Roberto de Carvalho Ferreira

Ferramenta computacional para a definição e geração de estruturas cristalinas

Dissertação apresentada ao Programa de Pós-graduação em Modelagem Computacional, da Universidade Federal de Juiz de Fora como requisito parcial à obtenção do grau de Mestre em Modelagem Computacional.

Orientador: Prof. D.Sc. Marcelo Lobosco

Coorientador: Marcelo Bernardes Vieira

Coorientador: Sócrates de Oliveira Dantas

Juiz de Fora

2012

Ferreira, Roberto de Carvalho

Ferramenta computacional para a definição e geração de estruturas cristalinas/Roberto de Carvalho Ferreira. – Juiz de Fora: UFJF/MMC, 2012.

XIII, 134 p. 29, 7cm.

Orientador: Marcelo Lobosco

Coorientador: Marcelo Bernardes Vieira

Coorientador: Sócrates de Oliveira Dantas

Dissertação (mestrado) – UFJF/MMC/Programa de Modelagem Computacional, 2012.

Referências Bibliográficas: p. 119 – 122.

1. Estruturas Cristalinas. 2. Objetos Implícitos. 3. Computação Geométrica. 4. Compiladores. 5. Simuladores. I. Lobosco, Marcelo *et al.*. II. Universidade Federal de Juiz de Fora, MMC, Programa de Modelagem Computacional.

Roberto de Carvalho Ferreira

Ferramenta computacional para a definição e geração de estruturas cristalinas

Dissertação apresentada ao Programa de Pós-graduação em Modelagem Computacional, da Universidade Federal de Juiz de Fora como requisito parcial à obtenção do grau de Mestre em Modelagem Computacional.

Aprovada em 29 de Agosto de 2012.

BANCA EXAMINADORA

Prof. D.Sc. Marcelo Lobosco - Orientador
Universidade Federal de Juiz de Fora

Prof. D.Sc. Marcelo Bernardes Vieira - Coorientador
Universidade Federal de Juiz de Fora

Prof. D.Sc. Sócrates de Oliveira Dantas - Coorientador
Universidade Federal de Juiz de Fora

Prof. D.Sc. Alexandre Fontes da Fonseca
Universidade Estadual Paulista

Prof. D.Sc. Ciro de Barros Barbosa
Universidade Federal de Juiz de Fora

Dedico este trabalho a toda a minha família e aos meus amigos.

AGRADECIMENTOS

Gostaria de agradecer primeiramente aos meus pais, Celso e Maria das Graças, e a minha irmã Mônica pelo apoio, amor e carinho.

A minha noiva Ana Cláudia, pela atenção, compreensão e auxílio nos momentos mais difíceis.

Aos meus orientadores, Marcelo Lobosco, Marcelo Bernardes e Sócrates.

Ao Mestrado em Modelagem Computacional e ao Grupo de Computação Gráfica.

À CAPES pela bolsa de mestrado.

"A educação não transforma o mundo. Educação muda pessoas. Pessoas transformam o mundo."

Paulo Freire

RESUMO

A evolução dos computadores, mais especificamente no que diz respeito ao aumento de sua capacidade de armazenamento e de processamento de dados, possibilitou a construção de ferramentas computacionais destinadas à simulação de fenômenos físicos e químicos. Com isso, a realização de experimentos práticos vem, em alguns casos, sendo substituída pela utilização de experimentos computacionais, que simulam o comportamento de inúmeros elementos que compõem o experimento original. Neste contexto, podemos destacar os modelos utilizados para a simulação de fenômenos em escala atômica. A construção desses simuladores requer, por parte dos desenvolvedores, um amplo estudo e definição de modelos precisos e confiáveis. Tal complexidade se reflete, muitas vezes, em simuladores complexos, destinados a simulação de um grupo restrito de estruturas, expressos de maneira fixa, utilizando algumas formas geométricas padrões.

Este trabalho propõe uma ferramenta computacional para a geração de um conjunto de estruturas cristalinas. Este conjunto é caracterizado pela organização espacial regular dos átomos que a compõe. A ferramenta é composta por a) uma linguagem de programação, que rege a criação das estruturas através da definição de um sistema cristalino e a construção de objetos a partir de funções características e operadores CSG (*Construtive Solid Geometry*), e b) um compilador/interpretador que analisa um código fonte escrito na linguagem, e gera a partir deste o objeto correspondente.

A ferramenta oferece aos desenvolvedores um mecanismo simples que possibilita a geração de um número irrestrito de estruturas. Sua aplicabilidade é demonstrada através da incorporação de uma estrutura, gerada a partir de um código fonte, ao simulador *Monte Carlo Spins Engine*, criado pelo Grupo de Computação Gráfica da Universidade Federal de Juiz de Fora.

Palavras-chave: Estruturas Cristalinas. Objetos Implícitos. Computação Geométrica. Compiladores. Simuladores.

ABSTRACT

The evolution of computers, more specifically regarding the increased storage and data processing capacity, allowed the construction of computational tools for the simulation of physical and chemical phenomena. Thus, practical experiments are being replaced, in some cases, by computational experiments that simulate the behavior of many elements that compose the original one. In this context, we can highlight the models used to simulate phenomena at the atomic scale. The construction of these simulators requires, by developers, the study and definition of accurate and reliable models. This complexity is often reflected in the construction of complex simulators, which simulate a limited group of structures. Such structures are sometimes expressed in a fixed manner using a limited set of geometric shapes.

This work proposes a computational tool that aims to generate a set crystal structures. Crystal structures are characterized by a set of atoms arranged in a regular way. The proposed tool consists of a) a programming language, which is used to describe the structures using for this purpose characteristic functions and CSG (*Construtive Solid Geometry*) operators, and b) a compiler/interpreter that examines the source code written in the proposed language, and generates the objects accordingly.

This tool enables the generation of an unrestricted number of structures. Its applicability is demonstrated through the incorporation of a structure, generated from the source code, to the Monte Carlo Spins Engine, a spin simulator developed by the Group of Computer Graphics of the Federal University of Juiz de Fora.

Keywords: Crystal Structures. Implicit Objects. Geometric Computing. Compilers. Simulators.

SUMÁRIO

1 INTRODUÇÃO

Os estudos de fenômenos físicos e químicos estão sempre relacionados a experimentos e observações. Entretanto, nem sempre a realização de tais experimentos pode ser feita de modo trivial e/ou sem grandes custos envolvidos. Em outras situações, experimentações reais são impossíveis de serem feitas, como por exemplo em algumas situações que envolvem a área de nanotecnologia. Assim, alternativas devem ser construídas, como, por exemplo, o emprego de simuladores.

Os simuladores são construídos a partir de modelos físicos, químicos, matemáticos e computacionais e propiciam o estudo dos fenômenos de maneira mais prática, tendo em vista que várias hipóteses podem ser testadas pelo pesquisador com a simples alteração dos parâmetros e constantes utilizadas na simulação. Informações detalhadas sobre os experimentos podem ser colhidas e utilizadas para uma melhor compreensão dos fenômenos estudados. Desta forma, muitos simuladores são construídos e implementados usando estruturas peculiares a um determinado campo de estudo, restringindo muitas vezes a simulação a um grupo restrito de estruturas.

O simulador Monte Carlo Spins Engine (MCSE) [1], por exemplo, possibilita a simulação da interação de *spins* em estruturas tridimensionais genéricas formadas por átomos com propriedades magnéticas. Nele a modelagem das estruturas está restrita a um conjunto básico de formas (esferas, cilindros e cubos) compostas por átomos igualmente espaçados, impossibilitando assim a simulação de um conjunto mais amplo de estruturas.

Utilizando o MCSE, outros trabalhos foram propostos com o intuito de diminuir o tempo gasto na simulação. O uso de múltiplas CPUs [2, 3] e GPUs [3], através de técnicas de computação paralela, possibilitou ganhos reais na computação dos cálculos relacionados a simulação. Contudo, esses trabalhos não abordaram nenhum mecanismo de incorporação de novas estruturas para o simulador.

Na tentativa de definir um mecanismo de geração de estruturas para o MCSE, Ferreira [4] apresentou uma linguagem de programação destinada à geração de objetos implícitos na simulação de *spins*. Baseado nesta linguagem foi desenvolvido uma nova, mais ampla, destinada à geração de estruturas cristalinas. Este trabalho apresenta essa linguagem e seu respectivo compilador.

1.1 Definição do Problema

O problema tratado neste trabalho é o de prover meios computacionais para a definição e geração de estruturas cristalinas para simuladores.

1.2 Objetivos

O objetivo primário deste trabalho é o desenvolvimento de uma ferramenta computacional destinada à geração de estruturas cristalinas. A ferramenta proposta é composta por um compilador-interpretador que analisa e gera uma estrutura cristalina, a partir de um código fonte escrito em uma linguagem de programação específica.

Os objetivos secundários deste trabalho envolvem a apresentação dos conceitos utilizados na construção da linguagem de programação, utilizada pela ferramenta, a exibição da estrutura de funcionamento dos compiladores e a demonstração, através de exemplos, da aplicabilidade da ferramenta.

1.3 Trabalhos Correlatos

Não foram encontrados, na literatura, trabalhos relacionados à descrição de uma estrutura cristalina a partir de uma linguagem de programação. Entretanto, existe um grande número de softwares que se propõem a construção e visualização de tais estruturas.

A construção dessas estruturas, nos softwares, é feita a partir de interfaces gráficas onde o usuário informa o retículo de *Bravais*, os átomos que compõe a base e os parâmetros da rede. Elementos relacionados à cristalografia, tais como grupos de simetria e índice de *Miller*, não abordados neste trabalho, também podem ser utilizados na definição.

Grande parte destas ferramentas apresentam recursos avançados, como a possibilidade incorporação de impurezas e falhas à estrutura e a simulação de alguns fenômenos, tais como a difração de raios X. Em contrapartida essas ferramentas não possibilitam que o usuário defina uma região espacial no qual a estrutura deva estar inserida. A estrutura é definida e expandida segundo uma célula unitária, sendo facultado ao usuário apenas a remoção pontual de átomos.

A seguir são apresentadas algumas dessas ferramentas:

ATOMS [5]: Software proprietário que possibilita ao usuário desenhar estruturas atômicas tridimensionais, incluindo cristais, polímeros e moléculas. Pertence a uma *suite*, conjunto de programas, denominada *Shape Software*. Essa *suite* possibilita a visualização da morfologia externa (faces) de cristais e simulação de análises vibracionais em moléculas e cristais.

Balls & Sticks [6, 7] : Software livre que oferece ao usuário uma interface gráfica amigável onde todas as funcionalidades são controladas com o *mouse*. Ele armazena as informações referentes ao grupos espacias, tais como a simetria e as condições de reflexão codificados em um arquivo texto. Caso o usuário precise utilizar uma configuração não padrão, ele pode incorporá-la ao arquivo usando um editor de texto comum.

CrystalMaker® [8]: Software proprietário composto por três programas. CrystalMaker® (responsável pela geração de estruturas cristalinas e moleculares), CristalDiffract® (responsável pela manipulação de propriedades de difração de materiais cristalinos) e SingleCrystal™ (ambiente de simulação de raios-X de nêutrons, e padrões de difração de elétrons a partir de cristais individuais). A ferramenta possibilita a criação de vídeos a partir das simulações realizadas.

DRAWxtl [9, 10]: Software livre que apresenta interface similar ao Balls & Sticks e oferece ao usuário a possibilidade de geração arquivos VRML (*Virtual Reality Modeling Language*), destinados a exibição de estruturas tridimensionais na *internet*.

A ferramenta proposta neste trabalho difere dos *softwares*, descritos acima, nos seguintes aspectos:

- Enquanto os *softwares* oferecem suporte à criação de estruturas monocristalinas, formadas a partir de uma única célula unitária, a ferramenta possibilita a geração de estruturas monocristalinas e policristalinas, formadas a partir de várias células unitárias equivalentes com orientações distintas;

- Nos *softwares*, a geração das estruturas é feita a partir da expansão de uma célula unitária. Na ferramenta, a geração é feita através do preenchimento de uma região espacial, determinada no código, por células unitárias;

- A linguagem, utilizada pela ferramenta, possibilita a construção de estruturas com as mais variadas formas e a atribuição de propriedades aos elementos que a constituem (átomos, íons ou moléculas), característica esta inexistente nos *softwares*.

1.4 Organização

Este trabalho é dividido em 10 capítulos. Nos próximos 5 capítulos são apresentados os conceitos utilizados na construção da ferramenta. O Capítulo 2 apresenta as características dos materiais sólidos que os definem como estruturas cristalinas e a categorização dessas estruturas em sete sistemas distintos.

No Capítulo 3 são apresentados os conceitos referentes à linguagens e gramáticas.

O Capítulo 4 apresenta o funcionamento dos compiladores, subdivido em duas fases: Analítica e Sintética.

No Capítulo 5 estão descritos os dois modelos utilizados para a representação de objetos matematicamente (Paramétrico e Implícito), discutindo as vantagens e desvantagens de cada um. Um conjunto de regras de agrupamentos de objetos definidos pelo modelo implícito é apresentado.

O Capítulo 6 exibe o algoritmo Gilbert-Johnson-Keerthi (GJK), utilizado neste trabalho para definição de interseção entre objetos. Além do funcionamento do algoritmo, são apresentados dois conceitos que o algoritmo utiliza (Regiões de *Voronoi* e Soma de *Minkowski*).

A ferramenta proposta neste trabalho é apresentada nos Capítulos 7 e 8. O Capítulo 7 apresenta uma linguagem de programação para geração de estruturas cristalinas descrevendo os elementos que a constituem.

O Capítulo 8 apresenta o compilador implementado na ferramenta. Seu funcionamento é descrito a partir de seis módulos: Gerenciador de Caracteres, Léxico, Sintático, Semântico, Interpretador e Gerenciador de Erros.

Por fim, o Capítulo 9 apresenta os resultados obtidos e o Capítulo 10 descreve as conclusões obtidas, indicando alguns trabalhos futuros.

2 ESTRUTURAS CRISTALINAS

2.1 Introdução

Materiais sólidos podem ser classificados de acordo com a regularidade com que átomos ou íons se arranjam entre si. Este capítulo tem por objetivo apresentar a classificação e os termos utilizados para descrever um conjunto peculiar de materiais sólidos, denominado estrutura cristalinas.

Um material cristalino é um no qual átomos estão situados numa disposição repetitiva ou periódica ao longo de grandes distâncias atômicas; isto é, existe uma ordenação de grande alcance tal que, na solidificação, os átomos se posicionarão entre si num modo tridimensional repetitivo. Metais, materiais cerâmicos e certos polímeros, sob determinadas condições de solidificação, formam estruturas cristalinas. Em materiais onde a solidificação não apresenta esta ordenação atômica são chamados de não-cristalinos ou amorfos [11].

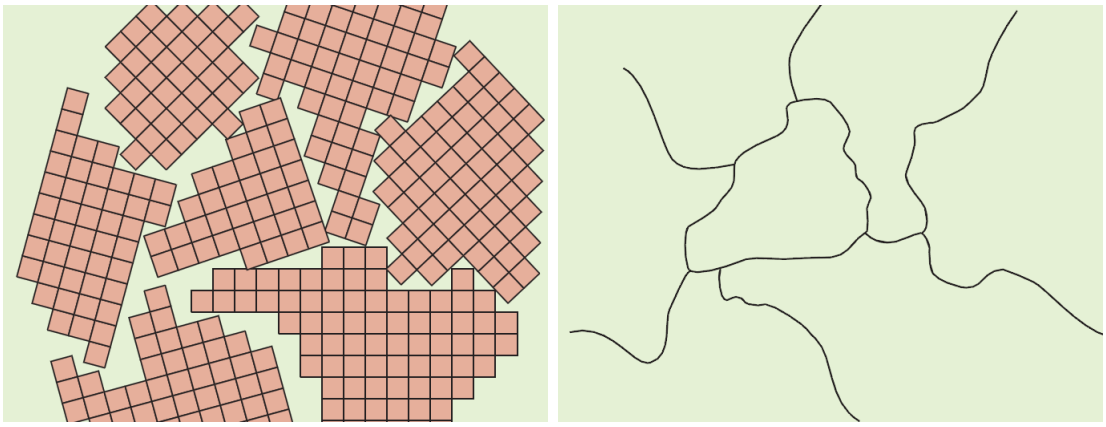
Algumas das propriedades dos sólidos cristalinos dependem da estrutura cristalina do material, a maneira na qual átomos, íons ou moléculas são espacialmente arranjados. Há na natureza uma grande variedade de estruturas cristalinas. As estruturas metálicas apresentam um agrupamento mais simples dos átomos enquanto estruturas cerâmicas ou poliméricas apresentam uma disposição mais complexa dos átomos que a constituem.

Independente das ligações existentes entre os átomos de uma estrutura cristalina, a posição dos mesmos pode ser descrita através de uma célula unitária transladada e uma base. Célula unitária é a menor subdivisão da estrutura, é a unidade estrutural básica ou o tijolo de construção da estrutura cristalina. A base representa átomos ou blocos de átomos que devem ser inseridos em cada ponto da célula.

Na natureza os elementos não se comportam dessa maneira tão regular. Muitas vezes no processo de cristalização de uma estrutura ocorrem o aparecimento de impurezas (átomos de elementos diferentes dos que constituem a estrutura se ligam a mesma), de falhas (rupturas e deslocamentos que causam uma mudança na disposição dos átomos ou moléculas em uma determinada parte da estrutura) e formação de grãos ou cristais (trechos distintos da estrutura onde os átomos obedecem uma distribuição regular definida pela

célula unitária e pela base, contudo a orientação da célula unitária e da base é distinta nesses trechos).

Um material cristalino composto por um único cristal é definido como monocristalino e um material cristalino onde há formação de vários cristais é conhecido como policristalino. A Figura 2.1 mostra a visualização de um elemento policristalino.



(a) Visualização dos grãos em um policristal. (b) Visualização de um policristal através de um microscópio.

Figura 2.1: Exibição de um elemento policristalino. Adaptado de [11].

Este trabalho ignora a existência de impurezas e falhas na estrutura, maiores detalhes sobre esses itens podem ser encontrados em [11] e [12]. Estrutura monocristalinas e policristalinas sem impurezas e falhas são o objeto de estudo.

Uma célula unitária em $\Re^3$ é descrita através de três vetores, $\{\vec{a},\vec{b},\vec{c}\}$, linearmente independentes. Um exemplo de célula unitária com os vetores que a compõe é exibido na Figura 2.2.

A rede oriunda da célula unitária definida acima é obtida através da combinação linear dos vetores que a compõe. Ela é descrita na Equação2.1.

$$\vec{R} = n_1\vec{a} + n_2\vec{b} + n_3\vec{c} \text{ tal que } n_1, n_2, n_3 \in \mathbb{Z} \tag{2.1}$$

A Figura 2.3 apresenta e exemplifica a definição de um cristal através de uma rede e uma base em $\Re^2$.

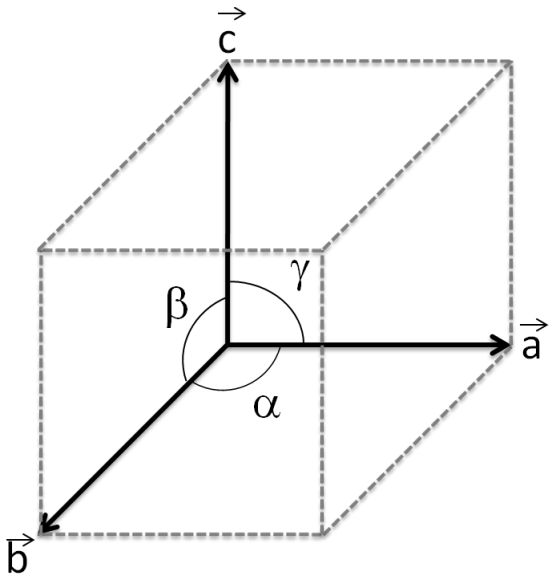
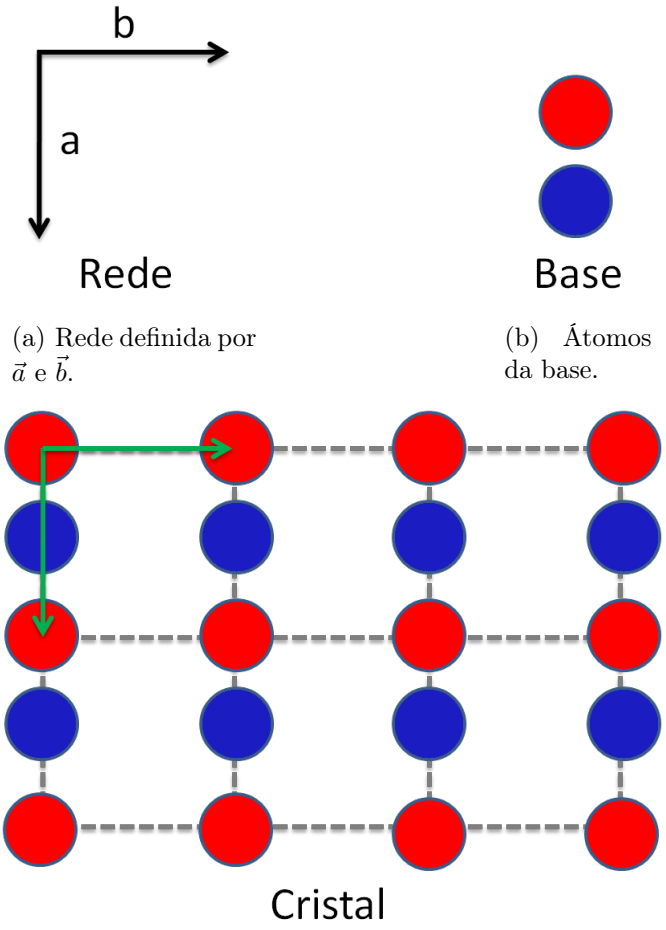


Figura 2.2: Definição de uma célula unitária a partir de três vetores.



(a) Rede definida por $\vec{a}$ e $\vec{b}$.

(b) Átomos da base.

(c) Cristal resultante.

Figura 2.3: Definição de um cristal a partir de uma rede e uma base.

2.2 Retículos de Bravais

Em meados do século passado, o cientista frânces *Auguste Bravais* propôs a utilização de sete sistemas cristalinos básicos, ou redes, para o estudo das estruturas cristalinas: cúbico, tetragonal, ortorrômbico, monoclínico, triclínico, romboédrico ou trigonal, e hexagonal. A partir desses sistemas, ele verificou a existência de quatorze maneiras de distribuir pontos regularmente no espaço. Essas quatorze configurações são conhecidas como Retículos de Bravais ou células convencionais [12].

As células convencionais são expressas pelos sistemas cristalinos e uma distribuição peculiar da base no mesmo. O nome dado a cada retículo é composto pelo sistema cristalino básico seguido da distribuição da base (sistema cristalino básico + distribuição da base).

A base pode ser replicada de quatro maneiras diferentes nos sistemas cristalinos. São elas:

simples (P): Define que a base é replicada em cada aresta da célula definida pelo sistema. O retículo é equivalente à célula unitária.

corpo centrado (I): A base é replicada em cada aresta e no centro da célula. O retículo equivale à duas células unitárias.

face centrada (F): A base é replicada em cada aresta e no centro de cada face da célula. O retículo é proveniente de três células unitárias.

base centrada (A, B ou C): A base é replicada em cada aresta e no centro de duas faces opostas na célula. A letras A, B, C definem quais são as faces cujo centro receberá os elementos da base. O retículo é composto por duas células unitárias.

As seções abaixo apresentaram os sete sistemas cristalinos e os quatorze retículos formados por eles. As células são definidas conforme a Figura 2.2:

- $\vec{a}$, $\vec{b}$, $\vec{c}$ são os vetores usados para definir os sistemas cristalinos. Eles serão exibidos nas cores vermelho, azul e verde;
- α, β e γ representam os ângulos formados pelos vetores ab, bc e ac respectivamente.

2.2.1 Sistema Cúbico (cubic)

Sistemas cúbicos são caracterizados pela igualdade entre os parâmetros de rede (nome atribuído ao valor dos módulos dos vetores que definem o sistema) e a ortogonalidade entre os vetores: $|\vec{a}| = |\vec{b}| = |\vec{c}|$ e $\alpha = \beta = \gamma = 90°$. A partir dele são definidos três retículos:

Cúbico P: Conhecido como *sc* (*simple cubic*).

Cúbico I: Conhecido como *bcc* (*body-centered cubic*).

Cúbico F: Conhecido como *fcc* (*face-centered cubic*).

A Figura 2.4 apresenta o Sistema Cúbico e a Figura 2.5 os retículos oriundos do mesmo.

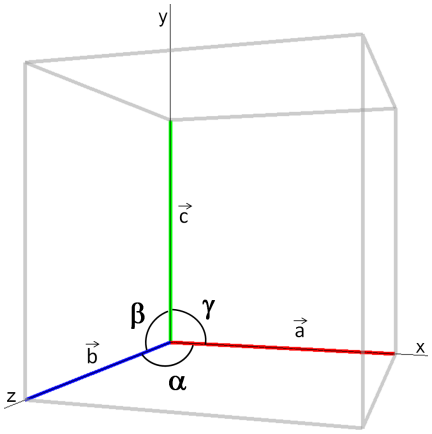
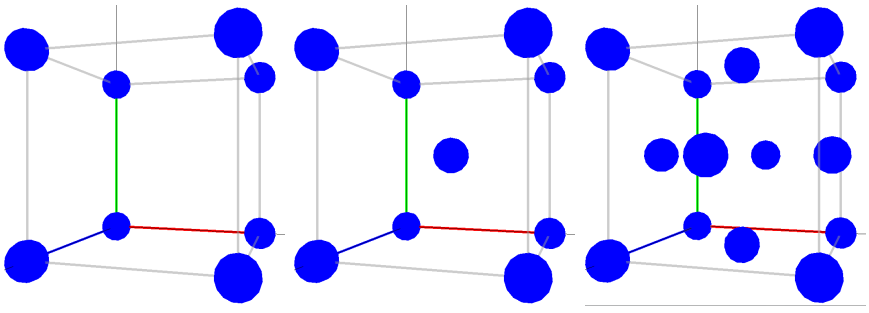


Figura 2.4: Sistema Cúbico.



(a) Cúbico P (b) Cúbico I (c) Cúbico F

Figura 2.5: Retículos Cúbicos com os vetores que definem sua rede em destaque.

2.2.2 *Sistema Tetragonal (tetragonal)*

Sistemas tetragonais, como os sistemas cúbicos, possuem os vetores ortogonais entre si, contudo, apenas dois parâmetros de rede são equivalentes: $|\vec{a}| = |\vec{b}| \neq |\vec{c}|$ e $\alpha = \beta = \gamma = 90°$. Dois retículos são definidos oriundos do Sistema Tetragonal: *Tetragonal P* e *Tetragonal I*.

A Figura 2.6 apresenta o Sistema Tetragonal e a Figura 2.7 exibe seus retículos.

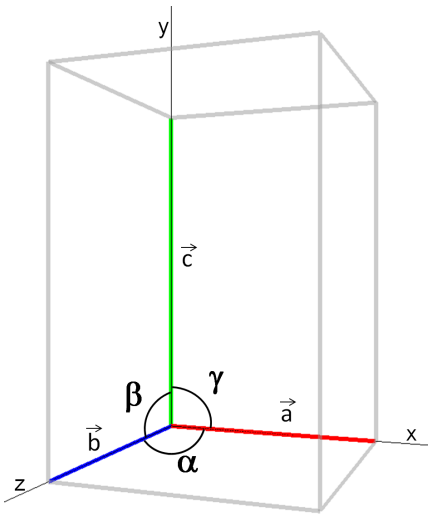
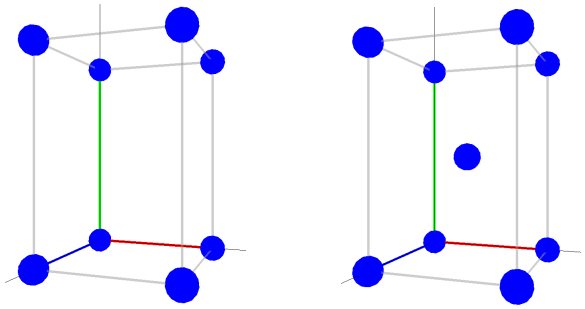


Figura 2.6: Sistema Tetragonal.



(a) Tetragonal P (b) Tetragonal I

Figura 2.7: Retículos Tetragonais com os vetores que definem sua rede em destaque.

2.2.3 *Sistema Ortorrômbico (orthorhombic)*

Sistemas ortorrômbicos possuem os parâmetros de rede distintos e os vetores ortogonais entre si: $|\vec{a}| \neq |\vec{b}| \neq |\vec{c}|$ e $\alpha = \beta = \gamma = 90°$. Os retículos *Ortorrômbico P*, *Ortorrômbico I*, *Ortorrômbico F* e *Ortorrômbico de bases centrada* (A, B ou C) são obtidos a partir desse sistema.

A Figura 2.8 apresenta o Sistema Ortorrômbico e a Figura 2.9 exibe retículos formados por ele.

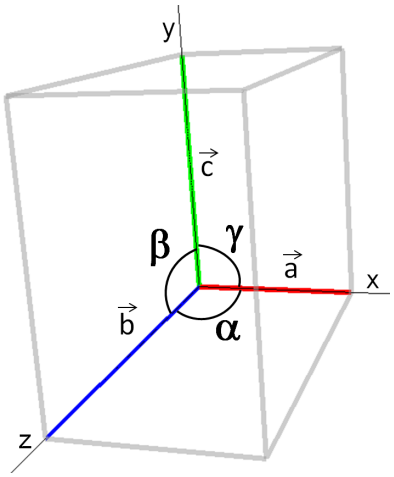
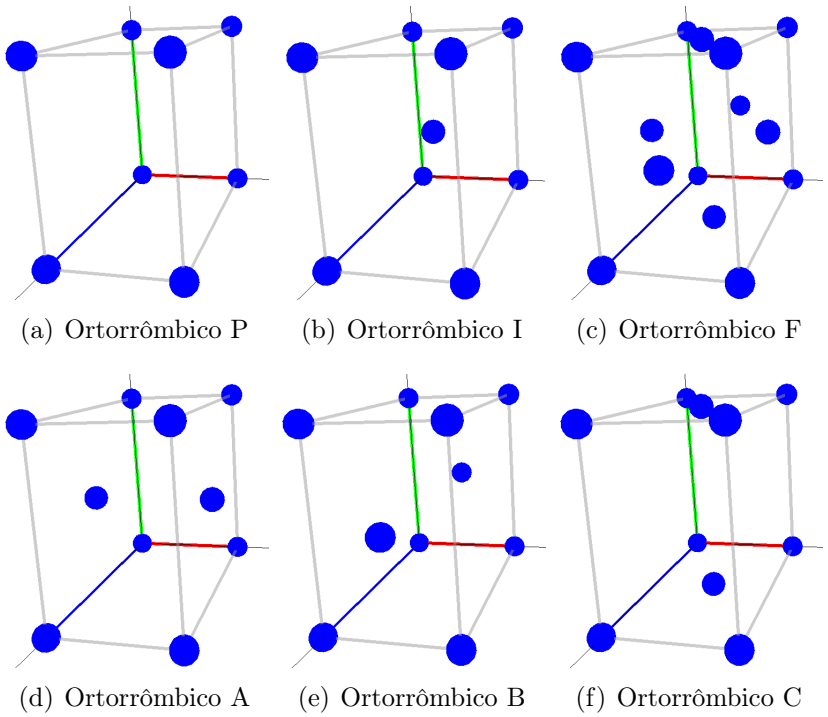


Figura 2.8: Sistema Ortorrômbico.



(a) Ortorrômbico P (b) Ortorrômbico I (c) Ortorrômbico F

(d) Ortorrômbico A (e) Ortorrômbico B (f) Ortorrômbico C

Figura 2.9: Retículos Ortorrômbicos com os vetores que definem sua rede em destaque.

2.2.4 Sistema Monoclínico (monoclinic)

Sistemas monoclínicos são caracterizados por possuírem parâmetros de rede distintos e dois dos três vetores ortogonais entre si: $|\vec{a}| \neq |\vec{b}| \neq |\vec{c}|$ e $\alpha = \gamma = 90° \neq \beta$. Dois retículos

são definidos oriundos do Sistema Monoclínico: *Monoclínico P* e *Monoclínico de bases centrada* (A, B ou C).

A Figura 2.10 apresenta o Sistema Monoclínico e a Figura 2.11 exibe seus retículos.

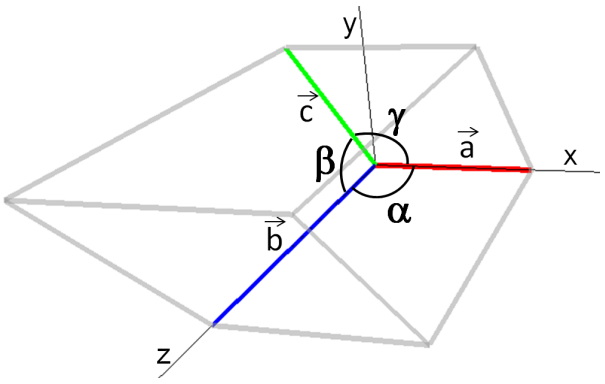
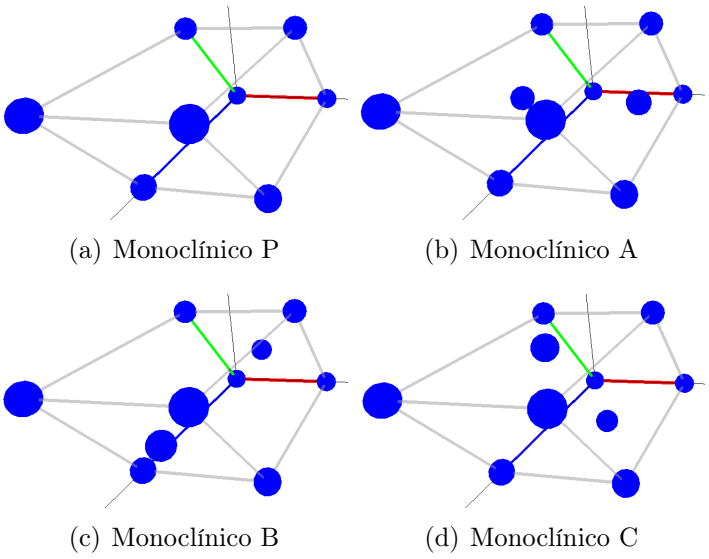


Figura 2.10: Sistema Monoclínico.



(a) Monoclínico P (b) Monoclínico A

(c) Monoclínico B (d) Monoclínico C

Figura 2.11: Retículos Monoclínicos com os vetores que definem sua rede em destaque.

2.2.5 Sistema Triclínico (triclinic)

Sistemas triclínicos possuem, por característica, parâmetros de rede distintos e os ângulos entre os vetores com valores diferentes: $|\vec{a}| \neq |\vec{b}| \neq |\vec{c}|$ e $\alpha \neq \beta \neq \gamma$. Define apenas um retículo, Triclínico P.

A Figura 2.12 apresenta o Sistema Triclínico e a Figura 2.13 exibe seu retículo.

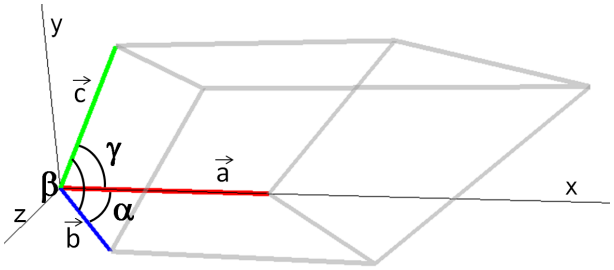


Figura 2.12: Sistema Triclínico.

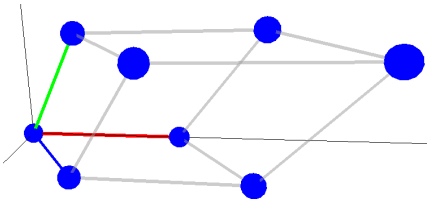


Figura 2.13: Retículo Triclínico P com os vetores que formam sua rede em destaque.

2.2.6 Sistema Romboédrico (rhombohedral)

Sistemas romboédricos são definidos por possuírem parâmetros de rede equivalentes e os ângulos entre os vetores com valores iguais, contudo o valor dos ângulos é diferente de 90°: $|\vec{a}| = |\vec{b}| = |\vec{c}|$ e $\alpha = \beta = \gamma \neq 90°$. O retículo Romboédrico P é definido por este sistema.

A Figura 2.14 apresenta o Sistema Romboédrico e a Figura 2.15 exibe seu retículo.

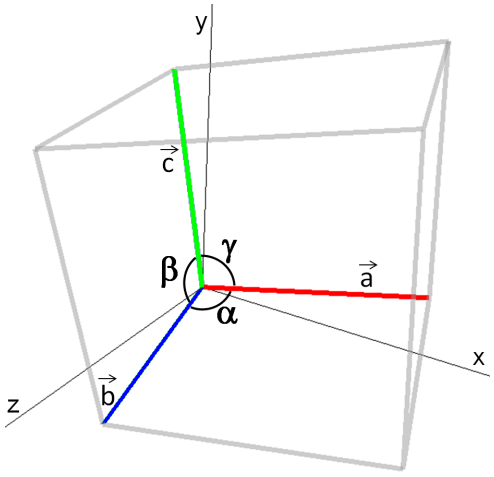


Figura 2.14: Sistema Romboédrico.

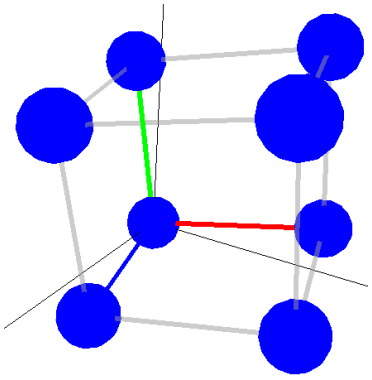


Figura 2.15: Retículo Romboédrico P com os vetores que formam sua rede em destaque.

2.2.7 Sistema Hexagonal (hexagonal)

Sistemas hexagonais são caracterizados por dois parâmetros de rede iguais com o ângulo formado por seus vetores igual a 120°. Os ângulos restantes são retos, 90°: $|\vec{a}| = |\vec{b}| \neq |\vec{c}|$ e $\alpha = 120°, \beta = \gamma = 90°$. Esse sistema apresenta apenas um retículo: Hexagonal P.

A Figura 2.16 apresenta o Sistema Hexagonal e a Figura 2.17 exibe seu retículo.

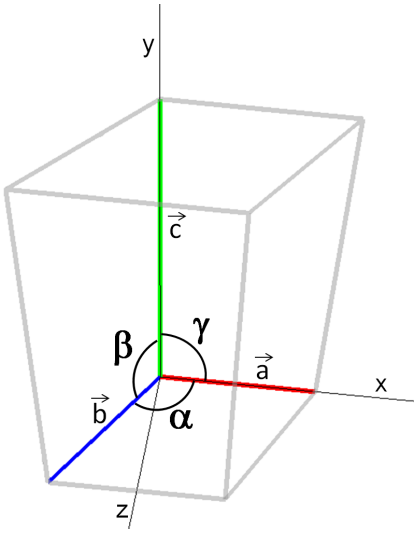


Figura 2.16: Sistema Hexagonal.

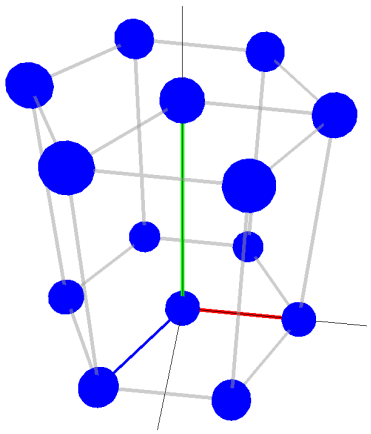


Figura 2.17: Retículo Hexagonal P com os vetores que formam sua rede em destaque.

3 DEFINIÇÃO DE LINGUAGENS E GRAMÁTICAS

3.1 Introdução

Linguagens, naturais ou artificiais, podem ser definidas como sendo a associação de símbolos básicos pertencentes a um conjunto [13]. Por exemplo, para a língua portuguesa, as palavras são formadas pela associação entre os símbolos pertencente ao conjunto $\{a, b, c, \cdots, z, A, B, \cdots, Z\}$. Vale salientar que algumas associações podem ser consideradas errôneas e que definição das associações corretas são feitas através de regras de produções descritas em uma gramática.

No âmbito da Teoria da Computação as linguagens podem ser classificadas de modo hierárquico segundo uma classificação proposta pelo linguista americano *Noam Chomsky*. Essa categorização possui quatro níveis e relaciona as linguagens de acordo com o nível de expressividade de sua gramática (Figura 3.1).

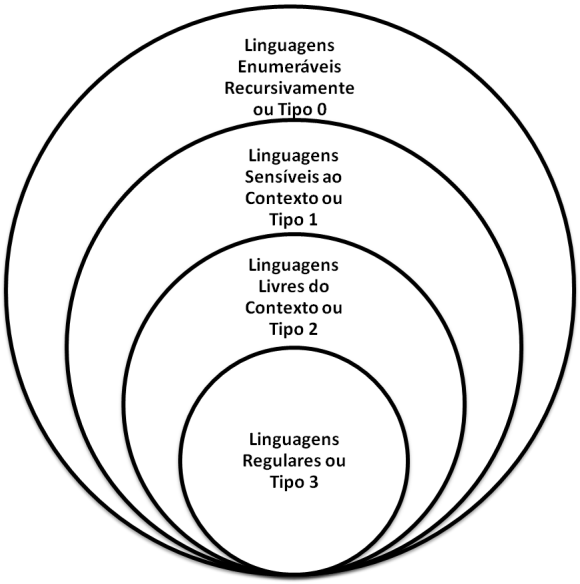


Figura 3.1: Hierarquia de *Chomsky*. Quanto menor o tipo, maior o poder de expressão da Linguagem.

Este capítulo tem por objetivo apresentar os conceitos relativos as linguagens e gramáticas. Isso se faz necessário para o entendimento da linguagem criada para a geração de

estruturas cristalinas (descrita no Capítulo 7). Serão apresentados elementos que possibilitam a descrição matemática de linguagens e gramáticas, definições e as características dos dois grupos de linguagens utilizados neste trabalho (Linguagens Regulares e Linguagens Livres do Contexto).

O conceitos serão apresentados de maneira sucinta. Uma apresentação mais detalhada pode ser obtida em [14] e [15].

3.2 Conceitos Básicos

Símbolo: Entidade abstrata básica que não é definida formalmente. Letras e dígitos são exemplos de símbolos.

Alfabeto: Conjunto finito de símbolos. Um alfabeto Σ contendo os símbolos a e b é representado na Equação 3.1.

$$\Sigma = \{a, b\} \tag{3.1}$$

Palavra ou Cadeia de Caracteres: Sequência finita de símbolos (pertencentes a um alfabeto) justapostos. Uma exceção para essa definição é a palavra vazia. Representada pelo símbolo ε, representa uma palavra sem símbolo.

O comprimento de uma palavra w é representado por $|w|$ e equivale a quantidade de símbolos que compõe w.

Σ^* representa o conjunto de todas as palavras pertencentes ao alfabeto Σ e Σ^+ equivale a $\Sigma^* - \{\varepsilon\}$.

Prefixo: Qualquer sequência de símbolos iniciais de uma palavra. *aba* é um prefixo da palavra *abacate*.

Sufixo: Qualquer sequência de símbolos finais de uma palavra. *cate* é um sufixo da palavra *abacate*.

Sub-palavra: Qualquer sequência contígua de símbolos de uma palavra. Prefixos e sufixos também são sub-palavras. *aba*, *baca* e *cate* são sub-palavras de *abacate*.

Linguagem Formal: Conjunto de palavras pertencentes a um alfabeto.

$\{a, aa, ab, aaa, aba, abb, \cdots\}$ denota a linguagem que representa todas as palavras com prefixo a sobre o alfabeto $\Sigma = \{a, b\}$

Concatenação: Operação binária sobre uma linguagem resultante da justaposição dos símbolos de duas de suas palavras. A concatenação entre as palavras *aba* e *cate* resulta na palavra *abacate*.

Supondo que v, w, t sejam palavras pertencentes a linguagem L, vw representa a concatenação da palavra v com a palavra w.

A concatenação satisfaz as seguintes propriedades:

- Associatividade — $v(wt) = (vw)t$
- Elemento Neutro à Esquerda e à Direita — $\varepsilon w = w = w\varepsilon$

Concatenação Sucessiva: Representa a concatenação de uma palavra com ela mesma um número determinado de vezes. Seja w uma palavra e n um número qualquer, w^n indica w concatenado sucessivamente n vezes com ela mesma.

Para $w \neq \varepsilon$ temos:

- $w^0 = \varepsilon$
- $w^n = w^{n-1}w$, para $n > 0$

Para $w = \varepsilon$ temos:

- $w^n = \varepsilon$, para $n > 0$
- w^0 não é definido

Gramática: Uma gramática é uma quádrupla ordenada $G = (V, T, P, S)$ onde:

- V é o conjunto finito de símbolos variáveis ou não-terminais;
- T é o conjunto finito de símbolos terminais, disjuntos de V;
- P é o conjunto finito de pares, denominados regras de produção, tal que a primeira componente é palavra de $(V \cup T)^+$ e a segunda componente é a palavra $(V \cup T)^*$;
- S é o elemento de V denominado variável inicial.

Uma regra de produção (α, β) é representada por $\alpha \to \beta$ e define as condições de geração das palavras da linguagem. Uma sequência de regras $\alpha \to \beta_1$, $\alpha \to \beta_2$, $\cdots$, $\alpha \to \beta_n$ pode ser representada resumidamente por $\alpha \to \beta_1|\beta_2|\cdots|\beta_n$.

Linguagem Gerada: Seja $G = (V, T, P, S)$ uma gramática. A união de todas as palavras geradas a partir dos símbolos pertencentes ao alfabeto T, deriváveis a partir do símbolo inicial S, formam a linguagem gerada pela gramática G. É denotada por $L(G)$ e pode ser definida como $L(G) = \{w \in T^* | S \Rightarrow^+ w\}$. Por exemplo, dada a gramática $G = (V, T, P, S)$, sendo:

- $V = \{S, V\}$;
- $T = \{a, b\}$ ou seja, conjunto finito de símbolos terminais disjuntos de V;
- $P = \{S \rightarrow aV, V \rightarrow aV|bV|\varepsilon\}$.

A gramática G gera a linguagem de todas as palavras do alfabeto $\{a, b\}$ que tem como prefixo a. Ou seja $L(G) = a(a|b)^*$.

3.3 Linguagens Regulares

As Linguagens Regulares compõem um conjunto restrito de linguagens, que na hierarquia de *Chomsky* são categorizadas como Tipo 3. Segundo Menezes [14], elas são oriundas de estudos biológicos de redes de neurônios e circuitos de chaveamentos. São utilizadas no desenvolvimento de analisadores léxicos (responsáveis pela identificação das estruturas básicas de uma linguagens como identificadores, números e palavras reservadas) de compiladores, em protocolos de comunicação e em editores de texto.

A descrição de uma Linguagem Regular pode ser feita de modo textual, contudo é mais empregado o uso de expressões simples denominadas expressões regulares. As expressões regulares são definidas como se segue:

- $\oslash$ é uma expressão regular que denota a linguagem vazia;
- ε é uma expressão regular que denota uma linguagem contendo a palavra vazia, ou seja $\{\varepsilon\}$;
- Um símbolo x pertencente ao alfabeto Σ é representado pela expressão regular x;
- Se r e s são expressões regulares que descrevem respectivamente as linguagens R e S, respectivamente, então:
  1. $(r|s)$ denota a linguagem $R \cup S$;

2. (rs) denota a linguagem uv tal que $u \in R$ e $v \in S$;

3. (r^*) denota a linguagem R^*.

Os parênteses podem ser omitidos, respeitando a seguinte precedência: Concatenação sucessiva (r^*), concatenação simples (rs) e união $(r|s)$.

A Tabela 3.1 exibe várias linguagens sobre o alfabeto $\Sigma = \{a, b\}$ descritas através de expressões regulares e modo textual.

Tabela 3.1: Exemplos de Linguagens Regulares descritas sobre o alfabeto $\Sigma = \{a, b\}$.

Expressão Regular	Descrição Textual
aa	Somente a palavra aa
$a(a\|b)^*$	Todas as palavras iniciadas pelo símbolo a
a^*ba^*	Todas as palavras que contém apenas um símbolo b
$(a\|\varepsilon)(b\|ba)^*$	Todas as palavras que não possuem dois símbolos a consecutivos

Linguagens Regulares são definidas como sendo qualquer linguagem que possa ser gerada a partir de uma Gramática Regular e o que define se uma gramática é regular ou não são suas regras de produção.

Dado gramática $G = \{V, T, P, S\}$ e sejam A e B elementos de V e w uma palavra de T^*, ela é dita regular se, e somente se, uma das condições abaixo é satisfeita:

1. Todas as regras de produção são da forma $A \to wB$ ou $A \to w$

2. Todas as regras de produção são da forma $A \to Bw$ ou $A \to w$

Na literatura, Gramática Regulares podem ser apresentadas com a denominação Gramática Lineares. Isso se deve ao fato das palavras especificadas pelas Gramáticas Regulares serem formadas através de antecessões (gramáticas lineares à esquerda, GLE's, obedecem a primeira condição descrita para Gramáticas Regulares) ou sucessões (gramáticas lineares à direita, GLD's, obedecem a segunda condição descrita para Gramáticas Regulares) de subpalavras compostas por terminais.

Cadeias de caracteres pertencentes a uma linguagem regular são reconhecidas por Autômatos Finitos Determinísticos (AFDs) e Autômatos Finitos Não-Determinísticos (AFNDs). AFD e AFND representam uma máquina com as seguintes características:

1. Um alfabeto de símbolos Σ;

2. Um conjunto finito de estados Q;

3. Uma função de transição:

   - $\delta : Q \times \Sigma \rightarrow Q$ para AFDs;
   - $\delta : Q \times \Sigma \rightarrow 2^Q$ para AFNDs.

   A função de transição é responsável pela mudança entre estados a medida que um elemento do alfabeto é consumido.

4. Um estado inicial q_0 pertencente a Q;

5. Um conjunto de estados finais F contido em Q.

AFND possibilita a transição de um estado para um conjunto de estados a medida que um símbolo é consumido, ao passo que em um AFD essa transição ocorre apenas para um estado.

Os autômatos, usualmente, são representados por um grafo onde os vértices (círculos) representam estados e as arestas ordenadas (setas) as transições. No centro de cada estado aparece seu nome e acima de cada transição é descrito qual símbolo é consumido. A Figura 3.2 exibe os elementos que representam visualmente um autômato finito.

(a) Estado Inicial

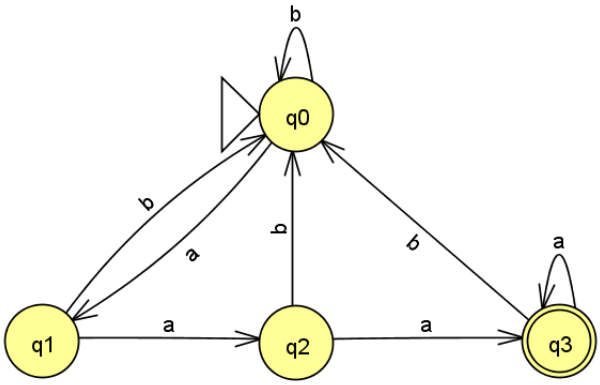
(b) Estado Intermediário

(c) Estado Final

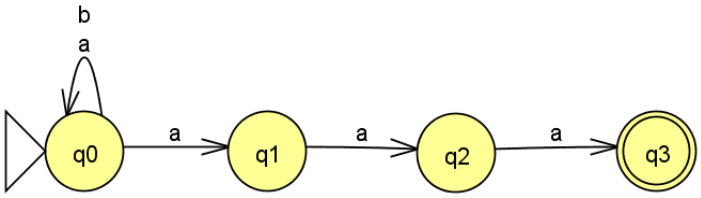
(d) Transição de Estado

Figura 3.2: Elementos constituindo da representação gráfica de um autômato.

AFD e AFND possuem o mesmo poder de expressão. AFND possuem compreensão mais simples, contudo possuem implementação computacional mais complicada (todos as transições possíveis devem ser analisadas para que uma entrada seja rejeitada). Segundo Aho [16], essa característica deve ser estudada antes da implementação de analisador léxico. A Figura 3.3 exibe dois autômatos que reconhecem a expressão regular $(a|b)*aaa$. Um AFD (3.3(a)) e um AFND (3.3(b))



(a) AFD



(b) AFND

Figura 3.3: Autômatos finitos que reconhecem a linguagem de todas as palavras com sufixo aaa no alfabeto $\Sigma = \{a, b\}$.

3.4 Linguagens Livres do Contexto

As Linguagens Livres do Contexto são categorizadas por *Chomsky* como sendo do Tipo 2. Elas representam ainda um conjunto limitado de linguagens, possuindo apenas mais expressividade que as Linguagens Regulares. Sua principal característica está no fato de conseguirem especificar corretamente estruturas sintáticas das linguagens de programação. Essas estruturas aparecem na construção de expressões aritméticas onde sub-expressões são delimitadas (através do uso de parênteses), na estruturação do fluxo de controle, em que comandos internos são inseridos como parte integrante de outros externos, e na estruturação do programa, em que blocos, módulos, procedimentos e funções são empregados para criar diferentes escopos. Uma Linguagem Livres do Contexto foi usada para especificar a linguagem de programação descrita neste trabalho.

Uma Linguagem Livre do Contexto é gerada a partir de uma Gramática Livre do Contexto. Como em Gramática Regular, o que define se uma gramática G pode ser categorizada como livre do contexto ou não são suas regras de produção. A categoria de uma gramática está relacionada com o grau de liberdade dado para a criação de suas regras de produção. A gramática G é livre do contexto se, e somente se, suas regras de

produção possuírem o seguinte formato: $A \rightarrow \alpha$, onde A é uma variável e α representa uma cadeia qualquer de terminais e variáveis.

A descrição de uma Linguagem Livre do Contexto é feita textualmente ou através das descrição de sua sintaxe (descrição das regras de produção de sua gramática). A literatura apresenta um formalismo para descrição de sintaxe denominado *Backus-Naum Form* (BNF). Esse formalismo foi criado e utilizado por *John Backus* para descrever a sintaxe da Algol em 1963 e consiste em um conjunto de regras com a seguinte forma: Os elementos entre os caracteres < e > representam os símbolos não terminais da gramática, regras de produção; o símbolo ::= indica a definição de uma regra; os símbolos terminais aparecem sem nenhuma formatação específica [17]. Usualmente é empregado o símbolo | ao inserir múltiplas definições para uma mesma regra. A Equação 3.2 apresenta uma regra da linguagem de geração de estruturas cristalinas descrita em BNF. A Linguagem completa descrita em BNF foi incluída no Apêndice A.

$$< \text{type} > \quad ::= \quad \text{int} \mid \text{float} \mid \text{bool} \tag{3.2}$$

4 COMPILADORES

4.1 Introdução

Segundo Aho [16], compilador é um programa destinado a leitura e compilação, tradução, de um código em outro. O código inicial (programa fonte) segue os padrões e as regras de produção definidas por uma linguagem denominada linguagem fonte. Da mesma maneira o código resultante (programa alvo) satisfaz as regras definidas por uma outra linguagem, a linguagem alvo.

Caso o código fonte contenha erros, é função do compilador informá-los e, se necessário, abortar o processo de compilação. Um fluxograma básico de funcionamento de um compilador é exibido na Figura 4.1

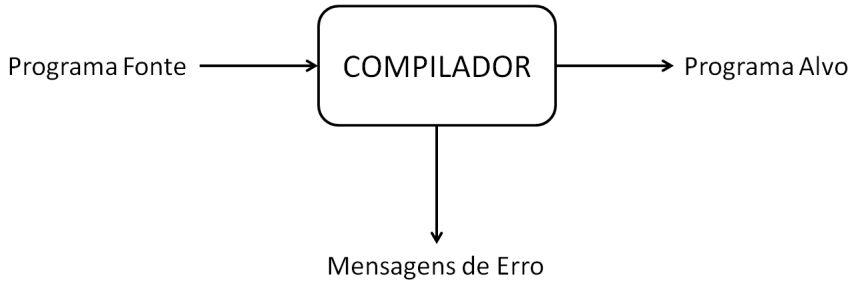


Figura 4.1: Fluxograma de funcionamento de um compilador.

A compilação é dividida em duas partes: análise e síntese. A análise tem por objetivo ler o programa fonte, reportar possíveis erros de cunho léxico, sintático e semântico e criar uma representação intermediária. Na síntese o programa alvo é gerado a partir da representação intermediária.

4.2 Fase Analítica

A análise do programa fonte é dividida em três etapas: *Análise Léxica*, *Análise Sintática* e *Análise Semântica*. A partir do trecho de código, descrito pelo Código 4.1, as etapas serão detalhadas.

```
valor = (x + 5) * 2;
```

Código 4.1: Trecho de código escrito na linguagem C.

4.2.1 Análise Léxica

Na análise léxica os caracteres que constituem o programa são lidos e organizados em *tokens*. Os *tokens* são compostos por um conjunto de caracteres, denominado lexema, e representam identificadores (nomes de variáveis e funções), palavras reservadas, pontuações e operadores definidos na linguagem fonte. A definição dos lexemas é descrita através de expressões regulares.

O *token* que representa um identificador, por exemplo, é definido na linguagem *C* como sendo uma letra seguida de mais letras ou dígitos. As expressões que o definem são mostradas a seguir.

$$
\begin{aligned}
LETRA &\rightarrow \text{‘A’|‘B’|‘C’|} \cdots \text{ ‘X’|‘Y’|‘Z’|‘a’|‘b’|‘c’|} \cdots \text{ ‘x’|‘y’|‘z’} \\
DIGITO &\rightarrow 0|1|2|3|4|5|6|7|8|9 \\
ID &\rightarrow LETRA(LETRA|DIGITO)^*
\end{aligned}
$$

Comentários, espaços em branco e caracteres especiais — como tabulações e quebras de linha — são desconsiderados, uma vez que esses trechos de códigos não são utilizados para a produção do programa alvo.

Um analisador léxico identificaria os seguintes *tokens* no Código 4.1:

1. O identificador “valor”
2. O operador atribuição “=”
3. O símbolo abre parêntese “(”
4. O identificador “x”
5. O operador soma “+”
6. O valor inteiro “5”

7. O símbolo fecha parêntese “)”

8. O símbolo do operador multiplicação “*”

9. O valor inteiro “2”

10. O símbolo de final de comando “;”

Vários lexemas distintos podem ser traduzidos em um mesmo *token*, portanto, é necessário que o compilador armazene esses lexemas. A Tabela 4.1 mostra os *tokens* e os lexemas armazenados a partir do código exemplo.

Tabela 4.1: Análise Léxica do código 4.1.

TEXTO LIDO	*TOKEN* GERADO	LEXEMA ARMAZENADO
valor	Identificador	valor
=	Atribuição	
(	AbreParentese	
x	Identificador	x
+	Soma	
5	Inteiro	5
)	FechaParentese	
*	Multiplicação	
2	Inteiro	2
;	PontoVirgula	

Fazendo alusão a língua portuguesa: O código fonte seria um texto e o analisador léxico um dispositivo que verifica se todas as palavras e símbolos contidos no texto podem ser usados na língua portuguesa.

Os erros captados pela análise léxica são oriundos da má formação de lexemas, ou seja, palavras não pertencentes a linguagem contidas no código.

4.2.2 Análise Sintática

Os *tokens* obtidos na análise léxica são analisados segundo as regras de produção definidas pela linguagem fonte. Essas regras determinam a correta relação entre os *tokens*. A análise sintática tem como objetivo verificar se a associação dos *tokens* gerados na analise léxica estão de acordo com as estruturas válidas da linguagem.

As regras de produção induzem uma análise hierárquica dos *tokens*, sendo usual a utilização de uma árvore (árvore sintática, ou gramatical) para representar o resultado

produzido pela análise. A Figura 4.2 exibe uma árvore gramatical criada a partir do Código 4.1.

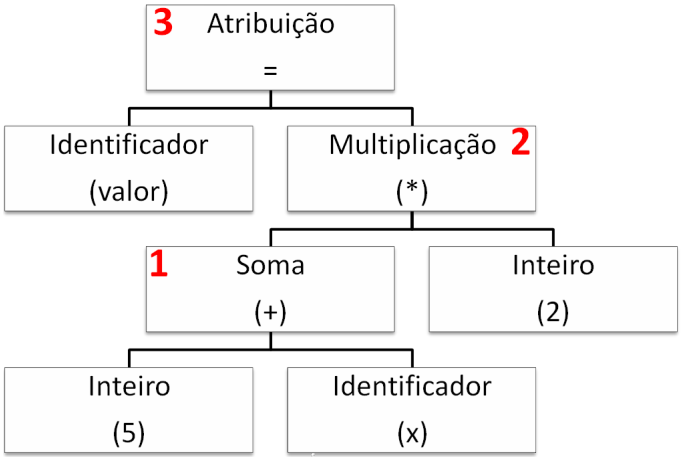


Figura 4.2: Árvore Gramatical gerada a partir do Código 4.1.

Os passos indicados numericamente na Figura 4.2 são descritos abaixo:

1. Verificação se existe na gramática uma regra que associa os *tokens* *Inteiro*, *Soma* e *Identificador*;

2. Verificação se existe uma associação entre o resultado obtido no item 1 e os *tokens* *Multiplicação* e *Inteiro*;

3. Por fim, se exite uma associação entre os *tokens* *Identificador*, *Atribuição* e o resultado obtido no item 2.

Novamente, fazendo alusão a língua portuguesa, o código fonte assumiria o papel de um texto e seria analisado no âmbito de frases, parágrafos e utilização correta de sinais de pontuação. A expressão a seguir exibe a interpretação sintática de uma frase.

$$\overbrace{\underbrace{\textit{Eles}}_{\textit{sujeito}}\ \underbrace{\textbf{joga}}_{(\textit{verbo})}\ \underbrace{\textit{bola}}_{\textit{predicado}}\ \underbrace{.}_{(\textit{ponto})}}^{\textit{Frase}}$$

A estrutura sintática da frase está correta (sujeito, verbo e predicado). Contudo, o verbo *jogar*, em negrito na expressão acima, está conjugado de maneira equivocada. É função da análise semântica verificar erros dessa natureza.

4.2.3 Análise Semântica

Cada nó da árvore gramatical corresponde a uma estrutura definida na linguagem e possui um tipo. A análise semântica percorre a árvore gramatical, indo dos nós das extremidades para os do topo, efetuando a verificação da relação entre esses tipos.

Exitem associações que, mesmo corretas sintaticamente, representam erros que devem ser descritos pelo compilador. Esses erros são descritos por regras semânticas, muitas vezes não expressas na linguagem. Por exemplo, na linguagem *C* uma variável (classificada pelo *token Identificador*) deve ser declarada antes de ser utilizada em qualquer expressão.

A Figura 4.3 exibe a árvore gramatical criada a partir do Código 4.1 com a ordem pela qual a análise semântica será efetuada.

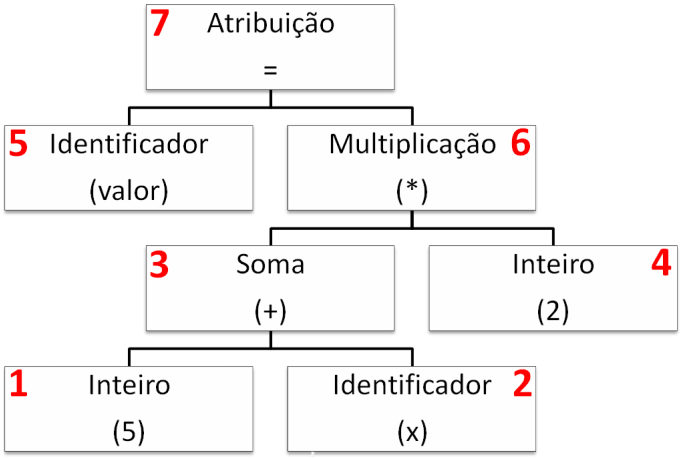


Figura 4.3: Análise semântica da árvore gramatical criada a partir do Código 4.1.

Os passos indicados numericamente na Figura 4.3 são descritos abaixo:

1. Retorna o tipo Inteiro;
2. Verifica se a variável representada pelo identificador *x* já foi declarada e retorna seu tipo;
3. Verifica se é possível efetuar soma, descrita pelo *token Soma*, entre os tipos retornados em 1 e 2;
4. Retorna o tipo Inteiro;
5. Verifica se é possível efetuar multiplicação (*token Multiplicação*) entre os tipos retornados em 3 e 4;

6. Verifica se a variável representada pelo identificador *valor* já foi declarada e retorna seu tipo;

7. Verifica se a atribuição pode ser feita entre os tipos retornados por 5 e 6.

4.3 Fase Sintética

A Fase Sintética consiste na transformação de uma estrutura denominada Árvore de Sintaxe Abstrata, ASA, na linguagem alvo. A transformação da ASA em linguagem alvo pode ser feita de duas maneiras: tradução ou interpretação.

Tradução: A ASA é transformada da linguagem alvo de maneira direta (quando a linguagem alvo é obtida diretamente da ASA) ou em partes (a partir da ASA é gerada outra representação intermediária, essa representação passa por um otimizador e a linguagem alvo é gerada).

Interpretação: As instruções descritas pela ASA são executadas, não há a criação de um nova linguagem. A ASA pode ser considerada a linguagem alvo.

A ASA é construída na Fase Analítica, a medida que o código fonte é analisado, e tem por objetivo representar, de forma hierárquica, a execução de todas as instruções descritas no código fonte. Um nó representa uma operação e seus filhos representam os argumentos para essa operação. O exemplo de uma ASA criada a partir do Código 4.1 é exibido na Figura 4.4.

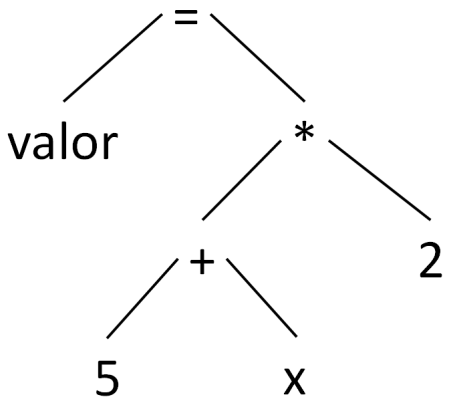


Figura 4.4: ASA criada a partir do Código 4.1.

Na fase analítica a verificação de erros depende da linguagem fonte. Na fase sintética, essa dependência se torna nula, uma vez que a ASA representa fielmente as instruções a

serem executadas. A síntese da ASA é feita se após o término de todas as análises (léxica, sintática e semântica) nenhum erro for encontrado.

5 REPRESENTAÇÃO DE OBJETOS

5.1 Introdução

Matematicamente, a geometria de um objeto (superfície ou sólido) pode ser descrita de duas formas, utilizando um modelo paramétrico ou um modelo implícito. Este capítulo tem por objetivo definir e fazer um comparativo entre os dois modelos existentes, explicar porque o modelo implícito foi adotado e apresentar as operações necessárias para efetuar combinações de objetos definidos de maneira implícita.

5.2 Modelo Paramétrico X Modelo Implícito

No modelo paramétrico todos os pontos do objeto são obtidos através da variação dos parâmetros de uma função dentro de um intervalo estabelecido. Por exemplo, um círculo unitário em $2D$ pode ser descrito na forma paramétrica através da Equação 5.1:

$$f(\theta) = (\cos\theta, \sin\theta), \theta \in [0, 2\pi] \tag{5.1}$$

A representação gráfica da Equação 5.1 é exibida na Figura 5.1.

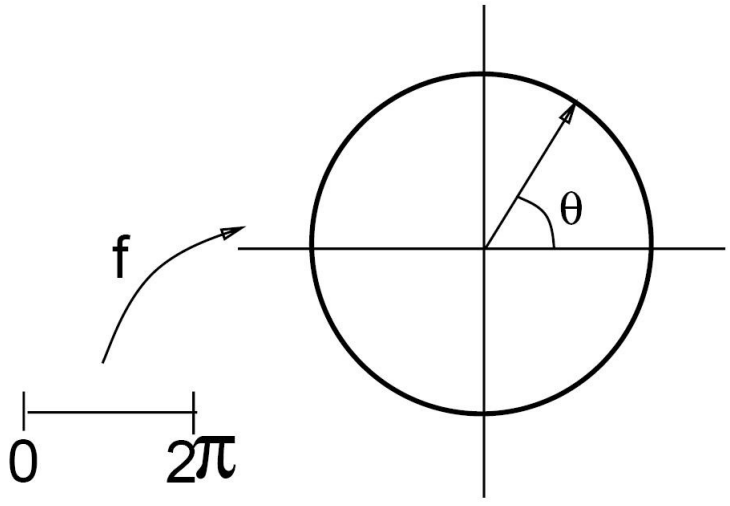


Figura 5.1: Representação paramétrica do círculo de raio unitário. Retirado de [18].

No modelo implícito ocorre uma caracterização dos pontos no espaço. O conjunto de pontos que define o objeto é especificado indiretamente, através de uma função de

classificação.

Um subconjunto $O \subset \Re^n$ é chamado um objeto implícito se existe uma função $F : U \to \Re^k, O \subset U$ e um subconjunto $V \subset \Re^k$, de tal forma que $O = F^{-1}(V)$. Isto é $O = \{p \in U : F(p) \in V\}$ [18]. Um objeto implícito é dito válido se define uma superfície em $\Re^n$.

Sendo assim, o círculo definido de forma paramétrica acima poderia ser definido implicitamente através de equação $F(x, y) = 0$, onde:

$$F(x, y) = x^2 + y^2 - 1, x, y \in \Re \tag{5.2}$$

A Figura 5.2 exibe o círculo unitário descrito pela equação 5.2.

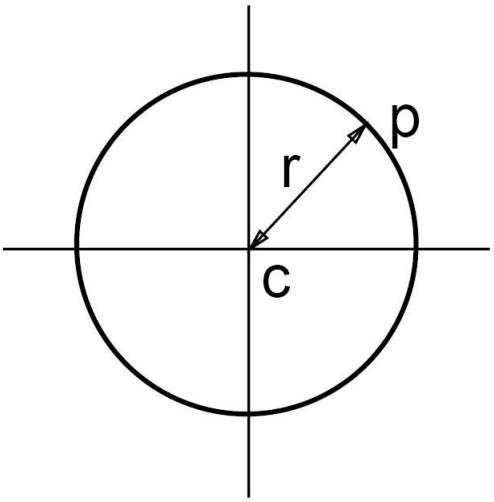


Figura 5.2: Representação implícita do círculo de raio unitário. Adaptado de [18].

O conjunto de pontos (x, y) que satisfazem $F(x, y) = 0$ equivalem ao círculo. A função F classifica os pontos do plano de acordo com a Equação 5.2. A substituição das coordenadas de um ponto $p = (x, y)$, qualquer, na equação $F(x, y) = x^2 + y^2 - 1$ indica sua posição em relação ao círculo definido implicitamente:

- p pertence ao interior do círculo caso $F(p) < 0$;
- p pertence ao borda do círculo caso $F(p) = 0$;
- p é exterior ao círculo caso $F(p) > 0$;

Vale salientar que não se pode afirmar qual representação é a melhor. A escolha depende da finalidade, contexto e dos tipos de operações que serão empregadas aos objetos.

Para desenhar uma aproximação de um círculo precisamos ligar linhas (retas) passando pelos pontos que compõem sua borda. Isto pode ser feito facilmente usando a forma paramétrica, variando o valor de θ pelo intervalo $[0, 2\pi]$, de modo que obtemos rapidamente os pontos pelos quais as linhas devem passar. Em contrapartida, para determinarmos, por exemplo, interseções entre objetos precisaremos testar se os pontos internos de um objeto também são internos no outro. Esta é uma operação simples usando a forma implícita, bastando apenas verificar o sinal de $F(p)$.

O espaço descrito pela linguagem apresentada neste trabalho será discretizado segundo critérios definidos pelo programador, na escrita do código. Portanto, os objetos serão representados através de pontos no espaço. Nesse cenário um modelo implícito que classifica imediatamente um ponto em relação a um objeto (interior, exterior ou pertencente a borda) apresenta o melhor desempenho. A linguagem descrita neste trabalho utiliza o modelo implícito para definição da localização espacial das estruturas cristalinas.

5.3 Combinando Objetos Implícitos

Objetos implícitos podem ser agrupados formando uma composição. Isto é realizado de acordo com um conjunto de regras que descrevem esses agrupamentos. Operações algébricas com um conjunto de funções implícitas produzem uma outra função correspondente à composição das operações com os objetos implícitos.

Uma função $F : \Re^n \rightarrow \Re$ associada a um objeto implícito O e um valor regular A é dita bem definida se sua imagem inversa $F^{-1}(A)$ é um objeto implícito válido. Dada uma função F, $F(x) \in (-\infty, \infty)$, então αF e $F + c$ são funções bem definidas. F^{β} seria uma função bem definida somente se F fosse definida apenas para valores positivos [18].

Se F_1 e F_2 são funções bem definidas então qualquer combinação entre elas, utilizando as operações descritas abaixo, geram uma outra função bem definida F_x:

- Soma: $F_1 + F_2$;
- Produto: $F_1 * F_2$;
- Composição: $F(F_1, \ldots, F_k)$;
- Máximo: $Max(F_1, F_2)$;
- Mínimo: $Min(F_1, F_2)$.

Vale salientar que as operações αF e F^{α}, com α inteiro e positivo, são casos particulares podendo ser expressas utilizando as operações acima.

Um método eficaz para se construir objetos complexos é a combinação de objetos mais simples através da utilização de operadores CSG, *Construtive Solid Geometry*, que se baseiam em operações entre pontos e conjuntos. A construção se dá através da união e intersecção de objetos primitivos. Outras operações, tais como a diferença, são definidas com a utilização do complemento ($complemento(F) = \overline{F} = -F$) [19].

5.3.1 Descrição funcional

P_i é um conjunto de funções primitivas de $\Re^n \rightarrow \Re$ de classe C^1. O conjunto de funções S_j geradas por P_i é definida como se segue:

- $P_i \subset S_j$;
- $F \in S_j \Rightarrow -F \in S_j$;
- $F_1, F_2 \in S_j \Rightarrow Max(F_1, F_2) \in S_j$;
- $F_1, F_2 \in S_j \Rightarrow Min(F_1, F_2) \in S_j$;

Se $F \in S_j$ não é uma função primitiva, então F é gerada a partir de funções primitivas através de operações Max e Min.

5.3.2 Objetos implícitos gerados a partir de CSG

Um sólido construído a partir de CSG é definido como qualquer conjunto de pontos em $\Re^n$ que satisfaça $F(x) \leq 0$ para algum $F \in S_j$. As operações booleanas são definidas como:

- $F_1 \cup F_2 = Min(F_1, F_2)$;
- $F_1 \cap F_2 = Max(F_1, F_2)$;
- $F_1 \setminus F_2 = F_1 \cap \overline{F_2} = Max(F_1, -F_2)$.

A Figura 5.3 representa as operações descritas acima.

Usualmente um objeto criado a partir de CSG pode ser representado por uma árvore (Figura 5.4) cujos nós internos representam as operações booleanas e as folhas representam os objetos primitivos utilizados. A raiz é o objeto gerado.

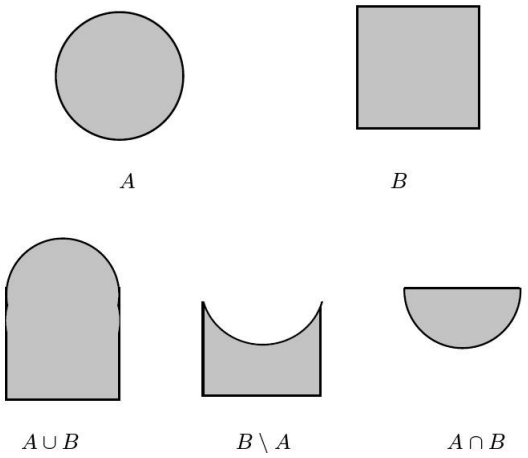


Figura 5.3: Operações CSG. Retirado de [18].

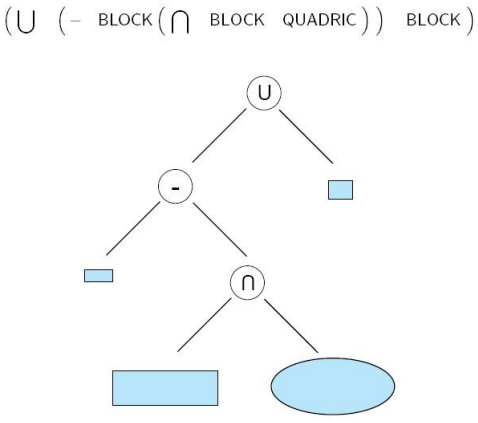


Figura 5.4: Representação de operações CSG utilizando uma árvore. Retirado de [18].

6 DETECÇÃO DE INTERSEÇÃO ENTRE OBJETOS

6.1 Introdução

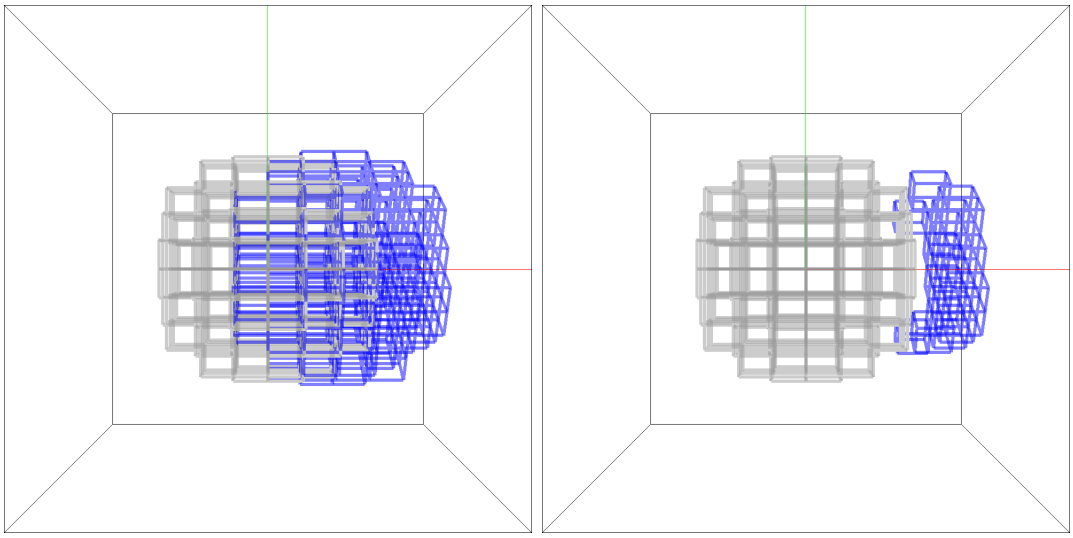
A detecção de interseção entre objetos é um campo bastante estudado e difundido no meio computacional. Tendo em vista sua aplicabilidade na computação gráfica e na robótica, existe uma vasta literatura referente a algoritmos que calculam ou detectam interseção entre objetos, estando eles estáticos ou em movimento.

A linguagem, proposta neste trabalho, provê mecanismos que possibilitam a criação de objetos, denominados estruturas cristalinas, com as mais variadas formas. Esses objetos são compostos por unidades básica (células unitárias), apresentadas no Capítulo 2. Na linguagem a criação de uma estrutura é dividida em duas etapas: definição e inserção. O desenvolvimento e a utilização da linguagem serão abordados nos capítulos subsequentes.

As interseções entre os objetos provenientes da etapa de definição são abordas e solucionadas pelos conceitos presentes no Capítulo 5.

As interseções provenientes da segunda etapa, inserção, representam colisões entre células unitárias de dois objetos distintos inseridos no espaço. Para tanto, se faz necessária a utilização de algum algoritmo capaz de verificar se duas células unitárias (provenientes de objetos distintos) se interceptam ou não, provendo mecanismos de correção (Figura 6.1).

O objetivo deste capítulo é apresentar o algoritmo Gilbert-Johnson-Keerthi (GJK), descrito em [20], utilizado neste trabalho para verificar interseção entre células unitárias de objetos distintos.



(a) Interseção entre células unitárias de dois objetos.

(b) Objetos com as interseções removidas.

Figura 6.1: Tratamento de interseções entre células de objetos distintos.

6.2 Algoritmo para cálculo de distância Gilbert-Johnson-Keerthi (GJK)

O algoritmo GJK é um método interativo de cálculo de distância entre objetos convexos [20], sendo um dos métodos mais eficazes e rápidos de detecção de colisões [21, 22]. Embora, originalmente, o algoritmo GJK tenha sido apresentado de maneira bastante técnica, sua implementação e entendimento não constituem uma tarefa árdua, bastando apenas a compreensão de alguns conceitos relacionados a análise convexa e a teoria de conjuntos convexos na qual o algoritmo se baseia [23, 24]. Alguns conceitos relevantes para o entendimento do algoritmo são apresentados a seguir.

6.2.1 Regiões de Voronoi

Dado um conjunto discreto S de pontos em um plano, a região de *Voronoi* de um ponto $p \in S$ é definida como o conjunto de pontos no plano mais próximos a p do que qualquer outro ponto pertencente ao conjunto S. Os conceitos de Regiões e Diagramas de *Voronoi* são amplamente utilizados na pesquisa de vizinhos, em geometria computacional e algoritmos de detecção interseção entre objetos [23].

Um triângulo, definido a partir de três pontos (V_1, V_2 e V_3), possui sete regiões de *Voronoi* (6.2). Elas definem a relação de proximidade de qualquer ponto no espaço com

os elementos que compõe o triângulo, por exemplo: A região E_2 engloba todos os pontos que estão mais próximos da aresta $\overline{V_1V_3}$; a região V_1, os pontos mais próximos do vértice V_1; e a região F, os pontos mais próximos da face do triângulo.

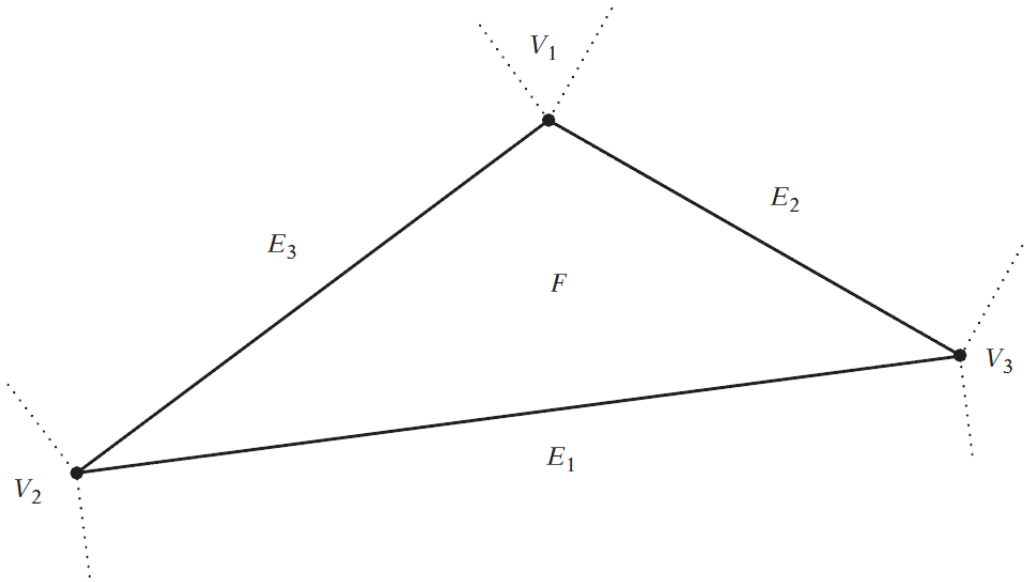


Figura 6.2: Três pontos determinam sete regiões de *Voronoi* em um plano. Um compreendida pela face F, três definidas pelas arestas $E1$, $E2$ e $E3$ e três definidas pelos vértices $V1$, $V2$ e $V3$. Retirado de [23].

6.2.2 Soma e Diferença de Minkowski

Dados dois conjuntos de pontos A e B, a soma de *Minkowski* é definida por:

$$A \oplus B = \{a + b : a \in A, b \in B\}$$

onde $a + b$ é o vetor soma dos vetores de posição a e b. A soma de *Minkowski* pode ser interpretada como sendo a região compreendida pela translação dos pontos de A em relação a B e vice-versa. A Figura 6.3 exemplifica a soma de *Minkowski*.

De forma análoga podemos definir a diferença de *Minkowski*. Dado dois conjuntos de pontos A e B a diferença de *Minkowski* é definida por:

$$A \ominus B = \{a - b : a \in A, b \in B\}$$

Geometricamente, a diferença de *Minkowski* $A \ominus B$ corresponde a $A \oplus (-B)$, sendo obtida através da soma de A com a reflexão de B sobre a origem. A Figura 6.4 apresenta

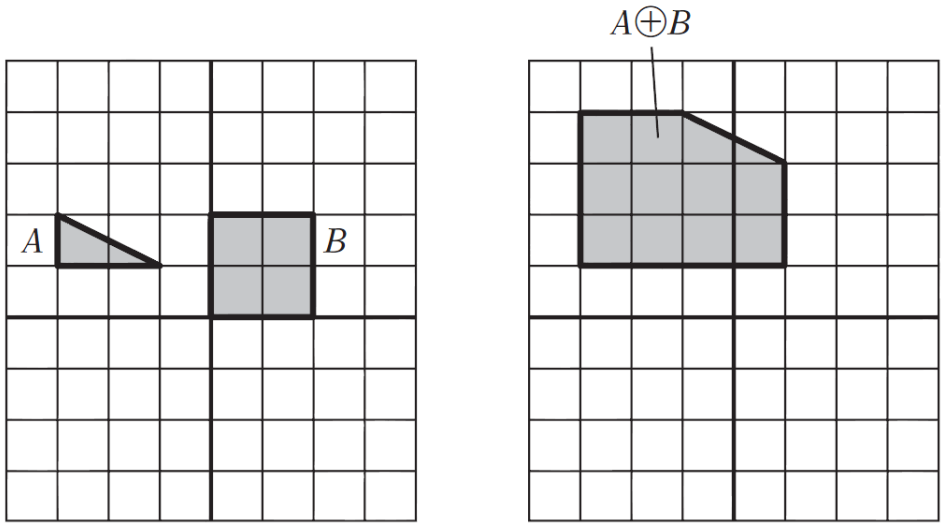


Figura 6.3: Soma de *Minkowski* definida pelos conjunto A e B. Retirado de [23].

a diferença de *Minkowski*.

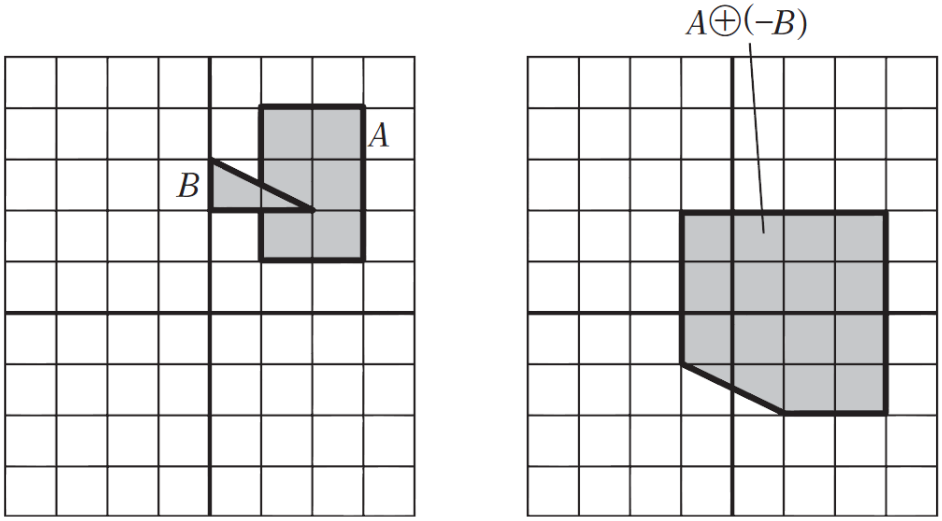


Figura 6.4: Diferença de *Minkowski* definida pelos conjunto A e B. Retirado de [23].

A diferença de *Minkowski* é muito utilizada em algoritmos de detecção de interseção entre objetos uma vez que, se A e B se interceptam, então $A \ominus B$ contém a origem. Isso é facilmente compreendido dada a existência de um ponto p, comum a A e B, que aplicado a definição da diferença de *Minkowski* resulta na origem.

6.2.3 Características do Algoritmo

O algoritmo GJK é baseado no fato de que a distância de separação entre dois objetos convexos A e B equivale à distância entre a diferença de *Minkowski* C, $C = A \ominus B$, e

a origem (Figura 6.5). O algoritmo busca pelo ponto $p \in C$ mais próximo a origem. O algoritmo não calcula todos os pontos da diferença de *Minkowski* C, tarefa essa que demandaria grande custo computacional. Ele utiliza uma função suporte (*support function*) para $C = A \ominus B$ em relação a uma direção, informada, d [23, 21]. Função suporte $S_C(d)$ é apresentada em Ericson [23] como uma associação que mapeia a direção C em um ponto contido em um conjunto convexo C. Em outras palavras, a função suporte fornece um ponto $p \in C$ mais afastado na direção d (Figura 6.6).

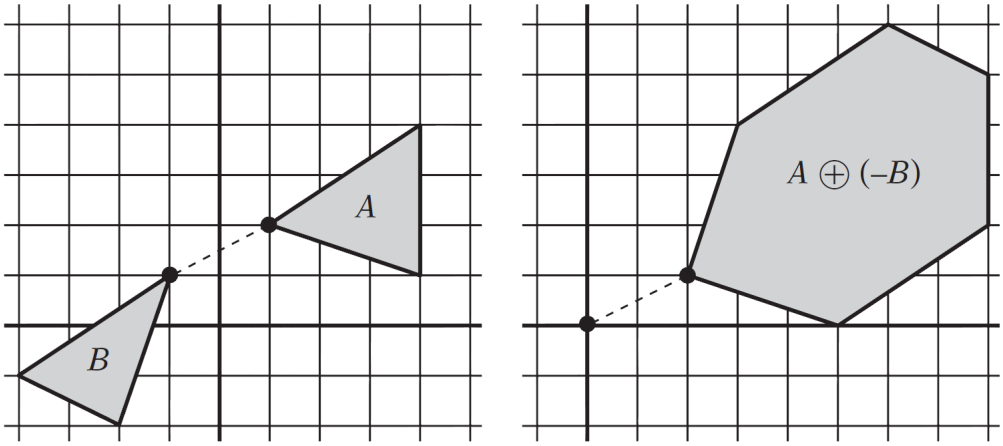


Figura 6.5: A distância entre A e B equivale a distância entre $A \ominus B$ e a origem. Retirado de [23].

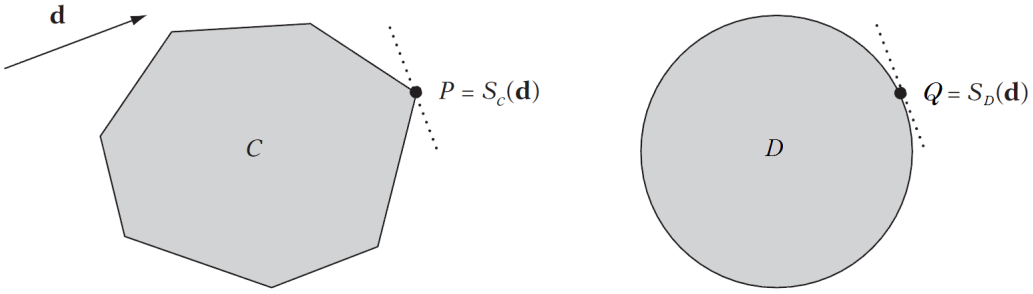


Figura 6.6: Função suporte definida para dois conjuntos, C e D, na direção d fornecida. Adaptado de [23].

6.2.3.1 Funcionamento do Algoritmo

O algoritmo recebe dois conjuntos convexos referentes aos pontos dos objetos A e B, dos quais se quer calcular a distância. Além dos objetos, o algoritmo armazena a distância encontrada e um conjunto convexo Q (denominado *simplex*) que tem por objetivo guardar os pontos pertencentes a diferença de *Minkowski* dos objetos que estão sendo utilizados no cálculo de distância. Inicialmente, o conjunto Q está vazio.

O algoritmo segue os seguintes passos:

1. Inicialmente é incorporado ao conjunto Q um ponto aleatório, w_0, pertencente a diferença de *Minkowski* dos objetos. Com apenas um ponto verificado, a distância máxima equivale a distância deste ponto a origem;

2. É adicionado ao *simplex*, conjunto Q, um ponto, w, definido por $w = S_{A \ominus B}(-v)$. w é o ponto calculado pelo função suporte da diferença de *Minkowski* na direção oposta ao vetor que representa o último ponto incluído no *simplex*;

3. Determinar um ponto v (contido no objeto descrito pelo *simplex*) mais próximo da origem e remover do *simplex* os pontos que não são necessários para a sua representação. As regiões de *Voronoi* do objeto formado pelos pontos do *simplex* são analisadas a fim de verificar a relação de proximidade da origem com o objeto. A seguintes relações são factíveis:

   Origem contida no objeto: v corresponde a origem e todos os pontos do simplex são removidos;

   Origem mais próxima de um vértice: v corresponde ao vértice e todos os pontos, exceto ao correspondente ao vértice, são removidos do *simplex*;

   Origem mais próxima de uma aresta: v corresponde ao ponto da aresta mais próximo à origem. Todos os pontos exceto os que representam a aresta são removidos;

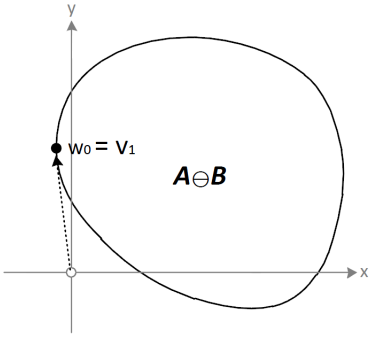
   Origem mais próximo a um plano: v corresponde ao ponto do plano mais próximo à origem. Os pontos do simplex que não pertencem ao plano são removidos;

4. Se v é a origem, os objetos se interceptam. O algoritmo termina indicando interseção (distância igual a zero);

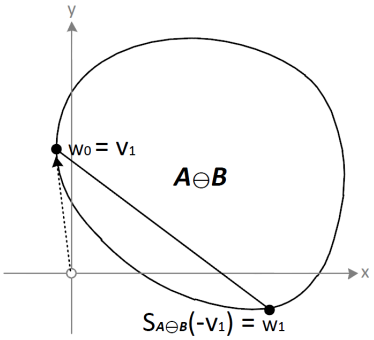
5. Atualiza a distância caso a distância entre v e a origem seja menor que a distância armazenada. Caso contrário o algoritmo termina indicando que os objetos não se interceptam e que a separação entre eles corresponde a distância armazenada;

6. Volta para o passo 2;

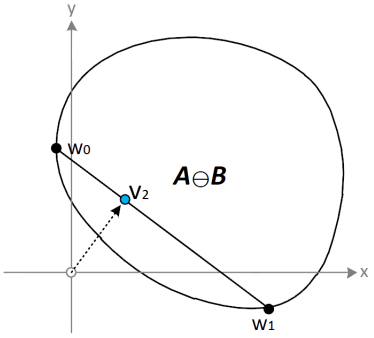
A Figura 6.7 demostra o funcionamento de algoritmo GJK descrito acima apresentando duas iterações do mesmo.



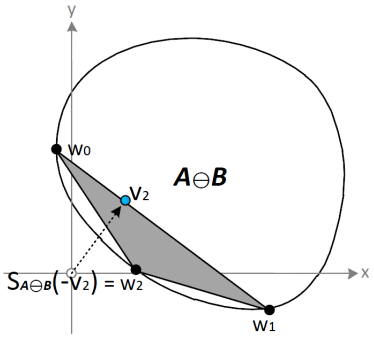
(a) Passo 1: Escolha de um ponto em $A \ominus B$.



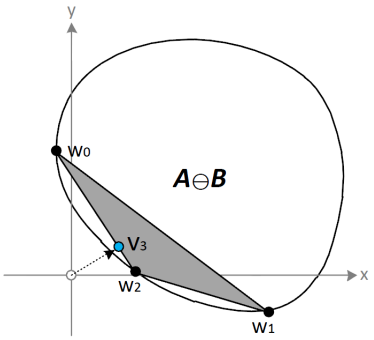
(b) Passo 2: Adiciona o ponto $w_1 = S_{A\ominus B}(-v_1)$ ao *simplex*.



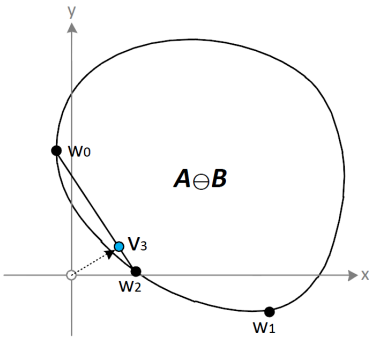
(c) Passos 3, 4, 5 e 6: Determina o ponto mais próximo da origem, v_2, não remove nenhum elemento do *simplex*, efetua as atualizações necessárias e volta ao Passo 2.



(d) Passo 2: Adiciona o ponto $w_2 = S_{A\ominus B}(-v_2)$ ao *simplex*.



(e) Passos 3: Determina o ponto mais próximo da origem, v_3.



(f) Passos 3, 4, 5 e 6: Remove o ponto w_1 (não utilizado para representar v_3) do *simplex*, efetua as atualizações necessárias e volta ao Passo 2.

Figura 6.7: Execução de duas iterações do algoritmo GJK.

7 A LINGUAGEM

7.1 Introdução

Uma linguagem descreve um conjunto organizado e coerente de instruções e regras pelo qual se expressam as ações executáveis por um computador. Para a construção de uma linguagem é necessário estudar e compreender o resultado final que se deseja obter: Por quais meios, instruções, esse resultado será obtido. A linguagem desenvolvida tem os seguintes objetivos:

- Possibilitar a definição de um Sistema Cristalino de Bravais;
- Possibilitar a definição de um sólido através da composição das definições de outros sólidos já definidos. Por exemplo, através de união ou interseção das funções que caracterizam os sólidos;
- Realizar operações e transformações com sólidos já definidos;
- Possibilitar, ao final de sua computação, a geração de uma Estrutura Cristalina;
- Não ser ambígua. Dado que o código não foi alterado, o resultado da computação deve gerar a mesma estrutura.

Este capítulo tem por objetivo apresentar a linguagem criada, exibindo suas estruturas gerais (comentários, variáveis, tipos básicos, funções pré-definidas, operadores, estruturas de controle e a instrução *insert*) e o processo de criação de uma estrutura cristalina através de quatro etapas:

Criação de Propriedades: Parte do código onde o programador define as propriedades para os átomos ou moléculas da estrutura.

Definição das características da estrutura: Trecho do código no qual são definidos a qual retículo de bravais a estrutura pertence, que vetores formam o retículo e qual a posição de cada átomo ou molécula da base.

Descrição de Sólidos: O programador define os sólidos ou composições que serão utilizados. Um sólido é constituído por uma expressão que define seu conjunto de pontos. Uma composição é um trecho de código onde ocorre uma agregação de sólidos.

Construção da Estrutura: Na construção da estrutura o programador define qual será a região espacial analisada e define quais sólidos ou composições devem ser inseridos nela.

7.2 Estrutura Geral da Linguagem

7.2.1 Comentários

Comentário são trechos de código destinados a explicações e conteúdos que não devem ser analisados pelo compilador. Na linguagem todo conteúdo escrito entre os elementos *{* e *}* são considerados comentários. O Código 7.1 exibe um exemplo de comentário.

```
{Exemplo de comentário
 Tudo digitado aqui é desconsiderado pelo compilador}
```

Código 7.1: Código contendo um comentário.

7.2.2 Variáveis

Na linguagem, as variáveis (também chamadas de identificadores) são definidas como qualquer cadeia de letras e dígitos, que não começam com um dígito. Esta definição está de acordo com a definição de nomes de variáveis na maioria das linguagens existentes. As palavras-chaves listadas na Tabela 7.1 não podem ser utilizadas como identificadores.

Vale ressaltar que a linguagem diferencia letras minúsculas de maiúsculas: *begin* é uma palavra reservada, mas *Begin* e *BEGIN* são dois nomes válidos diferentes.

Na Tabela 7.2 são apresentadas as cadeias de caracteres que definem outros itens léxicos.

O escopo das variáveis se restringe apenas ao local, o conteúdo de uma variável é acessível apenas no trecho de código onde a mesma foi iniciada, não sendo possível ao usuário definir variáveis com escopo global.

Tabela 7.1: Palavras Reservadas.

lattice	begin	end	triclinic	monoclinic
orthorhombic	tetragonal	rhombohedral	hexagonal	cubic
primitive	body	face	basea	baseb
basec	base	property	int	float
bool	true	false	solid	composite
insert	while	override	nooverride	error
cos	sin	tan	acos	asin
atan	atan2	cosh	sinh	tanh
exp	frexp	ldexp	log	log10
modf	pow	sqrt	ceil	fabs
floor	fmod	union	intersection	difference
not	rotatex	rotatey	rotatez	scale
shear	translate	x	y	z

Tabela 7.2: Outros itens léxicos.

+	−	*	/	=
,	;	==	<>	<
<=	>=	>	(	)
&&	||	{	}	

A atribuição de um valor a uma variável é feita da seguinte maneira:
ID = exp

- ID é o nome da variável;
- = é operador ASSIGN, atribuição;
- exp representa uma outra variável, um valor do tipo *Number* ou um valor do tipo *Solid*.

7.2.3 Tipos Básicos

A linguagem não prevê nenhum tipo de declaração de dados, ou seja, o tipo da variável é determinado dinamicamente. Existem dois tipos de dados primitivos na linguagem. São eles: *Number* e *Solid*.

O tipo *Number* se destina ao armazenamento de valores numéricos, não existindo distinção entre números inteiros e números reais. Todos os números são armazenados como sendo reais. O tipo *Solid* é utilizado para o armazenamento de sólidos. Sólidos representam uma porção do espaço descrito por uma ou mais funções.

7.2.4 Operadores Relacionais, Lógicos e Aritméticos

Todos os operadores relacionais, lógicos e aritméticos só podem ser utilizados com o tipo de dados *Number*. Uma exceção para esta regra é o operador = (atribuição) que pode ser utilizado pelos dois tipos básicos de dados. Os operadores relacionais são utilizados para comparação entre números, os lógicos em sentenças lógicas (resultante dos operadores relacionais) e os aritméticos descrevem as operações aritméticas básicas. Os operadores são exibidos na Tabela 7.3.

Tabela 7.3: Operadores Relacionais, Lógicos e Aritméticos.

OPERADOR	SÍMBOLO	SIGNIFICADO
EQUAL	==	igualdade
NOT_EQUAL	<>	diferença
LT	<	menor
LE	<=	menor ou igual
GE	>=	maior ou igual
GT	>	maior
AND	&&	E lógico
OR	||	OU lógico
PLUS	+	soma
MINUS	−	subtração
STAR	*	multiplicação
SLASH	/	divisão
ASSIGN	=	atribuição

7.2.5 Funções Pré-Definidas

A linguagem provê uma série de funções para a manipulação dos dados do tipo *Number* e *Solid*. As funções de manipulação do tipo *Number* são análogas às funções definidas pela biblioteca *math* na linguagem C, a Tabela 7.4 exibe suas especificações.

As funções para manipulação de sólidos (dados do tipo *Solid*) definem todas as operações entre sólidos que a linguagem possui: União, Interseção, Diferença, Negação, Rotação em torno de um eixo, Translação, Escalonamento e Cisalhamento. Estas operações são descritas a seguir:

União: É definida pela função *union* cuja assinatura é *Solid union(Solid s1, Solid s2)*. A função retorna um sólido correspondente a união dos sólidos *s1* e *s2*. A Figura 7.1(c) exemplifica a operação de união entre dois objetos.

Tabela 7.4: Funções Pré-definidas na linguagem.

FUNÇÃO	ASSINATURA	DESCRIÇÃO
cos	*Number cos(Number x)*	Retorna o cosseno de um ângulo x expresso em radianos
sin	*Number sin(Number x)*	Retorna o seno de um ângulo x expresso em radianos
tan	*Number tan(Number x)*	Retorna o tangente de um ângulo x expresso em radianos
acos	*Number acos(Number x)*	Retorna o valor, em radianos, do arco cujo cosseno é igual a x. O valor de x deve estar compreendido no intervalo $[-1, 1]$
asin	*Number asin(Number x)*	Retorna o valor, em radianos, do arco cujo seno é igual a x. O valor de x deve estar compreendido no intervalo $[-1, 1]$
atan	*Number atan(Number x)*	Retorna o valor, em radianos, do arco cuja tangente é igual a x. A função não determina em qual quadrante valor se encontra
atan2	*Number atan2(Number y, Number x)*	Retorna o valor, em radianos, do arco cuja tangente é igual a y/x (y representa um valor no eixo y, x representa um valor no eixo x)
cosh	*Number cosh(Number x)*	Retorna o cosseno hiperbólico de x
sinh	*Number sinh(Number x)*	Retorna o seno hiperbólico de x
tanh	*Number tanh(Number x)*	Retorna o tangente hiperbólica de x
exp	*Number exp(Number x)*	Retorna o valor de e^x
frexp	*Number frexp(Number x, Number y)*	Retorna um valor m no intervalo $[-1, 1]$ e altera o valor de *exp* tal que a relação $x = m * (2^y)$ seja verdadeira
ldexp	*Number ldexp(Number x, Number y)*	Retorna o resultado da expressão $x * 2^y$
log	*Number log(Number x)*	Retorna o logaritmo natural de x, $\ln x$
log10	*Number log10(Number x)*	Retorna o logaritmo de x na base 10, $\log_{10} x$
modf	*Number modf(Number x, Number i)*	Retorna a parte fracionária de x e atribui a parte inteira a i
pow	*Number pow(Number x, Number y)*	Retorna o valor da expressão x^y
sqrt	*Number sqrt(Number x)*	Retorna a raiz quadrada de x, $\sqrt{x}$
fabs	*Number fabs(Number x)*	Retorna o valor absoluto de x, $\|x\|$
ceil	*Number ceil(Number x)*	Arredondamento para cima, retorna o menor valor inteiro maior que x. Ex.: $ceil(2.3) = 3.0$, $ceil(-2.3) = -2.0$
floor	*Number floor(Number x)*	Arredondamento para baixo, retorna o maior valor inteiro menor que x. Ex.: $floor(2.3) = 2.0$, $floor(-2.3) = -3.0$
fmod	*Number fmod(Number x, Number y)*	Retorna o resto da divisão x/y, com x e y reais

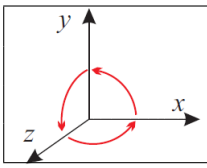
Interseção: É definida pela função *intersection* cuja assinatura é *Solid intersection(Solid s1, Solid s2)*. A função retorna um sólido correspondente a interseção dos sólidos *s1* e *s2*. A operação de interseção entre dois sólidos é exibida na Figura 7.1(d).

Diferença: É definida pela função *difference* cuja assinatura é *Solid difference(Solid s1, Solid s2)*. A função retorna um sólido composto por toda a região de *s1* que não possui interseção com *s2*. As Figuras 7.1(e) e 7.1(f) exibem a operação diferença.

Negação: É definida pela função *not* cuja assinatura é *Solid not(Solid s)*. A função retorna um sólido correspondente ao complemento do sólido *s*. O complemento equivale a toda região que não pertence ao sólido *s*. A operação de negação de um objeto pode ser visualizadas nas Figuras 7.1(g) e 7.1(h).

Rotações em torno de um eixo: São definidas pela funções *rotatex*, *rotatey* e *rotatez*. As assinatura são similares: *Solid rotatex(Solid s, Number ang)*, *Solid rotatey(Solid s, Number ang)*, *Solid rotatez(Solid s, Number ang)*. As funções retornam o sólido resultante da rotação do sólido *s* em torno do eixo *x*, função *rotatex*, do eixo *y*, função *rotatey*, ou do eixo *z*, função *rotatez*. O valor, em graus radianos, da rotação é expresso pelo segundo argumento(*ang*) em cada função. O sistema de coordenadas adotado pela linguagem usa a convenção da Mão Direita. Quando se olha de um eixo positivo para a origem, uma rotação de 90° no sentido anti-horário leva um eixo positivo em outro. A Tabela 7.5 mostra o sentido das rotações para valores positivos de *ang* em cada eixo.

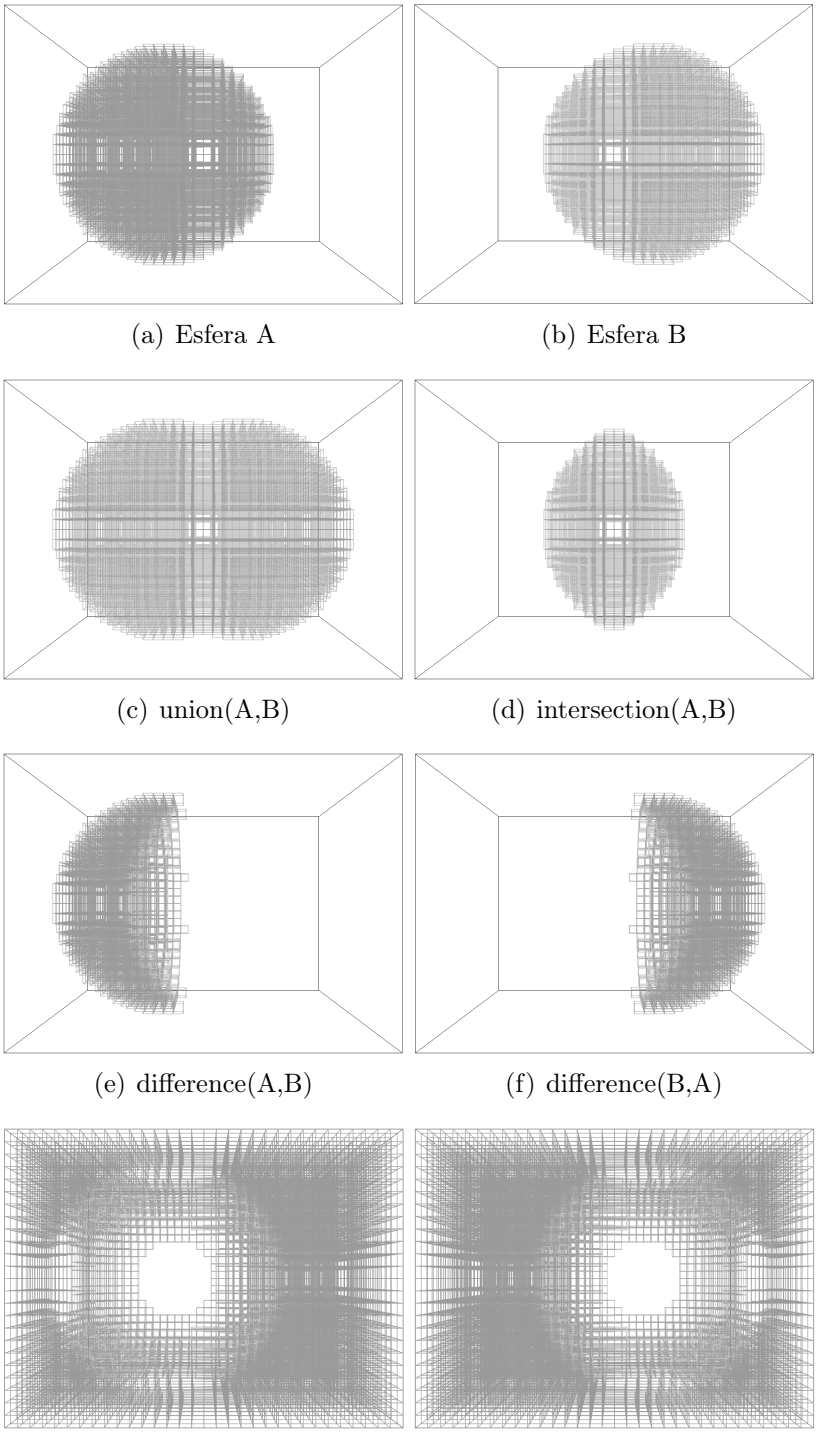
Tabela 7.5: Sentido das Rotações em torno dos eixos X, Y e Z.



EIXO DE ROTAÇÃO	SENTIDO DA ROTAÇÃO
X	Do eixo Y para o eixo Z
Y	Do eixo Z para o eixo X
Z	Do eixo X para o eixo Y

A Figura 7.2 exibe um objeto(7.2(a)) e descreve o resultado de sua rotação em torno dos eixos X(7.2(b)), Y(7.2(c)) e Z(7.2(d)).

Translação: É definida pela função *translate* cuja assinatura é *Solid translate(Solid s, Number tX, Number tY, Number tZ)*. A função retorna um sólido correspondente ao sólido *s* transladado pelo fator *tX* na direção do eixo *x*, pelo fator *tY* na direção do eixo *y* e pelo fator *tZ* na direção do eixo *z*. O resultado de translação de um



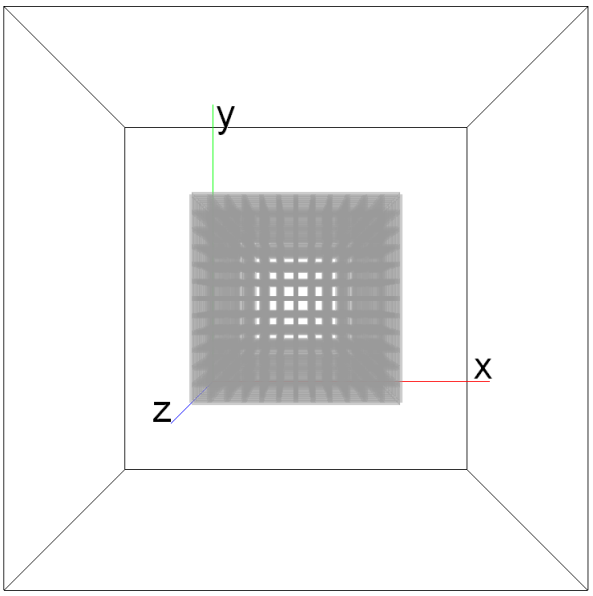
(a) Esfera A (b) Esfera B

(c) union(A,B) (d) intersection(A,B)

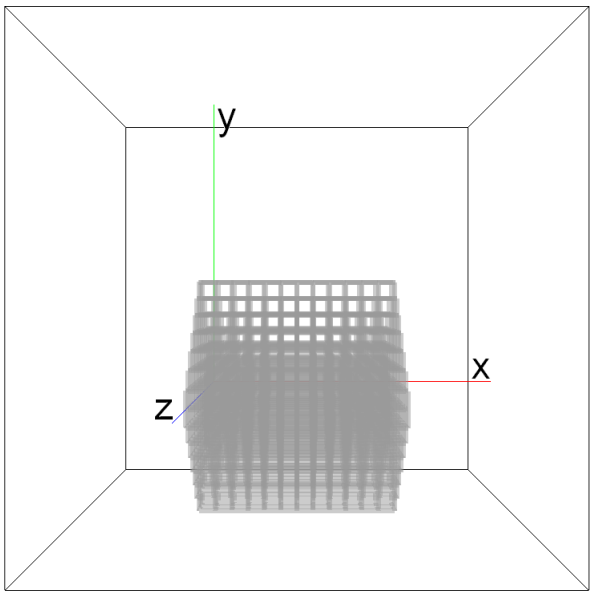
(e) difference(A,B) (f) difference(B,A)

(g) not(A) (h) not(B)

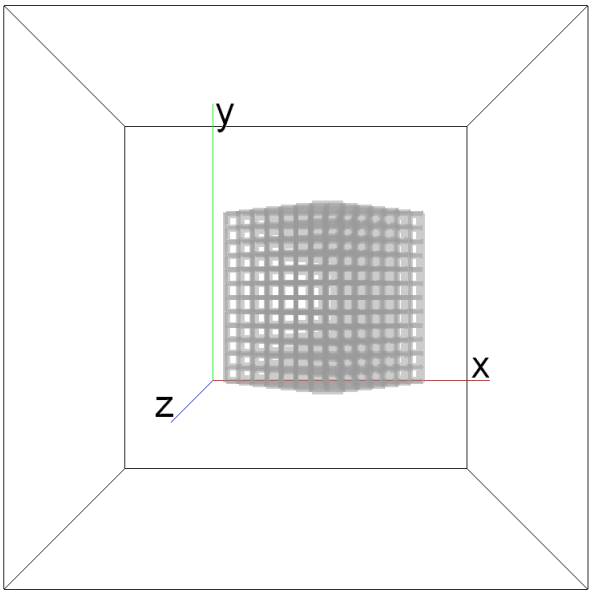
Figura 7.1: Operações entre sólidos.



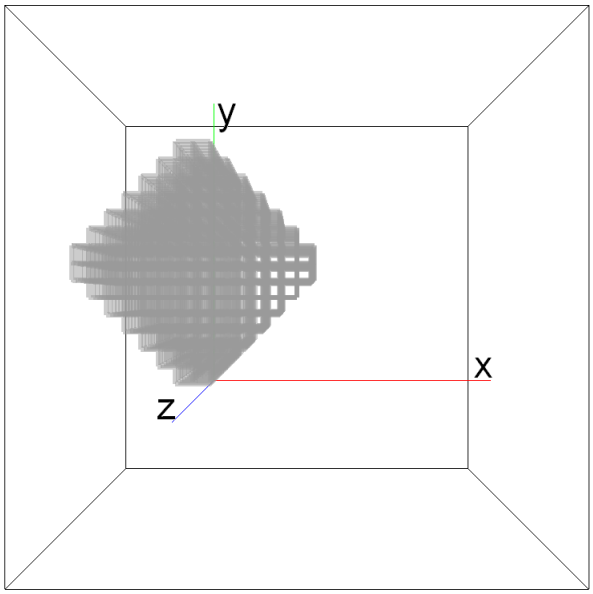
(a) Cubo



(b) Cubo rotacionado 45° em torno do eixo X, *rotatex(cubo,π/4)*



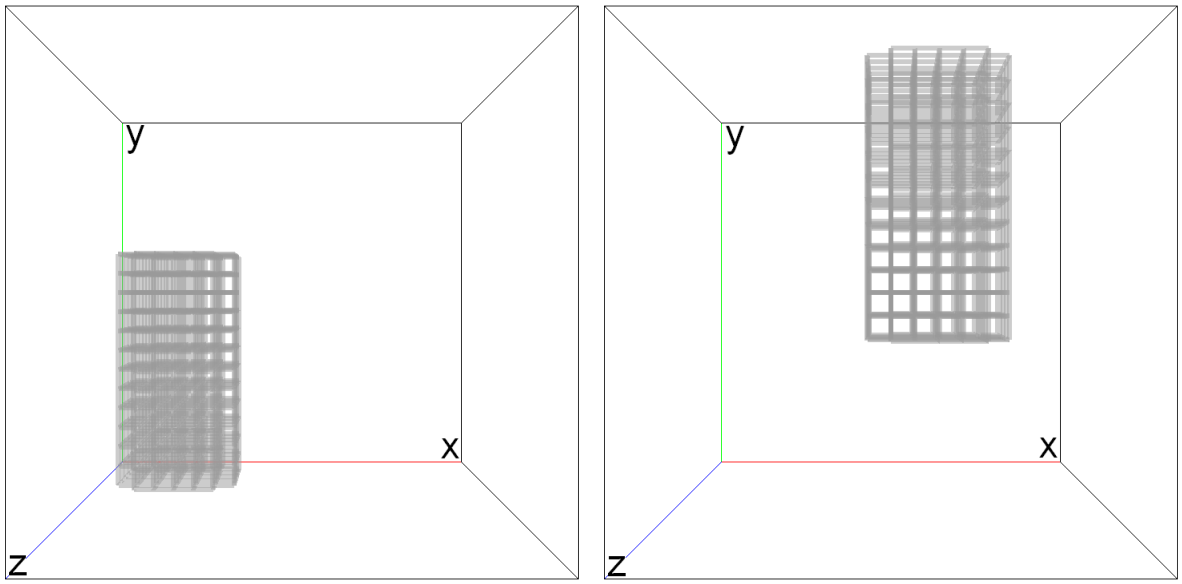
(c) Cubo rotacionado 45° em torno do eixo Y, *rotatey(cubo,π/4)*



(d) Cubo rotacionado 45° em torno do eixo Z, *rotatez(cubo,π/4)*

Figura 7.2: Rotações de um objeto em torno do eixo X, Y e Z.

cilindro de raio unitário, centrado em $(1, 0, 1)$, e com altura de quatro unidades é exibido na Figura 7.3.



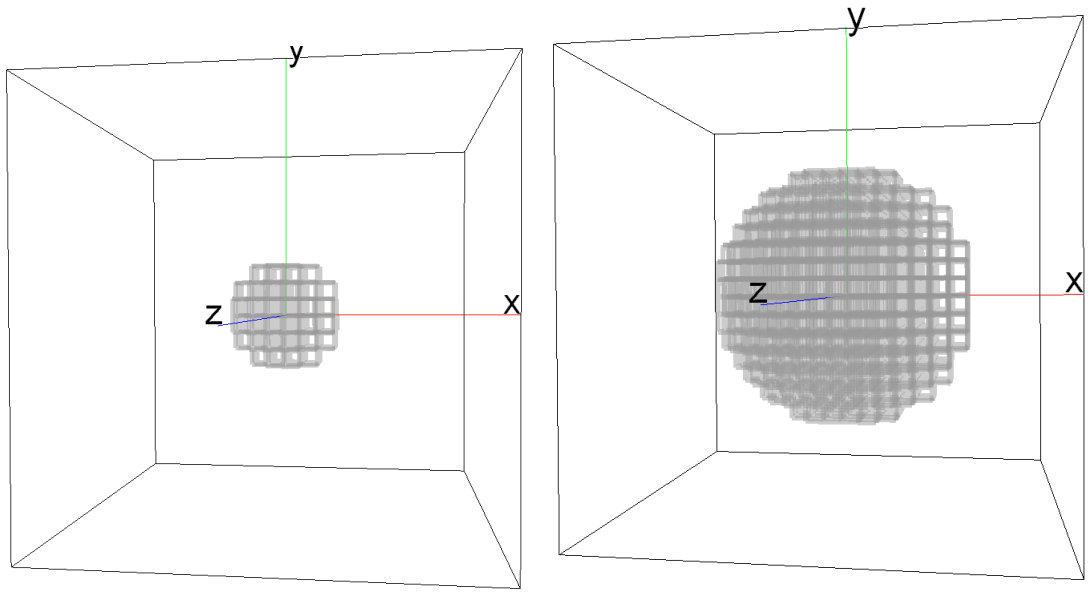
(a) Cilindro de raio unitário, centrado em $(1, 0, 1)$, com altura igual a 4 unidades.

(b) Cilindro Transladado em duas unidades em cada eixo, *translate(cilindro,2,2,2)*.

Figura 7.3: Translação de um sólido.

Escalonamento: É definida pela função *scale* cuja assinatura é *Solid scale(Solid s, Number sX, Number sY, Number sZ)*. A função retorna um sólido correspondente ao sólido s escalonado pelo fator sX na direção do eixo x, pelo fator sY na direção do eixo y e pelo fator sZ na direção do eixo z. A Figura 7.4(b) mostra o escalonamento de uma esfera, de raio unitário e centrada na origem, utilizando a função *scale* do seguinte modo: *scale(esfera,2,2,2)*

Cisalhamento: São definidos pelas funções *shearx*, *sheary* e *shearz*. As assinaturas são *Solid shearx(Solid s, Number sY, Number sZ)*, *Solid sheary(Solid s, Number sX, Number sZ)* e *Solid shearz(Solid s, Number sX, Number sY)*. A função retorna um sólido correspondente ao sólido s cisalhando o plano yz em relação a coordenada x (função *shearx*), o plano xz em relação a coordenada y (função *sheary*) ou o plano xy em relação a coordenada z (função *shearz*). A Figura 7.5 exibe um cubo e todos os cisalhamentos descritos na linguagem aplicados a ele.



(a) Esfera de raio unitário, centrada em $(0, 0, 0)$.

(b) Esfera escalonada, *scale(esfera,2,2,2)*.

Figura 7.4: Escalonamento de uma esfera.

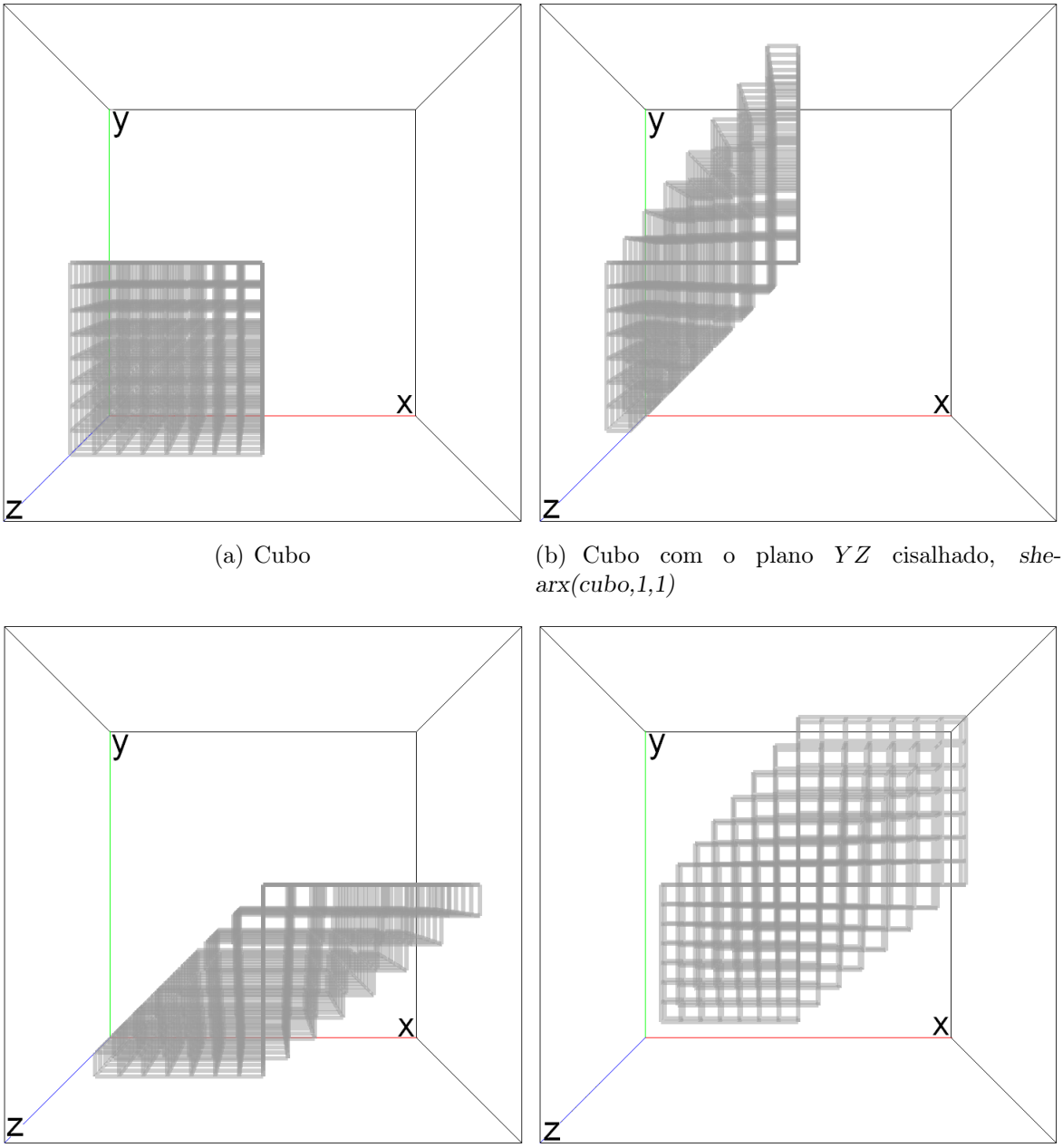
7.2.6 Estruturas de Controle

while é a única estrutura de controle disponibilizada pela linguagem. Essa estrutura indica que um determinado trecho de código deve ser executado enquanto uma condição for verdadeira. Sua sintaxe é descrita na Tabela 7.6.

Tabela 7.6: Sintaxe da estrutura de controle *while*.

while(condição) begin códigos end;		
ELEMENTO	SIGNIFICADO	VALORES POSSÍVEIS
while	Palavra reservada que indica o inicio da estrutura de controle *while*	
condição	Expressão lógica utilizada para avaliação se os comando descritos na instrução.	Qualquer expressão lógica: a < b, a = 10, c > 3 por exemplo
begin	Palavra reservada que indica o inicio dos comandos pertencentes a estrutura *while*	
códigos	Linhas de códigos que devem sem executadas dentro da estrutura *while*	insert(objeto);, a = 2; b = 3;, c = c + 1;
end	Palavra reservada que indica o término dos comandos pertencentes a estrutura *while*	

O Código 7.2, exibido abaixo, exemplifica utilização da estrutura de controle *while* e



(a) Cubo

(b) Cubo com o plano YZ cisalhado, *shearx(cubo,1,1)*

(c) Cubo com o plano XZ cisalhado, *shearY(cubo,1,1)*

(d) Cubo com o plano XY cisalhado, *shearz(cubo,1,1)*

Figura 7.5: Cisalhamentos de um Cubo em relação aos planos YZ, XZ e XY.

```
1 {Enquanto a Expressão (num < 5) for verdadeira os comandos
      contidos entre begin e end são executados}
2 while ( num < 5 )
3 begin
4         insert(circulo(num,num,num,10),override);
5         num = num + 1;
6 end;
```

Código 7.2: Trecho de código exemplificando a utilização do comando *while*.

pode ser compreendido da seguinte maneira: O código compreendido entre as palavras reservadas *begin* e *end*, linhas 4 e 5, é executando enquanto a condição *(num < 5)* for verdadeira.

7.2.7 Operações de Inserção

Inserção, na linguagem, representa a incorporação de um objeto a uma composição, quando o comando *insert* é utilizado dentro de uma composição, ou a inserção de um objeto no espaço definido pelo programador, quando o comando *insert* é utilizado no programa principal. A conceitos de incorporação a uma composição e inserção no espaço serão detalhadas na Seção 7.3. Por hora, será apresentada apenas a sintaxe do comando que realiza a inserção (Tabela 7.7).

Tabela 7.7: Sintaxe do comando *insert*.

COMANDO *insert* NA DEFINIÇÃO DE UMA COMPOSIÇÃO		
insert (objeto)		
COMANDO *insert* NO PROGRAMA PRINCIPAL		
insert(objeto , opçãoInserção [, (rot , eixo)]^n);		
ELEMENTO	SIGNIFICADO	VALORES POSSÍVEIS
insert	Palavra reservada que indica o inicio da instrução *insert*	
objeto	Objeto a ser incluído	Variável do tipo *Solid* ou chamada para a definição de um sólido ou composição
opçãoInserção	Define se o objeto a ser inserido deve sobrescrever, não sobrescrever ou gerar mensagem de erro se houver interseção do objeto a ser inserido com algum objeto já incluído	override (sobrescrever), nooverride (não sobrescrever), error (gerar erro)
rot	Valor em graus radianos que o retículo deve ser rotacionado	Variável do tipo *Number* ou um valor numérico
eixo	Eixo no qual o retículo será rotacionado	x, y ou z
O elemento *[, (rot , eixo)]^n* indica que o comando *insert* pode conter 0 ou n rotações do retículo em relação ao eixos x,y ou z.		

O trecho de Código 7.3 exibe as duas possibilidades de utilização do comando *insert*. Na primeira, linha 2, representa a utilização do comando *insert* dentro de uma composição e sua interpretação é trivial: Incorpore o objeto *circulo* à composição. O segundo exemplo, linha 5, representa a utilização do comando *insert* no programa principal e define a inserção

```
1 {Insere o objeto na definição de uma composição}
2 insert( circulo );
3
4 {Insere o objeto no espaço descrito na linguagem}
5 insert( circulo , override , (3.14, x),(3.14,y));
```

Código 7.3: Código exemplificando a duas utilizações do comando *insert*.

do objeto *circulo* no espaço descrito pelo usuário. O exemplo em questão indica que deve haver sobrescrição de qualquer outro objeto já incluído (opção *override*) e rotacionamento da célula unitária 3.14 radianos em relação a eixo x e depois em relação ao eixo y.

7.3 Criação do código

A criação de uma estrutura está atrelada ao seguinte conjunto de passos. O programador deve definir quais propriedades o sistema vai possuir, definir as características da estrutura, descrever sólidos e composições, definir uma região na qual os sólidos serão incluídos e, por fim, descrever as inclusões dos sólidos. As etapas serão detalhadas a seguir.

7.3.1 Criação de Propriedade

A criação da propriedade é a primeira etapa de construção da estrutura e possibilita ao programador definir propriedades para átomos ou moléculas da estrutura. A sintaxe da criação de uma propriedade e o detalhamento de seus itens podem ser verificados na Tabela 7.8.

O Código 7.4 exemplifica a definição de duas propriedades.

```
{Exemplo de definição de duas propriedades}
property bool metal = false {Define uma propriedade
    denominada metal, do tipo booleano com o valor padrão
    false}
property int numAtomico = 1 {Define uma propriedade
    denominada numAtomico, do tipo inteiro com o valor
    padrão 1}
```

Código 7.4: Código exemplificando a definição das propriedades de uma estrutura.

Todos os átomos ou moléculas da estrutura possuirão as propriedades criadas e, caso o programador não defina seus valores, as mesmas conterão seus valores padrões, definidos no momento de sua criação.

Tabela 7.8: Sintaxe para a criação de uma propriedade.

property tipo nome = valor		
ELEMENTO	SIGNIFICADO	VALORES POSSÍVEIS
property	Palavra reservada que indica a definição de uma propriedade	
tipo	Define o tipo de conteúdo que a propriedade armazenará.	Existem três valores possíveis: bool (indicando que a propriedade armazenará apenas os valores booleanos *true*, verdadeiro, ou *false*, falso), int (indicando que a propriedade armazena valores inteiros)e float (indicando que a propriedade armazena valores reais)
nome	Define um nome para a propriedade	Um identificador qualquer. Por exemplo: numAtomico, metal, massaAtomica
valor	Valor padrão associado a propriedade	Deve conter um valor booleano para uma propriedade do tipo bool, inteiro para uma propriedade do tipo int e real para uma propriedade do tipo float

7.3.2 Definição das Características da Estrutura

Após definir as propriedades, o programador deve definir as características da estrutura. Essas características foram apresentadas no Capítulo 2 e são definidas em dois passos pela linguagem: Definição da célula unitária e dos átomos ou moléculas da base.

A célula unitária é descrita na linguagem através do Sistema Cristalino, da distribuição dos elementos da base e da descrição dos vetores que a compõe. A sintaxe para a definição da célula unitária é exibida e descrita na Tabela 7.9. Por padronização, foi utilizada a língua inglesa para definir todos os elementos da linguagem, sendo assim, as palavras reservadas que denotam os termos descritos no Capítulo 2 são grafadas em inglês.

Quando o Sistema Cristalino adotado pelo programador possuir apenas um tipo de disposição dos elementos da base (sistemas *triclinic*, *rhombohedral* e *hexagonal*) a *Disposição* é omitida e a sintaxe de descrição da célula unitária é *Sistema (Vetor , Vetor , Vetor)*.

A descrição dos átomos ou moléculas que constituem a base com sua respectiva posição espacial, em relação a célula unitária, representam a segunda etapa da caracterização da estrutura. A Tabela 7.10 mostra e descreve sua sintaxe.

Tabela 7.9: Elementos constituintes da definição de uma célula unitária.

Sistema (Disposição , Vetor , Vetor , Vetor)		
ELEMENTO	SIGNIFICADO	VALORES POSSÍVEIS
Sistema	Sistema Cristalino Utilizado	*triclinic*, *rhombohedral*, *hexagonal*, *monoclinic*, *orthorhombic*, *tetragonal* e *cubic*
Disposição	Disposição do átomos ou moléculas na base. É omitida caso o Sistema Cristalino possua apenas uma disposição possível para os átomos ou moléculas	*primitive* para simples, *body* para corpo centrado, *face* para faces centradas e *basea*, *baseb* e *basec* para bases centradas
Vetor	Descrição de um vetor	$(1, 0, 0)$

Tabela 7.10: Definição de um átomo ou molécula da base.

base (Id , Vetor [, Propriedade]^n)		
ELEMENTO	SIGNIFICADO	VALORES POSSÍVEIS
base	Palavra reservada da linguagem que indica a definição de um elemento da base	
Id	Nome dado pelo usuário ao elemento da base	Um identificador qualquer. *cobre*, por exemplo
Vetor	Vetor que descreve o deslocamento, a partir da coordenada inicial de uma célula, necessário para encontrar a posição do elemento	$(0, 0, 0)$. Indica que a posição do elemento coincide com a posição inicial de cada célula unitária
Propriedade	Permite atribuir um novo valor para uma determinada propriedade (Seção 7.3.1). As propriedades omitidas recebem os valores padrões definidos na sua criação	nomePropriedade = novoValor. A propriedade já deve estar definida.

O Código 7.5 exibe, de maneira detalhada, a especificação de uma estrutura cúbica simples com dois elementos na base.

```
{Define uma propriedade denominada metal, do tipo booleano
    com o valor padrão false}
property bool metal = false
{Define uma propriedade denominada numAtomico, do tipo
    inteiro com o valor padrão 1}
property int numAtomico = 1

{Define que Sistema Cristalino utilizado será o Cúbico, que
     os vetores que descrevem sua célula são (1,0,0)
    ,(0,0,1),(0,1,0) e que a destribuição dos elementos da
    base será de corpo centrado (Sistema cúbico de corpo
    centrado contendo a célula cúbica descrita pelos
    vetores (1,0,0),(0,0,1),(0,1,0)}
cubic(body,(1,0,0),(0,0,1),(0,1,0))

{Descrição de dois elementos para a base:
H cuja posição coincide com as posições iniciais dos
    sistema e mantém os valores padrões das propriedades
Al cuja posição é definida pela posição inicial do sistema
    transladada pelo vetor (0,0,0.3) e possui as
    propriedades com valores modificados}
base( H, (0,0,0))
base( Al,(0,0,0.25), numAtomico = 13, metal = true)
```

Código 7.5: Código da definição de uma estrutura cúbica de corpo centrado.

7.3.3 Descrição de Sólidos

A descrição de sólidos consiste na definição dos moldes que serão utilizados no programa principal. Os moldes são representados por objetos (expressão lógica que descreve uma região) e por agrupamentos de objetos, denominados composições. No programa principal esses elementos indicaram quais regiões deverão ser preenchidas pela estrutura.

As descrições de sólidos e composições são descritas a seguir.

7.3.3.1 Definição de um objeto

Um objeto é definido na linguagem como sendo um conjunto de pontos que satisfazem uma determinada relação. Um objeto é descrito através do seu nome, de uma expressão lógica que determina seus pontos e de um conjunto de parâmetros que são utilizados na expressão lógica. A sintaxe para a definição de um objeto é exibida na Tabela 7.11.

Todos os parâmetros definidos na especificação de um objeto são classificados pela linguagem como sendo do tipo *Number* e são passíveis de serem utilizados, como variá-

Tabela 7.11: Sintaxe para a definição de um objeto.

solid nomeSolido ([parâmetro][, parâmetro]^n) exp;		
ELEMENTO	SIGNIFICADO	VALORES POSSÍVEIS
solid	Palavra reservada da linguagem que indica a definição de um sólido	
nomeSolido	Nome dado ao objeto	Um identificador qualquer. *esfera*, por exemplo
parâmetro	Nome dado a algum parâmetro que será utilizado na expressão que define o objeto	Um identificador qualquer. *a* por exemplo
exp	Expressão que define a função característica do sólido, conjunto de pontos que pertencentes ao sólido	$x < 2 \&\& y > 2$. *a* por exemplo
O elemento [parametro1][, parametro2]^n, descrito na sintaxe representa que a definição do sólido pode conter nenhum ou vários parâmetros e que os mesmos são separados pelo caractere “,”		

veis, na expressão que define o objeto. O Código 7.6 exibe, de maneira detalhada, a especificação de uma esfera na linguagem. A esfera possui as seguintes características:

- Centro, (x, y, z), definido pelos parâmetros a, b e c;
- Raio definido pelo parâmetro r;
- Expressão lógica que define seus pontos: $(x-a)^2 + (y-b)^2 + (z-c)^2 \leq r^2$, onde x, y e z representam os elementos da coordenada de um ponto qualquer.

```
{Definição de um objeto, chamado Esfera, correspondente a
    uma esfera centrada em (a,b,c) com raio r}
solid Esfera(a,b,c,r)
pow((x-a), 2) + pow((y-b), 2) + pow((z-c),2) <= pow(r,2);
```

Código 7.6: Código da definição de uma esfera de raio r centrada em (a, b, c).

7.3.3.2 Definição de um conjunto de objetos

Uma composição, ou conjuntos de objetos, é definido na linguagem com sendo a interseção dos pontos de todos os objetos que foram incorporados a ela. A definição de uma composição é composta por um identificador, representando o nome da composição, uma lista de parâmetros e um conjunto de instruções que, em conjunto com os parâmetros,

definirão quais objetos serão incorporados a composição. A Tabela 7.12 descreve a sintaxe para a criação de uma composição.

Tabela 7.12: Sintaxe para a definição de uma composição.

composite nomeComposição ([parâmetro][, parâmetro]^n) begin códigos end		
ELEMENTO	SIGNIFICADO	VALORES POSSÍVEIS
composite	Palavra reservada da linguagem que indica a definição de uma composição	
nomeComposição	Nome dado a composição	Um identificador qualquer. *uniaoEsferas*, por exemplo
parâmetro	Nome dado a algum parâmetro que será utilizado pelo código incluídos na composição	Um identificador qualquer. *a* por exemplo
begin	Palavra reservada da linguagem que indica o início dos códigos que definem a composição	
códigos	Sequência de instruções que devem ser executadas para a construção da composição. Todas as instruções devem ser separadas pelo caractere *;*	insert(esfera(0,0,0,2));
end	Palavra reservada da linguagem que indica o fim dos códigos que definem a composição	

No Código 7.7 é definida uma composição composta pela união de dois objetos *Esfera*. A interpretação do código é a seguinte:

Linhas 2 e 3: Definição do objeto *Esfera*;

Linha 6: Início da definição de uma composição denominada *UniaoEsferas* que não necessita receber parâmetros;

Linha 9: A variável *esf1* recebe como conteúdo uma esfera com centro em (0,0,0) e raio igual a 2, objeto *Esfera(0,0,0,2)*;

Linha 11: A variável *esf2* recebe como conteúdo uma esfera com centro em (2,0,0) e raio igual a 3, objeto *Esfera(2,0,0,3)*;

Linha 13: Incorpora o objeto *esf1* à composição;

Linha 15: Incorpora o objeto *esf2* à composição;

```
1  {Definição de uma Esfera centrada em (a,b,c) de raio r}
2  solid Esfera(a,b,c,r)
3  pow((x-a), 2) + pow((y-b), 2) + pow((z-c),2) <= pow(r,2);
4
5  {Definição de uma composição}
6  composite UniaoEsferas()
7  begin
8      {esf1 = esfera centrada em (0,0,0) com raio 2}
9      esf1 = Esfera(0,0,0,2);
10     {esf1 = esfera centrada em (2,0,0) com raio 3}
11     esf2 = Esfera(2,0,0,3);
12     {insere o objeto esf1 na composição}
13     insert(esf1);
14     {insere o objeto esf2 na composição}
15     insert(esf2);
16 end
```

Código 7.7: Código exemplificando a definição de uma composição.

A composição *UniaoEsferas* contém a união de duas esferas, uma centrada em (0,0,0) com raio igual a 2 e outra centrada em (2,0,0) com raio igual a 3.

7.3.4 Construção da Estrutura

Os elementos apresentados até esta seção possibilitam ao programador descrever as características (criação de propriedades, definição da célula unitária e definição dos átomos ou moléculas da base) da estrutura cristalina e os moldes (descrição de composições e sólidos) que poderão ser preenchidos pela estrutura. Para que uma estrutura seja, de fato, gerada é necessário que o programador defina uma região e quais os moldes serão usados.

A definição de uma região é feita através da palavra reservada *lattice*. O programador indicará as coordenadas mínimas e máximas em cada eixo. Após a definição do espaço, os códigos inseridos entre as palavras reservadas *begin* e *end* são responsáveis pela escolha de quais objetos serão utilizados na criação da estrutura.

A Tabela 7.13 descreve todo o processo. Esta região é denominada pela linguagem como programa principal.

Embora as inserções de objetos no programa principal sejam descritas pela instrução *insert*, é válido ressaltar que as ideias que norteiam a inclusão de objetos em uma composição e a inserção de um objeto no programa principal possuem algumas distinções. O uso dos verbos *incorporar* e *inserir* na Seção 7.2.7 e nesta seção são propositais, com o intuito de explicitar essas diferenças.

Enquanto incorporações de objetos em composição introduzem a ideia de aglutinação

Tabela 7.13: Sintaxe para a construção de uma estrutura.

lattice (Xmin , Ymin , Zmin , Xmax , Ymax , Zmax) begin códigos end		
ELEMENTO	SIGNIFICADO	VALORES POSSÍVEIS
lattice	Palavra reservada da linguagem que marca a definição de um espaço	
Xmin, Ymin, Zmin	Representam as menores coordenadas possíveis em cada eixo	Qualquer valor numérico
Xmax, Ymax, Zmax	Representam as maiores coordenadas possíveis em cada eixo	Qualquer valor numérico
begin	Palavra reservada que indica o inicio do código	
códigos	Códigos responsáveis criação da estrutura	Exemplo: a = 2; b = c;
end	Palavra reservada que indica o fim do código	

(união de objetos para a geração de um objeto diferente), inserções de objetos no programa principal representam a criação do objeto. É importante definir quais ações devem ser tomadas caso o objeto que se quer inserir sobreponha-se a outro objeto já inserido.

As palavras reservadas *override*, *nooverride* e *error* incluídas no comando *insert* definem qual ação tomar caso haja alguma sobreposição. Sobrescrever, não sobrescrever e gerar uma mensagem de erro são as ação definidas por elas.

Um outro detalhe importante é definir uma orientação para o retículo. Isto é feito através da rotação do mesmo em um dos eixos x, y e z. Isso se faz necessário para que a linguagem consiga gerar estruturas policristalinas. O programador deve definir cada grão e inseri-lo no espaço.

O Código 7.8 descreve a criação de uma estrutura policristalina composta por dois grãos. Um representado pelo objeto *esf* e outro pelo objeto *esf2*.

Nesta Seção foram apresentados os elementos constituintes de um código que descreve uma estrutura cristalina. A Figura 7.6 apresenta a estrutura policristalina gerada a partir da interpretação de um código fonte composto pelo seguintes itens:

- Propriedades presentes no Código 7.4;
- Sistema Cristalino definido pelo Código 7.5;
- Sólido descrito pelo Código 7.6;
- Programa principal apresentado no Código 7.8.

```
{Definição de um região compreendida pelo retângulo
    compreendido entre as coordenadas (-4,-4,-4) e (8,4,4)}
lattice(-4,-4,-4,8,4,4)
begin
      pi = 3.1416;
      esf = Esfera(0,0,0,4);
      esf2 = Esfera(4,0,0,4);
      {Insere o objeto esf mantendo a orientação do retículo
          }
      insert(esf,override);
      {Insere o objeto esf2 rotacionando o retículo}
      insert(esf2,override,(pi/3,z));
end
```

Código 7.8: Exemplo da construção de uma estrutura.

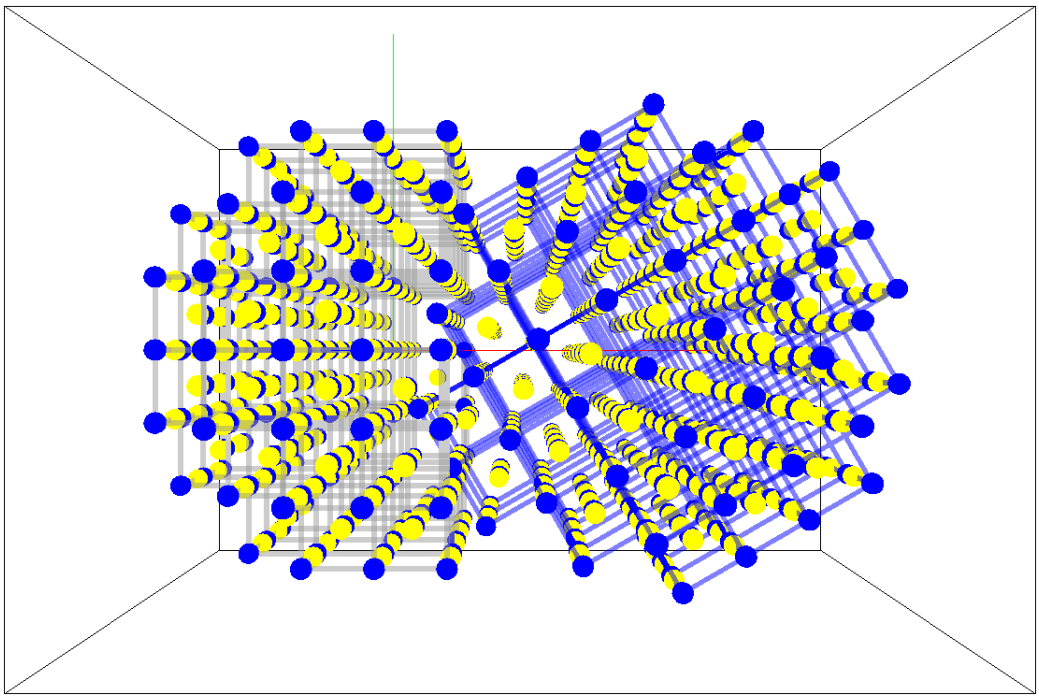


Figura 7.6: Exemplo de uma estrutura policristalina descrita por um código fonte. Na figura são apresentados as células unitárias e os átomos que compõem a estrutura.

8 COMPILADOR

8.1 Introdução

O compilador foi projetado para receber um programa, escrito na linguagem apresentada no Capítulo 7, e retornar ao usuário a estrutura definida no programa. A submissão do programa é feita via console através de um arquivo de texto comum, com extensão *txt*, passado como parâmetro.

A implementação foi feita utilizando a linguagem C++ e o compilador GNU GCC. O compilador é dividido em seis módulos:

Módulo Gerenciador de Caracteres (MGC): Efetua a leitura, o armazenamento e a distribuição dos caracteres provenientes do código fonte;

Módulo Léxico (MLX): Executa análise léxica fornecendo os *tokens* para o módulo Sintático;

Módulo Sintático (MSN): Executa a análise sintática construindo e gerenciando a Árvore de Sintaxe Abstrata (ASA);

Módulo Semântico (MSM): Percorre a ASA efetuando a análise semântica;

Módulo Interpretador (MIN): Interpreta a ASA gerando a estrutura cristalina;

Módulo Gerenciador de Erros (MGE): Gerencia e exibe os erros reportados em cada etapa do processo de compilação.

A estrutura geral de funcionamento do compilador e as comunicações realizadas entre os módulos são descritas na Figura 8.1.

A fim de expor os resultados obtidos foram incorporados dois módulos ao compilador. O primeiro responsável pela visualização da estrutura criada, Módulo Visualizador, e o outro pela incorporação da estrutura a um simulador, Módulo Transferência. O objetivo deste capítulo é apresentar o compilador implementado descrevendo seu funcionamento através de seus módulos.

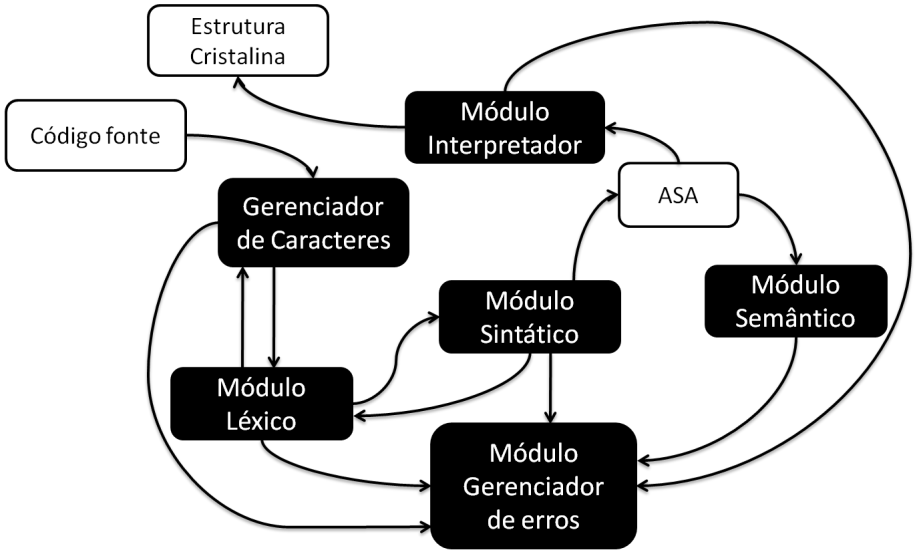


Figura 8.1: Visão Geral do Compilador.

8.2 Módulo Gerenciador de Caracteres (MGC)

O módulo de gerenciamento de caracteres agrupa um conjunto de funções e estruturas responsáveis pela manipulação dos caracteres do programa fonte. Essas funções são responsáveis pela leitura, contagem de linhas e colunas e armazenamento dos dados gerados a partir dos caracteres lidos.

Os caracteres do programa fonte são lidos em blocos (um número determinado de caracteres) e armazenados em um *buffer*. O MGC utiliza um *buffer* duplo com sentinela que, segundo Aho [16], é empregado com o intuito de tornar nula a necessidade de ler todo o programa de uma só vez e diminuir o *overhead* na leitura caso a mesma seja feita de forma fracionada.

O *buffer* duplo é composto por um arranjo de duas posições onde cada posição corresponde a um *buffer*. O preenchimento é feito da seguinte maneira: É lido do programa e armazenado no primeiro *buffer* a quantidade de caracteres que ele comporta. Em seguida esses caracteres são usados pelo compilador até que todos sejam consumidos. Quando isso acontece é lida mais uma parte do programa que é adicionada no segundo *buffer*. A mesma coisa acontece quando o segundo *buffer* é utilizado por completo, o primeiro *buffer* recebe os próximos elementos do arquivo. A sentinela é um vetor que indica qual posição do *buffer* está sendo utilizada. A Figura 8.2 exemplifica essa estrutura.

Além da leitura do programa principal o módulo de gerenciamento de caracteres é responsável pelo gerenciamento de outros itens: Os mesmos estão descritos abaixo, contudo

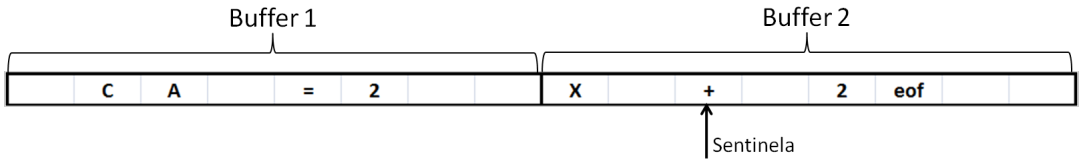


Figura 8.2: Buffer duplo com sentinela. O caractere *eof* representa final de arquivo.

serão explicados nos módulos que os utilizam.

8.3 Módulo Léxico (MLX)

O módulo léxico faz a leitura dos caracteres do programa fonte (com auxílio do *buffer* duplo implementado no MGC), agrupando-os em *tokens* e encaminhando-os ao módulo responsável pela análise sintática (MSN). Cada *token* foi descrito por uma expressão regular.

A Tabela 8.1 apresenta as expressões regulares utilizadas para descrever os *tokens* da linguagem.

Tabela 8.1: Expressões Regulares que descrevem os *tokens* da linguagem.

TOKEN	EXPRESSÃO REGULAR
ASSIGN	=
OPARENT	(
CPARENT	)
COLON	,
SEMICOLON	;
DOT	.
OP_EQUAL	==
OP_LESS	<
OP_LESSEQUAL	<=
OP_GREAT	>
OP_GREATEQUAL	>=
OP_DIFFERENT	<>
OP_PLUS	+
OP_MINUS	−
OP_TIMES	*
OP_DIVIDE	/
OP_OR	||
OP_AND	&&
ID	$LETRA(LETRA\|DIGITO)^*$
NUMBER	$(DIGITO(DIGITO)^*)\|$ $(DIGITO(DIGITO)^*.(DIGITO)^*)$
EOF	*eof* (caractere que representa final de arquivo)

LETRA e DIGITO não representam *tokens* e são reconhecidos pelas seguintes expressões regulares:

$$LETRA \rightarrow a|b|c \cdots x|y|z|A|B|C \cdots X|Y|Z$$

$$DIGITO \rightarrow 0|1|2|3|4|5|6|7|8|9$$

As expressões regulares foram incorporadas a um Autômato Finito Determinístico (AFD) e o mesmo foi implementado para fazer o reconhecimento dos *tokens*. A Figura 8.3 exibe o AFD implementado.

Os caracteres *DELIM*, *LETRA* e *DIGITO* apresentados nas transições do AFD representam, respectivamente, caracteres desprezados na análise do código (espaço em branco, tabulações e quebras de linhas), qualquer letra e qualquer dígito. O caractere *outro* descrito em algumas transições representa a leitura de qualquer caractere não previsto pela demais regras de transição do estado vigente. Este caractere não é consumido. O AFD é reiniciado assumindo que o caractere a ser lido não se alterou.

O autômato implementado não relaciona corretamente as palavras reservadas da linguagem com seus respectivos *tokens*. A princípio, todas as palavras reservadas são reconhecidas como identificadores, *token* ID. Isso simplifica muito a implementação do autômato, uma vez que as expressões regulares que denotam os *tokens* oriundos das palavras reservadas não são utilizadas. O equívoco em questão é solucionado na inicialização da Tabela de Símbolos.

8.3.1 Tabela de Símbolos

A fim de armazenar informações sobre os *tokens* encontrados na varredura do programa fonte foi criada uma estrutura denominada Tabela de Símbolos. Sua finalidade é armazenar características dos *tokens* encontrados que serão úteis nas análises subsequentes. Inicialmente, a única informação necessária é seu lexema. Apenas a primeira ocorrência do lexema é armazenada.

Para possibilitar uma recuperação rápida dos dados contidos na tabela optou-se por implementá-la utilizando o conceito de Tabelas *Hash*. Tabelas *Hash* são estrutura de dados muito utilizadas e têm por objetivo organizar os dados de modo que sua recuperação seja feita de modo direto, sem a necessidade de buscas.

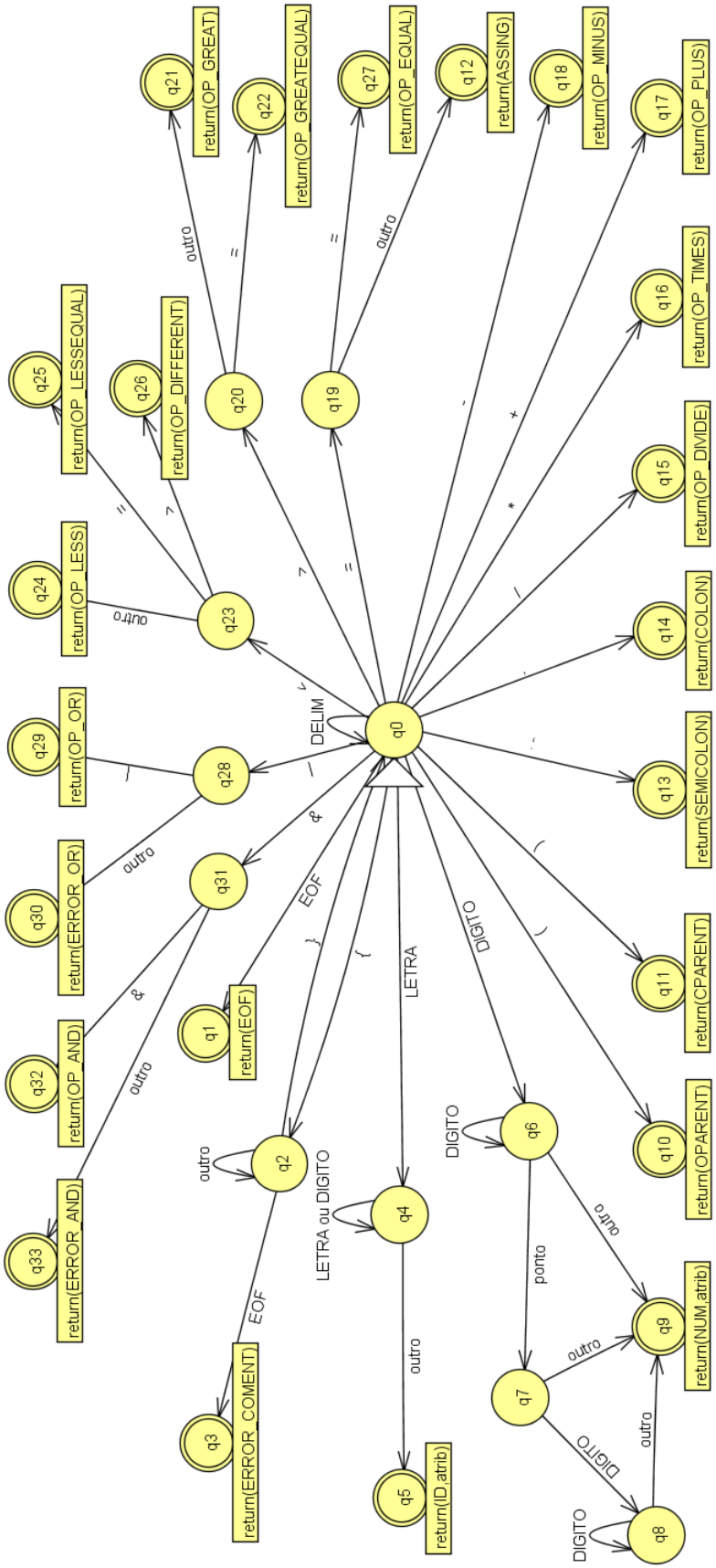


Figura 8.3: AFD implementado.

Uma Tabela *Hash* é implementada por um vetor com um tamanho preestabelecido k e uma função de espalhamento (dispersão) f que calcula o endereço, índice do vetor (0 a $k-1$), onde o dado será armazenado.

Existem várias questões relacionadas a escolha do tamanho k, a função de espalhamento e ao tratamento de colisões (situação que ocorre quando função de espalhamento indica a inserção de dois elementos em uma mesma posição do vetor). Não é objetivo primordial deste trabalho uma abordagem profunda desses conceitos. Eles podem ser encontrados nos trabalhos de Aho [16] e Ascencio [25].

A Tabela *Hash* implementada neste trabalho adotou $k = 211$, utilização de listas encadeadas para o tratamento de colisões e a aplicação da função espalhamento *P. J. Weinberger* (PJW) no lexema dos *tokens* para definir a posição de armazenamento. A implementação do algoritmo PJW é exibida em 8.1.

```
int hashPJW(char* ent){
    char* p;
    unsigned h = 0,g;
    for(p = ent; *p != '\0'; p = p + 1){
        h = (h << 4) + (*p);
        if (g = h&0xf000000){
            h = h ^ (g >> 24);
            h = h ^ g;
        }
    }
    return h % TAM_TABELA; //TAM_TABELA = 211
}
```

Código 8.1: Algoritmo P. J. Weinberger (PJW) implementado em C. Adaptado de [16].

A Figura 8.4 apresenta a esquematização dos elementos pertencentes a Tabela de Símbolos:

Tabela de Símbolos: Tabela *Hash* contendo ponteiros para uma lista encadeada definida pela estrutura *entradaTab*.

entradaTab: Lista encadeada responsável por armazenar dados referentes aos *tokens* encontrados na análise léxica. Guarda a posição inicial do lexema no *Buffer* de Lexemas, um valor numérico (reconhecido internamente pelo compilador) representando qual *token* foi armazenado e um ponteiro para um outro elemento inserido na mesma posição.

***Buffer* de Lexemas:** Vetor dinâmico destinado ao armazenamento sequencial dos lexemas necessários ao compilador (lexemas dos identificadores, das palavras reservadas e dos valores numéricos). O caractere '\0' indica o fim de um lexema.

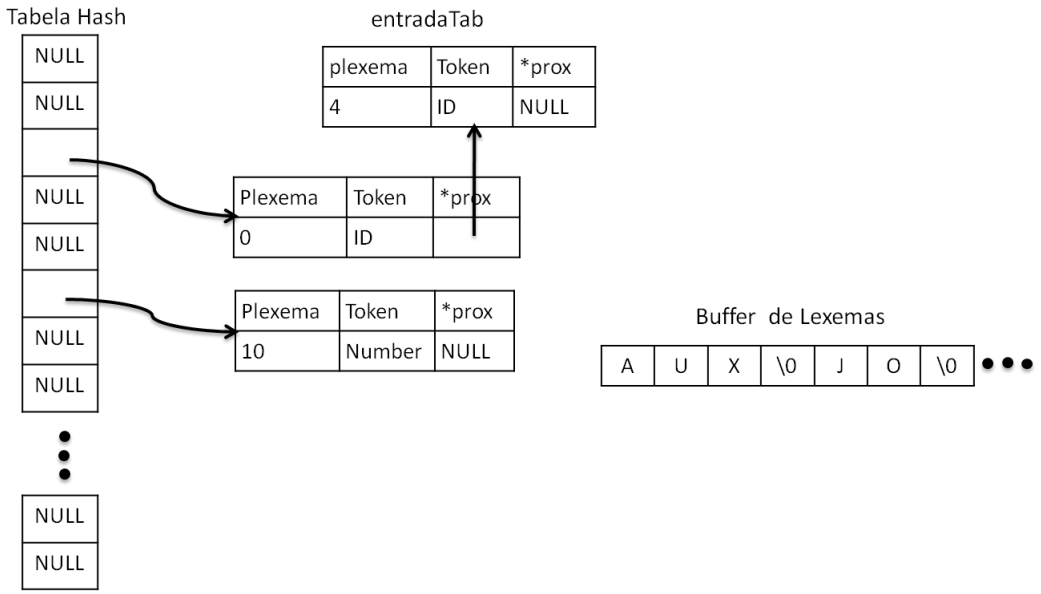


Figura 8.4: Esquema da implementação da Tabela de Símbolos.

8.3.2 Funcionamento do Módulo

Inicialmente é inserida uma estrada na Tabela de Símbolos para cada palavra reservada e incluído o lexema correspondente no *Buffer* de Lexemas, inicialização da Tabela de Símbolos. O MLX solicita caracteres ao MGC promovendo as transições descritas no AFD até que um *token* ou um erro seja reconhecido. Caso o *token* reconhecido seja *ID* ou *Number* o mesmo é incorporado a Tabela de Símbolos, caso não tenha sido incluído anteriormente.

Neste momento a estratégia de simplificação adotada no AFD (palavras reservadas reconhecidas como identificadores) é corrigida. Como apenas a primeira ocorrência de um lexema é armazenada, um lexema pertencente a uma palavra reservada, identificado como *ID* pelo AFD, não é inserido na Tabela de Símbolos e a identificação correta do *token* que o representa é feita através da leitura do *token* inserido na inicialização da Tabela de Símbolos.

Todos os erros reconhecidos são enviados ao MGE e o mesmo se encarrega da exibição ao programador. A Figura 8.5 descreve o funcionamento do MLX.

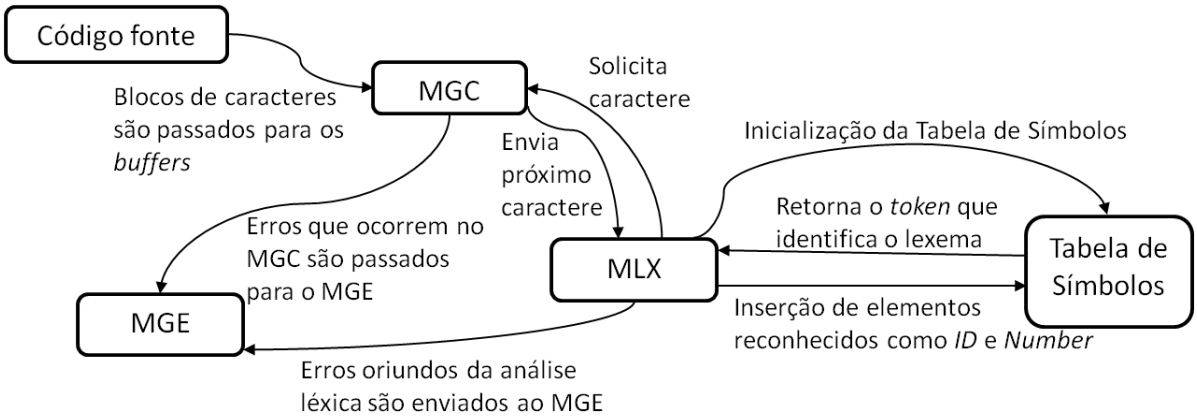


Figura 8.5: Fluxograma de funcionamento do MLX.

8.4 Módulo Sintático (MSN)

O MSN obtém os *tokens* provenientes do MLX, realiza a análise sintática (verificando se a organização dos *tokens* condiz com as regras descritas na gramática criada) e cria uma Árvore de Sintaxe Abstrata (ASA). Aho [16] cita dois algoritmos gerais para construção de analisadores sintáticos, *Cocke-Younger-Kasami* e *Earley*, relatando sua capacidade de tratar qualquer linguagem e sua ineficiência quando utilizados na produção de compiladores. Como alternativa apresenta os dois métodos menos abrangentes, com capacidade de analisar um grupo reduzidos de linguagens, contudo mais eficientes: Analisadores sintáticos *top-down* e *bottom-up*.

Analisadores Sintáticos (também conhecidos como *parsers*) *top-down* recebem essa classificação por gerarem a ASA da raiz (topo) para as folhas (fundo). Eles analisam um conjunto de GLC's (Gramáticas Livres do Contexto) denominado *LL* e são mais utilizados em *parsers* implementados manualmente [16].

Parsers bottom-up analisam um grupo de LLC's (Linguagens Livre do Contexto) denominado *LR* construindo a ASA a partir das folhas. Em geral são construídos de forma automática através da utilização de *softwares*.

Embora gramaticas LR's sejam mais expressivas se comparadas as LL's optou-se por implementar um *parser top-down*. Isso não trouxe nenhum prejuízo ao poder de expressão da gramática criada, uma vez que foi possível convertê-la em uma gramática LL. Na conversão foram retirados os seguintes itens:

Ambiguidade: Regras de produção que possibilitam a criação de mais de uma árvore gramatical para alguma palavra;

Recursão à Esquerda: Regras de produção onde o não-terminal A pode aparecer à esquerda a partir da derivação da regra que o define, $A \Rightarrow^{+} A\alpha$.

Os algoritmos utilizados para retirar ambiguidades e recursões à esquerda são descritos com mais detalhes por Aho [16].

8.4.1 Parser Top-Down

Em um *parser top-down* os *tokens* são analisados sequencialmente, na ordem que são encontrados no código fonte, e são agrupados através de uma derivação mais a esquerda, assim a árvore é criada em pré-ordem. Correlacionando com uma árvore binária, os filhos da esquerda são gerados antes que os filhos da direita. Daí vem a denominação LL (*left-left*) para as gramáticas reconhecidas por esse tipo de *parser*.

Um *parser top-down* pode implementar as derivações contidas na gramática de duas formas: Utilizando todas as regras possíveis a partir do símbolo terminal (*token*) lido, fazendo retrocessos caso a derivação seja equivocada (*backtracking*); ou prevendo qual derivação deve ser feita a partir da visualização antecipada dos próximos *tokens* a serem lidos (*look ahead*).

Com o intuito de evitar a computação desnecessária de derivações equivocadas optou-se por implementar a segunda forma, um *parser top-down* preditivo com um *token* de *look ahead*. Foi necessário fazer uma nova conversão na gramática através do algoritmo de fatoração à esquerda (descrito por Aho [16]) a fim de ser obter uma gramática LL(1). O número '1', entre parênteses, representa quantos *tokens* o *parser* deve visualizar adiante para saber qual derivação realizar.

Segundo Aho [16], uma gramática G é dita LL(1) se, e somente se, qualquer produção $A \rightarrow \alpha|\beta$ com $\alpha \neq \beta$ pertencente a G satisfaçam as seguinte condições:

1. $First(\alpha) \cap First(\beta) = \oslash$ (α e β não derivem ao mesmo tempo, cadeias começando por um mesmo terminal);

2. $\alpha \Rightarrow^{*} \varepsilon$ implica que $\beta \Rightarrow^{*} \varepsilon$ é falso (Somente α ou β podem derivar a cadeia vazia);

3. $\alpha \Rightarrow^{*} \varepsilon$ implica que $First(\beta) \cap Follow(\alpha) = \oslash$ (Se α deriva a cadeia vazia então após derivações de α não podem aparecer um terminal pertencente ao conjunto $First(\beta)$).

$First(\alpha)$ representa o conjunto de terminais que começam qualquer sequencia derivável de α.

$Follow(\alpha)$ representa o conjunto de terminais que podem aparecer à direita de um não-terminal α em uma sentença válida.

A gramática criada foi analisada segundo a definição acima a fim de corroborar se, de fato, ela é uma gramática LL(1).

8.4.2 ASA

A Árvore de Sintaxe Abstrata, conforme apresentado no Capítulo 4, é uma estrutura destinada ao armazenamento de todas as instruções contidas no programa fonte. A ASA foi implementada utilizando o conceitos de Orientação a Objetos. Cada classe define um tipo de nó pertencente ASA e representa uma construção descrita na gramática da linguagem. Todos os elementos descritos no Capítulo 7 foram agrupados segundo sua função. Esse agrupamento gerou as seguintes classes:

exp: Classe abstrata que representa todos os elementos da linguagem que produzem como resultado números e objetos: identificadores; operações lógicas, relacionais e aritméticas; operações de transformações de objetos; funções matemáticas.

statement: Classe abstrata representando todas as instruções possíveis dentro dos blocos de comando: inserção, atribuição e *while*;

program: Representa o programa;

Properties: Propriedades incluídas no programa;

DefStructure: Características definidas para a estrutura;

Base_List: Elementos pertencentes a base da estrutura;

Solid_List: Descrição dos Sólidos;

Lattice: Região definida para a estrutura;

Statement_List: Lista de comandos incluídos no programa. Possui um conjunto de objetos da classe *statement*;

Foi implementado um outro conjunto de classes que servem como apoio às classes do agrupamento acima. A classe *ID* por exemplo foi criada a fim de armazenar objetos da classe *exp* que representam identificadores, a classe *insert* representa um objeto da classe *statement* que define uma inserção.

A ASA pode ser vista como uma árvore. Assim cada nó pode ser definido por um objeto de uma classe ou ou conjunto de objetos. A Tabela 8.2 apresenta todos os nós da ASA e os elementos que os compõem. Os nós são exibidos em caixa alta (letras maiúsculas), atributos importantes em caixa baixa (letras minusculas) e o elemento *NULL* indica que o nó pode ser definido sem nenhum objeto.

Tabela 8.2: Estrutura da ASA .

NÓ	**FORMAS POSSÍVEIS**
PROGRAM	(PROPERTIES, DEFSTRUCTURE, BASE_LIST, SOLID_LIST, LATTICE, STATEMENT_LIST)
PROPERTIES	(PROPERTY, PROPERTIES) NULL
PROPERTY	(TYPE, ID, NUM) (TYPE, ID, BOOL)
NUM	(numero)
BOOL	(booleano)
DEFSTRUCTURE	(geometry, centering, VECTOR, VECTOR, VECTOR)
BASE_LIST	(DECL_BASE, BASE_LIST) NULL
DECL_BASE	(ID, VECTOR, PROPERTY_LIST)
ID	(identificador)
SOLID_LIST	(DECL_SOLID, SOLID_LIST) NULL
DECL_SOLID	(ID, ARGUMENT_LIST, EXP) (ID, ARGUMENT_LIST, STATEMENT_LIST)
ARGUMENT_LIST	(ID, ARGUMENT_LIST) NULL

Continua na próxima página

Tabela 8.2 – *Continua na página anterior*

NÓ	**FORMAS POSSÍVEIS**
VECTOR	(EXP, EXP, EXP)
PROPERTY_LIST	(PROPERTY_ASSIGN, PROPERTY_LIST) NULL
PROPERTY_ASSIGN	(ID, NUM) (ID, BOOL)
LATTICE	(EXP, EXP, EXP, EXP, EXP, EXP, EXP, EXP, EXP)
STATEMENT_LIST	(STATEMENT, STATEMENT_LIST) NULL
STATEMENT	(ASSIGN) (INSERT) (WHILE)
ASSIGN	(ID, EXP)
INSERT	(EXP, option, ROTATE_LIST)
ROTATE_LIST	(EXP, EIXO, ROTATE_LIST) NULL
WHILE	(EXP, STATEMENT_LIST)
EXP	(ID) (NUM) (x) (y) (z) (OR_OP) (AND_OP) (RELATIONAL_OP) (ADDITION_OP) (MULTIPLICATION_OP) (EQUAL_OP) (UNARY_OP) (FUNCTION_MATH)

Continua na próxima página

Tabela 8.2 – *Continua na página anterior*

NÓ	FORMAS POSSÍVEIS
	(CSG_OP) (NOT_OP) (TRANSFORMATION_OP) (OBJECT_CALL)
OR_OP	(EXP, EXP)
AND_OP	(EXP, EXP)
RELATIONAL_OP	(op_rel, EXP, EXP)
ADDITION_OP	(op_add, EXP, EXP)
MULTIPLICATION_-OP	(op_mul, EXP, EXP)
EQUAL_OP	(op_equal, EXP, EXP)
UNARY_OP	(EXP)
FUNCTION_MATH	(func_math, EXP) (func_math, EXP, EXP)
CSG_OP	(csg, EXP, EXP)
NOT_OP	(EXP)
TRANSFORMATION_OP	(transformation, EXP) (transformation, EXP, EXP, EXP, EXP)
OBJECT_CALL	(ID, EXPR_LIST)
EXPR_LIST	(EXP, EXPR_LIST) NULL

8.4.3 Funcionamento do Módulo

O *parser* implementa as regras de construção descritas na gramática, ou seja, solicita *tokens* ao MLX tentando enquadrá-los em algumas das regras gramaticais. Os *tokens* são consumidos de forma progressiva, assim as análises léxica e sintática são processadas de maneira simultânea, não havendo necessidade de armazenar ou guardar a ordem que os *tokens* são formados a partir do código fonte.

A medida que os *tokens* lidos são enquadrados nas regras gramaticais, elementos da ASA são construídos. Para tanto, o *parser* utiliza as classes definidas na ASA. O *parser* consume os *tokens* seguindo o conceito de prever quais regras de produção deve utilizar. Isso ocorre através da análise de dois *tokens*: O último e o próximo *token* a ser lido.

Quando o processo falha, não existe nenhuma associação relacionando os dois *tokens*, o que configura um erro sintático. Elementos da ASA que apresentam erros, embora tenham seus filhos analisados pelo *parser*, são descartados, isto é, não são incorporados as ASA. Assim como os erros léxicos, os erros sintáticos são reportados ao MGE. Para que o *parser* continue verificando o restante do código, é necessária a implementação de um mecanismo de recuperação de falhas. O mecanismo implementado no *parser* utiliza a estratégia *panic mode*: Desprezar a regra atual consumindo *tokens* até que uma nova regra possa ser utilizada. Todo o mecanismo funciona a partir de três funções:

match: Consome um *token* recebido como parâmetro e solicita o próximo *token* ao MLX;

skipto: Recebe como parâmetro uma lista de *tokens* e executa a função *match* até que o *token* recebido seja igual a um dos *tokens* da lista. A lista de *tokens* enviadas para a função *skipto* recebe o nome de *tokens* de sincronização e possibilita que o *parser* despreze uma estrutura descrita de modo equivocado pelo código e continue a análise.

matchorskip: Recebe como parâmetro um *token* e uma lista de *tokens*. O *token* recebido representa o *token* esperado pela regra analisada no momento e a lista representa seus *tokens* de sincronização. Caso o próximo *token* a ser lido seja igual ao esperado pela regra, a função *match* é utilizada. Caso contrário é reportado um erro ao MGE e a função *skipto* é utilizada para a recuperação do erro.

A Figura 8.6 ilustra o funcionamento do MSN.

8.5 Módulo Semântico (MSM)

O MSM é responsável pela verificação dos tipos de dados envolvidos em todas as expressões descritas no código fonte. Essa verificação é feita segundo um conjunto de regras, denominadas regras semânticas, que indicam quais tipos de dados podem ser utilizados em cada estrutura da linguagem. A fim de armazenar o tipo de dado atrelado a cada

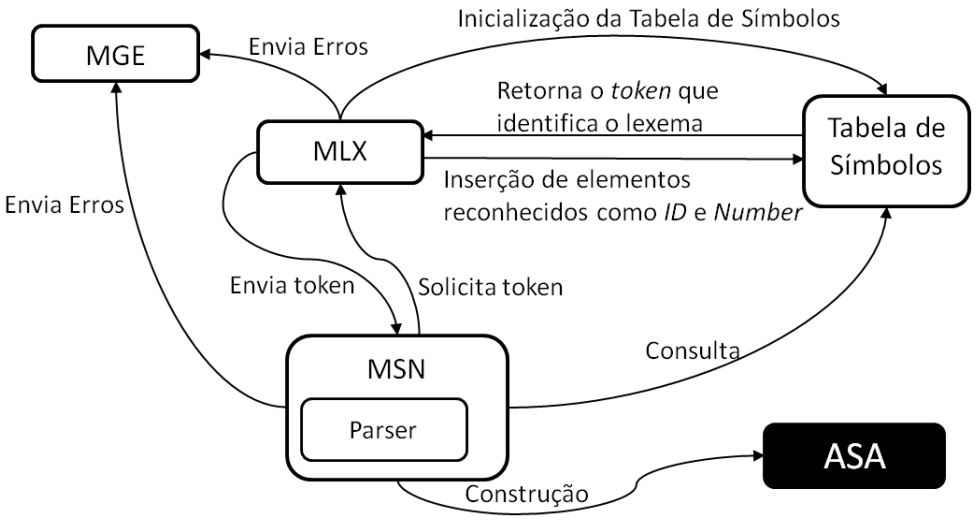


Figura 8.6: Fluxograma de funcionamento do MSN.

identificador descrito no código fonte, a Tabela de Símbolos foi expandida. Uma estrutura denominada *extEntradaTab* foi incorporada a tabela armazenando as seguintes características:

tipo: Guarda qual é o tipo do dado representado pelo identificador. *Number* representando valor numérico, *Solid* representando um sólido, *DefSolid* indicando a definição de um sólido, *DefComposite* indicando a definição de uma composição e *Base* indicando a definição de um elemento da base;

pDef: Ponteiro para o elemento da Tabela de Símbolos ao qual o elemento está subordinado.

parametro: Lista que referencia todos os elementos da Tabela de Símbolos que serão utilizados como parâmetros para a entrada em questão;

escopo: Guarda um número representando quantos escopos foram iniciados até a definição do elemento;

pProx: Ponteiro para um outro tipo de dado referenciado pelo identificador. Um identificador pode ser usado para identificar estruturas de dados diferentes em diferentes regiões do código fonte.

A Figura 8.7 exibe a implementação completa da Tabela de Símbolos.

Por estarem relacionadas ao contexto em que cada expressão aparece, as regras semânticas não são descritas pela gramática que descreve a linguagem. Elas são definidas

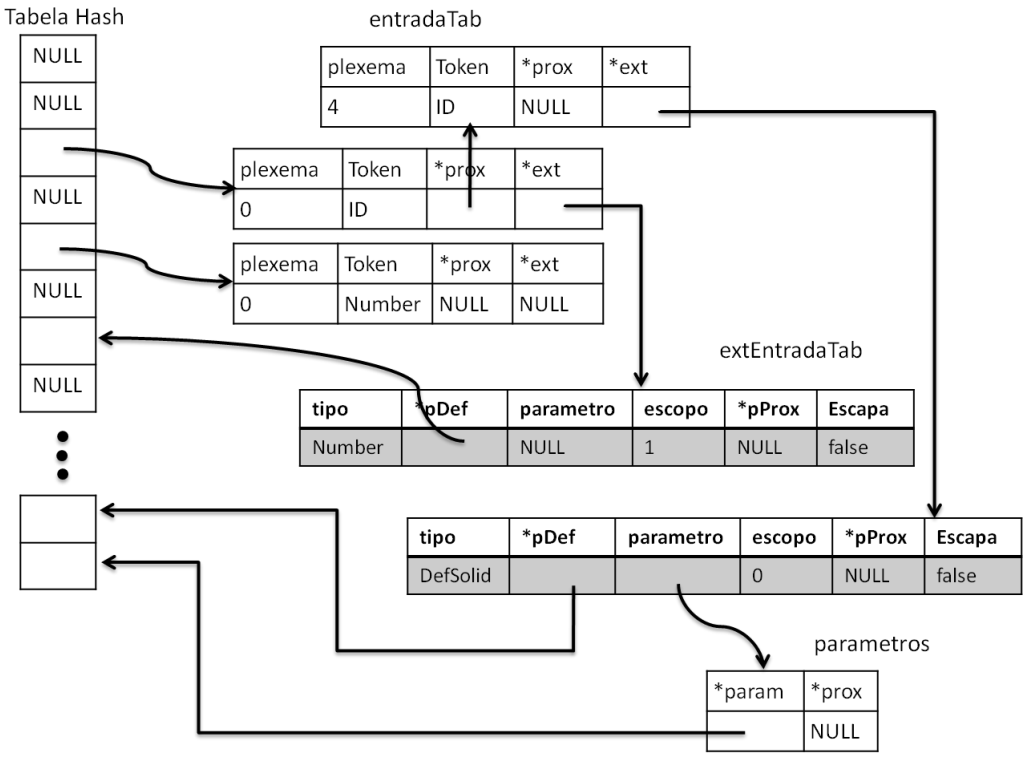


Figura 8.7: Tabela de Símbolos Completa.

de maneira empírica e indicam quais as associações, entre tipos dados e expressões, são suportadas pela linguagem. As seguintes regras semânticas foram utilizadas:

1. Os valores atribuídos a uma propriedade, na sua definição ou na alteração do seu valor para um elemento da base, devem corresponder ao tipo declarado no momento criação da mesma: *true* ou *false* para propriedades declaradas com o tipo *bool*, um valor inteiro para propriedades declaradas como *int* e um valor numérico qualquer para propriedades declaradas como *float*;

2. Para sistema cristalino *cubic* é admitido a disposição *primitive*, *body* ou *face*; sistema cristalino *monoclinic* admite disposição *primitive*, *basea*, *baseb* ou *basec*; sistema cristalino *tetragonal* admite disposição *primitive* ou *body*; sistema cristalino *orthorhombic* admite qualquer uma das distribuições possíveis;

3. Seja (α, β, δ) um vetor qualquer, utilizado na definição do sistema cristalino da estrutura ou na coordenada de uma base, α, β e δ devem ser valores numéricos, tipo *Number*;

4. Não podem existir duas declarações de sólidos ou composições definidas pelo mesmo identificador;

5. Todos os parâmetros passados para a construção de sólidos ou composições devem ser do tipo *Number*;

6. As expressões que definem o *lattice* devem ser do tipo *Number*;

7. Uma atribuição $\alpha = \beta$ é válida se e somente se β já estiver envolvido em alguma atribuição do tipo $\beta = \delta$ anterior ou esta atribuição estiver dentro da definição de um sólido (*solid* ou *composite*) e δ for um parâmetro;

8. As operações && (E lógico) e || (OU lógico) são permitidas apenas entre expressões que retornam um tipo booleano. A construção α && β é válida se e somente se o resultado da expressão α e da expressão β forem booleanos, ou seja *true* ou *false*;

9. As funções matemáticas, operações aritméticas e expressões relacionais descritas na linguagem, tais como, *sin*, *cos*, *atan*, $+$, -, *pow*, *sqrt* $>$, $<$, $==$ são válidas apenas quando as expressões envolvidas são do tipo *Number*. As construções $\alpha + \beta$, $pow(\alpha, \beta)$, $\alpha > \beta$ são válidas se e somente se o resultado da expressão α e o resultado da expressão β são numéricos;

10. As operações *union*, *intersection*, *difference* e *not* são válidas apenas quando as expressões envolvidas são do tipo *Solid*. A construção $union(\alpha, \beta)$ é válida se, e somente se o resultado da expressão α e o resultado da expressão β são sólidos;

11. O resultado da expressão utilizada na instrução *while* deve ser booleano.

12. As operações *rotatex*, *rotatey*, *rotatez*, *scale*, *translate* e *shear* são válidas quando a primeira expressão envolvida é do tipo *Solid* e as demais são do tipo *Number*. $scale(\alpha, \beta, \delta, \phi)$ é válido se e somente se α for do tipo *Solid* e β, δ e ϕ forem do tipo *Number*;

13. A única expressão que pode conter as palavras reservadas *x*, *y* e *z* é aquela que define a função característica de algum sólido. Essa expressão é utilizada na definição do sólido e é descrita na Tabela 7.11;

14. A primeira expressão atrelada à instrução *insert* deve ser do tipo *Solid*;

15. A instrução *insert* dentro de uma composição, *composite*, possui apenas um parâmetro, o sólido a ser incorporado a composição;

16. A expressão que define a rotação da célula unitária em uma inserção deve ser do tipo *Number*.

8.5.1 Funcionamento do Módulo

O MSM deve analisar todos os nós da ASA validando os tipos de dados encontrados. A validação é feita de maneira distinta, conforme o nó visitado. Isso se deve ao fato do nó poder ser um objeto de qualquer uma das classes definidas na Seção 8.4.2, representando elementos distintos da linguagem: Atribuição, expressão matemática ou a definição de um objeto, por exemplo.

A fim de não poluir as classes com código referente as validações semânticas o padrão de projetos *Visitor*, descrito em [26], foi adotado para implementação da análise semântica. As operações semânticas são encapsuladas por uma classe que define quais operações devem ser realizadas para cada nó da ASA. A Figura 8.8 apresenta o diagrama referente a implementação do padrão *Visitor*.

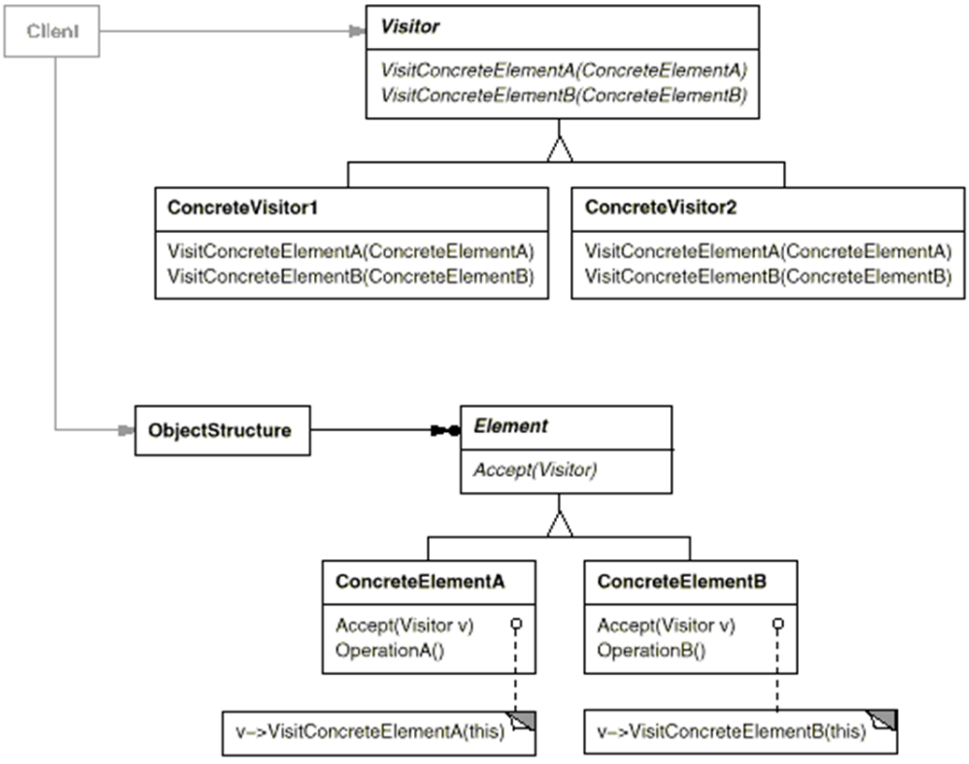


Figura 8.8: Diagrama de Implementação do Padrão *Visitor*. Retirado de [26].

Os elementos constituintes do padrão *Visitor* são descritos da seguinte maneira:

Visitor: Classe abstrata que declara uma operação de visita (*Visit*) para cada classe a ser

visitada (*ConcreteElement*). A classe *Visitor* desempenha esse papel no compilador, nela são declaradas as operações de visita para cada tipo de nó da ASA;

ConcreteVisitor: Classes concretas que implementam cada operação de visita declarada pelo *Visitor*. No compilador foram definidas duas classes concretas: *VisitorSemantico* (implementa as operações de visita com as instruções necessárias para a realização da análise semântica) e *VisitorImpressão* (implementa as operações de visita com as instruções necessárias para a impressão do conteúdo da ASA;

Element: Define a operação *Accept* que recebe um *Visitor* como argumento. As classes abstratas que definem elementos constituintes da ASA desempenham esse papel no compilador;

ConcreteElement: Implementa a operação *Accept* que recebe um *Visitor* como argumento. As classes concretas que definem elementos constituintes da ASA desempenham esse papel no compilador;

ObjectStructure: Objetos ou coleção de objetos da classe *Element* que utilizam a operação *Accept*. Representam os nós da ASA.

O módulo MSM executa a análise semântica através da invocação da operação *Accept* do nó inicial da ASA, que por sua vez executa as operações descritas por seu *visitor* e invoca o *Accept* dos nós subsequentes. Os erros semânticos encontrados são enviados ao MGE.

8.6 Módulo Interpretador (MIN)

Ao término de todas as análises, caso nenhum erro seja encontrado, o código fonte está apto a ser transformado em uma estrutura cristalina. Essa transformação, gerida pelo MIN, é feita através da interpretação do código fonte, nesse momento, expresso pela ASA. A interpretação do código é feita através da implementação das instruções descritas por cada nó da ASA.

Seguindo o mesmo conceito adotado no MSN (não poluir as classes da ASA com código referente as validações), o padrão *Visitor* foi utilizado. O código referente a interpretação

foi encapsulado em uma classe, denominada *visitorInterpretador*, e nas classes correspondentes a cada elemento da ASA foram incluídos um método *Accept*, responsável invocação do *visitor* interpretador.

Além das informações relacionadas à estrutura (propriedades, geometria, vetores que definem a célula unitária, elementos da base e definições de sólidos e composições), o conteúdo das variáveis utilizadas e as inserções realizadas são armazenados a medida que o programa principal é interpretado. Para tanto, as seguintes estruturas foram criadas:

Tabela de Variáveis: Tabela *Hash* implementada nos mesmos moldes que a Tabelas de Símbolos (Seção 8.3.1) e que tem por objetivo armazenar os dados referentes a todas as variáveis utilizadas pelo interpretador.

entradaTabVar: Lista encadeada que armazena, em cada item, as características de uma variável: Lexema, escopo (valor inteiro que representa em qual escopo a variável se encontra) e seu conteúdo (valor numérico, uma expressão ou um sólido).

Lista de inserções: Lista encadeada contendo todas as inserções feitas no programa principal. Cada elemento da lista representa uma inserção realizada e armazena as células provenientes da inserção.

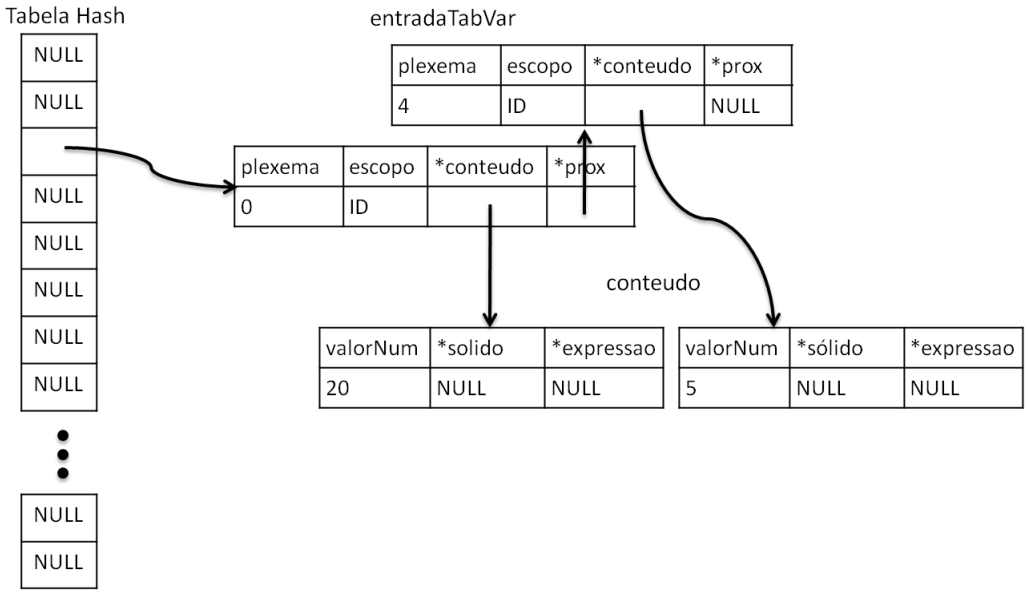
Célula: Armazena as características de uma célula unitária inserida. Possui as coordenadas referentes as extremidades da célula e os pontos (átomos ou moléculas) que a constituem.

Árvore AVL Multidimensional: Árvore AVL modificada responsável pelo armazenamento dos pontos das células.

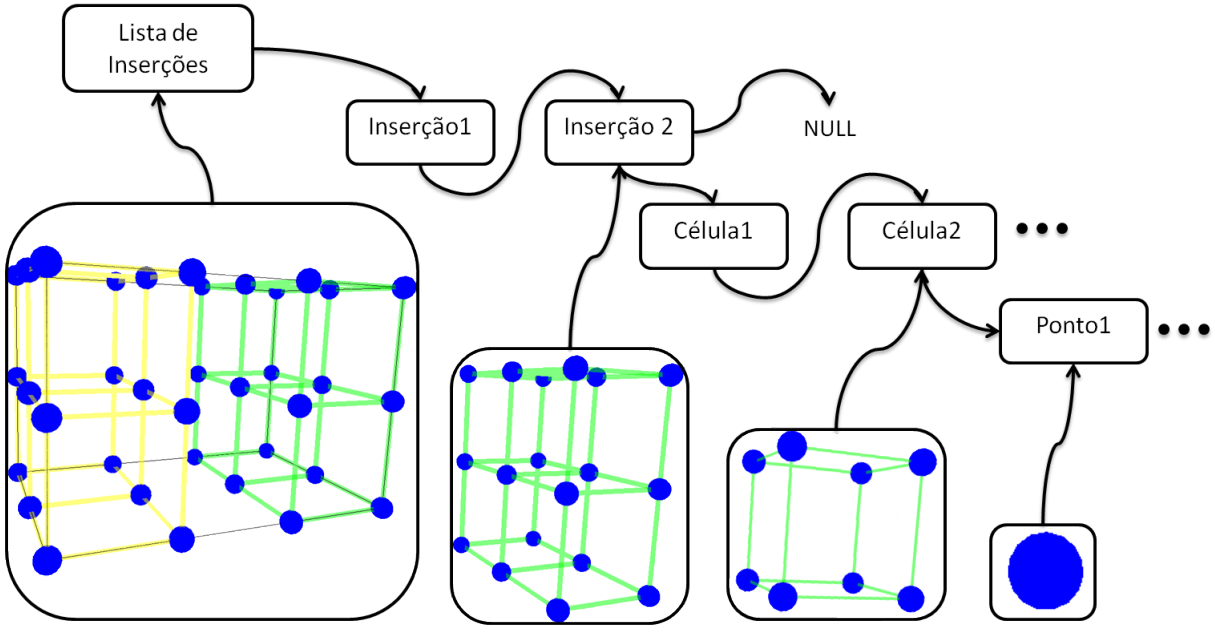
A Figura 8.9 apresenta as estruturas descritas acima e como as mesmas se relacionam.

8.6.1 Armazenamento dos elementos das células

A linguagem possibilita a inclusão de um número indeterminado de elementos (átomos ou moléculas) no espaço. Este número é determinado pelo *lattice* (instrução que define o espaço utilizado) e pela abrangência espacial dos objetos incorporados através da instrução *insert*. A cada novo objeto inserido, é necessário pesquisar por interseções entre as células



(a) Estruturas destinadas ao armazenamento do conteúdo das variáveis.



(b) Estruturas responsáveis pelo armazenamento das estruturas cristalinas inseridas.

Figura 8.9: Estruturas de armazenamento utilizadas pelo MIN.

do mesmo com as células dos objetos preexistentes, efetuando, se necessário, a remoção das células dos objetos antigos (*override*), indicação de erro ao usuário (*error*) ou não inclusão da célula do novo objeto (*nooverride*).

Para suprir todas as demandas enunciadas acima, é necessária a utilização de uma estrutura de dados que, além de armazenar, promova uma organização espacial dos átomos ou moléculas que otimize as operações de inserção, busca e remoção de elementos.

Na literatura são apresentadas várias estruturas de dados com aporte a distribuição espacial de dados, tais como: *k-d-Tree* [27], *BSP-Tree* [28], *R-Tree* [29], *Quad-Tree* [30] e *Grid File* [31]. Esses métodos, bem como algumas variações, são amplamente discutidos por Gaede e Günther [32].

A utilização das estruturas *k-d-Tree*, *BSP-Tree* e *R-Tree* se mostrou inviável devido as mesmas serem implementadas a partir de árvores binárias não balanceadas. A ordem na qual os elementos são inseridos influência no seu desempenho, no pior caso, transformando a árvore em uma lista cuja as operações de busca são feitas de modo sequencial [32].

Como a quantidade de elementos e o espaço no qual esses elementos estarão inseridos é indeterminado, optamos por não utilizar a estrutura *Grid File*. A definição da função *hash* de espalhamento no qual o método se baseia representaria um grande problema.

Segundo Gaede e Günther [32], existe uma infinidade de métodos de acesso espaciais. Mesmo para os peritos, torna-se cada vez mais difícil reconhecer as qualidades e os defeitos de cada método. Nenhum método provou-se superior uma vez que muitas vezes seu desempenho está atrelado ao contexto no qual é utilizado. Partindo dessa premissa, optou-se por não utilizar nenhuma adaptação das estruturas acima. Neste trabalho utilizamos uma árvore de busca binária balanceada adaptada para o armazenamento das coordenadas dos átomos.

8.6.1.1 Árvore AVL Multidimensional

Uma *Árvore AVL* é uma árvore binária que se mantém balanceada independente da ordem em que os elementos são inseridos [33]. O balanceamento ocorre de maneira dinâmica: A cada inserção é verificada se a diferença entre as alturas das subárvores à esquerda e à direita de um nó estão compreendidas entre -1 e 1. Caso isso não ocorra, os nós da árvore são rotacionados a fim de tornar essa relação verdadeira. A implementação de uma *árvore AVL*, com suas operações de inserção, busca, remoção e rotações para balanceamento é

apresentada por Ascencio [25].

As operações de inserção, busca e remoção de elementos em uma *árvore AVL* possuem no pior caso complexidade $O(\log n)$, onde n corresponde ao número de elementos da árvore. Se valendo desta característica, promovemos algumas alterações na árvore para que a mesma pudesse armazenar os elementos da estrutura, ordenado-os segundo sua localização espacial (x, y, z). Esta estrutura foi denominada Árvore AVL multidimensional.

Além de armazenar a informação de um átomo ou molécula, cada nó de uma *árvore AVL* convencional armazenaria dois ponteiros (apontam para as subárvores à direita e à esquerda do nó, caso existam) e dois inteiros (armazenam a altura das subárvores referenciadas pelo ponteiros). Na AVL multidimensional em cada nó é incluído o valor de umas das coordenadas do átomo (x, y ou z) e um ponteiro para uma outra árvore.

A AVL Multidimensional pode ser compreendida como um conjunto de árvores AVL interligadas. Ela é composta por uma árvore principal, responsável por armazenar as coordenadas referentes ao eixo x de todos os átomos, um conjunto de árvores secundárias cuja a função é armazenar as coordenadas referentes ao eixo y e um conjunto de árvores finais que, além de armazenar a coordenada relacionada ao eixo z de um átomo, armazenam as informações de um átomo em cada nó.

Cada nó da árvore principal se liga com o nó raiz de uma árvore secundária e cada nó da árvore secundária se liga à um nó raiz de uma árvore final. A Figura 8.10 apresenta a inclusão da informação de um átomo A, localizado na coordenada $(0, 1, 2)$, na árvore AVL multidimensional, supondo que a mesma não guarde nenhuma informação. Como resultado temos três árvores AVL interligadas. A partir desse resultado são inseridas informações referentes a outros dois átomos B e C, B situado na coordenada $(0, 1, 3)$ e C na coordenada $(-1, 1, 1)$ (Figura 8.11).

A árvore principal classifica todos os elementos segundo a coordenada x, agrupando os elementos distintos em um mesmo nó caso as coordenadas no eixo x sejam iguais (as informações desses elementos serão acessadas a partir de um mesmo nó). Em cada árvore secundária os elementos são classificados segundo a coordenada y, ocorrendo agrupamentos em um mesmo nó de todos os elementos que possuam valores iguais nessa coordenada. Deve-se ressaltar que todos os elementos acessíveis a partir de uma determinada árvore secundária possuem as coordenadas x iguais. Por fim cada árvore final classifica os elementos (com as coordenadas x e y iguais) segundo a coordena z.

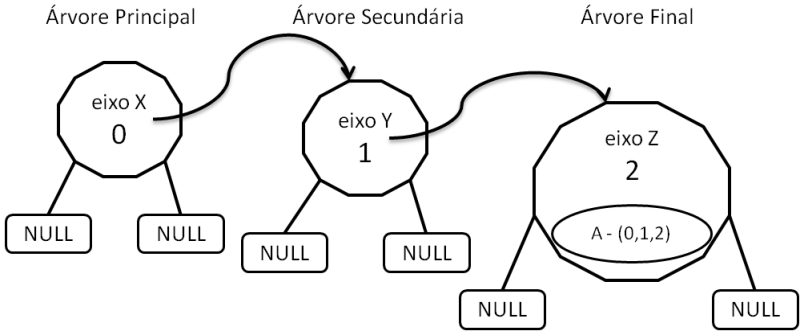


Figura 8.10: Inclusão das informações referentes a um átomo em uma árvore AVL multidimensional. Os retângulos com a palavra *NULL* indicam ausência de subárvore e as setas indicam a comunicação entre um nó e uma árvore distinta.

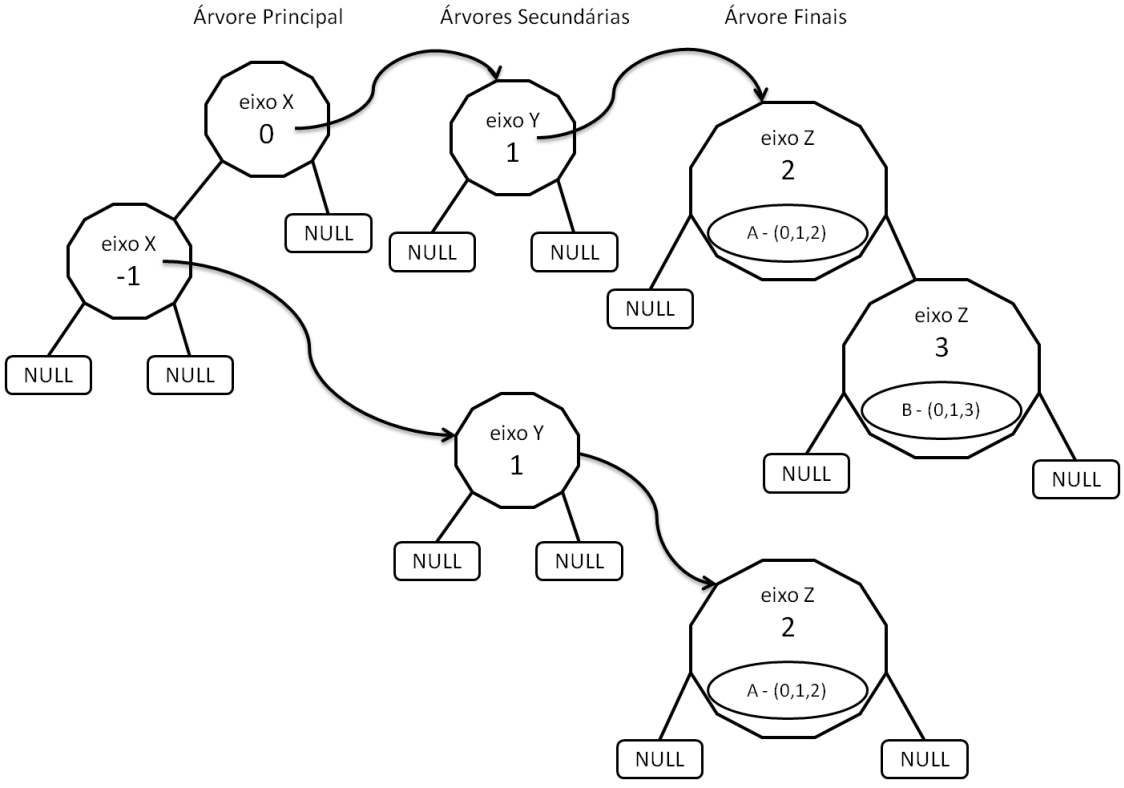


Figura 8.11: Representação de uma árvore AVL multidimensional composta por uma árvore principal, duas árvores secundárias e duas árvores finais. Os retângulos com a palavra *NULL* indicam ausência de subárvore e as setas indicam a comunicação entre um nó e uma árvore distinta.

Na Figura 8.11, a distribuição dos elementos é facilmente observada. As informações sobre os átomos A e B são acessíveis a partir do mesmo nó na árvore principal e na árvore secundária, uma vez que eles possuem as coordenadas x e y iguais. O átomo C é acessível através de outra árvore secundária e final, uma vez que o mesmo possui coordenada x diferente dos demais átomos.

8.6.2 Funcionamento do Módulo

O Código 8.2 apresenta todas as construções possíveis na linguagem. A partir dele as ações realizadas pelo MIN serão apresentadas.

```
1  {Definições dos Elementos da Estrutura}
2  property bool metal = false
3  property int numAtomico = 0
4  cubic(primitive,(1,0,0),(0,1,0),(0,0,1))
5  base(Cs, (0,0,0), metal = true, numAtomico = 55)
6  base(Cl, (0.5,0.5,0.5), numAtomico = 17)
7
8  {Esfera de centro (a,b,c) e raio r}
9  solid Esfera(a,b,c,r)
10     pow((x-a),2) + pow((y-b),2) + pow((z-c),2) <= pow(r,2);
11
12 {Casca Esférica esfera de centro (a,b,c) com raio interno
       r1 e externo r2}
13 composite CascaEsfera(a,b,c,r1,r2)
14 begin
15     insert(difference(Esfera(a,b,c,r2),Esfera(a,b,c,r1)));
16 end
17
18 {Programa Principal}
19 lattice(-12,-8,-9,12,8,6)
20 begin
21     pi = 3.1416;
22     rot = pi/3;
23     i = -6;
24     while (i <= 6)
25     begin
26         Esf = Esfera(i,0,0,5);
27         insert(Esf,override,(rot,x));
28         rot = 2 * rot;
29         i = i + 12;
30     end;
31     insert(CascaEsfera(0,0,0,6,9),nooverride);
32 end
```

Código 8.2: Código fonte destacando as estruturas analisadas pelas duas etapas do MIN.

O processo de interpretação é dividido em duas etapas: A primeira consiste no armazenamento das definições referentes à estrutura (propriedades, geometria, vetores que

definem a célula unitária, elementos da base e definições de sólidos e composições) e a segunda na interpretação do programa principal.

No Código 8.2 as linhas com definições referentes à estrutura estão compreendidas entre as linhas 2 e 16 e são analisadas, pelo MIN, da seguinte maneira:

linhas 2 e 3: As propriedades definidas são armazenadas. Uma propriedade booleana, definida pelo lexema *metal*, contendo o valor *false* (linha 2) e outra inteira, definida pelo lexema *numAtomico*, com o valor 0 (linha 3).

linha 4: Armazena os dados referentes ao retículo de *Bravais* escolhido verificando a correta relação entre os elementos que o constituem. No exemplo em questão, verifica se o retículo cúbico (*cubic*) possui a distribuição atômica *primitive* e pode ser representado pelos vetores ($(1,0,0)$, $(0,1,0)$ e $(0,0,1)$). Essa verificação é regida pelas características de cada retículo, apresentadas na Seção 2.2.

linhas 5 e 6: Os elementos da base são armazenados. O elemento de lexema *Cs* é armazenado com o valor *true* para a propriedade *metal*, *55* para a propriedade *numAtomico* e posição descrita pelo vetor $(0,0,0)$ (linha 5). O elemento *Cl* é armazenado com o valor *false* para a propriedade *metal* (o interpretador utiliza o valor padrão declarado para uma propriedade caso não seja feita a atribuição de um novo valor), *17* para a propriedade *numAtomico* e posição descrita pelo vetor $(0.5, 0.5, 0.5)$ (linha 6).

linhas 9 e 13: As referências para as árvores (elementos da ASA) que representam as expressões ou códigos que descrevem sólidos e composições são armazenadas. Uma referência para o nó da ASA que descreve a expressão do sólido referenciado pelo lexema *Esfera* é armazenada na linha 9 e uma referência para o nó da ASA que descreve as instruções necessárias para a geração da composição *CascaEsfera* é armazenada na linha 13.

A interpretação do programa principal, compreendido a partir da linha 19 no Código 8.2, ocorre através dos seguintes passos:

linha 19: O *lattice* descrito no código é armazenado para utilização nas inserções. No Código em questão ele indica que os objetos estarão compreendidos no interior do retângulo compreendido entre as coordenadas $(-12, -8, -9)$ e $(12, 8, 6)$.

linhas 21, 22, 23, 28 e 29: A expressão à direita do sinal de atribuição (=) é analisada e seu valor é atribuído à variável indicada pelo lexema à esquerda. Por exemplo, na linha 21, o valor 3.1416 é atribuído a variável *pi*. Os valores previamente armazenados na variável são removidos antes que um novo valor seja atribuído.

linha 24: Enquanto a expressão $i \leq 6$ for verdadeira a árvore que representa o código compreendido entre as linhas 25 e 30 é interpretada.

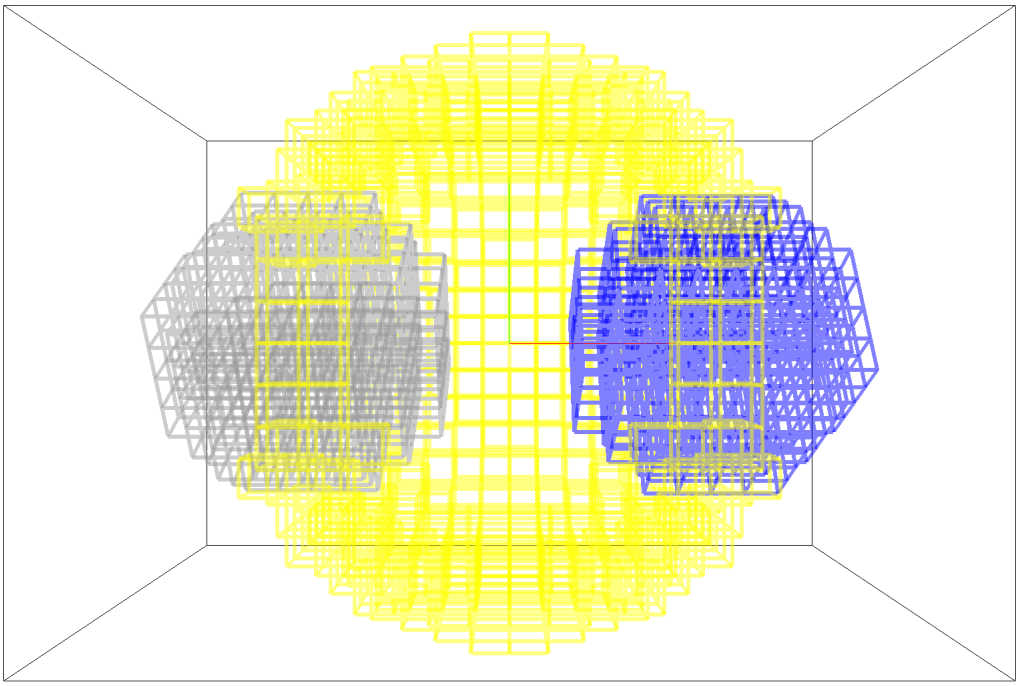
linha 26: O nó da ASA que representa o objeto, à direita do sinal de atribuição, é duplicado com suas expressões numéricas analisadas. Posteriormente, esse conteúdo é atribuído a variável *Esf*. Como nas atribuições envolvendo valores numéricos, o conteúdo previamente armazenado em *Esf* é removido.

linha 27: A inserção do objeto definido pela variável *Esf* é feita. Para tanto as seguintes ações são realizadas:

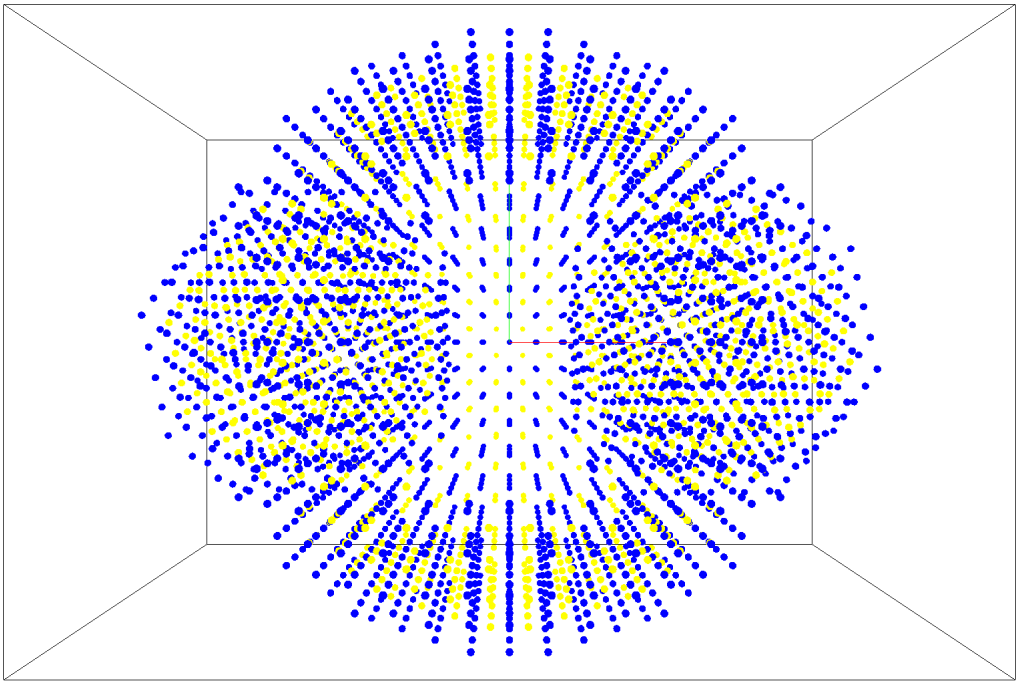
- Os vetores definidos para a célula unitária são rotacionados em relação ao eixo *x*, no valor definido pela variável *rot*;
- A árvore que descreve o objeto *Esf* é analisada para todos os pontos, gerados a partir dos vetores rotacionados, que estão contidos no *lattice*. A partir dessa análise as células são formadas;
- Apaga as células, oriundas de inserções anteriores, que interceptam as células criadas (opção *override*). A interseção entre as células é verificada através do Algoritmo GJK, descrito no Capítulo 6.

linha 31: A inserção da composição *CascaEsfera* é feita. A inserção segue os mesmos passos descritos na inserção feita na linha 27. Contudo os vetores utilizados para geração dos pontos equivalem aos vetores que compõem a célula unitária (não há rotação) e não há remoção das células, oriundas de inserções anteriores, que interceptam as novas células criadas. Com o parâmetro *nooverride*, as células novas que interceptam células de inserções anteriores não são inseridas.

As estrutura gerada a partir do Código 8.2 é exibida na Figura 8.12



(a) Visão das células unitárias pertencentes a estrutura gerada a partir do Código 8.2.



(b) Visão dos pontos pertencentes a estrutura gerada a partir do Código 8.2.

Figura 8.12: Estrutura Cristalina gerada a partir do Código 8.2. A Figura 8.12(a) exibe as células unitárias e a Figura 8.12(b) exibe os pontos (átomos) pertencentes a estrutura.

8.7 Módulo Gerenciador de Erros (MGE)

A criação de um módulo para gerenciamento de erros foi adotada apenas para uma maior organização do código do compilador. Sua função é reportar ao usuário todos os erros encontrados no processo de compilação, descrevendo suas características e localização (linha e coluna do código fonte). Os erros são divididos nas seguintes categorias:

Erros no arquivo fonte: Ocorre quando o arquivo especificado ao compilador não é encontrado, possui extensão incorreta ou não pode ser aberto.

Erros Léxicos: Representam os erros provenientes da análise léxica, sendo reportados pelo MLX. As mensagens enviadas ao usuário são formuladas a partir de uma constante de erro enviada pelo MLX.

Erros Sintáticos: Erros, oriundos da análise sintática, reportados pelo MSN. Como ocorre nos erros léxicos, as mensagens são produzidas a partir de uma constante de erro.

Erros Semânticos: Erros de cunho semântico, reportados pelo MSM. O MGE encaminha as mensagens de erro, formuladas pelo MSM, ao usuário.

Erros de Interpretação: Erros que ocorrem no processo de interpretação da ASA. Como nos erros semânticos, a mensagem é formulada pelo módulo que detecta o erro, MIN, cabendo ao MGE apenas reportá-lo ao usuário.

O Apêndice B apresenta todos os tipos de erros informados pelo compilador com suas respectivas mensagens de alerta.

9 RESULTADOS OBTIDOS

9.1 Introdução

A aplicabilidade da linguagem de geração de estruturas cristalinas proposta está diretamente ligada ao seu poder de expressão (capacidades de gerar as mais variadas estruturas pertencentes a qualquer um dos sistemas cristalinos existentes).

Este capítulo, além de apresentar os resultados obtidos, tem por objetivo mostrar toda a capacidade de definição e geração de estruturas cristalinas contidas na linguagem. Inicialmente são apresentados alguns elementos químicos cuja a geometria de cristalização é conhecida, descrevendo-os do modo como são encontrados na literatura, e em seguida as suas representações através da linguagem proposta neste trabalho. Também são apresentadas as representações gráficas dos elementos obtidos. Por fim, serão apresentados exemplos de sólidos gerados a partir da linguagem e um exemplo de incorporação de um objeto a um simulador existente.

9.2 Células Unitárias

9.2.1 Rede Cúbica Simples

A única substância simples que pode se cristalizar nessa forma é o polônio, *Po*. Existem, em contrapartida, vários compostos cujos elementos estão dispostos nesta rede. O cristal de cloreto de césio ($CsCl$), por exemplo, tem a posição dos seus átomos definidas a partir de uma rede cúbica simples. O cristal de $CsCl$ é descrito da seguinte maneira: Rede com tamanho equivalente a $4,11\mathring{A}$ (O *Ansgtron*, $\mathring{A}$, é a unidade básica de comprimento utilizada na literatura e equivale a $10^{-10}m$, $0,1nm$) tendo os átomos de Césio, Cs localizados nas posições definidas pela rede e os átomos de Cloro (Cl) no centro de cada célula unitária [34]. Adaptando as informações à linguagem de geração de estruturas temos:

- Os vetores ($\vec{a}$,$\vec{b}$,$\vec{c}$), que compõem a base, são ortogonais entre si e possuem mesmo módulo (condição imposta pelo sistema cúbico);
- O módulo dos vetores equivalem a $4,11\mathring{A}$;

- A base possui dois elementos: Cs e Cl tendo posição definida pelo vetores $(0, 0, 0)$ e $\frac{1}{2}(\vec{a} + \vec{b} + \vec{c})$, respectivamente.

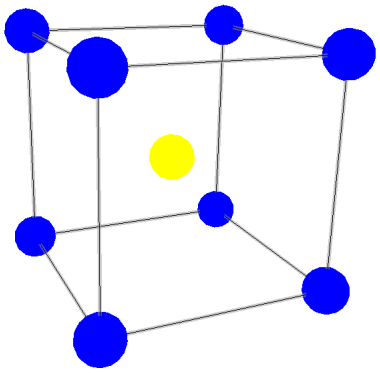
A Figura 9.1(b) apresenta o trecho de código que define a célula unitária do cristal $CsCl$ (9.1(a)) e a visualização de sua célula unitária(9.1(b)).

```
cubic(primitive,(4.11,0,0),(0,4.11,0),(0,0,4.11))

base(Cs,  (0,0,0))
base(Cl,  (2.055,2.055,2.055))
```

.

(a) Trecho de código contendo a especificação do cristal $CsCl$.



(b) Visualização da célula unitária do cristal $CsCl$. Em azul aparecem os átomos de Cs e em amarelo os átomos de Cl.

Figura 9.1: Definição da estrutura cristalina do Cloreto de Césio.

A Tabela 9.1 apresenta alguns compostos cuja cristalização pode ser descrita pela rede cúbica simples.

Tabela 9.1: Compostos que apresentam cristais descritos por uma rede cúbica simples. Adaptado de [34].

ELEMENTO	TAMANHO CÉLULA ($\AA$)	ELEMENTO	TAMANHO CÉLULA ($\AA$)
$CsCl$	4,11	NH_4Cl	3,87
$CsBr$	4,29	$CuZn$	2,94
CsI	4,56	$AgMg$	3,28
$TlCl$	3,84	$LiHg$	3,29
$TlBr$	3,97	$AlNi$	2,88
TlI	3,74	$BeCu$	2,70

9.2.2 Rede Cúbica de Faces Centradas

Muitos elementos apresentam estrutura cristalinas deste tipo, em virtude do grande fator de empacotamento desta rede (porção do espaço da célula unitária ocupada por átomos) [35].

O cloreto de sódio, sal de cozinha, apresenta estrutura cúbica de face centrada com parâmetro de rede igual a $5,64\mathring{A}$ e dois átomos na base (Na localizado no centro da célula unitária e Cl na origem, $(0,0,0)$).

Adaptando as informações à linguagem de geração de estruturas temos: Os vetores ($\vec{a}$,$\vec{b}$,$\vec{c}$) ortogonais entre si e com módulo correspondente a $5,64$ e base representada por um átomo de Na, com posição definida pelo vetor $\frac{1}{2}(\vec{a}+\vec{b}+\vec{c})$, e um átomo de Cl na coordenada $(0,0,0)$.

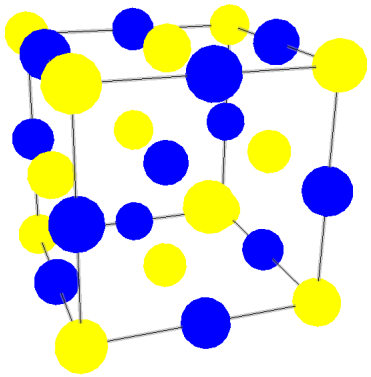
Na Figura 9.2 são exibidos o trecho de código que define a célula unitária do $NaCl$ (9.2(a)) e sua visualização (9.2(b)).

```
cubic(face,(5.64,0,0),(0,5.64,0),(0,0,5.64))

base(Na,  (2.82,2.82,2.82))
base(Cl,  (0,0,0))
```

.

(a) Trecho de código contendo a especificação da célula unitária do cristal $NaCl$.



(b) Visualização da célula unitária do cristal $NaCl$. Em azul aparecem os átomos de sódio e em amarelo os átomos de cloro.

Figura 9.2: Definição da estrutura cristalina do cloreto de sódio.

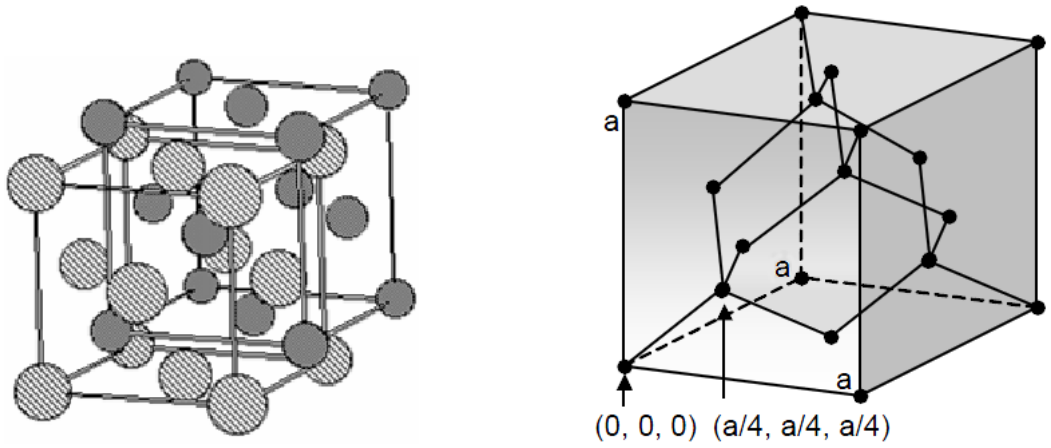
A Tabela 9.2 apresenta outros elementos que cristalizam em uma estrutura cúbica de face centrada.

Tabela 9.2: Compostos que apresentam cristais descritos por uma rede cúbica de face centrada. Adaptado de [34].

ELEMENTO	TAMANHO CÉLULA ($\mathring{A}$)	ELEMENTO	TAMANHO CÉLULA ($\mathring{A}$)
Cu	3,61	$NaCl$	5,63
Ag	4,08	LiF	4,02
Au	4,07	KCl	6,28
Al	4,04	$LiBr$	5,49

9.2.3 Estrutura Diamante

Além do carbono, o silício e germânio se cristalizam dessa forma. Esta estrutura está relacionada com a estrutura de vários compostos semicondutores binários importantes [34]. A estrutura diamante pode ser vista como duas redes cúbicas de faces centrada intercaladas, deslocadas ao longo da diagonal do cubo e distantes uma da outra de $\frac{1}{4}$ da diagonal da célula unitária. Ashcroft e Mermin [36] bem como Ibach e Luth [37] a descrevem a partir de uma rede *fcc* contendo dois átomos na base: Um localizado na origem e outra localizada à $\frac{1}{4}$ a partir da origem, na diagonal da célula inicial. A Figura 9.3 exemplifica a estrutura diamante, apresentando a visualização da mesma através da intercalação de duas redes *fcc*(9.3(a)) e através de sua especificação a partir de uma rede *fcc*.



(a) Intercalação de duas redes *fcc* formando a estrutura do diamante.

(b) Célula Unitária da estrutura diamante, descrita a partir de uma rede *fcc*. Adaptado de [35].

Figura 9.3: Representações da estrutura diamante.

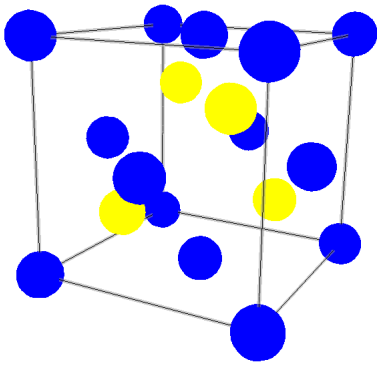
Exemplificando a estrutura diamante temos o carbono, C. este elemento pode se cristalizar dessa forma, apresentando célula unitária medindo $3,56\mathring{A}$ de comprimento. Sua especificação na linguagem é feita a partir de três vetores ($\vec{a}$, $\vec{b}$, $\vec{c}$), ortogonais entre si e com módulo igual a 3.56, e dois átomos na base (C1 e C2) tendo posição especificada pelos vetores $(0,0,0)$ e $\frac{1}{4}(\vec{a}+\vec{b}+\vec{c})$. Na Figura 9.4 são exibidos o trecho de código que define a célula unitária do diamante (9.4(a)) e sua visualização (9.4(b)).

```
cubic(face,(3.56,0,0),(0,3.56,0),(0,0,3.56))

base(C1,  (0,0,0))
base(C2,  (0.89,0.89,0.89))
```

.

(a) Trecho de código contendo a especificação da célula unitária do diamante.



(b) Visualização da célula unitária do diamante. Em azul aparecem os átomos gerados a partir de *C1* e em amarelos os gerados a partir de *C2*, ambos representando átomos de carbono.

Figura 9.4: Definição da estrutura cristalina do diamante.

9.2.4 Estrutura Hexagonal Compacta

A estrutura hexagonal compacta (*hcp*) não é uma rede cristalina. Contudo, como muitas estruturas apresentam a distribuição dos átomos descrito segundo essa forma, ela é amplamente discutida na literatura. Como o próprio nome sugere, a *hcp* é formada a partir de uma rede hexagonal tendo como base dois elementos: Um situado nos pontos definidos pela rede hexagonal e o outro pela posição $r = \frac{2}{3}\vec{a} + \frac{1}{3}\vec{b} + \frac{1}{2}\vec{c}$, sendo $\vec{a}$, $\vec{b}$ e $\vec{c}$ os vetores que definem a rede hexagonal [38].

O elemento químico Berílio, Be, se cristaliza dessa forma, sendo gerado pela estrutura cristalina hexagonal com as seguintes características: $|\vec{a}| = |\vec{b}| = 2,29\mathring{A}$ e $|\vec{c}| = 3,58\mathring{A}$.

Na Figura 9.5 são exibidos o trecho de código que define a *hpc* que representa a cristalização do Berílio (9.5(a)) e sua visualização (9.5(b)).

9.3 Criação de Objetos

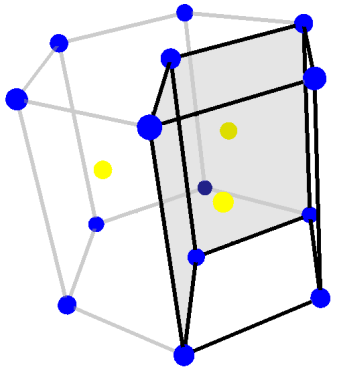
A definição de uma estrutura cristalina, na linguagem, segue os passos descrito no Capítulo 7. Devem estar inclusos no código fonte as definições das propriedades pertencentes átomos ou moléculas da estrutura (Seção 7.3.1); a definição das características da estrutura, tais como sistema cristalino, vetores que compõe a célula unitárias e átomos ou moléculas que constituem a base (Seção 7.3.2); as definições dos sólidos ou composições utilizados (Seção 7.3.3); e o programa principal que descreve quais e como os elementos

```
hexagonal((1.145,-1.983198,0),
          (1.145, 1.983198,0),
          (0,0,3.58))

base(Be1, (0,0,0))
base(Be2, (1.145, -0.661066, 1.79))
```

.

(a) Trecho de código contendo a especificação da *hpc* do Berílio.



(b) Visualização de uma célula hexagonal compacta do Berílio. Em azul aparecem os átomos gerados a partir da rede hexagonal e em amarelos os gerados a partir de $\frac{2}{3}\vec{a} + \frac{1}{3}\vec{b} + \frac{1}{2}\vec{c}$. A célula unitária aparece em destaque.

Figura 9.5: Definição da Estrutura Hexagonal Compacta que descreve a cristalização do elemento químico Berílio.

serão utilizados (Seção 7.3.4).

Nesta seção apresentamos dois código fontes e as estruturas cristalinas geradas a partir compilação dos mesmos.

9.3.1 Esfera Maciça de Berílio

O Código 9.1 descreve uma esfera maciça centrada na origem com raio equivalente a $20\mathring{A}$ preenchida por células *hpc* de Berílio. A descrição da estrutura é feita a partir da definição da geometria hexagonal compacta do elemento Berílio (apresentada em 9.2.4) e da utilização da definição do sólido *Esfera* (descrito na Seção 7.3.3.1) com os parâmetros $a = 0$, $b = 0$, $c = 0$ e $r = 20$. A Figura 9.6 apresenta a estrutura gerada após a compilação do código.

9.3.2 Casca Esférica Furada de Diamante

O Código 9.2 descreve uma casca esférica recortada por oito esferas. Essa estrutura é preenchida por células cúbicas de face centrada, *fcc*, diamante. A célula unitária é descrita a partir da definição da geometria apresentada em 9.2.3.

A estrutura é descrita a partir da utilização de duas composições e um sólido. São elas:

```
1  {Definição das Características da Estrutura}
2  hexagonal((1.145,-1.983198,0),
3            (1.145, 1.983198,0),
4            (0,0,3.58))
5
6  base(Be1, (0,0,0))
7  base(Be2, (1.145, -0.661066, 1.79))
8
9
10 {Esfera de centro (a,b,c) e raio r}
11 solid Esfera(a,b,c,r)
12     pow((x-a), 2) + pow((y-b), 2) + pow((z-c),2) <= pow(r
       ,2);
13
14 {Programa Principal}
15 lattice(-22,-22,-22,22,22,22)
16 begin
17       insert(Esfera(0,0,0,20),override);
18 end
```

Código 9.1: Código que descreve uma estrutura *hpc* de Berílio.

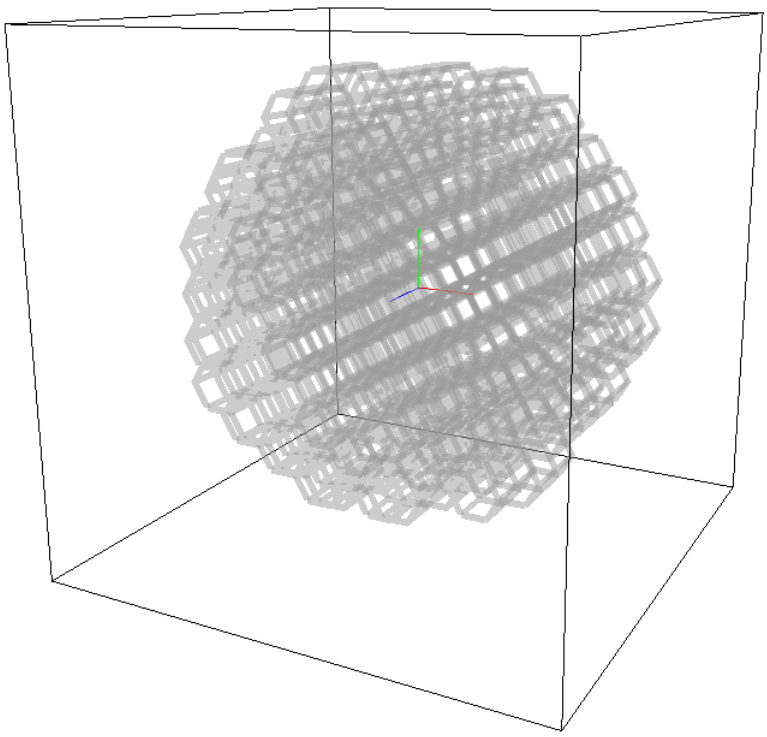


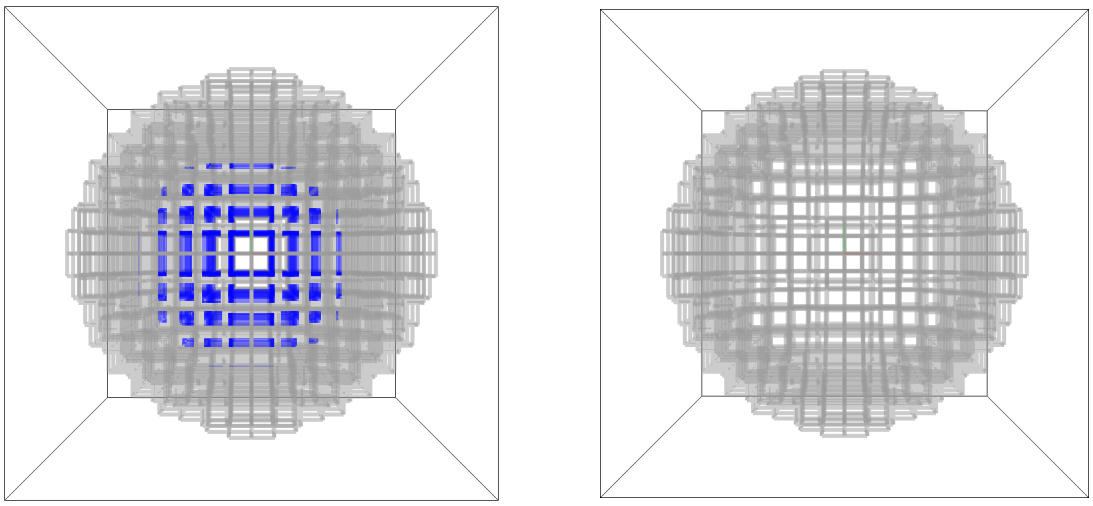
Figura 9.6: Esfera Maciça de Berílio. Proveniente da compilação do Código 9.1. Visualização das células unitárias.

```
1  cubic(face,(3.56,0,0),(0,3.56,0),(0,0,3.56))
2
3  base(C1, (0,0,0))
4  base(C2, (0.89,0.89,0.89))
5
6  {Esfera de centro (a,b,c) e raio r}
7  solid Esfera(a,b,c,r)
8      pow((x-a), 2) + pow((y-b), 2) + pow((z-c),2) <= pow(r
           ,2);
9
10 {casca esferica esfera central com raio r1 e esfera externa
       com raio r2}
11 composite CascaEsfera(a,b,c,r1,r2)
12 begin
13       insert(difference(Esfera(a,b,c,r2),Esfera(a,b,c,r1)))
             ;
14 end
15
16 {Casca esferica furada por oito esferas (de diametro igual
     ao diametro externo da casca)}
17 composite CascaFurada(a,b,c,r1,r2)
18 begin
19       casca = CascaEsfera(a,b,c,r1,r2);
20       casca = difference(casca,Esfera(a+r2,b+r2,c+r2,r2));
21       casca = difference(casca,Esfera(a+r2,b+r2,c-r2,r2));
22       casca = difference(casca,Esfera(a+r2,b-r2,c+r2,r2));
23       casca = difference(casca,Esfera(a+r2,b-r2,c-r2,r2));
24       casca = difference(casca,Esfera(a-r2,b+r2,c+r2,r2));
25       casca = difference(casca,Esfera(a-r2,b+r2,c-r2,r2));
26       casca = difference(casca,Esfera(a-r2,b-r2,c+r2,r2));
27       casca = difference(casca,Esfera(a-r2,b-r2,c-r2,r2));
28       insert(casca);
29 end
30
31 lattice(-30,-30,-30,30,30,30)
32 begin
33       insert(CascaFurada(0,0,0,20,30),override);
34 end
```

Código 9.2: Código que descreve esfera furada preenchida por células *fcc* diamante.

***Esfera* (linha 7):** Sólido que define os pontos pertencentes a um esfera maciça. O centro da esfera é definido pelos parâmetros a, b e c e o raio pelo parâmetro r. A inequação que define os pontos pertencentes a esfera foi apresentada em 7.3.3.1;

***CascaEsfera* (linha 11):** Composição que utiliza a definição do sólido *Esfera* para gerar um casca esférica. A casca esférica é formada subtraindo-se os pontos de duas esfera maciças concêntricas de raios distintos. O centro das duas esferas utilizadas são definidos pelos parâmetros a, b e c e o centro das esferas pelos parâmetros r_1 (centro da esfera menor) e r_2 (centro da esfera maior). Na Figura 9.7 são exibidas as duas esferas (9.7(a)) e a casca esférica gerada usando a composição (9.7(b));

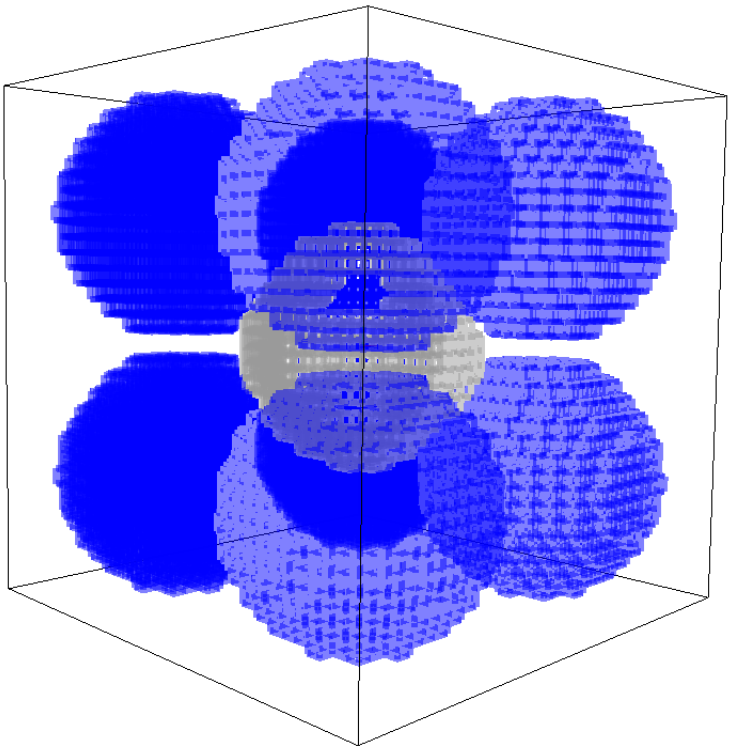


(a) Duas esferas maciças utilizadas para a construção de casca esférica.

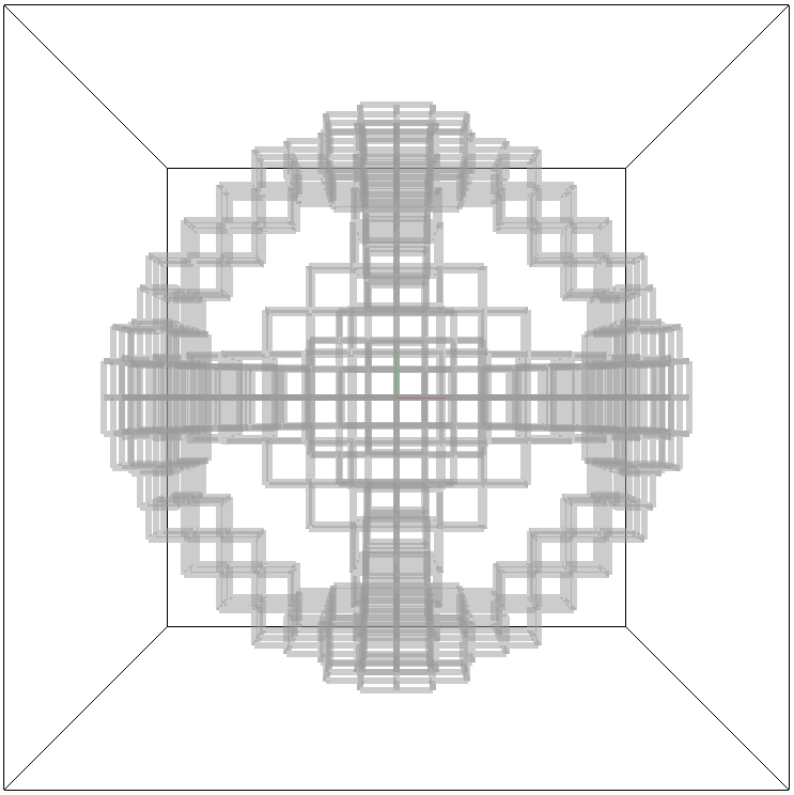
(b) Casca esférica definida pela composição *CascaEsfera*.

Figura 9.7: Composição *CascaEsfera*. Visualização das células unitárias.

***CascaFurada* (linha 17):** Composição que define uma casca esférica furada por oito esferas. A casca esférica é formada usando a composição *CascaEsfera* e são removidos dela os pontos que pertencem a oito esferas com raio equivalente a seu raio externo e centradas em $(a \pm r_2, b \pm r_2, c \pm r_2)$. O parâmetros utilizados por essa composição são os mesmos utilizados na composição *CascaEsfera*. A Figura 9.8 apresenta a casca esférica, com as esferas utilizadas para sua perfuração (9.8(a)), e a estrutura final (9.8(b)).



(a) *CascaEsfera* com as esferas utilizadas para sua perfuração.



(b) Estrutura definida pela composição *CascaFurada*.

Figura 9.8: Composição *CascaFurada*. Visualização das células unitárias.

9.4 Incorporação das estruturas criadas a um simulador

Foge do escopo deste trabalho o desenvolvimento de um módulo completo que permita a incorporação das estruturas cristalinas criadas em um simulador. Entretanto, para demonstrar a aplicabilidade da ferramenta, foi desenvolvido um código que transforma cada um dos pontos das estruturas criadas em uma tupla do tipo (x,y,z), onde x, y e z representam, respectivamente, a posição do ponto no espaço tridimensional. O simulador Monte Carlo Spins Engine também foi modificado, de modo que fosse possível realizar a incorporação dos pontos gerados pelo compilador no ambiente, bem como a sua simulação.

A Figura 9.9 apresenta a interface gráfica do simulador e o resultado obtido a partir da incorporação da estrutura *Casca Esférica Furada de Diamante* descrita pelo Código 9.2.

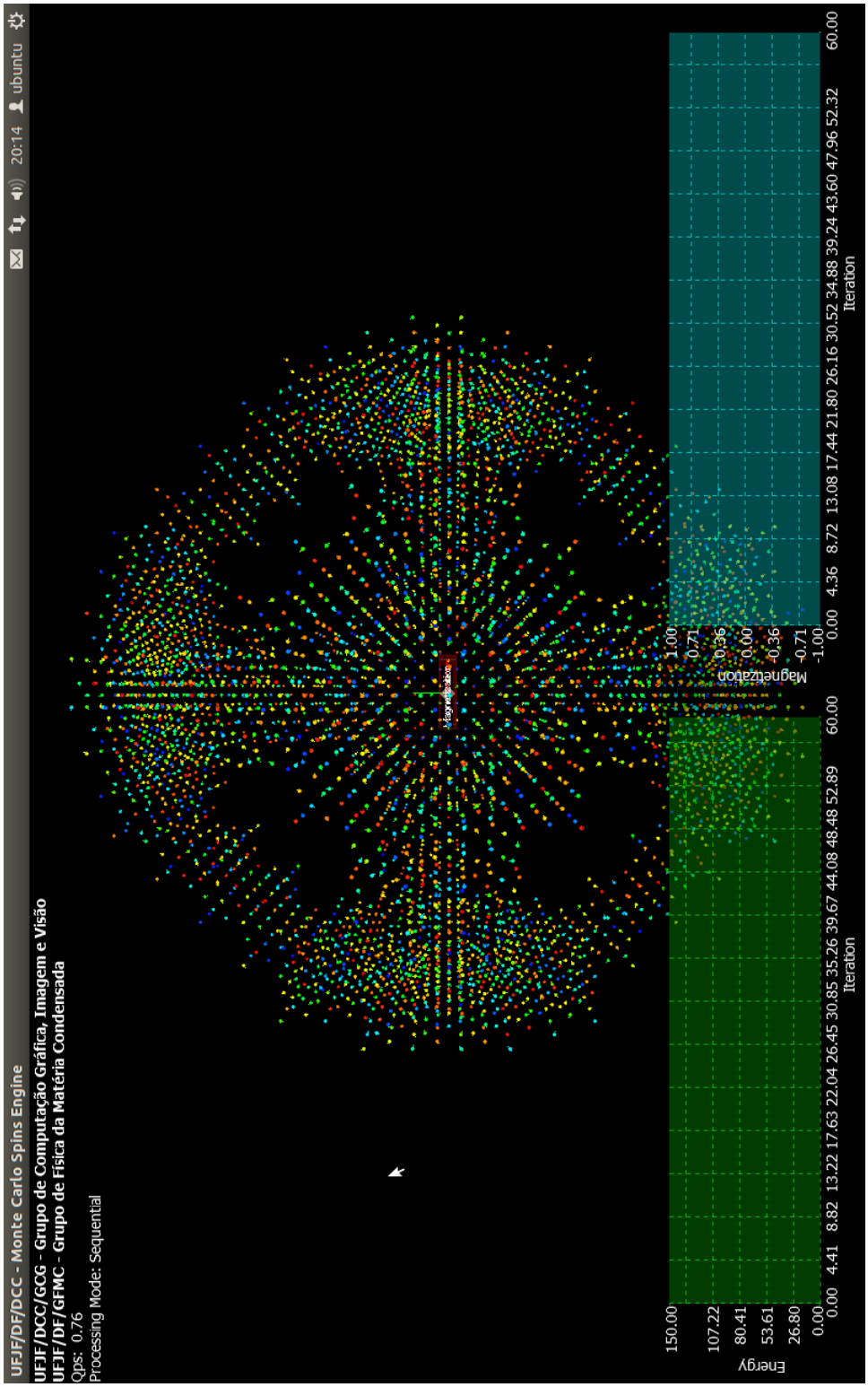


Figura 9.9: Incorporação de uma estrutura *Casca Esférica Furada* ao simulador *MCSE*.

10 CONCLUSÕES E PERSPECTIVAS FUTURAS

Neste trabalho foi apresentada uma ferramenta computacional destinada à criação de estruturas cristalinas para simuladores. Os elementos constituintes da linguagem de programação, utilizada pela ferramenta, foram descritos a partir da teoria relacionada a definição de sistemas cristalinos, a representação de objetos (segundo o modelo implícito) e a criação de objetos complexos (através dos operadores CSG, *Construtive Solid Geometry*).

A divisão e independência entre a definição dos elementos da estrutura (propriedades e características) e a definição de sólidos e composições, presente na linguagem, promoveu portabilidade e agilidade ao processo de criação de uma estrutura. Sólidos e composições, definidos no código de construção de uma determinada estrutura, podem ser utilizados no código da definição de outra, independente das características conferidas às estruturas, sendo necessário apenas que o programador copie o código referente à definição que deseja portar. Objetos complexos podem ser criados facilmente através das mais variadas composições e transformações de objetos já definidos.

O objetivo principal deste trabalho foi atingido, a ferramenta promoveu recursos para a definição e geração de estruturas cristalinas. A linguagem se mostrou simples e com o poder de expressão satisfatório, e o compilador valida de forma eficaz os códigos escritos na linguagem, oferecendo ao usuário um conjunto amplo de mensagens que localizam e descrevem possíveis erros.

O trabalho se mostrou inovador e a definição de uma estrutura cristalina a partir de uma linguagem de programação promissora. Como trabalhos futuros podemos destacar:

Melhorias na Linguagem: Embora a linguagem tenha se mostrado poderosa, ela não provê ao usuário mecanismos para a inclusão de impurezas e falhas estruturais, naturalmente presentes em estruturas cristalinas. Elementos cristalográficos, como planos de simetria, também não são oferecidos ao usuário.

Essas melhorias devem ser precedidas por pesquisas que apontem outros possíveis

cenários e características das estruturas cristalinas não captados pela linguagem deste trabalho.

Interface com Simuladores: A incorporação das estruturas, geradas pela ferramenta apresentada neste trabalho, ao simulador MCSE foi feita manualmente (o código da ferramenta foi incorporado ao código do simulador). É necessária a elaboração e definição de um padrão que promova a comunicação entre a ferramenta e os mais variados simuladores;

Documentação detalhada da linguagem: A fim de difundir a utilização da ferramenta, é necessária a elaboração de uma extensa documentação, que apresente de maneira detalhada todos os elementos pertencentes a linguagem e um conjunto de estruturas geradas.

Construção de uma IDE — *Integrated development environment*: Uma IDE é um programa de computador que reúne características e ferramentas que apoiam desenvolvedores de *software* com o objetivo de agilizar este processo. No contexto do nosso trabalho a construção de uma IDE é de suma importância. Essa IDE deveria incorporar um editor de texto especializado, um visualizador prévio das estruturas geradas, uma documentação da linguagem e uma interface de comunicação com simuladores.

Paralelização do Módulo Interpretador: A tarefa de maior custo computacional na ferramenta é a interpretação do código fonte, apresentada na Seção 8.6. No momento em que uma a instrução *insert*, no programa principal, é interpretada, é feita a análise de todos os pontos do *lattice* a fim de definir e construir as células pertencentes ao objeto a ser inserido.

Do modo como a ferramenta foi implementada, a verificação é feita sequencialmente, um ponto analisado por vez. O mesmo conjunto de instruções (destinadas à análise da árvore que compõe o objeto) é executado inúmeras vezes, alterando-se apenas a coordenadas do ponto analisado. Tal característica (aplicação de uma instrução ou conjunto fixo de instruções à um conjunto variável de dados) pode ser implementada utilizando técnicas de paralelização. Uma implementação paralela agilizaria a geração das estruturas.

REFERÊNCIAS

[1] PEÇANHA, J. P., CAMPOS, A. M., , PAMPANELLI, P., LOBOSCO, M., VIEIRA, M. B., DE O. DANTAS, S., "Um modelo computacional para simulação de interação de spins em elementos e compostos magnéticos", *XI Encontro de Modelagem Computacional*, 2008.

[2] PEÇANHA, J. P., CAMPOS, A. M., DE ALMEIDA, R. B., PAMPANELLI, P., LOBOSCO, M., VIEIRA, M. B., DE O. DANTAS, S., "Simulação Computacional da Interação de Spins em Sistemas Magnéticos Usando Múltiplos Fluxos de Execução", *XII Encontro de Modelagem Computacional*, 2009.

[3] CAMPOS, A. M., PEÇANHA, J. P., PAMPANELLI, P., DE ALMEIDA, R. B., LOBOSCO, M., VIEIRA, M. B., DE O. DANTAS, S., "Parallel implementation of the heisenberg model using Monte Carlo on GPGPU". In: *Proceedings of the 2011 international conference on Computational science and its applications - Volume Part III*, *ICCSA'11*, pp. 654–667, Springer-Verlag: Berlin, Heidelberg, 2011.

[4] FERREIRA, R. C., *Linguagem para geração de objetos implícitos para simulação de spins*, Monografia de graduação, Departamento de Ciência da Computação, Universidade Federal de Juiz de Fora, Juiz de Fora, MG, Brasil, 2009.

[5] "Atoms", http://www.shapesoftware.com, Acessado em 01/07/2012.

[6] OZAWA, T. C., KANG, S. J., "*Balls&Sticks*: easy-to-use structure visualization and animation program", *Journal of Applied Crystallography*, v. 37, n. 4, pp. 679, Aug 2004.

[7] "Balls & Sticks", http://www.toycrate.org, Acessado em 01/07/2012.

[8] "CrystalMaker", http://www.crystalmaker.com, Acessado em 01/07/2012.

[9] FINGER, L. W., KROEKER, M., TOBY, B. H., "*DRAWxtl*, an open-source computer program to produce crystal structure drawings", *Journal of Applied Crystallography*, v. 40, n. 1, pp. 188–192, Feb 2007.

[10] "DRAWxtl", http://www.lwfinger.net/drawxtl, Acessado em 01/07/2012.

[11] CALLISTER, W., RETHWISCH, D., *Materials science and engineering: an introduction*. John Wiley & Sons: New York, NY, USA, 2009.

[12] MERCIER, J., ZAMBELLI, G., KURZ, W., *Introduction to materials science*. *Series in Applied Chemistry and Materials Sciences*, Elsevier: Paris, France, 2002.

[13] RICARTE, I., *Introdução a Compilação*. Elsevier: Rio de Janeiro, RJ, Brasil, 2008.

[14] MENEZES, P., *Linguagens Formais e Autômatos*. SAGRA-LUZZATTO: Porto Alegre, RS, Brasil, 2000.

[15] HOPCROFT, J., MOTWANI, R., ULLMAN, J., *Introduction to automata theory, languages, and computation*. Pearson/Addison Wesley: Boston, MA, USA, 2007.

[16] AHO, A., *Compilers: principles, techniques, & tools*. *Addison-Wesley series in computer science*, Pearson/Addison Wesley, 2007.

[17] LEE, K., *Programming Languages: An Active Learning Approach*. Springer: New York, NY, USA, 2008.

[18] VELHO, L., GOMES, J., DE FIGUEIREDO, L., *Implicit Objects in Computer Graphics*. Springer: New York, NY, USA, 2002.

[19] REQUICHA, A. G., "Representations for Rigid Solids: Theory, Methods, and Systems", *ACM Comput. Surv.*, v. 12, n. 4, pp. 437–464, Dec. 1980.

[20] GILBERT, E., JOHNSON, D., KEERTHI, S., "A fast procedure for computing the distance between complex objects in three-dimensional space", *Robotics and Automation, IEEE Journal of*, v. 4, n. 2, pp. 193 –203, apr 1988.

[21] VAN DEN BERGEN, G., "A fast and robust GJK implementation for collision detection of convex objects", *J. Graph. Tools*, v. 4, n. 2, pp. 7–25, March 1999.

[22] KOCKARA, S., HALIC, T., IQBAL, K., BAYRAK, C., ROWE, R., "Collision detection: A survey". In: *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*, pp. 4046 –4051, oct. 2007.

[23] ERICSON, C., *Real-Time Collision Detection*. N. v. 1, *Morgan Kaufmann Series in Interactive 3D Technology*, Elsevier, 2005.

[24] VAN DEN BERGEN, G., *Collision Detection in Interactive 3D Environments*. *The Morgan Kaufmann Series in Interactive 3D Technology*, Morgan Kaufman Publishers, 2004.

[25] ASCENCIO, A., *Estruturas de Dados: algoritmos, análise da complexidade e implementações em JAVA e C/C++*. PEARSON BRASIL: São Paulo, SP, Brasil, 2010.

[26] GAMMA, E., *Design Patterns: Elements of Reusable Object-Oriented Software*. *Addison-Wesley Professional Computing Series*, Addison-Wesley, 1995.

[27] BENTLEY, J. L., "Multidimensional binary search trees used for associative searching", *Commun. ACM*, v. 18, n. 9, pp. 509–517, Sept. 1975.

[28] FUCHS, H., ABRAM, G. D., GRANT, E. D., "Near real-time shaded display of rigid objects", *SIGGRAPH Comput. Graph.*, v. 17, n. 3, pp. 65–72, July 1983.

[29] GUTTMAN, A., "R-trees: a dynamic index structure for spatial searching", *SIGMOD Rec.*, v. 14, n. 2, pp. 47–57, June 1984.

[30] FINKEL, R. A., BENTLEY, J. L., "Quad trees a data structure for retrieval on composite keys", *Acta Informatica*, v. 4, pp. 1–9, 1974, 10.1007/BF00288933.

[31] NIEVERGELT, J., HINTERBERGER, H., SEVCIK, K. C., "The Grid File: An Adaptable, Symmetric Multikey File Structure", *ACM Trans. Database Syst.*, v. 9, n. 1, pp. 38–71, March 1984.

[32] GAEDE, V., GÜNTHER, O., "Multidimensional access methods", *ACM Comput. Surv.*, v. 30, n. 2, pp. 170–231, June 1998.

[33] ADEL'SON-VEL'SKII, G., LANDIS, E. M., "An algorithm for the organization of information", *Soviet Math. Doklady*, v. 16, n. 2, pp. 263–266, 1962.

[34] KITTEL, C., *Introduction To Solid State Physics*. Wiley, 2005.

[35] PATTERSON, J., BAILEY, B., *Solid-State Physics*. Springer, 2006.

[36] ASHCROFT, N., MERMIN, N., *Solid state physics*. *Science: Physics*, Saunders College, 1976.

[37] IBACH, H., LÜTH, H., *Solid-State Physics: An Introduction to Principles of Materials Science*. *Advanced Texts in Physics*, Springer, 2009.

[38] MISRA, P., *Physics of Condensed Matter*. *Academic Press*, Academic Press, 2010.

APÊNDICE A - Gramática Descrita em BNF

```
<program>::= <properties>
             <defStructure>
             <defBase> <defBases>
             <defSolids>
             "lattice" "(" <exp> "," <exp> "," <exp> "," <exp> "," <exp> ",
                " <exp> ")"
             "begin"
                 <stmt_List>
             "end"

<defStructure>::= "triclinic" "(" <vector> "," <vector> "," <vector> ")"
                | "rhombohedral" "(" <vector> "," <vector> "," <vector> ")"
                | "hexagonal" "(" <vector> "," <vector> "," <vector> ")"
                | "monoclinic" "(" <centering> "," <vector> "," <vector> ",
                   " <vector> ")"
                | "orthorhombic" "(" <centering> "," <vector> "," <vector>
                   "," <vector> ")"
                | "tetragonal" "(" <centering> "," <vector> "," <vector> ",
                   " <vector> ")"
                | "cubic" "(" <centering> "," <vector> "," <vector> "," <
                   vector> ")"

<vector>::= "(" <exp> "," <exp> "," <exp> ")"

<centering>::= "primitive"
             | "body"
             | "face"
             | "basea"
             | "baseb"
             | "basec"

<defBases>::= <defBase> <defBases>
```

```
            | EPSILON

<defBase>::= "base" "(" ID "," <vector> <propertyList> ")"

<propertyList>::= "," <propertyAssign> <propertyList>
                | EPSILON

<propertyAssign>::= ID = <value>

<properties>::= <property> <properties>
              | EPSILON

<property>::= "property" <type> ID = <value>

<type>::= "int"
        | "float"
        | "bool"

<value>::= <exp>
         | "true"
         | "false"

<defSolids>::= <defSolid> <defSolids>
             | EPSILON

<defSolid>::= "solid" ID <arguments> <exp> ";"
            | "composite" ID <arguments>
              "begin"
                  <stmt_List>
              "end"

<arguments>::= "(" <id_List> ")"

<id_List>::= ID <id_list_>
           | EPSILON

<id_list_>::= "," ID <id_list_>
            | EPSILON
```

```
<exp>::= <exp1> <exp_>

<exp_>::= "||" <exp1> <exp_>
        | EPSILON

<exp1>::= <exp2> <exp1_>

<exp1_>::= "&&" <exp2> <exp1_>
        | EPSILON

<exp2>::= <exp3> <exp2_>

<exp2_> ::= "==" <exp3> <exp2_>
         | "<>" <exp3> <exp2_>
         | EPSILON

<exp3>::= <exp4> <exp3_>

<exp3_>::= "<" <exp4> <exp3_>
        | "<=" <exp4> <exp3_>
        | ">=" <exp4> <exp3_>
        | ">" <exp4> <exp3_>
        | EPSILON

<exp4>::= <exp5> <exp4_>

<exp4_>::= "+" <exp5> <exp4_>
        | "-" <exp5> <exp4_>
        | EPSILON

<exp5>::= <exp6> <exp5_>

<exp5_>::= "*" <exp6> <exp5_>
        | "/" <exp6> <exp5_>
        | EPSILON

<exp6>::= "cos" "(" <exp> ")"
       | "sin" "(" <exp> ")"
       | "tan" "(" <exp> ")"
```

```
        | "acos" "(" <exp>")"
        | "asin" "(" <exp> ")"
        | "atan" "(" <exp> ")"
        | "atan2" "(" <exp> "," <exp> ")"
        | "cosh" "(" <exp> ")"
        | "sinh" "(" <exp> ")"
        | "tanh" "(" <exp> ")"
        | "exp" "(" <exp> ")"
        | "frexp" "(" <exp> "," <exp> ")"
        | "ldexp" "(" <exp> "," <exp> ")"
        | "log" "(" <exp> ")"
        | "log10" "(" <exp> ")"
        | "modf" "(" <exp> "," <exp> ")"
        | "pow" "(" <exp> "," <exp> ")"
        | "sqrt" "(" <exp> ")"
        | "ceil" "(" <exp> ")"
        | "fabs" "(" <exp> ")"
        | "floor" "(" <exp> ")"
        | "fmod" "(" <exp> "," <exp> ")"
        | <exp7>

<exp7>::= "union" "(" <exp> "," <exp> ")"
        | "intersection" "(" <exp> "," <exp> ")"
        | "difference" "(" <exp> "," <exp>  ")"
        | "not" "(" <exp> ")"
        | "rotatex" "(" <exp> "," <exp> ")"
        | "rotatey" "(" <exp> "," <exp> ")"
        | "rotatez" "(" <exp> "," <exp> ")"
        | "scale" "(" <exp> "," <exp> "," <exp> "," <exp> ")"
        | "shear" "(" <exp> "," <exp> "," <exp> "," <exp> ")"
        | "translate" "(" <exp> "," <exp> "," <exp> "," <exp> ")"
        | <exp8>

<exp8>::= "-" <exp9>
        | <exp9>

<exp9>::= "(" <exp> ")"
        | <exp10>
```

```
<exp10>::= NUM
         | ID <expProperty>
         | "x"
         | "y"
         | "z"

<expProperty>::= "(" <exp_List> ")"
               | EPSILON

<exp_List>::= <exp> <exp_List_>
            | EPSILON

<exp_List_>::= "," <exp> <exp_List_>
             | EPSILON

<stmt_List>::= <statement> ";" <stmt_List>
             | EPSILON

<statement>::= ID = <exp>
             | "insert" "(" <exp> <optionsInsert> ")"
             | "while" "(" <exp> ")"
               "begin"
                  <stmt_List>
               "end"

<optionsInsert>::= "," <opInsert> <rotate_List>
                 | EPSILON

<opInsert>::= "override"
            | "nooverride"
            | "error"

<rotate_List> ::= "," "(" <exp> "," <exp> ")" <rotate_List>
                | EPSILON
```

Na Gramática, os elementos *ID*, *NUM* e *EPSILON* representam respectivamente, um identificador, um valor numérico a e palavra vazia (ε).

APÊNDICE B - Mensagens de Erro reportadas pelo MGE

Tabela B.1: Mensagens de erros na abertura do arquivo com o código fonte.

TIPO DE ERRO	MENSAGEM EXIBIDA
Arquivo não encontrado	Arquivo passado como parâmetro não foi encontrado.
Arquivo com extensão inválida	Extensão inválida do arquivo passado como parâmetro.

Tabela B.2: Mensagens de erros léxicos.

CÓDIGO DO ERRO	MENSAGEM EXIBIDA
ERROR_INVALID_CARACTER	Erro léxico na linha xx, coluna xx: Caractere inválido.
ERROR_COMMENT	Erro léxico na linha xx, coluna xx: Fim inesperado do arquivo em comentário.
ERROR_OR	Erro léxico na linha xx, coluna xx: Esperado o caractere '|'.
ERROR_AND	Erro léxico na linha xx, coluna xx: Esperado o caractere '&'.

Tabela B.3: Mensagens de erros sintáticos .

CÓDIGO DO ERRO	MENSAGEM EXIBIDA
TK_BEGIN	Erro sintático na linha xx, coluna xx: Esperada a palavra reservada 'begin'.
TK_END	Erro sintático na linha xx, coluna xx: Esperada a palavra reservada 'end'.
TK_EOF	Erro sintático na linha xx, coluna xx: Trecho de programa após 'end' final.
TK_PROPERTY	Erro sintático na linha xx, coluna xx: Esperada a palavra reservada 'property'.
TK_ID	Erro sintático na linha xx, coluna xx: Esperado um identificador.

Continua na próxima página

Tabela B.3 – *Continua na página anterior*

CÓDIGO DO ERRO	**MENSAGEM EXIBIDA**
TK_ASSIGN	Erro sintático na linha xx, coluna xx: Esperado o sinal de atribuição '='.
TK_SEMICOLON	Erro sintático na linha xx, coluna xx: Esperado um ';'.
TK_LATTICE	Erro sintático na linha xx, coluna xx: Esperado a palavra reservada 'lattice'.
TK_OPARENT	Erro sintático na linha xx, coluna xx: Esperado um '('.
TK_COLON	Erro sintático na linha xx, coluna xx: Esperada uma ','.
TK_CPARENT	Erro sintático na linha xx, coluna xx: Esperado um ')'.
TK_BASE	Erro sintático na linha xx, coluna xx: Esperada a palavra reservada 'base'.
ERR_NOT_VALUE	Erro sintático na linha xx, coluna xx: Esperado um valor (número, true ou false).
ERR_NOT_VALUE_INSERT	Erro sintático na linha xx, coluna xx: Esperado um valor (identificador, número, true ou false).
ERR_INVALID_TYPE	Erro sintático na linha xx, coluna xx: Esperado um tipo ('int', 'float' ou 'bool').
ERR_NOT_OPERATION	Erro sintático na linha xx, coluna xx: Esperada uma expressão (número, identificador ou um eixo).
ERR_NOT_STMT	Erro sintático na linha xx, coluna xx: Esperado um início de instrução (identificador, 'insert' ou 'while').
ERR_INVALID_STRUCTURE	Erro sintático na linha xx, coluna xx: Esperado um sistema cristalino de Bravais ('triclinic', 'monoclinic', 'orthorhombic', 'tetragonal', 'rhombohedral' , 'hexagonal' ou 'cubic').

Continua na próxima página

Tabela B.3 – *Continua na página anterior*

CÓDIGO DO ERRO	MENSAGEM EXIBIDA
ERR_INVALID_AXIS	Erro sintático na linha xx, coluna xx: Esperado um eixo (‘x’,‘y’ ou ‘z’).
ERR_NOT_OPTIONINSERT	Erro sintático na linha xx, coluna xx: Esperada uma opção de inserção (‘override’, ‘nooverride’, ‘error’).

Tabela B.4: Mensagens de erros semânticos .

MENSAGENS EXIBIDAS
Erro semântico na linha xx: Estrutura declarada previamente.
Erro semântico na linha xx: não foi atribuído nenhum valor ao identificador, tipo indefinido (O identificador não e uma variável ou sólido).
Erro semântico na linha xx: A coordenada ‘x’ só pode aparecer na expressão que define um sólido.
Erro semântico na linha xx: A coordenada ‘y’ só pode aparecer na expressão que define um sólido.
Erro semântico na linha xx: A coordenada ‘z’ só pode aparecer na expressão que define um sólido.
Erro semântico na linha xx: Tipos incompatíveis na operação.
Erro semântico na linha xx: solid ou composite não declarado.
Erro semântico na linha xx: Lista de parâmetros incorreta.
Erro semântico na linha xx: Somente números podem ser passados para a construção de sólidos.
Erro semântico na linha xx: Falta de parâmetros na chamada da função.
Erro semântico na linha xx: Excesso de parâmetros na chamada da função.
Erro semântico na linha xx: Existe uma definição de sólido declarada com o mesmo lexema do id.
Erro semântico na linha xx: Existe uma propriedade declarada com o mesmo lexema do id.

Continua na próxima página

Tabela B.4 – *Continua na página anterior*

MENSAGENS EXIBIDAS
Erro semântico na linha xx: Propriedade inexistente.
Erro semântico na linha xx: Propriedade do tipo bool não pode receber um valor numérico.
Erro semântico na linha xx: Propriedade do tipo int ou float não pode receber um valor booleano.
Erro semântico na linha xx: Propriedade do tipo int não pode receber um valor float.
Erro semântico na linha xx: Condição inválida para a instrução 'insert'. Expressão a ser inserida não é um sólido.
Erro semântico na linha xx: 'insert' dentro de 'composite' não pode conter opção de inserção.
Erro semântico na linha xx: 'insert' no programa principal deve conter uma opção de inserção - 'OVERRIDE', 'NOOVERRIDE' ou 'ERROR'.
Erro semântico na linha xx: 'insert' dentro de 'composite' não pode efetuar rotações.
Erro semântico na linha xx: Condição inválida para a instrução WHILE. O resultado da expressão deve ser booleano.
Erro semântico na linha xx: Existe um solid ou composite com o mesmo nome declarado no programa.
Erro semântico na linha xx: A expressão definida para o sólido deve resultar em um valor booleano.
Erro semântico na linha xx: Propriedade declarada como tipo Booleano mas valor numérico atribuído.
Erro semântico na linha xx: Propriedade declarada como tipo Inteiro mas valor numérico atribuído corresponde a um Float.
Erro semântico na linha xx: No lattice, Xmin não é um valor numérico.
Erro semântico na linha xx: No lattice, Ymin não é um valor numérico.
Erro semântico na linha xx: No lattice, Zmin não é um valor numérico.
Erro semântico na linha xx: No lattice, Xmax não é um valor numérico.
Erro semântico na linha xx: No lattice, Ymax não é um valor numérico.
Erro semântico na linha xx: No lattice, Zmax não é um valor numérico.

Continua na próxima página

Tabela B.4 – *Continua na página anterior*

MENSAGENS EXIBIDAS
Erro semântico na linha xx: Distribuição de átomos incorreta para a geometria MONOCLINIC.
Erro semântico na linha xx: Distribuição de átomos incorreta para a geometria TETRAGONAL.
Erro semântico na linha xx: Distribuição de átomos incorreta para a geometria CUBIC.
Erro semântico na linha xx: Valor não numérico na coordenada x do vetor.
Erro semântico na linha xx: Valor não numérico na coordenada y do vetor.
Erro semântico na linha xx: Valor não numérico na coordenada z do vetor.
Erro semântico na linha xx: Expressão que define a rotação da célula unitária deve conter um valor numérico.

Tabela B.5: Mensagens de erros na interpretação .

MENSAGENS EXIBIDAS
Erro na interpretação da linha xx: Xmin maior que Xmax no lattice.
Erro na interpretação da linha xx: Ymin maior que Zmax no lattice.
Erro na interpretação da linha xx: Zmin maior que Zmax no lattice.
Erro na interpretação da linha xx: Variável correspondente ao objeto não foi encontrada.
Erro na interpretação da linha xx: Parâmetro negativo passado para função log.
Erro na interpretação da linha xx: Parâmetro negativo passado para função log10.
Erro na interpretação da linha xx: Expressão 1 igual a 0 e Expressão 2 menor que 0 na função pow.
Erro na interpretação da linha xx: Parâmetro negativo passado para função sqrt.
Erro na interpretação da linha xx: Divisão por zero.
Erro na interpretação da linha xx: Primeiro vetor da base é nulo.
Erro na interpretação da linha xx: Segundo vetor da base é nulo.
Erro na interpretação da linha xx: Terceiro vetor da base é nulo.

Continua na próxima página

Tabela B.5 – *Continua na página anterior*

MENSAGENS EXIBIDAS
Erro na interpretação da linha xx: Vetores da base não são Linearmente Independentes.
Erro na interpretação da linha xx: Vetores com mesmo comprimento em Sistema Cristalino 'TRICLINIC'.
Erro na interpretação da linha xx: Ângulos entre os vetores com o mesmo valor em Sistema Cristalino 'TRICLINIC'.
Erro na interpretação da linha xx: Vetores com comprimento diferente em Sistema Cristalino 'RHOMBOHEDRAL'.
Erro na interpretação da linha xx: Ângulos entre os vetores com valor diferente em Sistema Cristalino 'RHOMBOHEDRAL'.
Erro na interpretação da linha xx: Ângulo valendo 90 graus em Sistema Cristalino 'RHOMBOHEDRAL'.
Erro na interpretação da linha xx: Ângulo valendo mais de 120 graus em Sistema Cristalino 'RHOMBOHEDRAL'.
Erro na interpretação da linha xx: |vetor1| diferente de |vetor2| em Sistema Cristalino 'HEXAGONAL'.
Erro na interpretação da linha xx: |vetor3| igual a |vetor1| ou |vetor2| em Sistema Cristalino 'HEXAGONAL'.
Erro na interpretação da linha xx: Ângulo gama ou beta diferente de 90 graus em Sistema Cristalino 'HEXAGONAL'.
Erro na interpretação da linha xx: Ângulo alfa diferente de 120 graus em Sistema Cristalino 'HEXAGONAL'.
Erro na interpretação da linha xx: Vetores com mesmo comprimento em Sistema Cristalino 'MONOCLINIC'.
Erro na interpretação da linha xx: Ângulo gama ou alfa diferente de 90 graus em Sistema Cristalino 'MONOCLINIC'.
Erro na interpretação da linha xx: Ângulo beta igual a 90 graus em Sistema Cristalino 'MONOCLINIC'.
Erro na interpretação da linha xx: Vetores com mesmo comprimento em Sistema Cristalino 'ORTHORHOMBIC'.

Continua na próxima página

Tabela B.5 – *Continua na página anterior*

MENSAGENS EXIBIDAS
Erro na interpretação da linha xx: Ângulos com valor diferente de 90 graus em Sistema Cristalino 'ORTHORHOMBIC'.
Erro na interpretação da linha xx: |vetor1| diferente de |vetor2| em Sistema Cristalino 'TETRAGONAL'.
Erro na interpretação da linha xx: |vetor3| igual a |vetor1| ou |vetor2| em Sistema Cristalino 'TETRAGONAL'.
Erro na interpretação da linha xx: Ângulos com valor diferente de 90 graus em Sistema Cristalino 'TETRAGONAL'.
Erro na interpretação da linha xx: Vetores com comprimentos diferentes em Sistema Cristalino 'CUBIC'.
Erro na interpretação da linha xx: Ângulos com valor diferente de 90 graus em Sistema Cristalino 'CUBIC'.
Erro na interpretação da linha xx: Vetores com comprimentos diferentes em Sistema Cristalino 'CUBIC'.
Erro na interpretação da linha xx: Ângulos com valor diferente de 90 graus em Sistema Cristalino 'CUBIC'.