# GERAÇÃO DE COMPILADORES BASEADA EM COMPONENTES

GABRIEL DE GODOY CORREA E CASTRO

# GERAÇÃO DE COMPILADORES BASEADA EM COMPONENTES

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais - Departamento de Ciência da Computação como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Roberto da Silva Bigonha
Coorientador: Fabio Tirelo

Belo Horizonte, MG

Agosto de 2014

Castro, Gabriel de Godoy Correa e

C355g Geração de Compiladores Baseada em Componentes / Gabriel de Godoy Correa e Castro. — Belo Horizonte, MG, 2014
xxi, 144 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais - Departamento de Ciência da Computação
Orientador: Roberto da Silva Bigonha
Coorientador: Fabio Tirelo

1. Computação - Teses. 2. Compiladores (Computadores) - Teses. 3. Linguagens de programação (Computadores). I. Orientador. II. Coorientador. III. Título.

CDU 519.6*33(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

# FOLHA DE APROVAÇÃO

Geração de compiladores baseada em componentes

**GABRIEL DE GODOY CORREA E CASTRO**

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. ROBERTO DA SILVA BIGONHA - Orientador
Departamento de Ciência da Computação - UFMG

PROF. FABIO TIRELO - Coorientador
Google

PROFA. MARIZA ANDRADE DA SILVA BIGONHA
Departamento de Ciência da Computação - UFMG

PROF. VLADIMIR OLIVEIRA DI IORIO
Departamento de Informática - UFV

Belo Horizonte, 21 de agosto de 2014.

# Resumo

A construção de um compilador de linguagens de porte real é um projeto grande e de alta complexidade. Dessa forma, são necessárias ferramentas que auxiliem esse processo de construção. Entretanto, os sistemas para a geração completa de um compilador ainda não se tornaram populares devido à sua complexidade de utilização, complicada pela baixa legibilidade de alguns sistemas e baixa reusabilidade de componentes de projetos similares.

Este trabalho de dissertação apresenta um ambiente de desenvolvimento de compiladores cujo enfoque está na legibilidade do código de implementação dos tradutores, com o objetivo de tornar o ambiente mais simples de utilizar. Esse ambiente realiza a tradução do código fonte para uma árvore de sintaxe abstrata e realiza, subsequentemente, ações semânticas para geração de código durante o caminhamento da AST gerada. O ambiente é auxiliado por uma infraestrutura de geração de código que encapsula conceitos fundamentais e recorrentes de importantes construções de linguagens de programação imperativas.

A infraestrutura é formada por componentes de ações semânticas, instruções de código intermediário e uma tabela de símbolos. Os componentes são utilizados pelo implementador da linguagem de programação para realizar as ações semânticas necessárias para a compilação das construções de sua linguagem. A tabela de símbolos é utilizada para controlar as declarações de variáveis utilizadas na infraestrutura e permite implementar diversas políticas de controle de escopo e alocação de memória.

O ambiente foi validado com a implementação do compilador da linguagem *Small*, aqui definida.

**Palavras-chave:** Linguagens de programação, compiladores, semântica.

# Abstract

The construction of a compiler of a real size programming language is a complex and big project. So, proper tools are needed to help in this task. However, compiler generation systems are still not popular yet. This is due to the complexity of these systems, complicated by their lack of readability and low reusability of language definitions used by other systems.

This dissertation presents a compiler development environment which is focused on the readability of the compiler code, making the environment simpler to use. This environment translates the source code to an abstract syntax tree and performs, subsequently, semantic actions to generate code while visiting the AST generated. The environment is supported by a code generator infrastructure that encapsulates fundamental and recurring concepts of the constructs of imperative programming languages.

The infrastructure consists of components of semantic actions, intermediary code instructions and a symbol table. The components are utilized by the programming language implementer to perform the semantic actions necessary to compile the language. The symbol table is used to control the declarations of the variables used in the infrastructure and allows the selection of different strategies for scope control and memory allocation.

The environment was validated with an implementation for the compiler of the language *Small*, defined herein.

**Keywords:** Programming languages, compilers, semantics.

# Lista de Figuras


1.1 Dependências das regras semânticas de cada construção de uma linguagem. 3

3.1 Diagrama de fluxo de dados das descrições de uma linguagem L. . . . . 32
3.2 Diagrama de link-edição do ambiente de desenvolvimento. . . . . 32
3.3 Diagrama de fluxo de dados da compilação de um programa P escrito em L. 33

4.1 Árvore de sintaxe abstrata do programa de *Small* da Listagem 4.3. . . . . 49
4.2 AST mostrando a geração da instrução de alocação. . . . . 50
4.3 AST mostrando a busca na tabela de símbolos. . . . . 50
4.4 AST mostrando a geração da instrução de carga de constante. . . . . 51
4.5 AST mostrando a geração da instrução de operação binária. . . . . 51
4.6 AST mostrando a geração da instrução de armazenamento. . . . . 51
4.7 AST mostrando a carga de uma variável. . . . . 52
4.8 AST mostrando a geração da instrução de argumento de função. . . . . 52
4.9 AST mostrando a geração da instrução de chamada de função. . . . . 53
4.10 AST mostrando o sequenciamento de instruções. . . . . 53

5.1 Árvore de *parsing* do programa **{float x; x = 1 + 1.0}**. . . . . 102
5.2 Árvore de sintaxe abstrata da construção **{float x; x = 1 + 1.0}**. . . . . 103
5.3 Árvore de sintaxe abstrata após primeira transformação de *Middle*. . . . . 106
5.4 Árvore de sintaxe abstrata após segunda transformação de *Middle*. . . . . 107

B.1 Árvore de *parsing* do programa da Listagem B.6. . . . . 138
B.2 AST inicial do programa da Listagem B.6. . . . . 139
B.3 AST resultante do primeiro programa de *Middle* sobre o programa da Listagem B.6. . . . . 139
B.4 AST resultante do segundo programa de *Middle* sobre o programa da Listagem B.6. . . . . 139

# Lista de Tabelas

2.1 Comparação de gramática de expressões. . . . . 17

3.1 Exemplos das transformações de *Front*. . . . . 37

# Lista de Listagens


2.1 Exemplo de um módulo de expressões no formalismo SDF. . . . . 8
2.2 Exemplo de um módulo de expressões em Stratego. . . . . 9
2.3 Exemplo de definição de uma calculadora em LISA. . . . . 10
2.4 Exemplo de gramática de expressões no sistema Eli. . . . . 12
2.5 Exemplo da avaliação de expressões no sistema JastAdd. . . . . 13
2.6 Exemplo de definição de escolha em Neverlang. . . . . 15
2.7 Exemplo da gramática de expressões em SableCC. . . . . 16
2.8 Exemplo da avaliação de expressões no sistema ANTLR. . . . . 19
2.9 Exemplo do reconhecimento de dígitos e letras em LEX. . . . . 21
2.10 Exemplo de uma grámatica de expressões em Yacc. . . . . 22
2.11 Modo simples para multiplicar uma variável por oito. . . . . 24
2.12 Modo simples otimizado para multiplicar uma variável por oito. . . . . 24
2.13 Modo complexo para multiplicar uma variável por oito. . . . . 24
2.14 Definição de uma estrutura de lista encadeada. . . . . 25
2.15 Exemplo de armazenamento. . . . . 26
2.16 Exemplo de chamada de função. . . . . 28
3.1 Exemplo de descrição da AST utilizada na descrição de uma calculadora. 34
3.2 Exemplo de descrição da sintaxe da calculadora. . . . . 38
3.3 Exemplo de descrição de *Middle* para declaração de variável e verificação de tipos . . . . . 40
3.4 Exemplo de descrição da geração de código de expressão . . . . . 41
4.1 Componente para sequência de instruções. . . . . 44
4.2 Método para ignorar os parâmetro de componentes. . . . . 44
4.3 Exemplo de um programa em *Small*. . . . . 48
4.4 Instruções da infraestrutura do programa *Small* da Listagem 4.3. . . . 48
4.5 Bytecode LLVM do programa *Small* da Listagem 4.3. . . . . 49
4.6 Componente para um literal inteiro. . . . . 54
4.7 Componente para um literal de ponto flutuante. . . . . 55

4.8 Exemplo da utilização do componente de literal. . . . . 55
4.9 Componente para a carga de um endereço calculado. . . . . 56
4.10 Componente para a recuperação do endereço de uma variável. . . . . 56
4.11 Exemplo da utilização dos componentes para recuperação e carga de uma variável. . . . . 56
4.12 Componente para cálculo do endereço base de uma posição de um arranjo. 57
4.13 Componente para cálculo do endereço de uma posição de um arranjo. . 57
4.14 Exemplo da utilização dos componentes para cálculo de índice de arranjo. 58
4.15 Componente para cálculo do endereço base de uma variável de uma base. 58
4.16 Componente para cálculo do endereço de uma variável de uma estrutura. 59
4.17 Exemplo da utilização dos componentes para cálculo membros de estruturas. . . . . 60
4.18 Componente para cálculo do endereço base de uma variável de uma classe. 60
4.19 Componente para cálculo do endereço de uma variável de uma classe. . 60
4.20 Exemplo da utilização dos componentes para cálculo membros de classes. 61
4.21 Método para criar a operação unária. . . . . 62
4.22 Componente para aplicação de operação únaria. . . . . 62
4.23 Exemplo da utilização dos componentes para aplicação de operação unária. 63
4.24 Componente para criar a operação binária. . . . . 63
4.25 Componente para aplicação de operação binária. . . . . 63
4.26 Exemplo da utilização dos componentes para aplicação de operação binária. . . . . 64
4.27 Componente para geração de chamada de função. . . . . 64
4.28 Exemplo da utilização do componente de chamada de função. . . . . 65
4.29 Componente para geração do argumento por valor da chamada de função. 65
4.30 Componente para geração do argumento por referência da chamada de função. . . . . 66
4.31 Componente para geração do argumento por nome da chamada de função. 66
4.32 Exemplo da utilização dos componentes de passagem de argumento. . . 67
4.33 Componente para geração de expressão condicional. . . . . 68
4.34 Exemplo da utilização dos componentes expressão condicional. . . . . 68
4.35 Componente para geração de alocação dinâmica de uma variável. . . . . 69
4.36 Exemplo da utilização do componente de alocação dinâmica. . . . . 69
4.37 Componente para geração do comando de atribuição. . . . . 70
4.38 Exemplo da utilização do componente de atribuição. . . . . 71
4.39 Componente para geração do comando condicional. . . . . 71
4.40 Exemplo da utilização do componente de comando condicional. . . . . 72

4.41 Componente para geração do comando de repetição. . . . . 73
4.42 Exemplo da utilização do componente do comando de repetição. . . . . 74
4.43 Componente para geração de retorno de função. . . . . 75
4.44 Exemplo da utilização do componente do comando de retorno de função. 76
4.45 Exemplo da utilização das palavras-chave para controle de escopo. . . . 77
4.46 Componente para geração da declaração de uma variável. . . . . 79
4.47 Componente para geração da alocação de uma variável. . . . . 79
4.48 Exemplo da utilização do componente de declaração de variável. . . . . 79
4.49 Exemplo da utilização do componente de alocação de variável. . . . . 80
4.50 Componente para geração da dimensão estática do arranjo. . . . . 80
4.51 Componente para declaração do arranjo estático. . . . . 81
4.52 Exemplo da utilização dos componentes de declaração de arranjo estático. 81
4.53 Componente para geração do dimensão do arranjo dinâmico. . . . . 82
4.54 Exemplo da utilização do componente de dimensão do arranjo dinâmico. 82
4.55 Componente para declaração do arranjo dinâmico. . . . . 83
4.56 Exemplo da utilização do componente de declaração de um arranjo dinâmico. . . . . 83
4.57 Método para definição de estruturas. . . . . 84
4.58 Exemplo da utilização do método de definição de uma estrutura. . . . . 84
4.59 Método para recuperação do tipo de estruturas definidas. . . . . 84
4.60 Componente para declaração de estruturas. . . . . 85
4.61 Exemplo da utilização do componente de declaração de uma estrutura. 85
4.62 Componente para listagem de parametros de função. . . . . 86
4.63 Componente para declaração de cabeçalho da função. . . . . 86
4.64 Exemplo da utilização dos componentes para declaração de uma função. 87
4.65 Componente para preenchimento do corpo da função. . . . . 87
4.66 Exemplo da utilização do componente para preechimento de uma função. 88
4.67 Componente para definição de classes. . . . . 89
4.68 Exemplo da utilização do método de definição de uma classe. . . . . 89
4.69 Método para recuperação do tipo de classes definidas. . . . . 89
4.70 Componente para declaração de classes. . . . . 89
4.71 Exemplo da utilização do componente de declaração de uma classe. . . 90
4.72 Método para recuperação do tipo de variável. . . . . 91
4.73 Exemplo da utilização do método de recuperação de tipo de variável. . 91
4.74 Método para recuperação do tipo do item armazenado no arranjo. . . . 91
4.75 Exemplo da utilização do método de recuperação de tipo de item de um arranjo. . . . . 91

4.76 Método para recuperação do tipo da estrutura mais interna. . . . . 92
4.77 Método para recuperação do tipo de estrutura. . . . . 92
4.78 Exemplo da utilização do método de recuperação de tipo de uma variável de uma estrutura. . . . . 93
4.79 Método para recuperação do tipo de retorno de função. . . . . 93
4.80 Exemplo da utilização do método de recuperação do tipo de uma função. 94
4.81 Método para recuperação do tipo da clase mais interna. . . . . 94
4.82 Método para recuperação do tipo de classe. . . . . 94
4.83 Exemplo da utilização do método de recuperação de tipo de um membro de uma classe. . . . . 95
4.84 Métodos para criação dos tipos primitivos. . . . . 95
4.85 Exemplo da utilização do método de criação de tipo um primitivo. . . . 96
5.1 Exemplo de programa na linguagem *Small*. . . . . 100
5.2 Descrição da árvore de sintaxe abstrata de *Small* na linguagem *AST*. . 101
5.3 Excerto da descrição de *Small* na linguagem *Front*. . . . . 104
5.4 Excerto da declaração de variáveis de *Small* na linguagem *Middle*. . . . 105
5.5 Excerto da verificação de tipos de *Small* na linguagem *Middle*. . . . . 107
5.6 Excerto da geração de código de *Small* na linguagem *Back*. . . . . 109
5.7 Outro excerto da geração de código de *Small* na linguagem *Back*. . . . 109
5.8 Exemplo da construção de repetição da linguagem Pascal. . . . . 110
A.1 Gramática da linguagem *AST* . . . . . 118
A.2 Gramática da linguagem *Front*. . . . . 120
A.3 Gramática da linguagem *Middle*. . . . . 122
A.4 Gramática da linguagem *Back*. . . . . 123
B.1 Descrição de Small utilizando a linguagem *AST*. . . . . 125
B.2 Descrição da sintaxe de Small utilizando a linguagem *Front*. . . . . 126
B.3 Descrição da declaração de variáveis de Small utilizando a linguagem *Middle*. . . . . 129
B.4 Descrição da verificação de tipos de Small utilizando a linguagem *Middle*. 130
B.5 Descrição da geração de código de Small utilizando a linguagem *Back*. . 134
B.6 Exemplo de programa na linguagem *Small*. . . . . 138
B.7 Instruções da infraestrutura do programa *Small* da Listagem B.6. . . . 140
B.8 Bytecode LLVM do programa *Small* da Listagem B.6. . . . . 140

# Sumário

**Resumo** **vii**

**Abstract** **ix**

**Lista de Figuras** **xi**

**Lista de Tabelas** **xiii**

**1 Definição do Trabalho** **1**

1.1 Objetivos . . . . . 4

1.2 Contribuições . . . . . 5

1.3 Organização da Dissertação . . . . . 5

**2 Estado da Arte em Geração de Compiladores** **7**

2.1 Sistemas de Implementação de Compiladores . . . . . 7

2.1.1 ASF+SDF . . . . . 8

2.1.2 Stratego/XT . . . . . 9

2.1.3 LISA . . . . . 10

2.1.4 Eli . . . . . 11

2.1.5 JastAdd . . . . . 12

2.1.6 Polyglot . . . . . 13

2.1.7 Neverlang . . . . . 14

2.1.8 SableCC . . . . . 16

2.1.9 Pappy . . . . . 16

2.1.10 ANTLR . . . . . 18

2.1.11 Lex/YACC . . . . . 20

2.2 Código Intermediário . . . . . 23

2.3 Conclusão . . . . . 28

**3 Ambiente de Desenvolvimento de Compiladores** **31**
3.1 Linguagens de Descrição . . . . . 33
3.2 Conclusão . . . . . 42

**4 Infraestrutura de Geração de Código** **43**
4.1 Tabela de Símbolos . . . . . 45
4.2 Instruções do Código Intermediário . . . . . 45
4.3 Expressões . . . . . 53
4.3.1 Valores Literais . . . . . 54
4.3.2 Carga de Variáveis . . . . . 55
4.3.3 Operações Binárias e Unárias . . . . . 61
4.3.4 Chamada de Função . . . . . 64
4.3.5 Condicional . . . . . 67
4.3.6 Alocação Dinâmica . . . . . 68
4.4 Comandos . . . . . 70
4.4.1 Atribuição . . . . . 70
4.4.2 Condicional . . . . . 71
4.4.3 Repetição . . . . . 72
4.4.4 Sequenciador de Retorno . . . . . 75
4.5 Escopo . . . . . 76
4.6 Declarações . . . . . 78
4.6.1 Variáveis . . . . . 78
4.6.2 Arranjo . . . . . 80
4.6.3 Estruturas . . . . . 83
4.6.4 Funções . . . . . 85
4.6.5 Classes . . . . . 88
4.7 Verificação de Tipos . . . . . 90
4.8 Conclusão . . . . . 96

**5 Avaliação do Ambiente** **99**
5.1 Legibilidade e Facilidade de Uso . . . . . 99
5.2 Generalidade dos Componentes . . . . . 110
5.3 Considerações Finais . . . . . 111

**6 Conclusão** **113**
6.1 Contribuições do Trabalho . . . . . 114
6.2 Trabalhos Futuros . . . . . 114

**A Linguagens do Sistema** **117**
A.1 Linguagem *AST* . . . . . 117
A.2 Linguagem *Front* . . . . . 119
A.3 Linguagem *Middle* . . . . . 121
A.4 Linguagem *Back* . . . . . 122

**B Compilador de Small** **125**
B.1 Árvore de Sintaxe Abstrata . . . . . 125
B.2 Sintaxe . . . . . 126
B.3 Reescritas . . . . . 129
B.4 Geração de Código . . . . . 134
B.5 Exemplo de Compilação . . . . . 137

**Referências Bibliográficas** **141**

# Capítulo 1

# Definição do Trabalho

O projeto de uma linguagem e seu compilador envolve diversas questões: tipos e estruturas que serão suportadas, maneira como as variáveis são tratadas e armazenadas, quais paradigmas a linguagem suporta, além de passar por etapas como verificação da sintaxe do código fonte e otimizações. Cada uma dessas questões torna ainda mais complexo o trabalho de desenvolver o compilador.

Esse problema pode ser ainda mais intensificado com o tamanho e complexidade dos recursos que se pretende acrescentar na linguagem com o decorrer do tempo. Por exemplo, a introdução de novos tipos de dados em uma linguagem podem requerer a modificação de diversos pontos do compilador. Claramente, formas para auxiliar esse trabalho de construção de um compilador são bem-vindas, pois possibilitam maior velocidade e facilidade em sua implementação.

Para facilitar o estudo e a construção de um compilador, Aho et al. [2007] dividem as principais fases de um compilador em análises léxica e sintática, seguidas pela análise semântica e pela geração do código alvo. Para a análise léxica, já existem ferramentas reconhecidas e amplamente utilizadas, como o LEX/FLEX [Lesk & Laboratories, 1987] e o JLex/JFlex [Berk & Ananian, 1997]. Tanto a ferramenta LEX quanto a JLex aceita expressões regulares e produz um analisador léxico dirigido por tabela. Para análise sintática, tem-se, por exemplo, o CUP [Hudson et al., 1999] e o Yacc/Bison [Johnson, 1975], que é um gerador de analisador sintático LALR(1) para gramática livre de contexto. Outras ferramentas para análise léxica e sintática utilizam *Parsing Expression Grammars (PEG)* [Ford, 2004] para definição de sintaxe das linguagens de programação.

Para a implementação das demais fases de um compilador, existem diversas técnicas, como Gramática de Atributos e mecanismos de reescrita. Entretanto, ferramentas existentes para auxiliar esse trabalho, como as apresentadas em Gray et al. [1992] e

Bravenboer et al. [2008], necessitam do aprendizado de linguagens específicas para sua utilização, tornando seu uso mais complexo. O trabalho de Gagnon & Hendren [1998] especifica um *framework* que gera nodos da árvore de sintaxe abstrata do programa fonte e permite seu caminhamento utilizando o padrão de projeto *visitor*. De modo semelhante, Cazzola & Poletti [2010] especifica um *framework* similar, mas utiliza uma mistura do formalismo PEG para gerar a árvore de sintaxe abstrata, e orientação a aspectos para realizar a geração do código. Outras ferramentas, como Rebernak et al. [2006] e Nystrom et al. [2003], não produzem código muito eficiente, desfavorecendo seu uso. Dessa forma, sistemas que geram compiladores são pouco utilizados devido a diversos problemas como eficiência, legibilidade e reusabilidade, portanto, o trabalho difícil e complexo de várias etapas de um compilador deve ser feito manualmente com pouco ou nenhum auxílio de ferramentas.

A tradução final do código para a linguagem da máquina alvo também requer muito trabalho. Devido a várias arquiteturas distintas disponíveis, cada uma com suas particularidades, é necessário que haja uma abstração dessa etapa. Segundo Lattner & Adve [2004], o modelo *Bytecode* realiza essa abstração, pois estabelece uma camada de software entre a máquina alvo e o código intermediário do *Bytecode*, permitindo ao desenvolvedor se preocupar somente com o código intermediário como alvo e possibilitando a reutilização do componente de emissão de instruções em outros projetos. O problema dessa abstração é seu potencial impacto prejudicial frente à eficiência, uma vez que é necessária uma máquina virtual que interpreta as instruções de *bytecode* para a máquina alvo. Entretanto, esse problema é atenuado com a utilização de técnicas de compilação *just-in-time* pela máquina de execução do *bytecode*. Em contraposição, a vantagem dessa abstração é que ela permite maior flexibilidade para um compilador, posto que o código gerado pode ser portado para qualquer arquitetura que possua uma máquina virtual capaz de executar código intermediário do *bytecode* e assim ele é utilizado sem requerer alterações. A abstração também simplifica a tradução final do código, dado que é preciso realizar essa tradução somente para um alvo, em oposição ao habitual, em que se é necessário ter diversos alvos e se requer a reaplicação de componentes similares. Um dos modelos de *bytecode* existentes é o do projeto LLVM [Lattner, 2002] que busca, por meio de otimizações, a geração de um código eficiente.

Componentes de software, segundo Johnson [1997], permitem o reaproveitamento de estruturas implementadas para um ambiente. Portanto, componentes são uma das principais formas de obter reusabilidade em um projeto de software. Entretanto, o projeto dos componentes não deve ser realizado de forma leviana, pois segundo Heineman & Councill [2001], o ideal é que o desenvolvedor que utiliza o componente não precise saber como ele está implementado, e que sua especificação seja fácil o

suficiente de se compreender sem necessitar de definições complementares.

Como forma de atenuar a complexidade de um compilador, Aho et al. [2007] mostra ser preciso dividir as fases realizadas por um compilador, permitindo limitar o foco em fases distintas. Entretanto, somente algumas fases, tais como as análises léxica e sintática, são bem auxiliadas por ferramentas populares, enquanto as demais permanecem sem o auxílio desse tipo de ferramenta, sendo necessário recorrer a sua implementação manual. Ademais, as ações semânticas necessárias para verificação e geração de código de construções podem, por não serem facilmente modularizáveis, gerar replicação de código. Isso reduz a legibilidade, manutenibilidade e possibilidade de reúso do código produzido.

As regras semânticas de cada construção de uma linguagem dependem de diversas estruturas do compilador, e tais estruturas não somente são similares entre compiladores de linguagens distintas como também podem possuir implementação similar a outras construções dentro de um mesmo compilador. No entanto, da maneira em que se é comumente realizada, não é simples reutilizar essas estruturas em outros projetos, nem eliminar a repetição de código necessária para implementá-las [Scott, 2009]. E isso pode comprometer tanto o reúso das regras semânticas implementadas quanto a legibilidade e manutenibilidade do projeto.

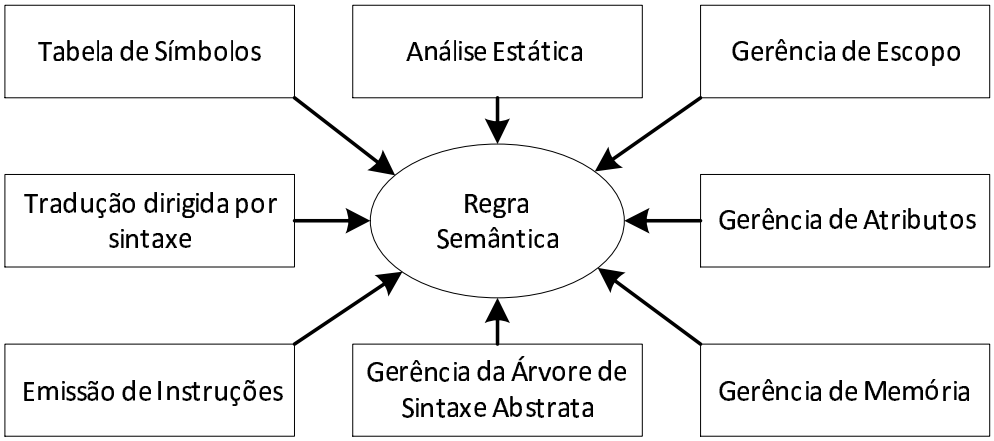


Figura 1.1: Dependências das regras semânticas de cada construção de uma linguagem.

O alto grau de dependência das regras de tradução com as estruturas de um compilador pode demonstrar a complexidade de um projeto de compilador. A Figura 1.1 ilustra essa dependência da regra semântica, ou regra de tradução, com as estruturas do compilador: a *tabela de símbolos*, que mantém registro de identificadores e seus tipos; a *análise estática*, que realiza as verificações de tipos; a *gerência de escopo*, que administra, entre outros detalhes, a visibilidade de identificadores; a *gerência de atributos*, que controla tanto os atributos herdados quanto os sintetizados durante a compilação;

a *gerência de memória*, que controla como as variáveis e outras informações são armazenadas; a *gerência da árvore de sintaxe abstrata (AST)*, que provê métodos para permitir a geração e o caminhamento na *AST*; a *tradução dirigida por sintaxe*, que, com base na sintaxe abstraída na *AST*, provê o mapeamento do código em seu devido comportamento; e, por fim, *emissão de instruções*, que escreve a saída do compilador. Todas essas estruturas são necessárias para se realizar o desenvolvimento de um compilador, demonstrando a complexidade de tal projeto e a necessidade de uma maneira de tornar sua implementação mais legível e reutilizável.

## 1.1 Objetivos

Observando esse cenário, o objetivo principal deste trabalho é a criação de um ambiente de desenvolvimento de compiladores que permita a definição de uma linguagem de programação e a geração de seu compilador com alto grau de reúso de código. O principal enfoque do ambiente proposto por este trabalho é a legibilidade, de maneira a permitir definições mais simples. Tomando como guia a Figura 1.1, nessa proposta, as regras semânticas e as estruturas necessárias são encapsuladas em uma infraestrutura subjacente ao ambiente, permitindo a implementação completa de um compilador.

O escopo de aplicação do ambiente é limitado às linguagens de programação imperativas, de forma a permitir encapsulamento de um conjunto coeso de conceitos fundamentais e recorrentes dessas linguagens que seja expressivo e genérico o suficiente para implementar os compiladores da maior parte das linguagens de programação.

Dentre as estruturas ilustradas na Figura 1.1, as linguagens do ambiente de desenvolvimento encapsulam e fornecem métodos para a *gerência da árvore de sintaxe abstrata*, uma vez que permitem a descrição da linguagem e do caminhamento que será realizado na *AST* gerada durante seu processamento. A *gerência de atributos* também é encapsulada pelas linguagens do ambiente de desenvolvimento, pois permite a declaração, preenchimento e utilização de atributos sintetizados da linguagem. As demais estruturas de um compilador: *tabela de símbolos*; *gerência de escopo*; *gerência de memória*; *emissão de instruções* e *tradução dirigida por sintaxe* são encapsuladas pela infraestrutura de geração de código. A *gerência de escopo*, a *gerência de memória* e *tradução dirigida por sintaxe* são disponibilizadas diretamente por meio dos componentes e as estruturas restantes disponibilizadas indiretamente, uma vez que são utilizadas pelos componentes.

## 1.2 Contribuições

Dentre as contribuições deste trabalho é possível citar a criação de um ambiente de desenvolvimento de compiladores e uma infraestrutura de geração de código, ambos focados na legibilidade do desenvolvimento, sem perder o poder de processamento e generalidade. A legibilidade do ambiente é resultado da separação dos interesses das linguagens de descrição utilizadas pelo ambiente, facilitando que o implementador foque em cada fase de desenvolvimento. Por outro lado, a legibilidade da infraestrutura de geração de código é proveniente do encapsulamento das ações semânticas necessárias para transformar código fonte no código intermediário e esse código em *bytecode* LLVM.

Outra contribuição é que, ao promover a separação dos interesses das fases de compilação, o ambiente de desenvolvimento estimula uma melhor atenção dispensada a cada fase sendo implementada, potencialmente melhorando a qualidade do código criado.

Como contribuição final, tem-se a implementação dos produtos deste trabalho: o ambiente de desenvolvimento de compiladores e a infraestrutura de geração de código, na linguagem de programação C++11, permitindo pronta utilização dos sistemas aqui descritos.

## 1.3 Organização da Dissertação

O Capítulo 2 discute o estado da arte em geração de compiladores, descrevendo diversos sistemas que realizam a geração completa ou incompleta de compiladores. Descreve-se também o bytecode LLVM. O Capítulo 3 apresenta a principal contribuição deste trabalho: um ambiente de desenvolvimento de compiladores. O Capítulo 4 discorre sobre a infraestrutura de geração de código utilizada pelo ambiente de desenvolvimento. No Capítulo 5, consta a validação do trabalho desenvolvido. Finalmente, no Capítulo 6, são apresentadas as considerações finais do trabalho. Os Apêndices, A e B, descrevem, respectivamente, as linguagens utilizadas pelo ambiente de desenvolvimento de compiladores e a implementação de uma linguagem *Small* desenvolvida utilizando o sistema.

# Capítulo 2

# Estado da Arte em Geração de Compiladores

Para auxiliar a criação de compiladores, existem diversos sistemas que realizam sua geração, mas que não são utilizados em larga escala, muitas vezes devido à sua complexidade e quantidade de técnicas e ferramentas necessárias para sua utilização [Tofte, 1990]. Na Seção 2.1, descrevem-se alguns desses sistemas e apontam-se alguns problemas que podem dificultar sua popularização. As técnicas empregadas por esses sistemas também são várias. Alguns, como os apresentados por Brand et al. [2002] e Bravenboer et al. [2008], utilizam transformações como a reescrita, que alteram o código fonte em outro, normalmente de nível intermediário. Além dessa, existe a técnica de gramática de atributos [Knuth, 1968], fazendo, principalmente, computações nos nodos da árvore de sintaxe abstrata. Essa técnica é utilizada pelos sistemas apresentados por Henriques et al. [2005] e Gray et al. [1992]. Há também aquelas ferramentas, como as apresentadas por Ekman & Hedin [2007] e Gagnon & Hendren [1998], que não se focam em uma técnica somente, e se utilizam de uma combinação de paradigmas de programação e outros métodos.

Na Seção 2.2, é apresentado o modelo de código intermediário de *bytecode* do projeto LLVM [Lattner, 2002], que é utilizado pelo trabalho desenvolvido como alvo de geração de código.

## 2.1 Sistemas de Implementação de Compiladores

Sistemas de transformações realizam modificações no código fonte, efetivamente transformando-o em outro. O código alvo pode ser código de máquina ou outra linguagem para a qual já exista compilador na arquitetura desejada.

### 2.1.1 ASF+SDF

Brand et al. [2002] apresentam um ambiente integrado de desenvolvimento (IDE) para implementação de definições de linguagens e ferramentas para geração de compiladores dessas linguagens. O formalismo ASF+SDF permite a definição sintática e semântica de linguagens, e o meta-ambiente apresentado auxilia a criação de ferramentas como compiladores, utilizando esse formalismo. As especificações nesse formalismo são executadas como regras de reescrita condicionais, que as transformam em código fonte da linguagem C. O ambiente utiliza um *parser* generalizado-LR sem *scanner* para realizar a leitura de gramáticas grandes para linguagens de programação. A definição da linguagem pode ser feita de forma modular, facilitando seu desenvolvimento, mas é possível prejudicar a legibilidade e manutenibilidade do sistema pelo fato de não haver restrições referentes aos módulos, podendo eles conterem a linguagem toda, ou conter somente uma definição. Em ambos casos, o entendimento do projeto é dificultado, uma vez que é necessário buscar as diferentes definições realizadas.

Esse sistema foi descontinuado em 2010[1] devido ao desenvolvimento de outra ferramenta que permite realizar o mesmo trabalho que ASF+SDF e que foi julgado pelos desenvolvedores como mais promissor, apesar de utilizar uma modificação de SDF para definir as linguagens. Para simplificar o novo sistema, ele é baseado em Java, não utilizando mais a linguagem C para seu desenvolvimento, e o código transformado é interpretado por uma versão própria da máquina virtual Java.

```
module Expressions
imports BasicNotions
exports
    context-free start-symbols Exp
    sorts Exp
    lexical syntax
        [0-9]+          -> Nat
        [a-z][a-z0-9]* -> Id
    context-free syntax
        Nat          -> Exp
        Id           -> Exp
        Exp "+" Exp -> Exp {left}
        Exp "*" Exp -> Exp {left}
        "(" Exp ")" -> Exp
    context-free priorities
        Exp "*" Exp -> Exp > {left: Exp "+" Exp -> Exp}
```

Listagem 2.1: Exemplo de um módulo de expressões no formalismo SDF.

A Listagem 2.1 apresenta um exemplo de definição de módulo para expressões no formalismo SDF. Em *sorts* são definidos os tipos utilizados no módulo, na parte *lexical syntax* é definido o formato de *Id* e *Nat*. Na parte de *context-free syntax*, é especificada

[1] http://meta-environment.blogspot.com.br/2010/01/future-of-asfsdf-and-meta-environment.html

a forma das expressões por meio de regras que podem conter uma indicação da associatividade, como o *left*, no caso de associatividade daquela regra ser à esquerda. A parte *context-free priorities* serve para contornar a ambiguidade da gramática, indicando a prioridade de escolha de cada regra. É interessante notar que a notação utilizada é inversa a normalmente empregada em gramáticas livres de contexto.

### 2.1.2 Stratego/XT

O Stratego/XT de Bravenboer et al. [2008] oferece um conjunto de ferramentas para auxiliar a reescrita. O Stratego é uma linguagem que provê as regras de rescrita e o XT é formado por ferramentas e linguagens complementares com componentes de reescrita para geração de sistemas dessa natureza, tais como compiladores e interpretadores. Como o principal objetivo do projeto é reusabilidade, ele foi desenvolvido com o foco em componentes que poderiam ser reutilizados posteriormente. Esse sistema foi integrado em um *plugin* da IDE *Eclipse* para facilitar sua utilização e desenvolvimento usando SDF e as ferramentas XT. Dessa forma, esse sistema conseguiu uma maior utilização, mas se restringe a criação de ferramentas para modificar programas ou facilitar a saída de informações, sendo pouco utilizada para compiladores completos.

```
module expressions
imports literals
exports
    sorts Exp
    context-free syntax
        Id            -> Exp {cons("Var")}
        Int           -> Exp {cons("Int")}
        Exp "*" Exp   -> Exp {cons("Mul"),assoc}
        Exp "+" Exp   -> Exp {cons("Add"),assoc}
        Exp "=" Exp   -> Exp {cons("Equ"),non-assoc}
        "(" Exp ")"   -> Exp {bracket}
    context-free priorities
        {left: Exp "*" Exp -> Exp }
      > {left: Exp "+" Exp -> Exp }
      > {non-assoc: Exp "="  Exp -> Exp }
```

Listagem 2.2: Exemplo de um módulo de expressões em Stratego.

Como a linguagem de definição de Stratego é similar ao formalismo utilizado pelo sistema ASF+SDF, seus problemas também são similares. Portanto, há a definição modular da linguagem, entretanto, não há restrições, permitindo a criação de módulos que contenham definições em excesso, ou que contenha a definição de uma regra somente. Em ambos casos, a legibilidade e manutenibilidade do compilador definido são prejudicadas.

O exemplo apresentado na Listagem 2.2, assim como o de ASF+SDF, define o módulo de expressões. Como ambos sistemas utilizam SDF como formalismo de definição de linguagens, seus exemplos são similares. Uma das diferenças se encontra no fato de que cada regra especificada em *context-free syntax* de Stratego/XT deve ser acompanhada da indicação do construtor que será utilizado durante a construção da árvore de sintaxe abstrata. Dessa forma, `cons("Var")` indica que para a produção `Id -> Exp` será utilizado o construtor previamente definido como `Var`. Quando não se quer executar o construtor, usa-se a palavra `bracket` que indica que, para regra associada, nodos na árvore de sintaxe abstrata não serão gerados. Isso pode ser observado na regra de `"(" Exp ")"  -> Exp`.

### 2.1.3 LISA

Knuth [1968] introduziu o conceito de gramáticas de atributos, uma generalização de gramáticas livres de contexto em que a cada símbolo, seja ele terminal ou não, é associado um conjunto de atributos para conter informações semânticas e cada produção da gramática pode ter um conjunto de regras semânticas associadas para cálculo desses atributos.

Henriques et al. [2005] apresentam um sistema de geração de compiladores a partir da definição de uma especificação de gramática de atributo. Tal como ASF+SDF, o sistema apresentado possui um ambiente de desenvolvimento que permite o trabalho ser realizado de forma textual ou visual, gerando árvores de visualização dos comandos especificados. O trabalho de Rebernak et al. [2006] estende o trabalho inicial, adicionando a programação orientada por aspectos de Kiczales et al. [1997] ao sistema LISA, buscando maior reusabilidade das especificações de linguagens implementadas. O sistema também utiliza do conceito de herança, desenvolvido em linguagem orientada por objetos, para permitir a extensão de linguagens previamente definidas.

```
language SimpleCalc {
 attributes int *.val;
 rule Start {
  S ::= E compute { S.val = E.val; };
 }
 rule Expression {
  E ::= E + E compute { E[0].val = E[1].val + E[2].val; };
  E ::= #Number compute { E[0].val = Integer.valueOf(#Number.value()); };
 }
}
```

Listagem 2.3: Exemplo de definição de uma calculadora em LISA.

O exemplo apresentado na Listagem 2.3 define uma calculadora simples em LISA,

mas sem utilizar a extensão de programação orientada por aspectos. A linguagem definida é um módulo nomeado `SimpleCalc` e a gramática utilizada terá os atributos definidos em `attributes`. No caso do exemplo, é definido que existirá um atributo do tipo `int` denominado val para todas variáveis da gramática. As regras de produção são definidas dentro dos campos identificados como `rules` e o que cada produção calcula é definido entre chaves após a palavra chave `compute`. No exemplo, é definido que a regra de `Start`, `S ::= E` calculará o atributo `val` de `S` como sendo o atributo calculado de `E`.

Para evitar a repetição de definições, utiliza-se uma gramática de atributos orientada a aspectos, criando pontos estáticos nas especificações para utilização de regras semânticas. Esses pontos estáticos podem ser empregados em produções distintas da hierarquia definida. A modularidade desse sistema é derivada da orientação a aspectos, mas não há definição clara sobre quando deve-se usar aspectos para definir uma linguagem e quando deve-se utilizar aspectos como mecanismo de acréscimo da linguagem. Outro problema desse sistema, referente à legibilidade, é a utilização de índices para empregar as diferentes variáveis com mesmo nome em uma mesma produção, como visto na listagem na produção `E + E`. Isso torna o código propenso a erro, uma vez que necessita que o implementador conte o índice das variáveis que são utilizadas, além de, em se alterando uma produção, é necessário alterar todo o código, avaliando os índices das variáveis. Entretanto o a utilização de aspectos permite evitar essa situação.

### 2.1.4 Eli

O sistema Eli, apresentado por Gray et al. [1992], é também baseado em gramáticas de atributo e oferece um conjunto de ferramentas que buscam resolver problemas e subproblemas da compilação como análise sintática, semântica e geração de código. O conjunto de ferramentas do sistema é controlado por um sistema especialista, que esconde a junção dos resultados das ferramentas, algo considerado complexo e difícil. Entretanto essa subdivisão das etapas de compilação em muitas ferramentas gera o problema de se ter que aprender e dominar uma quantidade de linguagens distintas superior à utilização de outros sistemas.

A Listagem 2.4 contém um exemplo de gramática de expressões no sistema Eli, sendo resolvido somente o problema da análise sintática nesse exemplo. Nesse sistema, o símbolo que separa o lado esquerdo de uma produção da gramática do lado direito é o dois pontos (`:`) e o ponto simples denota o final da descrição da produção. Também é utilizado os colchetes (`[]`) para demonstrar o que é opcional da gramática. Dessa forma, essa gramática é simples, somente descreve a gramática de expressão e determina

a precedência dos operadores por meio da estrutura da gramática. Para realizar o resto da compilação é necessário a utilização de outras ferramentas do sistema, cada uma com sua linguagem de especificação diferente, o que dificulta não somente a utilização geral do sistema Eli, mas também dificulta a manutenibilidade e legibilidade de um projeto de um compilador.

```
Expression: SimpleExpression [RelationalOperator SimpleExpression] .
RelationalOperator: '<' / '=' / '>' / '<=' / '<>' / '>=' .
SimpleExpression:
    [SignOperator] Term / SimpleExpression AddingOperator Term .
SignOperator: '+' / '-' .
AddingOperator: '+' / '-' / 'or' .
Term: Factor / Term MultiplyingOperator Factor .
MultiplyingOperator: '*' / 'div' / 'mod' / 'and' .
Factor:
    Numeral /
    VariableAccess /
    '(' Expression ')' /
    NotOperator Factor .
NotOperator: 'not' .
VariableAccess:
    VariableNameUse /
    VariableAccess '[' Expression ']' /
    VariableAccess '.' FieldNameUse .
```

Listagem 2.4: Exemplo de gramática de expressões no sistema Eli.

### 2.1.5 JastAdd

Ekman & Hedin [2007] apresentam o sistema JastAdd, criado para auxiliar o desenvolvimento de compiladores e ferramentas relacionadas. Esse sistema cria uma extensão de Java e utiliza o formalismo de *"Rewritable Circular Reference Attributed Grammar (ReCRAGs)"*, que usa técnicas de orientação por objetos para reescrever a árvore de sintaxe abstrata (AST).

O desenvolvimento de compiladores é realizado via definição de módulos que resolvem partes dos problemas de compilação e tais módulos podem ser reutilizados em outras ferramentas. Os autores sugerem que o uso de Java e seus "reconhecidos atributos de orientação por objetos" tornam o sistema fácil de aprender e utilizar e sugerem que a documentação simplifica o entendimento de gramáticas de atributos, aspectos e sistemas de reescrita em relação à programação orientada por objetos, não sendo necessário conhecimento prévio desses assuntos.

Para validar o sistema, os autores usaram-no para implementar um compilador de Java 1.4, e Ekman & Hedin [2008] analisam seu desempenho e o comparam com outros compiladores Java disponíveis. Para essa implementação, os autores modularizaram o *front-end* e o *back-end*, permitindo que o *front-end* fosse utilizado como uma

ferramenta separada para análise de código e que múltiplos *back-ends*, com ambientes de execução distintos, fossem testados. Dentre os ambientes testados estão a máquina virtual de Java e código fonte em C. Como experimento, partes de Java 5 foram implementadas, incluindo "atributos complexos da linguagem como classes genéricas que estendem significativamente o sistema de tipos". ASTs são representadas por classes e os atributos por métodos, dessa forma a avaliação dos atributos é feita como resultado de acessar o método. Apesar de o sistema JastAdd não possuir suporte para *"parsing"*, é possível utilizar qualquer gerador de *parsers* baseado em Java como JavaCC, ANTLR e CUP. O algoritmo de avaliação do JastAdd é um avaliador dinâmico e *lazy* com *"caching"* de atributos para facilitar a avaliação.

```
abstract Expr;
Literal : Expr ::= <Value:String>;
AddExpr : Expr ::= Left:Expr Right:Expr;
SubExpr : Expr ::= Left:Expr Right:Expr;

syn int Expr.value();
eq Literal.value() = Integer.parseInt(getValue());
eq AddExpr.value() = getLeft().value() + getRight().value();
eq SubExpr.value() = getLeft().value() - getRight.value();
```

Listagem 2.5: Exemplo da avaliação de expressões no sistema JastAdd.

Na Listagem 2.5, é possível ver a separação entre a definição da árvore de sintaxe abstrata da linguagem, realizada na primeira parte em que se define o formato de literais, expressões de adições e subtração, e a definição da semântica da linguagem na segunda parte, em que é definido como deve ser avaliada cada parte definida inicialmente. A palavra chave `abstract` define a criação da classe de expressões, e a palavra `syn` define um atributo sintetizado da classe de expressão que será utilizado na compilação. Além disso, a palavra `eq` determina que será definida a equação para o cálculo de um atributo, portanto `Literal.value() = Integer.parseInt(getValue());` especifica que o atributo `value()` de `Literal` será calculado como a representação em inteiro da cadeia de caracteres sendo lida no momento da execução dessa equação.

### 2.1.6 Polyglot

O Polyglot, sistema apresentado por Nystrom et al. [2003], é um *framework* para criação de extensões da linguagem de programação Java que busca facilitar a construção de linguagens de domínio específico (DSL). Para não ter o esforço de duplicação do compilador de Java, o *framework* foi implementado em Java e é um verificador semântico de Java com a possibilidade de se alterar outros passos do processo de compilação como a AST. Essa dependência do *framework* com a linguagem Java pode ser benéfica devido

à possibilidade de se utilizar todo poder de expressão de Java, mas pode prejudicar o *framework* por ser necessário verificar sua correção a cada atualização das ferramentas de Java.

Os passos realizados pelo *framework* envolvem compilar o código Java com as extensões dos autores, tendo como alvo a própria linguagem Java e, subsequentemente, invocar qualquer compilador de Java, como o javac, para terminar a compilação para *bytecode* que será executado pela máquina virtual do Java. Ekman & Hedin [2008] também comparou o desempenho do sistema Polyglot, descobrindo que diversos testes utilizados para a quinta versão de Java não funcionam com esse sistema, indicando conflitos na implementação do *framework*, impedindo a utilização dos recursos mais recentes da linguagem. Também verificou-se que o desempenho do tempo de compilação chegou a ser mais de cinco vezes pior em alguns testes, oferecendo um tempo de compilação cinco vezes superior às demais implementações e demonstrando uma implementação ineficiente.

### 2.1.7 Neverlang

O *framework* apresentado por Cazzola & Poletti [2010], denominado *Neverlang*, busca dar suporte a geração completa de compilador e interpretador, bem como facilitar modificações futuras na linguagem. Isso é feito via da composição de módulos básicos em uma linguagem de programação, como entrada de dados, verificação de tipos e geração de código. Dessa forma, o *framework* consiste em uma linguagem para definir os módulos básicos e suas composições e um mecanismo para realizar a geração do compilador.

Em Neverlang, a definição de linguagem é uma composição de componentes modulares, e cada um desses componentes possui tanto sua sintaxe definida como qual é o resultado de sua avaliação. Com essa definição, é gerado um compilador ou interpretador com o *front-end* gerando a árvore de sintaxe abstrata do código fonte a partir das definições sintáticas dos módulos, e o *back-end* aproveitando dessa árvore para gerar o código alvo. Para caminhar na árvore, é utilizado o paradigma de orientação a aspectos para superar as limitações percebidas pelo padrão de projeto *visitor*. Dentre as limitações desse padrão que são superadas, está o fato de que não é necessário modificar as classes dos nodos da árvore para implementar as avaliações, sendo realizada a junção desses detalhes por meio de aspectos.

O exemplo inserido na Listagem 2.6 contém a definição dos módulos de escolha em Neverlang e de uma linguagem que utilizaria esses módulos. Cada módulo possui uma função descrita em `role` que define onde o módulo será empregado. No exemplo,

```
module if_syntax {
 role(syntax) {
  Statement ^
  'if' '(' ExprB ')' '{' StatementL '}' 'else' '{' StatementL '}'
  Statement ^ 'if' '(' ExprB ')' '{' StatementL '}'
 }
}
module if_eval {
 role(evaluation) {
  0 {
   if (new Boolean($1.eval)) $2.eval;
   else $3.eval;
  }
  4 {
   if (new Boolean($5.eval)) $6.eval;
  }
 }
}
slice if {
 module if_syntax with role syntax
 module if_eval with role evaluation
}
language If_DSL {
 slices core less_then more_then equals string int print if
 roles syntax < eval
}
```

Listagem 2.6: Exemplo de definição de escolha em Neverlang.

o módulo `if_syntax` possui a função de análise sintática e, portanto, define o formato que um comando de escolha terá. O simbolo `^` é utilizado como separador entre o lado esquerdo da produção e o lado direito. O módulo `if_eval` possui a função de avaliação do comando, realizando sua semântica. Nessa função, os números se referem às variáveis definidas no módulo sintático em sua ordem, portanto o `0` se refere ao primeiro `Statement` e o 4 ao segundo. A palavra-chave `slice` serve para declarar os módulos e suas funções, definindo parte da linguagem. Por fim, a construção declarada por `language` define a conjunção da linguagem final, declarando quais pedaços de linguagem serão utilizados e quais funções serão executadas. A ordem de execução das funções é definida pelo símbolo `<`, sendo executado o caminhamento na árvore de sintaxe abstrata para cada função necessária, visitando os nodos e executando as instruções definidas.

Apesar do foco em modularidade, o modo utilizado por esse *framework* pode acabar tornando a descrição confusa, pois permite que a definição do módulo seja feito a qualquer momento, possibilitando que o usuário intercale o processo para realização da compilação, prejudicando sua legibilidade e entendimento. Além disso, a referência numeral das variáveis em regras semântica não somente prejudica a legibilidade como, caso seja realizada alguma modificação da sintaxe, torna o desenvolvimento propenso a falhas.

### 2.1.8 SableCC

Gagnon & Hendren [1998] introduzem um *framework* para gerar compiladores e interpretadores com linguagem alvo Java. Esse trabalho utiliza técnicas orientadas por objetos para construir automaticamente uma árvore de sintaxe abstrata e se utiliza do padrão de projeto *visitor* para gerar classes para realizar o caminhamento pela árvore construída. Entretanto, não há como definir, na linguagem de descrição utilizada, as ações que serão realizadas durante o caminhamento da árvore, sendo necessário que essas classes sejam implementadas manualmente. Dessa forma, a utilização desse sistema é complicada já que o implementador deve implementar classes manualmente em conjunto com classes geradas pelo sistema.

```
Tokens
 number = ['0'..'9']+;
 plus = '+'; minus = '-';
 l_par = '(';
 r_par = ')';
 blank = ' ';
Ignored Tokens blank;
Productions
 expr =
  {plus} expr plus number |
  {minus} expr minus number |
  {par} l_par expr r_par |
  {number} number |
  ( {op} [left]:expr op [right]:expr );
 op =
  ( {plus} plus ) |
  ( {minus} minus );
```

Listagem 2.7: Exemplo da gramática de expressões em SableCC.

Na Listagem 2.7, está um exemplo para gramática de expressões em SableCC. Na seção de `Tokens`, são definidos os terminais da linguagem definida. Isso é realizado por meio de expressões regulares atribuídas a um identificador que será utilizado na gramática. Subsequentemente, na seção `Ignored Tokens`, são definidos quais terminais devem ser ignorados pelo sistema e, portanto, nada geram. Por fim, é definida a seção `Productions` que contém a gramática da linguagem, definindo seu formato e sua árvore de sintaxe abstrata.

### 2.1.9 Pappy

Uma implementação de analisador sintático descendente recursivo com *backtracking* é *Parsing Expression Grammar* (PEG), apresentado por Ford [2004] que busca se diferenciar dos métodos usados comumente de gramáticas livres de contexto e expressões regulares para descrição de linguagens de programação. Uma característica desse for-

malismo é a unificação da descrição léxica e sintática em uma gramática única. Outra diferença é a definição do que o autor chama de *escolhas priorizadas*, em contraponto ao não-determinismo inerente à escolha de alternativas em gramáticas livres de contexto.

A ferramenta *Pappy* Ford [2002a] foi a primeira desenvolvida para implementar *Parsing Expression Grammar* [Ford, 2002b] e, mesmo utilizando uma técnica de *backtracking*, possui tempo linear de execução devido à utilização da técnica de memorização, armazenando os resultados e não os calculando novamente. Dessa forma, a técnica empregada na ferramenta permite a realização do *backtrack*, mas mantém o tempo de execução às custas do espaço utilizado.

Um dos problemas da *Pappy* e de *Parsing Expression Grammar* é a limitação para uso de recursão à esquerda. Como *Pappy* é baseado na simplicidade da técnica de descida recursiva com backtrack, ao tentar realizar a derivação à esquerda de uma gramática com recursão à esquerda o analisador sintático entrará em *loop* infinito. Apesar de ser possível reescrever gramáticas substituindo recursão à esquerda por recursão à direita, a legibilidade dessa rescrita pode ser ruim. Um exemplo desse problema está na Tabela 2.1. Na primeira coluna está a definição da gramática de expressões com recursão a esquerda, na segunda coluna com recursão à direita. Warth et al. [2008] tentam aprimorar o algoritmo de memorização de da técnica empregada por *Pappy* para melhor suportar recursão à esquerda, mas o modo realizado pode gerar tempos super-lineares de execução do analisador sintático, fazendo com que o tempo gasto cresça até exponencialmente em função do tamanho do texto de entrada, e tornando a ferramenta pouco atraente em comparação com as ferramentas já existentes.

Tabela 2.1: Comparação de gramática de expressões.

| **Recursão à Esquerda** | **Recursão à Direita** |
|---|---|
| E –> E "+" T<br>  \| T<br>T –> T "*" F<br>  \| F<br>F –> val \| "(" E ")" \| "–" E | E –> T E'<br>E'–> "+" T E' \| $\epsilon$<br>T –> F T'<br>T'–> "*" F T' \| $\epsilon$<br>F –> val \| "(" E ")" \| "–" E |

Outro problema de *Pappy* é que, devido à memorização, não é possível realizar ações semânticas no meio de produções, algo utilizado em analisadores sintáticos de C e C++, por exemplo, para construir a tabela de tipos e distinguir os tipos de identificador. Para realizar esse tipo de ação, seria necessário apagar a memorização realizada até o momento e refazê-la, tornando o analisador sintático extremamente ineficiente.

Outro exemplo que pode requerer esse tipo de ação é o armazenamento de informações contextuais do próprio texto sendo lido como o número da linha, utilizado para indicação de erros sintáticos.

O custo de espaço requerido por *Pappy*, devido a computadores modernos, pode ser ignorado em alguns casos. Arquivos fontes de programas em diversas linguagens de programação não chegam a possuir tamanho suficiente para ser um problema, principalmente devido às facilidades de modularização e separação de arquivos fontes distintos, mas arquivos de dados armazenados, por exemplo, em *XML* podem ser um problema, uma vez que a técnica possui custo de $n$, sendo $n$ o tamanho da entrada, mas multiplicado por uma constante que segundo Ford [2002b] pode ser alta demais. Mizushima et al. [2010] tentam amenizar esse problema inserindo um operador de *corte* na gramática para controlar o *backtrack*, apagando a memorização quando não mais necessária. Mas mesmo com essa alteração, somente um subconjunto de *XML* consegue ser lido sem que o custo de espaço se torne um problema.

Becket & Somogyi [2008] argumentam que os analisadores sintáticos usando a técnica empregada em *Pappy* são de implementação trivial mas que podem ser significativamente menos eficientes que um analisador sintático descendente recursivo com *backtrack* convencional. Indicam também que a implementação de Ford [2002b] da gramática de Java precisa de 400 *bytes* de memória para cada *byte* de entrada, demonstrando o quão ineficiente em relação à memória a técnica é.

### 2.1.10 ANTLR

Introduzido por Parr & Quong [1995] e atualmente em sua quarta versão Parr [2013], *ANTLR* (*ANother Tool for Language Recognition*) é uma ferramenta de linguagem com um *framework* para construir analisador sintático LL(*) como descrito em Parr & Fisher [2011]. A entrada da ferramenta é uma gramática livre de contexto acrescentada de predicados sintáticos e semânticos, bem como ações necessárias para tratar a linguagem sendo reconhecida. Os predicados são utilizados para definir o *lookahead* do LL(*) e, assim, auxiliar no reconhecimento da linguagem sem utilizar memória excessivamente como PEG. Para mais eficiência, a ferramenta busca gerar autômatos determinísticos finitos para cada não terminal da gramática e, somente no caso de falhar na geração de algum estado, utiliza *backtrack*. Apesar da imprevisibilidade, os autores afirmam que o analisador sintático só necessita do *backtrack* ocasionalmente sendo, em testes restritos, bastante eficiente.

Um dos problemas de ANTRL é similar ao encontrado em PEG com na ferramenta *Pappy*, sua gramática não pode possuir recursão à esquerda, potencialmente

tornando-a menos legível. Outro problema de legibilidade provém dos predicados utilizados e da forma em que as ações são inseridas. Na Listagem 2.8, se encontra a definição de uma pequena gramática de expressão com suas devidas ações e é possível perceber que, utilizando todos recursos necessários, a ferramenta tem sua legibilidade prejudicada.

O exemplo apresentado na Listagem 2.8 contém a avaliação de expressões no sistema ANTLR. Com a declaração inicial marcada pela palavra `grammar` é definido que será descrita uma gramática cujo identificador é `Expr`. Nos campos de `header` e `members` são especificados detalhes da linguagem Java que serão necessários para a gramática, sendo o primeiro campo utilizado para importações de bibliotecas e o segundo campo para a declaração de variáveis a serem utilizadas. Esses campos são seguidos pela definição da gramática sendo o lado esquerdo separado do lado direito pelo símbolo de dois pontos (`:`). Cada produção da gramática pode ser seguida de instruções entre chaves, a instrução especificada para a produção `stat: expr NEWLINE` na Linha 13 define que, ao reconhecer essa produção da gramática, será impresso no *prompt* de comando o valor calculado pela expressão. O valor calculado pela expressão só pôde ser utilizado devido a especificação de retorno na regra `expr` na Linha 19, a sintaxe que permitiu isso foi o `returns [int value]` que declara aquela produção retornará um valor do tipo inteiro cujo identificador será `value`. Para simplificar o uso das variáveis dentro dos colchetes é possível renomeá-las, como demonstrado em `e=multExpr` na Linha 20, redefinindo o nome utilizado no código. No fim está a definição dos terminais, demonstrada na Linha 41 pelo sequenciamento de caracteres determinando que um `ID` é qualquer conjunto de caracteres de 'a' até 'z' maísculos ou minúsculos. Também são definidas mudança de linha e espaço vazio, que devem ser ignoradas utilizando uma função Java denominada `skip`.

```
1  grammar Expr;
2
3  @header {
4  import java.util.HashMap;
5  }
6
7  @members {
8  HashMap memory = new HashMap();
9  }
10
11 prog:   stat+ ;
12
13 stat:   expr NEWLINE {System.out.println($expr.value);}
14     |   ID '=' expr NEWLINE
15         {memory.put($ID.text, new Integer($expr.value));}
16     |   NEWLINE
17     ;
18
```

```
19 expr returns [int value]
20     :   e=multExpr {$value = $e.value;}
21         (   '+' e=multExpr {$value += $e.value;}
22         |   '-' e=multExpr {$value -= $e.value;}
23         )*
24     ;
25
26 multExpr returns [int value]
27     :   e=atom {$value = $e.value;} ('*' e=atom {$value *= $e.value;})*
28     ;
29
30 atom returns [int value]
31     :   INT {$value = Integer.parseInt($INT.text);}
32     |   ID
33         {
34         Integer v = (Integer)memory.get($ID.text);
35         if ( v!=null ) $value = v.intValue();
36         else System.err.println("undefined variable "+$ID.text);
37         }
38     |   '(' expr ')' {$value = $expr.value;}
39     ;
40
41 ID  :   ('a'..'z'|'A'..'Z')+ ;
42 INT :   '0'..'9'+ ;
43 NEWLINE:'\r'? '\n' ;
44 WS  :   (' '|'\t')+ {skip();} ;
```

Listagem 2.8: Exemplo da avaliação de expressões no sistema ANTLR.

### 2.1.11 Lex/YACC

O sistema LEX [Lesk & Laboratories, 1987] é utilizado para geração de analisadores léxicos na linguagem C. Flex [Project, 2008] foi desenvolvido uma versão que gera código de C/C++, buscando melhorar a eficiência e ampliar os recursos disponíveis para o desenvolvimento. Ambos sistemas geram tabelas e, usualmente, fazem o reconhecimento de padrões léxicos definidos podendo retornar *tokens* para o desenvolvedor do compilador utilizar na leitura do código fonte.

O sistema *Yacc* [Johnson, 1975], gera analisadores sintáticos em código C. Assim como ocorreu com o LEX, Bison [Foundation, 2009] é uma versão do *Yacc* que gera o código C/C++. Para a geração do analisador é utilizada uma gramática livre de contexto, que é convertida em um analisador sintático LR ou GLR utilizando tabelas de analisador sintático LALR(1). Os sistemas LEX/FLEX e Yacc/Bison podem ser utilizados conjuntamente, sendo usados os *tokens* processados pelo primeiro como símbolos da gramática utilizada no segundo.

Em ambos casos, é possível construir um compilador completo utilizando somente esses sistemas, mas após a análise sintática não é oferecido nenhum recurso para facilitar

o trabalho ou para gerar o compilador. Sendo necessário, portanto, o desenvolvimento manual e trabalhoso do resto do compilador.

```
%{
#include <stdio.h>
#include "y.tab.h"
extern int yylval;
%}
digit       [0-9]
number      {digit}+
%%
" "         ;
{number}    {
              yylval = atoi(yytext);
              return(DIGITS);
            }
```

Listagem 2.9: Exemplo do reconhecimento de dígitos e letras em LEX.

Na Listagem 2.9, se encontra um exemplo utilizando o sistema Lex para reconhecer dígitos que são utilizados pelo sistema Yacc na Listagem 2.10. Os exemplos são utilizados em conjunto para realizar o *parsing* de uma calculadora. O sistema Lex é divido em três seções, iniciando com as declarações necessárias da linguagem C, seguida por uma seção em que as expressões regulares são identificadas para permitir sua reutilização. Por fim, a última seção contém expressões regulares, identificadas ou não, seguidas de ações para realizar o parsing do texto de entrada.

O sistema Yacc, cujo exemplo está na Listagem 2.10, também é divido em seções similares. A primeira seção é utilizada para realizar as declarações necessárias da linguagem C. Na segunda seção, são declarados os detalhes da gramática que está sendo definida. Entre os detalhes definidos nessa seção, estão a definição dos *tokens* que são utilizados pelo sistema Lex, o tipo de valor armazenado em cada nodo produzido pela gramática e a precedência de operadores na gramática. A seção seguinte contém a gramática, onde cada produção possui uma regra semântica. As variáveis de cada produção são referidas por meio do caractere `$` agregado ao índice da variável. Enfim, a última seção, contém as declarações das funções necessárias para o funcionamento do sistema. Entre as funções declaradas, estão a função principal do programa, a função de tratamento de erro e a função de finalização do *parsing*.

Pelos exemplos, é possível perceber que, assim como o sistema ANTLR, a combinação dos sistemas LEX/YACC podem ter sua legibilidade prejudicada, sendo a legibilidade do sistema Yacc piorada devido à utilização de índices para se referir às variáveis de uma produção. Há também a necessidade de se implementar funções básicas do sistema, como o tratamento de erro e a finalização do *parsing*, para que o mesmo funcione. Esses problemas tornam seu uso mais complicado.

```
%{
#include <stdio.h>
%}
%union {
     int val;
}
%token DIGITS
%type <val> exp

%left '+'
%left '*'
%left UMINUS  /*supplies precedence for unary minus */

%%                    /* beginning of rules section */

list: list stat '\n'
   | list error '\n'
   {
     yyerrok;
   } ;

stat: expr
   {
     printf("%d\n",$1);
   };

expr: '(' expr ')'
   {
     $$ = $2;
   }
   | expr '*' expr
   {
     $$ = $1 * $3;
   }
   | expr '+' expr
   {
     $$ = $1 + $3;
   }
   | '-' expr %prec UMINUS
   {
     $$ = -$2;
   }
   | number
   {
     $$ = $1
   }
   ;
%%
main()
{
   return(yyparse());
}

void yyerror(cont char* s)
{
   fprintf(stderr, "%s\n",s);
}

int yywrap()
{
   return 1;
}
```

Listagem 2.10: Exemplo de uma grámatica de expressões em Yacc.

## 2.2 Código Intermediário

O uso de uma representação intermediária como alvo inicial de tradução de um compilador tem a vantagem de facilitar a construção desse compilador, possibilitando que cada representação intermediária esteja focada em uma fase distinta do processo de compilação. Outra vantagem é que uma representação intermediária permite a emissão de código de distintas máquinas alvo sem necessitar de alterações no *Front-End* utilizado até a representação intermediária, facilitando o processo de produção do compilador.

O projeto *LLVM*, idealizado por Lattner [2002], é uma coleção de ferramentas para compilação modulares e reutilizáveis. Entre as ferramentas implementadas, estão otimizadores independentes de máquina, bibliotecas para geração de código intermediário LLVM e um compilador *Just-In-Time* para transformar a representação intermediária em código de máquina e, assim, conseguir um desempenho superior à execução de uma máquina virtual.

A representação intermediária de LLVM é uma representação de baixo nível que busca manter expressividade, extensibilidade e tipagem ao mesmo tempo. Outra questão abordada por essa representação é o uso da técnica SSA ("*Single Static Assignment*"), introduzida por Cytron et al. [1991], em que cada atribuição a variáveis é realizada somente uma vez, sendo necessário nomes distintos para a mesma variável sempre que lhe for atribuído um novo valor. Dessa maneira, a realização de diversas otimizações como propagação de constantes, eliminação de código morto, redução de custo ("*strength*") e eliminação parcial de redundância são mais facilmente realizadas nessa representação [Cooper & Torczon, 2007].

Um programa completo de LLVM comporta as definições de módulos que conterão funções e variáveis de escopo local e global. Cada função contém uma quantidade de blocos básicos que, internamente ao compilador, serão organizados em um Grafo de Fluxo de Controle (CFG) para facilitar a implementação de otimizações. Por fim, cada bloco básico contém um conjunto de instruções que deverão ser executadas em ordem e sem desvio.

Os identificadores utilizados podem ser de dois tipos básicos: global e local. Os identificadores globais servem para nomes de funções e variáveis globais, e sua declaração é precedida pelo caractere `'@'`. Os identificadores locais servem para nomear tipos e registradores sendo precedidos pelo caractere `'%'`.

Exceto instruções específicas como as de retorno, desvio e chamadas de procedimentos, cada instrução produz um resultado que é armazenado no identificador especificado, o qual pode ser local ou global. Cada instrução é identificada por seu código, que é seguida pelo tipo do valor operado por ela e, dependendo de qual seja a

instrução, os identificadores e constantes utilizadas em sua execução.

Nas Listagens 2.11, 2.12 e 2.13 estão exemplos de formas distintas para multiplicar o valor de uma variável por oito. No primeiro exemplo, é usada a instrução de multiplicação para realizar essa tarefa de uma forma simples, sendo otimizado por redução de custo no segundo exemplo, substituindo-se a multiplicação por um *shift left* em três casas binárias. O último exemplo realiza a tarefa de uma forma mais difícil, realizando adições sequencialmente para obter o resultado.

```
%result = mul i32 %x, 8
```

Listagem 2.11: Modo simples para multiplicar uma variável por oito.

```
%result = shl i32, %x, 3
```

Listagem 2.12: Modo simples otimizado para multiplicar uma variável por oito.

```
%0 = add i32 %x, %x
%1 = add i32 %0, %0
%result = add i32 %1, %1
```

Listagem 2.13: Modo complexo para multiplicar uma variável por oito.

A seguir, serão especificadas algumas instruções de LLVM, importantes para a realização deste trabalho.

### Sistema de Tipos

O sistema de tipos de LLVM possui números inteiros, números de ponto flutuante e rótulos como tipos primitivos, sendo caracteres e valores boleanos armazenados como inteiros de larguras distintas. Como tipos compostos, LLVM possui arranjos, estruturas e ponteiros. Dessa forma, linguagens que possuem um sistema de tipos mais simples como C são traduzidos diretamente para esse sistema de tipos. Outras linguagens que possuem tipos primitivos ou compostos distintos devem utilizar-se dos tipos compostos de LLVM para realizar sua geração. Por exemplo, uma tupla que em Python é um tipo composto precisa ser mapeada em uma estrutura de LLVM.

A sintaxe utilizada para definição de uma estrutura em LLVM é a seguinte:

```
%T1 = type { <type list> };
```

O campo `T1` é utilizado para identificar a estrutura definida, e o campo `<type list>` deve ser uma lista de tipos contidos pela estrutura. Também é possível

declarar tipos compostos por si próprios, por exemplo um nodo de uma lista encadeada pode ser definido como na Listagem 2.14. Nela é definida uma estrutura que terá um inteiro de 32 bits como dado para ser armazenado e um apontador para outro nodo com a mesma estrutura.

```
%node = type {i32, %node*}
```

Listagem 2.14: Definição de uma estrutura de lista encadeada.

A sintaxe utilizada para definição de um arranjo é:

```
[<# elements> x <elementtype>]
```

O campo `<# elements>` deve ser um número natural, utilizado para denotar o número de elementos do arranjo e o campo `<elementtype>` deve ser o tipo dos valores que o arranjo armazenará, podendo ser outro arranjo no caso de um arranjo multidimensional. O uso dessa definição de um arranjo pode ser realizada na definição de uma estrutura ou utilizada na alocação de um espaço com a instrução `alloca`.

### Constantes

As constantes simples da linguagem utilizada pelo *framework* LLVM são: booleanas, compreendendo os literais `true` e `false` que são mapeados para constantes do tipo inteiro de um *bit*; números inteiros, que são números naturais e negativos utilizados como constantes inteiras; números de ponto flutuantes, que podem ser representados por uma notação decimal comum em que os valores decimais são separados por um ponto ou usar a notação exponencial; ponteiro nulo, denotado pelo identificador `null`.

### Operações Binárias

As instruções de operação binária oferecidas pelo *framework* LLVM possuem, no geral, o seguinte formato:

```
<result> = <oper> <ty> <op1>, <op2>
```

O campo `<result>` especifica em qual variável ou temporário será armazenado o resultado da operação binária. O campo `<oper>` indica qual operação deve ser realizada, podendo ser operações de adição, subtração, multiplicação, divisão, resto de divisão ou operações realizadas com *bits*, como o deslocamento de *bits*, operações de `E`, `OU` e `XOR`. As operações binárias possuem variações para os tipos inteiros e para ponto flutuante, uma vez que suas otimizações são distintas. O campo `<ty>` indica o tipo de

variável sobre a qual será realizada a operação, definindo, por exemplo, se será realizada com inteiros de 32 *bits* ou de 64 *bits*. Por fim, os campos `<op1>` e `<op2>` indicam as variáveis ou constantes que serão utilizadas como operandos da operação.

### Armazenamento e Endereçamento de Memória

O armazenamento e endereçamento de memória em LLVM é realizado de forma indireta. A instrução `alloca` reserva um espaço na memória de *heap* e devolve um apontador para a área alocada. As instruções `load` e `store` são utilizadas para, respectivamente, ler e escrever na memória e utilizam como endereço o ponteiro gerado pela instrução `alloca`. Na Listagem 2.15, está um exemplo de armazenamento e endereçamento de memória. Na linha 1, é alocada uma área para um inteiro de 32 *bits* e retornado seu endereço. Na linha 2, é armazenado nesse endereço o valor 3. E na linha 3, o valor armazenado é carregado na variável `val`.

```
%ptr = alloca i32
store i32 3, i32* %ptr
%val = load i32* %ptr
```

Listagem 2.15: Exemplo de armazenamento.

Outra instrução importante para armazenamento é `getelementptr`, utilizado para obter o endereço de um subelemento de uma estrutura de dados agregados, ou seja, acessar elementos de um arranjo ou de uma estrutura. Sua sintaxe é:

```
<result> = getelementptr <pty>* <ptrval>{, <ty> <idx>}*
```

Como nas instruções de operação binária, o campo `<result>` especifica em qual variável ou temporário será armazenado o resultado da busca do endereço do subelemento. O campo `<pty>*` indica qual o tipo da estrutura de dados em que o subelemento será buscado. O campo `<ptrval>` deverá conter o endereço de memória do ponteiro da estrutura de dados em que o subelemento será buscado. Os campos `<ty>` e `<idx>` se referem, respectivamente, ao tipo do índice utilizado e o valor do índice utilizado. Os dois últimos campos podem ser repetidos quantas vezes forem necessárias, facilitando a busca de valores em arranjos multidimensionais.

### Desvio

A instrução `br` é utilizada para realizar controle de fluxo, transferindo para um bloco básico distinto. Existem dois formatos para essa instrução, um desvio condicional e um incondicional. A sintaxe da instrução incondicional é:

```
br label <dest>
```

O único campo dessa instrução é `<dest>` que indica o rótulo para o qual será desviado o controle. A sintaxe da instrução de desvio condicional é:

```
br i1 <cond>, label <iftrue>, label <iffalse>
```

O campo `<cond>` deve ser uma variável inteira de um *bit* que indicará qual rótulo será escolhido para desviar. Dessa forma os campos `<iftrue>` e `<iffalse>` devem conter rótulos para os desvios caso a condição seja, respectivamente, verdadeira ou falsa.

Para auxiliar no desvio, são necessárias as instruções `icmp` e `fcmp`, utilizadas, respectivamente, para comparar dois inteiros e para comparar dois números de ponto flutuante. Ambas instruções retornam um valor inteiro de um *bit*. A sintaxe de ambas instruções de comparação é:

```
<result> = icmp <cond> <ty> <op1>, <op2>
```

Como em outras instruções, o campo `<result>` indica a variável que armazenará o resultado, o campo `<ty>` determina qual o tipo de inteiro que será utilizado na comparação e os campos `<op1>` e `<op2>` são os operandos que serão comparados. O campo `<cond>` indica, com a utilização de palavras-chave de LLVM, qual será a comparação realizada entre os operandos, podendo, por exemplo, ser utilizada uma comparação de igualdade ou comparação de desigualdade - se o primeiro valor é maior que o segundo, ou o contrário. Outra comparação possível é saber se o primeiro valor é menor que o segundo e vice-versa. Com o resultado da comparação, é possível realizar o desvio com a instrução `br`, como explicado anteriormente.

### Chamada e Retorno de Função

A instrução `call` é utilizada para realizar uma chamada de função simples, transferindo o fluxo de controle para a função invocada até que haja o retorno. Sua sintaxe utilizada no *framework* LLVM é:

```
<result> = call <ty> <fnptrval>(<function args>)
```

O operando `<result>` serve para armazenar o valor que a função retornar, caso retorne algo, caso contrário esse campo deve ser omitido, não sendo necessária a atribuição. O campo `<ty>` é o tipo de valor que a função retorna, sendo `void` caso não retorne nada. O campo `<fnptrval>` é o ponteiro para a função que será chamada. Por fim, o campo `<function args>` é uma lista dos argumentos que serão utilizados pela

função. Na Listagem 2.16, está uma chamada simples de uma função que retorna um inteiro de 32 *bits* e passa a variável `argc` como argumento da função.

```
%retval = call i32 @test(i32 %argc)
```

Listagem 2.16: Exemplo de chamada de função.

Outra instrução importante relacionada a funções é `ret`, que termina a execução da função e retorna o fluxo de controle para o chamador da função. Existem dois formatos para retorno, um que retorna um valor e outro que somente termina a execução da função. A sintaxe da instrução é:

```
ret <type> <value>
```

Como em outras instruções, o campo `<type>` define o tipo do valor que será retornado e o campo `<value>` indica o valor retornado. No caso de não ser necessário retornar nada, ambos argumentos são substituídos por `void`.

### Outras instruções

Outro tipo de instrução é a de conversão que permite a realização de operações binárias entre tipos distintos. A sintaxe da conversão de inteiro para ponto flutuante é:

```
<result> = sitofp <ty> <value> to <ty2>
```

Assim como em outras instruções, `<result>` indica a variável que conterá o resultado da instrução, `sitofp` indica uma conversão de inteiro para ponto flutuante, `<ty>` indica qual a largura de inteiro que será utilizada, `<value>` qual variável será convertida e `<ty2>` para qual largura de ponto flutuante o valor será convertido.

## 2.3 Conclusão

Este capítulo apresentou diversos sistemas de implementação de compiladores na Seção 2.1. Apesar da vasta gama de sistemas geradores de compiladores, cada um possui alguma especificidade que não lhe permitiu ampla utilização. Entre os principais problemas percebidos estão o desempenho, modularidade, facilidade de uso e legibilidade. Alguns se utilizaram de técnicas de orientação por objetos para suprir a necessidade de modularidade, outros empregaram o paradigma de programação por aspectos, mas essas escolhas prejudicaram a legibilidade em alguns casos. Por facilidade de projeto,

muitos utilizaram a máquina virtual do Java como alvo de geração, prejudicando o desempenho devido ao pouco controle de suas otimizações.

Com os problemas de implementação e desempenho de PEG expostos em *Pappy*, os problemas de legibilidade encontrados em ANTLR e a falta de suporte a etapas avançadas de Lex/YACC, torna-se clara a necessidade de um sistema que busque atenuar ou corrigir esses problemas. Devido à possibilidade de tempo super-linear e restrições com derivações à esquerda, PEG se torna pouco atrativa ao ser comparado com outras soluções. Dessa forma, a criação de um sistema que utilize as ferramentas LEX/YACC internamente para geração das análises léxica e sintática, fornecendo suporte para outras etapas da compilações acaba sendo o ideal almejado pelo projeto implementado nesta dissertação, combinando facilidade de uso para geração com o desempenho linear em relação ao tamanho da entrada e uso de uma linguagem de programação condizente com a necessidade de desempenho.

O projeto do LLVM foi realizado de maneira modular, permitindo o desenvolvimento de novas partes, ou substituição de detalhes facilmente [Lattner & Adve, 2003]. A maior alternativa ao conjunto de ferramentas e *framework* oferecido pelo LLVM é o *Gnu C Compiler* (GCC), um sistema que permite a compilação de diversas linguagens como C/C++, Fortran e Java. Ambos oferecem meios para criação de novos compiladores e ambos são suportados por comunidades enormes, mas, diferentemente do LLVM, a comunidade do GCC tem discutido a viabilidade a longo prazo do sistema ter uma reescrita total de seu código[2]. Além disso, diversas empresas tem migrado suas ferramentas do GCC para o LLVM, como a NVidia[3] e Google[4]. Outras empresas de grande porte participam do conselho regulador do LLVM, como é o caso da Apple.

Dessa forma, é possível perceber que o LLVM possui uma perspectiva melhor para utilização no futuro frente ao GCC. Isso, combinado com a facilidade de utilização do projeto como *backend* de um compilador, permite sua escolha como *bytecode* alvo para implementação desta dissertação e possibilita que o enfoque do trabalho seja no *frontend*, buscando suprir uma necessidade e deixando a separação modular do projeto de um compilador.

[2]http://gcc.gnu.org/ml/gcc/2013-01/msg00313.html
[3]https://developer.nvidia.com/content/new-cuda-now-available
[4]http://google-engtools.blogspot.com.br/2011/05/c-at-google-here-be-dragons.html

# Capítulo 3

# Ambiente de Desenvolvimento de Compiladores

O ambiente de desenvolvimento de compiladores proposto é composto por quatro linguagens de descrição, uma infraestrutura de contexto e uma biblioteca de geração de código. As linguagens de descrição são: *AST*, que define os nodos de árvore utilizados para compor a Árvore de Sintaxe Abstrata (AST); *Front*, que permite a definição da sintaxe concreta da linguagem a ser compilada e a geração do tradutor de programas para AST; *Middle*, que permite realizar modificações na AST como resultado da análise de semântica estática e o que mais o projetista julgar necessário; e *Back*, que permite a geração de código a partir da AST de um programa. A infraestrutura de contexto possui recursos para o caminhamento nas árvores de sintaxe abstrata geradas, e a biblioteca de geração de código possui recursos para manipulação da tabela de símbolos e componentes que implementam a geração de código intermediário de construções recorrentes. Esses componentes implementam conjuntos de ações recorrentes e são genéricos e abrangentes o suficiente para permitir a implementação das principais construções das linguagens imperativas atuais. O código intermediário gerado pelos componentes é utilizado para gerar, ao fim do processo, *bytecode* LLVM, que é o alvo da compilação.

As Figuras 3.1, 3.2 e 3.3 mostram o funcionamento do ambiente de desenvolvimento proposto. A Figura 3.1 mostra o fluxo de dados para a geração dos diversos tradutores a partir das descrições das fases do compilador de uma linguagem L. O diagrama de composição do sistema, que ilustra como cada parte é interligada, está demonstrado na Figura 3.2. A Figura 3.3 exibe o diagrama de fluxo de dados do funcionamento do compilador gerado, mostrando como um programa P escrito na linguagem L é compilado.

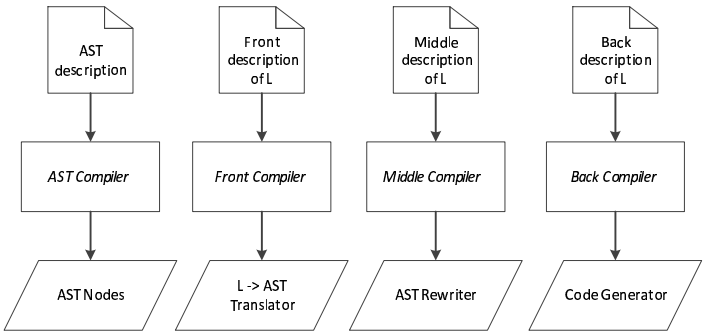


Figura 3.1: Diagrama de fluxo de dados das descrições de uma linguagem L.

Para gerar um compilador da linguagem L, o desenvolvedor deve, como demonstra a Figura 3.1, escrever as descrições do compilador em cada uma das linguagens do ambiente aqui apresentado. Essas descrições são processadas para gerar um conjunto de programas da linguagem C++, que, quando combinados, geram o compilador da linguagem L.

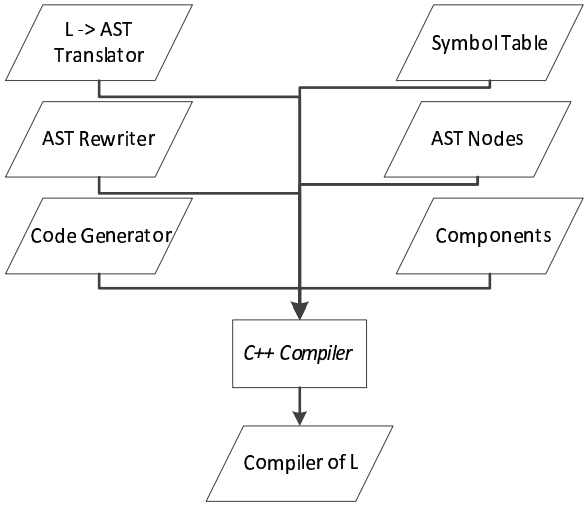


Figura 3.2: Diagrama de link-edição do ambiente de desenvolvimento.

A compilação de um programa P, pelo compilador de L, é descrito no diagrama da Figura 3.3. O código fonte de P é passado para o módulo tradutor de L que foi gerado pela linguagem *Front*, que realiza o *parsing* do código fonte para gerar a AST definida, que é passada para o módulo seguinte, o *AST Rewriter*. Esse módulo é gerado pela linguagem *Middle* e realiza modificações especificadas da AST e, como o módulo anterior, prepara a AST para o próximo módulo. O módulo *AST Rewriter* é utilizado para realizar as verificações da semântica estática da linguagem, portanto pode, se necessário, ser executado em vários passos. O módulo *Code Generator* é gerado pela linguagem *Back* e gera as instruções de código intermediário. O código intermediário

é utilizado pelo módulo *Instruction Generator* para, no fim, gerar o *bytecode* LLVM do programa P escrito na linguagem L. O ambiente oferece os componentes necessários para geração de *bytecode* LLVM, não havendo impedimento para gerar outro tipo de código. Entretanto o implementador deverá prover os componentes necessários.

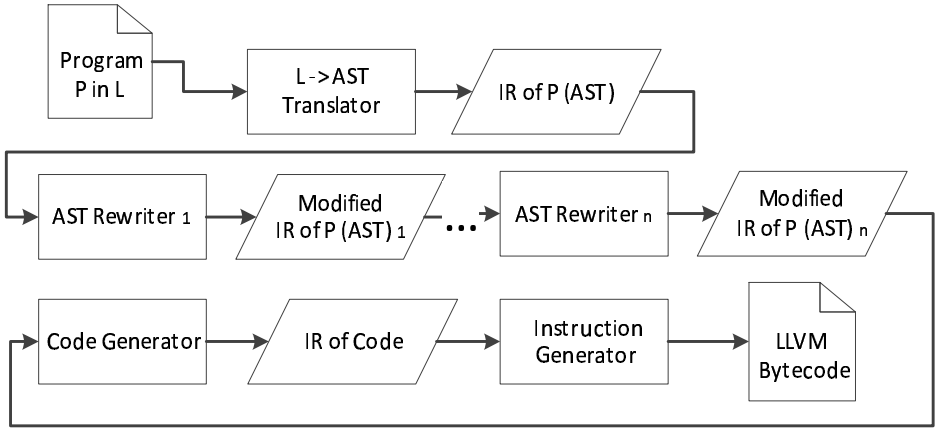


Figura 3.3: Diagrama de fluxo de dados da compilação de um programa P escrito em L.

## 3.1 Linguagens de Descrição

A estrutura das linguagens de descrição foi baseada na linguagem *SCRIPT* [Bigonha, 1998]. O método de análise sintática subjacente do ambiente é LALR(1). Cada uma das quatro linguagens está descrita nas seções a seguir.

### Linguagem *AST*

Essa linguagem destina-se a prover a definição ou especificação dos nodos da árvore de sintaxe abstrata (AST), bem como definir os atributos de cada nodo. A partir dessa descrição, são geradas as classes de C++ que definem esses nodos e que serão utilizadas nas outras partes do sistema. Aos nodos da AST podem ser associados atributos sintetizados, cujos valores são calculados nos programas escritos em *Middle* e *Back* como será descrito posteriormente.

Uma especificação na linguagem *AST* é organizada em quatro seções: uma seção de declaração de tipos complexos de C++, iniciada pela palavra-chave `@declarations`; uma seção iniciada pela palavra-chave `@attributes` para declaração de atributos que utilizam os tipos C++ básicos e os definidos na seção anterior; uma seção para descrição dos atributos de nodos do tipo folha, iniciada pela palavra-chave `@leaves`. Essa seção consiste em uma lista de declarações de variáveis cujos nomes indicam nodos da árvore;

e a última seção define o formato que a AST gerada terá, é iniciada pela palavra `@nodes` e pode conter descrições de nodos no seguinte formato:

$$\texttt{não-terminal ::= rhs}_1 \texttt{ | ... | rhs}_n \texttt{ ;}$$

O `não-terminal` representa um nodo para o qual gera-se uma classe que tem seu identificador como nome. Cada `rhs` fornece a estrutura do nodo representado pelo não-terminal que ocorre do lado esquerto. A partir dessa estrutura é gerado sua devida classe. Um `rhs` é uma combinação de símbolos terminais, grafados entre aspas, e símbolos não-terminais, grafados como identificadores possivelmente seguidos de índices, que são usados para diferenciar distintas ocorrências de um mesmo não-terminal. A concatenação desses símbolos é usada como nome da classe correspondente ao nodo dessa descrição, entretanto os índices dos não-terminais não são utilizados na nomenclatura da classe. Além disso a classe gerada estende a classe correspondente do `não-terminal`, estabelecendo uma hierarquia para a árvore de sintaxe abstrata.

Os membros de cada classe gerada são, além dos atributos descritos na devida seção de *AST*, apontadores para objetos das classes de nodos não-terminais, definindo as estruturas da árvore de sintaxe abstrata correspondente. Símbolos não-terminais com índices alteram somente o identificador gerado para o apontador, uma vez que a classe gerada não utiliza o índice em sua nomenclatura. Além das classes, é gerada também uma tabela contendo a descrição das classes geradas e dos terminais da árvore de sintaxe abstrata para verificação durante a compilação das linguagens subsequentes do ambiente.

```
1  @declarations
2  @attributes
3  exp { component::Type* type; } ;
4  @leaves
5  int num;
6  float float_point;
7  @nodes
8  start ::= "calc" exp ;
9  exp ::= "add" exp1 exp2
10   | "sub" exp1 exp2
11   | "mult" exp1 exp2
12   | "div" exp1 exp2
13   | num
14   | float_point
15   ;
```

Listagem 3.1: Exemplo de descrição da AST utilizada na descrição de uma calculadora.

A Listagem 3.1 apresenta a descrição da *AST* de uma linguagem de calculadora. Na linha 4, é definido que o nodo `exp` tem como atributo uma variável `type` cujo tipo é uma classe que faz parte da infraestrutura de geração de código e é detalhada

no Capítulo 4. Nas linhas 6-7 são definidos que os nodos `num` e `float_point` são do tipo inteiro e ponto flutuante, respectivamente. Por último, nas linhas 9-16, é definido que serão geradas as classes `start` e `exp` e suas subclasses, referentes às produções que definem `start` e `exp`. Para a alternativa `"add" exp1 exp2` é gerada uma classe cuja superclasse é `exp` e que conterá três campos internos: o literal `"add"` e os dois apontadores para `exp`.

### Linguagem *Front*

A linguagem *Front* tem como objetivo a descrição léxica e sintática da linguagem para a qual se implementa o compilador, tendo como saída um analisador sintático da linguagem que produzirá a árvore de sintaxe abstrata a ser utilizada nas próximas etapas.

Uma especificação na linguagem *Front* é uma gramática composta por símbolos não-terminais, terminais, algumas palavras-chaves, regras de produção concretas e regras de reescrita. As palavras chaves são: `@grammar`, que inicia a seção de projeto sintático, `@domains` que agrupa os não-terminais da gramática concreta em não-terminais da sintaxe abstrata, `@lexical`, que estabelece a seção de detalhamento léxico, e `UNIT`, que define os símbolos terminais especiais grafados como identificadores. Outros símbolos terminais são escritos entre aspas. Símbolos não-terminais são identificadores que ocorrem no lado esquerdo de uma regra de produção concreta ou em `UNIT` e utilizados no lado direito de regras de produção concreta ou em regras de reescrita.

A parte sintática de um programa *Front* é um conjunto de regras de produção de uma gramática livre de contexto, possivelmente acompanhadas de regras de reescrita para geração de nodos da AST. O formato geral utilizado pelas regras de produção é:

não-terminal ::= rhs$_1$ : regra de reescrita | ... | rhs$_n$ : regra de reescrita ;

Cada `rhs` é uma combinação de símbolos não-terminais e terminais, sendo os terminais grafados entre aspas e não-terminais grafados como identificadores podendo ser seguidos de índices para diferenciar símbolos iguais. O `rhs`, em conjunto com o `não-terminal`, é utilizado como regra de produção concreta para o reconhecimento sintático da linguagem e para geração da AST. Essas regras de produção concreta são utilizadas para gerar produções do reconhecedor sintático *Yacc*. Caso não haja regra de reescrita, os símbolos não-terminais e terminais são concatenados para geração do nodo da árvore de sintaxe abstrata dessa produção, os índices existentes nos símbolos são ignorados durante a concatenação e a classe necessária tem sua existência verificada na tabela gerada pela linguagem *AST*. Durante essa concatenação, os sím-

bolos não-terminais que participam de agrupamentos definidos na seção `@domains` são substituídos pelo correspondente símbolo não-terminal abstrato.

As regras de reescrita permitem controlar a estrutura dos nodos gerados para a AST. Para tanto, podem conter uma combinação de símbolos não-terminais e terminais encapsulada em colchetes denotando a estrutura do nodo da AST ou repetir um único não-terminal da regra de produção concreta, indicando que não é produzido nodo nessa produção. Os não-terminais de uma regra de reescrita devem ocorrer na regra de produção concreta associada, mas podem criar quantos símbolos terminais forem necessários. Assim como na regra de produção concreta, o nodo gerado é determinado pela concatenação dos símbolos não-terminais e terminais, e a existência de sua classe verificada na tabela gerada pela linguagem *AST*. Nessa concatenação, também ocorre a substituição dos símbolos não-terminais que participam de agrupamentos definidos na seção `@domains`.

Além das regras no formato `não-terminal ::= rhs`, na parte léxica é possível ter uma regra no formato `não-terminal === intervalo`, sendo cada intervalo definido por dois caracteres entre aspas separados por reticências e as alternativas separadas por uma barra vertical. Cada intervalo especificado denota que aquela variável pode produzir o que houver entre os caracteres utilizados, sendo equivalente utilizar alternativas para cada caractere dentro do intervalo. Por exemplo, `digit === "0" ... "9"` é equivalente a escrever `digit === "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"`.

O compilador da linguagem *Front* gera os devidos programas *Lex* e *Yacc* para realizar o *parsing* da linguagem concreta descrita e a construção da árvore de sintaxe abstrata. Como descrito previamente, é utilizada a composição dos símbolos terminais e não-terminais para gerar o nodo que será utilizado na construção da árvore de sintaxe abstrata, além de ser substituído, nas regras de reescrita, o identificador do não-terminal que tiver seu domínio definido. Para garantir que somente classes de nodos existentes sejam utilizadas, é utilizada a tabela das classes geradas pela linguagem *AST*.

A Tabela 3.1 contêm quatro exemplos para ilustrar as regras de produção concreta, as regras de reescrita e árvores de sintaxe abstrata correspondentes. Na primeira coluna estão exemplos da gramática de *Front* em sua parte sintática e na segunda coluna estão as ASTs correspondentes. O primeiro exemplo não faz uso de regra de reescrita, portanto sua AST é gerada pela sintaxe concreta. O segundo exemplo, apesar de usar as regras de reescrita, não altera as regras de produção concreta, portanto a AST gerada é a mesma da anterior. No terceiro exemplo, A segunda segunda regra de reescrita é utilizada para não produzir o nodo na regra de produção concreta, dessa

Tabela 3.1: Exemplos das transformações de *Front*.

| **Gramática de *Front*** | **AST Gerada** |
|---|---|
| A := B<br>B := C<br>C := a | A<br>↓<br>B<br>↓<br>C<br>↓<br>a |
| A := B : [B]<br>B := C : [C]<br>C := a | A<br>↓<br>B<br>↓<br>C<br>↓<br>a |
| A := B : [B]<br>B := C : C<br>C := a | A<br>↓<br>B<br>↓<br>a |
| A := B : [B]<br>B := C : [C]<br>C := a<br>A,B,C : X | X<br>↓<br>X<br>↓<br>X<br>↓<br>a |

forma a AST gerada é mais simples e direta que as anteriores. Por fim, no último exemplo, é utilizada uma definição de domínio para substituir os nodos da árvore, portanto a AST gerada contêm nodos distintos das ASTs anteriores.

O exemplo apresentado na Listagem 3.2 mostra descrição da sintaxe para a linguagem de expressões. No exemplo, a seção léxica define que as variáveis `num` e `float` são símbolos terminais sendo `num` um número inteiro e `float` qualquer decimal. Na seção sintática da descrição, são definidas as operações de adição, subtração, multiplicação e divisão, além da priorização de expressões entre parênteses e uso de valores como os definidos na seção léxica. Como definido pela linguagem *Front*, as produções que após os dois pontos só possuem uma variável não terão nodos específicos na AST. Portanto, durante a geração da AST, não será gerado um novo nodo ao se realizar o casamento das produções nas seguintes expressões: `exp ::= term : term`; `term ::= factor : factor` e `factor ::= "(" exp ")" : exp`. Por fim, as produções como `exp ::= exp "+" term : ["add" exp term]` indicam que ao realizar o

```
1  @grammar
2  start ::= exp : [ "calc" exp ];
3  exp ::= exp "+" term : ["add" exp term]
4          | exp "-" term : ["sub" exp term]
5      | term : term
6      ;
7  term ::= term "*" factor : ["mult" term factor]
8      | term "/" factor : ["div" term factor]
9          | factor : factor
10     ;
11 factor ::= num
12     | float_point
13     | "(" exp ")" : exp
14     ;
15 @domains
16 term, factor : exp;
17 @lexical
18 UNIT ::= num
19         | float_point ;
20 digit === "0" ... "9" ;
21 numeral ::= digit numeral | digit ;
22 num ::= numeral | "-" numeral ;
23 float_point ::= numeral "." numeral | "-" numeral "." numeral ;
```

Listagem 3.2: Exemplo de descrição da sintaxe da calculadora.

casamento dessas expressões será gerado um novo nodo com o formato indicado entre colchetes. Ou seja, na regra exemplificada, o nodo que será produzido não terá o formato `exp "+" term`, mas sim o formato `"add" exp term`. O mesmo ocorre para as demais produções. A definição de domínio `term, factor : exp`, assinala que o não-terminal `term` utilizado na produção abstrata anterior será substituído pelo nodo `exp` da árvore de sintaxe abstrata.

### Linguagem *Middle*

A linguagem *Middle* permite realizar transformações da estrutura da AST, registrar informações nessa árvore e, por meia dessas transformações e informações, permite realizar a análise semântica estática do programa na forma de AST. Um programa em *Middle* é um conjunto de especificações de transformações de nodos de AST. Genericamente, uma transformação possui o formato:

```
padrão de nodo 1 => padrão de nodo 2 where { EXTENDED_C++CODE } ;
```

Nodos da AST com o `padrão de nodo 1` será transformado em nodos com o `padrão de nodo 2` e o código `EXTENDED_C++CODE` é executado para calcular atributos da AST e permitir a criação de novos nodos em sua estrutura. Cada padrão de nodo possui o seguinte formato:

```
não-terminal ::= rhs
```

O `rhs` é uma combinação de símbolos não-terminais e terminais cuja concatenação forma o nome da classe da árvore de sintaxe abstrata. Como nas linguagens anteriores, os símbolos não-terminais podem ser seguidos de índices para diferenciá-los, caso haja símbolos iguais no padrão de nodo, entretanto os índices são ignorados durante concatenação do nome da classe. A verificação de existência da classe empregada é realizada por meio da tabela das classes geradas pela linguagem *AST*. Para o `padrão de nodo 1`, a classe definida é utilizada para a geração do padrão *Visitor*. Para o `padrão de nodo 2`, a classe definida é aplicada na instanciação do novo nodo que substitui o nodo visitado pelo `padrão de nodo 1`.

A criação de novos nodos da árvore é realizada via uma extensão da linguagem C++, reconhecida pelo parser do compilador da linguagem *Middle*. Essa extensão, acrescenta, entre as expressões da linguagem C++, construção para a criação de novos nodos como uma combinação de símbolos terminais e não-terminais entre colchetes. Como em outros casos, o identificador da classe do nodo é obtida pela concatenação desses símbolos e a verificação da existência de sua classe é realizada na tabela gerada pela linguagem *AST*. Após obtenção do identificador da classe, é criada uma nova instancia dela, que é atribuída de acordo com o código descrito. Ressalta-se que essas extensões oferecem grande poder de expressão, mas dificulta o reúso de ações semânticas.

Em uma especificação é possível omitir tanto o segundo padrão de nodo quanto o código, permitindo a realização de somente a transformação para a reordenação dos nodos da AST; somente a execução do código para o cálculo dos atributos da árvore e a criação de novos nodos ou nenhuma ação, realizando somente o caminhamento da AST.

*Middle* pode ter várias especificações que devem ser executadas em uma ordem predefinida pela marca `@pass n` em que n é o número de ordem, e marca o início de cada especificação. O compilador da linguagem *Middle* gera um módulo de *AST Rewriter* para cada `@pass` definido. Cada módulo gerado consiste na classe que implementa o padrão *Visitor* necessária para realizar o caminhamento em pós-ordem da AST e executar as ações descritas na linguagem. Assim como em *Front*, é utilizada a tabela de classes gerada pela linguagem *AST* para verificação da utilização correta dos nodos.

O exemplo apresentado na Listagem 3.3 realiza a declaração de uma variável e a verificação de tipos em dois passos de *Middle*. O primeiro passo é realizado a declaração da variável, sendo necessário a definição do tipo da variável. Essa descrição do tipo é realizada na linha 6 na produção de `"integer"` e na linha 7 na produção de `"float"` via atributo `_type`. Na Linha 3, é descrita a produção para declaração de variável, incluindo uma transformação da AST. Nessa transformação é retirado um nodo da

```
1  @declaration
2  @pass 1
3  decl ::= type id => decl ::= id where{
4          decl->decl_attr = cb->VariableDeclare(id, type->_type);
5      };
6  type ::= "integer" where{ type->_type = component::Type::GetLong(); };
7  type ::= "float" where { type->_type = component::Type::GetFloat(); };
8
9  @pass 2
10 exp ::= "add" exp1 exp2 where {
11         if(exp1->type->compare(exp2->type))
12             exp->type = exp1->type;
13         else{
14             if(exp1->type->isIntegerTy())
15                 exp1 = ["floatc" exp1]
16             else
17                 exp2 = ["floatc" exp2]
18             exp->type = component::Type::GetFloat();
19         }
20     };
21 expression ::= num where { exp->type = component::Type::GetInt(); };
22 expression ::= float where { exp->type = component::Type::GetFloat(); };
```

Listagem 3.3: Exemplo de descrição de *Middle* para declaração de variável e verificação de tipos

árvore, simplificando-a. No segundo passo, na verificação de tipos, para os literais, somente é necessário descrever qual o seu tipo, isso é realizado na linha 21 em `num`, e na linha 22 em `float` via atributo `type`. Por fim, as expressões devem realizar a verificação de tipos e, se necessário, transformar a árvore para realizar a conversão de inteiro para número de ponto flutuante. Como essa verificação é similar para outras operações da linguagem, está exemplificado somente uma das transformações. Essa conversão é demonstrada na Linha 15 com o comando `exp1 = ["floatc" exp1]` que cria um novo nodo da árvore com os filhos `"floatc"` e `exp1` e substitui o nodo na variável `exp1` pelo nodo que permitirá a realização da conversão do inteiro para ponto flutuante. De modo complementar ocorre na Linha 17 com a variável `exp2`. Assim como os valores, as expressões também devem definir seus tipos para subsequente análise.

### Linguagem *Back*

A linguagem *Back* tem como objetivo a geração de código a partir da AST trabalhada nas fases de compilação anteriores. Assim como em *Middle*, um programa em *Back* é um conjunto de regras de sintaxe abstrata com o seguinte formato:

$$\texttt{não-terminal ::= rhs}_1\ \texttt{\{ C++CODE}_1\ \texttt{\}} \mid \ldots \mid \texttt{rhs}_n\ \texttt{\{ C++CODE}_n\ \texttt{\}}\ ;$$

Cada `rhs` é uma combinação de símbolos terminais e não-terminais cuja concatenação é identificador da classe de *AST* utilizada para o caminhamento do padrão

*Visitor*. Como nas linguagens anteriores, os símbolos terminais são grafados entre aspas e os não-terminais são grafados como identificadores, podendo ser seguido de índices para diferenciar símbolos iguais. Também como nas linguagens anteriores, os índices são ignorados para a definição do nome das classes. Para verificação estática, durante a compilação da descrição da linguagem, os nodos descritos em *Back* possuem sua existência verificada na tabela gerada pela linguagem *AST*. De igual maneira, cada `C++CODE` é o código C++ executado como ação semântica para geração de código do nodo descrito. A gramática da linguagem é iniciada com a palavra chave `@back`, podendo ser precedida por uma seção de declarações C++ para uso de construções complexas durante a geração do código.

O compilador da linguagem *Back* gera o módulo *Code Generator*, e, assim como os módulos de *AST Rewriter*, gera a classe necessária para se realizar o caminhamento em pós-ordem na AST utilizando o padrão de projeto *Visitor*. Diferentemente de *Middle*, *Back* não produz como resultado uma nova AST, a AST de entrada é simplesmente percorrida para as ações de gerações de código.

```
1  @declarations
2  @back
3  start ::= "calc" exp {
4          cb->GenLLVM(exp->exp_attr);
5      };
6
7  exp ::= "add" exp1 exp2 {
8          exp->exp_attr = cb->Apply(component::GetBinOpe("+", exp1->_type),
               exp1->exp_attr, exp2->exp_attr);
9      }
10     | "sub" exp1 exp2 {
11         exp->exp_attr = cb->Apply(component::GetBinOpe("-", exp1->_type),
               exp1->exp_attr, exp2->exp_attr);
12     }
13     | "mult" exp1 exp2 {
14         exp->exp_attr = cb->Apply(component::GetBinOpe("*", exp1->_type),
               exp1->exp_attr, exp2->exp_attr);
15     }
16     | "div" exp1 exp2 {
17         exp->exp_attr = cb->Apply(component::GetBinOpe("/", exp1->_type),
               exp1->exp_attr, exp2->exp_attr);
18     }
19     | num   {
20         exp->exp_attr = cb->LoadInteger(integer_num,
               component::Type::GetInt());
21     }
22     | float_point {
23         exp->exp_attr = cb->LoadFP(float_num, component::Type::GetFloat());
24     }
25     | "floatc" exp1 {
26         exp->exp_attr = cb->Apply(component::GetUnOpe("cast",
               component::Type::GetFloat()), exp1->exp_attr);
27     };
```

Listagem 3.4: Exemplo de descrição da geração de código de expressão

O exemplo da Listagem 3.4 apresenta a geração de código para uma linguagem de expressões. Cada regra definida possui uma ação de geração de código utilizando componentes da biblioteca detalhada no Capítulo 4 para preencher os atributos `exp_attr` da gramática que contêm a lista dos códigos gerados pelos componentes. O código gerado pela execução dos componentes é agregado à lista de código do atributo de nodo sendo calculado. As regras `num` (Linha 19) e `float_point` (Linha 22) utilizam, respectivamente, `LoadInteger` e `LoadFP` para gerar o código intermediário necessário para empregar seus devidos literais. A produção responsável pela conversão de um valor inteiro para um valor de ponto flutuante da linha 25 utiliza o componente `Apply` para gerar a operação unária obtida por `GetUnOpe("cast",...)`. O componente `Apply` também é utilizado pelas outras operações, entretanto, por se tratar de operações binárias é utilizado o `GetBinOpe` para obter cada operação necessária. A raiz da árvore, na produção `start` (Linha 3), contém a ação de executar o método `GenLLVM` que produz um arquivo com o *bytecode* LLVM preparado para execução. Isso é possível pois a lista de código intermediário gerada pelo programa sendo compilado é um dos campos do atributo `exp_attr` do nodo "`exp`" dessa produção.

## 3.2 Conclusão

Neste capítulo, foi apresentado o ambiente de desenvolvimento proposto para geração de compiladores de linguagens de programação. Esse ambiente proporciona, via quatro linguagens de descrição, a separação dos detalhes de um compilador, permitindo que o implementador possa focar na resolução de cada etapa e assim projetar melhor a linguagem e o compilador.

# Capítulo 4

# Infraestrutura de Geração de Código

A geração de código oferecida pela infraestrutura funciona em duas etapas. Primeiro, o programa-fonte é traduzido para uma lista de instruções de um código intermediário e, posteriormente, percorrendo essa lista, é gerado código LLVM. A produção da lista de instruções é apoiada por uma biblioteca de componentes que implementam as ações de geração de código características das linguagens imperativas. Os componentes descritos neste capítulo podem receber e retornar um conjunto de atributos, entre eles está a lista de instruções geradas pelo componente. A geração de código LLVM é realizada pelo componente `GenLLVM`, que percorre a lista de instruções que lhe for passada e produz, utilizando uma estrutura da biblioteca LLVM denominada *IRBuilder*, o *bytecode* do código descrito.

A infraestrutura de geração possui uma tabela de símbolos que é preenchida pelos componentes que tratam dos detalhes referentes à declaração de variáveis e é utilizada por outros componentes para recuperação das informações nela armazenada. Dessa forma, o implementador não precisa preocupar-se com detalhes típicos de implementação de variáveis, tais como o local em que a variável será armazenada ou seu tamanho, uma vez que esses detalhes são encapsulados pelos componentes.

A infraestrutura de geração de código também compreende as classes necessárias para implementação dos componentes, entre as quais estão: `Attributes`; `Command_Attr`; `Expression_Attr`; `Declaration_Attr`; `Parameter_Attr`; `Dimension_Attr`; `BinOp`; `UnOp` e `Type`. As primeiras seis são utilizadas para declarar parâmetros denotando atributos que são passados aos componentes, as duas seguintes são utilizadas para identificar os operadores e a última classe é utilizada para determinar o tipo de uma variável ou literal. Objetos da classe `Attributes` contêm

um ponteiro para uma lista de instruções que será preenchida pelos componentes e será percorrida para geração do *bytecode* LLVM. As classes referentes a atributos são especializações de `Attributes`. Nas seções a seguir, essas classes são detalhadas, juntamente com cada componente que as utiliza.

Os componentes e métodos empregados no processo de geração de código estão divididos entre as categorias: *Expressões*; *Comandos*; *Declarações*; *Escopo* e *Verificação de tipos*. Os componentes referentes a *expressões* são responsáveis por gerar instruções para realizar as operações binárias, unárias, chamadas de função e por criar as instruções de carga de valores e variáveis da memória. Os componentes de *comandos* são responsáveis por gerar instruções para realizar os comandos de atribuição, condicionais, repetições e retorno de função. Os componentes de *declaração* são responsáveis por criar as amarrações das variáveis simples, dos arranjos, das estruturas, das classes e das funções, armazenando suas informações na tabela de símbolos para posteriormente alocá-las. Os métodos referentes ao *escopo* são responsáveis por controlar os blocos e permitir a manipulação da tabela de símbolos. Os métodos referentes a *verificação de tipos* permitem buscar as variáveis e funções declaradas e recuperar o tipo armazenado.

O componente `BuildSequence` (Listagem 4.1), cuja ação semântica é a concatenação de duas listas de instruções, é um componente genérico o suficiente para utilização em todas as categorias do processo de geração. Os parâmetros desse componente são os atributos contendo as listas de instruções que serão concatenadas.

```
Attributes* BuildSequence(Attributes* code1, Attributes* code2);
```

Listagem 4.1: Componente para sequência de instruções.

Outro componente genérico da infraestrutura de geração de código que permite que alguns componentes, como o de repetição, possam ignorar alguns de seus parâmetros é o `BuildSkip` (Listagem 4.2). Não é necessário nenhum parâmetro para esse componente, e seu retorno anula o atributo para permitir que o componente que o recebe como parâmetro possa gerar o código correto.

```
Attributes* BuildSkip();
```

Listagem 4.2: Método para ignorar os parâmetro de componentes.

Nas seções a seguir, os outros componentes de cada uma das categorias são detalhados.

## 4.1 Tabela de Símbolos

A tabela de símbolos implementada pela infraestrutura de geração de código possui a estrutura em que cada nível da tabela contém uma tabela de *hash*, como sugerido por Aho et al. [2007]. As operações implementadas para a tabela de símbolos permitem: a abertura de um novo nível da tabela de símbolos - cria um novo nível da tabela de símbolos; o fechamento do último nível aberto na tabela de símbolos - impede o acesso do último nível aberto; a inserção de um símbolo no nível atual; a busca de um símbolo nos níveis abertos - busca por um símbolo armazenado na tabela do último nível aberto, caso não o encontre, busca nos níveis abertos anteriores até não ter mais níveis abertos, retornando um valor nulo caso o símbolo não seja encontrado; e o *reset* da tabela de símbolos. Esse último método restabelece os níveis da tabela de símbolos como abertos e retorna ao nível inicial, sem alterar os símbolos neles armazenados. Isso é necessário porque cada fase da compilação realiza um caminhamento da $AST$ que, possivelmente, altera a tabela de símbolos. É importante ressaltar que um caminhamento subsequente do mesmo nodo deve ter acesso ao mesmo nível da tabela de símbolos que a fase anterior, o que não é possível sem a execução do método de *reset*, pois a cada fase de compilação, novos níveis da tabela são abertos e fechados, impedindo a busca correta das variáveis declaradas. Com a utilização do método *reset* é possível reiniciar a tabela de símbolos e, assim, com o caminhamento da AST e os métodos de abertura e fechamento de nível, utilizar os níveis previamente abertos na tabela de símbolos.

## 4.2 Instruções do Código Intermediário

O código intermediário tem o intuito de propiciar uma maior separação entre a linguagem fonte, os componentes e o bytecode LLVM, permitindo que o foco dos componentes seja somente as ações semânticas básicas necessárias para a geração de código, não centrando nas particularidades da arquitetura LLVM, mas sim em instruções abstratas.

São dois os formatos da representação do código intermediário criado, sendo a primeira forma a seguinte:

```
$i = instrução op1 op2 ...
```

O resultado da execução de uma instrução fica armazenado na variável `$i`, o seu conteúdo depende da instrução, sendo usualmente o valor que a mesma produzirá. Por exemplo, uma operação binária terá o valor de sua resposta armazenado na variável `$i` correspondente. O campo `instrução` define o que será executado e os operandos

necessários para sua execução. Cada campo `opK` é um operando que será utilizado pela instrução durante sua execução.

O segundo formato de representação do código intermediário criado é o seguinte:

```
instrução op1 op2 ...
```

Como no formato anterior, o campo `instrução` define o que será executado e os operandos necessários para sua devida execução. Igualmente, cada campo `opK` é um operando que será utilizado durante a execução. As instruções que utilizam esse formato são as que não produzem valor, por exemplo: a instrução de armazenamento; a instrução condicional; a instrução de repetição e a instrução de chamada de função.

A operação de alocação de espaço de memória é realizada pela instrução `Alloc`, que pode possuir até dois operandos. O primeiro operando é o tipo do valor a ser armazenado, que é um objeto da classe `Type`. O segundo operando é usado na alocação de arranjos e se refere à dimensão do arranjo alocado, utilizando um objeto da classe `Dimension_Attr` contendo as informações referentes ao tamanho do arranjo.

O resultado de uma alocação de memória é o endereço do espaço de memória alocado, ou seja, as instruções que necessitam do endereço de uma variável devem utilizar o resultado de sua alocação, dispensando a necessidade de se realizar cálculos referentes à memória para se acessar variáveis, sendo preciso calcular somente o índice de arranjos ou membros de estruturas e classes.

A operação de carga de um endereço da memória é realizada pela instrução `Load`, cujo único operando é o endereço de memória a ser carregado. Como supra-expendido, deve ser utilizada uma variável `$i` contendo o resultado de uma alocação de memória. Para a utilização de arranjos, estruturas ou classes é necessário calcular o deslocamento para um índice ou membro.

No caso de necessitar do cálculo de índice de um arranjo ou de um membro de uma estrutura ou de uma classe, é preciso da instrução `GEP` cujos operandos são uma variável `$i` com o endereço de memória e outra variável `$i` ou o valor constante do índice de deslocamento necessário. Para o cálculo do primeiro índice de arranjo ou primeira variável de uma estrutura ou primeiro membro de uma classe, o primeiro operando da instrução `GEP` deve ser o ponteiro para a instrução `Alloc` utilizada para alocar a variável. Os índices e membros subsequentes utilizam o valor produzido por essa instrução como primeiro operando. O resultado final é o endereço calculado que poderá ser utilizado pela instrução `Load` ou pela instrução de armazenamento.

O contrário da operação de carga, o armazenamento de um valor em um endereço de memória, é realizado pela instrução `Store` que possui como operandos o endereço em que o valor será armazenado e a instrução contendo o valor a ser armazenado.

Assim como a instrução `Load`, deve ser utilizada uma variável `$i` contendo o endereço de memória no qual o valor será armazenado. Essa variável `$i` pode ser proveniente tanto de uma instrução `Alloc`, quanto de uma instrução `GEP`, dependendo do tipo de seu valor armazenado, uma vez que arranjos, estruturas e classes necessitam calcular um deslocamento adicional a partir de sua alocação base. O valor que será armazenado deve ser uma variável `$i` produzida pelas instruções de operação binária, unária ou por instrução de chamada de função. Também é possível armazenar valores de constantes inteiras e de ponto flutuante. Essas constantes são usadas no formato de função, onde `ConstInt` denota constantes inteiras e `ConstFP` constantes de ponto flutuante. Ambas utilizam um objeto da classe `Type` como primeiro parâmetro para passar a largura da constante. O segundo parâmetro usado é o valor da constante, com valores inteiros para constantes de `ConstInt` e valores de ponto flutuante para `ConstFP`.

Uma operação binária ou unária sobre operandos podem ser realizada com, respectivamente, as instruções `BinOp` e `UnOp`. Ambas instruções possuem como operandos o operador a ser aplicado e os valores aos quais o operador será aplicado. Assim como a instrução de armazenamento, os operandos devem ser variáveis `$i` produzidas por outras operações. A utilização de constantes deve ser realizada como explicado previamente.

A chamada de uma função é realizada pela instrução `Call` que necessita, como operandos, uma referência para a função que será chamada e a lista de argumentos da chamada. Para a lista de argumentos da chamada, deve ser utilizada a variável `$i` produzida pela instrução `ArgList`, cujo operando deve ser um valor ou endereço que será avaliado durante a chamada da função. Como em outras instruções anteriores, esse valor pode ser uma constante ou uma variável `$i`, um endereço de uma variável ou o cálculo de um deslocamento de índice.

A finalização de uma função é realizada pela instrução `Return` que, no caso de uma função, também retornará um valor para seu chamador. O único operando dessa instrução é o valor ou variável que será retornado para o chamador. Caso seja o retorno de uma função com tipo de retorno vazio, não é necessário nenhum operando.

Tanto o comando quanto a expressão condicional são realizados pela instrução `Conditional`, cujos operandos são listas de instruções. O primeiro operando é uma lista contendo as instruções necessárias para se gerar a condição de escolha dos subcomandos definidos. Os dois operandos seguintes são listas de instruções representando os subcomandos possíveis do comando ou expressão condicional.

O comando de repetição é implementado pela instrução `Repetition` e, assim como a anterior, utiliza listas de instruções como operandos. O primeiro operando é uma lista de instruções utilizadas para declarar e inicializar variáveis de controle de

um comando de repetição. O segundo operando é a lista de instruções para se gerar a condição de finalização da repetição. O terceiro operando é uma lista de comandos utilizada para realizar incremento das variáveis de controle da repetição. Por fim, o quarto operando é a lista de instruções que serão executadas enquanto a condição do segundo operando for avaliada como verdadeira.

O exemplo de programa contido na Listagem 4.3 está na linguagem *Small* e é usado nesta seção para exemplificar as instruções e como essas são geradas. No Capítulo 5, mais exemplos com essa linguagem são descritos e minuciosamente analisados. A descrição completa da linguagem de programação *Small* se encontra no Apêndice B.

```
1 program{
2   integer x;
3   x := 5 + 6;
4   output x;
5 }
```

Listagem 4.3: Exemplo de um programa em *Small*.

A Listagem 4.4 mostra o código gerado pela compilação do programa da Listagem 4.3, utilizando as instruções do código intermediário. Na primeira linha, está a instrução utilizada para alocar o espaço de memória necessário para a variável declarada no programa. Na linha 2, está a instrução que aplicará a operação binária para se obter o valor necessário. Na linha 3, está a instrução que realiza o armazenamento do valor, para tanto ela necessita das variáveis `$0` e `$1` que contêm, respectivamente, o endereço de memória em que a variável declarada no programa se encontra e o valor calculado pela operação binária. Na linha 4, está a instrução `Load`, utilizada para carregar o valor da variável `x` armazenado na memória. Na linha 5, está a instrução `ArgList`, que cria a lista de argumentos para permitir a sua utilização em uma chamada de função. Por fim, na linha 6, está a instrução `Call` que realiza a chamada para a função de impressão. Isso é feito por meio da chamada da função *printf*, fornecida pela biblioteca de C.

```
1 $0 = Alloc type::long "x"
2 $1 = BinOp "+"  ConstInt(type::long, 5) ConstInt(type::long, 6)
3 Store $0 $1
4 $2 = Load $0
5 $3 = ArgList($2)
6 $4 = Call "printf" $3
```

Listagem 4.4: Instruções da infraestrutura do programa *Small* da Listagem 4.3.

A Listagem 4.5 mostra o código LLVM do programa apresentado na Listagem 4.3. Na linha 1, está declarada a cadeia de caracteres que é utilizada pela função *printf*,

que está definida na linha 2. A instrução da linha 5 realiza a alocação de variável requerida, sendo que essa variável possui seu valor armazenado pela instrução na linha 6. Como a operação binária do exemplo foi realizada sobre duas constantes, o emissor de instruções do LLVM otimizou essa operação, armazenando seu resultado na linha 6. Na linha 7, está a instrução que realiza a carga da variável no modelo *SSA* para que ela seja utilizada na linha 8 pela chamada da função *printf*. A linha 9 contém somente a instrução de retorno da função principal, encerrando sua execução.

```
1  @0 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
2  declare i32 @printf(i8*, ...)
3  define i32 @main() {
4  entry:
5    %x = alloca i64
6    store i64 11, i64* %x
7    %0 = load i64* %x
8    %1 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]*
         @0, i32 0, i32 0), i64 %0)
9    ret i32 0
10 }
```

Listagem 4.5: Bytecode LLVM do programa *Small* da Listagem 4.3.

Na Figura 4.1, está a representação da árvore de sintaxe abstrata (*AST*) do programa da Listagem 4.3. Essa *AST* será percorrida em ordem pós fixada e, dessa forma, serão geradas as instruções da linguagem utilizada pela infraestrutura.

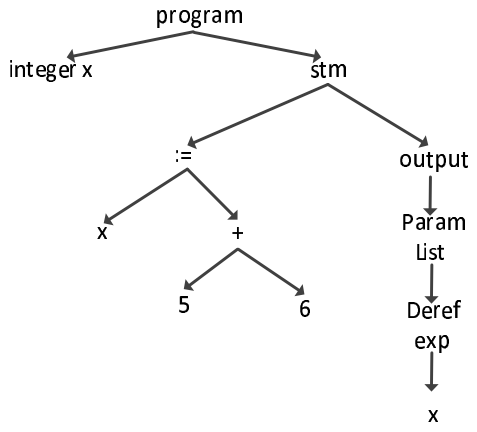


Figura 4.1: Árvore de sintaxe abstrata do programa de *Small* da Listagem 4.3.

Na Figura 4.2, está representada, junto à *AST*, a instrução gerada no nodo que contém a declaração `integer x`. A seta em vermelho indica que o resultado da instrução `Alloc` será armazenado na tabela de símbolos no local devido.

A árvore de sintaxe abstrata representada na Figura 4.3 mostra a recuperação do endereço da variável previamente declarada para uso no nodo "`x`". A seta vermelha

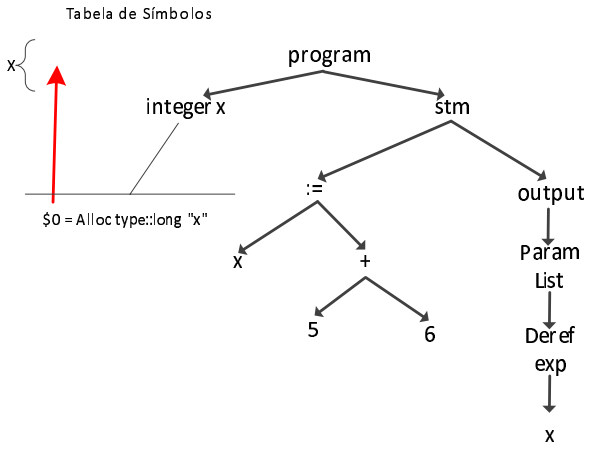


Figura 4.2: AST mostrando a geração da instrução de alocação.

indica que a variável `$0` cujo valor é produzido pela instrução `Alloc` e armazenada na tabela de símbolos será buscada e utilizada pelo nodo. A Figura 4.4 mostra a *AST* com geração de carga da constante 5, feita via função `ConstInt`. A geração da instrução do operador binário de adição está na Figura 4.5. A seta vermelha nessa figura indica a procedência das variáveis `$i` utilizadas na construção da instrução `BinOp` gerada pelo nodo "+". O comando referente à atribuição está vinculado ao nodo "`:=`" onde possui sua instrução gerada, fato ilustrado na Figura 4.6. Assim como a figura anterior, as setas vermelhas são utilizadas para indicar a procedência das variáveis `$i` empregadas na composição da instrução `Store`.

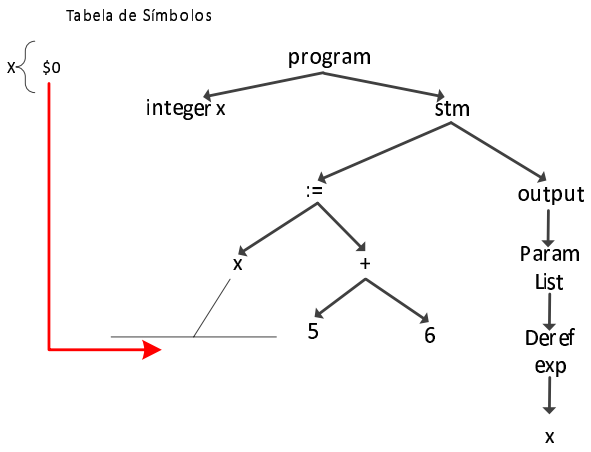


Figura 4.3: AST mostrando a busca na tabela de símbolos.

A Figura 4.7 apresenta a árvore de sintaxe abstrata, enfatizando a geração da instrução `Load`, que carrega o valor de um endereço de memória para a variável `$2`. A variável `$0` apontada pela seta vermelha tem procedência do nodo "`x`", que a buscou na tabela de símbolos. A Figura 4.8 ilustra a criação da instrução `ArgList`. Essa instrução

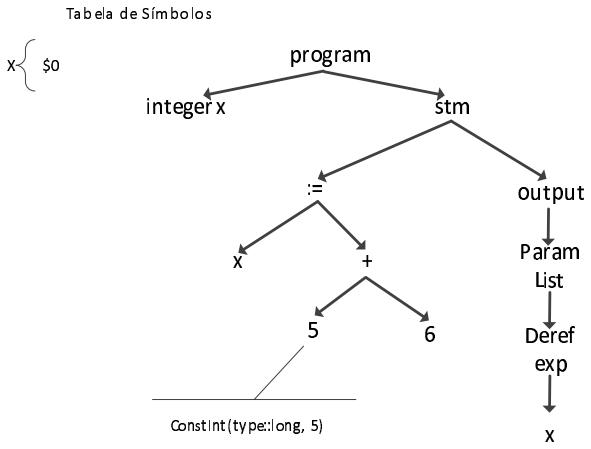


Figura 4.4: AST mostrando a geração da instrução de carga de constante.

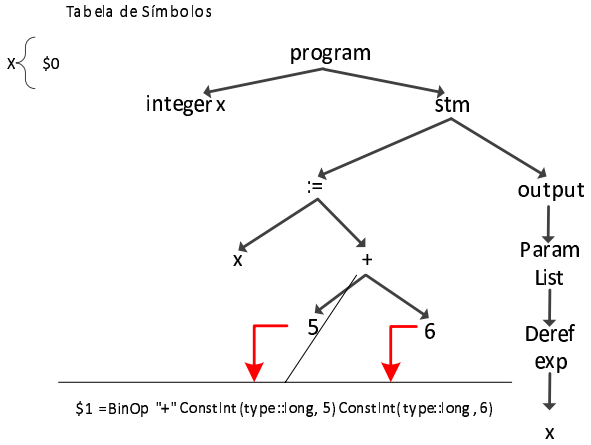


Figura 4.5: AST mostrando a geração da instrução de operação binária.

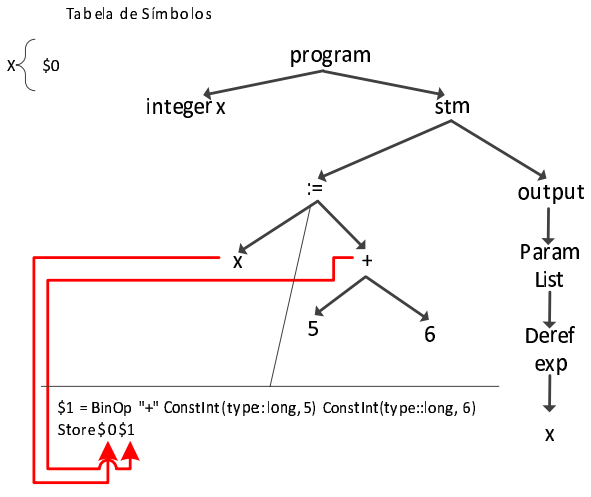


Figura 4.6: AST mostrando a geração da instrução de armazenamento.

é usada na construção da lista de argumentos de uma função, permitindo, nesse caso, a passagem do valor de uma variável. Como mostrado na Figura 4.9, o nodo "`output`" realiza a geração de uma instrução `Call` com a chamada da função *printf* da biblioteca da Linguagem C. A seta vermelha indica a procedência dos argumentos utilizados pela chamada.

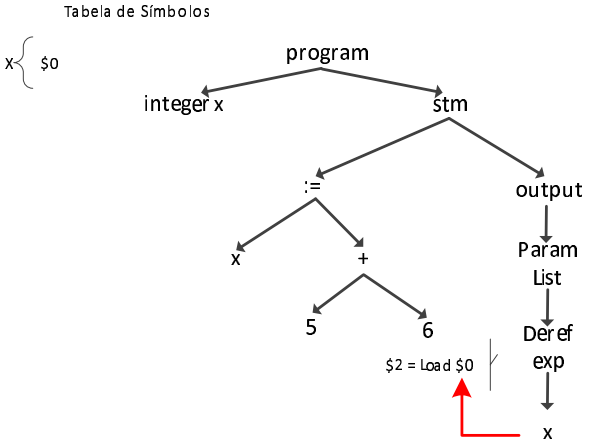


Figura 4.7: AST mostrando a carga de uma variável.

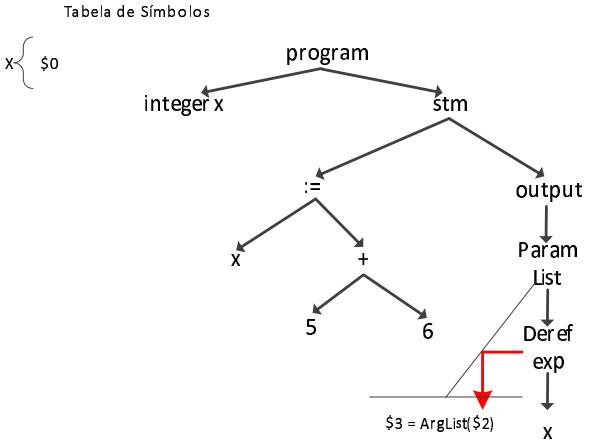


Figura 4.8: AST mostrando a geração da instrução de argumento de função.

A Figura 4.10 ilustra como ocorre, no nodo "`stm`", o sequenciamento de instruções. O nodo "`:=`" e o nodo "`output`" retornam, cada um, uma lista de instruções geradas que são concatenadas. Ao retornar para o nodo "`program`", ocorrerá a concatenação das listas de instruções do nodo "`integer x`" com a lista de instruções do nodo "`stm`". Essa concatenação final terá como resultado a lista de instruções descrita na Listagem 4.4.

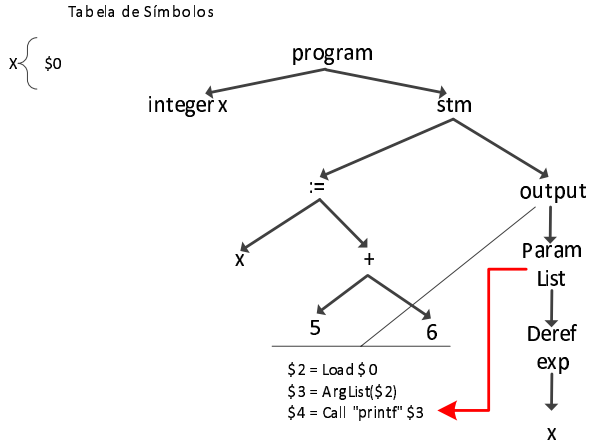


Figura 4.9: AST mostrando a geração da instrução de chamada de função.

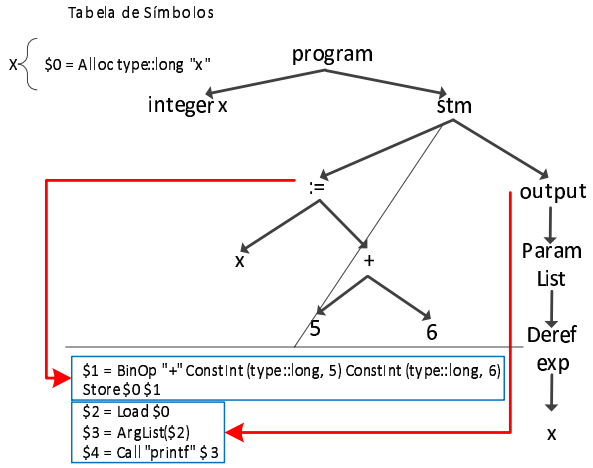


Figura 4.10: AST mostrando o sequenciamento de instruções.

## 4.3 Expressões

Segundo Watt & Findlay [2004], expressões são construções de um programa que, ao serem avaliadas, produzem um valor. Dessa forma, toda carga de valor, ou seja, toda instrução que carrega uma variável da memória ou nela armazena um valor é uma expressão, bem como a aplicação de operações binárias, unárias e a condicional. A chamada de função também tem seu componente na categoria de expressão.

As classes utilizadas pelos componentes que geram código de expressões são:

- `Expression_Attr` - classe dos atributos das expressões cujos objetos contêm, além da lista de código que é comum aos objetos de todas classes de atributos, o endereço da variável produzida pela expressão e um apontador para um objeto do tipo da expressão, o que permite realizar verificações internas de integridade do código.

- `Dimension_Attr` - classe dos atributos de dimensões de arranjos cujos objetos contêm, além da lista de código que é comum a todas classes de atributos, uma lista de dimensões para serem utilizadas na declaração de arranjos estáticos.

- `BinOp` - estrutura que contém uma representação de uma operação binária e o tipo dos seus operandos;

- `UnOp` - estrutura que contém uma representação de uma operação unária e o tipo de seu operando;

- `Type` - classe cujos objetos armazenam informações do tipo da variável. Possui métodos estáticos para facilitar a obtenção de uma instância da classe para as ações necessárias como criação de constantes ou verificação de tipos.

Também são utilizadas classes específicas para variáveis como arranjos, estruturas e classes. Os objetos de cada uma dessas classes armazenam as informações do tipo da variável e o endereço de sua alocação na memória de forma a permitir seu uso.

### 4.3.1 Valores Literais

Na infraestrutura de componente, são permitidos dois tipos de literais: valores do tipo inteiro e de ponto flutuante. A largura dos valores podem variar de acordo com a necessidade do implementador.

O componente `LoadInteger` (Listagem 4.6) gera o código para a constante de um literal inteiro. O primeiro parâmetro passado é o valor do literal e o segundo, seu tipo, que indicará a largura necessária para seu armazenamento. Esse componente gera a instrução de constante inteira e a armazena no atributo a ser retornado pelo componente.

```
Expression_Attr* LoadInteger(int64_t val, Type* tp);
```

Listagem 4.6: Componente para um literal inteiro.

O componente `LoadFloatPoint` (Listagem 4.7) gera o código para a instrução de um literal de ponto flutuante, funcionando, portanto, de forma idêntica a seu correlato para literal inteiro. O primeiro parâmetro passado é o valor, seguido pelo objeto da classe `Type` para indicar a largura do literal requerido. Com essa informação, gera-se uma instrução de constante de ponto flutuante que é armazenada no atributo que o componente retorna.

```
Expression_Attr* LoadFloatPoint(double val, Type tp);
```

Listagem 4.7: Componente para um literal de ponto flutuante.

Na Listagem 4.8, se encontra um exemplo da aplicação dos componentes de valores literais. É utilizada uma descrição de expressão na linguagem *Back* para esse exemplo. Na linha 2, é empregado o componente `LoadInteger` para gerar o valor literal, passando o literal como primeiro parâmetro e um objeto da classe `Type` da infraestrutura como segundo parâmetro. O atributo retornado pelo componente contém a instrução de literal e é utilizado para preencher o atributo do nodo "`exp`" na árvore de sintaxe abstrata.

```
1 exp ::= num {
2     exp->exp_attr = cb->LoadInteger(num, component::Type::GetInt());
3   };
```

Listagem 4.8: Exemplo da utilização do componente de literal.

### 4.3.2 Carga de Variáveis

Outro quesito importante para a compilação de uma linguagem é a instanciação de variáveis e seus acessos. Como é necessário realizar o cálculo do endereço de memória em que a variável está armazenada antes de realizar a carga, é possível dividir a carga de variáveis em duas etapas: o cálculo do endereço e a carga do valor apontado por um determinado endereço.

O cálculo do endereço de uma variável depende de seu tipo, podendo ser necessário somente a realização de uma busca na tabela de símbolos e, portanto, nesse caso só precisa do identificador da variável ou, caso a variável seja um arranjo, estrutura ou classe é necessário de informações extras como o índice ou identificador do membro que está sendo acessado.

Outro caso em que há a necessidade de se realizar o cálculo do endereço de variável é no componente utilizado pelo comando de atribuição, que armazena o valor calculado por uma expressão no endereço de memória alocado para uma variável.

O componente `LoadAddress` (Listagem 4.9) gera o código para a carga de uma variável. O único parâmetro passado é o atributo contendo o endereço calculado da variável. O componente então gera uma instrução de carga e a retorna como atributo do componente.

O componente `GetVariableAddress` (Listagem 4.10) recupera, da tabela de símbolos, o endereço de uma variável. O único parâmetro que deve ser passado para

```
Expression_Attr* LoadAddress(Expression_Attr* ptr);
```

Listagem 4.9: Componente para a carga de um endereço calculado.

esse componente é o identificador da variável que é buscada. O endereço calculado é armazenado como atributo para retornar do componente.

```
Expression_Attr* GetVariableAddress(std::string id);
```

Listagem 4.10: Componente para a recuperação do endereço de uma variável.

Na Listagem 4.11, está um exemplo da aplicação dos componentes que realizam a recuperação do endereço e carga do valor de uma variável. É utilizada uma descrição na linguagem *Back* com duas produções, uma para o identificador da variável e outra para uma expressão em que a variável será utilizada. Na linha 2, é empregado o componente `GetVariableAddress` para recuperar o endereço da variável, sendo usado o identificador da variável que será recuperada como parâmetro. Seu atributo retornado contém a instrução com o endereço da variável buscada e é utilizado para preencher o atributo de expressão do nodo "`access_exp`". Na linha 5, é aplicado o componente `LoadAddress` para gerar o código de carga do endereço recuperado anteriormente, utilizando como parâmetro o atributo de expressão da variável de produção `access_exp`. De modo similar ao outro componente, o atributo retornado é utilizado para preencher o atributo da expressão do nodo "`exp`", entretanto seu atributo retornado contém a instrução de carga do endereço de memória requerido.

```
1 access_exp ::= id{
2     access_exp->exp_attr = cb->GetVariableAddress(id);
3   };
4 exp ::= "deref" access_exp{
5     exp->exp_attr = cb->LoadAddress(access_exp->exp_attr);
6   };
```

Listagem 4.11: Exemplo da utilização dos componentes para recuperação e carga de uma variável.

O componente `ArrayBaseAddressCalc` (Listagem 4.12) gera a instrução para calcular o endereço do elemento considerado o primeiro índice de um arranjo. Diferentemente do componente anterior, em que é passado o identificador e o próprio componente é empregado para realizar a busca na tabela de símbolos, nesse outro componente é necessária a recuperação do arranjo com o método denominado `GetArray`, que por sua vez recebe o identificador do arranjo e retorna o objeto da classe `Array` necessário. O segundo parâmetro é o atributo contendo o valor do índice e o terceiro é uma variável

booleana que indica se é necessário realizar verificação dos limites durante o cálculo. O endereço calculado e o do componente anterior são armazenados como atributo e retornados.

```
Expression_Attr* ArrayBaseAddressCalc(Array* arr,
                                      Expression_Attr* ExpAddr,
                                      bool BoundCheck);
```

Listagem 4.12: Componente para cálculo do endereço base de uma posição de um arranjo.

O componente `ArrayAddressCalc` (Listagem 4.13) complementa o anterior, realizando o cálculo do endereço do arranjo considerando os índices subsequentes. Dessa forma seus parâmetros são similares: a instância do arranjo é passada como primeiro parâmetro, o índice é avaliado como segundo e o booleano de verificação de limite é estabelecido como último parâmetro. O parâmetro `TempAddr` deve receber o atributo contendo o valor produzido para o endereço do índice anteriormente calculado. O endereço calculado é armazenado no atributo que é retornado do componente.

```
Expression_Attr* ArrayAddressCalc(Array* arr,
                                  Expression_Attr* ExpAddr,
                                  Expression_Attr* TempAddr,
                                  bool BoundCheck);
```

Listagem 4.13: Componente para cálculo do endereço de uma posição de um arranjo.

Na Listagem 4.14, está um exemplo da aplicação dos componentes de cálculo de índice de um arranjo. É utilizada a linguagem *Back* para descrever uma produção de *AST* com duas alternativas, cujas ações semânticas calculam o índice do arranjo requerido. Na linha 2, é empregado o método `GetArray` para recuperar as informações do arranjo armazenadas na tabela de símbolos, sendo utilizado o identificador do arranjo como parâmetro do método. O retorno desse método é armazenado em um atributo do nodo "`array_exp`" para ser utilizado no cálculo dos índices subsequentes do arranjo. Na linha 3, é aplicado o componente `ArrayBaseAddressCalc` que gera as instruções de cálculo do primeiro índice do arranjo, sendo usado como primeiro parâmetro o atributo contendo as informações do arranjo, seguido do atributo de expressão da variável de produção `exp` que contém a instrução de valor do índice. O atributo retornado pelo componente contém a instrução de cálculo do índice gerada pelo componente e é utilizado para preencher o atributo do nodo "`array_exp`". A produção seguinte é referente ao cálculo dos índices subsequentes, em que na linha 6 o atributo contendo as informações do arranjo do nodo "`array_exp1`" é armazenado no atributo do nodo

"`array_exp`", permitindo a continuação do cálculo dos índices do arranjo. Na linha 7, o componente `ArrayAddressCalc` é usado para gerar o código do índice do arranjo, o primeiro parâmetro passado é o atributo contendo as informações do arranjo. O segundo parâmetro aplicado para o componente é, como no outro componente utilizado, o atributo de expressão da variável de produção `exp` que contém o endereço da instrução do índice sendo calculado. O terceiro parâmetro empregado para o componente é o atributo de expressão da variável de produção `array_exp1` que contém a instrução de valor do deslocamento do arranjo. De forma semelhante a que ocorre com o retorno do outro componente utilizado, o retorno é usado para preencher o atributo de expressão do nodo "`array_exp`".

```
1 array_exp ::= "arrayexp" id exp{
2     array_exp->array = cb->GetArray(id);
3     array_exp->exp_attr = cb->ArrayBaseAddressCalc(array_exp->array,
          exp->exp_attr);
4   }
5   | array_exp1 exp{
6     array_exp->array = array_exp1->array;
7     array_exp->exp_attr = cb->ArrayAddressCalc(array_exp->array,
          exp->exp_attr, array_exp1->exp_attr);
8   };
```

Listagem 4.14: Exemplo da utilização dos componentes para cálculo de índice de arranjo.

O componente `RecordBaseAddressCalc` (Listagem 4.15) gera a instrução para calcular o endereço de uma variável interna de uma estrutura. Assim como o componente para cálculo do endereço de arranjo, é necessário o método `GetRecord` que recebe o identificador da estrutura e retorna um objeto da classe `Record` contendo as informações necessárias para realizar esse cálculo. Diferentemente do componente de arranjo, o único outro parâmetro necessário para realizar esse cálculo é o identificador da variável armazenada na estrutura. Assim como em todos os outros componentes, o endereço calculado é armazenado no atributo retornado para uso subsequente.

```
Expression_Attr* RecordBaseAddressCalc(Record* rec, std::string id);
```

Listagem 4.15: Componente para cálculo do endereço base de uma variável de uma base.

O componente `RecordAddressCalc` (Listagem 4.16) gera a instrução para calcular o endereço de campos internos de estruturas, tal como os dois componentes para arranjo. No parâmetro `TempAddr`, deve ser passado o endereço calculado para a variável buscada anteriormente. E, assim como no caso dos arranjos, o componente anterior

deve ser o primeiro utilizado para se recuperar o endereço de uma variável. Da mesma forma que outros componentes citados anteriormente, o endereço calculado é armazenado no atributo e retornado.

```
Expression_Attr* RecordAddressCalc(Record* rec,
                                   std::string id,
                                   Expression_Attr* TempAddr);
```

Listagem 4.16: Componente para cálculo do endereço de uma variável de uma estrutura.

Na Listagem 4.17, está um exemplo da aplicação dos componentes de cálculo do endereço de variáveis de uma estrutura. É utilizada a linguagem *Back* para descrever uma produção de *AST* com duas alternativas, cujas ações semânticas calculam o endereço das variáveis da estrutura. Na linha 2, é aplicado o método `GetRecord` para recuperar as informações da estrutura armazenadas na tabela de símbolos, sendo utilizado o identificador da estrutura como parâmetro do método. O retorno desse método é armazenado em um atributo do nodo "`record_exp`" para ser utilizado no cálculo do endereço das variáveis da estrutura. Na linha 3, é empregado o componente `RecordBaseAddressCalc`, que gera a instrução de cálculo do endereço de uma variável de uma estrutura. O primeiro parâmetro usado é o atributo contendo as informações da estrutura e o segundo parâmetro é o identificador da variável buscada. O atributo retornado por esse componente contém a instrução gerada pelo componente e é utilizado para preencher o atributo de expressão do nodo "`record_exp`". A produção seguinte se refere ao cálculo do endereço dos campos internos da estrutura, sendo que na linha 6 o atributo contendo as informações da estrutura do nodo "`record_exp1`" são utilizadas para preencher as informações da estrutura do nodo "`record_exp`". Na linha 7, é aplicado componente `RecordAddressCalc` para gerar o código do cálculo do endereço do campo interno da estrutura, sendo utilizado o atributo contendo as informações da estrutura como primeiro parâmetro. O segundo parâmetro empregado é o identificador do campo sendo buscado, e o terceiro parâmetro é o atributo de expressão da variável de produção `record_exp1` que contém a instrução de endereço do cálculo dos campos da estrutura. De forma similar à que ocorre com o retorno do outro componente utilizado, o retorno desse componente é usado para preencher o atributo de expressão do nodo "`record_exp`".

O componente `ClassBaseAddressCalc` (Listagem 4.18) funciona de maneira similar ao `RecordBaseAddressCalc` (Listagem 4.15), exceto ao realizar a busca dos membros de uma classe, pois pode, dessa maneira, encontrar apontadores para variáveis ou para funções que também são membros da classe. Para utilizar esse componente, é necessário o componente `GetClass` que recebe o identificador da classe e retorna um

```
1 record_exp ::= "recordexp" id1 id2{
2     record_exp->record = cb->GetRecord(id1);
3     record_exp->exp_attr = cb->RecordBaseAddressCalc(record_exp->record,
          id2);
4   }
5   | record_exp1 id{
6     record_exp->record = record_exp1->record;
7     record_exp->exp_attr = cb->RecordAddressCalc(record_exp->record, id,
          record_exp1->exp_attr);
8   };
```

Listagem 4.17: Exemplo da utilização dos componentes para cálculo membros de estruturas.

objeto da classe `Class` contendo as informações referentes aos membros da classe. O segundo parâmetro passado ao componente é o nome do membro cujo endereço será calculado. O resultado é armazenado no atributo e retornado para uso subsequente.

```
Expression_Attr* ClassBaseAddressCalc(Class* cls, std::string id);
```

Listagem 4.18: Componente para cálculo do endereço base de uma variável de uma classe.

O componente `ClassAddressCalc` (Listagem 4.19), bem como o anterior, funciona de maneira similar ao `RecordAddressCalc` (Listagem 4.16), mas necessita do parâmetro `TempAddr` que contém o endereço do membro anterior ao que é calculado nesse componente.

```
Expression_Attr* ClassAddressCalc(Class* cls,
                                  std::string id,
                                  Expression_Attr* TempAddr);
```

Listagem 4.19: Componente para cálculo do endereço de uma variável de uma classe.

Na Listagem 4.20, está um exemplo da aplicação dos componentes de cálculo do endereço de membros de uma classe. É utilizada a linguagem *Back* para descrever uma produção de *AST* com duas alternativas, cujas ações semânticas calculam o endereço dos membros da classe. Na linha 2, é aplicado o método `GetClass` para recuperar as informações da classe armazenadas na tabela de símbolos, usando o identificador da classe como parâmetro do método. O retorno desse método é armazenado em um atributo do nodo "`class_exp`" para ser utilizado no cálculo do endereço dos membros de uma classe. Na linha 3, é empregado o componente `ClassBaseAddressCalc` que gera a instrução de cálculo do endereço de um membro de uma classe. O primeiro parâmetro usado é o atributo contendo as informações da classe e o segundo parâmetro é o identificador do membro buscado. O atributo retornado por esse componente

contém a instrução gerada pelo componente e é utilizado para preencher o atributo de expressão do nodo "`class_exp`". A produção seguinte se refere ao cálculo do endereço dos membros internos da classe, sendo que na linha 6 o atributo contendo as informações da classe do nodo "`class_exp1`" é utilizado para preencher o atributo da classe do nodo "`class_exp`". Na linha 7, é aplicado componente `ClassAddressCalc` para gerar o código do cálculo do endereço do membro interno da classe, sendo utilizado o atributo contendo as informações da classe como primeiro parâmetro. O segundo parâmetro empregado é o identificador do membro sendo buscado e o terceiro parâmetro é o atributo de expressão da variável de produção `class_exp1`, que contém a instrução de endereço do cálculo dos membros da classe. De forma similar a que ocorre com o retorno do outro componente utilizado, o retorno desse componente é usado para preencher o atributo de expressão do nodo "`class_exp`".

```
1  class_exp ::= "classexp" id1 id2{
2      class_exp->clss = cb->GetClass(id1);
3      class_exp->exp_attr = cb->ClassBaseAddressCalc(class_exp->clss, id2);
4    }
5    | class_exp1 id{
6      class_exp->clss = class_exp1->clss;
7      class_exp->exp_attr = cb->ClassAddressCalc(class_exp->clss, id,
           class_exp1->exp_attr);
8    };
```

Listagem 4.20: Exemplo da utilização dos componentes para cálculo membros de classes.

Os componentes acima permitem tanto realizar a carga de um valor apontado por um endereço quanto o cálculo dele: seja a variável simples, um arranjo, uma estrutura ou classe. É possível perceber a simplicidade desses componentes, o que os torna bastante úteis para a implementação de um compilador, pois permitem encapsular detalhes maiores do cálculo de endereçamento das variáveis, possibilitando que o projetista do compilador foque em outros detalhes da linguagem de programação.

### 4.3.3 Operações Binárias e Unárias

A infraestrutura de geração de código provê componentes para a geração de código para operações binárias de adição, subtração, multiplicação, divisão, resto da divisão, *shift* para esquerda, *shift* lógico para direita, *shift* aritmético para direita, as comparações de igualdade, inequalidade, maior que, menor que, maior ou igual, menor ou igual e as operações lógicas `E` e `OU`. As operações unárias implementadas são a negação lógica, a negação aritmética e as coerções de inteiro e ponto flutuante.

As operações são realizadas em duas etapas, sendo necessário primeiro obter o operador e em seguida aplicá-lo na expressão.

O método `GetUnOpe` (Listagem 4.21) serve para recuperar um operador unário que é utilizado em uma operação. O primeiro parâmetro passado é o identificador da operação requerida, obedecendo as operações implementadas na infraestrutura. O segundo parâmetro é o tipo do operador que deve ser utilizado, uma vez que existem instruções LLVM distintas para valores inteiros e valores de ponto flutuante. O que é retornado por esse método é uma estrutura contendo as informações do operando e seu tipo, ambos necessários para se realizar a operação.

```
UnOp GetUnOpe(std::string op, Type* tp);
```

Listagem 4.21: Método para criar a operação unária.

O componente `Apply` (Listagem 4.22) gera a instrução para aplicar a operação unária necessária a uma expressão. O primeiro parâmetro é o operador unário e o segundo é o atributo da expressão à qual o operador será aplicado. O componente gera uma instrução com essa operação e armazena-a no atributo retornado.

```
Expression_Attr* Apply(UnOp op, Expression_Attr* v);
```

Listagem 4.22: Componente para aplicação de operação únaria.

Na Listagem 4.23, está um exemplo da aplicação dos componentes da operação unária. É utilizada a linguagem *Back* para descrever uma produção de expressão de *AST*, cuja ação semântica é a aplicação da operação unária do oposto de um literal. Na linha 2, é empregado o componente `Apply` que gera o código da instrução necessária para aplicar a operação unária. Como primeiro parâmetro do componente é aplicado o método `GetUnOpe`, que recupera o operador para ser empregado pelo componente. O segundo parâmetro do componente é o atributo de expressão do nodo "`exp1`", que contém instrução que será utilizado como operando da operação. O primeiro parâmetro do método é uma cadeia de caracteres indicando a operação a ser aplicada, nesse caso a subtração. O segundo parâmetro do método é o atributo do nodo "`exp1`" contendo o tipo da expressão que é calculada, permitindo a recuperação do operador correto. O atributo retornado pelo componente empregado nessa produção contém a instrução gerada pelo componente e é utilizado para preencher o atributo de expressão do nodo "`exp`".

O método `GetBinOpe` (Listagem 4.24) serve para recuperar um operador binário que é utilizado na aplicação de uma operação. Assim sendo, ele funciona de igual

```
1 exp ::= "unary_minus" exp1 {
2     exp->exp_attr = cb->Apply(component::GetUnOpe("-", exp1->type),
          exp1->exp_attr);
3   }
```

Listagem 4.23: Exemplo da utilização dos componentes para aplicação de operação unária.

modo à sua contraparte unária, necessitando do identificador da operação e do tipo para retornar uma estrutura com as informações.

```
BinOp GetBinOpe(std::string op, Type* tp);
```

Listagem 4.24: Componente para criar a operação binária.

O componente `Apply` (Listagem 4.25) gera a instrução que aplica uma operação binária a dois atributos que contenham endereços das instruções necessárias à sua operação. Assim como a aplicação da operação unária, a operação binária necessita o operador fornecido pelo componente anterior. Entretanto, por ser uma operação binária, precisa de mais dois parâmetros contendo os atributos das expressões às quais a operação será aplicada. Igualmente à sua equivalente, o componente gera uma instrução com a operação requerida e armazena-a no atributo retornado para ser utilizado por outros componentes.

```
Expression_Attr* Apply(BinOp op, Expression_Attr* v1, Expression_Attr* v2);
```

Listagem 4.25: Componente para aplicação de operação binária.

Na Listagem 4.26, está um exemplo da aplicação dos componentes da operação binária. É utilizada a linguagem *Back* para descrever uma produção de expressão de *AST*, cuja ação semântica é a aplicação da operação binária da soma de dois valores. Na linha 2, é empregado o componente `Apply` que gera o código da instrução necessária para aplicar a operação binária. Como primeiro parâmetro do componente é aplicado o método `GetBinOpe`, que recupera o operador para ser empregado pelo componente. O segundo parâmetro do componente é o atributo de expressão do nodo "`exp1`", que contém a instrução que é utilizada como primeiro operando da operação binária, e o terceiro parâmetro é o atributo de expressão do nodo "`exp2`", que contém a instrução que é utilizada como segundo operando da operação binária. O primeiro parâmetro do método é uma cadeia de caracteres indicando a operação a ser aplicada, nesse caso a soma. O segundo parâmetro do método é o atributo do nodo "`exp1`" contendo o tipo da expressão que é calculada, permitindo a recuperação do operador correto. O atributo

retornado pelo componente empregado nessa produção contém a instrução gerada pelo componente e é utilizado para preencher o atributo de expressão do nodo "`exp`".

```
1 exp ::= "add" exp1 exp2{
2     exp->exp_attr = cb->Apply(component::GetBinOpe("+", exp1->type),
          exp1->exp_attr, exp2->exp_attr);
3   };
```

Listagem 4.26: Exemplo da utilização dos componentes para aplicação de operação binária.

Os componentes e métodos desta seção permitem a geração de código para operações binárias e unárias. Os dois métodos descritos possuem a finalidade de obter a estrutura do operador e, assim, obter as informações necessárias para que a operação seja realizada enquanto os componentes geram as instruções que realizam a aplicação desses operadores. Dessa forma, é possível perceber a dependência e complementaridade desses componentes.

### 4.3.4 Chamada de Função

Para utilizar argumentos de chamada de função são necessários componentes específicos que armazenam a lista de código dos argumentos para serem gerados, com suas devidas cargas e cálculos, durante a geração da chamada de função.

`CallFunction` (Listagem 4.27) é o componente que gera o código responsável pela chamada de uma função. Assim como os componentes que realizam os cálculos de endereço de arranjo, estrutura e classe, é necessário o método `GetFunction` que recebe o identificador da função e retorna as suas informações. O segundo parâmetro utiliza um objeto da classe `Expression_Attr` que contém a lista de código gerada para computar os argumentos encontrados previamente. Dessa maneira, é gerada uma instrução cujos operandos são a função e a lista de códigos gerados para os argumentos que, durante a etapa de geração LLVM, será percorrida para gerar as instruções de carga e cálculos necessários para a chamada da função requerida. O componente retorna um atributo contendo a instrução gerada.

No caso da função não possuir parâmetros, deve-se omitir o segundo parâmetro, que se refere aos argumentos.

```
Expression_Attr* CallFunction(Function* fun, Expression_Attr* args);
```

Listagem 4.27: Componente para geração de chamada de função.

Na Listagem 4.28, está um exemplo da utilização do componente para a chamada de função. É utilizada a linguagem *Back* para descrever uma produção de expressão da *AST* com duas alternativas, cujas ações semânticas são referentes a geração do código para a chamada da função. Na linha 2, é empregado o componente `CallFunction` que gera o código da instrução para a chamada da função. Como primeiro parâmetro é aplicado o método `GetFunction` para recuperar as informações da função armazenadas na tabela de símbolos. O parâmetro desse método é o identificador da função buscada. O segundo parâmetro do componente empregado é o atributo de expressão do nodo "`actualarg`", que contém a lista de instruções geradas para os argumentos da função. O atributo retornado pelo componente contém a instrução gerada para a chamada de função e é utilizado para preencher o atributo de expressão do nodo "`exp`". A segunda produção do exemplo é semelhante a primeira, diferindo somente na ausência de argumentos para a chamada da expressão. Portanto, na linha 5 é empregado o componente `CallFunction` de maneira similar ao anterior, entretanto, é aplicado unicamente o primeiro parâmetro com o método `GetFunction` para recuperar as informações da função. Assim como a produção anterior, o atributo resultante do componente contém a instrução gerada para a chamada de função e é utilizado para preencher o atributo de expressão do nodo "`exp`".

```
1 exp ::= "funccall" id actualarg{
2     exp->exp_attr = cb->CallFunction(cb->GetFunction(id),
          actualarg->exp_attr);
3   }
4   | "funccall" id{
5     exp->exp_attr = cb->CallFunction(cb->GetFunction(id));
6   };
```

Listagem 4.28: Exemplo da utilização do componente de chamada de função.

O componente `DerefArgument` (Listagem 4.29) gera a instrução necessária para passar por valor um argumento a uma função. Dessa forma, ele deve receber a expressão com o valor calculado. A instrução gerada é retornada como atributo para ser utilizada pelo componente `CallFunction` (Listagem 4.27).

```
Expression_Attr* DerefArgument(Expression_Attr* val);
```

Listagem 4.29: Componente para geração do argumento por valor da chamada de função.

`RefArgument` (Listagem 4.30) é o componente que gera a instrução necessária para o uso de um argumento de uma função cuja passagem seja por referência ou por valor-resultado. O parâmetro desse componente é o identificador da variável que

terá seu endereço buscado na tabela de símbolos e utilizado na chamada de função. Como o anterior, a instrução gerada é retornada como atributo para uso subsequente do componente `CallFunction` (Listagem 4.27).

```
Expression_Attr* RefArgument(std::string id);
```

Listagem 4.30: Componente para geração do argumento por referência da chamada de função.

O componente `NameArgument` (Listagem 4.31) gera a instrução necessária para passagem de um argumento por nome a uma função. Seu único parâmetro é o identificador da variável utilizada como argumento. Além da instrução do argumento, é gerado por esse componente a função *thunk*, necessária para realizar a passagem por nome na arquitetura do bytecode LLVM.

```
Expression_Attr* NameArgument(std::string id);
```

Listagem 4.31: Componente para geração do argumento por nome da chamada de função.

Na Listagem 4.32, está um exemplo da utilização dos componentes para geração de instruções referentes a passagem de argumentos para uma função. É utilizada a linguagem *Back* para descrever uma produção da *AST* com quatro alternativas, a primeira responsável pela construção da lista de argumentos e as restantes responsáveis pela geração de passagens diferentes de argumento. A primeira alternativa, cuja ação semântica é a construção da lista de argumentos utilizada pela chamada de função, aplica o componente `BuildSequence` para concatenar as listas de instruções dos argumentos. O primeiro parâmetro empregado é o atributo de expressão do nodo "`actualarg1`" e o segundo parâmetro o atributo de expressão do nodo "`actualarg2`". O retorno desse componente contém as duas listas de instruções passadas como parâmetros concatenadas e é utilizado para preencher o atributo de expressão do nodo "`actualarg`". A primeira alternativa da segunda produção é empregada para gerar a instrução relativa a passagem do argumento por valor, sendo aplicado o componente `DerefArgument` na linha 5 para esse fim. O parâmetro usado por esse componente é o atributo de expressão que contém a instrução a ser passada por valor. O retorno do componente contém a instrução gerada para a passagem do argumento e é utilizado para preencher o atributo de expressão do nodo "`actualarg`".

A terceira alternativa da produção é empregada para gerar a instrução relativa a passagem do argumento por referência, sendo aplicado o componente `RefArgument`,

na linha 8. O parâmetro usado por esse componente é o identificador armazenado no atributo do nodo "`access_exp`", o que permite gerar a instrução de acesso por referência da variável utilizada. O retorno do componente contém a instrução gerada para a passagem do argumento e é utilizado para preencher o atributo de expressão do nodo "`actualarg`". A última alternativa da produção é empregada para gerar a instrução relativa a passagem do argumento por nome, sendo aplicado o componente `NameArgument` na linha 11, para esse efeito. Como realizado no componente anterior, o parâmetro usado por esse componente é o identificador armazenado no atributo do nodo "`access_exp`", o que permite gerar a instrução de acesso e, nesse caso, a função *thunk* necessária para realizar a passagem do argumento por nome. O retorno do componente contém, além da instrução gerada para a passagem do argumento, a função *thunk* gerada e é utilizado para preencher o atributo de expressão do nodo "`actualarg`".

```
1  actualarg  ::= actualarg1 actualarg2{
2      actualarg->exp_attr = cb->BuildSequence(actualarg1->exp_attr,
           actualarg2->exp_attr);
3    }
4    | "deref_call" exp{
5      actualarg->exp_attr = cb->DerefArgument(exp->exp_attr);
6    }
7    | "ref_call" access_exp{
8      actualarg->exp_attr = cb->RefArgument(access_exp->_id);
9    }
10   | "name_call" access_exp{
11     actualarg->exp_attr = cb->NameArgument(access_exp->_id);
12   };
```

Listagem 4.32: Exemplo da utilização dos componentes de passagem de argumento.

Com esses componentes, é possível gerar código para a chamada de uma função a partir de seu identificador, sem preocupar com qualquer tratamento diferenciado para os argumentos devido aos diversos modos de passagem de parâmetro. Todo o cálculo de endereço e operação estão encapsulados em outros componentes, tornando sua utilização mais simples.

### 4.3.5 Condicional

Expressões condicionais também são conhecidas como operadores ternários, uma vez que recebem três operandos para produzir um valor. Esse tipo de expressão possui uma condição e duas alternativas de valores: caso seja verdadeira a condição, o primeiro valor é utilizado; caso contrário, o segundo valor será a opção escolhida.

O componente `BuildConditionalExpression` (Listagem 4.33) constrói a instrução condicional para a expressão passada. Seu primeiro parâmetro deve ser a condição

a ser avaliada para decidir qual valor utilizar, e os dois parâmetros subsequentes são os valores a escolher. A instrução construída é armazenada e retornada como nos componentes de expressão anteriores.

```
Expression_Attr* BuildConditionalExpression(Expression_Attr* cond,
                                            Expression_Attr* exp1,
                                            Expression_Attr* exp2);
```

Listagem 4.33: Componente para geração de expressão condicional.

Na Listagem 4.34, está um exemplo da utilização do componente para geração da instrução de expressão condicional. É utilizada a linguagem *Back* para descrever uma produção de expressão da *AST*, cuja ação semântica é a geração da expressão condicional. Na linha 2, é aplicado o componente `BuildConditionalExpression` que gera o código da instrução condicional. Como primeiro parâmetro, é usado o atributo de expressão do nodo "`exp1`", que contém a instrução que é avaliada para decidir o resultado da expressão condicional. O segundo parâmetro empregado é o atributo de expressão do nodo "`exp2`" que contém o endereço da instrução que é utilizada na avaliação verdadeira da expressão condicional, sendo o terceiro parâmetro o atributo de expressão do nodo "`exp3`" que contém o endereço da instrução que é usada caso a avaliação da expressão condicional seja falsa. O retorno do componente contém a instrução de condição gerada e é utilizado para preencher o atributo de expressão do nodo "`exp`".

```
1 exp ::= "ifexp" exp1 exp2 exp3{
2     exp->exp_attr = cb->BuildConditionalExpression(exp1->exp_attr,
          exp2->exp_attr, exp3->exp_attr);
3   };
```

Listagem 4.34: Exemplo da utilização dos componentes expressão condicional.

A simplificação realizada por esse componente deriva do encapsulamento dos detalhes das ações semânticas que gerenciam uma lista de desvios e condições. Isso torna a implementação de expressões condicionais mais fácil.

### 4.3.6 Alocação Dinâmica

A alocação dinâmica, apesar de não produzir um valor aplicável por outras expressões, é uma expressão que produz como valor o endereço de uma área de memória. Assim, é concedido a área de memória a uma variável durante a execução de um programa. Por conseguinte, esse componente é útil para linguagens que possuam variáveis cuja alocação seja realizada dinamicamente durante a execução do programa.

O componente `AllocDynamic`(Listagem 4.35) é responsável por realizar a geração da instrução de alocação dinâmica. Nesse caso, o primeiro parâmetro é o tipo da área que será alocada, e o segundo parâmetro, as dimensões da área. Esses últimos são requisitos para a alocação de um arranjo em que é necessário seu tipo e tamanho. Com essas informações, é construída uma instrução de alocação que é armazenada no atributo de retorno.

Para a alocação de uma estrutura ou classe, somente seu tipo é necessário. Portanto, é possível omitir o segundo parâmetro desse componente para a alocação desses tipos de dados.

```
Expression_Attr* AllocDynamic(Type* t, Dimension_Attr* size);
```

Listagem 4.35: Componente para geração de alocação dinâmica de uma variável.

Na Listagem 4.36, está um exemplo da utilização do componente de alocação dinâmica. É utilizada a linguagem *Back* para descrever uma produção de *AST* com duas alternativas, uma responsável pela alocação dinâmica da área de memória para um arranjo e outra responsável pela alocação dinâmica da área de memória de uma estrutura. A primeira alternativa, na linha 2, emprega o componente `AllocDynamic` para gerar a instrução de alocação da área de memória de um arranjo. O primeiro parâmetro é o atributo do nodo "`type_name`" usado para conseguir o tipo do valor que será armazenado na área de memória alocada. O segundo parâmetro é o atributo de dimensão do nodo "`dim_exprs`", aplicado para saber o tamanho da área de memória alocada. O retorno do componente contém a instrução de alocação da área de memória e é utilizado para preencher o atributo de expressão do nodo "`exp`". Na linha 5, para a segunda alternativa da produção, é empregado o componente `AllocDynamic` para gerar a instrução de alocação da área de memória de uma estrutura. O parâmetro utilizado pelo componente é o atributo do nodo "`type_name`", usado para conseguir o tipo da estrutura alocada. Como ocorre na alternativa anterior, o retorno do componente contém a instrução de alocação da área de memória e é utilizado para preencher o atributo de expressão do nodo "`exp`".

```
1 exp ::= "array_alloc" type_name dim_exprs{
2     exp->exp_attr = cb->AllocDynamic(type_name->_type,
          dim_exprs->dim_attr);
3   }
4   | "record_alloc" type_name{
5     exp->exp_attr = cb->AllocDynamic(type_name->_type);
6   };
```

Listagem 4.36: Exemplo da utilização do componente de alocação dinâmica.

Esse componente torna a implementação de alocação dinâmica simples, pois dispensa a necessidade de realizar cálculos relativos a endereçamento de memória ou ao tipo para se conseguir alocar uma variável durante a execução de um programa.

## 4.4 Comandos

Comandos, segundo Watt & Findlay [2004], são as construções executadas para atualizar variáveis e realizar tarefas. Uma atribuição de um valor a uma variável é considerado um comando por atualizar seu valor. Outros comandos são os condicionais e as repetições que consistem de uma condição e subcomandos para serem executados. Outro comando possível é o retorno de uma função que não atualiza diretamente uma variável, mas termina uma função e pode, indiretamente, fornecer o valor de atualização para uma variável.

As classes `Command_Attr` e `Expression_Attr` são utilizadas pelos componentes de geração de código de comando. Os objetos da primeira classe contêm somente a lista de instruções geradas, enquanto objetos da segunda classe contêm também o endereço da instrução que dão valor da expressão além do seu tipo.

### 4.4.1 Atribuição

A atribuição de valores básicos a endereços calculados ou variáveis simples é uma cópia trivial do valor ao endereço calculado.

O componente `BuildAssignmentStatement` (Listagem 4.37) gera a instrução de atribuição. O primeiro parâmetro é o endereço do destino da atribuição e o segundo parâmetro é o valor que será atribuído ao destino. Ao final, a instrução gerada é retornada como atributo do componente.

```
Command_Attr* BuildAssignmentStatement(Expression_Attr* exp1,
                                       Expression_Attr* exp2);
```

Listagem 4.37: Componente para geração do comando de atribuição.

A Listagem 4.38 apresenta um exemplo da utilização do componente do comando de atribuição. É usada a linguagem *Back* para descrever uma produção de *AST*, cuja ação semântica é a geração da atribuição de um valor a um endereço. Na linha 2, é empregado o componente `BuildAssignmentStatement` que gera essa instrução de atribuição. O atributo de expressão do nodo "`access_exp`", que contém o endereço em que será atribuído o valor, é usado como primeiro parâmetro. O segundo parâmetro é o

atributo de expressão do nodo "`exp`" que contém a instrução do valor que será atribuído. A instrução de atribuição gerada é agregada a lista de instruções do atributo retornado pelo componente, que é utilizado para preencher o atributo de comando do nodo "`stm`".

```
1 stm ::= "assign" access_exp exp{
2     stm->stm_attr = cb->BuildAssignmentStatement(access_exp->exp_attr,
          exp->exp_attr);
3   };
```

Listagem 4.38: Exemplo da utilização do componente de atribuição.

Com a utilização desse componente de atribuição e a combinação dos componentes para cálculo de endereço é possível reduzir a complexidade de geração de código, tornando a implementação do compilador mais simples.

### 4.4.2 Condicional

Os comandos condicionais permitem que, dada uma condição, seja escolhido um subcomando para ser executado.

`BuildConditionalStatement` (Listagem 4.39) é o componente de geração da instrução condicional. O primeiro parâmetro contém a lista de instruções utilizada para gerar a condição que define qual dos subcomandos será executado. Os parâmetros seguintes são, respectivamente, o atributo contendo a lista de instruções do primeiro subcomando e o atributo contendo a lista de instruções do segundo subcomando. Ao fim do componente, a instrução gerada é retornada como atributo do componente.

Caso não haja necessidade do último subcomando, no caso de um comando *if-then*, por exemplo, deve-se utilizar o componente `BuildSkip` (Listagem 4.2) como argumento.

```
Command_Attr* BuildConditionalStatement(Expression_Attr* exp,
                                        Command_Attr* stm1,
                                        Command_Attr* smt2);
```

Listagem 4.39: Componente para geração do comando condicional.

A Listagem 4.40 mostra um exemplo da utilização do componente do comando condicional. É utilizada a linguagem *Back* para descrever uma produção da *AST* com duas alternativas: uma opção cuja ação semântica é a geração do comando condicional completo e outra cuja ação semântica é a geração do comando *if-then* somente. A primeira alternativa, na linha 2, emprega o componente `BuildConditionalStatement` para gerar a instrução do comando condicional. O primeiro parâmetro usado é o

atributo de expressão do nodo "`exp`" contendo o endereço da condição utilizada para decidir qual dos subcomandos é executado. O segundo parâmetro aplicado é o atributo de comando do nodo "`block1`" que contém a lista de instruções do primeiro subcomando que é executado. De maneira análoga, o terceiro parâmetro é o atributo de comando do nodo "`block2`" que contém a lista de instruções do segundo subcomando que é executado. A instrução do comando condicional gerada é colocada na lista de instruções do atributo retornado pelo componente. Esse atributo retornado é utilizado para preencher o atributo de comando do nodo "`stm`". A segunda alternativa também emprega, na linha 5, o componente `BuildConditionalStatement` para gerar a instrução do comando condicional. De modo similar ao componente anterior, o primeiro parâmetro usado é o atributo de expressão do nodo "`exp`" que contém a instrução do valor da condição utilizada para decidir a execução do subcomando. O segundo parâmetro aplicado é o atributo de comando do nodo "`block`" que contém a lista de instruções do subcomando que pode ser executado. Para o terceiro parâmetro é utilizado o componente `BuildSkip`, responsável por anular o segundo subcomando do comando condicional. De maneira similar ao componente de geração anterior, a instrução do comando condicional gerada é colocada na lista de instruções do atributo retornado pelo componente. O atributo retornado é usado para preencher o atributo de comando do nodo "`stm`".

```
1 stm ::= "ifthenelse" exp block1 block2{
2     stm->stm_attr = cb->BuildConditionalStatement(exp->exp_attr,
          block1->stm_attr, block2->stm_attr);
3   }
4   | "ifthen" exp block{
5     stm->stm_attr = cb->BuildConditionalStatement(exp->exp_attr,
          block->stm_attr, cb->BuildSkip());
6   };
```

Listagem 4.40: Exemplo da utilização do componente de comando condicional.

Assim como os componentes previamente descritos, o referente à construção do comando condicional realiza o encapsulamento de suas ações semânticas, como sua contraparte de expressão, não necessitando do gerenciamento de desvios ou outros detalhes. Dessa forma, o componente simplifica a implementação do compilador de uma linguagem de programação que utilize esse tipo de construção.

### 4.4.3 Repetição

Um comando de repetição possivelmente causa a reiteração da execução de seu subcomando e por conseguinte possui, como o comando condicional, uma condição que

controlará a quantidade de vezes que o subcomando será executado, além de poder possuir a declaração de uma variável de controle e seu incremento.

O componente `BuildRepetitionStatement` (Listagem 4.41) gera a instrução de repetição. O primeiro parâmetro é a lista de instruções para declarar e iniciar as variáveis de controle da repetição, sendo utilizado o componente `BuildSkip` (Listagem 4.2) caso não seja necessário realizar esse passo. O segundo parâmetro deve ser o atributo com a lista de instruções referentes à condição que encerra a repetição e retoma a execução do programa. O terceiro parâmetro é a lista de código para realizar o incremento das variáveis de controle da repetição. Assim como o primeiro parâmetro, caso esse último não seja necessário, deve-se utilizar o componente `BuildSkip` (Listagem 4.2). O quarto parâmetro é o subcomando da repetição, ou seja, o atributo contendo a lista de instruções que deverão ser executadas enquanto a condição for verdadeira. O último parâmetro é um booleano para indicar, no caso de ser utilizado o componente `BuildSkip` (Listagem 4.2) no primeiro e terceiro parâmetros, se será gerada uma construção *do while* caso verdadeiro ou *while* caso falso. Ao final do componente, a instrução gerada é retornada como atributo do componente.

```
Command_Attr* BuildRepetitionStatement(Command_Attr* from,
                                       Expression_Attr* to,
                                       Command_Attr* by,
                                       Command_Attr* body,
                                       bool do_while);
```

Listagem 4.41: Componente para geração do comando de repetição.

O componente mostrado, assim como os anteriores, simplifica a implementação de um compilador por encapsular os detalhes da geração das diversas construções de repetições. Contudo, mesmo sendo genérico o suficiente para cobrir os comandos de repetição da maioria das linguagens, algumas, como Algol 60, precisam de componentes próprios para compilar suas particularidades, uma vez que elas não foram difundidas em outras linguagens de programação populares.

Na Listagem 4.42, está um exemplo da utilização do componente do comando de repetição. É utilizada uma descrição da *AST* na linguagem *Back* contendo uma produção com três alternativas: a primeira é responsável pela geração do comando de repetição *for*; a segunda é responsável pela geração do comando *while* e a última é responsável pela geração do comando *do while*. Na linha 2, é empregado o componente `BuildRepetitionStatement` para a geração da instrução do comando de repetição da primeira alternativa. O primeiro parâmetro usado é o atributo de comando do nodo "`stm1`" que contém a lista de instruções utilizadas para inicializar as variáveis de controle da repetição. O segundo parâmetro é o atributo de expressão do nodo "`exp`"

que contém o endereço da condição utilizada para encerrar a repetição. O terceiro parâmetro é o atributo de comando do nodo "`stm2`" que possui a lista de instruções utilizadas para incrementar as variáveis de controle da repetição. O quarto parâmetro empregado é o atributo de comando do nodo "`block`" que contém a lista de instruções do subcomando que será repetido pelo comando de repetição. A instrução do comando de repetição gerada é colocada na lista de instruções do atributo retornado pelo componente. O atributo retornado é usado para preencher o atributo de comando do nodo "`stm`".

Na linha 5, o componente `BuildRepetitionStatement` é utilizado para gerar a instrução do comando de repetição *while*. Diferentemente da alternativa anterior, a variável de controle não é declarada ou incrementada conjuntamente com a construção, portanto, para o primeiro e terceiro parâmetros é utilizado o componente `BuildSkip` que anula tais parâmetros, permitindo a geração da instrução correta. O segundo parâmetro é o atributo de expressão do nodo "`exp`" que contém o endereço da condição utilizada para encerrar a repetição. O quarto parâmetro aplicado por esse componente é o atributo de comando do nodo "`block`" que contém a lista de instruções do subcomando que será repetido pelo comando de repetição. A instrução gerada é colocada na lista de instruções do atributo retornado pelo componente e o atributo retornado é usado para preencher o atributo de comando do nodo "`stm`". O componente `BuildRepetitionStatement` é empregado na linha 8, na terceira alternativa, de maneira similar a segunda alternativa. É acrescentado somente o quinto parâmetro que indica a geração da instrução de repetição *do while*. Apesar da instrução gerada ser diferente, as ações realizadas não são alteradas, portanto ela é colocada na lista de instruções do atributo retornado pelo componente e o atributo retornado é usado para preencher o atributo de comando do nodo "`stm`".

```
1  stm ::= "for" stm1 exp stm2 block{
2      stm->stm_attr = cb->BuildRepetitionStatement(stm1->stm_attr,
           exp->exp_attr, stm2->stm_attr, block->stm_attr);
3    }
4    | "whiledo" exp block{
5      stm->stm_attr = cb->BuildRepetitionStatement(cb->BuildSkip(),
           exp->exp_attr, cb->BuildSkip(), block->stm_attr);
6    }
7    | "dowhile" block exp{
8      stm->stm_attr = cb->BuildRepetitionStatement(cb->BuildSkip(),
           exp->exp_attr, cb->BuildSkip(), block->stm_attr, true);
9    };
```

Listagem 4.42: Exemplo da utilização do componente do comando de repetição.

### 4.4.4 Sequenciador de Retorno

O sequenciador de retorno é a construção que finaliza a execução de uma função. Caso ocorra dentro de uma função que possua um tipo de retorno, o sequenciador deve transportar o valor para o chamador da função além de terminar sua execução.

O componente `BuildReturnStatement` (Listagem 4.43) gera uma instrução de retorno. O parâmetro que deve ser passado é o valor que será transportado ao chamador da função. Caso esse valor não exista devido ao retorno pertencer a uma função sem retorno, deve-se utilizar o componente `BuildSkip` (Listagem 4.2) para a passagem do parâmetro. Ao final gera-se a instrução que é retornada como atributo do componente.

```
Command_Attr* BuildReturnStatement(Expression_Attr* val);
```

Listagem 4.43: Componente para geração de retorno de função.

Esse componente simplifica, como outros, o tratamento de endereços calculados para poder retornar valores da função, uma vez que não é necessário que o implementador se preocupe com os detalhes de carga de variáveis para utilizar seus valores no retorno da função.

A Listagem 4.44 mostra o exemplo de utilização do componente do sequenciador de retorno. É usada a linguagem *Back* para descrever uma produção com duas alternativas: uma em que o sequenciador de retorno transporta um valor para o chamador da função e outra em que o sequenciador de retorno somente encerra a execução da função. Na linha 2, o componente `BuildReturnStatement` empregado na primeira alternativa utiliza como parâmetro o atributo de expressão do nodo "`exp`" que contém a instrução do valor que é retornado pela função. A instrução gerada pelo componente é agregada à lista de instruções do atributo retornado e esse atributo é utilizado para preencher o atributo de comando do nodo "`stm`". Na linha 5, o componente `BuildReturnStatement` é aplicado na segunda alternativa. É empregado como parâmetro o componente `BuildSkip`, que anula o valor do atributo usado, permitindo o uso da instrução de retorno vazia. Assim como na alternativa anterior, a instrução gerada pelo componente é agregada à lista de instruções do atributo retornado. Esse atributo é utilizado para preencher o atributo de comando do nodo "`stm`".

```
1 stm ::= "return" exp{
2     stm->stm_attr = cb->BuildReturnStatement(exp->exp_attr);
3   }
4   | "return" {
5     stm->stm_attr = cb->BuildReturnStatement(cb->BuildSkip());
6   };
```

Listagem 4.44: Exemplo da utilização do componente do comando de retorno de função.

## 4.5 Escopo

Segundo Scott [2009], o escopo de uma declaração é a parte de um programa em que as amarrações de uma linguagem de programação estão disponíveis. As amarrações de uma linguagem são associações entre um identificador e uma entidade de linguagem de programação como endereços. O escopo de uma linguagem de programação pode ser estático, em que as variáveis de uma função são avaliadas durante a definição, produzindo uma única amarração para cada variável, ou dinâmico, em que as amarrações das variáveis de uma função são avaliadas no momento de sua chamada. Optou-se por implementar somente o escopo estático na infraestrutura de geração de código, devido a sua maior simplicidade e compatibilidade com as linguagens de programação imperativa que, na maioria, empregam esse tipo de escopo [Watt & Findlay, 2004].

O método usado para realizar o controle do escopo é feito via tabela de símbolos, coordenando não somente as variáveis que serão amarradas pelos componentes de declaração, como também controlando a abertura e fechamento de níveis da tabela para administrar as camadas de funções e outras estruturas relevantes ao escopo da linguagem que está sendo definida.

Os componentes referentes a abertura e fechamento do escopo são empregados de maneira distinta dos demais. Utilizando as linguagens *Middle* e *Back* para definir as ações semânticas do compilador, é possível aplicar algumas palavras-chave responsáveis pela geração das chamadas dos métodos que realizam as operações pertinentes ao tratamento do escopo. Durante a geração das classes responsáveis pelo caminhamento da *AST*, são geradas, em conjunto com o padrão *visitor*, as chamadas para os métodos que realizam a abertura, o fechamento e o *reset* do escopo.

Para a abertura de um novo nível de escopo na tabela de símbolos, deve-se utilizar a palavra-chave `startblock`. Essa palavra-chave é utilizada entre as variáveis terminais e não-terminais da gramática de *Middle* e *Back*, da maneira similar a uma variável cuja produção na gramática não é especificada. Durante a geração da classe para caminhamento da árvore de sintaxe abstrata, será gerada a chamada para o método responsável pela abertura de um novo escopo na tabela de símbolos nos locais indicados

com a palavra-chave.

A ação contrária da abertura de um novo nível, isto é, o fechamento do último nível aberto, é realizada utilizando a palavra-chave `finishblock`. Ela é aplicada de maneira similar ao seu complemento, e gera, nos locais indicados com a palavra-chave, a chamada para o método responsável pelo fechamento de escopo durante a geração da classe que realiza o caminhamento da árvore de sintaxe abstrata.

Devido à forma como as linguagens de definição e tabela de símbolos implementada no trabalho operam, no fim de cada fase da compilação é preciso restabelecer a tabela de símbolos com a palavra-chave `resetsymboltable`. Isso se faz necessário pois, no término de uma fase de declaração, por exemplo, todos níveis abertos são fechados. Portanto, na próxima fase de verificação de tipos, ao tentar acessar a tabela, não existirá nenhuma variável declarada já que o escopo se encontra fechado.

A Listagem 4.45 mostra um exemplo do uso das palavras-chave para o controle de escopo. São empregadas duas produções de `AST` da linguagem *Middle*. A primeira produção define o programa como sendo um bloco de instruções. A palavra-chave `resetsymboltable` é aplicada nessa produção indicando que, após visitar o nodo "`block`", é realizado o *reset* da tabela de símbolos para que ela seja utilizada nas fases seguintes do compilador. A segunda produção define que bloco contém uma lista de declarações seguida de uma lista de instruções. A palavra-chave `startblock` aplicada nessa produção antes da variável `decl` indica que a chamada ao método para abertura do nível de escopo da tabela de símbolos ocorre antes de visitar o nodo "`decl`". De maneira análoga, a palavra-chave `finishblock` aplicada nessa produção após a variável `stm` indica que a chamada para o método de fechamento do último nível de escopo da tabela de símbolos ocorre após a visita do nodo "`stm`".

```
1 prog ::= block resetsymboltable ;
2 block ::= "dblock" startblock decl stm finishblock ;
```

Listagem 4.45: Exemplo da utilização das palavras-chave para controle de escopo.

Como é possível perceber pelo exemplo apresentado, o uso apenas dessas palavras-chave simplifica a implementação do escopo na linguagem de programação, além de deixar as ações semânticas mais limpas, visto que permite utilizar somente as palavras-chave na gramática para realizar as ações necessárias para se alterar o escopo, permitindo que a gramática definida seja mais legível.

## 4.6 Declarações

De acordo com Watt & Findlay [2004], declarações são construções de programas elaboradas para produzir amarrações e que podem ter efeitos colaterais como a criação de variáveis. Entre as declarações previstas pela infraestrutura de componentes implementada nesta dissertação estão a declaração de variáveis, a definição de funções e a definição de tipos. Entre as variáveis que podem ser declaradas estão: as simples, ou seja, variáveis inteiras e de ponto flutuante, ambas com largura definida durante a declaração; os arranjos que podem ser estáticos ou dinâmicos e multidimensionais; as estruturas que podem conter combinação de variáveis e as classes que se assemelham às estruturas, contudo, podem conter variáveis e também funções como membros.

As classes previamente explicadas que são utilizadas pelos componentes dessa seção são `Command_Attr`, `Dimension_Attr` e `Expression_Attr` (vide Seção 4.4). Objetos dessas classes são utilizados para comunicação de atributos entre os diversos componentes, todos objetos da classes possuem um campo para a lista de instruções geradas. Outra classe já explicada é a `Type` (vide Seção 4.3), cujos objetos armazenam as informações do tipo da variável ou valor que está sendo tratado. As novas classes utilizadas nesta seção são:

- `Declaration_Attr` - classe de atributos de declarações cujos objetos contêm, além da lista de instruções geradas, uma lista dos tipos das variáveis declaradas.

- `Parameter_Attr` - classe de atributos de parâmetros cujos objetos contêm, além da lista de instruções geradas, uma lista dos parâmetros da função sendo definida.

- `ParameterCall` - enumeração utilizada para decidir qual o tipo de passagem de um parâmetro. Exemplos desse tipo seriam a passagem por valor, por referência, valor-resultado e por nome.

### 4.6.1 Variáveis

A declaração de variáveis cria uma variável e realiza a ligação dessa a seu identificador por meio da tabela de símbolos. A declaração de uma variável necessita de seu identificador e tipo, além de comumente requerer outras informações como o tamanho de cada tipo, necessário para realizar sua devida alocação. Utilizando os componentes da infraestrutura implementada no trabalho aqui descrito, é possível encapsular essas informações e se concentrar somente no identificador da variável e seu tipo, tornando sua declaração mais simples.

O componente `VariableDeclare` (Listagem 4.46) realiza a instalação de uma variável na tabela de símbolos e gera uma instrução indicando sua declaração na lista de instruções. São necessários o identificador da variável e o objeto da classe *Type* com o tipo da variável declarada. A ação semântica desse componente é a instalação da variável na tabela de símbolos em seu devido escopo.

```
Declaration_Attr* VariableDeclare(std::string id, Type* tp);
```

Listagem 4.46: Componente para geração da declaração de uma variável.

O componente `VariableAlloc` (Listagem 4.47) gera a instrução de alocação de uma variável instalada na tabela de símbolos. É necessário o identificador da variável instalada na tabela de símbolos para que ela seja alocada. É gerada a instrução de alocação que é retornada como atributo do componente.

```
Declaration_Attr* VariableAlloc(std::string id);
```

Listagem 4.47: Componente para geração da alocação de uma variável.

Na Listagem 4.48, está o exemplo da aplicação do componente para a declaração de uma variável. É utilizada uma descrição na linguagem *Middle* com uma produção cuja semântica é declaração da variável e a transformação da *AST*, simplificando a produção para, utilizando a linguagem *Back*, gerar a instrução de alocação da memória requerida pela variável. Na linha 1, é definido que a produção de *AST* é transformada, retirando o nodo "`type_name`" da produção. Na linha 2, é aplicado o componente `VariableDeclare` que realiza a declaração da variável instalando seu identificador e tipo na tabela de símbolos. O primeiro parâmetro usado é o identificador da variável, o segundo parâmetro é o atributo do nodo "`type_name`", empregado para conseguir o tipo da variável declarada. O componente de declaração agrega o tipo da variável declarada à lista de tipos do atributo retornado. O atributo retornado é usado para preencher o atributo de declaração do nodo "`decl`".

```
1 decl ::= type_name id => decl ::= id where{
2     decl->decl_attr = cb->VariableDeclare(id, type_name->type);
3   };
```

Listagem 4.48: Exemplo da utilização do componente de declaração de variável.

Na Listagem 4.49, está o exemplo da aplicação do componente de alocação de uma variável declarada. É utilizada uma descrição na linguagem *Back* com a produção de *AST* criada pela transformação do exemplo da Listagem 4.48. Na linha 2, é empregado o componente `VariableAlloc` para gerar a instrução de alocação da variável

instalada na tabela de símbolos. O parâmetro aplicado é o identificador da variável que é alocada. A instrução gerada pelo componente é agregada a lista de instruções do atributo retornado que é utilizado para preencher o atributo de declaração do nodo "`decl`".

```
1 decl ::= id{
2     decl->decl_attr = cb->VariableAlloc(id);
3   };
```

Listagem 4.49: Exemplo da utilização do componente de alocação de variável.

### 4.6.2 Arranjo

Arranjo é um dos tipos compostos mais comuns, sendo utilizado para armazenar uma coleção de valores em posições consecutivas da memória. São duas as formas de alocação utilizadas para arranjos: estática, que é realizada durante a compilação e a dinâmica, que ocorre em tempo de execução.

As ações necessárias para utilização de um arranjo se referem a sua declaração, alocação e acesso. No caso do arranjo estático, a declaração e alocação devem ser realizadas no mesmo momento, portanto, somente um componente é necessário para essa ação. No caso do arranjo dinâmico há a separação entre a declaração e a alocação, sendo a primeira realizada pelo componente `BuildStaticDimension` (Listagem 4.50 e a segunda feita pelo componente `AllocDynamic` (Listagem 4.35).

O componente `BuildStaticDimension` (Listagem 4.50) tem o propósito de criar a lista de dimensão de um arranjo estático, portanto seus parâmetros referentes à dimensão devem ser números naturais. Esses parâmetros são agrupados em uma lista e retornados através do atributo `Dimension_Attr` para serem utilizados pela declaração do arranjo estático.

```
Dimension_Attr* BuildStaticDimension(uint64_t i0,
                                     uint64_t i1);
```

Listagem 4.50: Componente para geração da dimensão estática do arranjo.

O componente `StaticArrayDecl` (Listagem 4.51) é o que realiza a declaração de um arranjo estático. O primeiro parâmetro que lhe é passado é o identificador do arranjo, o segundo é o tipo dos itens armazenados por ele e o último parâmetro é sua dimensão estática para que seja feita sua alocação.

Na Listagem 4.52, está um exemplo da aplicação dos componentes de declaração de um arranjo estático. É utilizada a linguagem *Middle* para realizar a definição de

```
Declaration_Attr* StaticArrayDecl(std::string id,
                                  Type* ItemType,
                                  Dimension_Attr* dim);
```

Listagem 4.51: Componente para declaração do arranjo estático.

três produções de *AST*: uma é responsável por gerar a lista de dimensões; outra é responsável por realizar a concatenação das listas de dimensões e a terceira é responsável pela declaração do arranjo. A primeira produção emprega, na linha 2, o componente `BuildStaticDimension` para criar a lista de dimensões. Os dois parâmetros aplicados são números naturais utilizados para criar os limites do arranjo. O componente cria a lista com os números agrupados e a retorna no atributo que é empregado para preencher o atributo de dimensão do nodo "`a_size`". A segunda produção aplica, na linha 5, o componente `BuildSequence` para concatenar as listas de dimensões criadas pela produção anterior. Como parâmetro são utilizados os atributos de dimensões dos nodos "`a_sizes1`" e "`a_size`". O atributo retornado contém a lista de dimensão dos atributos de parâmetro concatenada. Esse atributo retornado é utilizado para preencher o atributo de dimensão do nodo "`a_sizes`".

```
1 a_size ::= "range" num1 num2 where{
2     a_size->dim_attr = cb->BuildStaticDimension(num1, num2);
3   };
4 a_sizes ::= a_sizes1 a_size where {
5     a_sizes->dim_attr = cb->BuildSequence(a_sizes1->dim_attr,
          a_size->dim_attr);
6   };
7 decl ::=  "array" id type_name a_sizes => decl ::= id where{
8     decl->decl_attr = cb->StaticArrayDecl(id, type_name->type,
          a_sizes->dim_attr);
9   };
```

Listagem 4.52: Exemplo da utilização dos componentes de declaração de arranjo estático.

A terceira produção define, na linha 7, a transformação da *AST* para simplificar a produção e permitir a alocação da variável na linguagem *Back*. Na linha 8, é utilizado o componente `StaticArrayDecl` para realizar a declaração do arranjo, criando seu tipo e instalando-o na tabela de símbolos juntamente com seu identificador. O primeiro parâmetro empregado pelo componente é o identificador do arranjo, o segundo parâmetro é o atributo do nodo "`type_name`", usado para conseguir o tipo do item armazenado pelo arranjo. O terceiro parâmetro é o atributo de dimensão do nodo "`a_sizes`" que contém a lista de dimensões utilizada para a criação do arranjo estático. Assim como o componente de declaração de variável, esse componente agrega o tipo criado para

o arranjo à lista de tipos do atributo retornado. O atributo retornado é usado para preencher o atributo de declaração do nodo "`decl`".

O componente `BuildDynamicDimension` (Listagem 4.53) tem o propósito de criar a dimensão de um arranjo dinâmico, sendo similar à sua contraparte estática mas recebendo como parâmetro atributos de expressão para que o tamanho do arranjo seja calculado durante a execução do programa compilado. Esse componente deve ser utilizado em conjunto com o `AllocDynamic` para realizar a alocação de um arranjo dinamicamente.

```
Dimension_Attr* BuildDynamicDimension(Expression_Attr* i0,
                                      Expression_Attr* i1);
```

Listagem 4.53: Componente para geração do dimensão do arranjo dinâmico.

Na Listagem 4.54, está o exemplo para a geração da instrução da dimensão para um arranjo dinâmico. É utilizada a linguagem *Back* para realizar a definição de duas produções de *AST*: uma responsável por gerar a lista de dimensões do arranjo dinâmico e outra responsável por realizar a concatenação das listas de dimensões. Na linha 2, é empregado o componente `BuildDynamicDimension` para gerar a lista de dimensões. São aplicados os atributos de expressão dos nodos "`exp1`" e "`exp2`" como parâmetros, em que ambos contém o endereço da instrução utilizada como valor do limite da dimensão gerada. O componente gera a lista com as instruções recebidas dos atributos e retorna um atributo que será empregado para preencher o atributo de dimensão do nodo "`dim_expr`". A segunda produção usa, na linha 5, o componente "`BuildSequence`" para concatenar as listas de dimensões. Os parâmetros aplicados são os atributos de dimensão dos nodos "`dim_expr1`" e "`dim_expr2`". O atributo que é retornado pelo componente contém uma lista com as dimensões passadas pelos atributos concatenadas. Esse atributo retornado é utilizado para preencher o atributo de dimensão do nodo "`dim_expr`".

```
1 dim_expr ::= "range" exp1 exp2 {
2     dim_expr->dim_attr = cb->BuildDynamicDimension(exp1->exp_attr,
          exp2->exp_attr);
3   };
4 dim_expr ::= dim_expr1 dim_expr2 {
5     dim_expr->dim_attr = cb->BuildSequence(dim_expr1->dim_attr,
          dim_expr2->dim_attr);
6   };
```

Listagem 4.54: Exemplo da utilização do componente de dimensão do arranjo dinâmico.

O componente de declaração de arranjo dinâmico `DynamicArrayDecl` (Listagem 4.55) é similar à sua contraparte estática, com a diferença de que não é alocado o

arranjo com as dimensões completas, mas um ponteiro para o tipo do arranjo, ou seja, não é necessário que um dos parâmetros seja a dimensão do arranjo.

```
Declaration_Attr* DynamicArrayDecl(std::string id,
                                   Type* ItemType);
```

Listagem 4.55: Componente para declaração do arranjo dinâmico.

A Listagem 4.56 contém o exemplo da declaração de um arranjo dinâmico. É utilizada uma produção descrita na linguagem *Middle* para realizar a declaração do arranjo e transformar a *AST*, simplificando a produção que é utilizada pela linguagem *Back* para gerar a alocação da variável declarada. Na primeira linha, é definida essa transformação, retirando o não-terminal `"array"` e o nodo "`type_name`". Na linha 2, é aplicado o componente `DynamicArrayDecl` que realiza a declaração do arranjo, criando seu tipo e instalando-o junto ao identificador na tabela de símbolos. O primeiro parâmetro do componente é o identificador utilizado pelo arranjo, sendo o segundo parâmetro o atributo do nodo "`type_name`", empregado para conseguir o tipo do item armazenado pelo arranjo. Assim como os outros componentes de declaração, o componente agrega o tipo criado para o arranjo à lista de tipos do atributo retornado. Esse atributo é utilizado para preencher o atributo de declaração do nodo "`decl`".

```
1 decl ::= "array" id type_name => decl ::= id where{
2     decl->decl_attr = cb->DynamicArrayDecl(id, type_name->type);
3   };
```

Listagem 4.56: Exemplo da utilização do componente de declaração de um arranjo dinâmico.

Os componentes para declaração de arranjo buscam facilitar a criação desse tipo de variável, sendo os componentes de dimensões encapsulamento das ações necessárias para a criação das lista dos limites, e os outros componentes encapsulamentos dos passos de alocação e amarração dos arranjos na tabela de símbolos. Esses encapsulamentos se tornam mais úteis devido aos arranjos dinâmicos, pois permitem que o implementador não precise se preocupar com os detalhes referentes aos ponteiros do arranjo.

### 4.6.3 Estruturas

Estruturas em linguagens de programação são tipos definidos pelo usuário que contêm outras variáveis. Dessa forma, são necessários métodos que os definam previamente e recuperem as estruturas já definidas, além de um componente para declaração de uma instância que pode ser tanto estática quanto dinâmica.

O método `RecordDefinition` (Listagem 4.57) se refere a definição de uma estrutura. É necessário seu identificador como primeiro parâmetro, e o atributo que contém a lista de tipos das declarações de variáveis como segundo parâmetro. Essa lista será percorrida e, a partir dela, o tipo da estrutura é criado e armazenado para, quando necessário, ser recuperado e assim instanciado.

```
void RecordDefinition(std::string id,
                      Declaration_Attr* decls);
```

Listagem 4.57: Método para definição de estruturas.

A Listagem 4.58 mostra um exemplo da utilização do componente para definição de uma estrutura. É empregada uma produção descrita na linguagem *Middle* para realizar a definição. Na linha 2, o método `RecordDefinition` é aplicado como primeiro parâmetro, e por conseguinte, o identificador que a estrutura terá. Como segundo parâmetro é usado o atributo de declaração do nodo "`decl`" que contém a lista das variáveis declaradas para definição dos campos da estrutura. Com essas informações, o método cria o tipo da estrutura e o armazena na tabela de símbolos para ser recuperado quando necessário, permitindo a instanciação de uma estrutura.

```
1 rec_def ::= "record" id decl where{
2     cb->RecordDefinition(id, decl->decl_attr);
3   };
```

Listagem 4.58: Exemplo da utilização do método de definição de uma estrutura.

`GetRecordType` (Listagem 4.59) é o método que recupera o tipo de uma estrutura definida previamente. O único parâmetro é o identificador da estrutura definida.

```
Type* GetRecordType(std::string id);
```

Listagem 4.59: Método para recuperação do tipo de estruturas definidas.

O componente `RecordDeclaration` (Listagem 4.60) possui a mesma utilidade que o componente `VariableDeclare` (Listagem 4.46), possibilitando a declaração de uma estrutura. Há três parâmetros necessários para realização dessa declaração, são eles: o identificador da estrutura que será declarada, o seu tipo recuperado utilizando o componente da Listagem 4.59 e, por último, um booleano para indicar se é criado somente um ponteiro para a estrutura, necessitando da execução do componente `AllocDinamic` para utilizar a estrutura ou se é alocada a estrutura por completo.

Na Listagem 4.61, está um exemplo para a utilização do componente de declaração de uma estrutura. São utilizadas duas produções da linguagem de descrição

```
Declaration_Attr* RecordDeclaration(std::string id,
                                    Type* RecType,
                                    bool pointer);
```

Listagem 4.60: Componente para declaração de estruturas.

*Middle*, uma que realiza a declaração da estrutura e outra que obtêm seu tipo definido previamente. A primeira produção emprega o componente `RecordDeclaration` na linha 2, para realizar a declaração de uma estrutura. Como primeiro parâmetro, é aplicado o identificador que a estrutura declarada terá. O segundo parâmetro é o atributo de tipo do nodo "`type_name`", contendo as informações do tipo da estrutura. O último parâmetro empregado define que a estrutura será alocada diretamente, não necessitando da execução do componente `AllocDinamic`. Então o componente declara a estrutura, armazenando seu tipo e identificador na tabela de símbolos, agrega o tipo criado à lista de tipos do atributo e o retorna. Esse atributo retornado é utilizado para preencher o atributo de declaração do nodo "`decl`". A segunda produção emprega o método `GetRecordType` para obter o tipo da estrutura previamente definida. O parâmetro utilizado pelo método é o identificador da estrutura definida. É retornado o tipo da estrutura buscada, sendo esse tipo usado para preencher o atributo de tipo do nodo "`type_name`".

```
1 decl ::= type_name id where{
2       decl->decl_attr = cb->RecordDeclaration(id, type_name->type, false);
3   };
4 type_name ::= "record" id where{
5     type_name->type = cb->GetRecordType(id);
6   };
```

Listagem 4.61: Exemplo da utilização do componente de declaração de uma estrutura.

### 4.6.4 Funções

Função é a realização de uma abstração de instruções do programa sendo compilado, uma vez que substitui-se uma sequência de instruções por uma chamada de função que as realize. Além disso, funções podem possuir parâmetros, permitindo que recebam variáveis e valores, fazendo com que se tornem mais genéricos. Outro detalhe, funções podem retornar valores calculados, aumentando mais a generalidade da abstração fornecida.

O componente `AddParameter` (Listagem 4.62) permite definir um parâmetro que é adicionado à lista de parâmetros do atributo retornado pelo componente. Essa definição é realizada de forma similar à declaração de uma variável, requerendo o iden-

tificador ao qual o parâmetro será amarrado e seu tipo. Também é necessário definir qual será a forma utilizada para a passagem do parâmetro, podendo ser: por valor; referência; valor-resultado ou por nome. Esses dados são agrupados e anexados a lista de parâmetros da classe de atributo `Parameter_Attr`.

```
Parameter_Attr* AddParameter(std::string id,
                             Type* tp,
                             ParameterCall parcall);
```

Listagem 4.62: Componente para listagem de parametros de função.

O componente `BuildFunctionHeader` (Listagem 4.63) é utilizado para construir o cabeçalho da função e realizar sua amarração ao identificador passado. Assim, o primeiro parâmetro passado é o nome da função; o segundo é seu tipo de retorno, sendo necessário utilizar o tipo `Void` caso seja preciso declarar uma função que não retorne um valor; e o último parâmetro é o atributo retornado pelo componente anterior, o que permite a declaração dos parâmetros da função. Caso a função não os possua, existe outro componente similar em que esse parâmetro é omitido, permitindo a declaração de uma função sem parâmetros. Com essas informações, o tipo da função que está sendo declarada é construído e armazenado na tabela de símbolos para recuperação posterior, seja para utilização durante a chamada da função ou para sua construção, aplicando o próximo componente.

```
Declaration_Attr* BuildFunctionHeader(std::string id,
                                      Type* rtn,
                                      Parameter_Attr* fpar);
```

Listagem 4.63: Componente para declaração de cabeçalho da função.

Na Listagem 4.64, está um exemplo da declaração do cabeçalho de uma função. É utilizada uma descrição na linguagem *Middle* com três produções: uma produção para definir um parâmetro da função; uma produção para concatenar as listas de parâmetros gerados e uma produção para declarar o cabeçalho da função. A primeira produção emprega o componente `AddParameter` que cria o objeto da classe `Type` que contém as informações do parâmetro para sua utilização pela função. Como primeiro parâmetro é usado o identificador do parâmetro. O segundo parâmetro do componente é o atributo de tipo do nodo "`type_name`" que contém as informações do tipo do parâmetro. O último parâmetro utilizado é um valor da enumeração de `ParameterCall` e, nesse caso, o parâmetro da linguagem definida é passado por referência. O componente cria um objeto da classe `Type` para representar o parâmetro e o agrega a lista de

parâmetros do atributo retornado que é empregado para preencher o atributo de parâmetro do nodo "`para_attr`". Na linha 5 é usado, pela segunda produção, o componente `BuildSequence` para concatenar as listas de parâmetros. Os parâmetros aplicados são os atributos de parâmetros dos nodos "`formalpar1`" e "`formalpar2`". O atributo que é retornado pelo componente contém uma lista com os parâmetros passados pelos atributos concatenados. Esse atributo retornado é utilizado para preencher o atributo de parâmetro do nodo "`formalpar`".

A produção referente a declaração do cabeçalho da função emprega o componente `BuildFunctionHeader` para essa tarefa. Como primeiro parâmetro é aplicado o identificador da função. O segundo parâmetro do componente é o atributo de tipo do nodo "`type_name`", que contém a informação de tipo que é utilizada para montar o retorno da função. Como último parâmetro é aplicado o atributo de parâmetro do nodo "`formalpars`" contendo a lista dos parâmetros da função, como explicado previamente. O componente realiza a instalação da função na tabela de símbolos e, como os componentes de declaração, agrega à lista de tipos do atributo retornado o tipo criado para a função. O atributo retornado é utilizado para preencher o atributo de declaração do nodo "`fnt_header`".

```
1 formalpar ::= "var_ref" type_name id where{
2     formalpar->para_attr = cb->AddParameter(id, type_name->type,
          component::CallByReference);
3   };
4 formalpar ::= formalpar1 formalpar2 where {
5     formalpar->para_attr = cb->BuildSequence(formalpar1->para_attr,
          formalpar2->para_attr);
6   };
7 fnt_header ::= "function" type_name id formalpar where{
8     fnt_header->decl_attr = cb->BuildFunctionHeader(id, type_name->type,
          formalpar->para_attr);
9   };
```

Listagem 4.64: Exemplo da utilização dos componentes para declaração de uma função.

O componente `BuildFunction` (Listagem 4.65) realiza o atrelamento de uma função que teve seu cabeçalho previamente declarado pelo componente anterior e o código que esta função executará. O primeiro parâmetro necessita do método `GetFunction` que recebe o identificador da função e retorna as informações necessárias, e o segundo é o atributo que contém a lista de instruções que a função deverá executar.

```
Declaration_Attr* BuildFunction(Function* fun, Declaration_Attr* stm);
```

Listagem 4.65: Componente para preenchimento do corpo da função.

Na listagem 4.66, está o exemplo da utilização do componente que realiza o atrelamento de uma função a seu cabeçalho. É utilizada uma produção descrita na linguagem *Back* para realizar essa tarefa. Na linha 2, é empregado o componente `BuildFunction` que realiza o atrelamento da função com a lista de instruções dada. Para o primeiro parâmetro é aplicado o método `GetFunction` para obter as informações da função na tabela de símbolos. O segundo parâmetro é o atributo de comando do nodo "`block`" que contém a lista de instruções a serem executadas pela função. O componente altera a definição da função, atrelando a lista de instruções dadas a sua definição, e agrega a definição da função à lista de instruções do atributo retornado. O atributo retornado é, então, utilizado para preencher o atributo de comando do nodo "`fnt`".

```
1 fnt ::= fnt_header block {
2     fnt->stm_attr = cb->BuildFunction(cb->GetFunction(fnt_header->id),
          block->stm_attr);
3   };
```

Listagem 4.66: Exemplo da utilização do componente para preechimento de uma função.

Nos componentes expostos nesta seção, a definição dos parâmetros de uma função, bem como sua forma de chamada, pode simplificar e facilitar a implementação de um compilador, principalmente considerando passagens de parâmetro mais complexas, como a passagem por nome que necessita a criação de *thunks* - funções que são utilizadas no lugar das variáveis. Dessa forma, é possível não somente facilitar como acelerar o processo de implementação do compilador de uma linguagem de programação.

### 4.6.5 Classes

Classes em linguagens de programação são abstrações de dados similares às estruturas, com a diferença de permitirem a definição de funções em conjunto com as variáveis. Por conseguinte, seus componentes funcionam de forma similar aos utilizados pelas estruturas.

O componente `ClassDefinition` (Listagem 4.67) realiza a definição de uma classe. Tal como é realizado no componente de estrutura, é necessário o identificador da classe como primeiro parâmetro, enquanto o segundo é o atributo com as declarações de membros da classe, sejam variáveis ou funções. Essas informações serão utilizadas para gerar o tipo da classe que será armazenado para uso subsequente.

A Listagem 4.68 contém um exemplo para a definição de uma classe. É utilizada uma produção descrita na linguagem *Middle* para realizar tal definição, onde o

```
void ClassDefinition(std::string id, Declaration_Attr* decls);
```

Listagem 4.67: Componente para definição de classes.

nodo "`field_declaration`" deve conter declarações de variáveis e funções. Na linha 2, é empregado o método `ClassDefinition`, sendo usado como primeiro parâmetro o identificador que a classe terá. Como segundo parâmetro é empregado o atributo de declaração do nodo "`field_declaration`". Com essas informações, o método cria o tipo da classe e o armazena na tabela de símbolos para ser recuperado quando necessário, permitindo a instanciação de uma classe.

```
1 type_declaration ::= "class_decl" id field_declaration where {
2     cb->ClassDefinition(id, field_declaration->decl_attr);
3   };
```

Listagem 4.68: Exemplo da utilização do método de definição de uma classe.

`GetClassType` (Listagem 4.69) é o método que recupera o tipo de uma classe definida para sua declaração. O parâmetro é o identificador da classe para que seja buscada sua definição.

```
Type* GetClassType(std::string id);
```

Listagem 4.69: Método para recuperação do tipo de classes definidas.

Assim como o componente de declaração de uma instância de estrutura, o componente `ClassDeclaration` (Listagem 4.70) tem como finalidade a declaração de uma instância de um objeto de uma classe. Dessa maneira, são usados como parâmetros: o identificador que será dado a instância; o tipo da classe sendo instanciada e booleando que indica se o objeto instanciado será criado como ponteiro para o objeto ou como o objeto completo. Caso seja criado o ponteiro para o objeto é necessário, portanto, o uso posterior do componente `AllocDinamic` (Listagem 4.35) para alocar o espaço utilizado pela instância do objeto da classe.

```
Declaration_Attr* ClassDeclaration(std::string id,
                                   Type* ClassType,
                                   bool pointer);
```

Listagem 4.70: Componente para declaração de classes.

Na Listagem 4.71, está um exemplo para a utilização do componente de declaração de uma classe. São utilizadas duas produções da linguagem de descrição *Middle*,

uma que realiza a declaração da classe e outra que obtêm seu tipo definido previamente. A primeira produção emprega o componente `ClassDeclaration` na linha 2, para realizar a declaração de uma classe. O identificador que a classe terá é aplicado como primeiro parâmetro. O segundo parâmetro é o atributo de tipo do nodo "`type_name`", contendo as informações do tipo da classe. O último parâmetro empregado define que a classe será alocada diretamente, não necessitando da execução do componente `AllocDinamic`. Então o componente declara a classe, armazenando seu tipo e identificador na tabela de símbolos, agrega o tipo criado à lista de tipos do atributo e o retorna. Esse atributo retornado é utilizado para preencher o atributo de declaração do nodo "`decl`". A segunda produção emprega o método `GetClassType` para obter o tipo da classe previamente definida. O parâmetro utilizado pelo método é o identificador da classe definida. É retornado o tipo da classe buscada, sendo esse tipo usado para preencher o atributo de tipo do nodo "`type_name`".

```
1 decl ::= type_name id where{
2     decl->decl_attr = cb->ClassDeclaration(id, type_name->type, false);
3   };
4 type_name ::= "type" id where {
5     type_name->_type = cb->GetClassType(id);
6   };
```

Listagem 4.71: Exemplo da utilização do componente de declaração de uma classe.

## 4.7 Verificação de Tipos

A verificação de tipos é realizada com o auxílio de diversos métodos que recuperam o tipo de uma variável declarada. Com essa informação é possível, no momento da verificação de tipos do compilador, realizar os testes necessários para executar as coerções implícitas de operações ou falhas. Um exemplo dessa coerção implícita ocorre ao realizar a soma de um inteiro e um ponto flutuante, em que é necessário converter um dos valores para realizar a operação corretamente. Entretanto, como nem todas as linguagens permitem a coerção implícita, esse mesmo exemplo poderá resultar na falha da compilação do programa.

O método `GetVariableType` (Listagem 4.72) busca o tipo de uma variável e o retorna. Esse método necessita somente do identificador da variável, e, ainda que seja mais genérico que os demais apresentados nesta seção, sempre retornará o tipo da variável. Dessa forma, a requisição do tipo de um arranjo consegue seu tipo, e não o tipo do item armazenado no arranjo.

```
Type* GetVariableType(std::string id);
```

Listagem 4.72: Método para recuperação do tipo de variável.

O exemplo da recuperação do tipo de uma variável está na Listagem 4.73. É utilizada uma produção descrita na linguagem *Middle* cuja ação semântica é a obtenção do tipo da variável. Na linha 2, é empregado o método `GetVariableType` para realizar essa tarefa. O parâmetro usado é o identificador da variável. O método busca o tipo da variável armazenado na tabela de símbolos e o retorna. O tipo retornado é utilizado para preencher o atributo do nodo "`acc_exp`".

```
1 acc_exp ::= id where{
2     acc_exp->type = cb->GetVariableType(id);
3   };
```

Listagem 4.73: Exemplo da utilização do método de recuperação de tipo de variável.

O método `GetArrayType` (Listagem 4.74) é responsável por recuperar o tipo do item armazenado pelo arranjo passado como parâmetro. Como os componentes de cálculo de endereço, é necessária a utilização do método `GetArray` para buscar as informações do arranjo na tabela de símbolos e passar o devido arranjo para o único parâmetro exigido.

```
Type* GetArrayType(Array* arr);
```

Listagem 4.74: Método para recuperação do tipo do item armazenado no arranjo.

```
1 array_exp ::= "arrayexp" id exp{
2   array_exp->array = cb->GetArray(id);
3   array_exp->type = cb->GetArrayType(array_exp->array);
4 };
5 array_exp ::= "array_access" array_access1 expression where {
6   array_exp->array = array_access1->array;
7   array_exp->type = array_exp1->type;
8 };
```

Listagem 4.75: Exemplo da utilização do método de recuperação de tipo de item de um arranjo.

Na Listagem 4.75, é exemplificada a recuperação do tipo dos itens de um arranjo. Para tanto, é utilizada a linguagem *Middle* para descrever duas produções referentes ao acesso de um arranjo. A primeira produção se refere ao acesso do primeiro índice e a segunda se refere ao acesso dos índices subsequentes. Como uma das ações semânticas da primeira produção, na linha 2, é aplicado o método `GetArray` para obter as

informações do arranjo armazenadas na tabela de símbolos. O retorno desse método é utilizado para preencher o atributo de arranjo do nodo "`array_exp`" para, posteriormente, facilitar a busca das informações e dispensar buscas na tabela de símbolos. Na linha 3, é empregado o método `GetArrayType` para recuperar o tipo de item armazenado pelo arranjo. Como parâmetro é usado o atributo de arranjo preenchido na linha anterior. O retorno do método é utilizado para preencher o atributo de tipo do nodo "`array_exp`". A segunda produção não aplica nenhum método ou componente, uma vez que o tipo das variáveis de um arranjo é o mesmo para o arranjo inteiro. Entretanto, dada a construção da árvore para arranjo multidimensionais, é preciso que os atributos obtidos na primeira produção sejam replicados na segunda, e tal operaçào foi realizada nas linhas 5-6.

O método `GetBaseRecordVarType` (Listagem 4.76) se refere à aquisição do tipo de uma variável de uma estrutura. São necessárias as informações da estrutura fornecidas pelo método `GetRecord` e o identificador da variável cujo tipo será buscado.

```
Type* GetBaseRecordVarType(Record* rec, std::string id);
```

Listagem 4.76: Método para recuperação do tipo da estrutura mais interna.

O método `GetRecordVarType` (Listagem 4.77) serve para obter o tipo de uma variável de uma estrutura interna à outra estrutura. Seu funcionamento é similar ao cálculo de endereço de estrutura apresentado na Seção 4.3.2, com a exceção de não precisar de um parâmetro com o atributo do endereço anterior calculado. Desse modo, os parâmetros são os mesmos do método anterior: as informações da estrutura fornecidos pelo `GetRecord` e o identificador da variável sendo buscada nesse nível, e o que difere os dois métodos são as ações internas. Esse método percorre uma lista das variáveis acessadas antes de retornar o tipo buscado, enquanto o anterior não necessita percorrer essa lista, somente iniciá-la.

```
Type* GetRecordVarType(Record* rec, std::string id);
```

Listagem 4.77: Método para recuperação do tipo de estrutura.

Na Listagem 4.78, está um exemplo para a recuperação do tipo de uma variável de uma estrutura. É utilizada a linguagem *Middle* para descrever duas produções: uma que recupera o tipo da primeira variável de uma estrutura e outra que recupera o tipo de campos internos de uma estrutura. A primeira produção tem sua ação semântica iniciada na linha 2, com a aplicação do método `GetRecord` para recuperar as informações da estrutura armazenadas na tabela de símbolos. O retorno desse método é

armazenado em um atributo do nodo "`record_exp`" para ser utilizado posteriormente. Na linha 3, é empregado o método `GetBaseRecordVarType` para conseguir o tipo da variável armazenada na estrutura. O primeiro parâmetro é o atributo de `record_exp` em que informações da estrutura foram armazenadas. O segundo atributo é o identificador da variável da estrutura que é buscada. O retorno do método é utilizado para preencher o atributo de tipo do nodo "`record_exp`". A segunda produção inicia na linha 5, copiando o atributo das informações da estrutura do nodo "`record_exp1`" para o nodo "`record_exp`". Na linha seguinte é empregado o método `GetRecordVarType` para recuperar o tipo de um campo interno da estrutura. Assim como na produção anterior, é aplicado o atributo de `record_exp` contendo as informações da estrutura como primeiro parâmetro e o identificador do campo como segundo parâmetro. O retorno do método também é utilizado para preencher o atributo de tipo do nodo "`record_exp`".

```
1 record_exp ::= "recordexp" id1 id2 where{
2     record_exp->record = cb->GetRecord(id1);
3     record_exp->type = cb->GetBaseRecordVarType(record_exp->record, id2);
4   };
5 record_exp ::= record_exp1 id where{
6     record_exp->record = record_exp1->record;
7     record_exp->type = cb->GetRecordVarType(record_exp->record, id);
8   };
```

Listagem 4.78: Exemplo da utilização do método de recuperação de tipo de uma variável de uma estrutura.

O método `GetFunctionType` (Listagem 4.79) é responsável por recuperar o tipo do retorno de uma função declarada. Seu único parâmetro são as informações fornecidas pelo método `GetFunction`.

```
Type* GetFunctionType(Function* fun);
```

Listagem 4.79: Método para recuperação do tipo de retorno de função.

Na Listagem 4.80, está o exemplo da recuperação do tipo de uma função. É utilizada a linguagem *Middle* para descrever duas produções responsáveis pela chamada de função, uma em que a chamada da função possui argumentos que serão utilizados e outra em que a chamada não possui argumentos. A ação semântica de ambas produções nesse caso é idêntica. É empregado o método `GetFunctionType` para conseguir o tipo de valor produzido pela função. Para aplicar o parâmetro do método é usado o outro método `GetFunction` que utiliza o identificador da função para recuperar suas informações armazenadas na tabela de símbolos. Ao fim, o tipo retornado pelo método empregado é utilizado para preencher o atributo de tipos do nodo "`exp`".

```
1 exp ::= "funccall" id actualarg where{
2     exp->type = cb->GetFunctionType(cb->GetFunction(id));
3   };
4 exp ::= "funccall" id where{
5     exp->type = cb->GetFunctionType(cb->GetFunction(id));
6   };
```

Listagem 4.80: Exemplo da utilização do método de recuperação do tipo de uma função.

`GetBaseClassVarType` (Listagem 4.81) é o método que realizará a aquisição do tipo de um membro da classe requerida. Dessa maneira, é preciso que sejam indicadas como parâmetros as informações da classe que são fornecidas pelo método `GetClass` e o identificador do membro da classe.

```
Type* GetBaseClassVarType(Class* cls, std::string id);
```

Listagem 4.81: Método para recuperação do tipo da clase mais interna.

O método `GetClassVarType` (Listagem 4.82) é responsável pela obtenção de um membro de uma classe que seja membro de outra classe. Assim sendo, esse método funciona de forma similar aos métodos de recuperação de tipo de estrutura, percorrendo previamente uma lista dos membros para possibilitar o retorno do membro buscado. Os parâmetros necessários são as informações fornecidas pelo método `GetClass` e o identificador do membro da classe buscado.

```
Type* GetClassVarType(Class* cls, std::string id);
```

Listagem 4.82: Método para recuperação do tipo de classe.

Na Listagem 4.83, está um exemplo para a recuperação do tipo de um membro de uma classe. É utilizada a linguagem *Middle* para descrever duas produções: uma que recupera o tipo do primeiro membro de uma classe e outra que recupera o tipo dos membros internos de uma classe. A primeira produção tem sua ação semântica iniciada na linha 2, com a aplicação do método `GetClass` para recuperar as informações da classe armazenadas na tabela de símbolos. O retorno desse método é armazenado em um atributo do nodo "`class_exp`" para ser utilizado posteriormente. Na linha 3, é empregado o método `GetBaseClassVarType` para conseguir o tipo membro definido na classe. O primeiro parâmetro é o atributo de `class_exp`, em que informações da classe foram armazenadas. O segundo atributo é o identificador do membro da classe que é buscado. O retorno do método é utilizado para preencher o atributo de tipo do nodo "`class_exp`". A segunda produção inicia na linha 5, copiando o atributo das informações da classe do nodo "`class_exp1`" para o nodo "`class_exp`".

Na linha seguinte é empregado o método `GetClassVarType` para recuperar o tipo de um membro interno da classe. Assim como na produção anterior, é aplicado o atributo de `class_exp` contendo as informações da classe como primeiro parâmetro e o identificador do membro como segundo parâmetro. O retorno do método também é utilizado para preencher o atributo de tipo do nodo "`class_exp`".

```
1 class_exp ::= "classexp" id1 id2 where{
2     class_exp->clss = cb->GetClass(id1);
3     class_exp->type = cb->GetBaseClassVarType(class_exp->clss, id2);
4   };
5 class_exp ::= class_exp1 id where{
6     class_exp->clss = class_exp1->clss;
7     class_exp->type = cb->GetClassVarType(class_exp->clss, id);
8   };
```

Listagem 4.83: Exemplo da utilização do método de recuperação de tipo de um membro de uma classe.

Além dos métodos para recuperação de tipo de variáveis, são utilizados métodos estáticos da classe `Type` para criar os tipos primitivos suportados pela infraestrutura de geração de código. Para a aplicação desses métodos não são necessários parâmetros e cada método foi nomeado de acordo com o tipo que é criado, facilitando sua utilização. Alguns dos métodos que realizam essa criação estão descritos na Listagem 4.84. O método `GetInt` cria o tipo usado por um número de 32 bits, enquanto o método `GetLong` cria o tipo empregado por número inteiro de 64bits. O método `GetFloat` cria o tipo de um número de ponto fluante de precisão simples, enquanto o `GetDouble` cria o tipo de um número de ponto flutuante de precisão dupla. Os métodos `GetIntPtr`, `GetLongPtr`, `GetFloatPtr` e `GetDoublePtr` são utilizados para criar os tipos de ponteiro.

```
static Type* GetInt()
static Type* GetLong()
static Type* GetFloat()
static Type* GetDouble()
static Type* GetIntPtr()
static Type* GetLongPtr()
static Type* GetFloatPtr()
static Type* GetDoublePtr()
```

Listagem 4.84: Métodos para criação dos tipos primitivos.

Na Listagem 4.85, está um exemplo da criação de um objeto da classe `Type` para a verificação de tipos de expressões. É utilizada a linguagem *Middle* para descrever a produção de um literal inteiro. Na linha 2, é utilizado o método `GetLong` para criar o tipo de inteiro de 64 bits e preencher o atributo de tipo do nodo "`exp`".

Os métodos para recuperação de tipo, apesar de não criarem instruções ou afetarem diretamente a infraestrutura de componentes, são necessários em linguagens que

```
1 exp ::= integer_n where{
2     exp->type = component::Type::GetLong();
3   };
```

Listagem 4.85: Exemplo da utilização do método de criação de tipo um primitivo.

utilizam mais de um tipo de literal ou variável. Isso se dá por causa da necessidade do tipo em definir o operador binário ou unário de uma operação.

## 4.8 Conclusão

Neste capítulo, foi apresentada a infraestrutura de componentes proposta e implementada nesta dissertação. Com ela, é possível simplificar o trabalho de implementação de um compilador para uma linguagem de programação imperativa, principalmente se utilizado em conjunto com as linguagens do ambiente descrito no Capítulo 3.

O uso dos componentes que realizam o cálculo do endereço de uma variável permite que o implementador do compilador canalize sua atenção para outros detalhes referentes à linguagem sem se preocupar com as minúcias de cada variável e tipo. De igual forma, ganha-se tempo com o uso dos componentes para declaração de variável, pois encapsulam os detalhes da criação e subsequente alocação de uma variável, tornando necessários somente o identificador e tipo da variável que será declarada. Ainda há componentes referentes a declaração de tipos de dados específicos que facilitam, por exemplo, a declaração de arranjos dinâmicos. Os componentes para geração de comandos de desvio e repetição também simplificam a implementação do compilador, visto que encapsulam as ações semânticas dessas construções e retiram a necessidade do implementador de preocupar-se com detalhes como a lista de desvios e condições. Os componentes para gerência de escopos, em conjunto com o ambiente de desenvolvimento de compiladores deste trabalho, simplificam a manipulação do escopo, uma vez que só requer a utilização de palavras-chave na gramática da linguagem que são utilizadas durante o caminhamento pós-ordem.

Os outros componentes, como a atribuição, o sequenciador de retorno e aplicação de expressão também são simplificadas em sua implementação, visto que encapsulam diversas ações semânticas necessárias para cada construção. A atribuição, por exemplo, encapsula os detalhes do endereço de armazenamento e valor armazenado. O sequenciador de retorno encapsula os detalhes do endereço dos valores utilizados para retorno de uma função. E a aplicação de expressão também encapsula os detalhes do endereço dos valores utilizados e permite, por meio de um método da infraestrutura, a escolha de qual operação é aplicada pela expressão.

Dessa forma, os componentes cumprem sua proposta, tornando o desenvolvimento do compilador mais simples e fácil, encapsulando as ações semânticas necessárias para realizar cada uma de suas compilações.

Além disso, é possível afirmar que os componentes podem ser utilizados para as construções de diversas linguagens de programação imperativas, uma vez que os componentes encapsulam as ações semânticas para essas construções. Infelizmente, construções como o *for* da linguagem *Algol 60* não é compilado pelos componentes da infraestrutura aqui apresentada, sendo necessário a criação de novos componentes para essa construção.

# Capítulo 5

# Avaliação do Ambiente

O ambiente de desenvolvimento de compiladores proposto e implementado nesta dissertação foi criado com o objetivo de prover facilidade de uso e legibilidade das linguagens de implementação das diversas fases do compilador. Nesse sentido, a infraestrutura de geração de código também possui como objetivo a generalidade da biblioteca de componentes, projetada para funcionar com o maior número de linguagens de programação imperativa possível. São baseadas questões que será discutida, neste capítulo, a avaliação do ambiente realizado nesta dissertação. Na Seção 5.1, é apresentado um estudo de caso para demonstrar a legibilidade e facilidade de uso do ambiente de desenvolvimento de compiladores em conjunto com a infraestrutura de programação. Esse estudo de caso é realizado utilizando a linguagem *Small*, criada pelo autor, que contêm como recursos: variáveis do tipo inteiro de 64 *bits* e do tipo ponto flutuante de precisão dupla; expressão e comando condicional; comando de repetição; funções com passagem de parâmetro por valor e as expressões suportadas pela infraestrutura de geração de código. Na Seção 5.2, é discutida a generalidade dos componentes implementados para a infraestrutura de geração de código.

## 5.1 Legibilidade e Facilidade de Uso

Esta seção apresenta um estudo de caso cujo objetivo é demonstrar a legibilidade e facilidade de uso do ambiente de desenvolvimento de compiladores disponibilizado neste trabalho. A partir do programa da Listagem 5.1, escrito na linguagem *Small*, criada pelo autor, esse estudo de caso mostra como construir um compilador para essa linguagem usando o ambiente oferecido nessa dissertação.

```
1 program {
2   float x;
3   x = 1 + 1.0;
4 }
```

Listagem 5.1: Exemplo de programa na linguagem *Small*.

### Descrição *AST*

Na Listagem 5.2, está a descrição na linguagem *AST* da sintaxe abstrata de *Small*. Com essa descrição, é possível ver quais construções a linguagem possui. Dentre elas, pode-se destacar: as funções; os comandos condicionais; o comando de repetição e os literais inteiro e de ponto flutuante. Na Seção `@attributes` das linhas 1-3, são declarados atributos dos nodos `type` e `exp` para realizar a verificação de tipos, os atributos utilizados pelos componentes da infraestrutura já são declarados automaticamente em todos nodos da linguagem. Na Seção `@leaves` das linhas 4-7, são definidos os tipos C++ dos tokens folhas da linguagem. Dessa forma, é definido que o token `integer_n` é do tipo `int64_t`, o token `float_n`, do tipo `double`, e o token id emprega o tipo `std::string`. O resto da descrição da linguagem contém produções de gramática para descrever a árvore de sintaxe abstrata. A partir dessa árvore, é possível saber quais construções a linguagem possui de uma forma mais simples que utilizando a sua gramática concreta, uma vez que seus detalhes sintáticos são ignorados nessa linguagem.

### Descrição *Front*

A Listagem 5.3 contém um excerto da descrição do *Front* da linguagem *Small*, que gera os analisadores léxico e sintático para criação da árvore de *parsing* e as regras de transformação dela na árvore de sintaxe abstrata. A Figura 5.1, mostra a árvore de *parsing* para o programa na linguagem *Small* apresentado na listagem 5.1 e a Figura 5.2 ilustra a sua árvore de sintaxe abstrata. Nessa árvore de *parsing*, cada nodo é montado com seus devidos filhos até chegar aos nodos folhas da árvore. Devido à ordem de precedência dos operadores, o nodo de "`exp`" tem como filho um nodo de "`orexp`" que tem como filho um nodo de "`andexp`" e assim sucessivamente até o nodo de "`addexp`". Esse nodo tem como filhos um nodo de "`addexp`", um *token* de `+` e um nodo de "`term`". Ambos nodos continuam se expandindo, até terem como filho um *token* considerado folha da árvore.

A árvore de sintaxe abstrata (Figura 5.2) obtida via a da transformação da árvore de *parsing*, torna a árvore mais compacta e simples. A principal alteração percebida é o achatamento do caminho do nodo "`exp`", possuindo três filhos na árvore de sintaxe abstrata e apresentando um caminho menor até os nodos folha. Outra alteração

```
1  @attributes
2  type { component::Type* type; } ;
3  exp { component::Type* type; } ;
4  @leaves
5  int64_t integer_n;
6  double float_n;
7  std::string id;
8  @nodes
9  prog ::= "program" decl fnt block resetsymboltable
10 | "program" decl block resetsymboltable
11 | "program" fnt block resetsymboltable
12 | "program" block resetsymboltable ;
13 fnt ::= fnt_header block  | fnt1 fnt2 ;
14 fnt_header ::= "function" type id formalpar
15 | "function" type id
16 | "procedure" id formalpar | "procedure" id ;
17 formalpar ::= "var" type id  | formalpar1 formalpar2 ;
18 block ::= "dblock" startblock decl stm finishblock
19 | "ublock" startblock stm finishblock ;
20 stm ::= "assign" acc_exp exp  | "output" exp
21 | "ifthenelse" exp block1 block2  | "ifthen" exp block
22 | "proccall" id actualarg  | "proccall" id
23 | "return" exp | "whiledo" exp block
24 | "block" block | stm1 stm2 ;
25 actualarg  ::= "deref_call" exp | actualarg1 actualarg2 ;
26 decl ::= type_name id | id | decl1 decl2 ;
27 type_name ::= "integer" | "float" ;
28 acc_exp ::= id ;
29 exp ::= "ifexp" exp1 exp2 exp3 | "and" exp1 exp2
30 | "or" exp1 exp2 | "equals" exp1 exp2
31 | "notequals" exp1 exp2 | "lesser" exp1 exp2
32 | "greater" exp1 exp2 | "lesserquals" exp1 exp2
33 | "greaterequals" exp1 exp2 | "add" exp1 exp2
34 | "sub" exp1 exp2 | "mult" exp1 exp2
35 | "div" exp1 exp2 | "rem" exp1 exp2
36 | "not" exp1 | "neg" exp1
37 | "funccall" id actualarg | "funccall" id
38 | "deref" acc_exp | "castfp" exp1
39 | integer_n | float_n ;
```

Listagem 5.2: Descrição da árvore de sintaxe abstrata de *Small* na linguagem *AST*.

relevante para a árvore de sintaxe abstrata é o aparecimento dos símbolos terminais em negrito que são utilizados, durante os caminhamentos da árvore, para acionar os componentes da infraestrutura de geração de código referentes ao escopo e à tabela de símbolos. Todas alterações da árvore de sintaxe abstrata foram descritas na linguagem *Front* (Listagem 5.3) por meio das regras de transformação e auxiliadas pelas definições de domínio. Por exemplo, a regra `["add" addexp term]` descreve como é alterada a produção concreta `addexp "+" term`, e a definição do domínio na linha 49 descreve que os terminais `addexp` e `term` devem ser substituídos por `exp` na construção da árvore de sintaxe abstrata. As regras de transformação que somente repetem o não-terminal indicam que não deverá ser produzido nenhum nodo, achatando a árvore de sintaxe abstrata.

Na última seção da Listagem 5.3, está a descrição léxica da linguagem *Small*. As

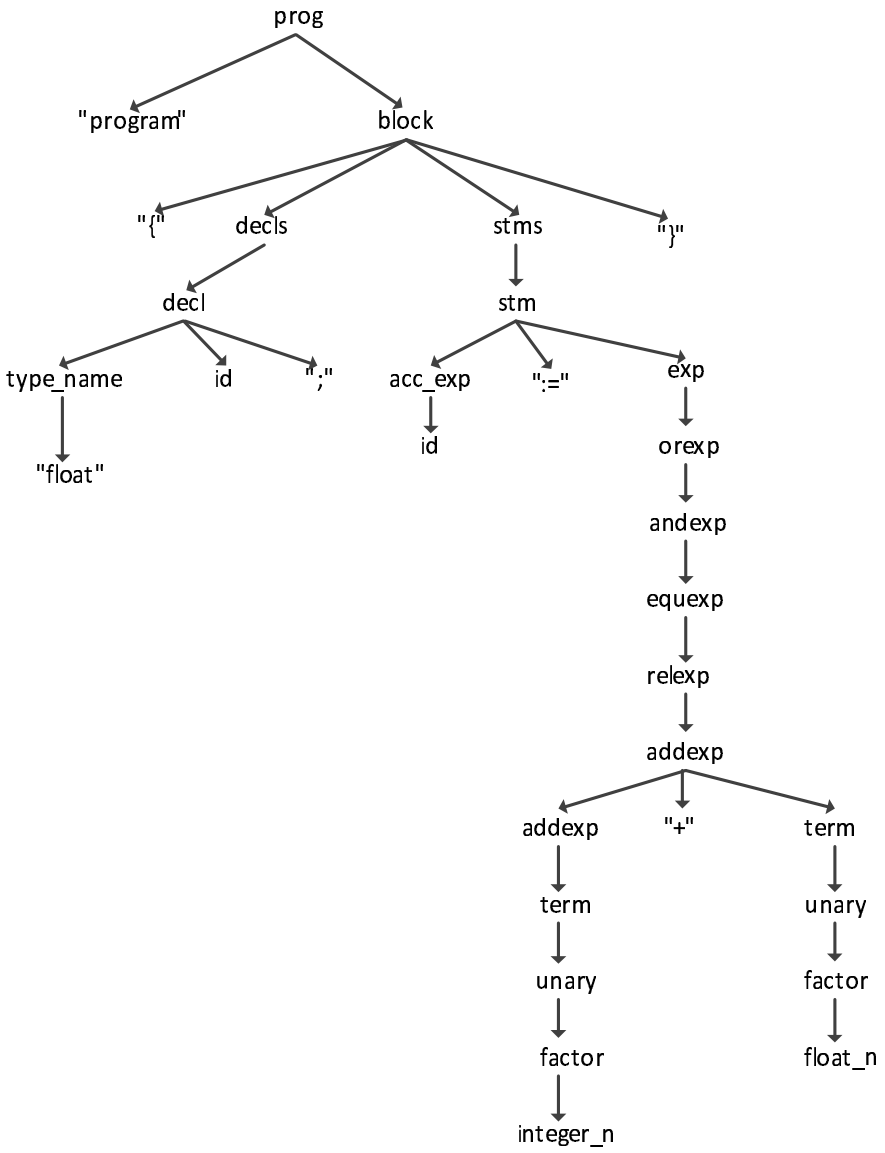


Figura 5.1: Árvore de *parsing* do programa **{float x; x = 1 + 1.0}**.

produções de `UNIT` definem quais são os símbolos terminais da gramática. Portanto, `id`, `integer_n` e `float_n` são considerados terminais da gramática e são definidos nessa seção, `id` é determinado como qualquer combinação de letras maiúsculas ou minúsculas; `integer_n` é definido como a combinação de qualquer número natural. Por fim, `float_n` é definido como a combinação de dois números naturais separados por um ponto.

Devido ao foco da linguagem *Front* na geração do programa de *parsing* e sua semelhança com as gramáticas de descrição de sintaxe concreta, é possível dizer que essa linguagem de descrição é simples de se utilizar e legível. O ponto em que a linguagem *Front* se diferencia de uma gramática de sintaxe concreta é a possibilidade

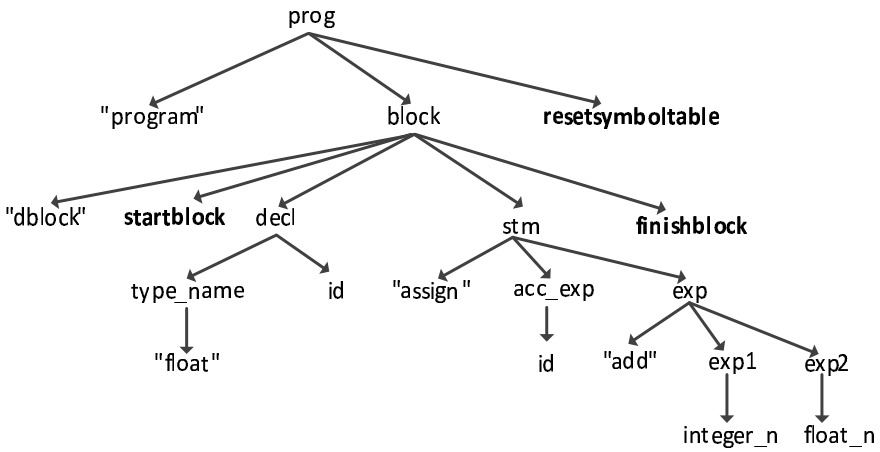


Figura 5.2: Árvore de sintaxe abstrata da construção **{float x; x = 1 + 1.0}**.

de descrever a configuração do nó na árvore de sintaxe abstrata. Como isso é realizado após cada alternativa de produção da gramática, não interferindo na parte concreta da linguagem, a legibilidade e facilidade de uso da mesma é mantida.

### Descrição Primeiro *Middle*

Após a transformação da árvore de *parsing* para a árvore de sintaxe abstrata, é utilizada a linguagem *Middle* do ambiente de desenvolvimento para que possa haver o processamento da declaração de variáveis e funções do programa sendo compilado. Na Listagem 5.4, está o excerto da descrição em *Middle* que gera o programa utilizado para processar a declaração de variáveis e na Figura 5.3, a árvore de sintaxe abstrata resultante de sua execução. Nessa descrição e nas próximas, são utilizados componentes e métodos da infraestrutura de geração de código para auxiliar em suas ações. Portanto, são utilizadas suas estruturas internas para compilação como a tabela de símbolos. Na segunda linha, está a produção inicial da *AST*, descrevendo que é visitado o nodo `block`, sendo finalizada com a palavra-chave da infraestrutura de geração de código `resetsymboltable`, que restabelece a tabela de símbolos interna dos componentes para que ela seja usada na próxima fase. De forma similar, a produção da linha 4 descreve que, ao entrar nesse nodo, devido a palavra-chave `startblock`, a tabela de símbolos é alterada, abrindo um nível. Isso é seguido pelas visitas dos nodos de `decl` e `stm`. Ao fim da execução do nodo, devido ao uso da palavra-chave `finishblock`, a tabela de símbolos é novamente alterada, fechando um nível. A produção da linha 6 descreve a declaração de uma variável e uma transformação do nodo da *AST* visitando, antes de executada a ação semântica, o nodo `type_name`. A ação semântica da declaração utiliza o componente `VariableDeclare` para instalar na tabela de sím-

```
1  @grammar
2  prog ::= ...
3   | "program" block : ["program" block resetsymboltable ]
4  ...
5  block ::= "{" decls stms "}" : ["dblock" startblock decl stms finishblock ]
6  stms ::= stms stm ";": [ stms stm ]
7   | stm ";" : stm ;
8  stm ::= acc_exp ":=" exp  : [ "assign" acc_exp exp ]
9   | "if" exp "then" block : ["ifthen" exp block]
10 ...
11 decls ::= decls decl : [ decls decl]
12  | decl : decl ;
13 decl ::= type_name id ";" : [ type_name id ] ;
14 type_name ::= "integer"
15  | "float"
16 acc_exp ::= id ;
17 exp ::= orexp "?" exp ":" exp  : ["ifexp" orexp exp exp]
18  | orexp : orexp ;
19 orexp ::= orexp "||" andexp : ["or" orexp andexp]
20  | andexp : andexp ;
21 andexp ::= andexp "&&" eqexp : ["and" andexp eqexp]
22  | eqexp : eqexp ;
23 eqexp ::= eqexp "=" relexp : ["equals" eqexp relexp]
24  | eqexp "!=" relexp : ["notequals" eqexp relexp]
25  | relexp : relexp ;
26 relexp ::= relexp "<" addexp : ["lesser" relexp addexp]
27  | relexp ">" addexp : ["greater" relexp addexp]
28  | relexp "<=" addexp : ["lesserquals" relexp addexp]
29  | relexp ">=" addexp : ["greaterequals" relexp addexp]
30  | addexp : addexp ;
31 addexp ::= addexp "+" term : ["add" addexp term]
32  | addexp "-" term : ["sub" addexp term]
33  | term : term ;
34 term ::= term "*" unary : ["mult" term unary]
35  | term "/" unary : ["div" term unary]
36  | term "%" unary : ["rem" term unary]
37  | unary : unary ;
38 unary ::= "!" unary : ["not" unary]
39  | "-" unary : ["neg" unary]
40  | factor : factor ;
41 factor ::= "(" exp ")" : exp
42  | id "(" actualargs ")" : [ "funccall" id actualargs ]
43  | id "(" ")" : [ "funccall" id ]
44  | acc_exp : [ "deref" acc_exp]
45  | integer_n
46  | float_n ;
47 @domains
48 stms : stm;
49 decls : decl;
50 orexp, andexp, eqexp, relexp, addexp, term, unary : exp;
51 @lexical
52 UNIT ::= id
53  | integer_n
54  | float_n ;
55 letter === "a" ... "z" | "A" ... "Z" ;
56 id ::= letter id | letter ;
57 digit === "0" ... "9" ;
58 numeral ::= digit numeral | digit ;
59 integer_n ::= numeral;
60 float_n ::= numeral "." numeral;
```

Listagem 5.3: Excerto da descrição de *Small* na linguagem *Front*.

bolos o tipo e identificador da variável sendo declarada. O retorno do componente é utilizado para preencher o atributo de declaração do nodo `decl`. A transformação do nodo da *AST* é feita repetindo a produção, mas retirando o nodo `type_name` que não é mais necessário. A produção da linha 9 só descreve os nodos que serão visitados, não descrevendo nenhuma ação semântica. Na linha 10, está a produção do *token* de tipo inteiro, dessa forma, não há nodos para serem visitados e sua ação semântica se refere ao preenchimento do atributo de tipo do nodo `type_name` com o devido objeto da classe `Type`. O mesmo ocorre na linha 13 para a produção do *token* do tipo de ponto flutuante.

```
1  @pass1
2  prog ::= "program" block resetsymboltable;
3  ...
4  block ::= "dblock" startblock decl stm finishblock ;
5  ...
6  decl ::= type_name id => decl ::= id where{
7      decl->decl_attr = cb->VariableDeclare(id, type_name->type);
8    }
9    | decl1 decl2;
10 type_name ::= "integer" where{
11     type_name->type = component::Type::GetLong();
12   };
13 type_name ::= "float" where{
14     type_name->type = component::Type::GetDouble();
15   };
```

Listagem 5.4: Excerto da declaração de variáveis de *Small* na linguagem *Middle*.

A árvore de sintaxe abstrata do programa da Listagem 5.1 após a primeira transformação da linguagem *Middle* (Figura 5.3) foi alterada de forma que, o nodo "`decl`", após a declaração da variável, não necessita mais do tipo da variável declarada. Como resultado, o nodo "`type_name`" é retirado da árvore de sintaxe abstrata, tornando-a mais simples para os subsequentes caminhamentos. A parte alterada está destacada em azul. Exceto essa alteração, a árvore permanece inalterada para os subsequentes módulos.

### Descrição Segundo *Middle*

A segunda descrição na linguagem *Middle* é utilizada para gerar o programa que realiza a verificação dos tipos das variáveis e transforma a árvore de sintaxe abstrata permitindo a realização de coerção de expressões do tipo inteiro para expressões do tipo ponto flutuante. Na Listagem 5.5, está o excerto dessa descrição com as definições necessárias para realizar: a verificação de tipo de uma variável; a verificação e transformação da expressão de adição e o preenchimento do atributo de tipo do nodo de expressão contendo um literal inteiro. A Figura 5.4 exibe a árvore de sintaxe abs-

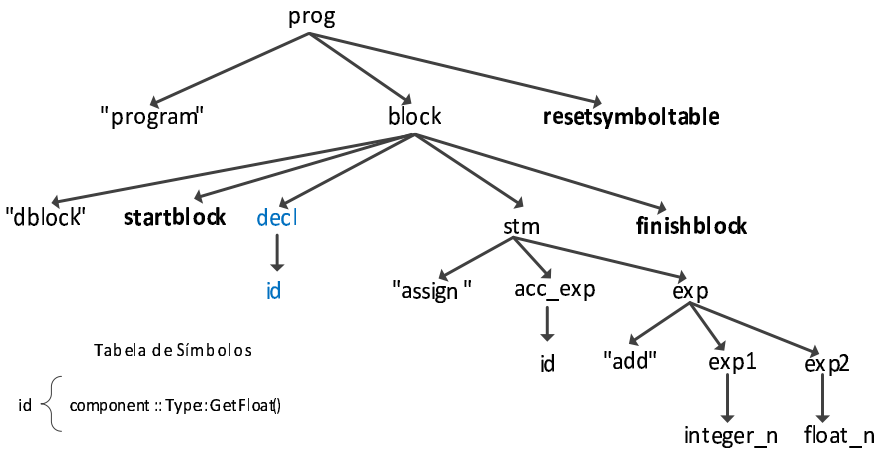


Figura 5.3: Árvore de sintaxe abstrata após primeira transformação de *Middle*.

trata resultante da execução desse programa. Na terceira linha da Listagem 5.5 está a regra de produção de acesso a uma variável, em sua ação semântica é utilizado o método `GetVariableType` da infraestrutura para conseguir o tipo da variável que está armazenado na tabela de símbolos e preencher o atributo do nodo `acc_exp`. A produção seguinte, na linha 6, é a expressão de adição. Sua ação semântica é complexa, sendo necessário comparar os atributos de tipo dos nodos `exp1` e `exp2` para decidir se é necessário realizar a coerção de uma das expressões. Caso não seja necessário, o tipo da expressão avaliada é igual ao do nodo `exp1`. Caso contrário, nas linhas 10 e 12, é definido a transformação da árvore de sintaxe abstrata através da criação de um novo nodo. Em ambos os casos é criado o nodo de coerção, diferindo em qual nodo, `exp1` ou `exp2`, será utilizado. A última produção do excerto é a expressão de um literal inteiro. Sua ação semântica é simplesmente preencher o atributo de tipo do nodo `exp` com o devido objeto da classe `Type` da infraestrutura de geração de código.

A árvore de sintaxe abstrata resultante da execução do segundo programa de *Middle* (Figura 5.4) altera a árvore calculando atributos de tipo das expressões e modificando a árvore para realizar a coerção do inteiro para um número de ponto flutuante no nodo "`exp1`". Novamente a parte modificada é exibida em azul, entretanto, por questões de legibilidade não são colocados os atributos calculados na figura. Após essas alterações a árvore é utilizada nos módulos subsequentes.

Devido ao formato das produções ser semelhante ao empregado pelas produções de *Front*, é possível dizer que a legibilidade das descrições de *Middle* é boa, uma vez que esse formato também é similar ao usado para descrição de gramáticas de sintaxe concreta. Além disso, o formato utilizado para as transformações de *AST* é tão simples e legível quanto o usado para as próprias produções. As transformações realizadas para

```
1  @pass2
2  ...
3  acc_exp ::= id where{
4      acc_exp->type = cb->GetVariableType(id);
5    };
6  exp ::= "add" exp1 exp2 where{
7      if(exp1->type->compare(exp2->type))
8        exp->type = exp1->type;
9      else{
10       if(exp1->type->isIntegerTy())
11         exp1 = [ "castfp" exp2 ]
12       else
13         exp2 = [ "castfp" exp1 ]
14       exp->type = component::Type::GetFloat();
15     }
16   };
17 exp ::= integer_n where{
18     exp->type = component::Type::GetLong();
19   };
```

Listagem 5.5: Excerto da verificação de tipos de *Small* na linguagem *Middle*.

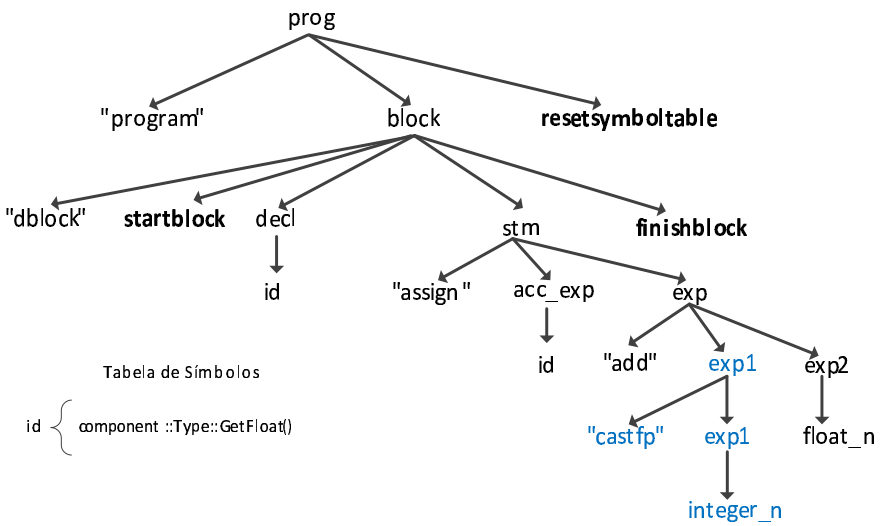


Figura 5.4: Árvore de sintaxe abstrata após segunda transformação de *Middle*.

alterar as produções só precisam das informações disponíveis pela própria produção, posto que transformam uma produção em outra como indicado por uma seta (=>). As transformações realizadas dentro do código das ações semânticas também são simples, dado que a criação de um novo nodo só necessita que o mesmo seja descrito com suas variáveis, terminais e não-terminais, entre colchetes, podendo ser atribuído a uma variável da produção cuja ação semântica está sendo executada.

### Descrição *Back*

Após a verificação de tipos, a árvore de sintaxe abstrata está preparada para utilização pelo programa da linguagem *Back* que realiza a geração de código intermediário. Na

Listagem 5.6 está um excerto desse programa, contendo as produções para realizar a geração de código das seguintes construções: declaração de variável; concatenação das listas de código de comandos; atribuição de uma expressão à uma variável; comando condicional; expressão de adição e expressão do literal inteiro. A primeira produção está na linha 2 e utiliza o componente `VariableAlloc` para realizar a alocação da variável declarada pelo programa da Listagem 5.4. Como descrito no Capítulo 4, o componente gera a instrução de alocação e a insere na lista de instruções do atributo que é retornado pelo componente. Esse atributo é, então, utilizado para preencher o atributo de declaração do nodo `decl`. Para a atribuição é aplicado o componente `BuildAssignementStatement` que, como os outros, gera sua instrução e a insere na lista de instruções retornada pelo atributo usado para preencher o atributo de comando do nodo `stm`. De igual forma ocorre com o componente `BuildConditionalStatement` empregado na produção do comando condicional da linha 9. A concatenação das listas de código de comando é realizada pelo componente `BuildSequence` na produção da linha 12. De maneira similar ao componente anterior, o componente empregado retorna o resultado da concatenação das listas como atributo e esse retorno é utilizado para preencher o atributo de comando do nodo `stm`. A expressão de adição é gerada pelo componente `Apply` na produção da linha 17. A diferença para os anteriores é que a instrução gerada não é somente agregada a lista de instruções, mas é também inserida no campo de endereço de instrução do atributo para utilização de seu valor por outras expressões. Entretanto, assim como ocorre com os demais componentes, o atributo retornado é utilizado para preencher o atributo de expressão do nodo `exp`. A expressão de literal inteiro funciona da mesma maneira que a expressão anterior, utilizando o componente `LoadInteger` e seu retorno para preencher o atributo de expressão do nodo `exp`.

A semântica de *Small* requer que, antes de encerrar a geração de código, seja criada a função principal que envolve o bloco das instruções. Isso está demonstrado no excerto da Listagem 5.7. A primeira ação semântica necessária é a declaração da função principal, que ocorre na linha 3 com o componente `3`. Como essa função não é utilizada por uma classe, não é necessário armazenar seu atributo retornado. Nas próximas linhas a função principal é construída. O componente `BuildFunction` é utilizado para inserir uma lista de código à uma função, entretanto, é necessário construir a instrução de retorno da função para que ela encerre a execução corretamente, por isso é necessário utilizar o componente `BuildSequence` com o atributo de comando do nodo `block`, que contém a lista de instrução do programa compilado e o componente `BuildReturnStatement` que cria a instrução de retorno necessária. Após o retorno do componente `BuildSequence`, a função principal é construída com o código interme-

```
1  ...
2  decl ::= id {
3      decl->decl_attr = cb->VariableAlloc(id);
4    };
5  ...
6  stm ::= "assign" acc_exp exp{
7      stm->stm_attr =
           cb->BuildAssignmentStatement(acc_exp->exp_attr,exp->exp_attr);
8    }
9    | "ifthen" exp block{
10     stm->stm_attr = cb->BuildConditionalStatement(exp->exp_attr,
           block->stm_attr, cb->BuildSkip());
11   }
12   | stm1 stm2 {
13     stm->stm_attr = cb->BuildSequence(stm1->stm_attr, stm2->stm_attr);
14   };
15 ...
16 exp ::=   ...
17   | "add" exp1 exp2{
18     exp->exp_attr = cb->Apply(component::GetBinOpe("add",
           component::Type::GetLong()), exp1->exp_attr, exp2->exp_attr);
19   }
20   | integer_num{
21     exp->exp_attr = cb->LoadInteger(integer_num,
           component::Type::GetLong());
22   }
23 ...
```

Listagem 5.6: Excerto da geração de código de *Small* na linguagem *Back*.

diário gerado pelos componentes. Na linha 7 está o método `GenLLVM` responsável pela transformação do código intermediário em bytecode LLVM passível de ser executado.

```
1  prog ::= ...
2      | "program" block resetsymboltable{
3        cb->BuildFunctionHeader("main", component::Type::GetInt());
4        prog->stm_attr = cb->BuildFunction(cb->GetFunction("main"),
5            cb->BuildSequence(block->stm_attr,
6            cb->BuildReturnStatement( cb->LoadInteger(0,
                 component::Type::GetInt()))));
7        cb->GenLLVM(prog->stm_attr);
8      };
9  ...
```

Listagem 5.7: Outro excerto da geração de código de *Small* na linguagem *Back*.

Devido ao formato das produções semelhante ao utilizado pelas produções de *Front*, assim como ocorre com a linguagem *Middle*, pode-se dizer que a legibilidade de *Back*, no que tange a descrição da linguagem, é boa, uma vez que é um formato similar ao utilizado para descrição de gramáticas de sintaxe concreta e, portanto, de fácil leitura. A legibilidade das ações semânticas depende do código utilizado pelo implementador. Os componentes da infraestrutura de geração de código foram criados focando a legibilidade, por conseguinte, são utilizados nomes que buscam evocar as ações que serão realizadas, tornando-os mais simples de usar. Além disso, os com-

ponentes encapsulam diversos passos de ação semântica necessários para realizar a compilação das construções, sendo possível dizer que eles melhoram a legibilidade ao trocar diversos passos pelo uso de um componente.

## 5.2 Generalidade dos Componentes

As construções das linguagens de programação imperativas, apesar de diferirem em sua sintaxe, muitas vezes possuem ações semânticas similares [Watt & Findlay, 2004]. Dessa forma, o componente de geração de código para o comando condicional pode ser utilizado de igual maneira para essas linguagens, bastando alterar o reconhecimento da sintaxe da linguagem.

Outras construções, como a repetição, podem ter sua semântica diferentes entre linguagens distintas, como ocorre na construção de repetição *do while* encontrada nas linguagens C e Java que difere da construção de repetição *repeat until* da linguagem Pascal. Ambas construções são utilizadas para repetir um ou mais subcomandos dada uma condição, entretanto, a construção das linguagens C e Java encerra a execução da repetição quando a condição não for mais satisfeita. O contrário ocorre na construção da linguagem Pascal, em que a execução é encerrada somente quando a condição for satisfeita. O componente de geração de código da infraestrutura deste trabalho gera, inicialmente, código para a construção das linguagens C e Java. No entanto, para utilizar o componente da infraestrutura para gerar o código da construção de repetição da linguagem Pascal, só é necessário aplicar a operação de negação lógica à condição dessa repetição, como está exemplificado na linha 5 da Listagem 5.8.

```
1 stm ::= "whiledo" exp block{
2     stm->stm_attr = cb->BuildRepetitionStatement(cb->BuildSkip(),
          exp->exp_attr, cb->BuildSkip(), block->stm_attr);
3   }
4   | "repeatuntil" block exp{
5     stm->stm_attr = cb->BuildRepetitionStatement(cb->BuildSkip(),
          cb->Apply(component::GetUnOpe("!", exp->type), exp->exp_attr),
          cb->BuildSkip(), block->stm_attr, true);
6   };
```

Listagem 5.8: Exemplo da construção de repetição da linguagem Pascal.

Dessa forma, é possível perceber a generalidade dos componentes da infraestrutura de geração de código, uma vez que encapsulam as ações semânticas necessárias para as construções recorrentes das linguagens de programação imperativas. Além de permitirem, por meio da sua combinação, a implementação de variações das construções de linguagens de programação imperativas. Assim, é possível a utilização deste

trabalho para facilitar a criação de um compilador para uma linguagem que utilize essas construções.

Infelizmente, nem toda construção é passível de ser gerada pelos componentes da infraestrutura. Um exemplo disso é o *for* da linguagem *Algol 60* que necessita de componentes específicos para sua geração, não sendo possível a utilização dos componentes genéricos demonstrados neste trabalho.

## 5.3 Considerações Finais

O ambiente de desenvolvimento proposto utiliza quatro linguagens de descrição, todas com foco na legibilidade e facilidade de uso. Devido ao formato simplificado utilizado pelas produções das linguagens, e sua proximidade com as gramáticas concretas, é possível dizer que as linguagens de descrição atingem o foco estabelecido. As produções das linguagens sempre utilizam uma combinação de símbolos terminais e não-terminais definidos pelo implementador e um pequeno conjunto de palavras-chave da infraestrutura sendo, portanto, de fácil utilização para a descrição. Além disso, como as produções utilizadas pelas linguagens de descrição são similares entre si, a facilidade de utilização do sistema é aumentada, uma vez que, ao se aprender as regras de produção da linguagem *AST*, só é necessário aprender pequenas variações para as linguagens *Front* e *Middle* em que é necessário realizar transformação na árvore de sintaxe abstrata.

Além das linguagens de descrição, há os componentes da infraestrutura de geração de código que focam não somente na legibilidade e facilidade de uso, mas também na generalidade. A legibilidade e facilidade de uso são creditadas ao nome dado a cada componente, sendo utilizados nomes para evocar as construções que os componentes encapsulam. O encapsulamento das ações semânticas torna o código escrito para um compilador mais simples, também auxiliando na legibilidade da escrita de um compilador. A generalidade dos componentes se deve ao fato de serem implementadas as construções recorrentes das linguagens de programação imperativa, o que permite a utilização desses componentes em uma maior gama de linguagens.

# Capítulo 6

# Conclusão

Este trabalho apresenta um ambiente de desenvolvimento de compiladores que, em conjunto com uma infraestrutura de geração de código também apresentada neste trabalho, permite a elaboração de todos passos de um compilador, desde a definição da sintaxe concreta da linguagem até a geração de *bytecode* LLVM.

O ambiente de desenvolvimento permite a programação do compilador de uma linguagem de programação por meio de quatro linguagens de domínio específico, cada qual tratando de um interesse próprio. Essa separação dos interesses possibilita que o projetista da linguagem de programação possa concentrar-se na solução de um problema por vez, mantendo atenção em uma fase de cada vez e diminuindo a dispersão, resultando assim em maior produtividade e menos ocorrências de erro. Outro resultado importante do ambiente de desenvolvimento é a maior legibilidade das descrições das linguagens de programação, devido à semelhança das linguagens de descrição do ambiente com as gramáticas de sintaxe abstrata. Isso torna o ambiente mais simples e fácil de utilizar.

A infraestrutura de geração de código realiza o encapsulamento das ações semânticas necessárias para a geração do código, tornando o trabalho de implementação do compilador mais simples e fácil. Ao ser utilizado em conjunto com o ambiente de desenvolvimento, permite a elaboração de compiladores completos e legíveis, visto que a implementação das construções sem a utilização da infraestrutura de geração de código podem necessitar de ações semânticas complexas, o que dificulta sua compreensão e legibilidade. Com a infraestrutura de geração de código, cada construção é implementada utilizando um componente que encapsula os detalhes das ações semânticas, deixando o código mais claro. Dessa forma, o código se torna mais simples de ser lido e compreendido.

Os componentes da infraestrutura foram implementados, encapsulando a semân-

tica de comandos recorrentes das linguagens de programação imperativa, buscando uma maior generalidade. Dessa forma os componentes podem ser utilizados para implementação de compiladores de uma vasta gama de linguagens de programação imperativa.

No entanto, linguagens de programação estão sempre evoluindo, de maneira que, conforme surjam novos conceitos, a infraestrutura de geração de código pode ter que ser alterada para incorporá-los.

## 6.1 Contribuições do Trabalho

A principal contribuição deste trabalho é a criação de um ambiente de desenvolvimento de compiladores suportado por uma infraestrutura de geração de código, ambos com o foco na legibilidade do desenvolvimento, mas sem perda de poder de processamento. A legibilidade obtida no ambiente é fruto da separação dos interesses que ocorrem durante a implementação de um compilador e da semelhança das linguagens de descrição utilizada pelo ambiente com as gramáticas de sintaxe abstrata. Já a legibilidade da infraestrutura, é fruto do encapsulamento das ações semânticas necessárias para transformar o código fonte em *bytecode* LLVM.

O ambiente de desenvolvimento apresentado promove a separação dos interesses das fases de compilação, separando a entrada de dados da geração da árvore de sintaxe abstrata e cada fase subsequente. Com isso, melhora-se a atenção dispensada a cada fase sendo implementada, potencialmente melhorando a qualidade do código sendo criado. Isso se deve à separação do foco das linguagens de descrição utilizadas no ambiente de desenvolvimento, dado que cada linguagem possui uma determinada fase como foco de geração.

Por fim, ambos produtos do trabalho, o ambiente de desenvolvimento de compiladores e a infraestrutura de geração de código, foram implementados utilizando a linguagem de programação C++11, permitindo a pronta utilização dos sistemas descritos neste trabalho.

## 6.2 Trabalhos Futuros

A complexidade inerente a grandes linguagens de programação e a falta de alguns conceitos implementados na infraestrutura de geração de código, como a concorrência de *threads* e tratamento de exceções, inviabilizou um melhor estudo de caso utilizando linguagens de programação com essas características. A linguagem Java, por exemplo, pode ter um subconjunto de suas características implementadas pela infraestrutura de

geração de código, entretanto o tamanho e complexidade dessa linguagem dificultam sua implementação completa utilizando a infraestrutura de geração de código. Dessa maneira, seria interessante expandir a implementação da infraestrutura de geração de código de forma a possibilitar a implementação da linguagem Java em todos seus aspectos e características para poder comparar não somente o compilador, como o código gerado por outras implementações da linguagem.

O trabalho apresentado possui seu escopo limitado para definição e geração de código das linguagens imperativas. Isso foi feito para que o trabalho possuísse um alvo tangível de componentes para serem criados. Seria interessante avaliar o acréscimo de outros paradigmas de programação à infraestrutura, ou até mesmo, a realização de outras infraestruturas similares para os diversos paradigmas existentes.

# Apêndice A

# Linguagens do Sistema

Neste apêndice são apresentadas as linguagens utilizadas pelo sistema desenvolvido. Elas são quatro: *AST*, uma linguagem para descrição dos nodos da árvore de sintaxe abstrata empregados durante a compilação da linguagem pelo sistema; *Front*, uma linguagem para a descrição de sintaxe da linguagem e sua transformação para a árvore de sintaxe abstrata; *Middle*, uma linguagem para descrição de transformações da árvore de sintaxe abstrata para realização das etapas distintas do compilador como a declaração de variáveis e verificação de tipos e, por fim, *Back*, uma linguagem para descrição das ações semânticas necessárias para geração do código alvo.

## A.1 Linguagem *AST*

A Listagem A.1 contém a gramática da linguagem *AST* cujo objetivo é descrever as classes utilizadas como nodos da árvore de sintaxe abstrata utilizada pelo sistema. A linguagem é dividida em quatro seções: declarações, atributos, folhas e nodos. A primeira seção permite a declaração de tipos C++ para utilização na seção de atributos. A segunda seção permite a declaração de atributos de cada nodo em um formato similar a declaração de variáveis em C++. A terceira seção permite a declaração do tipo utilizado por nodos cuja definição léxica é realizada em *Front*. Por fim a ultima seção permite a definição do formato da árvore de sintaxe abstrata por meio da descrição dos nodos utilizados.

O *token* `C++TEXT` é utilizado para reconhecer o código C++, nesse caso o código C++ é usado nos arquivos gerados sem alterações. Outros *tokens* utilizados nessa gramática são `NONTERMINAL`, `TERMINAL`, `VARID`, `C++TYPE` e `NAME`. `NONTERMINAL`, que são variáveis obrigatórias do lado esquerdo de uma produção, sempre começam com letra minúscula e podem ter qualquer combinação de letras maiúsculas e minúsculas

posteriormente. `TERMINAL` são os terminais que serão utilizados na gramática criada, sua configuração é similar a de literais na linguagem C++, sendo reconhecido qualquer combinação de caracteres entre aspas. `VARID` é similar a `NONTERMINAL` com a diferença de permitir a concatenação de números no fim da variável. Essa numeração é utilizada para distinguir uma variável da outra dentro de uma expressão. `C++TYPE` é utilizado para dar o tipo dos atributos e, devido a geração de código C++ como alvo, são reconhecidos utilizando o mesmo formato para tipos de C++. Da mesma forma que `C++TYPE` segue a formatação de C++, `NAME`, utilizado para nomear os atributos, usa a configuração de variáveis em C++.

A seção iniciada pela palavra-chave `@declarations` é usada para declaração de tipos complexos de C++, podendo ser utilizado o código C++ que o implementador quiser. A seção iniciada pela palavra-chave `@attributes` serve para declarar quais atributos os nodos não-terminais terão para uso em seus acessos nas outras linguagens de descrição. Dessa forma é necessário descrever qual o identificador e tipo do atributo que será gerado para o nodo não terminal. A seção iniciada pela palavra-chave `@leaves` serve para indicar quais os nodos serão utilizados como folhas da árvore de sintaxe abstrata, bem como qual é o tipo de dado que é armazenado por esta folha. Por fim, a seção iniciada pela palavra-chave `@nodes` define o formato que a árvore de sintaxe abstrata gerada possui. Cada regra de produção que define o formato da árvore de sintaxe abstrata é definida como sendo um não-terminal seguido do separador `::=` e uma combinação de terminais e não-terminais para definir o nodo da árvore.

```
start -> declarations attributes leaves nodes
declarations -> "@declarations" c++code
    | empty
attributes -> "@attributes" nonterminals
leaves -> "@leaves" type_decl
    | empty
nodes -> "@nodes" productions
nonterminals -> nonterminals NONTERMINAL "{" attributes "}" ";"
    | NONTERMINAL "{" attributes "}" ";"
type_decl -> type_decl C++TYPE symbol ";"
    | C++TYPE symbol ";"
attributes -> attributes C++TYPE NAME ";"
    | C++TYPE NAME ";"
productions -> productions production ";"
    | production ";"
production -> NONTERMINAL "::=" alternatives
alternatives -> alternatives "|" elements
    | elements
elements -> elements symbol
```

```
    | symbol
symbol -> NONTERMINAL
    | TERMINAL
    | VARID
c++code -> "{" C++TEXT "}"
```

Listagem A.1: Gramática da linguagem *AST*

## A.2 Linguagem *Front*

Na Listagem A.2 está a gramática da linguagem *Front* utilizada no sistema para descrever a transformação do código fonte para árvore de sintaxe abstrata. Essa linguagem é dividida em três seções, a sessão léxica, a sintática e a descrição dos domínios. A seção léxica sempre inicia com o *token* `@lexical` seguido da definição de `UNIT`. Essa definição permite a descrição dos *tokens* da linguagem sendo definida, fazendo com que os `NONTERMINAL` utilizados nessa definição sejam tratados como *tokens* terminais cujo significado é designado nas produções subsequentes. A seção sintática é iniciada pelo *token* `@grammar` e, como uma gramática normal, é seguida por produções que devem ter somente um não terminal no lado esquerdo e qualquer combinação de símbolos no lado direito. Além das produções é possível utilizar variáveis de intervalo com a denotação de que aquela variável pode produzir o que houver entre os caracteres utilizados, sendo equivalente utilizar alternativas para cada caractere dentro do intervalo. Por exemplo `digit === "0" ... "9"` é equivalente a escrever `digit === "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"`. O `===` significa que será produzido o que está do lado direito. Por outro lado, `=/=` fará o contrário, indicando que aquela variável produz tudo que não esteja do lado direito.

Além dessas regras, na parte sintática é possível definir uma regra de produção abstrata que descreve como a regra concreta deve ser representada na árvore de sintaxe abstrata (*AST*). Uma regra de produção abstrata é precedida pelo símbolo `:` e contém uma sequência de símbolos não-terminais e terminais encapsulada em colchetes denotando o nodo da AST. Essas regras de produção abstrata devem conter os mesmos não-terminais da regra de produção concreta, entretanto podem criar tantos terminais quanto for necessário. Caso não haja regra de produção abstrata, a regra de produção concreta é utilizada como formato do nodo da AST.

A parte da descrição dos domínios é iniciada pelo *token* `@domains` que pode conter diversas definições de domínio. Cada definição de domínio é um conjunto de *tokens* não-terminais entre vírgulas seguido de dois pontos e um não-terminal, sendo encerrada por um ponto e virgula. A semântica da definição de um domínio é a substituição

dos não-terminais definidos antes dos dois pontos, pelo nodo não-terminal durante a construção da árvore de sintaxe abstrata.

Os *tokens* `NONTERMINAL`, `VARID`, `TERMINAL` se referem a símbolos terminais descritos por expressões regulares. `NONTERMINAL` são variáveis que sempre começarão com uma letra minúscula e podem ter caracteres maiúsculos ou minúsculos depois, é reconhecido pela expressão regular `[a-z][a-zA-Z]*`. `VARID` é similar a `NONTERMINAL` mas é possível acrescentar números e apostrofes para diferenciar cada variável de transformação, é reconhecido pela expressão regular `[a-z][a-zA-Z]*[0-9]*"'"*?`. `TERMINAL` é qualquer combinação de caracteres entre aspas, sendo reconhecido pela expressão regular `\"(\\.|[\^\\"])*\"|` .

```
start -> grammar-prod domain-prod lexical-prod
   | grammar-prod lexical-prod
lexical-prod -> "@lexical" lexical ";"
grammar-prod -> "@grammar" prod-ranges ";"
domain-prod ->  "@domains" domains ";"
lexical -> unit-def prod-ranges ";"
unit-def -> "UNIT" "::=" nonterminals ";"
nonterminals -> nonterminals "|" NONTERMINAL
    | NONTERMINAL
prod-ranges -> prod-ranges ";" prod-range
    | prod-range
prod-range -> production | range
production -> NONTERMINAL "::=" alternatives
alternatives -> alternatives "|" alternative
    | alternative
alternative -> elements ":" "[" exps "]"
    | elements ":" NONTERMINAL
    | elements
elements -> elements symbol
    | symbol
symbol -> NONTERMINAL
    | TERMINAL
    | VARID
range -> NONTERMINAL "===" specs
    | NONTERMINAL "=/=" specs
specs -> specs "|" spec
    | spec
spec -> TERMINAL "-" TERMINAL
    | TERMINAL
exps -> exps symbol
    | symbol
domains -> domains ";" domain
```

```
    | domain
domain -> domain-elements ":" NONTERMINAL
domain-elements -> domain-elements "," symbol
    | symbol
```

Listagem A.2: Gramática da linguagem *Front*.

## A.3 Linguagem *Middle*

A Listagem A.3 mostra a gramática da linguagem *Middle* que tem por finalidade permitir a descrição de reescrita da árvore de sintaxe abstrata. É possível definir diversos passos que serão realizados na ordem definida, cada passo é iniciado com `@pass` seguido do *token* `PASSNUMBER` que é um número natural indicando a ordem do passo. Cada passo deve ter as definições das ações que serão realizadas na AST e são quatro tipos de ação possível, todas detalhadas, em ordem, pela produção `production` da gramática listada. O primeiro tipo de ação é a realização de nada, em que o nodo somente é redefinido sem alterar a sua configuração ou atributos. A semântica dessa ação é a visita, em ordem pós-fixada, dos nodos definidos. O segundo utiliza o *token* `C++CODE` para possibilitar o cálculo dos atributos dos elementos ou da raiz. A semântica dessa ação é a execução do código C++ dado após a visita dos nodos definidos. O terceiro tipo de ação permite a reordenação dos elementos da produção, além da simplificação de uma produção, lhe retirando elementos. A semântica dessa ação é transformar a árvore de sintaxe abstrata de acordo com a reordenação dos elementos definidos, isso é realizado após a visita dos nodos definidos. E o último tipo de ação combina a possibilidade de calcular os atributos com a reordenação dos nodos.

A linguagem *Middle* também permite a criação de novos nodos da árvore de sintaxe abstrata. Essa criação é feita dentro de `EXTENDED_C++TEXT` com uma pequena extensão da linguagem C++: a atribuição da combinação de elementos novos ou pre-existentes entre colchetes a uma variável da produção permite a criação de um novo nodo. Portanto essa extensão possui o seguinte formato `symbol "=" "[" elements "];` e tem como restrição somente que a nova configuração já esteja definida em AST de forma a possuir as classes necessárias para sua composição.

Assim como as linguagens anteriores, os *tokens* `NONTERMINAL`, `TERMINAL`, `VARID` e `C++TEXT` utilizam definições similares.

O *token* `EXTENDED_C++TEXT` é utilizado para reconhecer o código C++ com a modificação necessária para criação de novos nodos. A extensão de C++ inclui o

reconhecimento da seguinte expressão na linguagem:

```
NONTERMINAL = [ rhs ]
```

em que `NONTERMINAL` é o nodo ao qual o nodo criado será atribuído e `rhs` é uma combinação de símbolos terminais e não-terminais para descrever o novo nodo. O compilador de Middle gera as devidas instanciações e atribuições dos nodos a partir dessa extensão.

```
start -> declarations passes
declarations -> "@declarations" c++code
    | empty
passes -> passes "@pass" PASSNUMBER productions
    | "@pass" PASSNUMBER productions
productions -> productions production ";"
    | production ";"
production -> NONTERMINAL "::=" elements
    | NONTERMINAL "::=" elements "where" extended_c++code
    | NONTERMINAL "::=" elements "=>" NONTERMINAL "::=" elements
    | NONTERMINAL "::=" elements "=>" NONTERMINAL "::=" elements
      "where" extended_c++code
elements -> elements symbol
    | symbol
symbol -> NONTERMINAL
    | TERMINAL
    | VARID
c++code -> "{" C++TEXT "}"
extended_c++code -> "{" EXTENDED_C++TEXT "}"
```

Listagem A.3: Gramática da linguagem *Middle*.

## A.4 Linguagem *Back*

Na Listagem A.4 é apresentada a gramática da linguagem *Back*, empregada na descrição da transformação final da árvore de sintaxe abstrata para código alvo. Essa linguagem possui somente uma seção iniciada pelo *token* `@code` e seguida pelas produções da AST. Cada produção pode ser seguida de código C++ para realizar a tradução da árvore para o código alvo. A semântica de cada produção é a visita dos nodos que compõem a produção, seguida da execução do código C++ definido para a produção.

Os *tokens* utilizados são similares aos das linguagens anteriores, sendo eles `NONTERMINAL`, `TERMINAL`, `VARID` e `C++CODE`. Os três primeiros seguem a mesma lógica apresentada nas linguagens anteriores, sendo: `NONTERMINAL` variáveis iniciadas

com caractere minúsculo, seguido pela combinação de caracteres maiúsculos e minúsculos; `TERMINAL` a combinação de qualquer caractere entre aspas; `VARID` similar a `NONTERMINAL`, com a concatenação de números para distinguir as variáveis do lado direito e `C++CODE` código da linguagem C++ para realizar a tradução final. Esse código não possui nenhuma extensão, podendo utilizar os componentes e métodos da infraestrutura de geração de código que são classes e métodos C++.

```
start -> declarations back
declarations -> "@declarations" c++code
    | empty
back -> "@back" productions
productions -> productions production ";"
    | production ";"
production -> NONTERMINAL "::=" alternatives
alternatives -> alternatives "|" elements c++code
    | elements c++code
    | elements
elements -> elements symbol
    | symbol
symbol -> NONTERMINAL
    | TERMINAL
    | VARID
c++code -> "{" C++TEXT "}"
```

Listagem A.4: Gramática da linguagem *Back*.

# Apêndice B

# Compilador de Small

Este apêndice mostra a definição da linguagem *Small* utilizando as linguagens de descrição criadas neste trabalho. A linguagem *Small* implementada possui declarações de variáveis inteiras de 64 *bits*, variáveis de ponto flutuante de precisão dupla e funções, além dos comandos e expressões suportados pela infraestrutura de geração de código.

## B.1 Árvore de Sintaxe Abstrata

```
@attributes
type { component::Type* type; } ;
exp { component::Type* type; } ;
@leaves
int64_t integer_n;
double float_n;
std::string id;
@nodes
prog ::= "program" decl fnt block resetsymboltable
  | "program" decl block resetsymboltable
  | "program" fnt block resetsymboltable
  | "program" block resetsymboltable
  ;
fnt ::= fnt_header block
  | fnt1 fnt2
  ;
fnt_header ::= "function" type id formalpar
  | "function" type id
  | "procedure" id formalpar
  | "procedure" id
  ;
formalpar ::= "var" type id
  | formalpar1 formalpar2
  ;
block ::= "dblock" decl stm
  | "ublock" stm
  ;
```

```
stm ::= "assign" acc_exp exp
  | "output" exp
  | "ifthenelse" exp block1 block2
  | "ifthen" exp block
  | "proccall" id actualarg
  | "proccall" id
  | "return" exp
  | "whiledo" exp block
  | "block" block
  | stm1 stm2
  ;
actualarg  ::= "deref_call" exp
  | actualarg1 actualarg2
  ;
decl ::= type_name id
  | id
  | decl1 decl2
  ;
type_name ::= "integer"
  | "float"
  ;
acc_exp ::= id ;
exp ::= "ifexp" exp1 exp2 exp3
  | "and" exp1 exp2
  | "or" exp1 exp2
  | "equals" exp1 exp2
  | "notequals" exp1 exp2
  | "lesser" exp1 exp2
  | "greater" exp1 exp2
  | "lesserquals" exp1 exp2
  | "greaterequals" exp1 exp2
  | "add" exp1 exp2
  | "sub" exp1 exp2
  | "mult" exp1 exp2
  | "div" exp1 exp2
  | "rem" exp1 exp2
  | "not" exp1
  | "neg" exp1
  | "funccall" id actualarg
  | "funccall" id
  | "deref" acc_exp
  | "castfp" exp1
  | integer_n
  | float_n
  ;
```

Listagem B.1: Descrição de Small utilizando a linguagem *AST*.

## B.2 Sintaxe

```
@grammar
prog ::= "program" decls fnts block : ["program" decls fnts block
    resetsymboltable ]
    | "program" decls block : ["program" decls block resetsymboltable ]
    | "program" fnts block : ["program" fnts block resetsymboltable ]
```

```
    | "program" block : ["program" block resetsymboltable ]
    ;

fnts ::= fnts fnt : [fnts fnt]
    | fnt : fnt
    ;

fnt ::= fnt_header block ;

fnt_header  ::= type id "(" formalpars ")" : [ "function" type id
   formalpars ]
    | type id "(" ")" : [ "function" type id ]
    | "void" id "(" formalpars ")" : [ "procedure" id formalpars ]
    | "void" id "(" ")" : [ "procedure" id ]
    ;

formalpars ::= formalpars "," formalpar : [ formalpars formalpar ]
    |  formalpar : formalpar
    ;

formalpar ::= type id : [ "var" type id ] ;

block ::= "{" decls stms "}" : ["dblock" startblock decls stms finishblock]
       | "{" stms "}" : ["ublock" startblock stms finishblock];

stms ::= stms stm ";": [ stms stm ]
    | stm ";" : stm
    ;

stm ::= acc_exp ":=" exp  : [ "assign" acc_exp exp ]
    | "output" exp
    | "if" exp "then" block "else" block : ["ifthenelse" exp block block]
    | "if" exp "then" block : ["ifthen" exp block]
    | id "(" actualargs ")" : [ "proccall" id actualargs ]
    | id "(" ")" : [ "proccall" id ]
    | "return" exp
    | "while" exp "do" block : ["whiledo" exp block]
    | "do" block "while" exp : ["dowhile" block exp]
    | block : ["block" block]
    ;

actualargs ::= actualargs "," actualarg : [ actualargs actualarg ]
    |  actualarg : actualarg;

actualarg  ::= exp : ["deref_call" exp] ;

decls ::= decls decl : [ decls decl]
    | decl : decl
    ;

decl ::= type_name id ";" : [ type_name id ] ;

type_name ::= "integer"
    | "float"
    ;
```

```
acc_exp ::= id ;

exp ::= orexp "?" exp ":" exp  : ["ifexp" orexp exp exp]
    | orexp : orexp
    ;

orexp ::= orexp "||" andexp : ["or" orexp andexp]
    | andexp : andexp
    ;

andexp ::= andexp "&&" eqexp : ["and" andexp eqexp]
    | eqexp : eqexp
    ;

eqexp ::= eqexp "=" relexp : ["equals" eqexp relexp]
    | eqexp "!=" relexp : ["notequals" eqexp relexp]
    | relexp : relexp
    ;

relexp ::= relexp "<" addexp : ["lesser" relexp addexp]
    | relexp ">" addexp : ["greater" relexp addexp]
    | relexp "<=" addexp : ["lesserquals" relexp addexp]
    | relexp ">=" addexp : ["greaterequals" relexp addexp]
    | addexp : addexp
    ;

addexp ::= addexp "+" term : ["add" addexp term]
    | addexp "-" term : ["sub" addexp term]
    | term : term
    ;

term ::= term "*" unary : ["mult" term unary]
    | term "/" unary : ["div" term unary]
    | term "%" unary : ["rem" term unary]
    | unary : unary
    ;

unary ::= "!" unary : ["not" unary]
    | "-" unary : ["neg" unary]
    | factor : factor
    ;

factor ::= "(" exp ")" : exp
    | id "(" actualargs ")" : [ "funccall" id actualargs ]
    | id "(" ")" : [ "funccall" id ]
    | acc_exp : [ "deref" acc_exp]
    | integer_n
    | float_n
    ;

@domains
fnts : fnt;
formalpars : formalpar;
stms : stm;
```

```
actualargs : actualarg;
decls : decl;
orexp, andexp, eqexp, relexp, addexp, term, unary : exp;

@lexical
UNIT ::= id
    | integer_n
    | float_n
    ;

letter ::= "a" ... "z" | "A" ... "Z" ;
id ::= letter id | letter ;
digit ::= "0" ... "9" ;
numeral ::= digit numeral | digit ;
integer_n ::= numeral;
float_n ::= numeral "." numeral;
```

Listagem B.2: Descrição da sintaxe de Small utilizando a linguagem *Front*.

## B.3 Reescritas

```
@pass 1
prog ::= "program" decl fnt block resetsymboltable;
prog ::= "program" decl block resetsymboltable;
prog ::= "program" fnt block resetsymboltable;
prog ::= "program" block resetsymboltable;


fnt ::= fnt_header block ;
fnt ::= fnt1 fnt2;

fnt_header ::= "function" type id formalpar where{
        fnt_header->decl_attr = cb->BuildFunctionHeader(id, type->_type,
            formalpar->para_attr);
    };

fnt_header ::= "function" type id where{
        fnt_header->decl_attr = cb->BuildFunctionHeader(id, type->_type);
    };

fnt_header ::= "procedure" id formalpar where{
        fnt_header->decl_attr = cb->BuildFunctionHeader(id,
            component::Type::GetVoid(), formalpar->para_attr);
    };

fnt_header ::= "procedure" id where{
        fnt_header->decl_attr = cb->BuildFunctionHeader(id,
            component::Type::GetVoid());
    };


formalpar ::= "var" type id where{
        formalpar->para_attr = cb->AddParameter(id, type->_type);
    };
```

```
formalpar ::= formalpar1 formalpar2 where {
        formalpar->para_attr = cb->BuildSequence(formalpar1->para_attr,
            formalpar2->para_attr);
    };

block ::= "dblock" startblock decl stm finishblock ;
block ::= "ublock" startblock stm finishblock ;


stm ::= "ifthenelse" exp block1 block2 ;
stm ::= "ifthen" exp block ;
stm ::= "whiledo" exp block ;
stm ::= "block" block ;
stm ::= stm1 stm2 ;

decl ::= type id => decl ::= id where{
        decl->decl_attr = cb->VariableDeclare(id, type->_type);
    };

decl ::= decl1 decl ;

type ::= "integer" where{
        type->_type = component::Type::GetLong();
    };

type ::= "float" where {
        type->_type = component::Type::GetDouble();
    };
```

Listagem B.3: Descrição da declaração de variáveis de Small utilizando a linguagem *Middle*.

```
@pass2
prog ::= "program" decl fnt block resetsymboltable;
prog ::= "program" decl block resetsymboltable;
prog ::= "program" fnt block resetsymboltable;
prog ::= "program" block resetsymboltable;

fnt ::= fnt_header block ;
fnt ::= fnt1 fnt2;

block ::= "dblock" startblock decl stm finishblock ;
block ::= "ublock" startblock stm finishblock ;


stm ::= "assign" acc_exp exp ;
stm ::= "output" exp ;
stm ::= "ifthenelse" exp block1 block2 ;
stm ::= "ifthen" exp block ;
stm ::= "proccall" id actualarg ;
stm ::= "return" exp ;
stm ::= "whiledo" exp block ;
stm ::= "block" block ;
stm ::= stm1 stm2 ;
```

```
actualarg ::= "deref_call" exp ;
actualarg ::= actualarg1 actualarg2 ;

acc_exp ::= id where{
        acc_exp->type = cb->GetVariableType(id);
    };

exp ::= "ifexp" exp1 exp2 exp3 => exp ::= "ifexp" exp1 exp2 exp3 where {
        if(exp2->type->compare(exp3->type))
            exp->type = _exp2->type;
        else{
            if(exp2->type->isIntegerTy())
                exp2 = ["castfp" exp1]
            else
                exp3 = ["castfp" exp2]
            exp->type = component::Type::GetFloat();
        }
    };
exp ::= "and" exp1 exp2 where{
        exp->type = exp2->type;
    };

exp ::= "or" exp1 exp2 where{
        exp->type = exp2->type;
    };

exp ::= "equals" exp1 exp2 => exp ::= "equals" exp1 exp2 where {
        if(exp1->type->compare(exp2->type))
            exp->type = _exp1->type;
        else{
            if(exp1->type->isIntegerTy())
                exp1 = ["castfp" exp1]
            else
                exp2 = ["castfp" exp2]
            exp->type = component::Type::GetFloat();
        }
    };
exp ::= "notequals" exp1 exp2 => exp ::= "notequals" exp1 exp2 where {
        if(exp1->type->compare(exp2->type))
            exp->type = _exp1->type;
        else{
            if(exp1->type->isIntegerTy())
                exp1 = ["castfp" exp1]
            else
                exp2 = ["castfp" exp2]
            exp->type = component::Type::GetFloat();
        }
    };
exp ::= "lesser" exp1 exp2 => exp ::= "lesser" exp1 exp2 where {
        if(exp1->type->compare(exp2->type))
            exp->type = _exp1->type;
        else{
            if(exp1->type->isIntegerTy())
                exp1 = ["castfp" exp1]
```

```
            else
                exp2 = ["castfp" exp2]
            exp->type = component::Type::GetFloat();
        }
    };
exp ::= "greater" exp1 exp2 => exp ::= "greater" exp1 exp2 where {
        if(exp1->type->compare(exp2->type))
            exp->type = _exp1->type;
        else{
            if(exp1->type->isIntegerTy())
                exp1 = ["castfp" exp1]
            else
                exp2 = ["castfp" exp2]
            exp->type = component::Type::GetFloat();
        }
    };
exp ::= "lesserquals" exp1 exp2 => exp ::= "lesserquals" exp1 exp2 where {
        if(exp1->type->compare(exp2->type))
            exp->type = _exp1->type;
        else{
            if(exp1->type->isIntegerTy())
                exp1 = ["castfp" exp1]
            else
                exp2 = ["castfp" exp2]
            exp->type = component::Type::GetFloat();
        }
    };
exp ::= "greaterequals" exp1 exp2 => exp ::= "greaterequals" exp1 exp2
   where {
        if(exp1->type->compare(exp2->type))
            exp->type = _exp1->type;
        else{
            if(exp1->type->isIntegerTy())
                exp1 = ["castfp" exp1]
            else
                exp2 = ["castfp" exp2]
            exp->type = component::Type::GetFloat();
        }
    };
exp ::= "add" exp1 exp2 => exp ::= "add" exp1 exp2 where {
        if(exp1->type->compare(exp2->type))
            exp->type = _exp1->type;
        else{
            if(exp1->type->isIntegerTy())
                exp1 = ["castfp" exp1]
            else
                exp2 = ["castfp" exp2]
            exp->type = component::Type::GetFloat();
        }
    };
exp ::= "sub" exp1 exp2 => exp ::= "sub" exp1 exp2 where {
        if(exp1->type->compare(exp2->type))
            exp->type = _exp1->type;
        else{
            if(exp1->type->isIntegerTy())
```

```
                exp1 = ["castfp" exp1]
            else
                exp2 = ["castfp" exp2]
            exp->type = component::Type::GetFloat();
        }
    };
exp ::= "mult" exp1 exp2 => exp ::= "mult" exp1 exp2 where {
        if(exp1->type->compare(exp2->type))
            exp->type = _exp1->type;
        else{
            if(exp1->type->isIntegerTy())
                exp1 = ["castfp" exp1]
            else
                exp2 = ["castfp" exp2]
            exp->type = component::Type::GetFloat();
        }
    };
exp ::= "div" exp1 exp2 => exp ::= "div" exp1 exp2 where {
        if(exp1->type->compare(exp2->type))
            exp->type = _exp1->type;
        else{
            if(exp1->type->isIntegerTy())
                exp1 = ["castfp" exp1]
            else
                exp2 = ["castfp" exp2]
            exp->type = component::Type::GetFloat();
        }
    };
exp ::= "rem" exp1 exp2 => exp ::= "rem" exp1 exp2 where {
        if(exp1->type->compare(exp2->type))
            exp->type = _exp1->type;
        else{
            if(exp1->type->isIntegerTy())
                exp1 = ["castfp" exp1]
            else
                exp2 = ["castfp" exp2]
            exp->type = component::Type::GetFloat();
        }
    };
exp ::= "not" exp1 where{
        exp->type = exp1->type;
    };
exp ::= "neg" exp1 where{
        exp->type = exp1->type;
    };
exp ::= "funccall" id actualarg where{
        exp->type = cb->GetFunctionType(cb->GetFunction(id));
    };
exp ::= "funccall" id where{
        exp->type = cb->GetFunctionType(cb->GetFunction(id));
    };
exp ::= "deref" acc_exp where{
        exp->type = acc_exp->type;
    };
exp ::= integer_n where{
```

```
        exp->type = component::Type::GetLong();
    };
exp ::= float_n where{
        exp->type = component::Type::GetDouble();
    };
```

Listagem B.4: Descrição da verificação de tipos de Small utilizando a linguagem *Middle*.

## B.4 Geração de Código

```
@llvm
prog ::= "program" decl fnt block{
        cb->BuildFunctionHeader("main", component::Type::GetInt());
        prog->stm_attr = cb->BuildSequence(fnt->stm_attr,
            cb->BuildFunction(cb->GetFunction("main"),
            cb->BuildSequence(block->stm_attr,
            cb->BuildReturnStatement(cb->LoadInteger(0,
            component::Type::GetInt())))));
        cb->GenLLVM(prog->stm_attr);
    }
    | "program" decl block{
        cb->BuildFunctionHeader("main", component::Type::GetInt());
        prog->stm_attr = cb->BuildFunction(cb->GetFunction("main"),
            cb->BuildSequence(block->stm_attr,
            cb->BuildReturnStatement(cb->LoadInteger(0,
            component::Type::GetInt()))));
        cb->GenLLVM(prog->stm_attr);
    }
    | "program" fnt block{
        cb->BuildFunctionHeader("main", component::Type::GetInt());
        prog->stm_attr = cb->BuildSequence(fnt->stm_attr,
            cb->BuildFunction(cb->GetFunction("main"),
            cb->BuildSequence(block->stm_attr,
            cb->BuildReturnStatement(cb->LoadInteger(0,
            component::Type::GetInt())))));
        cb->GenLLVM(prog->stm_attr);
    }
    | "program" block {
        cb->BuildFunctionHeader("main", component::Type::GetInt());
        prog->stm_attr = cb->BuildFunction(cb->GetFunction("main"),
            cb->BuildSequence(block->stm_attr,
            cb->BuildReturnStatement(cb->LoadInteger(0,
            component::Type::GetInt()))));
        cb->GenLLVM(prog->stm_attr);
    };

fnt ::= fnt_header block {
        fnt->stm_attr = cb->BuildFunction(cb->GetFunction(fnt_header->id),
            block->stm_attr);
    }
    |
    fnt1 fnt2 {
        fnts->stm_attr = cb->BuildSequence(fnt1->stm_attr, fnt2->stm_attr);
    }
    ;
```

```
block ::= "dblock" startblock decl stm finishblock{
        block->stm_attr = cb->BuildSequence(decl->decl_attr,
            stm->stm_attr);
    }
    | "ublock" startblock stm finishblock{
        block->stm_attr = stm->stm_attr;
    }
    ;


stm ::= "assign" acc_exp exp{
        stm->stm_attr = cb->BuildAssignmentStatement(acc_exp->exp_attr,
            exp->exp_attr);
    }
    | "output" exp{
        stm->stm_attr = cb->CallFunction("printf",
            cb->DerefArgument(exp->exp_attr));
    }
    | "ifthenelse" exp block1 block2{
        stm->stm_attr = cb->BuildConditionalStatement(exp->exp_attr,
            block1->stm_attr, block2->stm_attr);
    }
    | "ifthen" exp block{
        stm->stm_attr = cb->BuildConditionalStatement(exp->exp_attr,
            block->stm_attr, cb->BuildSkip());
    }
    | "proccall" id actualarg{
        stm->stm_attr = cb->CallFunction(cb->GetFunction(id),
            actualarg->exp_attr);
    }
    | "proccall" id{
        stm->stm_attr = cb->CallFunction(cb->GetFunction(id));
    }
    | "return" exp{
        stm->stm_attr = cb->BuildReturnStatement(exp->exp_attr);
    }
    | "whiledo" exp block{
        stm->stm_attr = cb->BuildRepetitionStatement(cb->BuildSkip(),
            exp->exp_attr, cb->BuildSkip(), block->stm_attr);
    }
    | "block" block{
        stm->stm_attr = block->stm_attr;
    }
    | stm1 stm2 {
        stms->stm_attr = cb->BuildSequence(stm1->stm_attr, stm2->stm_attr);
    }
    ;

actualarg  ::= "deref_call" exp{
        actualarg->exp_attr = cb->DerefArgument(exp->exp_attr);
    }
    | actualarg1 actualarg2{
```

```
        actualargs->exp_attr = cb->BuildSequence(actualarg1->exp_attr,
            actualarg2->exp_attr);
    }
    ;

decl ::= id{
        decl->decl_attr = cb->VariableAlloc(id);
    }
    | decl1 decl2{
        decls->decl_attr = cb->BuildSequence(decl1->decl_attr,
            decl2->decl_attr);
    }
    ;

acc_exp ::= id{
        acc_exp->exp_attr = cb->GetVariableAddress(id);
    }
    ;

exp ::= "ifexp" exp1 exp2 exp3{
        exp->exp_attr = cb->BuildConditionalExpression(exp1->exp_attr,
            exp2->exp_attr, exp3->exp_attr);
    }
    |"and" exp1 exp2{
        exp->exp_attr = cb->Apply(component::GetBinOpe("and",
            component::Type::GetLong()), exp1->exp_attr, exp2->exp_attr);
    }
    | "or" exp1 exp2{
        exp->exp_attr = cb->Apply(component::GetBinOpe("or",
            component::Type::GetLong()), exp1->exp_attr, exp2->exp_attr);
    }
    | "equals" exp1 exp2{
        exp->exp_attr = cb->Apply(component::GetBinOpe("equals",
            component::Type::GetLong()), exp1->exp_attr, exp2->exp_attr);
    }
    | "notequals" exp1 exp2{
        exp->exp_attr = cb->Apply(component::GetBinOpe("notequals",
            component::Type::GetLong()), exp1->exp_attr, exp2->exp_attr);
    }
    | "lesser" exp1 exp2{
        exp->exp_attr = cb->Apply(component::GetBinOpe("lesser",
            component::Type::GetLong()), exp1->exp_attr, exp2->exp_attr);
    }
    | "greater" exp1 exp2{
        exp->exp_attr = cb->Apply(component::GetBinOpe("greater",
            component::Type::GetLong()), exp1->exp_attr, exp2->exp_attr);
    }
    | "lesserquals" exp1 exp2{
        exp->exp_attr = cb->Apply(component::GetBinOpe("lesserequals",
            component::Type::GetLong()), exp1->exp_attr, exp2->exp_attr);
    }
    | "greaterequals" exp1 exp2{
        exp->exp_attr = cb->Apply(component::GetBinOpe("greaterequals",
            component::Type::GetLong()), exp1->exp_attr, exp2->exp_attr);
    }
```

```
| "add" exp1 exp2{
    exp->exp_attr = cb->Apply(component::GetBinOpe("add",
        component::Type::GetLong()), exp1->exp_attr, exp2->exp_attr);
}
| "sub" exp1 exp2{
    exp->exp_attr = cb->Apply(component::GetBinOpe("sub",
        component::Type::GetLong()), exp1->exp_attr, exp2->exp_attr);
}
| "mult" exp1 exp2{
    exp->exp_attr = cb->Apply(component::GetBinOpe("mult",
        component::Type::GetLong()), exp1->exp_attr, exp2->exp_attr);
}
| "div" exp1 exp2{
    exp->exp_attr = cb->Apply(component::GetBinOpe("div",
        component::Type::GetLong()), exp1->exp_attr, exp2->exp_attr);
}
| "rem" exp1 exp2{
    exp->exp_attr = cb->Apply(component::GetBinOpe("rem",
        component::Type::GetLong()), exp1->exp_attr, exp2->exp_attr);
}
| "not" exp1{
    exp->exp_attr = cb->Apply(component::GetUnOpe("not",
        component::Type::GetLong()), exp1->exp_attr);
}
| "neg" exp1{
    exp->exp_attr = cb->Apply(component::GetUnOpe("neg",
        component::Type::GetLong()), exp1->exp_attr);
}
| "funccall" id actualarg{
    exp->exp_attr = cb->CallFunction(cb->GetFunction(id),
        actualarg->exp_attr);
}
| "funccall" id{
    exp->exp_attr = cb->CallFunction(cb->GetFunction(id));
}
| "deref" acc_exp{
    exp->exp_attr = cb->LoadAddress(acc_exp->exp_attr);
}
| integer_n{
    exp->exp_attr = cb->LoadInteger(integer_num,
        component::Type::GetLong());
}
| float_n{
    exp->exp_attr = cb->LoadFP(float_num,
        component::Type::GetDouble());
}
;
```

Listagem B.5: Descrição da geração de código de Small utilizando a linguagem *Back*.

## B.5 Exemplo de Compilação

Esta seção mostra um exemplo dos passos realizados para a compilação de um programa em *Small* utilizando o compilador gerado pelo ambiente de compiladores descrito neste

trabalho.

```
1 program {
2   float x;
3   x = 1 + 1.0;
4 }
```

Listagem B.6: Exemplo de programa na linguagem *Small*.

O programa utilizado para o exemplo encontra-se na listagem B.6. A Figura B.1 ilustra a árvore de *parsing* do programa e a Figura B.2 mostra a árvore de sintaxe abstrata correspondente gerada pela descrição do programa de *Front* da Listagem B.2. Na Figura B.3, está a árvore de sintaxe abstrata após aplicação do primeiro programa *Middle*, descrito na Listagem B.3. E a Figura B.4 mostra o formato da AST resultante da execução do segundo programa de *Middle*, descrito na Listagem B.4. O programa de Back da Listagem B.5, percorre a AST para gerar o código intermediário apresentado na Listagem B.7, e na Listagem B.8 está o *bytecode* LLVM gerado ao percorrer a lista de instruções de código intermediário.

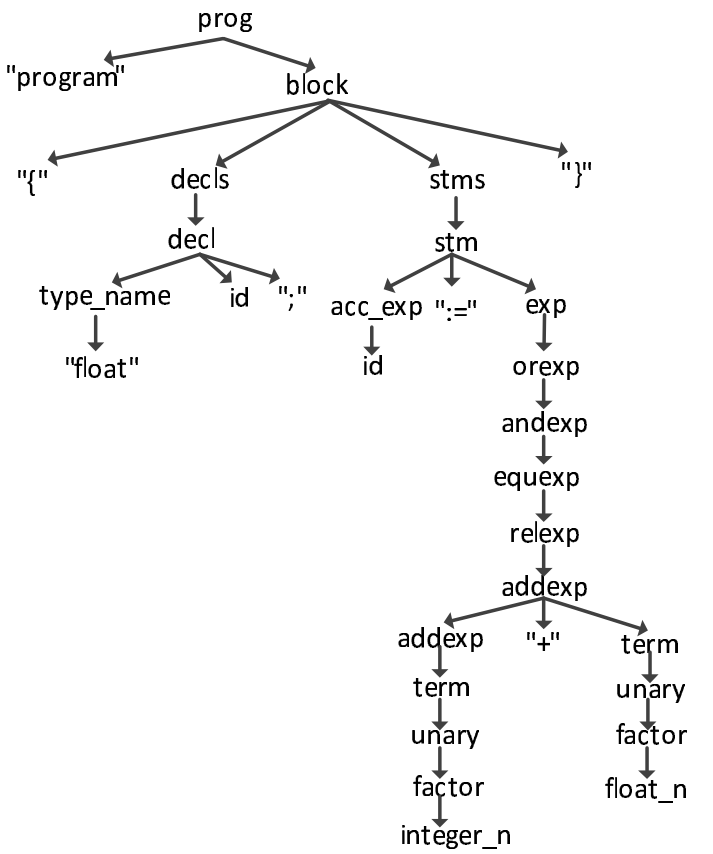


Figura B.1: Árvore de *parsing* do programa da Listagem B.6.

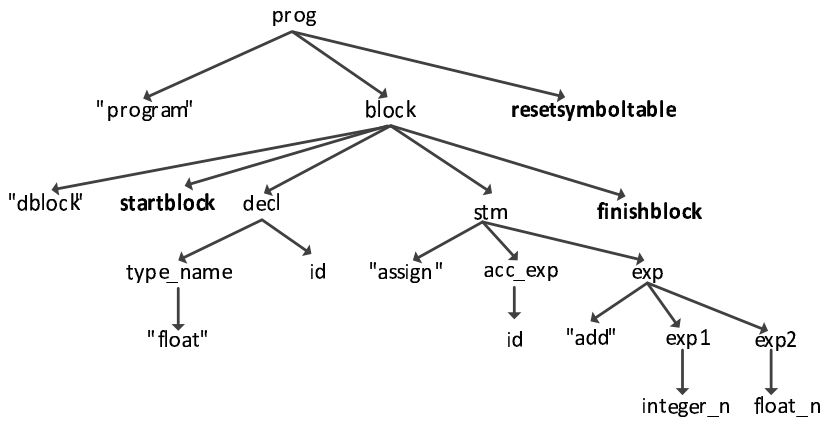


Figura B.2: AST inicial do programa da Listagem B.6.

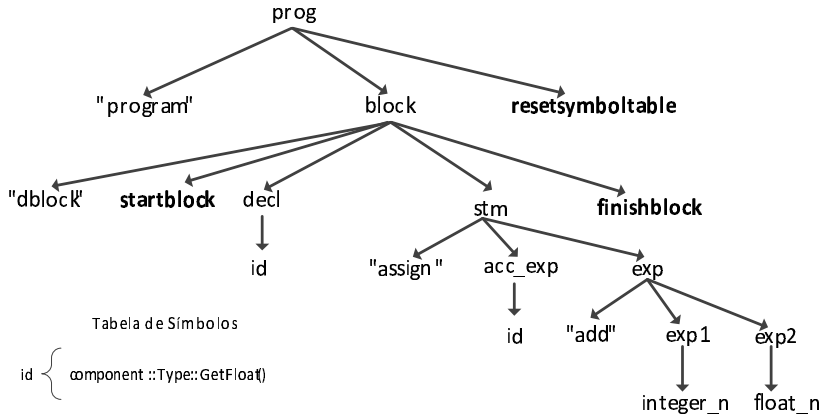


Figura B.3: AST resultante do primeiro programa de *Middle* sobre o programa da Listagem B.6.

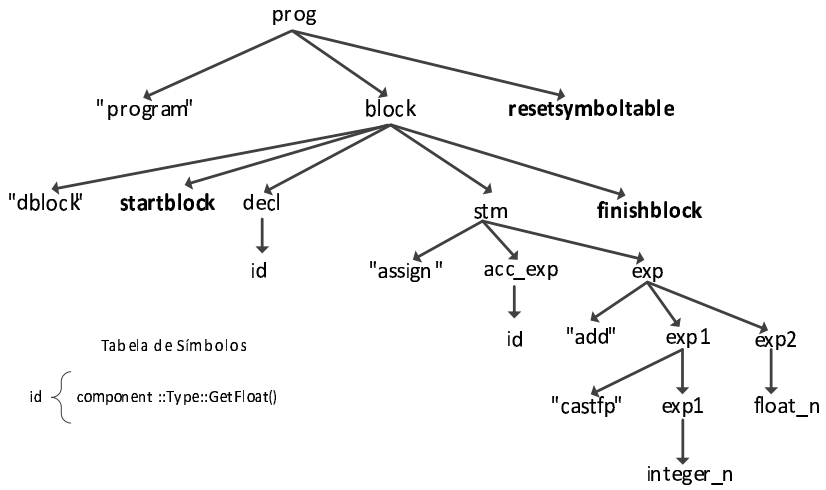


Figura B.4: AST resultante do segundo programa de *Middle* sobre o programa da Listagem B.6.

```
1 $0 = Alloc type::double "x"
2 $1 = UnOp "castfp" ConstInt(type::long, 1)
3 $1 = BinOp "+"  $1 ConstFP(type::double, 1.0)
4 Store $0 $1
```

Listagem B.7: Instruções da infraestrutura do programa *Small* da Listagem B.6.

```
1 define i32 @main() {
2 entry:
3   %x = alloca double
4   store double 2.000000e+00, double* %x
5   ret i32 0
6 }
```

Listagem B.8: Bytecode LLVM do programa *Small* da Listagem B.6.

# Referências Bibliográficas


Aho, A. V.; Lam, M. S.; Sethi, R. & Ullman, J. D. (2007). *Compilers: Principles, Techniques, & Tools with Gradiance*. Addison-Wesley Publishing Company, USA, 2nd edição. ISBN 0321547985, 9780321547989.

Becket, R. & Somogyi, Z. (2008). Dcgs + memoing = packrat parsing but is it worth it? Em *Proceedings of the 10th international conference on Practical aspects of declarative languages*, PADL'08, pp. 182--196, Berlin, Heidelberg. Springer-Verlag.

Berk, E. & Ananian, C. S. (1997). Jlex: A lexical analyzer generator for java. *Online: http://www.cs.-princeton.edu/ appel/modern/java/JLex*.

Bigonha, R. S. (1998). The revised report on the language *SCRIPT* for denotational semantics. Relatório técnico, Universidade Federal de Minas Gerais, Departamento de Ciência da Computação.

Brand, M.; Heering, J.; Klint, P. & Olivier, P. (2002). Compiling Rewrite Systems: The ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334--368.

Bravenboer, M.; Kalleberg, K.; Vermaas, R. & Visser, E. (2008). Stratego/XT 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52--70.

Cazzola, W. & Poletti, D. (2010). Dsl evolution through composition. Em *Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution*, RAM-SE '10, pp. 6:1--6:6, New York, NY, USA. ACM.

Cooper, K. & Torczon, L. (2007). *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edição.

Cytron, R.; Ferrante, J.; Rosen, B. K.; Wegman, M. N. & Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451--490.

Ekman, T. & Hedin, G. (2007). The jastadd system - modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14--26. ISSN 0167-6423.

Ekman, T. & Hedin, G. (2008). The jastadd extensible java compiler. *SIGPLAN Not.*, 42(10):1--18. ISSN 0362-1340.

Ford, B. (2002a). *Packrat parsing: a practical linear-time algorithm with backtracking*. Tese de doutorado, Massachusetts Institute of Technology.

Ford, B. (2002b). Packrat parsing:: simple, powerful, lazy, linear time, functional pearl. *SIGPLAN Not.*, 37(9):36--47. ISSN 0362-1340.

Ford, B. (2004). Parsing expression grammars: a recognition-based syntactic foundation. *SIGPLAN Not.*, 39(1):111--122. ISSN 0362-1340.

Foundation, F. S. (2009). Bison gnu parser generator. http://www.gnu.org/software/bison/.

Gagnon, E. & Hendren, L. (1998). Sablecc, an object-oriented compiler framework. Em *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*, pp. 140 –154. ISSN .

Gray, R.; Levi, S.; Heuring, V.; Sloane, A. & Waite, W. (1992). Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121--130.

Heineman, G. & Councill, W. (2001). *Component-based software engineering: putting the pieces together*. ACM Press Series. Addison-Wesley. ISBN 9780201704853.

Henriques, P.; Pereira, M.; Mernik, M.; Lenic, M.; Gray, J. & Wu, H. (2005). Automatic generation of language-based tools using the lisa system. *Software, IEE Proceedings* -, 152(2):54 – 69. ISSN 1462-5970.

Hudson, S. E.; Flannery, F.; Ananian, C. S.; Wang, D. & Appel, A. W. (1999). Cup parser generator for java. *Online: http://www.cs.princeton.edu/appel/modern/java/CUP*.

Johnson, R. E. (1997). Frameworks = (components + patterns). *Commun. ACM*, 40(10):39--42. ISSN 0001-0782.

Johnson, S. C. (1975). YACC—yet another compiler-compiler. Relatório técnico CS-32, AT & T Bell Laboratories, Murray Hill, N.J.

Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; marc Loingtier, J. & Irwin, J. (1997). Aspect-oriented programming. Em *ECOOP*. SpringerVerlag.

Knuth, D. (1968). Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145. ISSN 0025-5661.

Lattner, C. (2002). LLVM: An Infrastructure for Multi-Stage Optimization. Dissertação de mestrado, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL.

Lattner, C. & Adve, V. (2003). Architecture for a Next-Generation GCC. Em *Proc. First Annual GCC Developers' Summit*, Ottawa, Canada.

Lattner, C. & Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. Em *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California.

Lesk, M. & Laboratories, B. (1987). *Lex: A lexical analyzer generator*. Bell Laboratories.

Mizushima, K.; Maeda, A. & Yamaguchi, Y. (2010). Packrat parsers can handle practical grammars in mostly constant space. Em *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '10, pp. 29--36, New York, NY, USA. ACM.

Nystrom, N.; Clarkson, M. R. & Myers, A. C. (2003). Polyglot: an extensible compiler framework for java. Em *Proceedings of the 12th international conference on Compiler construction*, CC'03, pp. 138--152, Berlin, Heidelberg. Springer-Verlag.

Parr, T. (2013). *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edição. ISBN 1934356999, 9781934356999.

Parr, T. & Fisher, K. (2011). Ll(*): the foundation of the antlr parser generator. *SIGPLAN Not.*, 46(6):425--436. ISSN 0362-1340.

Parr, T. J. & Quong, R. W. (1995). Antlr: a predicated-ll(k) parser generator. *Softw. Pract. Exper.*, 25(7):789--810. ISSN 0038-0644.

Project, F. (2008). flex: The fast lexical analyzer. http://flex.sourceforge.net/.

Rebernak, D.; Mernik, M.; Henriques, P. R. & Pereira, M. J. V. (2006). Aspectlisa: An aspect-oriented compiler construction system based on attribute grammars. *Electronic Notes in Theoretical Computer Science*, 164(2):37 – 53. ISSN 1571-0661.

Scott, M. L. (2009). *Programming Language Pragmatics (3. ed.)*. Academic Press. ISBN 978-0-12-374514-9.

Tofte, M. (1990). *Compiler generators: what they can do, what they might do, and what they will probably never do*. Springer-Verlag New York, Inc., New York, NY, USA. ISBN 0-387-51471-6.

Warth, A.; Douglass, J. R. & Millstein, T. (2008). Packrat parsers can support left recursion. Em *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '08, pp. 103--110, New York, NY, USA. ACM.

Watt, D. A. & Findlay, W. (2004). *Programming language design concepts*. Wiley. ISBN 978-0-470-85320-7.