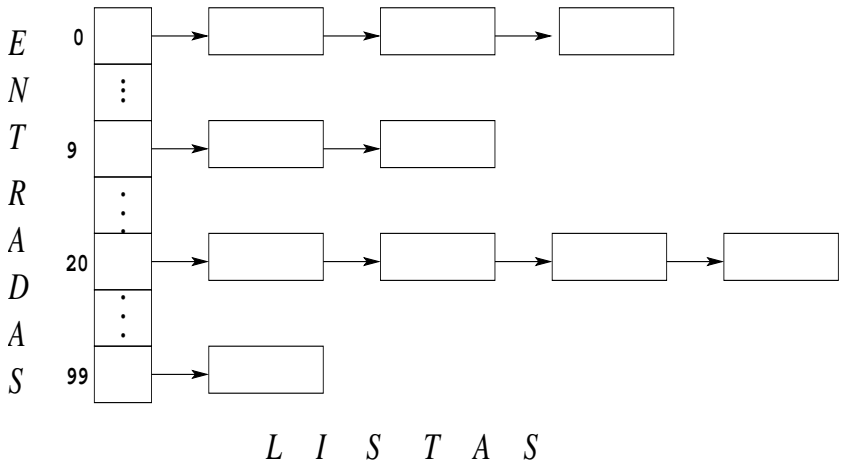

Figura 2.10: Tabela de símbolos implementada por uma tabela *hash*

Cada entrada aponta para uma lista encadeada. Cada símbolo aparece em somente uma dessas listas. Para saber em qual dessas listas o símbolo deve estar, o método usa uma função *hash*. Esta técnica tem um custo $O(1)$ para calcular a entrada onde o símbolo deve estar. As listas são usadas porque, eventualmente, dois ou mais símbolos podem cair na mesma entrada.

Encontrar o número ideal de "*buckets*" e uma função *hash* que minimize os conflitos (símbolos caírem na mesma entrada) são os pontos cruciais da implementação de uma tabela *hash*.

2.3 Geração de Código

A geração de código ocorre em diferentes fases de compilação. Em geral, pelo menos dois geradores de código existem: o gerador de código intermediário e o gerador de código alvo. Primeiro, aspectos dependentes da linguagem fonte, tais como resolução de nomes, identificação de operadores e regras de tipos são analisados e, através do gerador de código intermediário, é produzida uma representação intermediária (IR). A IR passa por diversos processos de otimização, tais como eliminação de expressões comuns e de código morto, otimização de *loops*, dentre outros [2]. A IR otimizada é então usada como entrada para o gerador de código alvo, que produz código para a máquina desejada.

Segundo Ganapathi et al [19], dentre os principais tópicos associados à geração de código, podem ser citados:

- Ordem de avaliação de operandos e expressões.
- Alocação de registradores.
- Alocação de memória.
- Seleção de instruções.
- Otimizações dependentes da máquina.

A proliferação de linguagens de programação e *front-ends* têm criado a necessidade de automatização dos geradores de código intermediário nos compiladores. Por outro lado, o surgimento em grande escala de novas arquiteturas de computadores, cada vez mais especializadas, têm criado a necessidade de automatização dos geradores de código. A automatização dos geradores de código pode ser conseguida utilizando alguma ferramenta de síntese de um gerador de código, como a ferramenta Olive [45]. Para melhor acompanhar a evolução das arquiteturas de computadores, o compilador deve ter geração de código redirecionável, descrita a seguir.

2.3.1 Geração de Código Redirecionável

Um compilador é dito ser redirecionável se ele pode ser aplicado para um amplo domínio de processadores alvo. Isto significa que o modelo do processador alvo não pode ser uma parte

implícita do compilador, mas ele deve ser explícito [34]. Compiladores redirecionáveis são compiladores que podem ser adaptados rapidamente para diferentes conjuntos de instruções alvo sem a necessidade de modificar o núcleo (código fonte) do compilador [18][28]. A independência entre as ferramentas, como o compilador e o montador, em relação à máquina alvo é essencial para se conseguir a redirecionabilidade [31].

Para separar o compilador do processador alvo, compiladores redirecionáveis baseiam-se em um modelo de máquina externo ao compilador [29]. Estes modelos de máquina, ou descrição de máquina, são necessários para formalizar as características da máquina alvo. Possíveis modelos incluem descrições de: memória, topologia, modelo funcional, recursos e conjunto de instruções [12].

Segundo G. Hadjiyiannis et al [22], uma descrição de máquina mais abrangente deve possuir as seguintes características:

- especificar uma grande variedade de arquiteturas.
- suportar restrições explícitas que definem grupos de operações válidas para uma instrução.
- ser de fácil entendimento e de rápida modificação.
- suportar geração automática de um gerador de código.
- suportar geração automática de um montador.
- suportar geração automática de um simulador em nível de instruções (ILS).
- permitir a inclusão de informações para atender às otimizações de código.

No entanto, a quantidade de detalhes requeridos por cada descrição pode tornar muito difícil a representação das informações [12]. Por isso, algumas descrições de máquina alvo, como a descrição Olive [45], fornecem suporte apenas para a geração automática de um gerador de código. Este gerador de código, assim como proposto em outros trabalhos [3][19][20], concentra-se na automatização da seleção de instruções.

Compiladores redirecionáveis buscam diminuir o impacto gerado quando o compilador precisa ser direcionado para uma nova máquina alvo. Este impacto normalmente recai sobre o tempo de desenvolvimento. A medida mais efetiva para medir a redirecionabilidade

é a quantidade de código que pode ser reusado durante o processo de redirecionamento [44]. Especificamente, uma infra-estrutura é dita ser altamente redirecionável se o desenvolvedor é capaz de fazer uso de uma quantidade significativa de código fonte que já existe no compilador.

Impulsionado pelo aumento do uso de processadores de aplicação específica (ASIPs) na implementação de sistemas dedicados, o interesse por compiladores redirecionáveis tem ganho aumento significativo ultimamente.

O projeto de microprocessadores está gradualmente incorporando arquiteturas que podem fornecer ganhos, não somente em aplicações numéricas e simbólicas, mas também na manipulação de som e imagem. Isto pode ser observado no crescente uso de processadores dedicados de áudio e vídeo e na introdução de novas instruções especializadas nos microprocessadores de uso geral, como por exemplo: Intel Multimedia Extensions (MMX), Sun Visual Instruction Set (VIS) e MIPS Digital Media eXtension (MDMX). A adição de características especializadas nas arquiteturas aumenta o seu desempenho, mas também aumenta a complexidade do projeto dos microprocessadores bem como sua posterior programação.

A solução de compromisso entre adicionar instruções especializadas no *hardware* e seu impacto na facilidade de programá-las é um dos desafios tanto para os projetistas de processadores quanto para os projetistas de compiladores. No projeto de processadores RISC [23], o objetivo maior é maximizar o desempenho de instruções que são chamadas de instruções de caso comum (*common case*). Essas instruções podem ser classificadas como sendo aquelas que mais freqüentemente aparecem em aplicações de uso geral, ou seja, aplicações que naturalmente demandam um estilo regular de arquitetura. Isso porém não é verdade para o caso de aplicações especializadas, onde arquiteturas dedicadas com suas funções específicas conduzem a um melhor desempenho. Neste caso, a regra de maximizar o desempenho das instruções *common case* também pode ser aplicada, porém, ao contrário de processadores RISC, essa abordagem resulta em arquiteturas irregulares e bastante especializadas.

O principal impacto da especialização de instruções é sentido quando o código precisa ser gerado por um compilador. Infelizmente, a capacidade dos compiladores em capturar a existência de uma dada funcionalidade especializada é muito restrita.

Uma solução para o impasse da especialização de instruções é a introdução de in-

formações sobre a arquitetura alvo dentro do compilador, porém isto afeta a flexibilidade com que a arquitetura alvo pode ser mudada. Uma outra solução é descrever o processador usando uma linguagem de descrição de *hardware* [43] e usar esta descrição para sintetizar a arquitetura e o gerador de código. Uma terceira abordagem é utilizar uma gramática para descrever o conjunto de instruções do processador alvo, e utilizar um gerador de geradores de código para sintetizar automaticamente o gerador de código para o novo processador alvo.

2.3.2 Métodos de Geração de Código Redirecionável

No trabalho de Ganapathi et al [19] classifica-se a geração de código redirecionável em três categorias:

- Geração de código interpretado.
- Geração de código *pattern-matched*.
- Geração de código *table-driven*.

O método de geração de código interpretado gera código para uma máquina virtual, e então expande para o código alvo utilizando uma linguagem que descreve o processo de geração de código. Essa linguagem pode adotar uma das duas classes de métodos de geração de código interpretado. A primeira classe implementa um mapeamento um-para-um ou um-para-muitos entre as instruções da máquina virtual e as instruções da máquina real. A segunda classe permite que o gerador de código coloque várias instruções da máquina virtual em uma única instrução da máquina real. O principal problema deste método é que existe uma interdependência muito grande entre a descrição da máquina e o gerador de código, o que acaba comprometendo a redirecionabilidade. Para gerar código para uma nova máquina o gerador de código quase sempre precisa sofrer muitas modificações.

O método de geração de código `pattern-matched` separa a descrição da máquina do seu algoritmo de geração de código. A idéia por trás deste método é introduzir um casamento de padrões para evitar interpretação. O gerador de código aceita uma árvore sintática como entrada e armazena os *tokens* (terminais das árvores, como por exemplo, identificadores e operadores) dessa árvore até que uma regra padrão adequada, definida

pelo algoritmo de geração de código, possa casar-se com alguma sub-árvore sintática. O conjunto de regras padrão codifica as características da máquina alvo. A qualidade do código gerado depende principalmente da estratégia de seleção de instruções mais apropriada e da definição do conjunto de regras. Por exemplo, se a máquina alvo permitir uma instrução de "incremento", esta pode ser utilizada ao invés de uma instrução "adição de um".

A Geração de código *table-driven* emprega uma descrição formal da máquina alvo associada a um gerador de gerador de código para produzir automaticamente o módulo de geração de código do compilador. A descrição formal da máquina alvo especifica locais de armazenamento, modos de endereçamento, conjunto de instruções, via de dados (*datapath*), etc. Tal descrição é uma entrada para o gerador de gerador de código. Este método tenta dividir o problema de geração de código em três partes principais: alocação de registradores, alocação de memória e seleção de instrução.

Dos três métodos de geração de código analisados, o *table-driven* pode ser visto como o mais avançado em questão de facilidade de geração de compiladores redirecionáveis. No Xingó, utiliza-se o método *pattern-matched*, que é o método empregado pelo Olive [45]. Como será visto no capítulo 5, o Olive é o gerador de geradores de código que faz a síntese de um novo gerador de código no compilador Xingó.

2.4 Projetos de Compiladores Redirecionáveis

2.4.1 Compilador SPAM

O compilador SPAM [41][44] é um compilador redirecionável. O SPAM foi desenvolvido em Princeton, e seu foco de compilação está voltado para *fixed-point* DSPs (*Digital Signal Processors*).

A representação intermediária (IR) do compilador SPAM é implementada pelo SUIF (*Stanford University Intermediate Format*)[21][30], que foi, inicialmente, desenvolvido para estudar questões relacionadas à paralelização de código.

A IR do SUIF é capaz de representar algumas estruturas de alto nível. Mais precisamente, SUIF mantém dois grupos de estruturas de alto nível: *loops* e *arrays*. A propagação de tais estruturas do programa-fonte para a IR, pode dar oportunidade ao *back-end* de mapeá-las para um aspecto especializado da arquitetura alvo, como, por exemplo, oti-

mização de *loop* e aritmética de auto-incremento, para gerar código de endereçamento mais eficiente para acessos a *arrays*. No entanto, segundo A. Sudarsanam [44], tais estruturas do SUIF não trazem algumas informações da máquina alvo desejáveis à compilação para DSPs. Algumas das informações que ainda faltam são modo de endereçamento *bit-reverse*, modo de endereçamento de auto incremento ou decremento e uma operação de normalização, que transforma um número inteiro em um número de ponto-flutuante.

O SPAM toma como entrada a IR que é gerada pelo SUIF a partir de um código C, e aplica uma série de otimizações independentes de máquina. Depois, o *back-end* do SPAM, denominado TWIF, realiza otimizações dependentes de máquina, aproveitando, em muitos casos, as estruturas de alto nível representadas na IR do SUIF.

O TWIF é composto de dois componentes: uma biblioteca de estruturas de dados, que encapsula as várias representações do programa fonte, e uma biblioteca de algoritmos, que realizam a geração de código e as otimizações dependentes de máquina.

Os algoritmos de geração de código do TWIF traduzem a IR já otimizada em um código *assembly* preliminar. Dentre estes algoritmos, está o gerador de geradores de código Olive [45]. Olive toma como entrada uma gramática que descreve os padrões de árvore da IR do compilador que mapeiam o conjunto de instruções da máquina alvo, e então constrói automaticamente o gerador de código que executa, em tempo linear, a seleção de instruções para as árvores de expressões. Olive também faz parte do compilador Xingó, e será visto com mais detalhes no capítulo 5.

Depois que o código *assembly* preliminar é gerado, o TWIF aplica os algoritmos que realizam otimizações dependentes de máquina, com o intuito de reduzir o tamanho do código gerado.

Essas otimizações fornecem suporte para as características especializadas da arquitetura do processador alvo. Elas são executadas após a geração do código *assembly* preliminar, ao invés de serem executadas antes, devido ao fato de que as informações sobre a arquitetura, que são necessárias nas otimizações, estarem presentes no código *assembly* e não na IR, que não traz os detalhes específicos da máquina alvo.

O Xingó está estruturado de forma semelhante ao SPAM, aproveitando, inclusive, o gerador de geradores de código Olive, que foi desenvolvido no compilador SPAM. Relativo à representação intermediária, O Xingó implementa uma IR mais simples, mas que não mantém estruturas de alto nível, como *loops* e *arrays*.

2.4.2 Infra-Estrutura Zephyr

O projeto Zephyr [5] tem como um de seus objetivos, possibilitar a construção de sistemas completos de compilação através da substituição ou modificação somente dos módulos que são importantes para o projeto em foco. Com isso, O Zephyr tenta reduzir os custos de se realizar pesquisas em sistemas de computação, no que se refere a tempo e dinheiro, encorajando novas pesquisas e aumentando a freqüência com que novos resultados são produzidos.

O projeto Zephyr usa o *front-end* SUIF [21][30], que foi incluído ao projeto para propiciar um formato comum de representação interna (representação intermediária) de programas, no qual diferentes linguagens de programação podem ser aceitas pelo compilador, e o programa de entrada traduzido para este formato comum.

Toda a infra-estrutura do Zephyr é construída em torno do *Very Portable Optimizer* (VPO) [24]. O VPO aceita otimizações de baixo nível em programas representados em nível de instruções de máquina. O Zephyr não foca especificamente em um compilador, mas ele foca em estratégias de compilação, que podem ser utilizadas para se construir diferentes compiladores.

Para construir um novo compilador, o desenvolvedor deve fornecer um *front-end* e escolher uma IR como alvo deste *front-end*. Adicionalmente, o Zephyr fornece um conjunto de ferramentas, tais como: otimizador independente de máquina; componentes para *back-ends*; e *Glue* para conectar todos os componentes (módulos) do compilador. O Zephyr ainda fornece um guia para conectar a IR ao otimizador e ao *back-end*.

O VPO é o elemento chave do Zephyr. Sem o VPO, os demais módulos conseguem gerar apenas um código muito simples, enquanto que com a inclusão do VPO, o código gerado é competitivo com o gerado pelo GCC [17], sendo que em algumas ocasiões, o código gerado pelo VPO é melhor que o gerado pelo GCC.

Para ser otimizado pelo VPO, um programa deve ser expresso como um conjunto de procedimentos, onde cada procedimento deve ser representado como um grafo de fluxo de controle (CFG), sendo os elementos individuais, denominados por Zephyr RTLs (*Register-Transfer Lists*).

Para facilitar o processo de geração de código para novas arquiteturas, os componentes dependentes de máquina são gerados através de especificações da máquina alvo, o que simplifica a redirecionabilidade do compilador Zephyr.

Jung e Paek [24] utilizaram a infra-estrutura do Zephyr para desenvolver um compilador para DSPs, e puderam demonstrar a capacidade de portabilidade desta infra-estrutura.

A idéia do Zephyr de manter os componentes dependentes de máquina externos ao compilador também foi adotada pela infra-estrutura de geração de código do Xingó. Um pouco diferente do Zephyr, o Xingó já utiliza um *front-end* predefinido, que é o LCC [14], ao passo que o Zephyr aceita outros *front-ends*.

2.4.3 Compilador LANCE

O compilador LANCE [27] é um compilador redirecionável. Ele foi desenvolvido com o objetivo de ser uma infra-estrutura de desenvolvimento de compiladores C, com facilidades para geração de código para novos processadores e pesquisas em técnicas de otimização de código.

O *front-end* utilizado pelo LANCE reconhece o C padrão ANSI. Este *front-end* recebe como entrada um programa C e constrói uma representação intermediária independente da máquina alvo. Somente o tamanho e o alinhamento dos tipos de dados presentes na linguagem C é que devem ser especificados. Essa especificação é feita por meio de um arquivo de configuração.

A IR utilizada pelo LANCE consiste em código de três endereços, isto é, pode ter até três operandos por operação (dois operandos fonte e um operando destino). Assim como na IR do Xingó[2], as construções de alto nível, como desvios condicionais, *loops* e aritmética implícita para o acesso a *arrays* não estão presentes na IR do LANCE, pois estas construções são quebradas em seqüências de operações simples.

Uma característica especial da IR utilizada pelo LANCE é que ela é mantida em uma espécie de *assembly* de baixo nível, que apresenta sintaxe próxima à da linguagem C. Desta forma, a IR pode ser convertida em um programa C e ser compilada da mesma forma que o programa fonte original. Tal característica torna a representação adotada pelo LANCE fácil de entender, e é uma forte auxiliar no processo de validação do *front-end* e de novas otimizações. A representação intermediária Xingó, como será visto no capítulo 3, apresenta uma caracterísitca muito similar à do LANCE. A IR Xingó apresenta sintaxe próxima à da linguagem C e também pode ser convertida em um programa C. Na

[2] A IR do Xingó é descrita no capítulo 3.

dissertação de mestrado de W. Attrot [7], foi construído um módulo que gera a partir da IR Xingó, um código C compilável.

O LANCE contém uma biblioteca de otimizações independentes de máquina, tais como: *constant folding, dead code elimination*, bem como otimizações em *loops*. Dependendo do nível de otimização desejado, tais otimizações podem ser chamadas de forma separada ou executadas via *shell script*.

LANCE fornece ainda uma ferramenta de visualização de grafos, que é usada para demonstrar o fluxo de controle e dos dados de um programa C, facilitando a análise deste fluxo. Essa ferramenta também auxilia o entendimento das estruturas do programa e facilita o desenvolvimento de *back-ends* para geradores de código *assembly*.

O *back-end* do compilador LANCE transforma o código de três endereços em *data-flow trees* (DFTs). Cada DFT representa um fragmento da computação do código C, e compreende argumentos, operações, posições de memória, bem como as dependências de dados existentes entre elas. O formato de uma DFT gerada pelo LANCE é compatível com geradores de geradores de código como o Iburg [15] e o Olive [45]. Desta forma, *back-ends* para arquiteturas específicas podem ser desenvolvidos de forma rápida.

2.4.4 Linguagem ISDL

ISDL (*Instruction Set Description Language*) [22] é uma linguagem que pode ser usada por um compilador redirecionável para descrever o conjunto de instruções de uma arquitetura alvo. A descrição ISDL é usada pelo *back-end* para produzir o código *assembly* e também é usada para gerar, automaticamente, o montador que transforma o código *assembly* produzido pelo compilador para um arquivo binário, que é executado por um simulador do processador alvo.

A geração automática de um montador permite que os programas *assembly* sejam testados no simulador mesmo sem nenhum compilador disponível.

Um ponto de destaque da descrição ISDL é permitir restrições explícitas, tornando mais rápida a descrição de uma nova arquitetura, principalmente arquiteturas VLIW (*Very Long Instruction Word*). Nessas arquiteturas, o número de combinações de operações válidas sobre uma instrução pode ser muito maior que o número de combinações inválidas. Com suporte à restrições explícitas, ao invés de serem listadas todas as combinações de operações válidas, podem ser especificadas algumas poucas combinações de operações

inválidas.

No Xingó, a linguagem que é utilizada para descrever o conjunto de instruções da máquina alvo é uma gramática do próprio Olive. Através dessa gramática, tem se conseguido produzir um código *assembly*, mas não, ainda, um montador para este código. Para produzir um montador, é necessário incluir informações adicionais à gramática, o que deve exigir modificações no Olive.

2.4.5 FLEXWARE

O ambiente de desenvolvimento de *software* dedicado FLEXWARE [40] tem como objetivos principais, desempenho e fácil redirecionabilidade. O ambiente FLEXWARE é usado para um amplo domínio de ASIPs (processadores para aplicações específicas), em aplicações de áudio, telecomunicações sem fio, processamento de vídeo, imagem digital, processamento em rede e outras aplicações.

O FLEXWARE é composto por quatro classes de tecnologias:

- FlexCC: compilador C de alto-desempenho.
- FlexSim: simulador do conjunto de instruções.
- FlexGdb: depurador.
- FlexPerf: permite análise de desempenho.

A geração de código redirecionável no FLEXWARE é realizada pelo CODESYN [39]. A entrada do CODESYN consiste de uma descrição em alto-nível da aplicação e de uma descrição do processador alvo. A descrição da aplicação é convertida para estruturas de dados hierárquicas conhecidas como *control-data flow graphs*, ou CDFGs. A descrição do processador alvo é dada por uma descrição do conjunto de instruções, por um grafo estrutural que especifica todos os possíveis movimentos de dados no processador e por uma classificação de todos os recursos nesse grafo. CODESYN usa essa descrição do processador para gerar código a partir da representação CDFG da aplicação.

O FLEXWARE ainda realiza otimizações, tais como eliminação de subexpressões comuns, *array-to-pointer*, propagação de constantes e *loop em hardware*.

A infra-estrutura fornecida pelo Xingó ainda se encontra no início. Assim como o FLEXWARE, futuramente o Xingó poderá ter adicionado um simulador do conjunto

de instruções da máquina alvo, que poderá ser gerado automaticamente. Similar ao FLEXWARE, o gerador de código do Xingó utiliza uma representação hieráquica baseada em árvores para gerar o código para o processador alvo. A estrutura do processador alvo não pode ser descrita, mas tão somente o seu conjunto de instruções.

2.4.6 Sistema MIMOLA

O Sistema de Síntese MIMOLA (*Machine Independent Microprogramming Language*) [32] é direcionado para a síntese de processadores DSPs, através de um ambiente que permite uma especificação comportamental em alto nível do processador.

A entrada para o Sistema MIMOLA é uma especificação que consiste de 4 elementos principais:

- programas de aplicações típicos, utilizados para determinar o comportamento do *hardware*.
- conjunto de regras que mapeia cada construção de alto nível das aplicações para uma construção estrutural equivalente.
- frequência dinâmica de execução, que serve como parâmetro na decisão de custo e benefício das partes implementadas no *hardware*.
- descrição dos recursos de *hardware* disponíveis.

A característica chave do método MIMOLA é a síntese de um processador através de uma descrição de programas que possuem estruturas típicas das aplicações que serão executadas naquele processador. Desse modo, a arquitetura final será estruturada de acordo com as características das aplicações, e não do modo contrário, quando as aplicações são estruturadas de acordo com as características do *hardware*.

O Sistema MIMOLA, além da síntese em alto nível do processador, ainda realiza *post-optimization* e geração de código redirecionável [35].

O gerador de código redirecionável mapeia algoritmos para estruturas predefinidas. Ele tenta traduzir programas para código de controle binário de uma dada estrutura de *hardware*. Se o código binário puder ser gerado pelo compilador, a estrutura juntamente com o código binário implementam o comportamento do processador. Se o código binário

não puder ser gerado pelo compilador, a estrutura de *hardware* está incorreta ou o compilador não tem informações semânticas suficientes.

Um aspecto importante do Sistema MIMOLA é que os compiladores desenvolvidos — MSSV [33] e MSSQ [37][38] — são redirecionáveis. Estes compiladores foram projetados para que nenhuma parte do gerador de código precise ser reescrita para uma nova máquina alvo considerada.

2.4.7 Sistema MARION

O Sistema MARION [8] é um sistema de geração de código redirecionável projetado especificamente para RISCs que contêm múltiplas unidades funcionais e operações multiciclos.

MARION possui três componentes principais: gerador de gerador de código, um *front-end* e um *back-end*. O gerador de código é produzido pelo gerador de geradores de código a partir de uma descrição de máquina conhecida como Maril, que descreve os estágios do *pipeline*, unidades funcionais e registradores da máquina alvo. O gerador de código realiza seleção e escalonamento de código, e faz alocação global de registradores. O *front-end* é implementado pelo LCC [14]. O *back-end* realiza casamento de padrões sobre a linguagem intermediária gerada pelo LCC, utilizando o gerador de código sintetizado a partir da descrição de máquina.

2.4.8 Compilador CBC

O Compilador CBC [12] utiliza a linguagem de descrição de máquina nML [13] para descrever o modelo estrutural e comportamental do processador. O modelo estrutural especifica todos os elementos de armazenamento na arquitetura (tais como memória, registradroes e barramentos), enquanto o modelo comportamental especifica todos os possíveis fluxos entre estes elementos de armazenamento.

O gerador de código CBC consiste de diversas fases, cada qual sendo implementada por um programa separado. O resultado de cada fase é propagado para a fase seguinte por meio de um arquivo de texto, que contém uma descrição legível da representação intermediária.

Capítulo 3

A Representação Intermediária Xingó

O Xingó é um compilador otimizante, que realiza múltiplos passos sobre o código de um programa para tentar melhorar a qualidade do código gerado. Implementar um compilador multi-passos requer uma representação intermediária (IR) do código que está sendo compilado [10]. A representação intermediária é o repositório principal das informações internas de um programa em compilação.

Neste capítulo, será apresentada a Representação Intermediária Xingó, conhecida como Xingó IR, ou simplesmente XIR. A XIR é uma representação intermediária linear, independente de máquina, que usa um código de três endereços na forma de quádruplas. A XIR é obtida convertendo-se a representação em forma de DAGs do LCC para uma representação linear em forma de quádruplas, utilizada pelo Xingó. Tal representação é muito próxima da linguagem C e, por isso, pode ser convertida para um código fonte C e compilada.

Pelo fato da XIR ser independente de máquina, ela favorece a construção de um compilador redirecionável. A independência de ferramentas como o compilador (que inclui a sua representação intermediária) em relação a máquina alvo é essencial para conseguir a redirecionabilidade [31]. A Representação Intermediária Xingó também foi projetada para ter uma ligação automatizada com os geradores de código produzidos a partir de Olive [45]. Com isso, é possível fazer redirecionabilidade de código sem muitos esforços.

A representatividade da XIR tem sido avaliada através de duas visões: geração de

código C e geração de código para o processador MIPS R3000. Além disso, outras partes do compilador Xingó (análise de fluxo de dados, otimização de código, alocação de registradores, etc) têm usado satisfatoriamente os dados representados em XIR.

Neste capítulo, o objetivo principal é descrever a estrutura e os detalhes de implementação da Representação Intermediária Xingó. No anexo A, estão descritos os métodos da interface pública de cada classe da XIR, para que desenvolvedores [1] que pretendem adicionar novas funcionalidades ao compilador possam se orientar.

3.1 Visão Geral

A Representação Intermediária Xingó é constituída de várias classes implementadas em C++, que correspondem às mais diversas estruturas de um programa, como instruções, variáveis, funções e tipos. A partir destas classes, o Xingó constrói uma estrutura que será usada durante todo o processo de compilação. A Representação Intermediária Xingó está hierarquicamente estruturada. Esta hierarquia, que está ilustrada na figura 3.1, é dividida em três níveis:

- ***Arquivo.*** Refere-se ao nível mais alto da hierarquia. Neste nível, além de apontadores para todos os procedimentos do arquivo, também é mantida a tabela de símbolos global. A tabela de símbolos global é compartilhada por todos os procedimentos do arquivo. A classe `File`, apresentada na seção 3.2, implementa este nível da hierarquia.

- ***Procedimentos.*** Refere-se ao nível intermediário da hierarquia. O nível de procedimentos tem a representação de cada função declarada no arquivo de entrada. Um procedimento também possui uma tabela de símbolos, que chamaremos de tabela de símbolos local ou tabela de símbolos de procedimento. Um procedimento é implementado pela classe `Procedure` (ver seção 3.3).

- ***Instruções.*** Refere-se ao nível mais baixo da hierarquia, aonde são encontradas as instruções de cada procedimento. A forma como as instruções são representadas internamente está descrita com detalhes na seção 3.5.

[1] Neste contexto, serão chamados de desenvolvedores aquelas pessoas com relativo conhecimento na construção de compiladores.

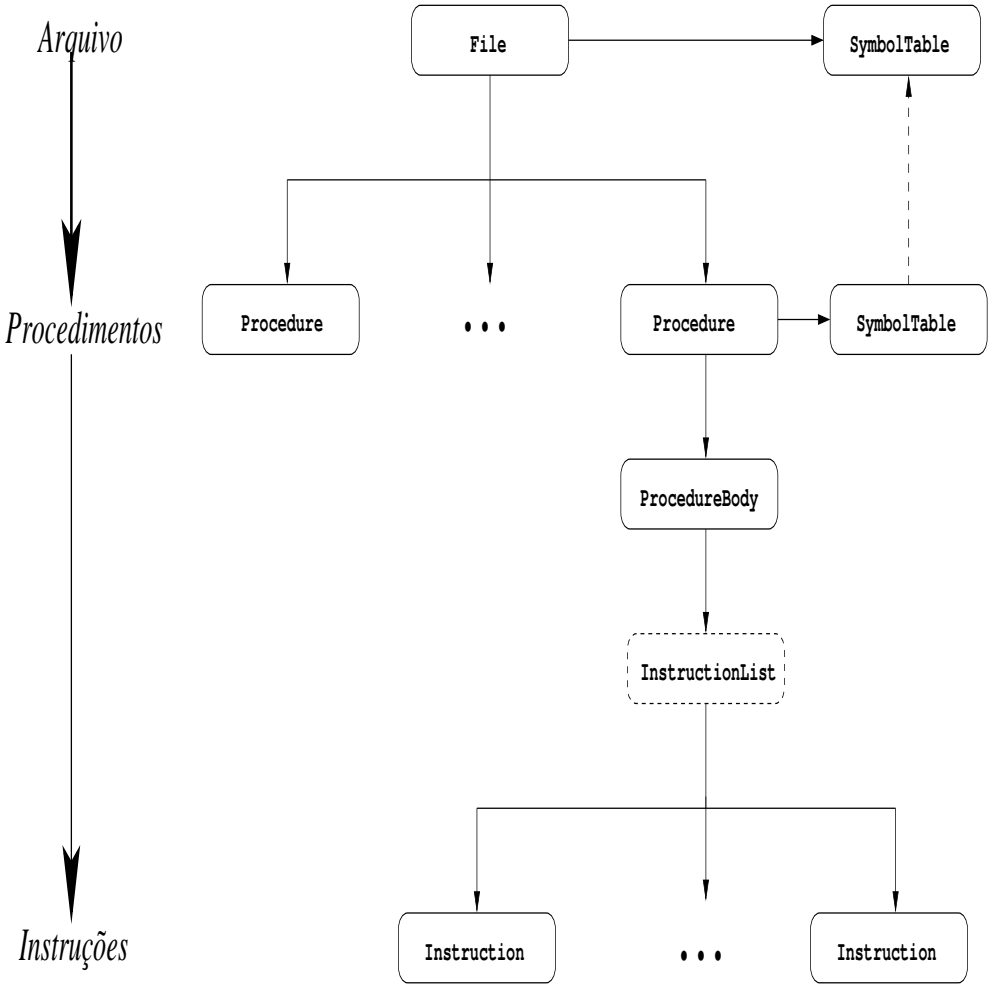

Figura 3.1: Hierarquia da Representação Intermediária Xingó

Um arquivo (classe `File`) aponta para vários procedimentos e para uma tabela de símbolos (classe `SymbolTable`), que é a tabela de símbolos global. Um procedimento (classe `Procedure`) aponta para um corpo de procedimento e para uma tabela de símbolos, que é a tabela de símbolos local. Um corpo de procedimento (classe ProcedureBody) aponta para uma lista de instruções, que representa a lista de instruções de um procedimento. Uma lista de instruções (classe `InstructionList`[2]) aponta para diversos objetos `Instruction`. `InstructionList` é uma classe gerada por uma classe que implementa

[2]InstructionList foi colocada dentro de uma caixa pontilhada, significando que ela não é uma classe definida pela XIR, mas é implementada por outra parte do compilador Xingó.

uma lista genérica duplamente encadeada. Além de listas encadeadas, o Xingó ainda implementa classes para manipular vetores e tabelas *hash* de objetos genéricos.

A seguir serão descritas as representações de cada uma das classes que compõem a Representação Intermediária Xingó. Iniciar-se-á com a representação de um arquivo, que é implementada pela classe `File`.

3.2 Representação de um Arquivo

A classe `File` foi projetada para implementar o nível superior da hierarquia da Representação Intermediária Xingó. Esta classe corresponde ao arquivo que está sendo compilado. Ela armazena diversas informações sobre o arquivo, como por exemplo o nome do arquivo, a lista de procedimentos que o arquivo implementa, a lista de procedimentos externos (que estão presentes em outros arquivos) ao arquivo e que são utilizados internamente, bem como uma tabela de símbolos que armazena todos os símbolos globais, como variáveis e tipos estruturados. A figura 3.2 ilustra a representação dos dados da classe `File`.

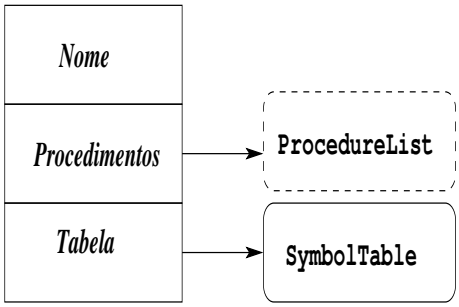

Figura 3.2: Representação dos dados da classe `File`

Existe uma instância de `Procedure` apontada por uma instância de `File`, se e somente se o procedimento é implementado no arquivo de entrada. Significa dizer que se um procedimento é implementado em outro arquivo, ainda que ele seja chamado no arquivo corrente, não vai existir uma instância de `Procedure` para ele. Contudo, se o procedimento é chamado no arquivo corrente, é possível saber o seu protótipo. O protótipo de um procedimento compreende o tipo do ítem que ele retorna, o seu nome, e os tipos

dos seus parâmetros. Também faz parte do protótipo, a informação se o procedimento tem quantidade variável de argumentos. O gerador de código C necessita conhecer os protótipos de todos os procedimentos para gerar um código compilável [7].

Para todo procedimento implementado, ou simplesmente chamado em um arquivo, existe uma variável que representa este procedimento. Esta variável, armazenada na tabela de símbolos global, contém apenas o protótipo do procedimento, mas não as suas operações. Desse modo, os procedimentos externos possuem associados a eles apenas uma variável como esta. Os procedimentos internos possuem uma variável e também um objeto `Procedure` correspondente.

Durante a compilação de um programa, existe apenas uma instância da classe `File`. Após a representação intermediária ser construída, uma variável visível a todos os módulos do compilador aponta para essa instância, que pode ser usada para acessar a tabela de símbolos global e os procedimentos do programa.

3.3 Representação de um Procedimento

O nível de procedimentos, ilustrado na figura 3.1, compreende a lista de procedimentos implementados no arquivo em compilação. Cada procedimento presente no arquivo tem associado a si um objeto `Procedure`. Tal objeto contém informações como o número identificador e o nome do procedimento, o seu tipo de retorno, a tabela de símbolos locais, os argumentos, o corpo do procedimento (ver seção 3.4) e o arquivo pai. A representação dos dados da classe `Procedure` está ilustrada na figura 3.3.

Um procedimento possui um identificador único, que é um número inteiro que o distingüe dos demais procedimentos (não existem dois procedimentos com identificadores iguais). Este identificador é gerado automaticamente, na criação do objeto `Procedure`. Uma vez criado o objeto, o seu identificador não pode mais ser alterado.

Todo procedimento possui um nome, que é o mesmo nome a ele atribuído pelo programador no arquivo de entrada que está sendo compilado pelo Xingó.

Um procedimento aponta para uma instância da classe `Type` que representa o tipo do ítem retornado por ele. Este objeto `Type` tem sempre o operador _FUNCTION (ver seção 3.8), que serve para indicar que o tipo se refere ao tipo de retorno de um procedimento.

Um procedimento possui uma tabela de símbolos locais. Essa tabela contém variáveis

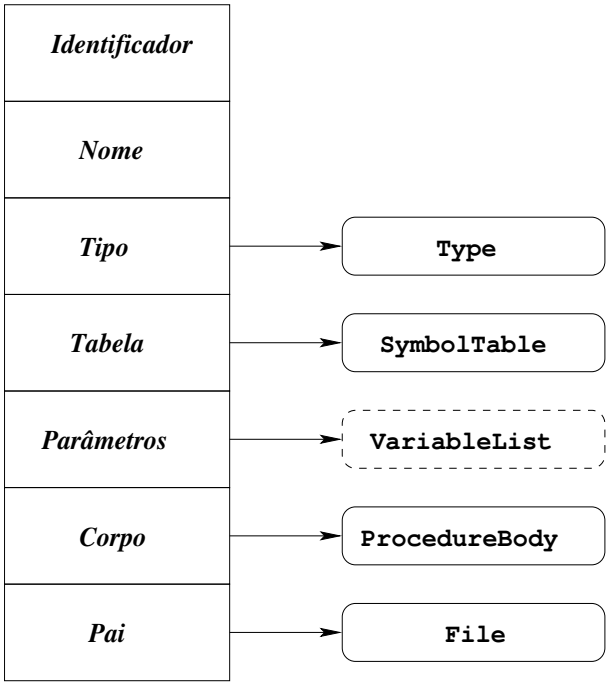

Figura 3.3: Representação dos dados da classe `Procedure`

locais definidas no procedimento, variáveis temporárias geradas pelo LCC ou pelo Xingó, e *labels*, alvos de desvio.

Um parâmetro é representado por uma variável. Os parâmetros declarados no procedimento são armazenados em uma lista de variáveis, mantida por uma instância de `VariableList`. Na lista, os parâmetros aparecem na mesma ordem em que foram declarados no procedimento correspondente.

O corpo de um procedimento é implementado pela classe `ProcedureBody`, e será melhor descrito na próxima seção.

Em certas situações, dentro de um procedimento existe a necessidade de buscar algumas informações que não estão presentes no escopo do mesmo, como por exemplo, uma variável global. Desse modo, um apontador para o objeto `File` que representa o arquivo corrente, chamado de arquivo pai, foi colocado como parte da representação de um procedimento na Representação Intermediária Xingó.

3.4 Representação do Corpo do Procedimento

Procedimentos possuem parâmetros e variáveis locais, têm um tipo de retorno, e também possuem um corpo. O corpo de um procedimento é a parte do procedimento que implementa as suas operações, isto é, as suas instruções. Por exemplo, a função `soma`, abaixo, que faz a adição de dois números inteiros e retorna o valor calculado, tem os parâmetros `x` e `y`, uma variável local `z`, e tem um tipo de retorno igual a `int`. A grosso modo, o corpo desse procedimento expressa as seguintes operações: somar `x` e `y`, atribuir o resultado a `z`, e retornar o valor de `z`.

```
int soma(int x, int y) { int z; z = x+y; return z; }
```

O corpo de um procedimento deve expressar todas as operações que o procedimento realiza.

A classe `ProcedureBody` implementa um corpo de procedimento na Representação Intermediária Xingó. Esta classe armazena a lista de instruções proveniente da conversão da representação em forma de DAGs do LCC para as instruções da XIR, um apontador para o procedimento que contém estas instruções e uma possível referência para um grafo de fluxo de controle. A Figura 3.4 mostra a representação dos dados da classe `ProcedureBody`.

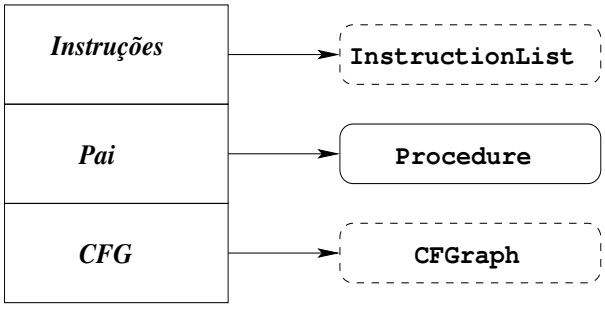

Figura 3.4: Representação dos dados da classe `ProcedureBody`

3.5 Representação de uma Instrução

Como foi visto no começo deste capítulo, cada instrução em XIR constitui uma quádrupla, onde existe um único operador (*opcode*) e, no máximo, três operandos (*destino*, *fonte1* e *fonte2*). Uma instrução em XIR admite expressões da forma:

$$destino \leftarrow fonte1\ opcode\ fonte2$$

O *opcode* corresponde ao tipo de operação que a instrução realiza, como por exemplo uma soma ou uma multiplicação. Os operandos correspondem aos símbolos que são manipulados pela instrução. Cada um dos operandos corresponde a um identificador de uma variável, de um *label*, de um tipo (somente em instruções de conversão de tipos), ou a um valor constante.

Os *opcodes* definidos na representação XIR se parecem muito com aqueles encontrados no conjunto de instruções de computadores reais, especialmente microprocessadores RISC. Foram definidos 52 *opcodes*, que são descritos nas tabelas[3] 3.1 à 3.8. Os *opcodes* definem operações de acesso a memória, atribuição, conversão de tipos, operações aritméticas, operações binárias, rotação de *bits*, chamadas a procedimento, e operações de desvio.

Uma instrução em XIR é implementada pela classe `Instruction`, que engloba todas as informações necessárias para a correta interpretação da instrução, bem como outras informações do projetista. A figura 3.5 ilustra a representação dos dados da classe `Instruction`. Nesta classe, o *opcode* é representado por um identificador numérico e os operandos (destino, fonte1 e fonte2) são armazenados em um vetor de inteiros de três posições. Ao invés de armazenar o próprio objeto que corresponde ao operando, cada posição do vetor armazena o identificador numérico do operando. Cada posição é usada por um dos operandos. No caso geral, operando destino fica na posição de índice zero do vetor, fonte1 na de índice 1, e fonte2 na posição de índice 2.

Caso a instrução manipule alguma constante, o valor dessa constante poderá ser armazenado diretamente no vetor. Os valores de constantes armazenados diretamente são aqueles do tipo `char`, `short`, ou `int`. Não são armazenados valores constantes de outros tipos, pois estes podem não caber em um ítem de dado do tipo `int`, que é tipo do vetor

[3]Para maior legibilidade do texto, as tabelas com os opcodes das instruções foram colocadas no final da seção.

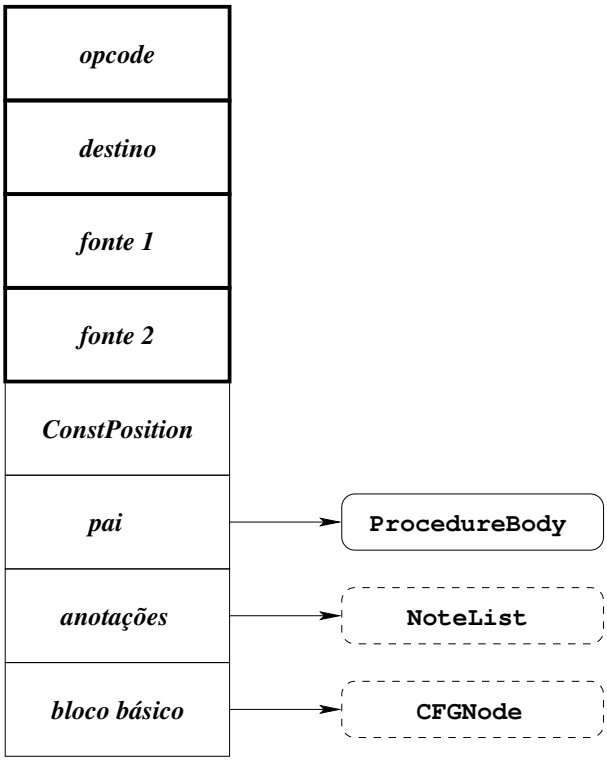

Figura 3.5: Representação dos dados da classe `Instruction`

de operandos. Para instruções que envolvem constantes, o campo *ConstPosition* é utilizado para saber se a constante está armazenada na posição destinada ao primeiro ou ao segundo operando fonte. Isso é necessário para atender operações não-comutativas, onde a ordem dos operandos altera o resultado da expressão. Para instruções comutativas que envolvam constantes, como por exemplo uma soma ou uma multiplicação, a constante sempre será armazenada na posição do segundo operando fonte. De uma maneira geral, se uma instrução qualquer fizer uso de alguma constante, sempre que possível, essa constante será armazenada na posição do segundo operando fonte. Nunca os dois operandos fontes de uma instrução serão simultaneamente uma constante, pois tal tipo de expressão é avaliada pelo compilador e transformada em uma única constante.

Em instruções que utilizam menos de três operandos, como uma atribuição (que utiliza dois), os campos não utilizados irão conter o valor zero, o qual não é utilizado como identificador por nenhum objeto.

Além de armazenar uma quádrupla, uma instrução aponta de volta para o corpo de procedimento (`ProcedureBody`) ao qual ela pertence. Uma instrução também possui anotações (ver seção 3.10). As anotações ficam armazenadas em uma lista de anotações, representada pela classe `NoteList`. Também é mantido dentro de um objeto `Instruction`, um apontador para o bloco básico ao qual a instrução pertence.

`LOAD`	*leitura da memória*
`STORE`	*escrita na memória*
`STOREI`	*escrita de um imediato na memória*

Tabela 3.1: Opcodes para operações de acesso a memória

`MOVE`	*atribuição*
`MOVEI`	*atribuição de um imediato*
`MOVES`	*atribuição de n* bytes
`ADDR`	*atribuição de um endereço*

Tabela 3.2: Opcodes para operações de atribuição

`CVRT`	*conversão de tipo*

Tabela 3.3: opcode para operação de *cast*

ADD	*adição*
ADDI	*adição com imediato*
SUB	*subtração*
SUBI	*subtração com imediato*
MUL	*multiplicação*
MULI	*multiplicação com imediato*
DIV	*divisão*
DIVI	*divisão com imediato*
MOD	*resto da divisão*
MODI	*resto da divisão com imediato*

Tabela 3.4: opcodes para operações aritméticas

AND	*e binário*
ANDI	*e binário com imediato*
OR	*ou binário*
ORI	*ou binário com imediato*
XOR	*ou exclusivo*
XORI	*ou exclusivo com imediato*
NEG	*negação binária*
COM	*complemento binário*

Tabela 3.5: Opcodes para operações lógicas

LSR	*rotação lógica a direita*
LSRI	*rotação lógica a direita com imediato*
ASR	*rotação aritmética a direita*
ASRI	*rotação aritmética a direita com imediato*
LSL	*rotação lógica a esquerda*
LSLI	*rotação lógica a esquerda com imediato*
ASL	*rotação aritmética a esquerda*
ASLI	*rotação aritmética a esquerda com imediato*

Tabela 3.6: Opcodes para operações de rotação de *bits*

ARG	*argumento de um procedimento*
ARGI	*argumento imediato de um procedimento*
CALL	*chamada a um procedimento*
RET	*retorno de valor*
RETI	*retorno de valor imediato*

Tabela 3.7: Opcodes para operações de chamada a procedimento

JUMP	*desvio incondicional*
JE	*desvia se igual*
JEI	*desvia se igual com imediato*
JNE	*desvia se não igual*
JNEI	*desvia se não igual com imediato*
JLT	*desvia se menor do que*
JLTI	*desvia se menor do que com imediato*
JGT	*desvia se maior do que*
JGTI	*desvia se maior do que com imediato*
JLE	*desvia se menor ou igual*
JLEI	*desvia se menor ou igual com imediato*
JGE	*desvia se maior ou igual*
JGEI	*desvia se maior ou igual com imediato*
LABEL	*alvo de desvio*

Tabela 3.8: Opcodes para operações de desvio

3.6 Tabelas de Símbolos

As tabelas de símbolos constituem o repositório principal de todas as informações simbólicas no compilador. Todas as partes do compilador utilizam as tabelas de símbolos para inserir e recuperar as informações relativas aos símbolos (como variáveis) e aos tipos do programa em compilação.

Foram implementadas tabelas de símbolos para armazenar *labels*, variáveis e tipos. Constantes não são armazenadas diretamente na tabela de símbolos. As constantes do tipo *char*, *short* e *int* que aparecem no arquivo de entrada são transformadas em operandos imediatos de instruções. As demais constantes são armazenadas em variáveis auxiliares. Essas variáveis tem qualificador "*const static*", para que não possam ser alteradas (o dado continua sendo constante) e para que sejam visíveis apenas no escopo do arquivo que está sendo compilado, não conflitando assim com outras variáveis de outros arquivos que serão usados para gerar o código executável do programa em compilação. Na verdade, essas variáveis auxiliares são criadas para guardar valores constantes que não cabem em um ítem de dado do tipo *int*. Como visto na seção 3.5, os operandos de uma instrução são todos do tipo *int*, não podendo armazenar um valor do tipo $float$, por exemplo.

Antes de qualquer outra parte do compilador (otimizador, gerador de código, etc) modificar os dados da representação intermediária, as tabelas de símbolos têm as informações simbólicas importadas das estruturas de dados da Representação Intermediária do LCC e variáveis temporárias e auxiliares criadas pelo módulo `XIR Generator`, descrito no capítulo 4. Posteriormente, outras partes do compilador, durante as demais fases de compilação, adicionam, atualizam e removem símbolos de uma tabela, através da interface pública descrita no anexo A.

Na figura 3.6 está ilustrada a representação de uma tabela de símbolos no compilador Xingó. Uma tabela de símbolos é implementada pela classe `SymbolTable`. A tabela de símbolos armazena símbolos e tipos. Os símbolos são divididos em variáveis e *labels*. A tabela de símbolos possui também um ponteiro para a tabela pai, que representa a tabela que contém os símbolos do escopo imediatamente acima. Para uma tabela de símbolos locais, o ponteiro para a tabela pai aponta para a tabela de símbolos global, que por sua vez não aponta para nenhuma outra tabela.

Os símbolos e os tipos de uma tabela ficam armazenados em tabelas do tipo *hash*, que como dito no capítulo 2, provém um acesso mais rápido aos elementos.

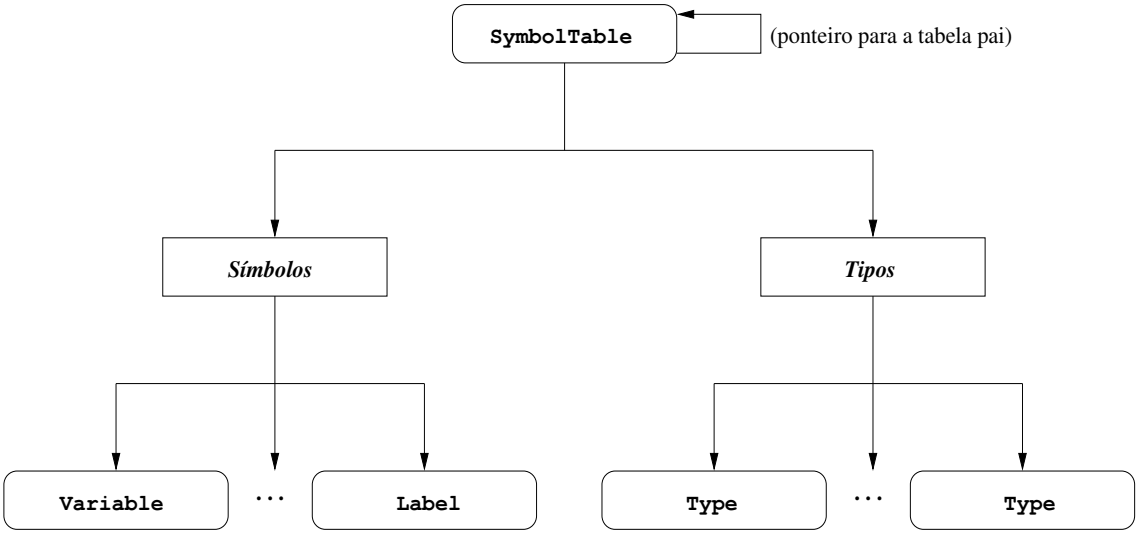

Figura 3.6: Representação da tabela de símbolos

Na Representação Intermediária Xingó, existem dois tipos de tabelas de símbolos: tabela de símbolos locais e tabela de símbolos globais.

As tabelas de símbolos locais estão associadas aos procedimentos do arquivo. Uma tabela de símbolos locais armazena *labels*, variáveis locais e variáveis temporárias. A tabela de símbolos globais[4] está associada ao arquivo em compilação. Ela armazena tipos, variáveis globais, variáveis auxiliares geradas para armazenar constantes, além de variáveis que indicam definição de procedimento (ver seção 3.7.1).

Labels só aparecem numa tabela de símbolos locais, e tipos, por decisão de projeto, são todos colocados na tabela de símbolos globais.

3.7 Símbolos

Um símbolo representa uma variável ou um *label*. Um símbolo é representado pela classe `Symbol`, que deriva as classes `Variable` e `Label`.

Na figura 3.7, está ilustrada a representação dos dados da classe `Symbol`. Um símbolo possui um identificador único, que é um código inteiro que o distingue dos demais símbolos. O identificador não é único apenas em relação aos outros símbolos da mesma tabela, ele

[4]Note que existe apenas uma tabela de símbolos globais na Representação Intermediária Xingó.

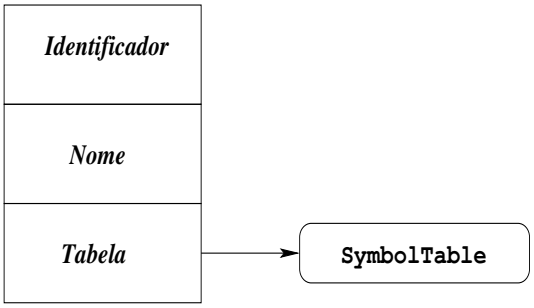

Figura 3.7: Representação dos dados da classe `Symbol`

é único em relação a todos os símbolos da representação intermediária.

Todo símbolo tem um nome. Para símbolos temporários gerados pelo módulo *XIR Generator* (ver capítulo 4), seu nome é uma *string* com prefixo ".t" seguida de uma seqüência de dígitos. O nome do temporário gerado é único apenas para a tabela de símbolos a que ele pertence, ou seja, podem existir dois temporários com o nome ".t125", mas eles estarão em tabelas de símbolos diferentes. O LCC também gera símbolos temporários. O nome de um símbolo temporário gerado pelo LCC é uma seqüência de dígitos, que, durante a geração da Representação Intermediária Xingó, é prefixada com a *substring* ".l". Por exemplo, se um temporário gerado pelo LCC tem nome igual a 21, ele passa a ter o nome ".l21" na Representação Intermediária Xingó. Para símbolos que representam um *label*, o nome é sempre uma *string* que começa com ".L", seguida por uma seqüência de dígitos. Os demais símbolos possuem o mesmo nome que foi dado a ele no arquivo de entrada.

Para todo símbolo, também existe um campo que aponta para a tabela de símbolos a qual ele pertence.

3.7.1 Variáveis

Na XIR, uma variável também é representada por uma classe, a classe `Variable`. Esta classe é derivada da classe `Symbol`. A classe `Variable` contém todas as informações importantes relativas a variável, como o seu nome (herdado da classe `Symbol`), o segmento a que ela pertence e o tipo da mesma. Na figura 3.8 está ilustrada a representação dos dados na classe `Variable`.

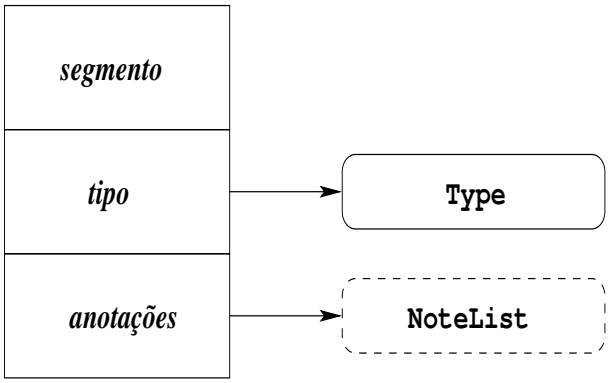

Figura 3.8: Representação dos dados da classe `Variable`

Uma variável deve pertencer a pelo menos um dos quatro segmentos que são definidos na XIR. Estes segmentos são:

- D_LIT: segmento de constantes.
- D_CODE: segmento de código. Se alguma variável pertencer a este segmento, ela provavelmente será uma variável local.
- D_DATA: segmento de variáveis globais que estão inicializadas.
- D_BSS: segmento de variáveis globais que não estão inicializadas.

Também é armazenado na classe `Variable`, um ponteiro para o tipo da variável, que pode ser qualquer um dos tipos descritos na seção 3.8.

As anotações de uma variável, tipicamente contêm informações sobre a atribuição de valores iniciais.

Variáveis podem ser globais, locais, parâmetros, temporários, ou definições de procedimento. Uma variável é global se ela é visível em qualquer procedimento da XIR. Ela é local se o seu escopo é apenas dentro do procedimento no qual ela foi declarada. Parâmetros também só são visíveis no seu procedimento. Temporários são variáveis geradas durante a geração da Representação Intermediária Xingó (ver capítulo 4). Todo procedimento implementado, ou simplesmente chamado no arquivo de entrada, tem uma

variável a ele associada. Essa variável é denominada *variável de definição de procedimento*. Uma variável de definição de procedimento serve para informar o protótipo do procedimento.

3.7.2 Labels

Labels são objetos alvo de um desvio. Eles são usadas para garantir o fluxo de controle entre as instruções de um procedimento. A classe `Label` é derivada da classe `Symbol`, e não adiciona nenhum dado em relação à classe herdada.

3.8 Tipos

Tipos de itens em C são usualmente escritos da seguinte forma: um operador para o tipo seguido pelo seu operando. Por exemplo, a declaração `int *p` declara `p` como sendo do tipo ponteiro para `int`, onde `*`, que significa ponteiro para, é o operador, e `int` é o operando. Similarmente, `char *(*strings)[10]` declara `strings` como um ponteiro para um vetor de 10 ponteiros para `char`.

O operador de um tipo em XIR, que também pode ser denominando de tipo primário, é um código inteiro dado por um dos seguintes elementos da tabela 3.9.

_CHAR	_POINTER	_ENUM	_FUNCTION
_SHORT	_LONG	_STRUCT	_CONST
_INT	_FLOAT	_UNION	_VOLATILE
_UNSIGNED	_DOUBLE	_ARRAY	_VOID

Tabela 3.9: Operadores de um tipo

Os operadores `_CHAR, _SHORT, _INT, _UNSIGNED, _ENUM e _LONG` definem tipos inteiros. `_FLOAT e _DOUBLE` são operadores para tipos em ponto flutuante.

Um tipo enumerado, operador `_ENUM`, é compatível com um tipo inteiro. Para o LCC, um item de dado de um tipo enumerado é tratado como sendo do tipo `int`. Como os valores de um tipo enumerado são constantes inteiras, eles serão operandos imediatos de instruções.

Os operadores _STRUCT e _UNION identificam tipos agregados. O operador _ARRAY identifica um tipo que é um vetor.

Os operadores _POINTER e _FUNCTION definem, respectivamente, um tipo ponteiro e um tipo para função. O operador _VOID identifica o tipo void.

_CONST e _VOLATILE são operadores que qualificam os operadores citados acima, exceto o _VOID. Eles dizem se um ítem de dado é constante ou volátil.

A classe Type foi implementada para armazenar as diversas informações sobre um tipo. Na figura 3.9, está ilustrada a representação dos dados da classe Type.

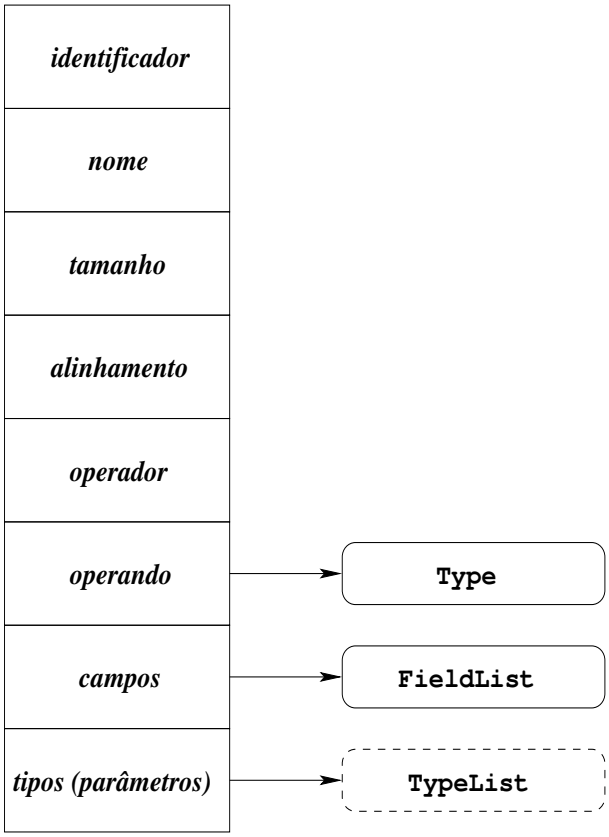

Figura 3.9: Representação dos dados da classe Type

Um tipo, assim como os demais objetos em XIR, possui um identificador, que o distingue dos demais tipos.

Tipos possuem um nome. Se o operador do tipo é _CHAR, _SHORT, _INT, _UNSIGNED, _LONG, _FLOAT, _DOUBLE, ou _VOID, o nome do tipo é igual ao do operador. Se o operador

do tipo é * (ponteiro para), a ele é dado o nome "pointer". O nome de um tipo para uma função é "function" e para um vetor é "array". Tipos que representam uma estrutura agregada (*struct* ou *union*) *têm* o nome igual *à tag* da estrutura. Por exemplo, na definição

```
struct circle { int x, y, raio;};
```

a *tag* é "*circle*".

Os tipos cujo operador é um qualificador `_const` ou `_volatile`, recebem o mesmo nome do operador.

Tipos possuem tamanho e alinhamento. O tamanho é a quantidade em *bytes* ocupada por um objeto daquele tipo, e o alinhamento diz como os dados referentes a este objeto estão alinhados na memória. O tamanho é sempre um múltiplo do alinhamento.

Tipos compostos, como por exemplo um ponteiro para inteiros, são formados agrupando-se vários objetos `Type`, de tal forma que uma lista de tipos é construída. Isso é conseguido utilizando o campo operando (ver figura 3.9). Uma variável que seja do tipo ponteiro para inteiros, terá o seu tipo representado por uma lista de tipos, onde o primeiro elemento da lista é um objeto `Type` representando o tipo *pointer* e o segundo elemento da lista é um objeto `Type` representando o tipo *int*. Um *array* de inteiros também tem o seu tipo representado como uma lista de objetos `Type`, onde o primeiro representa o tipo *array* e o segundo o tipo *int*. Tal esquema se aplica a todos os tipos que se precise representar internamente no compilador Xingó.

Tipos agregados, como `struct circle`, têm que manter um ponteiro para a lista de campos. Cada campo (no caso, x, y, e raio) vai ser representado por um objeto `Field` (ver seção 3.9), e armazenado numa lista (`FieldList`) mantida na classe `Type`.

Quando o tipo representa uma função, operador _FUNCTION, existe uma lista de tipos (`TypeList`) que contém os tipos dos parâmetros da função. O gerador de código C [7] precisa dessa informação para gerar corretamente os protótipos das funções, sobretudo daquelas que não são declaradas e nem implementadas no arquivo corrente.

Todos os objetos `Type` ficam armazenados na tabela de símbolos global, mantida pela classe `File`, descrita na seção 3.2. Mesmo os tipos declarados localmente são colocados na tabela de símbolos globais, pois após a fase de análise semântica, a verificação de escopos para os tipos já foi realizada.

Nunca são criados dois objetos `Type` que representem o mesmo tipo, ou seja, uma vez que existe um objeto `Type` para, por exemplo, representar o tipo *char*, não será mais

necessário criar outro objeto `Type` para este tipo. O Xingó faz uma reutilização dos objetos `Type`, de forma que um único objeto `Type` representando o tipo *char* será utilizado para representar todas as variáveis que sejam deste tipo ou que sejam de algum tipo composto envolvendo *char*, como um *char**, *unsigned char* ou *char[]*. Essa solução evita manter replicações desnecessárias de um tipo, e, conseqüentemente, utiliza menos memória.

3.9 Campos de Tipos Agregados

Tipos agregados (*struct* e *union*) possuem campos. A estes campos estão associados nome, *offset* dentro da estrutura, quantidade de *bits* e tipo. A classe `Field` foi implementada para representar um campo de um tipo agregado.

Na figura 3.10 está ilustrada a representação dos dados da classe `Field`.

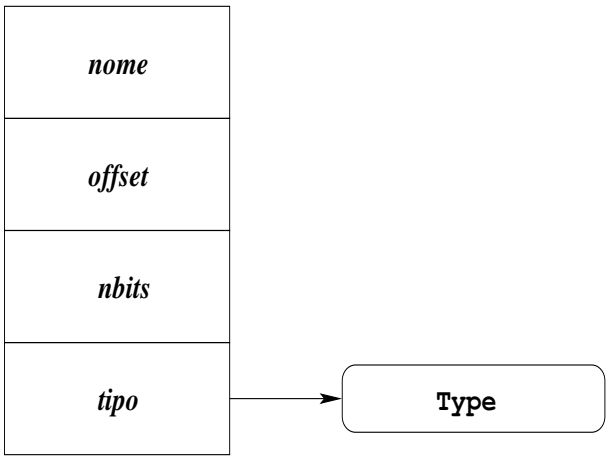

Figura 3.10: Representação dos dados da classe `Field`

O nome do campo é o mesmo nome a ele dado no arquivo de entrada. O *offset* dá o deslocamento em *bytes* do campo em relação ao início da estrutura. O *offset* para campos de uma *union* é sempre zero, pois esta estrutura só consegue guardar um único valor.

Quando um campo é um campo de *bits*[5], o seu tipo é *int* ou *unsigned*, porque estes são os únicos dois tipos permitidos para este tipo de campo. Para um campo de *bits*, *nbits* informa a quantidade de *bits* ocupada por ele. Caso contrário, *nbits* é sempre zero.

[5]Um campo de *bits* é aquele que é definido para conter uma quantidade certa de *bits*, como por exemplo 1, 2, ou 4 *bits*.

O ponteiro *tipo* aponta para o objeto `Type` que representa o tipo do campo.

3.10 Anotações

O compilador Xingó foi projetado para ser flexível e permitir extensão. A representação intermediária do compilador não foge a esta regra. As anotações são informações arbitrárias (informações extras) que podem acompanhar os objetos do compilador. Dessa maneira, é possível agregar aos objetos Xingó informações arbitrárias além daquelas predefinidas pelas classes destes objetos.

Certas instruções e variáveis associam a si alguma informação extra que pode ser útil em algum momento do processo de compilação, como a geração do código C ou durante a geração de código *assembly*. Para armazenar estas informações, a XIR se utiliza da classe `Note`. Na figura 3.11 está ilustrada a representação dos dados da classe `Note`.

Uma anotação contém dois campos: um identificador, que é um código inteiro que identifica a anotação, e uma lista de objetos da classe `Immed`, descrita na próxima seção. Os objetos desta lista representam os dados da anotação.

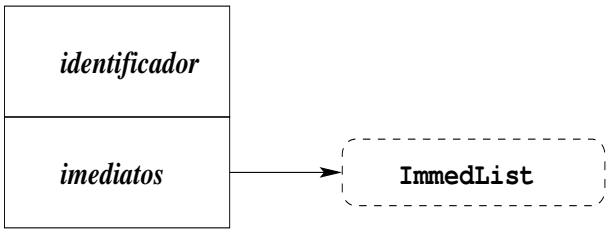

Figura 3.11: Representação dos dados da classe `Note`

A classe `Note` pode ser usada para armazenar o valor com que uma determinada variável do segmento `D_DATA` será inicializada, o valor das cadeias de um *array* de *strings*, como também pode estar associada a uma instrução do programa e armazenar informações relativas a uma *Jump Table*.

O compilador Xingó possui um conjunto de anotações predefinidas. O desenvolvedor pode ainda criar outras anotações. Para isso, ele precisa definir um código para a anotação e ele próprio dar a semântica apropriada à lista de imediatos que agrega as informações da anotação. As anotações predefinidas estão relacionadas a seguir. Por convenção, a

constante que define o código de uma anotação é iniciada por `NOTE_`.

`NOTE_INT size value`
Definição de um dado do tipo *int*. *size* informa o tamanho em *bytes* do dado e *value* informa o valor do dado.

`NOTE_UNSIGNED size value`
Definição de um dado do tipo *unsigned*.

`NOTE_FLOAT size value`
Definição de um dado do tipo *float*. *value* é tratado como um inteiro não sinalizado.

`NOTE_POINTER size value`
Definição de um endereço absoluto.

`NOTE_ADDRESS address offset`
Definição de um endereço simbólico. *address* informa o nome do símbolo endereçado e *offset* informa o deslocamento em relação ao endereço base do símbolo.

`NOTE_DOUBLE size value1 value2`
Definição de um dado do tipo *double*. *size* informa o tamanho em *bytes* do valor, *value1* fornece o valor da primeira metade dos *bytes* e *value2* o valor da segunda metade dos *bytes*. *value1* e *value2* são tratados como inteiros não sinalizados.

`NOTE_SPACE n`
Definição de *n bytes* com o valor zero.

`NOTE_JUMP_TABLE n L1 L2...Ln`
Definição de *Jump Table*. *n* é o número de endereços alvo (*labels*) possível. *Li* é o conjunto de *labels* destino. Cada Li é um apontador para um objeto do tipo `Label`.

3.11 Imediatos

Os dados de uma anotação são representados por valores imediatos. Um valor imediato deve ser de um ítem de dado de um tipo escalar (*char, int, float,* etc), ou pode ser também uma referência a um símbolo, dado pela classe `Symbol`.

É importante não confundir os imediatos referidos aqui com o valor imediato de uma instrução. A classe `Immed` representa um imediato de uma lista de anotações, e não um valor constante de uma instrução.

Na figura 3.12 está ilustrada a representação dos dados da classe `Immed`.

Figura 3.12: Representação dos dados da classe `Immed`

Um objeto `Immed` contém dois campos: o valor, e o *kind*, que deve ser um dos ítens do tipo enumerado `immedKinds`.

```
enum immedKinds{
   im_char = 'c',
   im_short = 'h',
   im_int = 'i',
   im_unsigned = 'u',
   im_long = 'l',
   im_float = 'f',
   im_double = 'd',
   im_string = 's',
   im_sym = 'p',
};
```

Capítulo 4

Geração de Código Intermediário Xingó

Como foi dito no capítulo 1, o compilador LCC implementa o *front-end* do compilador Xingó. A representação intermediária do LCC é baseada em DAGs e a representação intermediária do Xingó é baseada em quádruplas. Este capítulo descreve a geração da Representação Intermediária Xingó, que é obtida convertendo-se a representação em forma de DAGs do LCC para uma representação linear em forma de quádruplas, utilizada pelo Xingó.

A decisão de construir uma representação intermediária própria, e não utilizar diretamente a do LCC, se deu por duas razões principais:

1. O LCC é escrito em C, o Xingó em C++. Apesar de o casamento entre as duas linguagens ser perfeitamente possível, isso poderia trazer problemas de manutenção de código, dificuldade de escalabilidade, dentre outros.

2. Uma modificação no *front-end*, ou na representação intermediária do LCC, poderia acarretar uma mudança em todas as partes do Xingó, e não somente no módulo *XIR Generator* (ver seção 4.2), que faz a geração da XIR.

A apresentação da geração de código intermediário Xingó será iniciada com uma descrição do LCC.

4.1 LCC

O LCC [14] é um compilador redirecionável, de produção, que compila programas escritos na linguagem de programação ANSI C. Ele tem sido usado para compilar programas de produção desde 1988, sendo usado por centenas de programadores C, e em outros projetos de pesquisa na área de compiladores [5][21].

No contexto do compilador Xingó, o LCC tem sido utilizado como *front-end*, mas poderia perfeitamente, em outros contextos, ser utilizado para gerar código alvo para uma máquina real. O LCC apresenta geradores de código para MIPS R3000, SPARC, Intel 386, dentre outras plataformas. Ele também tem sido adaptado para outros propósitos além da compilação tradicional. Por exemplo, ele tem sido usado para construção de *C browser* e para geração de *stubs* de declaração em chamadas de procedimento remoto [14].

O *front-end* do LCC tem aproximadamente 9000 linhas de código, que além das tarefas básicas de análise léxica, sintática e semântica do código de entrada, realiza também algumas técnicas simples de otimização, tais como eliminação de subexpressões comuns locais e propagação de constantes, que melhoram a qualidade do código local.

O LCC foi ecolhido como *front-end* do Xingó pelo fato de ser razoavelmente estável e rápido (como dito anteriormente, ele vem sendo desenvolvido e aperfeiçoado desde 1988), por ser um compilador de código aberto e pelo fato de que sua documentação é gerada utilizando-se *literate programming*, isto é, ela é feita por meio de um livro. Tal livro [14], mostra todo o funcionamento e a estrutura interna do compilador. Desta forma, torna-se mais simples entender, usar e modificar o LCC.

4.1.1 A Representação Intermediária do LCC

A Representação Intermediária do LCC, que vai ser aqui denominada simplesmente por LIR, é baseada em DAGs. Os símbolos que aparecem nos DAGs estão armazenados em tabelas de símbolos. Um símbolo pode ser uma variável, um *label* ou uma constante. Variáveis e constantes têm tipo, que são colocados em uma tabela específica para tipos. Ainda existem tabelas para constantes, para símbolos externos, símbolos globais, tabela de identificadores e tabela para *labels*.

Inicialmente, o *front-end* traduz expressões para ASTs (Árvores Sintáticas Abstratas), que depois são transformadas em DAGs (Grafos Direcionados Acíclicos). As operações

de desvio, *jumps* e *labels*, também são representadas por DAGs.

O código executável de uma função é especificado por DAGs. O corpo de uma função é uma seqüência, ou floresta, de DAGS, que são colocados juntos numa lista. Para gerar código relativo à função, deve-se percorrer a tal lista. Os campos que formam o nodo de um DAG são os seguintes:

1. um operador, que diz a operação realizada pelo nodo. Este operador corresponde a um dos 36 operadores descritos na tabela 4.1.

2. um contador, que dá a quantidade de vezes que o nodo é referenciado por outros.

3. apontadores para símbolos utilizados pelo nodo. Um nodo pode utilizar até três símbolos, sendo que os dois primeiros devem ser usados pelo *front-end*, e o último pelo *back-end* (no caso, Codificador Xingó).

4. apontadores para os nodos filhos. Um nodo pode ter até dois filhos.

5. um apontador para o nodo raiz da próxima floresta.

4.2 O Módulo *XIR Generator*

O Módulo *XIR Generator* faz a geração da Representação Intermediária Xingó. Mais especificamente, este módulo faz a tradução da representação intermediária usada pelo LCC para a representação intermediária usada pelo Xingó. De um modo geral, *XIR Generator* é responsável pela interface entre o compilador LCC e o compilador Xingó. Na figura 4.1, é apresentado um diagrama que mostra como o módulo *XIR Generator* está integrado ao Xingó.

O *front-end* do LCC valida um arquivo C, que será chamado de arquivo de entrada, e gera código para a Representação Intermediária do LCC, aqui denominada de LIR. Então, o módulo *XIR Generator* transforma o programa representado em LIR para um programa equivalente na Representação Intermediária Xingó, ou XIR.

Depois que a geração da XIR é completada, não existe mais nenhum vínculo entre os dados representados em LIR e os dados representados em XIR. As outras partes do compilador Xingó passam a trabalhar apenas sobre a XIR, e não precisam fazer nenhum

Operador	**descrição**
ADDRF	*endereço de um parâmetro*
ADDRG	*endereço de um símbolo global*
ADDRL	*endereço de um símbolo local*
CNST	*constante*
BCOM	*complemento binário*
CVC	*converte de char*
CVD	*converte de double*
CVF	*converte de float*
CVI	*converte de int*
CVP	*converte de ponteiro*
CVS	*converte de short*
CVU	*converte de unsigned*
INDIR	*leitura da memória*
NEG	*negação*
ADD	*adição*
BAND	*e binário*
BOR	*ou binário*
BXOR	*ou exclusivo*
DIV	*divisão*
LSH	*deslocamento de bits a esquerda*
MOD	*resto de divisão*
MUL	*multiplicação*
RSH	*deslocamento de bits a direita*
SUB	*subtração*
ASGN	*atribuição*
EQ	*salta se igual*
GE	*salta se maior ou igual*
GT	*salta se maior*
LE	*salta se menor ou igual*
LT	*salta se menor*
NE	*salta se não igual*
ARG	*argumento*
CALL	*chamada a uma função*
RET	*retorno de função*
JUMP	*salto incondicional*
LABEL	*definição de label*

Tabela 4.1: Operadores do LCC

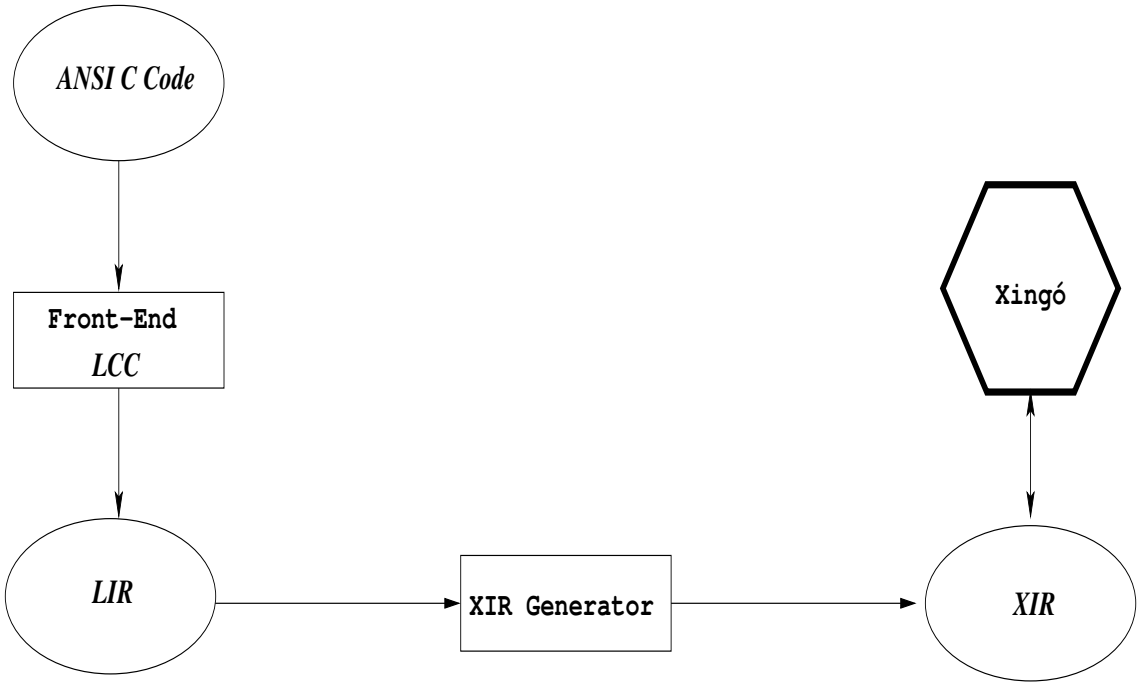

Figura 4.1: `XIR Generator`: Módulo de interface entre LCC e Xingó

tipo de referência às estruturas de dados do LCC. Na figura 4.2 é mostrada com mais detalhes, a interface entre LCC e Xingó.

Como passo intermediário na geração da XIR, é gerado um arquivo temporário (*XIR File*) que contém um código abstrato. O código abstrato é gerado pelo módulo *XIR Coder*, que codifica as informações das estruturas de dados da LIR. Depois o código abstrato é decodificado pelo módulo *XIR Decoder*, que monta as estruturas de dados da Representação Intermediária Xingó. Este código abstrato é visto com mais detalhes na seção 4.3.

O arquivo com o código de máquina abstrato (*XIR File*) será denominado simplesmente por *arquivo xingó*. Logo depois que o *arquivo xingó* é decodificado, ou seja, é completada a conversão do programa em LIR para o programa em XIR, ele é apagado do sistema, exceto quando o usuário escolhe a opção de gerar como saída o próprio *arquivo xingó*.

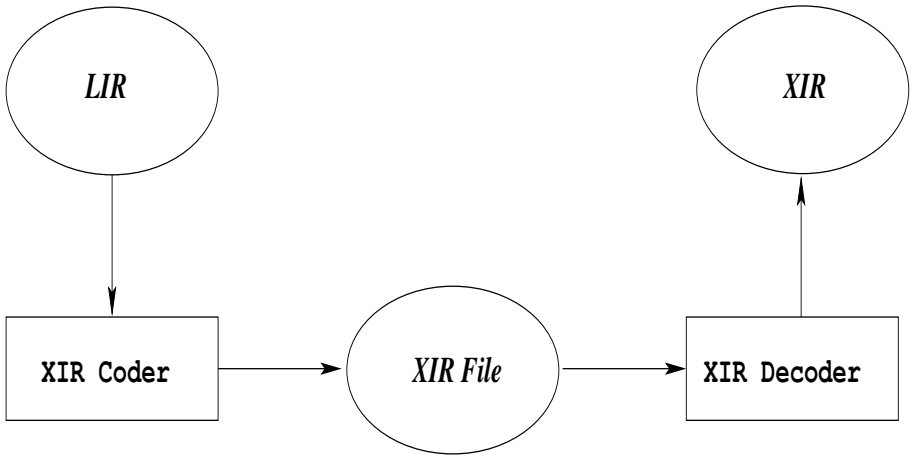

Figura 4.2: Codificador e Decodificador Xingó

4.3 Código Abstrato Xingó

O Código Abstrato Xingó utiliza uma simbologia simples para codificar as informações das estruturas de dados da LIR. O *arquivo xingó*, que contém o código abstrato, é editável, e por isso pode ser visualizado em um editor de texto comum. Inclusive, durante o desenvolvimento deste projeto de mestrado, por diversas vezes foi necessário recorrer ao *arquivo xingó* para descobrir o erro na geração da XIR.

A opção por gerar o *arquivo xingó*, e não diretamente construir a IR do Xingó a partir da IR do LCC, foi pelo simples motivo de trazer maior modularidade ao compilador. Assim, alguma modificação no LCC ou na sua IR não implica, necessariamente, uma modificação na IR do Xingó. Desse modo, existe um único ponto de ligação entre o LCC e o Xingó, que é o *arquivo xingó*. Isso abre também a possibilidade de incluir, de maneira mais simples, novos *front-ends* ao compilador Xingó, considerando que seja gerado um mesmo código abstrato (com mesma sintaxe) pelos diferentes *front-ends*.

O Código Abstrato Xingó possui dois tipos de informações, que serão chamadas de informações declarativas e informações de operação. Exemplos de informações declarativas são definições de variáveis, de procedimentos, de símbolos *extern*, dentre outras. As informações de operação codificam as instruções que representam o código do programa em compilação.

4.3.1 Informações Declarativas

As informações simbólicas da LIR são codificadas através das denominadas informações declarativas. Cada informação declarativa tem associada a ela um número inteiro que a identifica. No *arquivo xingó*, uma informação declarativa começa no início de uma linha, e é reconhecida porque inicia com um caractere "/". Após este caractere, seguem o código da informação e os campos que a constitui. Na tabela 4.2 são relacionados todos os códigos das informações declarativas e, em seguida, são descritos os campos que compõem estas informações.

Código	**Descrição**
D_CODE	*início de segmento CODE*
D_DATA	*início de segmento DATA*
D_BSS	*início de segmento BSS*
D_LIT	*início de segmento LIT*
D_INT	*constante inteira sinalizada*
D_UNSIGNED	*constante não sinalizada*
D_POINTER	*constante que é um ponteiro*
D_FLOAT	*constante flutuante de precisão simples*
D_DOUBLE	*constante flutuante de precisão dupla*
D_STRING	*informa uma string*
D_ADDRESS	*define endereço*
D_SPACE	*define n bytes com valor zero*
D_ALIGN	*define alinhamento*
D_EXPORT	*informa um símbolo a ser exportado*
D_IMPORT	*informa um símbolo importado*
D_PROC	*início de um procedimento*
D_ENDPROC	*fim de um procedimento*
D_LOCAL	*define uma variável local*
D_PARAM	*define um parâmetro*
D_VARIABLE	*define uma variável global*

Tabela 4.2: Informações declarativas

Descrição dos Campos de uma Informação Declarativa

As informações declarativas `D_CODE`, `D_DATA`, `D_LIT`, `D_BSS` e `D_ENDPROC` não possuem nenhum campo associado a elas. `D_CODE`, `D_DATA`, `D_LIT` e `D_BSS` informam o início de um segmento lógico, explicado nas próximas páginas. `D_ENDPROC` simplesmente indica o término das informações a respeito de um procedimento. As demais declarações possuem um ou mais campos associados, que serão descritos a seguir.

`code`	*size*	*value*

- `code`: `D_INT`, `D_UNSIGNED`, `D_POINTER`, ou `D_FLOAT`.
- ***size*:** tamanho em *bytes*.
- ***value*:** valor da constante.
- *obs:* No caso de `D_FLOAT`, o valor da constante é codificado como um inteiro não sinalizado.

`D_DOUBLE`	*size*	*value1*	*value2*

- ***size*:** tamanho em *bytes*.
- ***value1*:** valor da primeira metade dos *bytes* da constante.
- ***value2*:** valor da segunda metade dos *bytes* da constante.
- *value1* e *value2* são representados como inteiros não sinalizados.

`D_STRING`	*length*	*string*

- ***length*:** quantidade de caracteres da *string*, excluído o terminador "\0".
- ***string*:** seqüência de caracteres da *string*. Os caracteres são escritos em hexadecimal.

`D_ADDRESS`	*name*	*offset*

- ***name*:** nome do símbolo endereçado.

- ***offset*:** *offset* a ser utilizado.

D_SPACE	*nbytes*

- ***nbytes*:** número de *bytes* que devem ser inicializados com o valor 0.

D_ALIGN	*align*

- ***align*:** alinhamento dos dados de um ítem.

D_EXPORT	*name*

- ***name*:** nome do símbolo a ser exportado.

D_IMPORT	*name*	*type*

- ***name*:** nome do símbolo declarado em outro arquivo.
- ***type*:** tipo do símbolo.

D_PROC	*name*	*varargs*	*ncalls*	*sclass*	*type*

- ***name*:** nome do procedimento.
- ***varargs*:** se 1, o procedimento tem quantidade de argumentos variável. Se zero, o procedimento tem todos os argumentos bem definidos.
- ***ncalls*:** número de chamadas feitas a outros procedimentos.
- ***sclass*:** classe[1] do procedimento.
- ***type*:** tipo de retorno do procedimento.

D_LOCAL	*offset*	*name*	*sclass*	*type*

- ***offset*:** *offset* da variável na pilha.
- ***name*:** nome da variável.

[1] A classe de um símbolo será explicada a seguir.

- ***sclass*:** classe da variável.
- ***type*:** tipo.

D_PARAM	*offset*	*name*	*type*

- mesmo que D_LOCAL, exceto que não é necessário registrar a classe, uma vez que ela é sempre AUTO.

D_VARIABLE	*name*	*sclass*	*type*

- ***name*:** nome da variável.
- ***sclass*:** classe da variável.
- ***type*:** tipo.

Segmentos Lógicos

O *front-end* do LCC gerencia quatro segmentos lógicos que separam código, dados e constantes. Os segmentos são *CODE, DATA, BSS* e *LIT*.

O segmento *CODE* informa o início de um trecho de código. *DATA* é o segmento onde são definidas as variáveis inicializadas, enquanto as não inicializadas são definidas no segmento *BSS*. Constantes são definidas no segmento *LIT*.

Cada máquina alvo tem uma forma própria de mapear os segmentos lógicos, mas normalmente, *CODE* e *LIT* podem ser mapeados para segmentos de somente leitura. *BSS* e *DATA* devem ser mapeados para segmentos que podem ser lidos e escritos.

Classe de um Símbolo

O *front-end* do LCC define quatro classes para um símbolo: *STATIC, EXTERN, REGISTER,* e *AUTO*.

Se o símbolo tem classe *STATIC* ele é visível apenas no arquivo onde foi declarado. Se ele é *EXTERN*, ele deve ser exportado, podendo ser visível em outros arquivos. Um símbolo que tem classe *REGISTER* deve ser mapeado em um registrador. Se o símbolo não é de nenhuma das classes anteriores, ele é *AUTO*.

4.3.2 Informações de Operação

As operações (instruções) de um procedimento em LIR são codificadas através das denominadas informações de operação. Cada informação de operação tem um código inteiro que a identifica. No *arquivo abstrato xingó*, uma informação de operação começa no início de uma linha, e é reconhecida porque inicia-se com um caractere "∗". Após este caractere, seguem o código da operação e os campos que a constitui.

O código da operação corresponde a um dos códigos que foram relacionados na tabela 4.1, incluído mais um operador, `ADDRX`, que foi criado para informar uma operação que se refere ao endereço de um procedimento chamado. Este operador foi criado porque o *front-end* do LCC codifica uma operação de tomada de endereço de um procedimento, usando o mesmo operador (`ADDRG`) utilizado para codificar uma tomada de endereço de uma variável global. Para separar esses dois tipos de tomada de endereço que foi criado o operador `ADDRX`.

Descrição dos Campos de uma Informação de Operação

CNST	*type*	*value*

- ***type*:** tipo da constante.
- ***value*:** valor.

ARG	*type*	*variadic*

- ***type*:** tipo do argumento.
- ***variadic*:** 1, se é um dos argumentos da parte de argumentos variáveis de uma função. Caso contrário, 0.

ASGN	*type*

- ***type*:** tipo do valor atribuído.

INDIR	*type*

- ***type*:** tipo do conteúdo lido da memória.

CVC	*type*

- ***type*:** tipo destino.
- CVD, CVF, CVI, CVP, CVS e CVU apresentam o mesmo campo.

NEG	*type*

- ***type*:** tipo do resultado.
- ADD, SUB, LSH, MOD, RSH, BAND, BCOM, BOR, BXOR, DIV, e MUL apresentam o mesmo campo.

CALL	*type*

- ***type*:** tipo de retorno da função.

RET	*type*

- ***type*:** tipo do ítem retornado.

code	*type*	*name*	*offset*

- code: ADDRG ou ADDRX.
- ***type*:** tipo do símbolo endereçado.
- ***name*:** nome do símbolo.
- ***offset*:** o *offset* a ser utilizado dentro do símbolo.

code	*type*	*slot*	*offset*

- code: ADDRL ou ADDRF.
- ***type*:** tipo do símbolo endereçado.
- ***slot*:** *slot* do símbolo na pilha.
- ***offset*:** o *offset* a ser utilizado dentro do símbolo.

code	*type*	*label*

- code: EQ, GT, LE, LT, ou NE.
- ***type*:** tipo dos operandos comparados.
- ***label*:** nome do *label* alvo do desvio.

JUMP	*islabel*

- ***islabel*:** 1, se o endereço de desvio é dado por um *label*. 0, se o endereço é dado pelo conteúdo de uma variável.

LABEL	*name*

- ***name*:** nome do *label*.

4.3.3 Layout do Arquivo Xingó

O *arquivo xingó* apresenta, na maioria dos casos, o *layout* ilustrado na figura 4.3. Fala-se "na maioria dos casos" porque a ordem com que são colocadas as informações sobre os procedimentos, variáveis globais não inicializadas e constantes pode ser trocada. No entando, variáveis globais inicializadas estão sempre no início do arquivo. O *layout* de um procedimento é sempre este apresentado na figura 4.3. Primeiro vêm os parâmetros declarados, seguidos das variáveis locais do procedimento e das operações. Todas as camadas apresentadas na figura são facultativas, ou seja, podem ou não aparecer no *arquivo xingó*.

4.4 O Codificador Xingó

O LCC tem um ponto forte, a modularidade. Isso facilita o desenvolvimento de um compilador redirecionável. Para construir um *back-end* para o LCC, basicamente o desenvolvedor precisa implementar uma interface de geração de código e definir algumas configurações sobre a máquina alvo. É evidente que, dependendo da complexidade do *back-end*, ou das informações que se queira extrair do LCC, alterações ou adição de novas informações no LCC podem ser necessárias, como foi o caso dos módulos implementados e

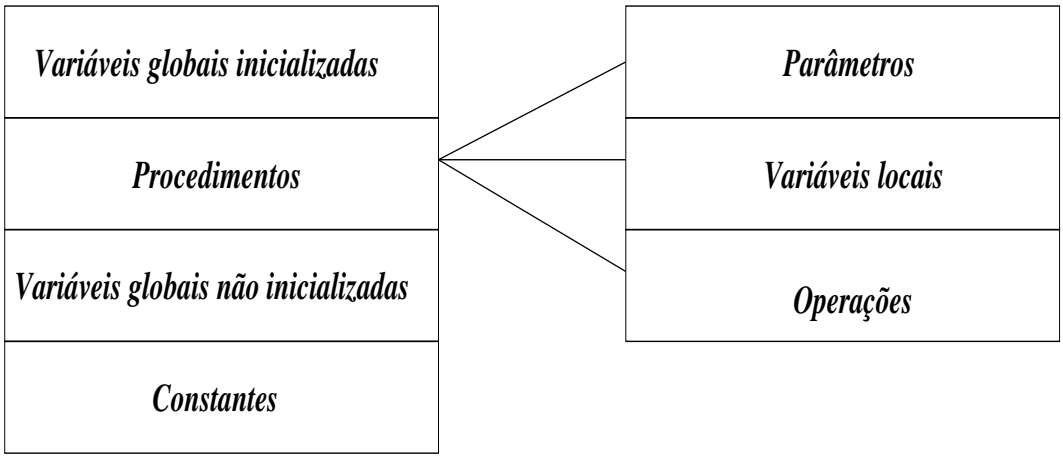

Figura 4.3: Layout do arquivo xingó

descritos neste capítulo. Mas, via de regra, é possível gerar código alvo apenas utilizando essa interface de geração de código.

A interface de geração de código do LCC define a ligação entre o *front-end* e o *back-end* do compilador. Ela consiste de umas poucas estruturas de dados compartilhadas, 18 funções e 36 operadores (tabela 4.1) que especificam o código de um programa. O *front-end* e o *back-end* compartilham alguns campos das estruturas de dados compartilhadas, mas outros campos são privados ao *front-end* ou ao *back-end*. Dentre as estruturas de dados compartilhadas, estão símbolos diversos (procedimentos, variáveis, *labels*, etc) e tipos.

Além das funções e dos operadores, também faz parte da interface de geração de código, um registro com a configuração da máquina alvo.

A interface de geração de código, no caso do compilador Xingó, não foi utilizada para construir um *back-end* para uma máquina real, mas foi utilizada para construir o Codificador Xingó, que faz a geração do *arquivo xingó*, que contém o código abstrato para montar a Representação XIR.

4.4.1 Implementação do Codificador

A implementação do codificador é a implementação da própria interface de geração de código do LCC. Abaixo são descritas as implementações de cada função da interface.

- `static void I(progbeg)(int argc, char *argv[])`
  `progbeg` é chamada no início do *front-end*, antes mesmo de iniciar a compilação do programa de entrada. Ela recebe como entrada as opções de compilação. Na implementação de `progbeg` não foi codificada nenhuma ação, pois as opções de compilação do Xingó são tratadas em outro módulo.

- `static void I(progend)(void)`
  `progend` é chamada ao término do *front-end*. Esta função emite no *arquivo xingó*, uma linha com o *caracter* "$", que é o marcador de final de Código Abstrato Xingó.

- `static void I(defsymbol)(Symbol p)`
  `defsymbol` define um novo símbolo. Se o símbolo é um parâmetro ou uma variável local, ele é definido e inicializado pela função `function`, ao passo que aqueles símbolos que representam computação de endereço, pela função `address`. `defsymbol` recebe como entrada o símbolo a ser definido. Inicialmente, é dado um nome ao símbolo definido. Se o símbolo for uma constante, o nome do símbolo será o valor desta constante. Se o símbolo for um *label*, seu nome será uma *string* com prefixo ".L". O sufixo é uma seqüência de dígitos, que foi o nome dado pelo *front-end* a esse *label*. Se o símbolo for um símbolo gerado pelo *front-end*, ele é nomeado com uma *string* começando por ".C" seguida de dígitos. É garantido ter sempre nomes únicos, pois a seqüência de dígitos é gerada por uma função do *front-end* que sempre retorna um número único. Se o símbolo for global ou tiver escopo "*static*", o nome do símbolo continua sendo o mesmo nome a ele dado no arquivo de entrada.

- `static void I(export)(Symbol p)`
  A função `export` é chamada para anunciar que um símbolo pode ser exportado para outros módulos. Ela recebe como entrada o símbolo a ser exportado. `export` é chamada apenas para símbolos que tenham classe diferente de *static*. O *front-end* chama `export` antes da definição do símbolo, feita por `defsymbol`. A função `export` gera a informação declarativa `D_EXPORT`.

- `static void I(import)(Symbol p)`
  A função `import` é chamada para anunciar que um símbolo é importado de outro módulo. Ela recebe como entrada o símbolo importado. Um símbolo é importado

se ele não está declarado no arquivo, ou seja, só existe o seu uso. O *front-end* chama `import` a qualquer momento, antes ou depois da definição do símbolo. A função `import` gera a informação declarativa `D_IMPORT`.

- `static void I(global)(Symbol p)`
  A função `global` é chamada pelo *front-end* para que seja emitido código para definir um símbolo global. Ela recebe como entrada o símbolo global e armazena, no *arquivo xingó*, duas informações declarativas: `D_ALIGN` e `D_VARIABLE`.

- `static void I(local)(Symbol p)`
  A função `local` anuncia uma variável local. Ela recebe como entrada o símbolo que representa a variável local. Uma variável local é aquela definida dentro do escopo de um procedimento. Variáveis temporárias criadas pelo *front-end* também são anunciadas pela chamada à função `local`. `local` gera a informação declarativa `D_LOCAL`.

- `static void I(address)(Symbol q, Symbol p, long n)`
  A função `address` é usada para inicializar o símbolo q como um símbolo que representa um endereço da forma $x+n$, onde x é o endereço representado por p, e n é um número inteiro positivo ou negativo. `address` aceita símbolos globais, locais e parâmetros, e é chamada somente depois que os símbolos tiverem sido inicializados. `address` realiza duas operações: nomea o símbolo q, e define o seu endereço, que é dado pelo seu *offset*. O nome dado a q é o mesmo nome do símbolo p, e o seu *offset* é a soma do *offset* de p e o valor n passado como parâmetro para a função.

- `static void I(segment)(int s)`
  A função `segment` anuncia o segmento ao qual as próximas definições pertencerão. Ela recebe como entrada o número do segmento e gera as informações declarativas `D_CODE`, `D_DATA`, `D_LIT` e `D_BSS`.

- `static void I(defconst)(int suffix, int size, Value v)`
  A função `defconst` inicializa uma constante numérica. *suffix* é o sufixo do tipo da constante. *size* é o tamanho em *bytes* ocupado pela constante. v guarda o valor da constante. Os sufixos do *front-end* são:

  – F: tipo *float*.

  - D: tipo *double*.
  - C: tipo *char*.
  - S: tipo *short*.
  - I: tipo *int*.
  - U: tipo *unsigned*.
  - P: tipo ponteiro.
  - V: tipo *void*.
  - B: tipo *struct/union*.

  O LCC considera os seguintes sufixos como sendo permitidos para uma constante: C, S, I, U, P, F e D.

  Na implementação de `defconst`, considera-se:

  - Se o sufixo é igual a C, S ou I, gera-se a informação declarativa `D_INT`.
  - Se o sufixo é igual a U, gera-se a informação declarativa `D_UNSGINED`.
  - Se o sufixo é igual a P, gera-se a informação declarativa `D_POINTER`.
  - Se o sufixo é igual F, gera-se a informação declarativa `D_FLOAT`.
  - Se o sufixo é igual D, gera-se a informação declarativa `D_DOUBLE`.

- `static void I(defaddress)(Symbol p)`
  A função `defaddress` inicializa um ponteiro para um símbolo, como por exemplo, um ponteiro para uma função. `defaddress` gera a informação declarativa `D_ADDRESS`.

- `static void I(defstring)(int len, char *s)`
  A função `defstring` deve emitir código para inicializar uma *string* de tamanho *len* e seqüencia de caracteres apontada por *s*. `defstring` gera a informação declarativa `D_STRING`.

- `static void I(space)(int n)`
  A função `space` emite código para inicializar *n bytes* com o valor zero. `space` gera a informação declarativa `D_SPACE`.

- `static void I(blockbeg)(Env *e)`
  `static void I(blockend)(Env *e)`
  As funções `blockbeg` e `blockend` são chamadas para tratar aquelas porções de programa que limitam o tempo de vida de variáveis locais. `blockbeg` é chamada no início de um bloco e `blockend` no fim. Na representação XIR, todos os símbolos definidos dentro de um procedimento ficam na tabela de símbolos local, portanto, não existe na representação XIR diferença se o símbolo foi declarado dentro ou fora de um bloco. Essa informação de escopo é mais útil para o *front-end*. Por este motivo, as implementações de `blockbeg` e `blockend` não geram nenhuma informação declarativa.

- `static Node I(gen)(Node p)`
  `static void I(emit)(Node p)`
  `static void I(function)(Symbol f, Symbol caller[], Symbol callee[], int ncalls)`
  `gen, emit` e `function` são apresentadas juntas porque elas trabalham de maneira integrada. O *front-end* analisa completamente a função corrente antes de passar qualquer parte dela para o *back-end*. No fim de cada função, o *front-end* chama `function` para gerar e emitir código. A função `gen` dá ao *back-end* a oportunidade de realizar alguma alteração nos DAGs, e deve ser chamada antes de `emit`. A função `emit` emite código para o DAG. O argumento f de `function` aponta para o símbolo que é a função corrente, e *ncalls* é o número de chamadas para outras funções feitas pela função corrente. Em *caller* estão os parâmetros da função como vistos pelo chamador. Em *calle* estão os parâmetros como vistos pela função corrente.

4.4.2 Codificação de uma Função

Toda vez que o *front-end* compila uma função ele chama a função de interface `function`, que é quem gera o código abstrato relativo a função corrente. Assim, são realizadas, nesta ordem, as seguintes ações:

1. caso o segmento lógico corrente não seja `CODE`, emite-se a informação declarativa `D_CODE`. Isso assegura que o código e os dados de uma função ficam no segmento lógico de código, e não no de dados, por exemplo.

2. Calcula-se o *offset* dos parâmetros e emite para cada um deles a informação declarativa `D_PARAM`.

3. Emite a informação declarativa `D_LOCAL` para cada variável local.

4. Anuncia-se o início do código do procedimento, através da informação declarativa `D_PROC`.

5. Emite o código do procedimento, que é constituído de informações de operação.

6. Anuncia o fim do procedimento, através da informação declarativa `D_ENDPROC`.

4.4.3 Codificação de Operações

A função de interface `emit` é chamada para emitir código para uma função. Ela recebe como entrada o nodo raiz da primeira árvore que forma o DAG da função corrente. O nodo raiz de uma árvore, além de apontar para os nodos filhos, também aponta para o nodo raiz da próxima árvore. O que é feito na implementação de `emit` é percorrer as árvores através de um caminhamento pós-ordem, emitindo código abstrato xingó para cada uma das árvores. Utilizando caminhamento pós-ordem, primeiro emite-se código para os nodos filhos, depois para o nodo pai. Isso faz com que as operações no Código Abstrato Xingó estejam baseadas numa representação em notação pós-fixa [2], podendo ser decodificadas utilizando-se uma pilha, como de fato é feito.

4.4.4 Codificação de Tipos

As informações sobre o tipo de uma variável, procedimentos e campos de tipos agregados (*struct* e *union*) também são codificadas no *arquivo xingó*. Como visto no capítulo 3, um tipo é composto de operador e operandos. Nesta seção, será mostrada a codificação do tipo de um símbolo, que implica codificar operador e operandos, recursivamente. Na figura 4.4 é ilustrado o diagrama da codificação de um tipo no *arquivo xingó*.

Se o tipo já foi codificado, significa que as informações sobre o seu operador e os seus operandos já foram escritas no *arquivo xingó*. Nesse caso, apenas o identificador do tipo precisa ser armazenado. O identificador do tipo foi um dos campos adicionados às estruturas de dados do LCC pelo Xingó.

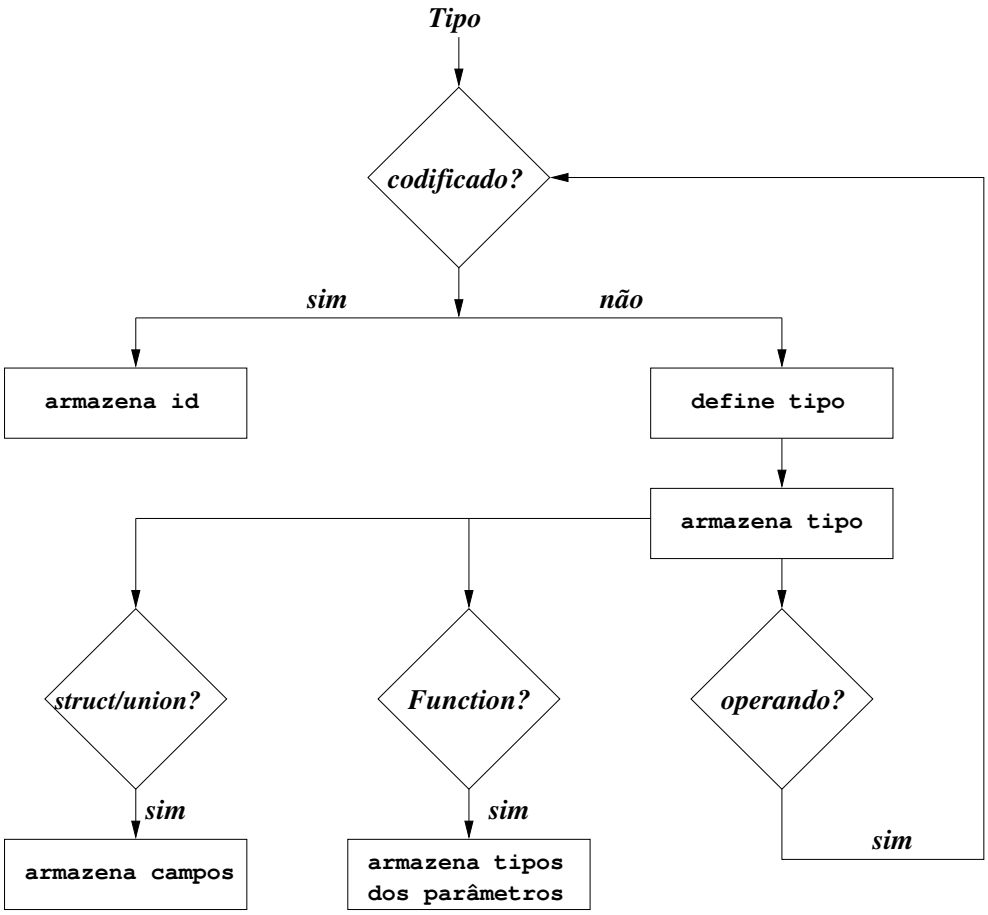

Figura 4.4: codificação de informações sobre tipos

Caso o tipo ainda não tenha sido codificado, ele é definido. Definir o tipo é dar a ele um identificador, armazenar no *arquivo xingó* o caracter "0" e o seu identificador, e marcá-lo como codificado. Para o Decodificador, o caracter "0" vai significar que o tipo está sendo definido pela primeira vez. Depois de definido o tipo, a codificação continua com o armazenamento das informações sobre o seu operador no *arquivo xingó*. As informações de um operador compreendem o código do operador, o tamanho, o alinhamento, e o nome, nesta ordem. O nome fica entre "(" e ")".

Se o tipo é de um *struct* ou de um *union*, os seus campos também são codificados. A codificação dos campos é descrita na seção seguinte. Caso o tipo seja de uma *função*, os tipos de seus parâmetros são codificados em seguida. Com isso, é possível chegar ao

protótipo da função. Se o tipo tem operando, este é passado para realimentar o esquema.

Por exemplo, a declaração `double **p;` codificaria como tipo de `p`

```
0
50 7 4 4 (pointer) 0
51 7 4 4 (pointer) 0
52 2 8 4 (double)
```

O código do tipo *ponteiro para* é 7, e do tipo *double* é 2. No caso, o tamanho e o alinhamento de um ponteiro são iguais a 4, e de um *double* o tamanho é 8 e o alinhamento é 4.

Se logo abaixo, no arquivo de entrada, houver uma declaração `double *q;` seria codificado como tipo de `q`:

```
51
```

ou seja, apenas o identificador (identificador 51) do seu tipo (*ponteiro para double*), que já foi codificado.

4.4.5 Codificação de Campos

Os tipos cujo operador é `_STRUCT` ou `_UNION` têm campos. Os campos, no *arquivo xingó*, ficam entre "{" e "}".

As informações codificadas de um campo são: nome, *offset*, a quantidade de *bits* (caso seja um campo de *bits*) e tipo. Se não for um campo de *bits*, a quantidade de *bits* é definida como zero. Por exemplo, a seguinte declaração:

```
struct campos{
int x;
int y;
unsigned b:1;
float f;
};
```

seria codificada como:

```
0
50 9 16 4 ( campos )
```

```
{
x 0 0 0
51 5 4 4 ( int )
y 4 0 51
b 8 1 0
52 6 4 4 ( unsigned )
f 12 0 0
53 1 4 4 ( float)
}
```

No caso, o tipo *struct campos* está sendo codificado pela primeira vez. Isso pode ser observado pelo *caracter* "0" na primeira linha. Esse tipo *struct* recebe o identificador 50. As informações sobre o tipo struct campos são: 9 (informa que é um *struct*); 16 (quantidade de *bytes* ocupados pelo tipo); 4 (alinhamento do tipo).

O campo x tem *offset* 0, não é um campo de *bits* (valor 0 na informação quantidade de *bits*) e tem tipo *int*, que está sendo codificado pela primeira vez e recebeu o identificador 51. y tem *offset* 4, também não é um campo de *bits*, e tem tipo *int*. Observe como foi emitido apenas o identificador do tipo *int* (51). O campo b tem *offset* 8, é um campo de *bits* (1 *bit*) e tem tipo *unsigned* (recebeu identificador 52), codificado pela primeira vez. f tem *offset* 12, não é um campo de *bits* e tem tipo *float*.

4.4.6 Codificação de Tipos de Parâmetros

Para codificar informações sobre tipos de parâmetros, a primeira informação emitida diz se a função tem quantidade de argumentos variável. Se ela tiver, é armazenado no *arquivo xingó*, o valor 1. Caso contrário, o valor zero. Em seguida, são listados os tipos dos parâmetros. Os tipos dos parâmetros ficam entre "(" e ")". Por exemplo, a conhecida função `printf` tem o seguinte protótipo: `int printf(const char* fmt, ...)`. Os tipos dos seus parâmetros seriam codificados como segue:

```
1 ( 0 50 7 4 4 ( pointer ) 0
51 15 1 1 ( const ) 0
52 3 1 1 ( char ) )
```

O primeiro *caracter* que aparece, "1", informa que é uma função com quantidade de argumentos variável.

Uma outra função, por exemplo: `int soma(int a, int b)` teria os tipos dos parâmetros codificados da seguinte maneira:

```
0 ( 0 50 5 4 4 ( int )
50 )
```

Nesse caso, o primeiro *caracter* que aparece é "0", ou seja, não é uma função com quantidade de argumentos variável.

4.5 O Decodificador Xingó

O Decodificador Xingó percorre o *arquivo xingó* e monta as estruturas de dados da representação XIR. Para montar a representação XIR, são utilizados os métodos públicos descritos no anexo A.

A seguir, são descritas aquelas decodificações que envolvem uma complexidade maior.

4.5.1 Decodificação de Variáveis

O decodificador reconhece a declaração de uma variável quando ele encontra uma informação declarativa `D_VARIABLE`, `D_PARAM` ou `D_LOCAL`.

No caso de `D_PARAM` e `D_LOCAL` é simples, a própria informação declarativa contém todas as informações sobre a variável. Basta criar um objeto `Variable` a partir das informações decodificadas, e adicioná-lo na lista de parâmetros do procedimento corrente ou na tabela de símbolos local.

No caso de `D_VARIABLE` é mais complexo. Primeiro por que o objeto `Variable` referente a essa informação declarativa já pode ter sido criado, caso esta variável tenha sido usada antes de ter sido declarada. Se ela é uma variável que precisa ser exportada a outros módulos, antes de `D_VARIABLE` vai existir uma informação declarativa `D_EXPORT`, que já cria o objeto `Variable` e o adiciona na tabela de símbolos global. Considere agora o caso da variável ser global mas não inicializada. Ela é declarada no *arquivo xingó* somente

no segmento de variáveis não inicializadas, que pode ser emitido depois do código dos procedimentos. Acontece que um procedimento pode usar essa variável antes da sua declaração. Toda vez que um símbolo aparece numa operação, se ele não possui ainda um objeto correspondente em XIR, um objeto associado ao símbolo é criado e adicionado na tabela de símbolos que deve contê-lo. Depois, quando a declaração deste símbolo for feita, as informações a seu respeito que só aparecem na sua declaração são atualizadas.

4.5.2 Decodificação de Jump Tables

Um outro caso mais complexo é a decodificação de *jump tables*. No compilador Xingó, representa-se uma *jump table* através de uma variável, que no Código Abstrato Xingó está associada a uma informação declarativa `D_VARIABLE`.

Reconhece-se que uma informação declarativa representa uma *jump table* pelas seguintes razões:

- ela é declarada sempre dentro do segmento lógico `LIT`, dado pela informação declarativa `D_LIT`;
- neste caso, a informação `D_LIT` vem sempre imediatamente após uma informação de operação JUMP. Além disso, vai existir uma troca de segmentos, de CODE para LIT. Depois da declaração da *jump table*, a troca é desfeita, e o segmento lógico volta a ser CODE.

A variável que representa uma *jump table* é inicializada com os endereços para os quais a instrução JUMP pode saltar. O problema de *jump tables* aparece na geração de código C. Acontece que o endereço alvo da operação JUMP é calculado dinamicamente, por isso é que existe a *jump table*. Este endereço é dado por algo semelhante a: aponta para o endereço base da variável que representa a `jump table`, soma a este endereço um *offset*, lê da memória o conteúdo endereçado pela soma, e salta para este endereço. O *offset* serve de índice para o elemento da tabela que dá o endereço para onde a execução deve desviar. Até agora tudo normal, pois é isso mesmo que o compilador deve fazer. Infelizmente, fazer um "*goto*" em C (operação JUMP), só é possível se o endereço alvo do desvio puder ser resolvido estaticamente, ou seja, se o *label* alvo do desvio puder ser conhecido antecipadamente, o que não é o caso de *jump tables*.

Para solucionar o problema da geração de código C para *jump tables*, foi criada uma solução que permite tanto ao gerador de código *assembly* quanto ao gerador de código C trabalharem sobre a mesma seqüência de instruções na representação XIR. Como visto no capítulo 3, uma variável e uma instrução possuem anotações. Assim, associa-se uma anotação `NOTE_JUMP_TABLE` à variável que representa a *jump table* e também à instrução JUMP que salta para um dos endereços armazenados nessa variável. O gerador de código C e o gerador de código *assembly* vão implementar uma *jump table* de maneiras diferentes, no entanto, corretas.

A idéia é que o gerador de Código C construa um bloco *switch/case* onde as cláusulas sejam índices que se associam aos *labels* contidos em `NOTE_JUMP_TABLE`. Por exemplo, "case 0: goto L1" é uma cláusula criada para saltar para o *label* L1, primeiro *label* da *jump table*. "case 1: goto L2" é uma cláusula criada para saltar para o *label* L2, segundo *label* da *jump table*, e assim por diante. Para isso, a variável que representa a *jump table* deve ser tratada como um "vetor de inteiros", cujos elementos são estes números que correspondem às cláusulas. Assim, a operação JUMP é transformada em um bloco *switch/case*, e o elemento indexado no vetor é o valor que seleciona a cláusula *case* correspondente ao *label* desejado.

4.5.3 Decodificação de Procedimentos Struct

A decodificação de um procedimento começa sempre com uma informação declarativa `D_PROC`, `D_PARAM` ou `D_LOCAL`. Ela só irá começar com `D_PROC` se o procedimento não possuir parâmetros nem variáveis locais.

No início da decodificação de um novo procedimento, é criado um objeto `Procedure` para representar este procedimento em XIR. Os símbolos locais são inseridos na tabela de símbolos local, e os parâmetros são inseridos na lista de parâmetros do procedimento.

Procedimentos que retornam uma estrutra colocam uma complicação a mais para serem gerados. O LCC apresenta duas opções para trabalhar com procedimentos que retornam uma estrutura. Na primeira, ele cria um argumento extra, e o passa ao procedimento através da pilha. Neste caso, há uma mudança na declaração formal dos parâmetros e do tipo do procedimento, que passa a retornar *void*. Na segunda, é convencionado que a primeira variável local do procedimento aponta para o endereço onde se deve armazenar o resultado de retorno. O LCC não atualiza o ponteiro dessa variável local, ou seja,

se o gerador de código não o fizer, ela fica apontando para um endereço inválido. Na implementação do compilador Xingó, optou-se pela segunda solução[2].

Para garantir a geração correta de código para um procedimento que retorna uma estrutura, é criada uma variável temporária do mesmo tipo retornado pelo procedimento. Então, de forma forçada, é gerada uma instrução que faz a primeira variável local do procedimento receber o endereço desta variável temporária. Esta instrução é a primeira instrução do procedimento. Esta solução foi testada para diversos programas e para todos foi gerado código C correto.

4.5.4 Decodificação de Tipos

A decodificação de tipos é simples. Se o tipo já foi decodificado anteriormente, existe uma instância de `Type` associada a ele, e nada precisa ser feito, a não ser referenciar esta instância. Se o tipo está sendo declarado pela primeira vez, é criado um objeto `Type` para representar o seu operador. Este objeto é inserido na tabela de símbolos global. Se o tipo tem operando, e este operando já foi decodificado, simplesmente faz-se o tipo operador, recém instanciado, apontar para o tipo que representa o operando. Se o operando também está sendo declarado pela primeira vez, é criado um objeto `Type` para ele e faz-se o tipo operador apontar para este objeto, que também é inserido na tabela de símbolos globais. A decodificação de tipos segue desta maneira, recursivamente.

4.5.5 Decodificação de Operações

No final da seção 4.4, foi dito que as operações no *arquivo xingó* estão baseadas numa representação em notação pós-fixa e que a decodificação dessas operações poderia ser realizada através de uma pilha. Para decodificar as operações, é utilizada uma pilha que auxilia na montagem das instruções da representação XIR.

A lógica na decodificação das operações é simples. Consulta-se o código da operação, com isso é possível saber quantos operandos devem ser desempilhados, e depois, caso seja necessário, empilha-se de volta o operando que recebe o resultado da instrução. Pelo fato da representação XIR ser baseada em quádruplas, todos os operandos devem ter nomes

[2]Usando-se a primeira solução, diminui a semelhança entre o código C de entrada e o código C gerado, por causa da mudança na declaração formal do procedimento.

explícitos. Desse modo, variáveis temporárias são criadas para receber a computação de resultados parciais de expressões.

4.6 Configuração do *Front-End*

Para adicionar ao Xingó um gerador de código para uma nova máquina alvo, o desenvolvedor deve criar uma nova interface de configuração do *front-end*. Essa interface é bastante simples. Ela define métricas para os tipos, algumas *flags* para montar corretamente o código intermediário, e as funções da interface de geração de código que devem ser chamadas pelo *front-end*.

O desenvolvedor precisa também registrar essa nova interface. O registro de uma interface é feito no arquivo "`bind.c`", que faz parte do pacote do LCC.

Até o presente momento (janeiro de 2005), o Xingó define duas interfaces de configuração, uma para geração de código C compilável e outra para geração de código para o processador MIPS R3000. Na figura 4.5 é mostrada a configuração da interface para a geração de código C compilável. A configuração da interface MIPS R3000 é detalhada no capítulo 6.

O desenvolvedor não precisa reimplementar as funções da interface de geração de código, pois elas já foram implementadas para gerar corretamente a representação XIR para qualquer visão posterior do código, seja código C ou código *assembly*.

4.6.1 Definição de Métricas de Tipos

Para cada tipo primitivo existe uma métrica que especifica o tamanho e o alinhamento do tipo. Existe ainda um *flag* que, quando setado com o valor 1, não permite que constantes do tipo apareçam em DAGs. Como um operando imediato de uma instrução em XIR tem que ser inteiro, é permitido que apenas constantes do tipo *char*, *short* e *int* tenham esse *flag* igual a zero.

De acordo com a especificação do LCC, o tamanho e alinhamento para caracteres (tipo *char*) deve ser 1. Um ponteiro deve caber em um inteiro não sinalizado (*unsigned int*). O alinhamento de uma *struct* é o máximo entre os alinhamentos de seus campos e o alinhamento definido na métrica. Por isso, define-se o alinhamento da métrica para *struct* como sendo 1.

As macros `sizeof` e `__alignof__` são utilizadas para automaticamente fornecer o tamanho e o alinhamento do tipo quando o código alvo for código C compilável. Isso facilita a redirecionabilidade, pois essas informações não precisam ficar sendo alteradas manualmente.

```
Interface xingoC = {
//type metrics
sizeof(char),          __alignof__(char),         0,  // char
sizeof(short),         __alignof__(short),        0,  // short
sizeof(int),           __alignof__(int),          0,  // int
sizeof(long),          __alignof__(long),         1,  // long
sizeof(long long),     __alignof__(long long),    1,  // long long
sizeof(float),         __alignof__(float),        1,  // float
sizeof(double),        __alignof__(double),       1,  // double
sizeof(long double),   __alignof__(long double),  1,  // long double
sizeof(void*),         __alignof__(void*),        1,  // pointer
0,                     1,                         1,  // struct
//interface flags
0,                     // little_endian
1,                     // mulops_calls
1,                     // wants_callb
1,                     // wants_argb
1,                     // left_to_right
0,                     // wants_dag
//interface functions
I(address),
I(blockbeg),
I(blockend),
I(defaddress),
I(defconst),
I(defstring),
I(defsymbol),
I(emit),
I(export),
I(function),
I(gen),
I(global),
I(import),
I(local),
I(progbeg),
I(progend),
I(segment),
I(space),
};
```

Figura 4.5: Interface para geração de código C compilável

4.6.2 Definição de Flags da Interface

As *flags* da interface configuram o *front-end* para montar código intermediário correto para a máquina alvo.

A primeira *flag* diz se a máquina é *little* ou *big endian*. Se o valor desta *flag* é 1, a máquina é *little endian*, caso contrário, *big endian*. A máquina que está sendo utilizada para realizar os testes de geração de código C é *big endian*.

A *flag mulops_calls* deve ser zero se o *hardware* implementa multiplicação, divisão e resto de divisão. Se o *hardware* deixa estas operações para rotinas de biblioteca, o valor desta *flag* deve ser 1.

A *flag wants_callb* diz se o *front-end* pode emitir código para uma função retornar uma estrutura (*struct* ou *union*). O desenvolvedor deve sempre considerar o valor desta *flag* como sendo 1, pois a Geração de Código Intermediário Xingó foi implementada considerando apenas essa situação.

A *flag wants_argb* diz se o *front-end* pode emitir nodos para passar argumentos em forma de estrutura. Pelo mesmo motivo anterior, o desenvolvedor deve colocar o valor dessa *flag* como sendo 1.

A *flag left_to_right* diz ao *front-end* que os argumentos devem ser avaliados e apresentados da esquerda para a direita. Se *left_to_right* é zero, argumentos são avaliados da direita para esquerda. Toda nova interface deve considerar *left_to_right* como sendo igual a 1, pois o Gerador de Código Intermediário Xingó avalia o código da esquerda para a direita.

A *flag wants_dag* diz ao *front-end* se o Codificador Xingó quer receber DAGs. Se o valor desta *flag* é zero, o *front-end* não cria um novo DAG para os nodos com quantidade de referências acima de um. Ele cria um temporário, atribui o nodo para o temporário, e usa o temporário toda vez que o nodo for referenciado. Quando *wants_dag* é zero, a quantidade de referências para todos os nodos é zero ou um, portanto, somente árvores são passadas ao Codificador Xingó. O desenvolvedor poderia sempre definir esta *flag* como sendo igual a zero.

No restante da interface de configuração, apenas referencia-se as funções que implementam a interface de geração de código do LCC. Essa parte é sempre igual, pois as funções são sempre as mesmas, e já estão implementadas.

Capítulo 5

Infra-Estrutura de Geração de Código no Xingó

O aparecimento de novos processadores, cada vez mais especializados, tem exigido que os compiladores sejam capazes de gerar código altamente otimizado e que possam ser readaptados rapidamente para gerar código para uma nova arquitetura. De modo a acompanhar a evolução dos processadores, novos geradores de código devem ser desenvolvidos e testados. Para isso, foi desenvolvida uma infra-estrutura simples, modular e com parametrizações[1], que possibilita novas pesquisas na área de geração de código.

O compilador Xingó é um projeto que tem como um de seus objetivos facilitar a construção de novos geradores de código, de modo que o esforço necessário para gerar código para uma nova arquitetura seja reduzido. Compiladores que apresentam esta característica têm sido denominados compiladores redirecionáveis.

Para facilitar o processo de geração de código, o uso de ferramentas que realizam síntese automática de geradores de código aparece como uma solução interessante no desenvolvimento de compiladores redirecionáveis. Como parte deste trabalho, a ferramenta Olive foi integrada ao compilador Xingó. A partir de Olive [45] não é possível descrever todos os detalhes do processador alvo, tais como a descrição formal de locais de armazenamento, modos de endereçamento ou via de dados (*datapath*) do processador. Contudo, é possível descrever padrões que mapeiam a representação de código interna do compilador para o conjunto de instruções do processador alvo. Tais padrões são descritos utilizando uma

[1] utilização de parâmetros de configuração.

gramática com regras customizáveis, o que viabiliza uma seleção de instruções otimizada.

A inclusão de Olive no compilador contribui de forma significativa no processo de construção de novos *back-ends*. Esse novo *back-end* pode ser integrado ao restante do compilador sem muito esforço. Via de regra, para gerar código para um novo processador, o desenvolvedor deve escrever uma gramática Olive para a arquitetura alvo e implementar algumas classes específicas para representar as operações do código *assembly*. A implementação destas classes é relativamente simples, pois elas são derivadas de classes genéricas fornecidas pela infra-estrutura de geração de código do compilador.

Neste capítulo, será descrita a infra-estrutura de geração de código do compilador Xingó. Na última seção deste capítulo, será apresentado um roteiro que explica em detalhes como construir um novo gerador de código para o compilador. Espera-se que esta infra-estrutura possa encorajar novas pesquisas em geraçaõ de código. No capítulo 6 é apresentado o gerador de código para o processador MIPS R3000, que foi desenvolvido usando a infra-estrutura descrita aqui.

5.1 Estrutura do *Back-End*

Durante o processo de compilação, é comum o código de um programa estar representado de formas diferentes, dependendo da fase de compilação e das técnicas utilizadas [10]. Como visto no capítulo 4, o programa em LIR — Representação Intermediáia do LCC — baseada em grafos, é traduzido para um programa em XIR — Representação Intermediária Xingó — que apresenta uma estrutura linear baseada em quádruplas.

Antes de chegar à LIR, entretanto, o código já havia sido representado por ASTs e DAGs. Dentro do compilador Xingó, além da representação intermediária principal (XIR), em certas etapas, o programa pode ainda estar representado por árvores binárias de expressões (ASTs) ou por uma representação de código *assembly*. Foram implementadas estruturas de dados, através de classes em C++, que representam o código nestas diferentes formas, bem como os módulos que convertem o código de uma representação origem para uma representação destino, seguindo o fluxo normal de compilação.

No processo de geração de código para a máquina alvo, existem quatro representações de programa a se considerar: *XIR*, *XOR*, *XAR* e *Assembly Code*. Além disso, deve-se considerar outros três módulos que fazem a tradução de uma representação origem para

uma representação destino: *XOR Generator*, *Code Generator* e *Emit Assembly Code*, como mostrado na figura 5.1.

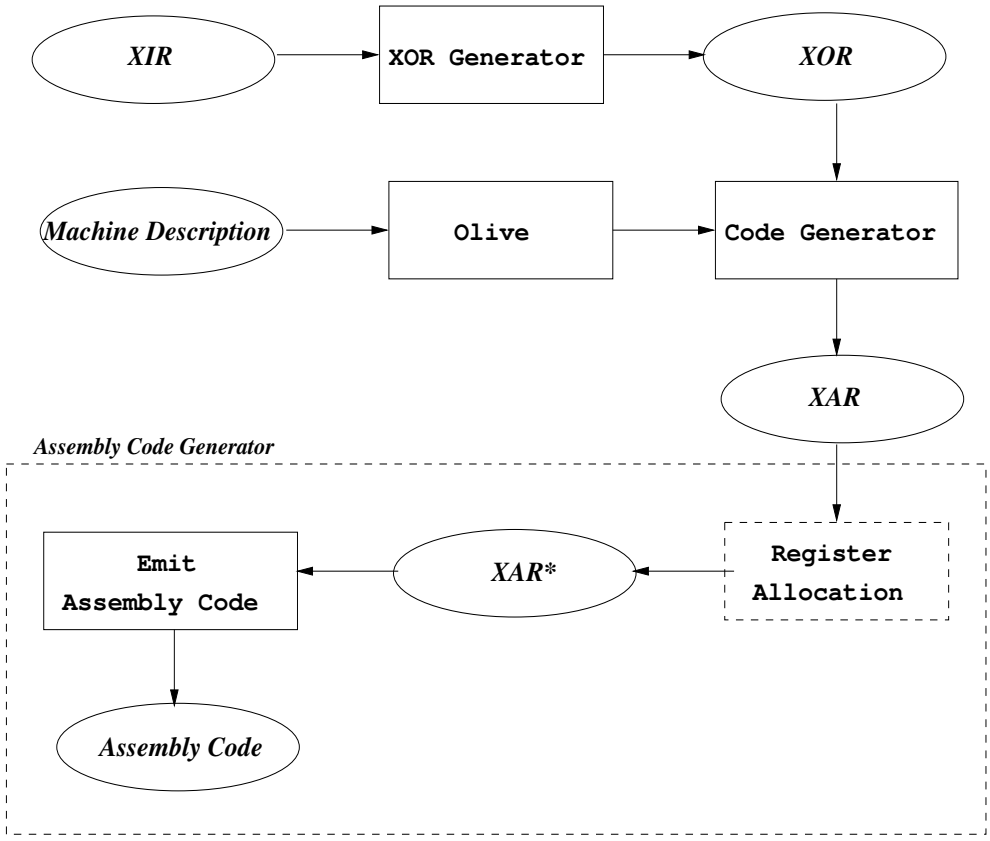

Figura 5.1: Contexto do back-end Xingó

Primeiramente, um programa representado em XIR é traduzido em uma representação denominada *XOR* — Representação Olive Xingó, ou simplesmente árvores para o Olive. Esse processo é realizado por meio de um módulo denominado *XOR Generator*. A representação *XOR* armazena as operações do programa sob a forma de árvores binárias. Tais árvores são entrada para um gerador de código sintetizado por Olive (*Code Generator*), que produz como saída um código para a representação *XAR — Representação Assembly Xingó* — que, inicialmente, utiliza registradores virtuais para computar resultados de expressões. O módulo *Register Allocation* aloca registradores físicos para os registradores virtuais. Este módulo, no entanto, preserva a estrutura da XAR. Por último, o módulo *Emit Assembly Code* caminha sobre a representação XAR, gerando como saída,

um arquivo que representa o programa *assembly* para a máquina alvo.

Os módulos e as representações de programa ilustrados na figura 5.1 e utilizados no contexto do *back-end* Xingó estão descritos abaixo:

- ***XIR*:** Representação Intermediária Xingó (ver capítulo 3), que possui uma estrutura linear baseada em quádruplas.
- ***XOR Generator:*** converte o programa representado em XIR para a representação XOR.
- ***XOR:*** Representação das expressões do programa em compilação sob a forma de árvores binárias.
- ***Machine Description:*** Gramática Olive que descreve as regras para mapear as árvores de expressões representadas em XOR para o conjunto de instruções representadas em XAR.
- ***Olive:*** sintetiza um módulo de geração de código a partir das regras gramaticais descritas em *machine description*.
- ***Code Generator:*** Seleciona instruções para a máquina alvo, empregando a técnica de casamento de padrões. O código a ser emitido deve ser descrito no bloco de ação de cada regra gramatical em *machine description*.
- ***XAR:*** Representação do código *assembly* para a máquina alvo gerada por *Code Generator*. Nesta representação, é assumido que a máquina alvo possui um banco de registradores infinito. Assim, os registradores selecionados podem ser virtuais.
- ***Register Allocation:*** Realiza alocação de registradores para o código *assembly* gerado por *Code Generator*.
- ***XAR*:*** Representação XAR depois de passar pela alocação de registradores. Aqui, os registradores selecionados já são todos físicos, adequados às convenções da arquitetura alvo.
- ***Emit Assembly Code:*** Armazena em um arquivo de saída, o código *assembly* final.
- ***Assembly Code:*** Código assembly final para a máquina alvo.

5.2 Olive

Olive [45] é um gerador de geradores de código (do inglês, *code-generator generator*) que utiliza uma descrição gramatical (gramática Olive) do conjunto de instruções do processador alvo para produzir automaticamente o módulo de geração de código do compilador.

Olive foi desenvolvido por Steve Tjiang, e é baseado nas ferramentas de síntese de geradores de código Twig [3] e Iburg [15][16]. Em Olive, o conjunto de instruções do processador é descrito usando-se um conjunto de regras gramaticais, onde cada regra pode levar a uma ou mais instruções do processador alvo. Regras também podem, simplesmente, calcular o operando de uma instrução. Os terminais da gramática Olive estão relacionados com as operações na representação intermediária do compilador, enquanto os não-terminais são elementos de armazenamento (registradores ou memória) na via de dados do processador. A cada símbolo terminal é atribuído um valor numérico, enquanto aos não-terminais são associadas funções em C, que podem receber parâmetros e retornar dados. Uma produção, ou regra, da gramática Olive tem o seguinte formato:

location : tree-pattern { pattern matching } = { action }

onde:

- *location* é um elemento de armazenamento que guarda o resultado computado pela instrução;
- *tree-pattern* é uma seqüência de terminais e não-terminais que determinam um padrão de árvore que pode aparecer na representação intermediária do compilador;
- *pattern matching* é um trecho de código escrito na linguagem C que calcula o custo de casar *tree-pattern* a uma determinada árvore de expressão;
- *action* é um trecho de código na linguagem C, que emite as ações necessárias se a regra for selecionada. Por exemplo, emite uma instrução *assembly*.

Cada regra possui dois trechos de código a serem avaliados, sendo estes contidos entre chaves e separados por um sinal de igual. O primeior trecho (*pattern matching*) é usado para determinar o custo de se casar o *tree-pattern* com uma árvore de expressão do programa. O segundo trecho (*action*) irá emitir uma instrução se esta regra for selecionada. A regra selecionada é aquela que tem o menor custo. Regras de produção são

selecionadas por meio de uma versão modificada do algoritmo de programação dinâmica de Aho-Johnson [1]. Regras são casadas com árvores de expressão do programa usando como custo de casamento, por exemplo, o número de ciclos de execução das instruções.

A fim de demonstrar melhor a estrutura de uma produção da gramática Olive, é ilustrado, na figura 5.2, um exemplo de uma regra que descreve uma instrução para somar dois valores inteiros. Na figura 5.3, é ilustrada uma árvore de expressão com a qual esta regra casa. Considere que a árvore foi gerada a partir da expressão $a \times 2 + a \times 2 \times b$, e que a e b sejam variáveis do tipo *int* em C.

Todo nó de uma árvore Olive no compilador Xingó possui um *opcode* numérico (terminal da gramática), definido por uma constante formada por um prefixo e um sufixo. No caso, os prefixos são `nADD`, `nMUL`, `nVAR` e `nCON`, e o sufixo é `I`. Os prefixos indicam um operando ou uma operação. `nADD` representa uma operação de adição e `nMUL` representa uma operação de multiplicação. `nVAR` representa um operando que corresponde a uma variável e `nCON` representa um operando que corresponde a uma constante. Os sufixos determinam o tipo do dado manipulado (*int*, *float*, *long*, *struct*, etc). O sufixo `I` indica um valor de tipo *int*.

Os símbolos da gramática são numerados da esquerda para a direita, começando com 0 para o não-terminal `reg` à esquerda do sinal de dois pontos (figura 5.2). O terminal `nADDI`, que tem índice 1 na regra, é o *opcode* do nodo raiz da árvore que casa a produção `reg`. O *opcode* `nADDI` informa uma operação que soma dois operandos e produz um resultado de tipo inteiro (*int* em C). Os não-terminais `reg` e `rc` à direita do sinal de dois pontos correspondem aos operandos da soma. Estes dois não-terminais têm, respectivamente, índice 2 e 3. Eles, na verdade, representam duas sub-árvores, apontadas pelo nodo raiz (`nADDI`) da árvore de expressão. `reg` representa um registrador e `rc` representa um registrador ou uma constante (um valor imediato). Ao lado de cada nó da árvore na figura 5.3 são colocadas as regras que fazem o casamento de cada subárvore da expressão $a \times 2 + a \times 2 \times b$.

```
reg:  nADDI(reg , rc)
{
   $cost[0].cost = 1 + $cost[2].cost + $cost[3].cost;
}=
{
   AsmRegister* r1 = new AsmRegister(TheRegAlloc->NewVirtualReg(IREG));
   AsmRegister* r2 = $action[2](aol);
   AsmOperand* op = $action[3](aol);
   AsmOperation* ao = new AsmOperation(aADD, r1, r2, op);
   aol->Append(ao);
   return r;
};
```

Figura 5.2: Regra Olive de uma operação de soma de inteiros

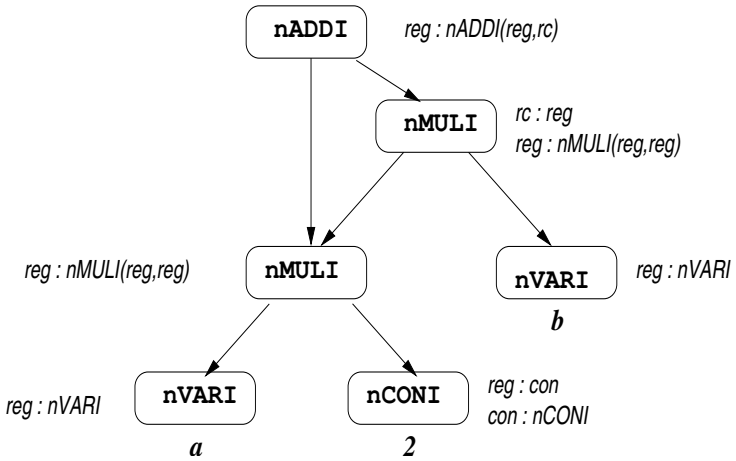

Figura 5.3: Árvore Olive para a expressão $a \times 2 + a \times 2 \times b$

A estrutura `$cost`, na primeira parte entre chaves da regra (figura 5.2), é usada para computar o custo de se casar a árvore de expressão com raiz no não-terminal `reg` à esquerda do sinal de dois pontos. Se esta regra casa, com menor custo, a raiz de alguma árvore que tenha este padrão, então o código na segunda parte entre chaves é executado. A função `action` escalona as sub-árvores com raiz nos não-terminais `reg` e `rc` à direita do sinal de dois pontos. A chamada a `$action[2](aol)` retorna um registrador, enquanto a chamada a `$action[3](aol)` retorna um registrador ou uma constante. No fim da ação da regra, é criada uma operação `aADD`, que corresponde à instrução a ser emitida se esta regra for selecionada para casar a árvore de expressão. Essa operação é acrescentada à lista de instruções *assembly* (`aol`) do programa alvo.

Olive foi incorporado ao compilador Xingó para facilitar o processo de geração de código para uma nova arquitetura alvo. Olive transforma uma descrição da máquina alvo em um módulo de geração de código para esta máquina. No Xingó, este módulo faz simplesmente a seleção de instruções para a máquina alvo. Vale lembrar que uma descrição Olive do processador alvo especifica somente o modelo do conjunto de instruções da arquitetura como vista pelo programador, sem os detalhes internos da microarquitetura.

5.3 Representação XOR

Como visto no capítulo 3, a representação intermediária principal do compilador Xingó (XIR) é baseada em uma estrutura linear, na forma de quádruplas. O compilador Xingó utiliza Olive para produzir automaticamente o gerador de código do compilador. O gerador de código sintetizado por Olive caminha sobre uma representação de programa baseada em árvores binárias para fazer o casamento de padrões. Assim, foi necessário implementar estruturas de dados para representar as operações do programa em uma forma conveniente para o Olive.

A representação XOR mantém as operações do programa sob a forma de Árvores Sintáticas Abstratas (ASTs). A representação XOR não modifica a estrutura de representação de todas as informações da XIR, mas somente a estrutura das informações de operações, ou seja, apenas as instruções de cada procedimento passam a ser representadas por ASTs, ao invés de quádruplas. As informações simbólicas, que inclui tabelas de tipos, de variáveis e de parâmetros, continuam representadas da mesma forma.

A representação XOR é constituída de um conjunto de árvores. Cada árvore representa uma expressão formada a partir de um conjunto de instruções da XIR. As árvores na representação XOR, denominadas *Árvores Olive*, são construídas a partir de objetos da classe `OTreeNode`. Esta classe implementa um nodo da árvore. Cada nodo possui apontadores para outros dois objetos da classe `OtreeNode`. Com isso, é possível construir uma árvore completa apenas ligando os nodos. Além dos apontadores para os nodos filhos, um objeto `OTreeNode` possui um *opcode* numérico que informa qual a operação realizada pelo nodo. Nodos folhas[2] correspondem aos operandos da expressão que deu origem à árvore, e, neste caso, o *opcode* de um operando diz se ele é uma variável ou um valor imediato. Assim, um nodo de `OTreeNode` possui também apontadores para um objeto `Variable` e `Immed` da XIR.

Os *opcodes* são armazenados no tipo enumerado `OliveTerm`. A tabela 5.1 lista todos os *opcodes* de uma *árvore Olive*. Um *opcode* possui um prefixo, que informa o tipo da operação, e um sufixo, que informa o tipo do dado retornado pela operação. Por exemplo, a operação `nADDI` tem prefixo nADD e sufixo I. O operador genérico nADD ainda tem as variantes `nADDU`, `nADDP`, `nADDF` e `nADDD`. Para efeito de ilustração, prefixo e sufixo serão separados. A coluna *Operador* informa o prefixo do *opcode*, e a *Tipo do Sufixo* determina os sufixos possíveis para cada operador. Estes sufixos estão relacionados na tabela 5.2. A coluna *Filhos* informa a quantidade de filhos que um nodo com o *opcode* referenciado tem. A última coluna diz o significado de cada operação.

O tipo do dado computado pelo nodo e o número do registrador virtual a ele atribuído são outras informações armazenadas em cada objeto da classe `OtreeNode`. Para os operandos que correspondem a um parâmetro ou uma variável global, o valor do registrador virtual é -1. Aos demais operandos, são atribuídos registradores virtuais com valor maior que zero.

O sufixo de uma operação de conversão de tipos é um caso especial. É necessário saber o tipo origem e o tipo destino da conversão. Neste caso, o sufixo agrega os dois tipos requeridos. Por exemplo, `nCVRTIC` expressa uma operação que converte um valor de *int* (I) para *char* (C).

[2]Nodos folhas são aqueles que não possuem filhos.

Operador	**Tipo do Sufixo**	**Filhos**	**Significado**
`nLOAD`	C-S-I-U-P-L-F-D-T	1	leitura da memória
`nSTORE`	C-S-I-U-P-L-F-D-T	2	escrita na memória
`nMOVE`	C-S-I-U-P-L-F-D-T	2	cópia de operando
`nCVRT`	CI-CU-DI-DF-FD-IC-IS-IU ID-PU-SI-SU-UC-US-UI-FI-IF	1	conversão de tipos
`nADD`	I-U-P-F-D	2	adição
`nSUB`	I-U-P-F-D	2	subtração
`nMUL`	I-U-F-D	2	multiplicação
`nDIV`	I-U-F-D	2	divisão
`nAND`	U	2	AND *bit* a *bit*
`nOR`	U	2	OR *bit* a *bit*
`nXOR`	U	2	OR *bit* a *bit* exclusivo
`nNEG`	I-F-D	1	negação
`nLSR`	I-U	2	*shift* lógico para a direita
`nASR`	I-U	2	*shift* aritmético para a direita
`nLSL`	I-U	2	*shift* lógico para a esquerda
`nASL`	I-U	2	*shift* aritmético para a esquerda
`nARG`	I-U-P-F-D-T	2	argumento
`nMOD`	I-U	2	resto da divisão
`nCALL`	I-U-P-F-D-T-V	2	chamada de procedimento
`nRET`	I-P-F-D-T	1	retorno de procedimento
`nJUMP`	V	0	desvio incondicional
`nJE`	I-U-F-D	2	salta se igual
`nJNE`	I-U-F-D	2	salta se diferente
`nJLT`	I-U-F-D	2	salta se menor do que
`nJGT`	I-U-F-D	2	salta se maior do que
`nJLE`	I-U-F-D	2	salta se menor ou igual
`nJGE`	I-U-F-D	2	salta se maior ou igual
`nLABEL`	V	0	definição de *label*
`nADDR`	P	1	endereço de um símbolo
`nCOM`	U	1	complemento *bit* a *bit*
`nVAR`	C-S-I-U-P-L-F-D-T-X	0	uma variável
`nCON`	C-S-I-U-P-L-F-D	0	um imediato
`nNOP`	sem sufixo	0	sem operação

Tabela 5.1: Operadores de uma árvore Olive

Significado	**Sufixo**
char	C
short	S
int	I
unsigned	U
long	L
float	F
double	D
ponteiro para	P
struct ou union	T
procedimento	X
void	V

Tabela 5.2: Tipos de sufixos de um operador de um nodo em XOR

O operador `nNOP` é utilizado como uma espécie de delimitador de lista de argumentos. Por exemplo, numa chamada a um procedimento é necessário passar uma lista contendo todos os argumentos de entrada para a função. Essa lista deve ser representada através de uma árvore binária. Para tal adequação, é criado para cada argumento um nodo `nARG`. O filho esquerdo deste nodo é a raiz da sub-árvore que expressa este argumento. O filho direito representa outro nodo `nARG`, que corresponde ao próximo argumento da lista. O nodo `nARG` raiz da árvore corresponde ao *n-ésimo* argumento da função. O *(n-1)-ésimo* argumento é o segundo nodo `nARG` e assim por diante. O algoritmo considera sempre uma inclusão mais à direita na árvore, com os argumentos inseridos na ordem inversa. Quando a lista de argumentos terminar, deve ser inserido à arvore um nodo `nNOP`, para indicar o fim da lista. Se nenhum argumento precisa ser passado para a função, a árvore terá no nodo raiz o operador `nNOP`.

A figura 5.4 ilustra uma árvore exemplo para uma chamada a uma função. O nodo raiz da árvore é `nMOVEI`, que tem filhos `nVARI` e `nCALLI`. A variável temporária *.t23* recebe o retorno de *printf*[3]. `nVARX` corresponde à uma variável que está associada a um procedimento. No caso, esta variável é associada à função *printf*. `nCALL` aponta para o procedimento chamado e para a raiz da árvore que corresponde à sua lista de argumentos. A variável *.C3* é uma variável que aponta para a *string* de formatação — "%d %d %d".

[3]A função *printf* retorna um valor, que é a quantidade de *bytes* realmente escritos.

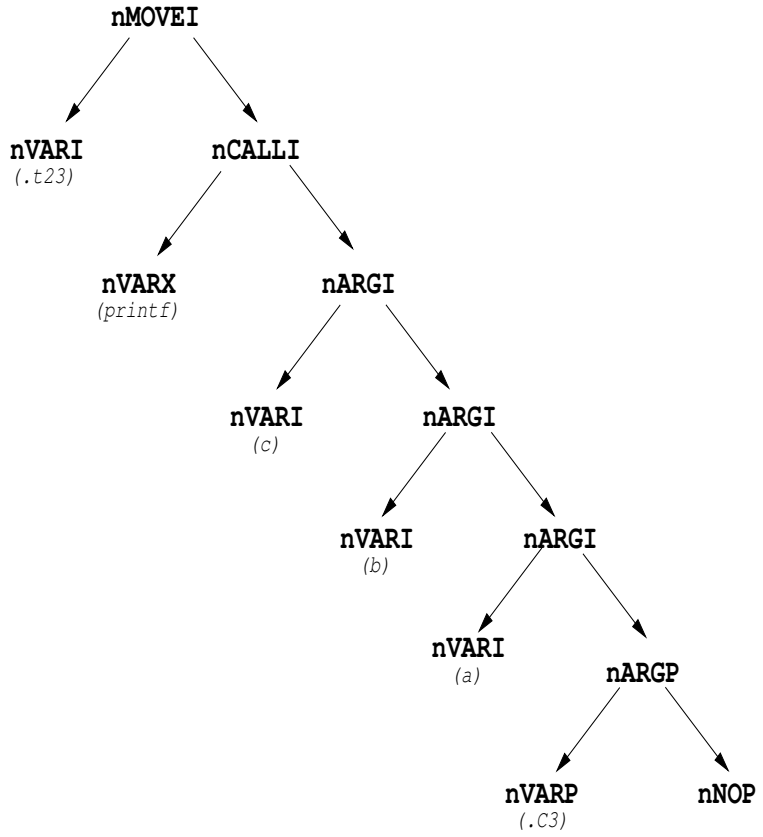

Figura 5.4: Árvore Olive para uma chamada à função `printf("%d %d %d", a, b, c)`

A figura 5.5 ilustra o exemplo de uma floresta de árvores Olive para a expressão em C `i = *p++`. A floresta possui duas árvores, cujas raízes são `nMOVEI` e `nMOVEP`. A primeira árvore avaliada pelo gerador de código é a árvore de raiz `nMOVEI`, que calcula $*p$ e atribui a i. Depois, o gerador avalia a árvore de raiz `nMOVEP`, que calcula $p{+}{+}$, atualizando p. Observe como o nodo `nVARP`, que corresponde à variável p, foi utilizado nas duas árvores.

Os operadores genéricos que podem aparecer como raiz de uma árvore Olive são: `nSTORE`, `nMOVE`, `nLABEL`, `nJUMP`, `nJE`, `nJNE`, `nJLT`, `nJGT`, `nJLE`, `nJGE` e `nRET`, e mais o operador específico `nCALLV`. Nas folhas da árvore estão sempre os operadores genéricos `nVAR`, `nCON` e `nNOP`. Os demais operadores só aparecem nos nodos intermediários da árvore.

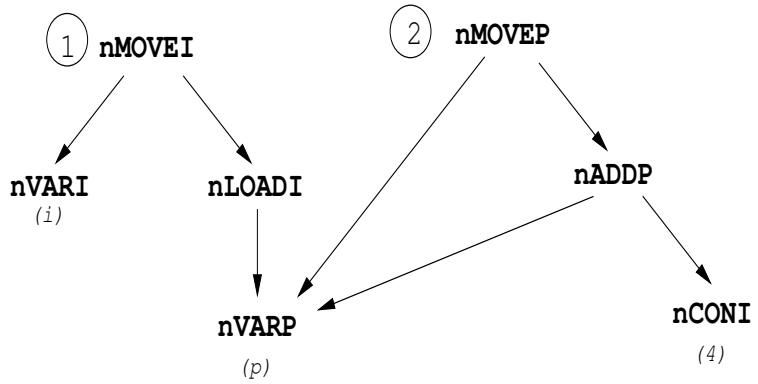

Figura 5.5: Floresta para int i, $*p$; f() { i = $*p$++; }

5.4 Geração da Representação XOR

O módulo *XOR Generator* transforma o código representado em XIR, baseada em quádruplas, para um código na representação XOR, baseada em ASTs.

O módulo *XOR Generator* não traduz todas as informações representadas em XIR, mas somente as informações de operações de cada procedimento. As informações simbólicas, que inclui tabelas de tipos, de variáveis e de parâmetros, continuam representados da mesma forma. Mais especificamente, *XOR Generator* percorre todos os procedimentos da XIR e gera uma floresta de ASTs para cada um dos procedimentos. As ASTs produzidas são então usadas como entrada do gerador de código sintetizado por Olive, que faz seleção de instruções para cada uma das árvores.

O processo de tradução de um programa na representação XIR para um programa na representação XOR é realizado pela função `GenerateOTree`. Abaixo segue o algoritmo desta função:

Algoritmo 5.1: GenerateOTree — Geração de árvores Olive
Entrada:
Lista L contendo as n-ésimas instruções de um procedimento.
Instruction i ← consome uma instrução de L
int VirtualReg ← número da *web* de i
escolha *opcode* de i

caso `LOAD, MOVE, MOVEI, MOVES, NEG, COM` ou `ADDR`
    Variable v ← operando destino de i
    OTreeNode dst ← **novo** OTreeNode `nVAR` (v)
    OTreeNode nodo ← make1op (i)
    Se VirtualReg ≠ −1
        hash→Inserir(VirtualReg, nodo)
        dst→VirtualReg ← VirtualReg
    nodo ← **novo** OTreeNode `nMOVE` (*dst* ← *node*)
    retornar nodo
caso `ADD, ADDI, SUB, SUBI, MUL, MULI, DIV, DIV, MOD, MODI, AND, ANDI`
  `OR, ORI, XOR, XORI, LSR, LSRI, ASR, ASRI, LSL, LSLI, ASL` ou `ALSI`
    v ← operando destino de i
    dst ← **novo** OTreeNode `nVAR` (v)
    nodo ← make2op (i)
    Se VirtualReg ≠ −1
        hash→Inserir(VirtualReg, nodo)
        dst→VirtualReg ← VirtualReg
    nodo ← **novo** OTreeNode `nMOVE` (*dst* ← *node*)
    retornar nodo
caso `CVRT`
    v ← operando destino de i
    dst ← **novo** OTreeNode `nVAR` (v)
    nodo ← makecvt (i)
    Se VirtualReg ≠ −1
        hash→Inserir(VirtualReg, nodo)
        dst→VirtualReg ← VirtualReg
    nodo ← **novo** OTreeNode `nMOVE` (*dst* ← *node*)
    retornar nodo
caso `CALL`
    nodo ← makecall (i)
    v ← operando destino de i
    Se v ≠ NULL

```
            dst ← novo OTreeNode nVAR (v)
            Se VirtualReg ≠ −1
                hash→Inserir(VirtualReg, nodo)
                dst→VirtualReg ← VirtualReg
            nodo ← novo OTreeNode nMOVE (dst ← node)
        retornar nodo
    caso STORE ou STOREI
        retornar makestore (i)
    caso JE, JEI, JNE, JNEI, JLT, JLTI, JGT, JGTI, JLE, JLEI, JGE ou JGEI
        retornar makejc (i)
    caso JUMP
        retornar makeuc (i)
    caso LABEL
        retornar makelabel (i)
    caso RET ou RETI
        retornar makeret (i)
```

`GenerateOTree` recebe como entrada uma lista (L) com as n-ésimas instruções de um procedimento na representação XIR e retorna o nodo raiz da árvore gerada. A lista L começa a partir da instrução que ainda não foi traduzida em árvore. Para produzir uma árvore, `GenerateOTree` consome um conjunto de instruções da lista de entrada. Pelo algoritmo acima, a função consome apenas uma instrução de L, mas as demais instruções são consumidas por funções auxialiares (`make1op, make2op, makestore, makejc, makejuc, makelabel, makeret, makecvt` e `makecall`) que produzem as sub-árvores que formam a AST. Para reaproveitar os nodos já inseridos na árvore, é mantida uma tabela *hash*[4] cujos elementos são raízes de sub-árvores pré-construídas.

Inicialmente, o algoritmo faz i receber a primeira instrução de L, e VirtualReg (registrador virtual) receber o número do registrador virtual associado à instrução. Um registrador virtual é dado por uma *web*, que será melhor referenciada na seção 5.7. O gerador de árvores Olive assume que cada *web* dará origem a um registrador virtual.

O algoritmo prossegue testando o *opcode* da instrução. Se o *opcode* entra no primeiro,

[4]O Xingó possui classes que implementam tabelas *hash* de elementos genéricos.

segundo ou terceiro casos, será criado para o operando destino de *i* um novo nodo (*dst*) na árvore. Este nodo tem operador genérico `nVAR`. No primeiro caso, a função auxiliar `make1op` é chamada para criar uma sub-árvore cujo nodo raiz possui um único filho. No segundo caso, `make2op` é chamada para criar uma sub-árvore cujo nodo raiz possui dois operandos filhos, isto é, aponta para outras duas sub-árvores. No terceiro caso (caso CVRT), `makecvt` é chamada para criar uma sub-árvore que faz conversão entre tipos. Nos três casos, se o registrador virtual é diferente de -1, o nodo retornado (variável *nodo*) por `make1op`, `make2op` ou `makecvt` é inserido na tabela *hash* e define-se o número do registrador virtual do nodo destino (*dst*). Por fim, é criada e retornada pela função, a raiz da árvore de expressão gerada. Esta árvore tem operador genérico `nMOVE` e filhos *dst* e *nodo*.

O caso `CALL` primeiro chama `makecall` para criar uma sub-árvore que reproduz a chamada ao procedimento expressado na instrução *i*. Depois, o algoritmo testa se a instrução tem uma variável destino. Ter uma variável destino significa que a função chamada retorna valor. Caso contrário, ela é uma função *void*. Se a variável destino existe, o trecho de código considerado realiza as mesmas ações dos casos anteriores — insere o nodo na tabela *hash* e define o registrador virtual do operando destino.

Os últimos cinco casos são mais simples, pois a instrução não possui variável destino e nem *web* associada. Logo, basta retornar a raiz da árvore gerada pela função auxiliar correspondente a cada caso.

As funções auxiliares `make1op, make2op, makestore, makejc, makejuc, makeret, makelabel, makecvt` e `makecall` criam sub-árvores para a instrução passada como entrada. Antes de criar um novo nodo na árvore, essas funções fazem uma consulta à tabela *hash* para verificar se já existe um nodo correspondente. Essa consulta é feita utilizando o número do registrador virtual associado ao operando da instrução. Se existe na tabela *hash* um nodo para o operando, basta simplesmente referenciar este nodo. Se não existe, um novo nodo é criado e inserido na tabela, para que possa ser reaproveitado pelas próximas árvores.

A floresta de ASTs gerada para cada procedimento deve ser passada como entrada do gerador de código sintetizado por Olive, para que ele faça a seleção de instruções e armazene-as na representação XAR (descrita na seção 5.5). O gerador de código seleciona código de uma única árvore por vez, e não de uma floresta inteira. Então, é necessário

chamar repetidas vezes a função que faz seleção de instruções, mais precisamente, para cada árvore na representação XOR. Para ocultar do desenvolvedor os detalhes da geração de árvores Olive e sua integração com o gerador de código (*Code Generator*), foi implementada a função `GenerateCode`. Para gerar código *assembly* na representação XAR para um determinado procedimento, o desenvolvedor precisa apenas chamar a função `GenerateCode`, ilustrada no algoritmo abaixo.

Algoritmo 5.2: GenerateCode — Geração de código *assembly*
Entrada:
Lista L contendo todas as instruções de um procedimento.
Criar um *array* para armazenar as raízes das árvores
Criar tabela *hash* para armazenar nodos referenciados múltiplas vezes
Enquanto L não estiver vazia
OTreeNode nodo ← GenerateOTree(L)
Adicionar nodo ao *array*
Criar uma lista *aol* de operações *assembly*
Enquanto não passar por todos os elementos do *array* **faça**
nodo ← próximo elemento do *array*
top_action(nodo, aol)
retornar aol

`GenerateCode` recebe como entrada uma lista (L) contendo as instruções de um procedimento na representação XIR e retorna uma lista (*aol*) contendo as instruções *assembly* geradas na representação XAR.

Inicialmente, são criados um *array*, para armazenar as raízes das árvores Olive geradas, e uma tabela *hash*, para armazenar os nodos referenciados múltiplas vezes. A função `GenerateOTree` é quem gerencia essa tabela *hash*. Enquanto não forem consumidas todas as instruções da lista, `GenerateOTree` é chamada para gerar árvores Olive para um grupo de instruções de L. A raiz da árvore retornada por `GenerateOTree` é adicionada ao *array*.

Depois de geradas todas as árvores para o procedimento, o *array* é percorrido, da primeira à última raiz das árvores, gerando código *assembly* para cada uma delas. A função `top_action` é a função do gerador de código chamada para fazer casamento de

padrões e seleção de instruções. Ela recebe como entrada a raiz da árvore e os argumentos da função que corresponde ao não-terminal de partida da gramática Olive. Como foi dito na seção 5.2, para cada não-terminal da gramática Olive, é associada uma função em C. Essa função pode retornar valor e receber parâmetros. De acordo com a implementação Xingó, é preciso que o desenvolvedor faça com que essas funções recebam como parâmetro uma lista de instruções *assembly*. Todas as funções, exceto a do não-terminal de partida, podem retornar qualquer tipo de dado, inclusive *void*. A função do não-terminal de partida deve, necessariamente, retornar um apontador para uma lista de instruções *assembly*.

5.5 Representação XAR

O código gerado por *Code Generator* não precisa ser, imediatamente, um arquivo *assembly* para a máquina alvo. Neste caso, é necessário que exista um código *assembly* preliminar que represente o programa na memória. Com isso, um módulo denominado *Register Allocation* pode então caminhar sobre as estruturas de dados das instruções *assembly* para realizar alocação de registradores, permitindo um melhor uso dos registradores da máquina. Além disso, tal representação pode permitir otimizações de código dependente de máquina.

Para facilitar alocação de registradores, foram implementadas estruturas de dados genéricas, por meio de classes abstratas, para representar o código *assembly*. Estas classes são organizadas hierarquicamente, assim como acontece na XIR. A figura 5.6 ilustra a estrutura da repr'esentação de código *assembly*.

A classe `AsmFile` representa a classe superior da hierarquia. Nesta classe existe um apontador para a classe `File` da XIR e uma lista de procedimentos *assembly*. Cada procedimento na representação XAR aponta para o procedimento correspondente na representação XIR e também para uma lista contendo as operações *assembly* que ele manipula. Uma operação *assembly* é implementada pela classe `AsmOperation`. Tal classe armazena o *opcode* da operação, os seus operandos e, possivelmente, outras informações caso sejam necessárias, como por exemplo, os registradores que a operação utiliza. Na figura 5.7 está ilustrada a representação de uma instrução *assembly* genérica no compilador Xingó.

Os operandos da instrução *assembly* são todos derivados da classe `AsmOperand`. Todos os operandos que a instrução *assembly* for utilizar devem ser implementados por

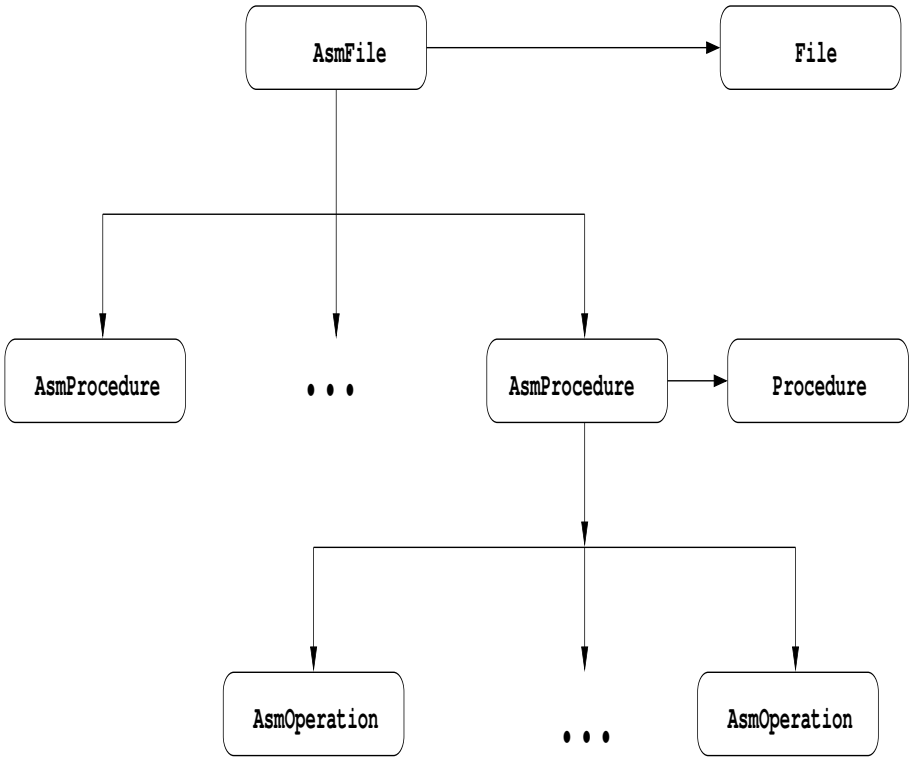

Figura 5.6: Estrutura da representação de código *assembly*

classes que herdam a classe `AsmOperand`. O Xingó define alguns operandos base como `AsmRegister`, `AsmVariable`, `AsmLabel` e `AsmImmediate`. De acordo com as necessidades impostas pela arquitetura alvo, este conjunto pode ser ampliado de forma a refletir a nova situação. De qualquer modo, para cada novo gerador de código, devem ser criadas classes que derivam estas classes bases. No capítulo 6, são apresentados detalhes da construção de um gerador de código para o processador MIPS R3000, e lá são descritas as classes que foram herdadas de `AsmOperand`.

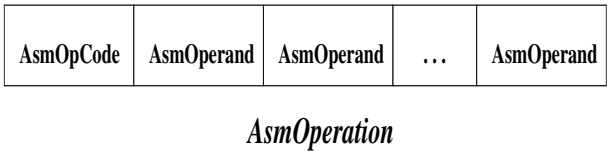

Figura 5.7: Representação de uma instrução *assembly* genérica

5.5.1 Configuração

O Xingó utiliza um arquivo chamado "asmconfig.h", que é usado para armazenar informações que são específicas da arquitetura alvo, tais como o *opcode* das instruções, a quantidade de registradores disponíveis, o tipo dos registradores (*int* ou *float*) e se os mesmos podem ser utilizados para alocação. É necessário que o desenvolvedor configure este arquivo para toda nova máquina alvo considerada. As configurações necessárias são:

- **Número máximo de operandos por instrução**
  Definir a constante `MAX_OPERANDS_PER_OPERATION`.

- **Classes dos operandos *assembly***
  Devem ser configuradas em `AsmOperandKind`, que é um tipo enumerado onde cada *tag* associa-se a uma das classes de operandos das instruções *assembly*.

- ***Opcodes assembly***
  Devem ser configurados em `AsmOpCode`, que é um tipo enumerado onde cada *tag* associa-se a um dos *opcodes* das instruções *assembly*.

- **Conjunto de instruções da máquina alvo**
  Deve ser configurado em `InstructioSetTable`, que é um vetor onde cada elemento tem a seguinte estrutura: nome da instrução na linguagem *assembly*, número de operandos usados e número de operandos definidos. Deve haver uma correspondência direta entre `AsmOpCode` e `InstructionSetTable`, isto é, o primeiro `opcode` definido em `AsmOpCode` deve corresponder ao primeiro elemento de `InstructionSetTable`, o segundo *opcode* ao segundo elemento do vetor, e assim por diante.

- **Conjunto de registradores da máquina alvo**
  Deve ser configurado em `RegisterSetTable`, que é um vetor onde cada elemento tem a seguinte estrutura: nome do registrador na linguagem *assembly*, registrador *int* ou *float*, disponibilidade para alocação. Se o registrador é *int*, deve ser dado o valor 0. Do contrário, o valor dado deve ser 1. Se o registrador é disponível para alocação deve ser dado o valor 1 para o campo. Se não, valor 0.

```
class AsmFile {
  public:
  AsmFile ( File* f );
  ~AsmFile ( ) { };

  AsmProcedureList* Procs( );
  void InsertProc ( AsmProcedure* proc );
  virtual void Print( int indent, FILE* fp=stdout ) = 0;
};
```

Figura 5.8: Classe AsmFile

5.5.2 Classes Genéricas

O Xingó possui estruturas de dados genéricas, implementadas por meio de classes abstratas, para representar o código *assembly*. Para construir um gerador de código para uma determinada máquina alvo, é preciso que o desenvolvedor realize as seguintes tarefas:

- primeiro, uma classe específica deve ser derivada de uma classe genérica.
- segundo, na definição da classe específica, uma definição deve ser dada para cada método virtual da classe genérica.

A grande justificativa para utilizar classes genéricas é reusabilidade de código e flexibilidade. Um bom exemplo dessa flexibilidade é a classe genérica que implementa uma instrução *assembly*. Nessa classe, o número de operandos e os seus tipos são configurados pelo próprio desenvolvedor, como visto anteriormente. No quesito reusabilidade, as classes genéricas já provêm uma série de dados e métodos para representar e manipular um arquivo, procedimentos, instruções e operandos *assembly*.

A interface pública das classes genéricas estão ilustradas nas figuras 5.8 a 5.14. As classes são `AsmFile, AsmProcedure, AsmOperand, AsmVariable, AsmLabel, AsmImmediate` e `AsmRegister`.

```
class AsmProcedure {
  public:
  AsmProcedure (Procedure* p);
  ~AsmProcedure ( ) { };

  AsmOperationList* AsmList ( );
  void InsertAsmOperation ( AsmOperation* op );
  virtual void Print( int indent, FILE* fp=stdout ) = 0;
};
```

Figura 5.9: Classe AsmProcedure

```
class AsmOperand {
  public:
  AsmOperand ( );
  virtual ~AsmOperand ( );

  virtual AsmOperandKind Kind ( );
  AsmOperation* Parent ( );
  virtual void Print ( int indent, FILE* fp=stdout );
};
```

Figura 5.10: Classe AsmOperand

```
class AsmVariable : public AsmOperand {
  public:
  AsmVariable ( Variable* v );
  ~AsmVariable ( );

  virtual AsmOperandKind Kind ( );
  Variable* Var ( );
  virtual void Print ( int indent, FILE* fp=stdout );
};
```

Figura 5.11: Classe AsmVariable

```
class AsmLabel : public AsmOperand {
  public:
  AsmLabel ( Label* l );
  ~AsmLabel ( );

  virtual AsmOperandKind Kind ( );
  Label* Lab ( );
  virtual void Print ( int indent, FILE* fp=stdout );
};
```

Figura 5.12: Classe AsmLabel

```
class AsmImmediate : public AsmOperand {
  public:
  AsmImmediate ( Immed* i );
  ~AsmImmediate ( );

  virtual AsmOperandKind Kind ( );
  Immed* Immediate ( );
  virtual void Print ( int indent, FILE* fp=stdout );
};
```

Figura 5.13: Classe AsmImmediate

```
class AsmRegister : public AsmOperand {
  public:
  AsmRegister ( char* nm );
  AsmRegister ( int n );
  ~AsmRegister ( );

  virtual AsmOperandKind Kind ( );
  boolean IsPseudo ( );
  int Ordinal ( );
  int PhysicalReg ( );
  char* Name ( );
  virtual void Print ( int indent, FILE* fp=stdout );
};
```

Figura 5.14: Classe AsmRegister

5.6 Descrição de Máquina

A descrição da máquina alvo é feita através de uma gramática (gramática Olive) que descreve as regras para mapear as árvores de expressões em XOR para o conjunto de instruções representadas em XAR. A gramática Olive toma como entrada a *machine description* para produzir automaticamente o módulo de geração de código do compilador. No compilador Xingó, este módulo faz apenas seleção de instruções.

Como já foi mencionado, para gerar código para uma nova arquitetura, é necessário que o desenvolvedor faça uma descrição da máquina alvo, ou seja, é necessário que ele escreva uma nova *machine description*.

Vale ressaltar que uma descrição Olive do processador alvo especifica somente o modelo do conjunto de instruções da arquitetura como vista pelo programador, sem os detalhes internos da microarquitetura. Significa dizer que a gramática Olive não descreve todos os detalhes do processador alvo, tais como a descrição formal de locais de armazenamento, modos de endereçamento ou via de dados (*datapath*) do processador, mas descreve padrões que mapeiam a representação de código interna do compilador para o conjunto de instruções do processador alvo. Tais padrões são descritos através de regras customizáveis, o que viabiliza uma seleção de instruções otimizada.

A figura 5.15 ilustra a estrutura geral de uma *machine description*. Essa estrutura está dividida em seções, denominadas *arquivos de cabeçalhos, macros, variáveis e funções auxiliares, definição de símbolos terminais, definição de símbolos não-terminais e regras*. A gramática Olive utiliza alguns delimitadores, que são %{, %} e %%. Quando for especificar uma *machine description*, o desenvolvedor deve utilizar corretamente os delimitadores para separar as seções, da mesma forma como está ilustrada na figura.

Na seção *arquivos de cabeçalho* (5.15) são incluídos os arquivos que implementam parte das funcionalidades necessárias para a geração de código. Nesta seção, deve constar pelo menos os seguintes arquivos de cabeçalho.

```
#include <otreenode.h>
#include <cost.h>
#include <codegen.h>
```

Os arquivos de cabeçalho citados são necessários ao gerador de código, para cálculo

%{

< arquivos de cabeçalho >

< macros >

< variáveis e funções auxiliares >

%}

< definição de símbolos terminais >

< definição de símbolos não-terminais >

%%

< regras >

%%

Figura 5.15: Estrutura de uma *machine description*

de custo e definição da estrutura dos nodos das árvores de entrada.

A seção *macros* é uma seção que não precisa ser modificada pelo desenvolvedor. Ela tem sempre o mesmo código, independente da arquitetura alvo. Na seção de *macros* são incluídas algumas definições necessárias ao gerador de código. Esta seção deve ter sempre as seguintes linhas:

```
#define GET_KIDS(r)       ((r)->get_kids())
#define OP_LABEL(r)       ((r)->op_label())
#define STATE_LABEL(r)    ((r)->state_label())
#define SET_STATE(r,s)     (r)->set_state(s)
#define DEFAULT_COST      break
#define PANIC             printf
#define NO_ACTION(x)       NO_ACTION()
```

```
#define CHECK(a)          if (!(a)) return 0
#define COST_ZERO         COST ( 0, USES_NONE );
#define COST_INFINITY     COST (1000000,USES_ACC|USES_TREG|USES_PREG);
#define COST_LESS(x,y)     ((x).cost < (y).cost)
```

Quanto à seção de *variáveis e funções auxiliares*, é recomendado que o desenvolvedor coloque o qualificador "static" para todas as funções e variáveis declaradas, de modo a evitar confusão de nomes com outras variáveis e funções declaradas globalmente no compilador.

A seção *definição de símbolos terminais* serve para informar ao Olive quais são os operadores (terminais) nas árvores de entrada para o gerador de código. O desenvolvedor deve declarar todos os terminais das árvores. No Xingó, o tipo enumerado `OliveTerm` armazena os operadores de uma árvore Olive. Os terminais devem ser declarados na mesma seqüência que eles aparecem em `OliveTerm`, porque o Olive associa a cada terminal um valor numérico seqüencial, atribuindo zero ao primeiro terminal declarado. Terminais são declarados utilizando a diretiva %term. Esta seção é outra que deve ser igual para qualquer descrição de máquina. Por exemplo, as primeiras linhas da seção *definição de símbolos terminais* devem ser, necessariamente, as seguintes.

```
%term nLOADC
%term nLOADS
%term nLOADI
%term nLOADU
%term nLOADP
%term nLOADL
%term nLOADF
%term nLOADD
%term nLOADT
```

Na seção *definição de símbolos não-terminais* são declarados os símbolos que derivam uma produção, ou regra, Olive. Aos símbolos não-teminais são associadas funções em C. Portanto, na declaração de um não-terminal devem ser informados o tipo de retorno

e os tipos dos parâmetros que a função deve receber. De acordo com a implementação Xingó, é preciso que o desenvolvedor faça com que essas funções recebam como parâmetro uma lista de instruções *assembly*. Todas as funções, exceto a do não-terminal de partida, podem retornar qualquer tipo de dado, inclusive *void*. O não-terminal de partida deve, necessariamente, retornar um apontador para uma lista de instruções *assembly* (*AsmOperationList*). A diretiva %declare é utilizada para declarar um símbolo não-terminal. Abaixo, segue um exemplo de declaração de dois não-terminais — *top* e *reg* — onde *top* é o não-terminal de partida.

```
%declare<AsmOperationList*>     top <AsmOperationList* aol>;
%declare<MIPSAsmRegisger*>      reg <AsmOperationList* aol>;
```

Na seção *regras* são descritos os padrões que mapeiam a representação de código na XOR para o conjunto de instruções representado na XAR. Na seção 5.2 foi comentada uma regra gramatical Olive para uma instrução de adição. No capítulo 6 vão ser explicadas as regras implementadas para descrever o processador MIPS R3000, bem como todas as outras partes componentes de uma *machine description*.

5.7 Alocação de Registradores

Logo depois que as instruções *assembly* foram geradas por *Code Generator*, a alocação de registradores ainda não foi efetuada. Desta forma, a maior parte dos registradores considerados são virtuais. Cada registrador virtual representa uma *web*. Uma *web* [36] é uma máxima união de *Du-Chains* (cadeias definição-uso) tal que, para cada definição $d = d0, ..., u0, dn, un = u$, e para cada i, ui pertence a cadeia definição-uso de di e $di + 1$. Na dissertaçaõ de mestrado de W. Attrot [7] são dados detalhes sobre o cálculo de *webs* no compilador Xingó.

Para o desenvolvedor, deve ficar compreendido que, através da utilização de *webs*, o código *assembly* gerado pelo Xingó e expressado na Representação XAR pode utilizar um número ilimitado de registradores virtuais. Com isso, ele precisa implementar um alocador de registradores para atribuir a um registrador virtual algum local de armazenamento na máquina alvo, que pode ser um registrador físico ou uma célula de memória.

Para poder alocar registradores, é necessário ter informações sobre a longevidade dos registradores virtuais, isto é, informações de *Liveness* [36]. A maneira como uma instrução *assembly* usa ou define registradores é específico da máquina alvo, por isso, a *Liveness Analysis* deve ser implementada de acordo com essa arquitetura. De forma a fornecer os meios de realizar *Data Flow Analysis* [2][10][36] no código *assembly*, o Xingó possui um conjunto de classes base para realizar *Data Flow Analysis* no código *assembly* da arquitetura alvo [7]. Além de fornecer dados sobre o fluxo da informação, a *Data Flow Analysis* pode servir como base para a realização de otimizações dependentes de máquina.

Para a alocação de registradores, o Xingó utiliza informações provenientes das classes `PostPassDFAInfo` e `PostPassLivenessClass`. O programador deve criar novas classes que irão herdar estas duas classes e implementar os métodos abstratos que tratam de aspectos específicos da arquitetura alvo.

5.8 Emissão de Código Assembly

O módulo `Emit Assembly Code` é relativamente simples. Tendo implementado todos os métodos virtuais *Print*[5] das classes específicas, o desenvolvedor só precisa chamar estes métodos na ordem correta para produzir como saída, o arquivo contendo o código em *assembly* para a máquina alvo.

Se o desenvolvedor levar em conta que existe uma hierarquia na representação XAR, o método *Print* da classe de um nível acima podem chamar os métodos *Print* das classes de nível imediatamente abaixo. Assim, para gerar o arquivo de saída, basta chamar o método *Print* do objeto que representa a classe `AsmFile`.

5.9 Roteiro para Construir um Gerador de Código

Da forma como foi implementada a infra-estrutura de geração de código no Xingó, as modificações a serem realizadas no compilador para gerar código para uma nova máquina alvo podem ser realizadas de forma relativamente rápida, devido principalmente ao uso do Olive e das classes genéricas que podem ser reusadas por diversos geradores de código.

[5] O método *Print* imprime as informações de um objeto *Assembly* no arquivo *assemby* de saída.

A implementação modular do compilador Xingó, que separa características independentes de máquina dos aspectos dependentes de máquina, fortalece a capacidade de geração de código redirecionável.

Para o desenvolvedor gerar código para uma nova máquina, ele não precisa conhecer todo o código do compilador. Via de regra, ele deve conhecer: (a) a interface pública da Representação Intermediária Xingó (XIR), descrita no anexo A; (b) a estrutura e padrões das árvores Olive, descritos na seção 5.3; (c) as classes genéricas e configurações da *Representação XAR* (seção 5.5); (d) e, finalmente, implementar alocação de registradores específicos para a máquina alvo e emitir o código *assembly* final.

Para gerar código para uma nova máquina alvo, o desenvolvedor tem a opção de seguir uma das alternativas listadas abaixo:

- **Opção 1**: **é o processo sugerido pelos projetistas do compilador Xingó**. Neste caso, o desenvolvedor precisa escrever um *Machine Description* para a máquina alvo e implementar os módulos *Register Allocation* e *Emit Assembly Code*. Além disso, é necessário que ele especialize as classes genéricas da XAR e, se for preciso, criar novas classes além das já existentes.

- **Opção 2**: **Não criar um módulo específico para realizar alocação de registradores (excluir *Register Allocation*)**. Assim, esta tarefa deve ser realizada na própria descrição da máquina alvo (*Machine Description*). Essa abordagem tende a restringir uma melhor alocação de registradores, além de complicar a especificação de *Machine Description*.

- **Opção 3**: **Gerar um arquivo *assembly* (*Assembly Code*) diretamente, sem passar pela XAR**. Não seria inesperado, se alguns desenvolvedores escolhessem essa opção, sobretudo se a máquina alvo for extremamente simples. Contudo, vale o que foi dito para a opção anterior: complica a especificação de *Machine Description* e pode diminuir as chances de uma melhor otimização de código para a máquina destino.

Independente da opção escolhida, é necessário escrever um *Machine Description* para cada processador alvo. Acredita-se que os melhores *back-ends* devem vir da implementação da *opção 1*.

Abaixo, são brevemente comentadas todas as configurações e módulos a serem implementados para gerar código para uma nova arquitetura alvo, levando em conta a abordagem descrita na *opção 1.* Além disso, serão referenciadas as partes do texto que o desenvolvedor deve buscar informações de suporte.

- **Configuração do *Front-End***. Para adicionar ao Xingó um gerador de código para uma nova máquina alvo, o desenvolvedor deve criar uma nova interface de configuração do *front-end*. Essa interface é bastante simples. Ela define métricas para os tipos (*int, float, double,* etc), algumas *flags* para montar corretamente o código intermediário, e as funções da interface de geração de código que devem ser chamadas pelo *front-end*. O desenvolvedor precisa ainda registrar essa nova interface. O registro de uma interface é feito no arquivo "`bind.c`". Na seção 4.6 é descrita, com detalhes, a configuração do *front-end*. Para entender o restante dessa configuração, é necessário ter lido a seção mencionada acima. Apenas valores imediatos do tipo *char*, *short* e *int* podem aparecer nas árvores de expressão. Se o desenvolvedor configura de outra forma este parâmetro, o compilador pode trabalhar de maneira não precisa. Também é necessário que o desenvolvedor sempre configure o tipo *struct* com os parâmetros "0, 1 e 1". Não é possível definir, na configuração, o tamanho e nem o alinhamento do tipo *struct*, já que *struct* define um tipo construído pelo programador. Das *flags* de interface, a *little_endian* é a única que deve ser mudado o valor. As demais *flags* devem manter os valores *default* estabelecidos na ligação entre o LCC e o compilador Xingó. A configuração das funções de interface também não podem ser alteradas pelo desenvolvedor.

- **Configuração da Representação XAR**. A representação XAR e a sua configuração são explicadas na seção 5.5. A configuração deve ser realizada no arquivo "asmconfig.h". Devem ser configuradas neste arquivo as informações que são específicas da arquitetura alvo, tais como o *opcode* das instruções, a quantidade de registradores disponíveis, o tipo dos registradores (*int* ou *float*) e se os mesmos podem ser utilizados para alocação. As configurações necessárias são: número máximo de operandos por instrução, classes dos operandos assembly, Opcodes assembly, conjunto de instruções da máquina alvo e conjunto de registradores da máquina alvo.

- **Implementação de classes na Representação XAR**. Para representar o código

do programa na Representação XAR, o desenvolvedor deve criar classes específicas que se derivam das classes genéricas descritas na seção 5.5. Na definiçaõ de uma classe específica, uma definição deve ser dada para cada método virtual da classe genérica.

- **Escrita de uma descrição de máquina**. Para que o Olive produza um gerador de código para a máquina alvo, uma descrição dessa máquina deve ser fornecida a ele. A descrição da máquina é feita através de uma gramática, denomina *machine description*. O desenvolvedor deve escrever *machine description* seguindo as orientações e convenções apontadas na seção 5.6. É importante ler também a seção 5.2 para entender o funcionamento do gerador de geradores de código Olive.

- **Alocação de registradores**. É necessário implementar um alocador de registradores para atribuir a um registrador virtual algum local de armazenamento na máquina alvo, que pode ser um registrador físico ou uma célula de memória. A maneira como uma instrução *assembly* usa ou define registradores é específico da máquina alvo, por isso, o alocador de registradores deve variar muito de arquitetura para arquitura. Na seção 5.7 são fornecidos alguns detalhes do que considerar na alocação de registradores no Xingó. Maiores informações sobre alocação de registradores no compilador Xingó podem ser encontradas em [7].

- **Emissão de código *assembly***. Este processo deve escrever em um arquivo de saída, o código *assembly* final. Como dito na seção 6.7, o módulo *Emit Assembly Code* é simples. Basta implementar todos os métodos virtuais *Print* das classes na Representação XAR, e então chamá-los para escrever o código em *assembly* no arquivo de saída.

Capítulo 6

Geração de Código MIPS R3000

6.1 Arquitetura MIPS R3000

A arquitetura MIPS R3000 [25] é uma arquitetura RISC (*Reduced Instruction Set Computer*). Ela possui um conjunto reduzido de instruções de 32 *bits* e modos de endereçamento uniformes e simples. O acesso a memória só pode ser realizado através de instruções explícitas de *load* e *store*. A convenção de chamadas a procedimentos estabelece alguns argumentos que devem ser passados em registradores e outros que devem ser passados através da pilha.

Os operandos de uma instrução podem estar armazenados em registradores, em memória (instrução *load*) ou podem ser um valor constante. O formato de cada instrução estabelece a quantidade de operandos e os seus tipos.

A arquitetura fornece ao programador 32 registradores de 32 *bits* que, para o montador, são conhecidos como *$i*. O coprocessador, que realiza operações de ponto-flutuante, adiciona mais 32 registradores de 32 *bits*, que são usualmente tratados como 16 pares de registradores de 64 *bits*. Cada registrador de ponto-flutuante é conhecido pelo montador como *$fi*. O processador consegue endereçar 2^{30} palavras de memória. Cada palavra de memória tem 4 *bytes*, e o acesso pode ser feito por *byte*, *half-word*, *word* ou *double-word*, que, respectivamente, representam acessos para ler ou escrever 1 *byte*, 2 *bytes*, 4 *bytes* ou 8 *bytes*. A tabela 6.1 lista os locais de armazenamento (registradores e memória) de operandos na arquitetura MIPS R3000.

Os registradores de inteiros são os locais de armazenamento com acesso mais rápido.

Operandos	**Domínio**
32 registradores de inteiros	*$0*, *$1*, *$2*, ..., *$31*
32 registradores de ponto-flutuante	*$f0*, *$f1*, *$f2*, ..., *$f31*
2^{30} palavras de memória	Memory[0], Memory[4], ..., Memory[4293967292]

Tabela 6.1: Locais de armazenamento na arquitetura MIPS R3000

O registrador *$0* tem sempre o valor zero. *$1* é reservado para o montador manipular *pseudo* instruções e constantes muito grandes. Por exemplo, o *hardware* permite somente 16 *bits* de *offset* em cálculos de endereço, mas o montador permite *offset* de 32 *bits*. Para isso, o montador adiciona instruções extras que permitem utilizar um *offset* grande através de *$1*.

Os registradores de ponto-flutuante são usados em pares para representar números de precisão dupla. Esses pares de registradores não podem ser utilizados em instruções aritméticas ou de desvio (*branch*), mas são utilizados em instruções de transferência de dados (*load* e *store*), representando uma das metades do par de registradores de precisão dupla.

A memória armazena estruturas de dados (vetores, registros, etc) e a pilha de execução (registradores salvos em chamadas de procedimentos, área de passagem de argumentos, etc).

Para uso dos registradores, o gerador de código implementado observa as convenções adotadas em outros compiladores [14][17] para satisfazer a interoperabilidade com as bibliotecas e depuradores padrão. A tabela 6.2 apresenta as convenções para uso dos registradores.

Todos os nomes de registradores começam com um cifrão ($). O nome de um registrador pode ser aquele conhecido pelo montador (coluna *Nome* da tabela 6.2), ou pode ser um apelido, usualmente preferido pelos programadores.

O sistema operacional e o montador usam os registradores $1, $26, $27, $28 e $29 para propósitos específicos. Tentar utilizar estes registradores para outros propósitos, pode produzir resultados inesperados.

Pela convenção, *$2*-*$3* e *$f0*-*$f2* são reservados para retornar valores em chamadas de procedimentos, mas assim como no LCC [14], o Xingó utiliza apenas a primeira metade, ou seja, *$2* e *$f0*. Algumas linguagens, como Fortran, permitem tipos aritméticos com-

Nome	Apelido	Convenção de uso
$0		sempre zero
$1	$at	reservado para o montador
$2..$3	$v0..$v1	valor de retorno de uma função
$4..$7	$a0..$a3	passagem de argumentos
$8..$15	$t0..$t7	registradores temporários
$16..$23	$s0..$s7	registradores salvos
$24	$t8	registrador temporário
$25	$t9 ou $jp	Jump PIC (código independente de posição)
$26..$27	$k0..$k1	reservados para o sistema operacional
$28	$gp	ponteiro global
$29	$sp	ponteiro para a pilha
$30	$fp ou $s8	registrador salvo ou ponteiro para o *frame* (se necessário)
$31	$ra	endereço de retorno de um procedimento
$f0..$f2	$fv0..$fv1	valor de retorno de uma função
$f4..$f10	$ft0..$ft3	registradores temporários
$f12..$f14	$fa0..$fa1	passagem de argumentos de ponto-flutuante
$f16..$f18	$ft4..$ft5	registradores temporários
$f20..$f30	$fs0..$fs5	registradores salvos

Tabela 6.2: Convenção de uso dos registradores

plexos, e, nesse caso, poderiam ser utilizados os conjuntos completos. A linguagem C não fornece este tipo de dados, mas compiladores C devem respeitar a convenção para poder interoperar com código Fortran.

6.2 Configuração do *Front-End*

Na seção 5.9, foi apresentado o roteiro necessário para que se possa construir um novo gerador de código para o compilador Xingó. A modularidade provida pelo compilador, adicionada à clara definição dos parâmetros a serem configurados e módulos a serem implementados, facilita o processo de construção de novos geradores de código.

Esta seção descreve a configuração do *front-end*, ilustrada na figura 6.1, e as seções subseqüentes mostram a configuração da representação XAR e as implementações dos módulos (classes na representação XAR, descrição da máquina, alocação de registradores e emissão de código *assembly*) específicos para gerar código MIPS R3000.

```
Interface xingoMIPS = {
//type metrics
1, 1, 0, // char
2, 2, 0, // short
4, 4, 0, // int
4, 4, 1, // long
4, 4, 1, // long long
4, 4, 1, // float
8, 8, 1, // double
8, 8, 1, // long double
4, 4, 1, // pointer
0, 1, 1, // struct
//interface flags
0, // little_endian
1, // mulops_calls
1, // wants_callb
1, // wants_argb
1, // left_to_right
0, // wants_dag
//interface functions
I(address),
I(blockbeg),
I(blockend),
I(defaddress),
I(defconst),
I(defstring),
I(defsymbol),
I(emit),
I(export),
I(function),
I(gen),
I(global),
I(import),
I(local),
I(progbeg),
I(progend),
I(segment),
I(space),
};
```

Figura 6.1: Configuração do *front-end*

A interface `xingoMIPS` configura o *front-end* para gerar código exclusivamente para a máquina MIPS R3000. As métricas de tipos foram definidas seguindo as convenções de tamanho e alinhamento de dados estabelecidas pela arquitetura e adotadas pelos compiladores [14][17]. Um dado do tipo *char* deve ocupar 1 *byte* na memória e o alinhamento considerado é por *byte*. Dados do tipo *short* devem ocupar 2 *bytes* na memória e estão alinhados de 2 em 2 *bytes*. Além do tamanho e alinhamento do tipo, o desenvolvedor precisa configurar um terceiro parâmetro, que diz se o *front-end* pode ou não gerar na árvore de expressões um valor imediato daquele tipo. Como imposição do compilador Xingó, apenas valores imediatos do tipo *char*, *short* e *int* podem aparecer nas árvores de expressão. Se o desenvolvedor configura de outra forma este parâmetro, o compilador pode trabalhar de maneira não precisa. Também é necessário que o desenvolvedor sempre configure o tipo *struct* com os parâmetros "0, 1 e 1". Não é possível definir na configuração, o tamanho e nem o alinhamento do tipo *struct*, já que *struct* define um tipo construído pelo programador.

Como dito no roteiro, a única *flag* de interface que deve ser mudada pelo desenvolvedor é a *little_endian*. As demais *flags* devem manter os valores padrão estabelecidos na ligação entre o LCC e o Xingó. As arquiteturas MIPS R4000 e anteriores, podem usar tanto a ordenação de *bytes big-endian* quanto a *little-endian*. Quando configurado como *big-endian*, o *byte* 0 é sempre o mais significante (*leftmost*). Quando configurado como *little-endian*, o *byte* 0 é sempre o menos significante (*rightmost byte*). O simulador [42] de arquitetura MIPS R3000 utilizado para os testes, considera que a máquina é *big endian*, por isso o valor da *flag little_endian* foi definido como zero.

A configuração das funções de interface também não devem ser alteradas pelo desenvolvedor. Para qualquer arquitetura alvo, a definição que se deve dar para as funções de interface são estas apresentadas na figura 6.1.

De acordo com o roteiro, a interface XingoMIPS foi criada no arquivo "`xingocode.c`" e depois registrada no arquivo "`bind.c`".

6.3 Configuração da Representação XAR

Ao tipo enumerado `AsmOperandKind` foram adicionadas *tags* para informar a qual das classes de operandos implementadas na representação de código *assembly* (ver seção 6.4)

o operando se associa. A configuração de `AsmOperandKind` está ilustrada na figura 6.2. Por exemplo, `OpREGISTER` associa-se à classe `MIPSRegister` e `OpBASEREGISTER` associa-se à classe `MIPSBaseRegister`.

O outro tipo enumerado que precisa ser configurado é o `AsmOpCode`. Este tipo define os *opcodes* das instruções na representação de código *assembly*. Na figura 6.3 estão relacionados os primeiros *opcodes* definidos para o MIPS R3000.

Como descrito no roteiro de construção de novos geradores de código, é também necessário configurar o conjunto de instruções *assembly*. Essa configuração é feita no vetor `InstructionSetTable`. Na figura 6.4, estão relacionadas algumas das linhas de configuração de `InstructioSetTable`. As instruções aritméticas e lógicas definem[1] um operando (registrador destino) e usam[2] dois (operandos fonte), enquanto as instruções de desvio usam três operandos (dois registradores de comparação e um *label* alvo do desvio), mas não definem nenhum.

Por fim, foi configurado `RegisterSetTable`, que é um vetor onde cada elemento tem a seguinte estrutura: nome do registrador na linguagem *assembly*; *flag* que diz se é um registrador *int* ou *float*; e *flag* que indica a disponibilidade de alocação, útil para o alocador de registradores. Se o registrador é *int*, deve ser dado o valor 0. Do contrário, o valor dado deve ser 1. Se o registrador é disponível para alocação deve ser dado o valor 1 para o campo. Se não, valor 0. A figura 6.5 mostra a configuração de `RegisterSetTable`.

[1]Entende-se por definição, a atualização (escrita) do operando.
[2]Entende-se por uso, a simples leitura do operando.

```
enum AsmOperandKind {

    OpUNKNOWN,
    OpVARIABLE,         // MIPSVariable
    OpIMMEDIATE,        // MIPSImmediate
    OpLABEL,            // MIPSLabel
    OpREGISTER,         // MIPSRegister
    OpADDRREGISTER,     // MIPSAddrRegister
    OpEXPRREGISTER,     // MIPSExprRegister
    OpBASEREGISTER,     // MIPSBaseRegister
    OpSTRUCTREGISTER    // MIPSStructRegister

};
```

Figura 6.2: Configuração de AsmOperandKind

```
enum AsmOpCode{

    //Arithmetic and Logical Instructions
    aADD = 0,    aADDU,    aADDI,    aADDIU,
    aAND,        aANDI,    aDIV,     aDIVU,
    aMUL,        aMULU,    aNOR,     aOR,
    aASLL,       aSLLV,    aSRA,     aSRAV,
    aSRL,        aSRLV,    aSUB,     aSUBU,
    aXOR,        aXORI,

     ...

};
```

Figura 6.3: Configuração de AsmOpCode

```
InstructionSetTable[] ={

    //Arithmetic and Logical Instructions
    {"add",    2,    1 }
    {"addu",   2,    1 }
    {"addi",   2,    1 }
    {"addiu",  2,    1 }
    {"and",    2,    1 }
    {"andi",   2,    1 }

           ...

    //Branch Instructions
    {"bge",    3,    0 }
    {"bgeu",   3,    0 }
    {"bgt",    3,    0 }
    {"bgtu",   3,    0 }
    {"ble",    3,    0 }
    {"bleu",   3,    0 }

           ...

};
```

Figura 6.4: Configuração de InstructionSetTable

```
RegisterSetTable[] ={

    { "$0", 0, 0 },   { "$1", 0, 1 },
    { "$2", 0, 1 },   { "$3", 0, 1 },
    { "$4", 0, 1 },   { "$5", 0, 1 },
    { "$6", 0, 1 },   { "$7", 0, 1 },
    { "$8", 0, 1 },   { "$9", 0, 1 },
    { "$10", 0, 1 },   { "$11", 0, 1 },
    { "$12", 0, 1 },   { "$13", 0, 1 },
    { "$14", 0, 1 },   { "$15", 0, 1 },
    { "$16", 0, 1 },   { "$17", 0, 1 },
    { "$18", 0, 1 },   { "$19", 0, 1 },
    { "$20", 0, 1 },   { "$21", 0, 1 },
    { "$22", 0, 1 },   { "$23", 0, 1 },
    { "$24", 0, 1 },   { "$25", 0, 1 },
    { "$26", 0, 0 },   { "$27", 0, 0 },
    { "$28", 0, 0 },   { "$29", 0, 0 },
    { "$30", 0, 0 },   { "$31", 0, 0 },
    { "$f0", 1, 1 },   { "$f2", 1, 1 },
    { "$f4", 1, 1 },   { "$f6", 1, 1 },
    { "$f8", 1, 1 },   { "$f10", 1, 1 },
    { "$f12", 1, 1 },   { "$f14", 1, 1 },
    { "$f16", 1, 1 },   { "$f18", 1, 1 },
    { "$f20", 1, 1 },   { "$f22", 1, 1 },
    { "$f24", 1, 1 },   { "$f26", 1, 1 },
    { "$f28", 1, 1 },   { "$f30", 1, 1 },

};
```

Figura 6.5: Configuração de RegisterSetTable

6.4 Classes na Representação XAR

O Xingó possui uma série de classes genéricas para armazenar o código na Representação Assembly Xingó. Para implementar o gerador de código MIPS R3000, foram criadas classes específicas que se derivam destas classes genéricas. Mais precisamente, foram implementadas as seguintes classes:

- `MIPSFile e MIPSProcedure`: Estas classes são derivadas, respectivamente, das classes genéricas `AsmFile` e `AsmProcedure`. `MIPSFile` corresponde a um arquivo *assembly* e possui uma lista de objetos da classe `MIPSProcedure`, que representa um procedimento *assembly*.

- `MIPSOperand, MIPSVariable, MIPSImmediate e MIPSLabel`: Estas classes são derivadas, respectivamente, das classes `AsmOperand, AsmVariable, AsmImmediate` e `AsmLabel`. Elas não incluem nenhuma outra propriedade em relação às classes herdadas, portanto, apenas implementam os métodos virtuais das classes genéricas. A classe `MIPSOperand` é a classe base de um operando MIPS. A classe `MIPSVariable` corresponde a uma variável global ou a um parâmetro de uma função. A classe `MIPSImmediate` é usada para instanciar objetos que representam um valor imediato de uma instrução. `MIPSLabel` representa um *label* alvo de desvio.

- `MIPSRegister`: Esta classe é derivada da classe genérica `AsmRegister`. Ela corresponde a um registrador MIPS. `MIPSRegister` adiciona um *flag* que determina se o registrador é virtual ou se ele é físico. Também foi colocada nesta classe, uma propriedade que informa se o registrador é de inteiros ou de ponto-flutuante.

- `MIPSBaseRegister`: Esta classe e as demais abaixo não são diretamente derivadas das classes genéricas. Elas são classes que foram desenvolvidas exclusivamente para representar os diversos tipos de operandos de instruções *assembly* MIPS. A classe `MIPSBaseRegister` representa um operando *assembly* da forma *(register)*, onde *register* endereça uma célula de memória. Por exemplo, a instrução *lw $8, ($9)* utiliza *$9* para fornecer o endereço de memória a partir do qual serão carregados 4 *bytes* e armazenados em *$8*.

- `MIPSAddrRegister`: Esta classe representa um operando *assembly* da forma *addr (register)*. Por exemplo, a instrução *lw $8, vetor($9)* utiliza a variável *vetor* para

fornecer o endereço base de memória e *$9* o deslocamento em relação a este endereço base. A soma de *vetor* e *$9* expressa o endereço do qual devem ser lidos os 4 *bytes*.

- `MIPSExprRegister`: Esta classe representa um operando *assembly* da forma *expr (register)*. A diferença deste operando em relação a um operando da classe anterior é que *expr* é dado por um valor imediato e *addr* por uma variável.

- `MIPSStructRegister`: Esta classe é utilizada por operandos que se originam de variáveis do tipo *struct*. Como um *struct* normalmente é maior do que 4 *bytes* (tamanho máximo suportado por um registrador MIPS), é necessário agrupar vários registradores para armazenar um operando deste tipo. A classe `MIPSStructRegister` foi implementada para reunir esses vários registradores e os associar a um operando do tipo *struct*.

6.5 Descrição da Máquina MIPS R3000

Como foi dito no capítulo 5, para gerar código para uma nova arquitetura, o desenvolvedor precisa fazer uma descrição da máquina alvo. Esta descrição é chamada de *machine description*. A partir de *machine description*, Olive [45] produz um gerador de código para a máquina alvo.

Como foi especificado na seção 5.6, uma *machine description* é formada pelas seguintes seções: arquivos de cabeçalho, macros, variáveis e funções auxiliares, definição de símbolos terminais, definição de símbolos não-terminais e regras.

Serão abordadas aqui neste texto, apenas as seções definição de símbolos não-terminais e regras. As seções macros e definição de símbolos terminais são sempre implementadas da mesma forma, independente da máquina alvo, e já foram detalhadas no capítulo anterior. O não detalhamento de quais os arquivos de cabeçalho e quais as variáveis e funções auxiliares declarados ou utilizados na definição da gramática, não influenciam a compreensão da descrição da máquina MIPS R3000 utilizando Olive, que é o objetivo pretendido a partir de agora.

As regras para o MIPS R3000 foram definidas utilizando-se oito não-terminais. Olive associa, a cada não-terminal, uma função típica em C. Esta função, como qualquer outra, pode ser declarada para receber parâmetros e retornar dados. Todos os não-terminais re-

cebem como entrada um ponteiro para uma lista de operações *assembly*[3] em XAR, ou mais tecnicamente, um ponteiro para uma lista (`AsmOperationList`) de objetos `AsmOperation`. Abaixo, são listados os não-terminais declarados:

- *top*: não terminal de partida da gramática. Retorna um ponteiro para uma lista de operações *assembly*. A declaração de *top* é apresentada abaixo:

  ```
  %declare <AsmOperationList*> top <AsmOperationList* aol>;
  ```

- *acon*: define regras que expressam um operando que pode ser um endereço de memória ou um valor imediato. Retorna um ponteiro para um objeto `MIPSOperand`.

  ```
  %declare <MIPSOperand*> acon <AsmOperationList*aol>;
  ```

- *addr*: define regras que expressam um operando que referencia um endereço de memória. Retorna um ponteiro para um objeto `MIPSOperand`.

  ```
  %declare <MIPSOperand*> addr <AsmOperationList*aol>;
  ```

- *reg*: define regras que expressam um operando que deve estar em um registrador. Retorna um ponteiro para um objeto `MIPSRegister`.

  ```
  %declare <MIPSRegister*> reg <AsmOperationList*aol>;
  ```

- *args*: define regras que expressam um argumento passado para um procedimento. Não retorna valor, ou seja, é do tipo *void*.

  ```
  %declare <void> args <AsmOperationList*aol>;
  ```

- *rc*: define regras que expressam um operando que pode ser um registrador ou um valor imediato. Retorna um ponteiro para um objeto `MIPSOperand`.

  ```
  %declare <MIPSOperand*> rc <AsmOperationList*aol>;
  ```

- *rc5*: define regras que expressam um operando que pode ser um registrador ou um valor imediato, mas que tenha um valor representável com 5 *bits*. Retorna um ponteiro para um objeto `MIPSOperand`.

  ```
  %declare <MIPSOperand*> rc5 <AsmOperationList*aol>;
  ```

[3]Considere que uma operação *assembly* referida nesta seção é uma representação, em memória, da instrução *assembly* realmente reconhecida pelo montador para a arquitetura alvo. Esta representação é feita por um objeto `MIPSOperation`, descrito na seção 6.4.

- *con*: define regras que expressam um operando que é um valor imediato. Retorna um ponteiro para um objeto `MIPSImmediate`.

  ```
  %declare <MIPSImmediate*> con <AsmOperationList*aol>;
  ```

As regras que serão descritas mais abaixo permitem o mapeamento das árvores de expressão na representação XOR para o conjunto de instruções na representação XAR. Essas regras, apesar de simplificadas neste texto, dão uma boa noção da descrição da máquina MIPS R3000 utilizando Olive. A gramática que foi implementada aqui é baseada numa gramática Iburg [15] para o MIPS R3000. Esta gramática está bem documentada no livro do LCC [14].

Para descrever as regras Olive implementadas para o MIPS R3000, serão colocados, nesta ordem, o não-terminal que deriva a regra, o padrão e a(s) operação(ões) *assembly* gerada(s) e armazenadas na Representação XAR. As operações *assembly* são descritas com o *opcode* seguido dos tipos dos objetos que representam os operandos da operação. Todas as regras, exceto aquelas derivadas do não-terminal *top*, retornam um operando, mas nem todas geram operações *assembly*.

É importante reforçar que Olive trabalha com regras customizáveis, o que viabiliza uma seleção de instruções otimizada. O custo de casar uma árvore de expressão é o custo de casar a raiz da árvore somado aos custos calculados para casar todas as subárvores da expressão. Por exemplo, logo na primeira regra "*top:nSTOREC(addr,reg)*", que será mostrada a seguir, o custo de casar esta regra é a soma dos custos de casar a subárvore representada por *addr* e a subárvore representada por *reg* acrescido do custo de casar `nSTOREC`.

Na geração de código MIPS R3000, o custo definido baseia-se no tamanho do código gerado. Isso quer dizer que se existirem duas regras que casam a mesma árvore de expressão, a regra escolhida vai ser aquela que gerar o menor número de operações *assembly*. Caso duas regras gerem o mesmo número de operações, o gerador de código pode escolher qualquer uma delas, sem prejuízo.

Algumas operações *assembly* têm um sufixo que identifica o tipo do dado que a operação opera. Este sufixo, na maioria dos casos, é o mesmo sufixo utilizado para as instruções *assembly* reconhecidas pelo montador. Os sufixos *S* e *D* identificam operações de ponto-flutuante de precisão simples e de precisão dupla, e *B, H* e *W* identificam instruções inteiras de *8, 16 e 32 bits*, respectivamente. O sufixo opcional *U* identifica as

operações como não sinalizadas. Se ele é omitido, a operação é sinalizada. Existe um sufixo especial, que é o *T*. Este sufixo refere-se às operações que trabalham com dados do tipo *struct*. O sufixo *T* não é válido para as instruções armazenadas no arquivo *assembly* de saída. Ele é utilizado apenas nas operações representadas em XAR, que devem, portanto, ser mapeadas para instruções *assembly* realmente válidas para o montador MIPS R3000.

As regras *top* mapeiam uma árvore de expressão a partir da sua raíz. As outras regras, que são derivadas a partir dos outros não-terminais, mapeiam as sub-árvores que denotam a expressão.

As regras a seguir mapeiam árvores para instruções que fazem escrita na memória. Na representação XAR, as instruções com prefixo *aS* simbolizam uma operação de *store*:

```
top:nSTOREC(addr,reg)   aSB  MIPSOperand,MIPSRegister
top:nSTORES(addr,reg)   aSH  MIPSOperand,MIPSRegister
top:nSTOREI(addr,reg)   aSW  MIPSOperand,MIPSRegister
top:nSTOREU(addr,reg)   aSWU MIPSOperand,MIPSRegister
top:nSTOREP(addr,reg)   aSW  MIPSOperand,MIPSRegister
top:nSTOREL(addr,reg)   aSW  MIPSOperand,MIPSRegister
top:nSTOREF(addr,reg)   aSS  MIPSOperand,MIPSRegister
top:nSTORED(addr,reg)   aSD  MIPSOperand,MIPSRegister
top:nSTORET(addr,reg)   aST  MIPSOperand,MIPSRegister
```

As árvores, cujo operador é derivado de `nMOVE`, podem levar à operações de *store* ou de *move*. Elas levam à operações de *move* caso tenha sido atribuído algum registrador virtual à variável destino, representada na árvore por um nodo com prefixo `nVAR`. O caso particular é `nMOVET`, que sempre leva à uma operação de *store*, pois, normalmente, um operando do tipo *struct* não cabe em um registrador (32 *bits*, no máximo) da arquitetura.

```
top:nMOVEC(nVARC,reg)  aSB   MIPSVariable,MIPSRegister ou
                       aMOVE MIPSRegister,MIPSRegister
top:nMOVES(nVARS,reg)  aSH   MIPSVariable,MIPSRegister ou
                       aMOVE MIPSRegister,MIPSRegister
top:nMOVEI(nVARI,reg)  aSW   MIPSVariable,MIPSRegister ou
                       aMOVE MIPSRegister,MIPSRegister
```

```
top:nMOVEU(nVARU,reg)  aSW   MIPSVariable,MIPSRegister ou
                       aMOVE MIPSRegister,MIPSRegister
top:nMOVEP(nVARP,reg)  aSW   MIPSVariable,MIPSRegister ou
                       aMOVE MIPSRegister,MIPSRegister
top:nMOVEL(nVARL,reg)  aSW   MIPSVariable,MIPSRegister ou
                       aMOVE MIPSRegister,MIPSRegister
top:nMOVEF(nVARF,reg)  aSS   MIPSVariable,MIPSRegister ou
                       aMOVS MIPSRegister,MIPSRegister
top:nMOVED(nVARD,reg)  aSD   MIPSVariable,MIPSRegister ou
                       aMOVD MIPSRegister,MIPSRegister
top:nMOVET(nVART,reg)  aST   MIPSVariable,MIPSRegister
```

A operações de desvio incondicional `aB` (desvio para um endereço fixo) e `aJ` (desvio para um endereço contido em um registrador) usam as regras abaixo:

```
top:nJUMPV(acon) aB MIPSLabel
top:nJUMPV(reg)  aJ MIPSRegister
```

Blocos do tipo *switch* podem ser implementados utilizando *jump tables*, que precisam da operação `aJ`. Todos os outros desvios incondicionais usam `aB`.

Os desvios condicionais que comparam dois registradores do tipo inteiro e saltam para um *label* se o resultado da comparação é verdadeiro, são dados pelas regras abaixo:

```
top:nJEI(reg,reg)   aBEQ  MIPSRegister,MIPSRegister,MIPSLabel
top:nJNEI(reg,reg)  aBNE  MIPSRegister,MIPSRegister,MIPSLabel
top:nJGEI(reg,reg)  aBGE  MIPSRegister,MIPSRegister,MIPSLabel
top:nJGEU(reg,reg)  aBGEU MIPSRegister,MIPSRegister,MIPSLabel
top:nJGTI(reg,reg)  aBGT  MIPSRegister,MIPSRegister,MIPSLabel
top:nJGTU(reg,reg)  aBGTU MIPSRegister,MIPSRegister,MIPSLabel
top:nJLEI(reg,reg)  aBLE  MIPSRegister,MIPSRegister,MIPSLabel
top:nJLEU(reg,reg)  aBLEU MIPSRegister,MIPSRegister,MIPSLabel
top:nJLTI(reg,reg)  aBLT  MIPSRegister,MIPSRegister,MIPSLabel
top:nJLTU(reg,reg)  aBLTU MIPSRegister,MIPSRegister,MIPSLabel
```

Os desvios condicionais de ponto-flutuante testam um *flag* condicional, ligado[4] através de uma operação de comparação. Por exemplo, `aCLTD r1,r2` liga o *flag* se o valor de precisão dupla em *r1* é menor do que o valor de precisão dupla contido em *r2*. `aBC1T` vai promover o desvio para um *label* destino se o *flag* está ligado, e `aBC1F` se o *flag* está desligado.

```
top:nJEF(reg,reg)   aCEQS MIPSRegister,MIPSRegister e aBCT1T MIPSLabel
top:nJED(reg,reg)   aCEQD MIPSRegister,MIPSRegister e aBCT1T MIPSLabel
top:nJLEF(reg,reg)  aCLES MIPSRegister,MIPSRegister e aBCT1T MIPSLabel
top:nJLED(reg,reg)  aCLED MIPSRegister,MIPSRegister e aBCT1T MIPSLabel
top:nJLTF(reg,reg)  aCLTS MIPSRegister,MIPSRegister e aBCT1T MIPSLabel
top:nJLTD(reg,reg)  aCLTD MIPSRegister,MIPSRegister e aBCT1T MIPSLabel
```

Comparações de ponto-flutuante no MIPS R3000 implementam somente as instruções: menor que (*less-than*), menor ou igual a (*less-than-or-equal*), e igual (*equal*). Então, assim como o LCC [14], o Xingó implementa o resto das instruções de comparação através da inversão da relação desejada, seguida de uma operação `aBC1F`:

```
top:nJNEF(reg,reg)  aCEQS MIPSRegister,MIPSRegister e aBCT1F MIPSLabel
top:nJNED(reg,reg)  aCEQD MIPSRegister,MIPSRegister e aBCT1F MIPSLabel
top:nJGEF(reg,reg)  aCLTS MIPSRegister,MIPSRegister e aBCT1F MIPSLabel
top:nJGED(reg,reg)  aCLTD MIPSRegister,MIPSRegister e aBCT1F MIPSLabel
top:nJGTF(reg,reg)  aCLES MIPSRegister,MIPSRegister e aBCT1F MIPSLabel
top:nJGTD(reg,reg)  aCLED MIPSRegister,MIPSRegister e aBCT1F MIPSLabel
```

Um *label*, alvo de desvio, é definido pela regra seguinte. Ele gera uma operação *assembly* `aLABEL`.

```
top:nLABELV  aLABEL MIPSLabel
```

As chamadas a procedimentos que não retornam dados, ou seja, funções do tipo *void*, são descritas pelas seguintes regras:

```
top:nCALLV(nVARX,args)  aJAL MIPSVariable
top:nCALLV(nVARP,args)  aJAL MIPSVariable
```

[4]Considere ligar um *flag*, o ato de tornar o seu valor igual a 1 (binário).

`nVARX` refere-se a uma variável que representa um procedimento e `nVARP` refere-se, no caso, a uma variável que representa um ponteiro para um procedimento.

Os procedimentos que retornam dados implementam uma operação que armazena o valor de retorno. O valor retornado deve ser colocado em um local específico. Retorno de valores de ponto-flutuante deve ser feito através do registrador de precisão dupla *$f0*, e todos os outros valores devem ser retornados em *$2*:

```
top:nRETI(reg)   aMOVE $2,MIPSRegister
top:nRETP(reg)   aMOVE $2,MIPSRegister
top:nRETF(reg)   aMOVS $f0,MIPSRegister
top:nRETD(reg)   aMOVD $f0,MIPSRegister
```

Constantes e variáveis globais auto-representam-se na linguagem *assembly*. No caso das variáveis globais, o próprio identificador pode ser usado para fornecer o seu endereço de memória. As regras definidas por *acon*, que refere-se a um endereço de memória (que pode ser dado por uma variável) ou a uma constante, são descritas a seguir:

```
acon:con  (não gera operação assembly)
```

As regras seguintes servem para produzir operandos *assembly* que referem-se ao endereço de uma variável global. Se as variáveis representadas por `nVARC, nVARS, ..., nVARD` forem globais, o custo atribuído a regra é 1. Se não, é atribuído um custo infinito (custo máximo possível), para que esta regra não venha ser selecionada.

```
acon:nADDRP(nVARC) (não gera operação assembly)
acon:nADDRP(nVARS) (não gera operação assembly)
acon:nADDRP(nVARI) (não gera operação assembly)
acon:nADDRP(nVARU) (não gera operação assembly)
acon:nADDRP(nVARP) (não gera operação assembly)
acon:nADDRP(nVARL) (não gera operação assembly)
acon:nADDRP(nVARF) (não gera operação assembly)
acon:nADDRP(nVARD) (não gera operação assembly)
```

As regras "*acon*"não geram nenhuma operação *assembly*. Elas simplesmente retornam um operando que representa um valor constante ou uma variável global.

As instruções *assembly* para acesso à memória (*load* e *store*) usam um operando que deve informar o endereço de memória. Este operando, dentre algumas das combinações válidas, pode ser dado pela soma de um valor constante ou endereço simbólico e o conteúdo de um registrador.

```
addr:nADDI(reg,acon)  (não gera operação assembly)
addr:nADDU(reg,acon)  (não gera operação assembly)
addr:nADDP(reg,acon)  (não gera operação assembly)
```

As regras listadas imediatamente acima não geram nenhuma operação *assembly*. Elas simplesmente retornam um objeto `MIPSAddrRegister`, que é formado pela união de um registrador e um adicional, que pode ser um valor constante ou uma variável.

Um endereço pode ser dado ainda por um valor absoluto e por um endereçamento indireto, através de um registrador:

```
addr:acon (não gera operação assembly)
addr:reg  (não gera operação assembly)
```

Um endereço na pilha, dado a alguns parâmetros e variáveis locais, também é freqüentemente utilizado por instruções de *load* e *store*. As regras a seguir geram operandos *assembly* para corresponder à parâmetros e variáveis locais.

```
addr:nADDRP(nVARC)  (não gera operação assembly)
addr:nADDRP(nVARS)  (não gera operação assembly)
addr:nADDRP(nVARI)  (não gera operação assembly)
addr:nADDRP(nVARU)  (não gera operação assembly)
addr:nADDRP(nVARP)  (não gera operação assembly)
addr:nADDRP(nVARL)  (não gera operação assembly)
addr:nADDRP(nVARF)  (não gera operação assembly)
addr:nADDRP(nVARD)  (não gera operação assembly)
addr:nADDRP(nVART)  (não gera operação assembly)
```

Os nodos de prefixo *nVAR* são testados em cada uma das regras, para saber se eles realmente correspondem à variáveis locais ou parâmetros, pois poderiam, naturalmente, ser uma variável global, não podendo, assim, aplicar-se a mesma semântica.

Diversas operações *assembly* consideram um operando como sendo um registrador. A instrução *la* (na linguagem do montador) realiza o cálculo de um endereço e armazena o resultado em um registrador. Por exemplo, la $2,x($4) soma o conteúdo de $4 e x e armazena o valor em $2. A regra seguinte mapeia uma árvore de expressão que gera uma operação `aLA` (na representação XAR), que será posteriormente convertida em uma instrução *la*:

```
reg:addr  aLA MIPSRegister,MIPSOperand
```

Sempre que uma constante tiver o valor zero, é possível utilizar o registrador $0 (tem sempre o valor zero) no seu lugar. Com isso, não é preciso gerar nenhuma instrução adicional. As regras seguintes verificam se o valor da constante (dada pelo nodo de prefixo `nCON`) é zero, e caso seja, a regra devolve um operando `MIPSRegister` que corresponde ao registrador físico $0 da arquitetura.

```
reg:nCONC  (não gera operação assembly)
reg:nCONS  (não gera operação assembly)
reg:nCONI  (não gera operação assembly)
reg:nCONU  (não gera operação assembly)
reg:nCONP  (não gera operação assembly)
```

As árvores de expressão que referem-se ao carregamento do conteúdo de um endereço de memória para um registrador são mapeadas em instruções *load*. Essas árvores caracterizam-se por ter na raiz um nodo de prefixo `nLOAD`, que aponta para uma subárvore que representa um endereço de memória. Abaixo, seguem as regras para gerar operações de *load*:

```
reg:nLOADC(addr)  aLB  MIPSRegister,MIPSOperand
reg:nLOADS(addr)  aLH  MIPSRegister,MIPSOperand
reg:nLOADI(addr)  aLW  MIPSRegister,MIPSOperand
reg:nLOADU(addr)  aLWU MIPSRegister,MIPSOperand
reg:nLOADP(addr)  aLW  MIPSRegister,MIPSOperand
reg:nLOADL(addr)  aLW  MIPSRegister,MIPSOperand
reg:nLOADF(addr)  aLS  MIPSRegister,MIPSOperand
reg:nLOADD(addr)  aLD  MIPSRegister,MIPSOperand
```

Essas regras selecionam um novo registrador virtual para receber o conteúdo lido da memória.

Quando uma variável (prefixo `nVAR`) aparece como um nodo de uma árvore de expressão e não tem um nodo `nADDR` como pai, significa que está utilizando-se o seu valor, e não o seu endereço. Se a variável já possui um registrador virtual associado, tudo a ser feito pela regra é retornar o objeto `MIPSRegister` que simboliza este registrador virtual. No caso dessa variável ter um registrador físico já definido (por exemplo, a variável é um argumento passado em um registrador), este registrador físico também é retornado. De outro modo, a variável está em memória e, portanto, seu valor deve ser carregado para um registrador, que no caso é virtual. As regras abaixo fazem exatamente as verificações (variável em registrador virtual, em registrador físico ou em memória) ditas e, somente no caso da variável estar em memória, gera-se uma operação *load*:

```
reg:nVARC  aLB MIPSRegister,MIPSVariable
reg:nVARS  aLH MIPSRegister,MIPSVariable
reg:nVARI  aLW MIPSRegister,MIPSVariable
reg:nVARU  aLW MIPSRegister,MIPSVariable
reg:nVARP  aLW MIPSRegister,MIPSVariable
reg:nVARL  aLW MIPSRegister,MIPSVariable
reg:nVARF  aLS MIPSRegister,MIPSVariable
reg:nVARD  aLD MIPSRegister,MIPSVariable
```

É possível verificar que algumas árvores de expressão em XOR expressam as seguintes operações: carrega um conteúdo de um endereço de memória para um registrador e depois faz uma conversão de tipos deste conteúdo. A conversão de tipos, neste caso, é propagar um *bit* de sinal para preencher a parte mais significativa do registrador (operações sinalizadas) ou, ao contrário, preencher com zeros (operações não sinalizadas). As instruções *assembly lb* e *lh* propagam *bit* de sinal, e *lbu* e *lhu* preenchem com zeros a parte mais significativa do registrador. Desse modo, ao invés de usar duas operações: uma que lê da memória e outra que faz a conversão de tipos, é possível utilizar uma única operação de leitura da memória. Esta "otimização" só se aplica para a conversão de *char/short* para *int*, ou de *char/short* para *unsigned*:

```
reg:nCVRTCI(nLOADC(addr))  aLB  MIPSRegister,MIPSOperand
reg:nCVRTSI(nLOADS(addr))  aLH  MIPSRegister,MIPSOperand
reg:nCVRTCU(nLOADC(addr))  aLBU MIPSRegister,MIPSOperand
reg:nCVRTSU(nLOADS(addr))  aLHU MIPSRegister,MIPSOperand
```

Todas as operações inteiras de multiplicação, divisão e resto aceitam dois registradores fonte e colocam o resultado em um registrador destino:

```
reg:nMULI(reg,reg)  aMUL  MIPSRegister,MIPSRegister,MIPSRegister
reg:nMULU(reg,reg)  aMULU MIPSRegister,MIPSRegister,MIPSRegister
reg:nDIVI(reg,reg)  aDIV  MIPSRegister,MIPSRegister,MIPSRegister
reg:nDIVU(reg,reg)  aDIVU MIPSRegister,MIPSRegister,MIPSRegister
reg:nMODI(reg,reg)  aREM  MIPSRegister,MIPSRegister,MIPSRegister
reg:nMODU(reg,reg)  aREMU MIPSRegister,MIPSRegister,MIPSRegister
```

As demais operações inteiras binárias também apresentam uma forma imediata, na qual o segundo operando fonte pode ser um valor constante:

```
reg:nADDI(reg,rc)  aADDU MIPSRegister,MIPSRegister,MIPSOperand
reg:nADDP(reg,rc)  aADDU MIPSRegister,MIPSRegister,MIPSOperand
reg:nADDU(reg,rc)  aADDU MIPSRegister,MIPSRegister,MIPSOperand
reg:nANDU(reg,rc)  aAND  MIPSRegister,MIPSRegister,MIPSOperand
reg:nANDI(reg,rc)  aAND  MIPSRegister,MIPSRegister,MIPSOperand
reg:nORU(reg,rc)   aOR   MIPSRegister,MIPSRegister,MIPSOperand
reg:nORI(reg,rc)   aOR   MIPSRegister,MIPSRegister,MIPSOperand
reg:nXORU(reg,rc)  aXOR  MIPSRegister,MIPSRegister,MIPSOperand
reg:nXORI(reg,rc)  aXOR  MIPSRegister,MIPSRegister,MIPSOperand
reg:nSUBI(reg,rc)  aSUBU MIPSRegister,MIPSRegister,MIPSOperand
reg:nSUBP(reg,rc)  aSUBU MIPSRegister,MIPSRegister,MIPSOperand
reg:nSUBU(reg,rc)  aSUBU MIPSRegister,MIPSRegister,MIPSOperand
```

Operações de *shift*, entretanto, requerem constantes entre zero e 31, que é o deslocamento máximo possível, pois um registrador tem 32 *bits*:

```
reg:nLSLI(reg,rc5)  aSLL MIPSRegister,MIPSRegister,MIPSOperand
```

```
reg:nLSLU(reg,rc5)  aSLL MIPSRegister,MIPSRegister,MIPSOperand
reg:nLSRI(reg,rc5)  aSRA MIPSRegister,MIPSRegister,MIPSOperand
reg:nLSRU(reg,rc5)  aSRL MIPSRegister,MIPSRegister,MIPSOperand
```

Operações unárias usam sempre um registrador como operando fonte:

```
reg:nCOMU(reg)   aNOT  MIPSRegister,MIPSRegister
reg:nNEGI(reg)   aNEGU MIPSRegister,MIPSRegister
reg:nNEGF(reg)   aNEGS MIPSRegister,MIPSRegister
reg:nNEGD(reg)   aNEGD MIPSRegister,MIPSRegister
```

As operações binárias de ponto-flutuante permitem somente registradores como operandos fonte:

```
reg:nADDF(reg,reg)  aADDS MIPSRegister,MIPSRegister,MIPSRegister
reg:nADDD(reg,reg)  aADDD MIPSRegister,MIPSRegister,MIPSRegister
reg:nSUBF(reg,reg)  aSUBS MIPSRegister,MIPSRegister,MIPSRegister
reg:nSUBD(reg,reg)  aSUBD MIPSRegister,MIPSRegister,MIPSRegister
reg:nMULF(reg,reg)  aMULS MIPSRegister,MIPSRegister,MIPSRegister
reg:nMULD(reg,reg)  aMULD MIPSRegister,MIPSRegister,MIPSRegister
reg:nDIVF(reg,reg)  aDIVS MIPSRegister,MIPSRegister,MIPSRegister
reg:nDIVD(reg,reg)  aDIVD MIPSRegister,MIPSRegister,MIPSRegister
```

As regras para árvores de expressão que denotam chamadas a procedimentos que retornam valor, devem colocar este valor em um registrador específico da arquitetura, como, por exemplo, $2 ou $f0 (como visto na seção 6.1). Estas regras devem ainda emitir uma operação `aJAL`, que expressa o desvio para o procedimento chamado:

```
reg:nCALLI(nVARX,args)  aJAL MIPSVariable
reg:nCALLU(nVARX,args)  aJAL MIPSVariable
reg:nCALLP(nVARX,args)  aJAL MIPSVariable
reg:nCALLF(nVARX,args)  aJAL MIPSVariable
reg:nCALLD(nVARX,args)  aJAL MIPSVariable
```

Os argumentos de um procedimento são expressados através das regras abaixo:

```
args:nARGI(reg,args)  aSW   MIPSExprRegister,MIPSRegister ou
                      aMOVE MIPSRegister,MIPSRegister
args:nARGU(reg,args)  aSW   MIPSExprRegister,MIPSRegister ou
                      aMOVE MIPSRegister,MIPSRegister
args:nARGP(reg,args)  aSW   MIPSExprRegister,MIPSRegister ou
                      aMOVE MIPSRegister,MIPSRegister
args:nARGF(reg,args)  aSS   MIPSExprRegister,MIPSRegister ou
                      aMOVS MIPSRegister,MIPSRegister
args:nARGD(reg,args)  aSD   MIPSExprRegister,MIPSRegister ou
                      aMOVD MIPSRegister,MIPSRegister
args:nNOP             (não gera operação assemlby)
```

Esta última regra serve para indicar que a lista de argumentos terminou. Uma operação `aSW, aSS` ou `aSD` é gerada quando o argumento precisa ser passado através da pilha. Nessas operações, o operando `MIPSExprRegister` fornece o registrador base da pilha ($sp) e o deslocamento (*offset*) em relação à base. Caso o argumento seja passado através de um registrador, utiliza-se as operações `aMOVE, aMOVS` e `aMOVD` para mover o conteúdo do registrador origem para o registrador físico apropriado, seguindo a convenção de passagens de argumentos. Uma complexidade a mais pode aparecer quando um argumento *float* ou *double* precisa ser passado em um registrador inteiro. Nesta situação, devem ser geradas as operações `aMFC1` e `aMFC1D`, respectivamente antes de `aMOVS` e `aMOVD`.

Como visto anteriormente, algumas operações podem utilizar um operando constante ou um registrador. As regras *rc* fazem essse papel:

```
rc:reg  (não gera operação assembly)
rc:con  (não gera operação assembly)
```

As regras *rc5* são utilizadas para reconhecer árvores de expressão em XOR que denotam uma constante de 5 *bits* ou um registrador:

```
rc5:nCONI (não gera operação assembly)
rc5:reg   (não gera operação assembly)
```

O valor da constante em `nCONI` é testado para ver se ele está entre zero e 31. Caso esteja entre zero e 31, a regra recebe o custo 1. Se não, ela recebe o máximo custo possível, para

que ela não seja selecionada.

As constantes são reconhecidas pelas seguintes regras:

```
con:nCONC   (não gera operação assembly)
con:nCONS   (não gera operação assembly)
con:nCONI   (não gera operação assembly)
con:nCONU   (não gera operação assembly)
```

Esta descrição da máquina MIPS R3000 ainda não está completa, podendo sofrer alterações. Os testes que serão descritos na seção 7.1 trazem mais detalhes do que ainda deve ser acrescentado à geração de Código MIPS R3000 e, naturalmente, isto implica em modificações na descrição da máquina.

6.6 Alocação de Registradores

Como descrito no capítulo anterior, é necessário implementar um módulo de alocação de registradores (*Register Allocation*) específico para a arquitetura alvo. O alocador de registradores deve atribuir aos registradores virtuais, algum registrador físico ou alguma célula de memória.

O alocador de registradores implementado para a geração de código MIPS R3000 é simples. Ele atribui uma célula de memória para cada registrador virtual, ou seja, não há alocação de registradores físicos. A alocação de registradores físicos não fazia parte do escopo desta dissertação, mas devido a necessidade de testar os programas gerados, adotou-se a estratégia de se colocar todos os registradores virtuais na memória. Mais especificamente, a estratégia foi reservar na pilha, espaço para armazenar todos os registradores virtuais do procedimento. Os registradores virtuais ficam armazenados na *área de variáveis locais* da pilha, como ilustrado na figura 6.6. A pilha de execução de procedimentos será melhor explicada na seção 6.7.

Para todo procedimento, instanciado na memória por um objeto `MIPSProcedure`, o alocador de registradores percorre a lista de instruções *assembly* do procedimento, verificando quais os operandos das instruções são registradores virtuais. Se o operando é um registrador virtual, a ele é atribuído um *slot* na pilha. É verificado também se o

registrador virtual já recebeu um *slot*, para que não sejam dados dois *slots* diferentes ao mesmo registrador. Para os registradores virtuais de inteiros, os *slots* variam de 0 até $n - 1$, onde n é o número de registradores virtuais de inteiros do procedimento. Para os registradores virtuais de ponto-flutuante, os *slots* variam de 0 até $m - 1$, onde m é o número de registradores virtuais de ponto-flutuante do procedimento. O tamanho da área a ser alocada na pilha para armazenar os resgistradores virtuais é dado por: $n * 4 + m * 8$. Cada registrador de inteiros ocupa 4 *bytes* e um registrador de ponto-flutuante ocupa 8 *bytes*.

Como o endereço base da pilha ($sp), o tamanho do *frame* e o tamanho da área de *variáveis locais* (ver figura 6.6) são conhecidos, o endereço do registrador virtual na pilha pode ser calculado através de uma operação simples. No caso de o registrador ser de inteiros, o cálculo é dado por:

$$\text{endereço} \leftarrow \$sp + \text{frame} - \text{slot} * 4$$

Se o registrador for de ponto-flutuante, o cálculo é dado por:

$$\text{endereço} \leftarrow \$sp + \text{frame} - \text{área de variáveis locais} + \text{slot} * 8$$

Da forma como os cálculos acima foram realizados, pode-se concluir que os registradores virtuais de inteiros são agrupados e armazenados nos endereços mais altos da *área de variáveis locais*, enquanto os registradores virtuais de ponto-flutuante são armazenados nos endereços mais baixos.

6.7 Emissão de Código *Assembly*

A emissão de código *assembly* é a última fase do processo de geração de código no compilador Xingó e, pode-se dizer, do próprio processo de compilação. Esta fase segue a recomendação feita no roteiro de construção de novos geradores de código. A recomendação é que para a emissão do código *assembly*, o desenvolvedor chame o método `Print` do objeto que corresponde ao arquivo *assembly* que, iterativamente, chama os métodos `Print` dos objetos de um nível imediatamente abaixo na hierarquia da representação XAR (ver figura 5.6 do capítulo anterior). No caso, o objeto que corresponde ao arquivo *assembly* é instanciado a partir da classe `MIPSFile`.

Um objeto `MIPSFile` contém uma tabela com todos os símbolos (inicializados, não inicializados, constantes e dados externos) do arquivo e aponta para uma lista contendo todos os procedimentos do arquivo. Os procedimentos (objetos `MIPSProcedure`), por sua vez, contêm uma lista de operações, ou instruções, *assembly* que devem ser emitidas para o arquivo de saída.

O algoritmo 6.1 expressa a emissão de código a partir do método `Print` de `MIPSFile`. Este e os demais algoritmos serão descritos em linguagem bem próxima à utilizada em todo o texto, de modo que pretende-se que eles sejam auto-explicativos. Estes algoritmos relatam, de forma simplificada, a maneira como o arquivo *assembly* é gerado.

Algoritmo 6.1: Método `Print` de `MIPSFile`

Emita ".set reorder"
Emita Segmento de Dados Inicializados
Emita Procedimentos
Emita Segmento de Dados não Inicializados
Emita Segmento de Dados Externos
Emita Segmento de Dados Constantes

Inicialmente, é emitida a *string* ".set reorder"para o arquivo *assembly* (arquivo de saída). Esta *string* corresponde a uma diretiva *assembly*. A opção *reorder* deixa o montador reordenar as instruções da linguagem de máquina para tentar melhorar o desempenho do programa gerado. De outro modo, poderia ser emitida a diretiva ".set noreorder", que não permitiria ao montador, reordenar as instruções. Depois são chamados métodos que emitem, respectivamente, o segmento de dados inicializados, os procedimentos do arquivo, os dados não inicializados, os dados externos (dados definidos em outros arquivos) e os dados constantes. A seguir, são listados os algoritmos referentes à emissão dos diversos segmentos de dados do arquivo.

Algoritmo 6.2: Emissão do segmento de dados inicializados
Para todas as variáveis globais do arquivo **faça**
Se variável está no segmento de dados inicializados
Então
Se variável não tem qualificador *static*
Então Emita diretiva ".globl"
Emita diretiva ".data"
Emita diretiva ".align"
Emita nome da variável seguido de ":"
Emita inicialização da variável usando as diretivas
".byte", ".half", ".word"ou ".double"

As diretivas que aparecem nos algoritmos requerem alguns parâmetros, que não serão completamente enumerados aqui. Os detalhes destas diretivas podem ser encontrados na literatura afim [25].

Algoritmo 6.3: Emissão do segmento de dados não inicializados
Para todas as variáveis globais do arquivo **faça**
Se variável está no segmento de dados não inicializados
Então
Se variável tem qualificador *static*
Então
Emita diretiva ".lcomm"
Senão
Se variável não tem qualificador *extern*
Então
Emita diretiva ".globl"
Emita diretiva ".comm"

Algoritmo 6.4: Emissão do segmento de dados extenos
Para todas as variáveis globais do arquivo **faça**
Se variável está no segmento de dados não inicializados
Então
Se variável tem qualificador *extern*
Então
Emita diretiva ".extern"

Algoritmo 6.5: Emissão do segmento de dados constantes
Para todas as variáveis globais do arquivo **faça**
Se variável está no segmento de dados constantes
Então
Emita diretiva ".rdata"
Emita diretiva ".align"
Emita nome da variável seguido de ":"
Emita inicialização da variável usando as diretivas
".byte", ".half", ".word"ou ".double"

A Emissão de código para os procedimentos foi deixada por último porque ela é mais complexa que as anteriores. Para cada procedimento, deve ser reservado espaço na pilha e devem ser emitidos códigos para o *prolog* (início do procedimento), para as instruções e para o *epilog* (final do procedimento). Tudo isso deve ser realizado de acordo com as convenções da arquitetura MIPS R3000 [25], que inclui, por exemplo, a forma como passar e receber argumentos através da pilha. A figura 6.6 mostra o *layout* de uma pilha MIPS.

Para entender melhor, considere o *frame*[5] do procedimento em execução. Este *frame* está em destaque (bordas mais largas) na figura. O *frame* de um procedimento tem local para armazenar variáveis locais e temporárias onde, pela implementação feita aqui neste trabalho, foram armazenados os registradores virtuais. A área *Registradores Salvos* é utilizada para salvar o conteúdo de registradores quando o procedimento for chamar algum outro procedimento. Na área *argumentos de saída*, que tem os endereços mais baixos do *frame*, devem ser passados os argumentos para o procedimento chamado. No

[5]O frame de um procedimento é a área da pilha que lhe é reservada.

caso de não ser necessário passar argumentos para nenhum dos procedimentos chamados, essa área poderá ter tamanho zero. Caso contrário, o tamanho dessa área tem que ter no mínimo 16 *bytes*, e deve ser um múltiplo de 8 *bytes*. Mesmo que se passe, por exemplo, apenas 4 *bytes* de argumentos, devem ser reservados 16 *bytes*. Os argumentos de entrada para a função corrente ficam no *frame* do procedimento chamador.

O algoritmo a seguir expressa a função que emite código para um procedimento.

Algoritmo 6.6: Emissão de um procedimento

Emita diretiva ".text"
Emita diretiva ".align"
Se procedimento não tem qualificador *static*
Então
Emita diretiva ".globl"

{Prolog — Início do Procedimento}
Emita diretiva ".ent"
Emita nome do procedimento seguido de ":"
Emita diretiva ".frame"
Emita diretiva ".set noreorder"
Emita diretiva ".cpload"
Emita diretiva ".set reorder"
Se tamanho do $frame > 0$
Então
cria pilha
salva registradores
{Código — operações do procedimenot}
Emita Instruções Assembly
{Epilog — Fim do Procedimento}
Restaura registradores salvos
Limpa a pilha criada
Emita instrução j $31 {retorna para o procedimento chamador}
Emita diretiva ".end"

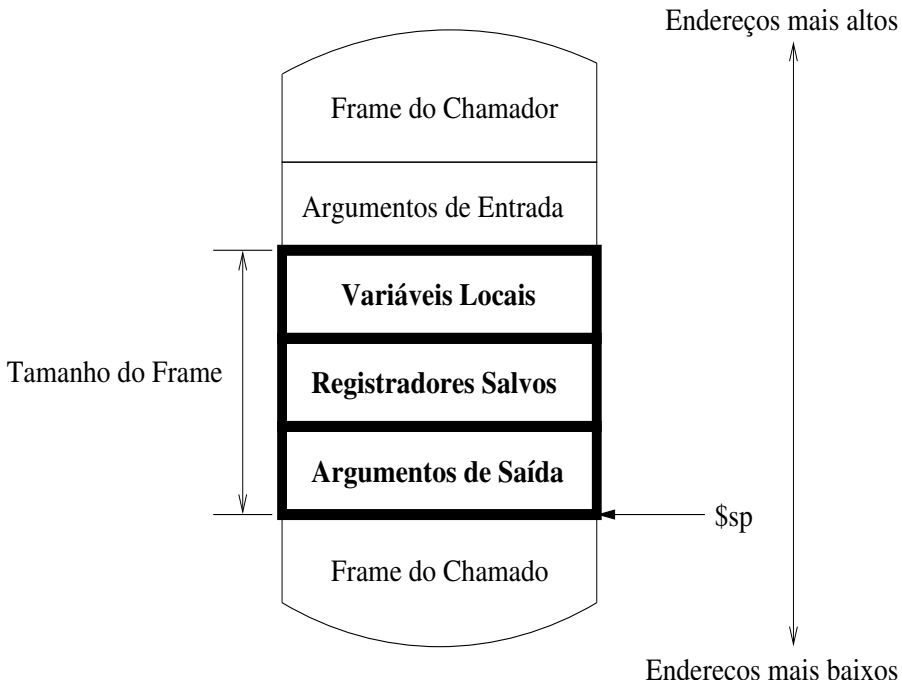

Figura 6.6: Layout da pilha MIPS

A emissão de código para as instruções *assembly* segue a seguinte lógica. Quando um registrador virtual é usado como operando fonte de uma instrução, o seu conteúdo, que está na pilha, é carregado para um registrador físico. Quando ele é usado como operando destino, o valor computado pela instrução é antes colocado em um registrador físico, que depois terá o seu conteúdo carregado para o endereço da pilha associado ao registrador virtual de destino. Por exemplo, na instrução *add r1,r2,r3*, onde r_i é um registrador virtual, são emitidas as instruções mostradas na figura 6.7.

O registrador virtual *r2*, que no caso tem *offset* 36 na pilha, é carregado para o registrador \$9. O registrador virtual *r3* (*offset* 40 na pilha) é carregado para o registrador \$10. A instrução *add* é computada utilizando como operandos fonte, os registradores \$9 e \$10, e o resultado é armazenado em \$8. Em seguida, o conteúdo de \$8 é carregado para o *offset* 28 da pilha, que no caso corresponde ao endereço associado ao registrador virtual *r1*. Como foi dito na seção 6.6, durante a alocação de registradores, é atribuído a cada registrador virtual, um *offset* na pilha. Utilizando o endereço base da pilha e o *offset*

associado ao registrador virtual, fica simples encontrar o seu endereço real. O endereço real é dado pela soma do endereço base da pilha e do *offset*. Como mostrado na figura 6.6, o registrador $sp contém o endereço base da pilha.

Para os demais operandos (variáveis, valores imediatos e os próprios registradores físicos), a emissão de instruções *assembly* é direta, basta emitir o nome da variável, o valor imediato ou o nome do registrador físico.

```
lw $9, 36($sp)
lw $10, 40($sp)
add $8, $9, $10
sw $8, 28($sp)
```

Figura 6.7: Instruções *assembly* emitidas para add r1, r2, r3 (registradores virtuais)

Os registradores físicos escolhidos para substituir os registradores virtuais são os registradores temporários disponíveis na arquitetura (ver tabela 6.2). Normalmente, tem-se utilizado os registradores $9 e $10 para registradores virtuais de inteiros, quando estes são operandos fonte. Quando eles são operandos destino, o registrador físico escolhido é o $8. Para registradores virtuais de ponto-flutuante, os registradores normalmente escolhidos são $f4 (para operando destino), $f6 (para o primeiro operando fonte) e $f8 (para o segundo operando fonte). Quando estes 6 registradores não são suficientes para emitir uma instrução representada em XAR, os outros registradores temporários são utilizados.

Capítulo 7

Conclusões

7.1 Resultados Experimentais

Este trabalho teve como foco principal, a implementação das principais representações de programa no compilador Xingó, juntamente com os módulos que fazem a transformação de um programa, em uma representação origem, para um programa em uma representação destino.

Os testes realizados até o presente momento (janeiro de 2005) se dividem em dois grupos:

- Avaliação da Representação Intermediária Xingó (XIR) (descrita no capítulo 3) e do Gerador de Código Intermediário (*XIR Generator*) (visto no capítulo 4).

- Testes para avaliar a síntese automática de um gerador de código. Esta síntese é parte das características da infra-estrutura de geração de código no Xingó (descrita no capítulo 5). Para avaliar esta infra-estrutura, foram realizados testes sobre o gerador de código MIPS R3000 (visto no capítulo 6), que já validam parte da infra-estrutura de geração de código e abrem caminho para o desenvolvimento de novos geradores de código.

Os testes estão sendo realizados utilizando-se basicamente três *benchmarks*: *NullStone* [4], *Mediabench* [26] e *SPEC* [11].

O *benchmark NullStone* (www.nullstone.com) realiza uma série de testes com o objetivo de avaliar o desempenho e a corretude do compilador. O *NullStone* é reconhecido

como o melhor *benchmark* de compiladores de produção do mundo e foi adquirido com o apoio do projeto FAPESP 00/15083-9.

Os *benchmarks Mediabench* e *SPEC* estão sendo utilizados principalmente com o objetivo de avaliar as corretudes do Código Intermediário Xingó e do Gerador de Código C. Os programas de entrada do *Mediabench* e *SPEC* apresentam uma complexidade sintática maior que a apresentada pelos programas de entrada do *NullStone*.

7.1.1 Código Intermediário Xingó

A primeira parte dos testes foi realizada para avaliar a representatividade do Código Intermediário Xingó. Com estes testes, é possível analisar o *front-end* do LCC, a Representação Intermediária Xingó (XIR) e o Gerador de Código Intermediário (*XIR Generator*). Mais que isso, é possível analisar a integração de todos eles. Como visto no capítulo 4, *XIR Generator* é responsável por produzir a representação intermediária do programa que está sendo compilado. Para isso, ele faz uma interface entre LCC e Xingó, convertendo a Representação Intermediária do LCC para a Representação Intermediária do Xingó.

Se o módulo *XIR Generator* falha, o Xingó não pode garantir a precisão de nenhuma de suas fases subseqüentes, tais como otimização de código, geração de código C ou geração de código *assembly*, então o porquê da maior parte dos testes realizados neste trabalho ter se concentrado no intuito de deixar o módulo *XIR Generator* o mais correto e estável possível.

Para testar a corretude do Código Intermediário Xingó, tem se utilizado o Gerador de Código C implementado no trabalho de mestrado de W. Attrot [7]. Avaliar a corretude da Representação Intermediária Xingó, apesar de ela ser legível, é uma tarefa difícil de ser realizada para programas de entrada muito grandes (com muitas linhas de código), a menos que ela seja convertida em um programa que possa ser executado.

O gerador de código C gera, a partir da XIR, um código fonte em C, que pode ser compilado (por exemplo, usando-se o GCC) e executado. A geração de código C pode ser realizada porque a Representação Intermediária Xingó apresenta uma sintaxe muito próxima à da linguagem C. Se o código C gerado produzir o resultado esperado, significa que o LCC, a Representação Intermediária Xingó e o Gerador de Código Intermediário estão funcionando corretamente para aquele programa que está sendo compilado. Caso o código C gerado não produza o resultado esperado, o erro pode estar no LCC, na XIR,

no *XIR Generator* ou no próprio Gerador de Código C. Assim, em casos de erros, todos estes módulos precisam ser analisados.

Boa parte dos resultados de geração de código C já havia sido apresentada em março de 2004, na referida dissertação de mestrado de W. Attrot [7]. Tais resultados já eram muito animadores para aquele estágio do compilador. Mas o compilador Xingó continuou evoluindo, e os resultados a serem apresentados agora são ainda melhores.

A geração de código C foi bem sucedida em 100% dos testes realizados com o *NullStone*. Isso significa que para todos os programas do *NullStone* (6611 programas), o Código Intermediário Xingó está correto. Como já havia sido dito por W. Attrot [7], entende-se por geração de código C bem sucedida, a correta compilação e execução do código C gerado pelo Xingó. A tabela 7.1 ilustra os resultados conseguidos utilizando o *benchmark NullStone*.

Observando a lista de otimizações realizadas pelo *NullStone*, é possível deduzir um pouco das características dos programas de entrada deste *benchmark*. Estes programas apresentam campos de *bits* (*BitField Optimization*), *switchs* (*Unswitching* e *Cross Jumping*) que levam a *jump tables*, ponteiros (*pointer optimization*), dentre outras características.

Estes mesmos testes feitos com o *NullStone* usando-se Xingó também foram realizados utilizando-se o GCC [17] (versão 3.2). O GCC apresentou 6 erros de compilação, quando testada a otimização *CSE optimization*.

A geração de código C para o *MediaBench* foi bem sucedida para os programas *ADPCM, EPIC, JPEG* e *PEGWIT*. Tais programas foram testados tanto com a aplicação como sem a aplicação de otimizações. A geração de código C para os programas *GSM* e *RASTA* ainda não está sendo bem sucedida. Como dito anteriormente, o erro pode estar no LCC, na XIR, no *XIR Generator* ou no próprio Gerador de Código C. O foco do erro está sendo investigado. *MPEG* ainda não foi testado. A tabela 7.2 ilustra os resultados obtidos utilizando o *MediaBench*.

Otimizações	**Programas de Testes**	**Erros de Compilação**	**Erros de Execução**	**Total de Erros**
Alias (by byte)	66	–	–	–
Alias (const-qualified)	9	–	–	–
Alias (by address)	44	–	–	–
Alias (scalar vs array)	20	–	–	–
BitField Optimization	3		–	–
Branch Elimination	15	–	–	–
Instruction Combining	2008	–	–	–
Constant Folding	56	–	–	–
Constant Propagation	15	–	–	–
CSE Elimination	2128	–	–	–
Dead Code Elimination	256	–	–	–
Integer Divide Optimization	92	–	–	–
Expression Simplification	163	–	–	–
If Optimization	69	–	–	–
Function Inlining	33	–	–	–
Induction Variable Elimination	4	–	–	–
Strength Reduction	2	–	–	–
Hoisting	38	–	–	–
Loop Unrolling	16	–	–	–
Loop Collapsing	4	–	–	–
Loop Fusion	2	–	–	–
Unswitching	22	–	–	–
Block Merging	1	–	–	–
Cross Jumping	4	–	–	–
Integer Modulus Optimization	92	–	–	–
Integer Multiply Optimization	99	–	–	–
Address Optimization	22	–	–	–
Pointer Optimization	15	–	–	–
Printf Optimization	3	–	–	–
Forward Store	99	–	–	–
Value Range Optimization	30	–	–	–
Tail Recursion	4	–	–	–
Register Allocation	47	–	–	–
Narrowing	3	–	–	–
SPEC Conformance	2	–	–	–
Static Declarations	1	–	–	–
String Optimization	4	–	–	–
Volatile Conformance	125	–	–	–
Totais	**6611**	**0**	**0**	**0**

Tabela 7.1: Resultados do código intermediário Xingó para o *NullStone*

Programa	Sem Otimização	Com Otimização
ADPCM	OK	OK
EPIC	OK	OK
G721	OK	OK
GSM	Fail	–
JPEG	OK	OK
MPEG	–	–
RASTA	Fail	–
PEGWIT	OK	OK

Tabela 7.2: Resultados do código intermediário Xingó para o *MediaBench*

Para os programas do *benchmark SPEC*, os testes têm sido realizados sem a aplicação de nenhuma otimização. Os programas *bzip2, gzip, mcf* e *twolf* tiveram a geração de código C bem sucedida. Outros programas pertencentes ao *SPEC* ainda estão sendo avaliados.

Um problema enfrentado na utilização dos *benchmarks Mediabench* e *SPEC* é que nem todos os seus programas são aceitos pelo LCC, *front-end* do Xingó. Uma vez que um programa não passe pelo LCC, o Xingó não tem como compilá-lo.

O compilador Xingó está em constante desenvolvimento, pois tem como objetivo de longo prazo alcançar um desempenho igual ou superior ao do compilador GCC [17], no que se refere a otimização de código, bem como no número de programas aceitos. Para isto, a cada novo programa que não é aceito pelo compilador, os devidos ajustes estão sendo realizados. Também estão sendo introduzidas modificações no LCC, de modo que ele reconheça como corretos uma gama maior de programas de entrada.

7.1.2 Síntese Automática de um Gerador de Código

Os testes que são apresentados nesta seção foram realizados com o intuito de avaliar a síntese automática de um gerador de código pelo Xingó. Esta síntese, como vista no capítulo 5, é feita usando-se o Olive [45]. Para testar essa síntese automática, foi implementado um gerador de código para o processador MIPS R3000 [25].

Para validar o gerador de código MIPS R3000 (descrito no capítulo 6) e, conseqüentemente, avaliar o Olive e toda a Infra-estrutura de Geração de Código no Xingó, uma série

de testes está sendo iniciada no compilador Xingó. Os resultados que são apresentados aqui são resultados iniciais, pois o Gerador de Código MIPS R3000 ainda se encontra na sua versão inicial, a qual passará ainda por diversos testes. Como foi dito, o Xingó está em constante desenvolvimento.

Durante a realização dos testes, um problema apareceu. Este problema foi a falta de um processador MIPS R3000 para executar os programas gerados. Na ausência de um processador MIPS, foi utilizado um simulador para o MIPS. O simulador foi implementado no Projeto ArchC [42], também desenvolvido pelo Laboratório de Sistemas de Computação (LSC) da UNICAMP, que é o mesmo laboratório que desenvolve o Projeto Xingó. Além do simulador, o ArchC fornece também um compilador GCC *cross-compiler* para o MIPS, o que permite montar os programas *assembly* gerados.

Os testes realizados não garantem a corretude completa do Gerador de Código MIPS R3000, mas consolidam parte das funcionalidades que ele deve conter. A metodologia utilizada para os testes foi a seguinte: testar algumas construções, ou algumas estruturas, que comumente aparecem em programas escritos na linguagem C (linguagem de entrada do compilador Xingó). Para isso, antes de testar o gerador de código para um programa grande e complexo, como os que pertencem aos *benchmarks NullStone, MediaBench* e *SPEC*, foram utilizados programas pequenos, que testam apenas uma determinada especificidade de um programa típico em C. Seguindo esta lógica, foram realizados testes para:

1. Variáveis globais inicializadas.
2. Variáveis globais não inicializadas.
3. Variáveis do tipo *extern* (definidas em outro arquivo).
4. Variáveis do tipo *static* (válidas apenas no escopo do arquivo onde foram declaradas).
5. valores imediatos.
6. Instruções aritméticas inteiras.
7. Comandos condicionais (if-then-else).

8. Estruturas de repetição (while, do while, for).
9. Passagem de argumentos.
10. Chamadas à funções declaradas pelo programador.
11. Chamadas à funções do sistema, como *printf*.
12. Chamadas recursivas.
13. Operações sobre *bits*.
14. Ponteiros para dados.

O Gerador de Código MIPS R3000 tem gerado código correto para as estruturas ou caracterísitcas relacionadas acima. Alguns outros testes, para outras estruturas de programas, ainda estão sendo realizados. Estes testes envolvem:

1. Aritmética de ponto-flutuante.
2. Operações que envolvem *Structs* e *Unions*.
3. Funções que retornam estruturas (*struct*).
4. *Jump Tables*.
5. Algumas conversões de tipos.
6. Ponteiros para funções.

Como resultados mais relevantes, dois testes importantes com *benchmarks* já foram bem sucedidos[1]. Estão sendo gerados códigos *assembly* MIPS corretos para os programas ADPCM e EPIC, do *MediaBench*. O ADPCM, inclusive, é executado corretamente pelo simulador MIPS utilizado nos testes. Os outros programas do *MediaBench*, do *SPEC* e do *NullStone* serão avaliados em breve e, logo, mais programas poderão ser corretamente compilados para o processador MIPS.

No atual estágio de geração de código *assembly*, o objetivo geral tem sido gerar um código correto e, não necessariamente, um código altamente eficiente. Futuramente, com

[1]Considere um programa bem sucedido para a geração de código *assembly*, aquele que teve o código *assembly* gerado corretamente.

um maior suporte à inclusão de otimizações de código dependente de máquina no compilador, os programas *assembly* gerados pelo Xingó poderão ter sua eficiência comparada à dos programas gerados por outros compiladores, como o GCC [17].

Como visto no capítulo 5, uma das contribuições deste trabalho foi o projeto e implementação da Infra-Estrutura de Geração de Código no Xingó. Uma parte importante desta infra-estrutura é o gerador de geradores de código Olive [45]. Olive foi adicionado ao Xingó para possibilitar a construção mais rápida de um novo gerador de código. Um dos objetivos específicos dos testes foi justamente avaliar a qualidade do Olive e o seu comportamento dentro da infra-estrutura montada no compilador Xingó. Os trabalhos publicados sobre Olive são poucos, e esta dissertação, assim acredita-se, também contribui para um melhor entendimento e avalição da ferramenta Olive.

Os testes realizados e os resultados obtidos podem comprovar a qualidade do Olive que, até o momento, tem-se mostrado estar perfeitamente integrado ao Xingó. Com isso, a síntese automática de um gerador de código no Xingó fica mais confiável e a redirecionabilidade de código se torna mais rápida.

Quanto ao gerador de código MIPS R3000, ainda que incompleto, pode ser considerado um importante ponto de partida para novas pesquisas em geração de código utilizando Olive e Xingó. É importante dizer que o Gerador de Código MIPS R3000 é o primeiro gerador de código implementado no Xingó.

7.2 Contribuições desta Dissertação

O Xingó é um compilador redirecionável que tem como um de seus objetivos proporcionar uma infra-estrutura que forneça facilidades de pesquisas em otimização e geração de código.

Referente à infra-estrutura geral do compilador Xingó, esta dissertação contribuiu em dois grandes módulos do compilador: o *front-end* e o *back-end*. O foco deste trabalho foi projetar e implementar as principais representações internas de programa do compilador e os módulos que traduzem um programa, em uma representação origem, para um programa em uma representação destino. Estes módulos de tradução de código, juntamente com as representações de programa, auxiliam em quase todo o processo de conversão de um código de entrada em C para um código de saída em C ou em *assembly*.

No sentido de otimização, a contribuição vem do fato de que foi projetada e implementada a representação intermediária do compilador Xingó (XIR). A representação intermediária é necessária em compiladores otimizantes, pois o otimizador de código deve passar múltiplas vezes sobre o código do programa em compilação.

Quanto à geração de código, a infra-estrutura construída no compilador Xingó fornece facilidades no desenvolvimento de novos geradores de código. Além de modular, essa infra-estrutura integra o Olive, que pode produzir um novo gerador de código mais rapidamente, uma vez que o conjunto de instruções do processador alvo pode ser descrito através de uma descrição de máquina. Espera-se que esta infra-estrutura possa encorajar novas pesquisas em geração de código.

Uma outra contribuição importante deste trabalho é o fato de que a XIR pode ser convertida em um código C compilável. Isto se deve ao fato de que a XIR foi projetada para representar o programa numa sintaxe muito próxima à da linguagem C e, principalmente, pelo fato de que ela é de leitura simples, como foi observado no trabalho de W. Attrot [7].

Os resultados dos testes apresentados na seção 7.1 demonstram uma boa qualidade do Código Intermediário Xingó e validam parte da Infra-Estrutura de Geração de Código. Os testes mostram que o Código Intermediário está correto para todos os programas do *benchmark NullStone* e para uma quantidade razoável de programas dos *benchmarks MediaBench* e *SPEC*. Os testes também demonstram que, através do estudo de caso: geração de código MIPS R3000, a síntese automática de um gerador de código utilizando Olive e Xingó é possível e apresenta-se como uma solução interessante no sentido de redirecionablidade de código, mesmo os resultados sendo iniciais.

7.3 Trabalhos Futuros

Além de concluir o gerador de código MIPS R3000, entre os trabalhos futuros que podem ser realizados no Xingó, estão incluídos otimização de código dependente de máquina, alocação de registradores e geração de código para uma nova máquina alvo, como por exemplo, um processador x86. Estes trabalhos podem ajudar a validar a Infra-Estrutura de Geração de Código do Xingó e trazer novas soluções que facilitem o processo de geração de código redirecionável.

A geração de Código Intermediário Xingó utilizando outros *benchmarks* também pode ser relacionada como trabalho futuro pois, assim, pode-se ter maior garantia da corretude desse processo no compilador Xingó. Os *benchmarks* cogitados são o *MiBench* e o *c-tortute* (utilizado pelo GCC).

Como o LCC apresenta problemas no reconhecimento de alguns programas de entrada, as modificações no seu código (que já foram iniciadas) devem continuar, de modo que o Xingó reconheça o máximo de programas possíveis e possa vir a competir com outros compiladores, como o GCC.

Uma das modificações sugeridas no LCC, mas não devido aos problemas de reconhecimento de programas, é fazê-lo recuperar estruturas de alto nível, tais como *loops* e vetores. Se estas estruturas pudessem ser representadas na Representação Intermediária do Xingó, seria bastante útil para algumas otimizações de código que poderão ser implementadas futuramente.

No sentido de avançar na geração de código redirecionável, um trabalho futuro bastante interessante é gerar automaticamente a descrição de máquina utilizada pelo Olive. Por exemplo, pode-se tentar utilizar ArchC [42] para gerar uma descrição Olive e um simulador para uma máquina alvo.

Referências Bibliográficas

[1] A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *J. ACM*, 23(3):488–501, 1976.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.

[3] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Trans. Program. Lang. Syst.*, 11(4):491–516, 1989.

[4] NULLSTONE: Automated Compiler Performance Analysis. Disponível em http://www.nullstone.com. Acessado em 28/01/2005.

[5] J. Davidson Andrew W. Appel and N. Ramsey. The zephyr compiler infrastructure. Technical report, University of Virginia, http://www.cs.virgnia.edu/zephyr, 1998. Acessado em 28/01/2005.

[6] Andrew W. Appel. *Modern Compiler Implementation in C; Basic Techniques*. Cambridge Univ. Press, 1997. APP a 97:1 1.Ex.

[7] Wesley Attrot. Xingó - compilação para uma representação intermediária executável. Master's thesis, Instituto de Computação - UNICAMP, abril 2004.

[8] David G. Bradlee, Robert R. Henry, and Susan J. Eggers. The marion system for retargetable instruction scheduling. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementaion*, pages 229–240, 1991.

[9] Xingó Compiler. http://websoc.lsc.ic.unicamp.br/xingo. Acessado em 28/01/2005.

[10] K. D. Cooper and L. Torzon. *Engineering a Compiler*. Morgan Kaufmann, 2003.

[11] Standard Performance Evaluation Corporation. Disponível em http://www.spec.org. Acessado em 28/01/2005.

[12] A. Fauth. Beyond tool-specific machine descriptions. In *Code generation for embedded processors, Eds. Boston:kluwer*, 1995.

[13] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. In *Proc. European Design and Test Conf.*, pages 503–507, 1995.

[14] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[15] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[16] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering efficient code generators using tree matching and dynamic programming. Technical Report TR-386-92, 1992.

[17] GCC Home Page. GNU Project. Free Software Foundation (FSF). http://www.gnu.org/software/gcc/. Acessado em 28/01/2005.

[18] M. Ganapathi and J. Hennessy and C. N. Fischer. Surveyor's forum: Retargetable code generators. *ACM Comput. Surv.*, 15(3):283–284, 1983.

[19] Mahadevan Ganapathi, Charles N. Fischer, and John L. Hennessy. Retargetable compiler code generation. *ACM Comput. Surv.*, 14(4):573–592, 1982.

[20] R. Steven Glanville and Susan L. Graham. A new method for compiler code generation. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 231–254. ACM Press, 1978.

[21] The Standford SUIF Compiler Group. http://suif.stanford.edu. Acessado em 28/01/2005.

[22] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. ISDL: An instruction set description language for retargetability. In *Design Automation Conference*, pages 299–302, 1997.

[23] J. L. Henessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[24] Sungjoon Jung and Yunheung Paek. The very portable optimizer for digital signal processors. In *CASES '01: Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 84–92. ACM Press, 2001.

[25] Gerry Kane. *MIPS RISC architecture*. Prentice-Hall, Inc., 1988.

[26] C. Lee, M. Potkonjak, and W. H. Magione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Micro-30, 30th Annual International Symposium on Microarchitecture*, 1997.

[27] R. Leupers. Lance: A c compiler platform for embedded dsps. *Embedded Systems/Embedded Intelligence*, Feb. 2001.

[28] Rainer Leupers. HDL-based modeling of embedded processor behavior for retargetable compilation. In *ISSS*, pages 51–, 1998.

[29] Rainer Leupers. Code generation for embedded processors. In *ISSS '00: Proceedings of the 13th international symposium on System synthesis*, pages 173–178. IEEE Computer Society, 2000.

[30] J. Anderson et al. M. Hall. Multiprocessor performance with the suif compiler. *IEEE Computer*, 29(12):84–89, december 1996.

[31] Peter Marwedel. A retargetable microcode generation system for a high-level microprogramming language. In *MICRO 14: Proceedings of the 14th annual workshop on Microprogramming*, pages 115–123. IEEE Press, 1981.

[32] Peter Marwedel. The mimola design system: Tools for the design of digital processors. In *DAC '84: Proceedings of the 21st conference on Design automation*, pages 587–593. IEEE Press, 1984.

[33] Peter Marwedel. Mssv: Tree-based mapping of algorithms to predefined structures. Technical report, Computer Science Dpt., University of Dortmund, 1993.

[34] Peter Marwedel. Code generation for core processors. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 232–237. ACM Press, 1997.

[35] Peter Marwedel and Wolfgang Schenk. Cooperation of synthesis, retargetable code generation and test generation in the mss. In *European Design and Test Conf.* IEEE Computer Society Press, 1993.

[36] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., 1997.

[37] L. Nowak and P. Marwedel. Verification of hardware descriptions by retargetable code generation. In *DAC '89: Proceedings of the 26th ACM/IEEE conference on Design automation*, pages 441–447. ACM Press, 1989.

[38] Lothar Nowak. Graph based retargetable microcode compilation in the mimola design system. In *MICRO 20: Proceedings of the 20th annual workshop on Microprogramming*, pages 126–132. ACM Press, 1987.

[39] Pierre G. Paulin, Clifford Liem, Trevor C. May, and Shailesh Sutarwala. Codesyn: a retargetable code synthesis system. In *ISSS '94: Proceedings of the 7th international symposium on High-level synthesis*, page 94. IEEE Computer Society Press, 1994.

[40] Pierre G. Paulin and Miguel Santana. Flexware: A retargetable embedded-software development environment. *IEEE Design & Test of Computers*, 19(4):59–69, 2002.

[41] The SPAM Project. http://www.ee.princeton.edu/spam. Acessado em 28/01/2005.

[42] Sandro Rigo, Guido Araujo, Marcus Bartholomeu, and Rodolfo Azevedo. Archc: A systemc-based architecture description language. In *to appear in the 16th Symposium on Computer Architecture and High Performance Computing - Foz do Iguacu, Brazil*, October 2004.

[43] Moe Shahdad. An overview of vhdl language and technology. In *Proceedings of the 23rd ACM/IEEE conference on Design automation*, pages 320–326. IEEE Press, 1986.

[44] A. Sudarsanam. *Code Optimization Libraries For Retargetable Compilation For Embedded Digital Signal Processors*. Phd thesis, Princeton University Departament of EE, May 1998.

[45] S.W.K. Tjiang. An olive twig. Technical report, Synopsys Inc., 1993.

Apêndice A

Interface Pública da Xingo IR

A.1 Classe `File`

- **File(char* name)**
  Construtor da classe, que recebe como entrada o nome do arquivo.
- **~File()**
  Destrutor da classe.
- **char* Name()**
  Retorna o nome do arquivo.
- **ProcedureList* Procs()**
  Retorna um ponteiro para a lista contendo os procedimentos do arquivo.
- **SymbolTable* Table()**
  Retorna um ponteiro para a tabela de símbolos do arquivo (tabela de símbolos global).
- **void SetName(char* nm)**
  Altera o nome do arquivo.
- **Procedure* LookupProc(char* nm)**
  Dado o nome de um procedimento, a função procura e retorna um ponteiro para este procedimento. Se não existe nenhum procedimento com o nome dado, a função retorna NULL.

- **void RemoveProc(char* nm)**
  Remove da lista o procedimento cujo nome é dado por `nm`.

- **void InsertProc(Procedure* p)**
  Adiciona ao final da lista de procedimentos do arquivo, o procedimento apontado por `p`.

- **void Print(FILE* fp, int indent)**
  Imprime para o arquivo `fp`, na indentação `indent`, as informações sobre o arquivo.

A.2 Classe `Procedure`

- **Procedure(File* f)**
  Construtor da classe, que recebe como entrada um ponteiro para um objeto `File`(arquivo pai).

- **Procedure()**
  Destrutor da classe.

- **int Id()**
  Retorna o identificador do procedimento.

- **char* Name()**
  Retorna o nome do procedimento.

- **Type* ReturnType()**
  Retorna um ponteiro para o tipo do item retornado pelo procedimento.

- **int NumArgs()**
  Retorna a quantidade de argumentos do procedimento. Se o procedimento possui quantidade de argumentos variável, o valor retornado corresponde à quantidade fixa, que, de acordo com o padrão ANSI C, deve ser maior que zero.

- **VariableList* Params()**
  Retorna um ponteiro para a lista contendo os parâmetros do procedimento.

- **SymbolTable* Syms()**
  Retorna um ponteiro para a tabela de símbolos do procedimento.

- **ProcedureBody* Block()**
  Retorna um ponteiro para o corpo do procedimento.

- **File* Parent()**
  Retorna um ponteiro para o arquivo pai.

- **boolean IsExtern()**
  Retorna TRUE se o procedimento tem qualificador de escopo `extern`.

- **boolean IsStatic()**
  Retorna TRUE se o procedimento tem qualificador de escopo `static`.

- **boolean IsVariadic()**
  Retorna TRUE se o procedimento tem quantidade de argumentos variável.

- **boolean IsLeaf()**
  Retorna TRUE se o procedimento é um procedimento folha. Um procedimento folha é aquele que não faz chamadas a nenhum outro, ou a si próprio (chamadas recursivas).

- **int NumCalls()**
  Retorna a quantidade de chamadas feitas por este procedimento a outros. Se o valor retornado por este método é zero, o procedimento é folha, e uma chamada a `IsLeaf()` retornaria TRUE.

- **void SetExtern(boolean e)**
  Dado o argumento lógico, se igual a TRUE, determina que o procedimento deve ter qualificador de escopo `extern`. Se igual a FALSE, apenas determina que ele não é `extern`.

- **void SetStatic(boolean s)**
  Dado o argumento lógico, se igual a TRUE, determina que o procedimento deve ter qualificador de escopo `static`. Se igual a FALSE, apenas determina que ele não é `static`.

- **void SetVariadic(boolean v)**
  Dado o argumento lógico, se igual a TRUE, determina que o procedimento possui quantidade de argumentos variável. Se igual a FALSE, determina que todos os argumentos são bem definidos.

- **void SetNumCalls(int n)**
  Atualiza a quantidade de chamadas feitas a outros procedimentos.

- **void SetName(char* nm)**
  Atualiza o nome do procedimento.

- **void SetReturnType(Type* t)**
  Atualiza o tipo retornado pelo procedimento.

- **void InsertParam(Variable* sym)**
  Dado um ponteiro para um objeto `Variable`, este objeto é adicionado ao final da lista de parâmetros.

- **void SetProcBody(ProcedureBody* pb)**
  Dado um ponteiro para um objeto `ProcedureBody`, este objeto passa a representar o corpo do procedimento.

- **Type* ParamType(int n)**
  Dado o índice do parâmetro, retorna um ponteiro para o tipo deste parâmetro. O primeiro parâmetro tem índice zero, o segundo índice 1, e assim por diante. Antes de chamar este método, o desenvolvedor poderia certificar-se que existe o parâmetro de índice `n`. Para isso, pode chamar o método `NumParams()`.

- **Variable* LookupParam(int n)**
  Dado o índice do parâmetro, retorna um ponteiro para o objeto `Variable` que representa este parâmetro.

- **Variable* LookupParamId(int i)**
  Dado o identificador da variável que representa o parâmetro, retorna um ponteiro para essa variável.

- **void Print(FILE* fp, int indent)**
  Imprime para o arquivo `fp`, na indentação `indent`, as informações sobre o procedimento.

A.3 Classe `ProcedureBody`

- **ProcedureBody(Procedure* p)**
  Construtor da classe, que recebe como entrada um ponteiro para um objeto `Procedure`, que é o procedimento pai.

- **~ProcedureBody()**
  Destrutor da classe.

- **Procedure* Proc()**
  Retorna um ponteiro para o procedimento pai.

- **InstructionList* Body()**
  Retorna um ponteiro para a lista de instruções.

- **SymbolTable* Syms()**
  Retorna a tabela de símbolos do procedimento pai.

- **void InsertInstruction(Instruction* ins)**
  Dado um ponteiro para um objeto `Instruction`, este objeto é inserido ao final da lista de instruções.

- **void SetCFGraph(CFGraph* graph)**
  Dado um ponteiro para um objeto `CFGraph`, este objeto é determinado para ser o CFG do procedimento pai.

- **CFGraph* CFG()**
  Retorna um ponteiro para o CFG do procedimento pai.

- **void Print(FILE* fp, int indent)**
  Imprime para o arquivo `fp`, na indentação `indent`, as informações sobre o corpo de procedimento.

A.4 Classe `Instruction`

- **Instruction(ProcedureBody* pb)**
  Construtor da classe, que recebe como entrada um ponteiro para o corpo de procedimento que a instrução pertence.

- **~Instruction()**
  Destrutor da classe.

- **int OpCode()**
  Retorna o *opcode* da instrução.

- **unsigned int ConstPosition()**
  Retorna a posição da constante (imediato). Se o valor de retorno for zero, o imediato está no primeiro operando fonte. Se o valor de retorno for 1, o imediato está no segundo operando fonte.

- **NoteList* Notes()**
  Retorna um ponteiro para a lista contendo as anotações da instrução.

- **CFGNode* CfgNode()**
  Retorna um ponteiro para o bloco báscico da instrução.

- **void SetCFGNode(CFGNode* n)**
  Dado um ponteiro para um bloco báscico, este método faz a instrução apontar para este bloco básico.

- **int Operand(int index)**
  Retorna o valor do operando de índice `index`. O operando destino tem índice igual a zero, o primeiro operando fonte igual a 1, e o segundo operando fonte igual a 2. O valor retornado corresponde a um identificador de uma variável, a um identificador de uma *label*, a um identificador de um tipo, ou a um valor imediato.

- **Symbol* OperandAsSymbol(int index)**
  Retorna um ponteiro para um objeto `Symbol`, que corresponde ao operando de índice `index`. A diferença deste método em relação ao anterior é que este retorna

um ponteiro para o símbolo, enquanto o outro retorna um identificador, ou um imediato.

- **ProcedureBody* Proc()**
  Retorna um ponteiro para o corpo de procedimento pai.

- **void SetOpCode(int op)**
  Troca o `opcode` de uma instrução.

- **void SetOperand(int index, int value)**
  Atualiza o valor do operando de índice `index`.

- **void SetConstPosition(unsigned pos)**
  Atualiza a posição da constante (imediato) na instrução. Se o valor de `pos` for zero, indica que o imediato fica no primeiro operando fonte, se for 1, o imediato fica no segundo operando fonte.

- **int OperandsCount()**
  Retorna a quantidade de operandos fonte da instrução.

- **int NumArgs()**
  Retorna o número de argumentos do procedimento chamado pela instrução. Este método só deveria ser invocado se a instrução é CALL.

- **Type* ArgType(int index)**
  Retorna um ponteiro para o tipo do argumento de índice `index` do procedimento chamado pela instrução. Este método só deveria ser invocado se a instrução é CALL.

- **Type* OperandType(int index)**
  Retorna um ponteiro para o tipo do operando de índice `index`.

- **Symbol* Target()**
  Retorna um ponteiro para o símbolo que é o alvo de desvio da instrução. Este método só deveria ser invocado por uma instrução de desvio.

- **void SetTarget(Symbol* lab)**
  Dado um ponteiro para um objeto `Symbol`, este objeto passa a ser o alvo de desvio da instrução. Este método só deveria ser invocado por uma instrução de desvio.

- **Label* Lab()**
  Retorna um ponteiro para um objeto `Label`. Este método só deveria ser invocado por uma instrução com *opcode* igual a LABEL.

- **Variable* CallDst()**
  Retorna um ponteiro para o objeto `Variable` que recebe o valor de retorno do procedimento chamado pela instrução. Se este método retornar NULL, significa que o procedimento chamado pela instrução é do tipo `void`. Este método só deveria ser invocado se a instrução é CALL.

- **Variable* CallAddr()**
  Retorna um ponteiro para o objeto `Variable` associado ao procedimento chamado pela instrução. Este método só deveria ser invocado se a instrução é CALL.

- **Variable* Argument(int index)**
  Retorna um ponteiro para o objeto `Variable` que representa o parâmetro de índice `index` do procedimento chamado pela instrução. Este método só deveria ser invocado se a instrução é CALL.

- **Type* ResultType()**
  Retorna um ponteiro para o objeto `Type` que diz o tipo do resultado operado pela instrução.

- **void InsertNote(Note* n)**
  Dado um ponteiro para um objeto `Note`, este objeto é inserido ao final da lista de anotações da instrução.

- **boolean IsVariadic()**
  Retorna TRUE se o procedimento chamado pela instrução tem quantidade de argumentos variável. Caso contrário, retorna FALSE. Este método só deveria ser invocado se a instrução é CALL.

- **Type* CvrtFrom()**
  Retorna um ponteiro para o objeto `Type` que corresponde ao tipo do dado que está sendo convertido pela instrução.

- **Type* CvrtTo()**
  Retorna um ponteiro para o objeto `Type` que corresponde ao tipo para o qual o dado está sendo convertido pela instrução.

- **void Print(FILE* fp, int indent)**
  Imprime para o arquivo `fp`, na indentação `indent`, as informações sobre a instrução.

A.5 Classe `SymbolTable`

- **SymbolTable(SymbolTable* st = NULL)**
  Construtor da classe, que recebe como entrada um ponteiro para a tabela de símbolos pai.

- **~SymbolTable()**
  Destrutor da classe.

- **SymbolList* Symbols()**
  Retorna um ponteiro para uma lista contendo os símbolos armazenados na tabela.

- **TypeList* Types()**
  Retorna um ponteiro para uma lista contendo os tipos armazenados na tabela.

- **SymbolTable* Parent()**
  Retorna um ponteiro para a tabela de símbolos pai.

- **unsigned NextId()**
  Retorna o próximo número de identificação para um símbolo. A cada chamada a este método, o número retornado é o incremento do anterior. Se a tabela possui uma tabela pai, recursivamente é a tabela pai quem fornece o próximo identificador. Isso garante que o identificador retornado é sempre único.

- **unsigned CurrentId()**
  Retorna o identificador corrente.

- **void SetParent(SymbolTable* st)**
  Dado um ponteiro para um objeto `SymbolTable`, este passa a ser a tabela pai da tabela que invocou este método.

- **Type* NewType(int id, int prim, int size, int align, Type* ty = NULL)**
  Cria um novo tipo, o adiciona para a tabela e retorna um ponteiro para ele. O método recebe como entrada o identificador, o tipo primário, o tamanho, o alinhamento, e um ponteiro para o operando do tipo. Se o tipo não possui operando, `ty` não precisar ser passado.

- **Variable* NewVar(char *nm, int seg, Type *t = NULL)**
  Cria uma nova variável, a adiciona para a tabela e retorna um ponteiro para ela. O método recebe como entrada o nome, o segmento lógico, e um ponteiro para o tipo dessa variável. Se a variável ainda não tem um tipo definido, `t` não precisa ser passado. Este método não garante que a variável terá nome único dentro da tabela. Para isso, o desenvolvedor deve chamar o método seguinte.

- **Variable* NewUniqueVar(int seg, Type* t = NULL, char* base = NULL)**
  Cria uma nova variável, a adiciona para a tabela e retorna um ponteiro para ela. O método recebe como entrada o segmento lógico, um ponteiro para o tipo, e um prefixo para o nome a ser dado a essa variável. `t` e `base` poderiam não ser passados. Se `base` é passado, o nome da variável vai ser formado pelo prefixo `base` concatenado a uma seqüência de dígitos. Temos optado por não passar nenhum prefixo, deixando o método usar o prefixo *default*, que é "_t". Idealizamos que os símbolos gerados pelo compilador (aqueles símbolos que não são definidos no arquivo de entrada) começassem sempre com "_", e o "_t", no caso, significaria que o símbolo é um temporário gerado pela xingo IR. Manter essa nomenclatura padrão poderia facilitar a legibilidade da representação intermediária. Evidentimente que outros desenvolvedores podem querer usar outros prefixos.

  Este método garante que a variável criada terá nome único dentro da tabela.

- **Label* NewLabel(char *nm)**
  Cria uma nova *label*, a adiciona para a tabela e retorna um ponteiro para ela. A função recebe como entrada o nome dessa *label*. Este método não garante que a *label* terá nome único dentro da tabela.

- **Label* NewUniqueLabel(char *base = NULL)**
  Cria uma nova *label*, a adiciona para a tabela e retorna um ponteiro para ela. O

método recebe como entrada o prefixo para o nome a ser dado a essa *label*. `base` poderia não ser passado para o método. Se `base` é passado, o nome da *label* é a concatenação desse prefixo com uma seqüência de dígitos. Temos optado por não passar nenhum prefixo, deixando o método usar o prefixo *default*, "_L", que no caso, significaria que o símbolo é uma *label* gerada pela xingo IR. Como dissemos no método anterior, manter essa nomenclatura poderia facilitar a legibilidade da representação intermediária.

Este método garante que a *label* terá nome único dentro da tabela.

- **Type* LookupType(int id)**
  Dado o identificador do tipo, o método procura e retorna um ponteiro para o objeto `Type` que possui este identificador. Se o tipo não é encontrado na tabela, ele é procurado na tabela pai. O método retorna NULL se o tipo não for encontrado.

- **Symbol* LookupSym(char* name, symKinds k)**
  Dado o nome e a natureza do símbolo(SYMVAR ou SYMLABEL), o método procura e retorna um ponteiro para o objeto `Symbol` que tem o nome e a natureza desejados. Se o símbolo não é encontrado na tabela, ele é procurado na tabela pai. O método retorna NULL se o símbolo não for encontrado.

- **Symbol* LookupSym(int id, symKinds k)**
  Dado o identificador e a natureza do símbolo (SYMVAR ou SYMLABEL), o método procura e retorna um ponteiro para o objeto `Symbol` que tem o identificador e a natureza desejados. Se o símbolo não é encontrado na tabela, ele é procurado na tabela pai. O método retorna NULL se o símbolo não for encontrado.

- **Symbol* LookupSymId(int id)**
  Dado o identificador do símbolo, o método procura e retorna um ponteiro para o objeto `Symbol` que tem o identificador desejado. Se o símbolo não é encontrado na tabela, ele é procurado na tabela pai. O método retorna NULL se o símbolo não for encontrado.

- **Variable* LookupVar(char *name)**
  Dado o nome da variável, o método procura e retorna um ponteiro para o objeto

`Variable` que tem o nome desejado. Se a variável não é encontrada na tabela, ela é procurada na tabela pai. O método retorna NULL se a variável não for encontrada.

- **Variable* LookupVar(int id)**
  Dado o identificador da variável, o método procura e retorna um ponteiro para a o objeto `Variable` que tem o identificador desejado. Se a variável não é encontrada na tabela, ela é procurada na tabela pai. O método retorna NULL se a variável não for encontrada.

- **void AddSym(Symbol *s)**
  Adiciona na tabela o símbolo apontado por `s`.

- **void AddType(Type *t)**
  Adiciona na tabela o tipo apontado por `t`.

- **void RemoveSym(Symbol *s)**
  Remove da tabela o símbolo apontado por `s`. O símbolo é removido da tabela, não da memória.

- **void RemoveType(Type* t)**
  Remove da tabela o tipo apontado por `t`. O tipo é removido da tabela, não da memória.

- **void Print(FILE *fp, int indent)**
  Imprime para o arquivo `fp`, na indentação `indent`, as informações sobre a tabela.

A.6 Classe `Symbol`

- **Symbol(SymbolTable* st, char *nm)**
  Construtor da classe, que recebe como entrada um apontador para a tabela de símbolos, e o nome do símbolo.

- **Symbol()**
  Destrutor da classe.

- **int Id()**
  Retorna o identificador do símbolo.

- **char *Name()**
  Retorna o nome do símbolo.

- **SymbolTable* Parent()**
  Retorna um apontador para a tabela de símbolos a qual o símbolo pertence.

- **boolean IsGenerated()**
  Retorna TRUE se o símbolo é um temporário gerado pelo xingo.

- **boolean IsStatic()**
  Retorna TRUE se o símbolo é `static`.

- **boolean IsExtern()**
  Retorna TRUE se o símbolo é `extern`.

- **void SetStatic(boolean s)**
  Dado o valor lógico `s`, determina se o símbolo deve ser ou não `static`. Se o argumento dado é TRUE, o símbolo é definido para ser `static`.

- **void SetExtern(boolean e)**
  Dado o valor lógico `e`, determina se o símbolo deve ser ou não `extern`. Se o argumento dado é TRUE, o símbolo é definido para ser `extern`.

- **void SetGenerated(boolean g)**
  Dado o valor lógico `g`, determina se o símbolo foi ou não gerado pela xingo IR. Se o argumento dado é TRUE, o símbolo foi gerado.

- **void SetName(char *nm)**
  Atualiza o nome do símbolo.

- **boolean IsVar()**
  Retorna TRUE se o símbolo é uma variável.

- **boolean IsLabel()**
  Retorna TRUE se o símbolo é uma *label*.

- **virtual symKinds Kind() = 0**
  Método puramente virtual, que retorna a natureza do símbolo, SYMVAR (uma variável) ou SYMLABEL (uma *label*).

- **void RemoveFromTable(void)**
  Remove o símbolo da tabela a qual ele pertence. O símbolo só é removido da tabela, ele não é excluído da memória.

- **void AddToTable(SymbolTable* st)**
  Dado um apontador para uma tabela de símbolos, o método adiciona o símbolo para a tabela apontada. Se o símbolo já pertencia a alguma outra tabela, ele é removido daquela tabela e adicionado à tabela apontada por `st`.

- **void Print(FILE* fp, int indent)**
  Imprime para o arquivo `fp`, na indentação `indent`, as informações sobre o símbolo.

A.7 Classe `Variable`

- **Variable(SymbolTable* st, char *nm, int seg, Type* t= NULL)**
  Construtor da classe, que recebe como entrada um ponteiro para a sua tabela de símbolos, o seu nome, o seu segmento lógico, e um ponteiro para o seu tipo. Se o tipo da variável ainda não tiver sido definido, não é necessário passar o último argumento. Quando for definido, este pode ser atualizado usando o método `SetType`.

- **Variable()**
  Destrutor da classe.

- **Type* Typ()**
  Retorna um ponteiro para o tipo da variável.

- **int Align()**
  Retorna o alinhamento em *bytes* da variável.

- **int Size()**
  Retorna o tamanho em *bytes* ocupado pela variável.

- **int Offset()**
  Retorna o *offset* da variável. Geradores de código podem desejar saber o *offset* da variável na pilha.

- **NoteList* Notes()**
  Retorna um ponteiro para uma lista contendo as anotações da variável.
- **varKinds VarKind()**
  Retorna a especialidade da variável (GLOBAL, LOCAL, PARAM, TEMP, ou PROCDEF).
- **int Segment()**
  Retorna o segmento lógico da variável.
- **void SetType(Type* t)**
  Atualiza o tipo da variável.
- **void SetAlign(int n)**
  Atualiza o alinhamento em *bytes* da variável.
- **void SetSize(int n)**
  Atualiza o tamanho em *bytes* da variável.
- **void SetOffset(int n)**
  Atualiza o *offset* da variável.
- **void SetVarKind(varKinds k)**
  Atualiza a especialidade da variável.
- **void SetSegment(int seg)**
  Atualiza o segmento lógico da variável.
- **void SetAddrTaken(boolean isTaken)**
  Se `isTaken` for TRUE, indica que o endereço da variável é tomando.
- **boolean IsGlobal()**
  Retorna TRUE se a variável é global.
- **boolean IsLocal()**
  Retorna TRUE se a variável é local.
- **boolean IsParam()**
  Retorna TRUE se a variável é um parâmetro.

- **boolean IsTemp()**
  Retorna TRUE se a variável é um temporário.

- **boolean IsProcDef()**
  Retorna TRUE se a variável é uma definição de procedimento.

- **boolean IsAddrTaken()**
  Retorna TRUE se a variável teve o endereço tomado.

- **symKinds Kind()**
  Retorna sempre o valor SYMVAR.

- **void InsertNote(Note* n)**
  Dado um ponteiro para uma anotação, esta função insere este ponteiro para a lista de anotações da variável.

- **void Print(FILE *fp, int indent)**
  Imprime para o arquivo `fp`, na indentação `indent`, as informações sobre a variável.

A.8 Classe `Label`

- **Label(SymbolTable* st, char *nm)**
  Construtor da classe, que recebe como entrada um ponteiro para a tabela de símbolos, e o nome da *label*.

- **Label()**
  Destrutor da classe.

- **symKinds Kind()**
  Retorna sempre SYMLABEL.

- **void Print(FILE *fp, int indent)**
  Imprime para o arquivo `fp`, na indentação `indent`, as informações sobre a *label*.

A.9 Classe `Type`

- **Type(int identifier, int prim, int s, int a, Type* t = NULL)**
  Construtor da classe, que toma como entrada o identificador, o tipo primário, o tamanho, o alinhamento e o operando do tipo. O identificador do tipo poderia ser gerado automaticamente, mas não é feito porque já o definimos na programação da interface do lcc (ver capítulo 4). O último argumento não deve ser passado, caso o tipo não tenha operandos.

- **~Type()**
  Destrutor da classe.

- **int Id()**
  Retorna o identificador do tipo.

- **char* Name()**
  Retorna o nome do tipo.

- **int PrimType()**
  Retorna o tipo primário (operador).

- **int Size()**
  Retorna o tamanho em *bytes* ocupado por um item do tipo.

- **int Align()**
  Retorna o alinhamento em *bytes* de um item do tipo.

- **Type* Typ()**
  Retorna um ponteiro para um outro objeto `Type`, que representa o operando do tipo.

- **FieldList* Fields()**
  Retorna um ponteiro para a lista de campos de um tipo agregado. Este método só deveria ser chamado se o tipo representa uma *struct*, ou uma *union*.

- **boolean IsVariadic()**
  Retorna TRUE se o tipo é de uma função com quantidade de argumentos variável. Este método só deveria ser chamado se o tipo é de uma função.

- **TypeList* ParamsType()**
  Retorna um ponteiro para a lista com os tipos dos parâmetros de uma função. Este método só deveria ser chamado se o tipo é de uma função.

- **void SetName(char* nm)**
  Atualiza o nome do tipo.

- **void SetId(int identifier)**
  Atualiza o identificador do tipo.

- **void SetPrimType(int p)**
  Atualiza o tipo primário (operador) do tipo.

- **void SetSize(int s)**
  Atualiza o tamanho em *bytes* ocupado por um item do tipo.

- **void SetAlign(int a)**
  Atualiza o alinhamento em *bytes* de um item do tipo.

- **void SetVariadic(boolean v)**
  Se `v` é igual TRUE, informa que o tipo representa uma função que tem quantidade de argumentos variável. Este método só deveria ser invocado se o tipo representa uma função.

- **void SetType(Type* t)**
  Atualiza o operando do tipo. `t` é um ponteiro para um objeto `Type` que representa o operando.

- **void InsertField(Field* field)**
  Dado um ponteiro para um objeto `Field`, este é inserido ao final da lista de campos do tipo. Este método só deveria ser invocado se o tipo é agregado.

- **void InsertParamType(Type* t)**
  Dado um ponteiro para um objeto `Type`, este é inserido ao final da lista de tipos dos parâmetros. Este método só deveria ser invocado se o tipo é de uma função.

- **boolean IsPtr()**
  Informa se o tipo é um ponteiro.

- **boolean IsArray()**
  Informa se o tipo é um *array*(vetor).

- **boolean IsInt()**
  Informa se o tipo é *int*.

- **boolean IsFloat()**
  Informa se o tipo é *float*.

- **boolean IsStructured()**
  Informa se o tipo é agregado (*struct* ou *union*).

- **Type* Clone()**
  Cria e retorna um ponteiro para um objeto `Type`, que é uma cópia exata do tipo. A diferença é que a cópia tem outro endereço de memória.

- **Type* PredominantType(Type* t)**
  Dado um ponteiro para um objeto `Type`, este método retorna um ponteiro para o objeto passado como argumento, ou para o objeto que invocou o método. Este método retorna o tipo predominante. O tipo predominante é aquele que implicitamente comporta o outro.

- **void Print(FILE* fp, int indent)**q
  Imprime para o arquivo `fp`, na indentação `indent`, as informações sobre o tipo.

A.10 Classe `Field`

- **Field(Type* ty, char* nm, int offset, int size)**
  Construtor da classe, que toma como entrada um ponteiro para o tipo, o nome, o *offset* e o tamanho do campo. `size` poderia ser passado sempre zero se o campo não representa um campo de *bits*. De outro modo, `size` deve indicar a quantidade de *bits* do campo.

- **char* Name()**
  Retorna o nome do campo.

- **Type* Typ()**
  Retorna um ponteiro para tipo do campo.

- **int Offset()**
  Retorna o *offset* do campo.

- **int FieldSize()**
  Retorna a quantidade de *bits* do campo. Este método só deveria ser chamado para campos de *bits*.

- **void SetName(char* nm)**
  Atualiza o nome do campo.

- **void SetOffset(int offset)**
  Atualiza o *offset* do campo.

- **void SetType(Type* ty)**
  Atualiza o tipo do campo.

- **void SetFieldSize(int size)**
  Atualiza a quantidade de *bits* de um campo de *bits*.

- **boolean IsBitField()**
  Retorna TRUE se o campo é um campo de *bits*.

- **void Print(FILE* fp, int indent)**
  Imprime para o arquivo `fp`, na indentação `indent`, as informações sobre o campo.

A.11 Classe `Note`

- **Note(int identifier, ImmedList *d = NULL)**
  Construtor da classe, que recebe como entrada o identificador e um ponteiro para a lista contendo os dados da anotação. Se a anotação não tem dados, `d` não precisa ser passado.

- **~Note()**
  Destrutor da classe.

- **int Id()**
  Retorna o código de identificação da anotação.
- **ImmedList* Data()**
  Retorna um ponteiro para a lista contendo os dados da anotação.
- **void SetId(int n)**
  Altera o código de identificação da anotação.
- **void SetData(ImmedList *d)**
  Dado um ponteiro para uma lista de imediatos, a função faz essa lista representar os dados da anotação.
- **void InsertData(Immed *im)**
  Insere o valor imediato apontado por `im` no final da lista de dados da anotação.
- **void Print(FILE *fp, int indent)**
  Imprime para o arquivo `fp`, na indentação `indent`, as informações sobre a anotação.

A.12 Classe `Immed`

- **Immed(char val)**
  Construtor da classe, que cria um imediato do tipo *char*.
- **Immed(short int val)**
  Construtor da classe, que cria um imediato do tipo *short int*.
- **Immed(int val)**
  Construtor da classe, que cria um imediato do tipo *int*.
- **Immed(unsigned val)**
  Construtor da classe, que cria um imediato do tipo *unsigned*.
- **Immed(long int val)**
  Construtor da classe, que cria um imediato do tipo *long int*.
- **Immed(float val)**
  Construtor da classe, que cria um imediato do tipo *float*.

- **Immed(double val)**
  Construtor da classe, que cria um imediato do tipo *double*.

- **Immed(char* val)**
  Construtor da classe, que cria um imediato que aponta para uma cadeia de caracteres.

- **Immed(Symbol* val)**
  Construtor da classe, que cria um imediato que aponta para uma instância da classe `Symbol`.

- **boolean IsChar()**
  Retorna TRUE se o valor imediato é do tipo *char*.

- **boolean IsShort()**
  Retorna TRUE se o valor imediato é do tipo *short int*.

- **boolean IsInteger()**
  Retorna TRUE se o valor imediato é do tipo *int*.

- **boolean IsUnsigned()**
  Retorna TRUE se o valor imediato é do tipo *unsigned*.

- **boolean IsLong()**
  Retorna TRUE se o valor imediato é do tipo *long int*.

- **boolean IsFloat()**
  Retorna TRUE se o valor imediato é do tipo *float*.

- **boolean IsDouble()**
  Retorna TRUE se o valor imediato é do tipo *double*.

- **boolean IsString()**
  Retorna TRUE se o valor imediato é uma string.

- **boolean IsSymbol()**
  Retorna TRUE se o valor imediato é ponteiro para uma instância da classe *Symbol*.

- **char Char()**
  Retorna o valor imediato como sendo um *char*.

- **short int Short()**
  Retorna o valor imediato como sendo um *shortint*.

- **int Integer()**
  Retorna o valor imediato como sendo um *int*.

- **unsigned Unsigned()**
  Retorna o valor imediato como sendo um *unsigned*.

- **long int Long()**
  Retorna o valor imediato como sendo um *longint*.

- **float Float()**
  Retorna o valor imediato como sendo um $float$.

- **double Double()**
  Retorna o valor imediato como sendo um *double*.

- **char* String()**
  Retorna o valor imediato como sendo uma *string*.

- **Symbol* Symb()**
  Retorna o valor imediato como sendo um ponteiro para uma instância da classe `Symbol`.

- **void Print(FILE* fp, int indent)**
  Imprime para o arquivo `fp`, na indentação `indent`, as informações sobre o imediato.