LINGUAGEM PARA GERAÇÃO DE OBJETOS IMPLÍCITOS PARA SIMULAÇÃO DE SPINS

Roberto de Carvalho Ferreira

Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharel em Ciência da Computação

Orientador: Marcelo Bernardes Vieira

Juiz de Fora, MG
Julho de 2009

LINGUAGEM PARA GERAÇÃO DE OBJETOS IMPLÍCITOS PARA SIMULAÇÃO DE SPINS

Roberto de Carvalho Ferreira

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada pela banca constituída pelos seguintes membros:

_____

Marcelo Bernardes Vieira – orientador
D.Sc. em Ciência da Computação, ENSEA-UCP/France, 2002

_____

Marcelo Lobosco
D. em Eng. de Sistemas e Computação, UFRJ/Brasil, 2005

_____

Sócrates de Oliveira Dantas
Doutor em Física, UNICAMP/Brasil, 1994

Juiz de Fora, MG
Julho de 2009

Agradecimentos

A minha família, por ter me possibilitado toda a infra-estrutura necessária para estudar.
Aos meus professores, por terem me auxiliado na difícil tarefa do aprendizado.
A Ana Cláudia, por ter me auxiliado na construção e detecção de erros no compilador implementado para a análise da linguagem proposta por este trabalho.

Sumário

Lista de Figuras.....vi

Lista de Tabelas.....vi

Resumo..... viii

Capítulo 1 9

Introdução..... 9

1.1 Definição do problema..... 9

1.2 Objetivos 9

Capítulo 2 11

Fundamentos.....11

2.1 Definições Linguagens e Gramáticas 11

2.2 Estrutura Geral de Compiladores 13

2.2.1 Fase Analítica 14

2.2.2 Fase Sintética 17

2.3 Representação de Objetos 17

2.3.1 Combinando Objetos Implícitos 19

2.3.2 Operações booleanas 20

2.3.3 Descrição funcional.....20

2.3.4 Objetos implícitos gerados a partir de CSG 21

Capítulo 3 23

Definição da Linguagem 23

Capítulo 4 32

Descrição do Compilador.....32

4.1 Gerenciador de Caracteres 33

4.2 Analisador Léxico e Sintático 34

4.3 Analisador Semântico.....37

4.4 Gerenciador de Erros 38

Capítulo 5 40

Resultados 40

5.1 Esfera Maciça.....40
5.2 Cilindro Maciço.....42
5.3 Composição definida pela diferença entre o Cilindro e a Esfera.....43
Capítulo 6.....46
Conclusão.....46
Referências Bibliográficas.....47

Lista de Figuras

Figura 1 – Definição de um compilador 13

Figura 2 – Árvore Gramatical.....15

Figura 3 – Análise semântica de uma Árvore Gramatical 16

Figura 4 – Representação paramétrica do círculo de raio unitário [Velho, 2002].....18

Figura 5 - Representação implícita do círculo de raio unitário [Velho, 2002].....18

Figura 6 - Operações CSG [Velho, 2002]. 21

Figura 7 - Representação de geração de um objeto através de CSG [Velho, 2002]. 22

Figura 8 - Gramática Criada 26

Figura 9 - Conjunto First e Follow 28

Figura 10 – Diagrama Estrutural do Compilador 32

Figura 11 - Estruturas de armazenamento de informações 34

Figura 12 - Autômato que representa todas as expressões regulares da gramática 35

Figura 13 - Código de criação de uma esfera 40

Figura 14 - Impressão da ASA 41

Figura 15 - Código para a criação de um cilindro 42

Figura 16 - ASA gerada a partir do código do cilindro 43

Figura 17 - Cilindro menos Esfera.....44

Figura 18 - ASA referente ao Cilindro menos a Esfera 45

Lista de Tabelas

Tabela 1 - Expressões Regulares referentes às palavras da Linguagem de geração de objetos implícitos. 29

Tabela 2 - Estrutura da ASA 37

Tabela 3 - Mensagens de Erro 39

Resumo

Este trabalho expõe uma linguagem de apoio à construção de objetos implícitos para simulação de spins. São apresentadas as definições de linguagens e gramáticas e são relatadas as características que uma linguagem de programação deve atender para fornecer, de maneira rápida e precisa, objetos bem formados para simuladores físicos ou químicos. Foi construído um compilador que valida e constrói uma representação intermediária para a linguagem. A construção e implementação do compilador estão incorporadas a este trabalho. Experimentos realizados pela compilação de código de primitivos básicos mostram o potencial de aplicabilidade da linguagem proposta.

Palavras-Chave: Objetos implícitos, Compiladores, Simulação

Capítulo 1

Introdução

Os estudos de fenômenos físicos e químicos estão sempre relacionados a experimentos e observações. O uso de simuladores vem sendo cada vez mais disseminado no meio acadêmico, tendo em vista que experimentos e observações envolvendo estruturas reais muitas vezes são custosos, demorados e não são obtidos de maneira trivial. Algumas vezes experimentos com objetos reais são impossíveis de ser feitos, como na área de nanotecnologia, por exemplo.

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 as condições de estudo podem ser mudadas de maneira simples e rápida.

Muitos simuladores são construídos e implementados usando estruturas peculiares ao estudo ou mais simples de modo que o enfoque maior seja dado aos fenômenos estudados e não na estrutura na qual a simulação ocorreria.

Este trabalho apresenta uma linguagem e um compilador-interpretador que auxilia a criação de estruturas tridimensionais implícitas para utilização em simuladores.

1.1 Definição do problema

O problema tratado nesta monografia é o de prover meios computacionais para o a definição e cálculo da função característica de objetos sólidos ou, conjunto de objetos, para simuladores científicos.

1.2 Objetivos

O objetivo primário deste trabalho é o desenvolvimento de uma linguagem e um compilador-interpretador para definição de estruturas tridimensionais implícitas. Como objetivos secundários oriundos do objetivo primário temos:

- apresentação da teoria envolvida na construção de compiladores;
- explicitar os conceitos matemáticos relacionados a definição de objetos implícitos;
- expor os resultados obtidos e propor possíveis melhorias;
- tendo em vista os principais usuários da linguagem – físicos e químicos – a linguagem deve ser sucinta, de fácil compreensão e possibilitar a criação dos mais variados formatos de estruturas.

Capítulo 2

Fundamentos

2.1 Definições Linguagens e Gramáticas

Para o entendimento de linguagens e gramáticas são necessárias algumas definições [Menezes, 2000].

Alfabeto: Conjunto finito de símbolos. Um símbolo é uma entidade abstrata básica a qual não é definida formalmente. Letras e dígitos são exemplos de símbolos.

Exemplo: $\sum = \{a, b\}$, alfabeto que contém os símbolos a e b.

Palavra, Cadeia de Caracteres ou Sentença: Sequência finita de símbolos (pertencentes a um alfabeto) justapostos. A palavra vazia, representada pelo símbolo ε, é uma palavra sem símbolo. O comprimento de uma palavra (quantidade de símbolos que a compõe) w é dado por $|w|$. Se $\sum$ representa um alfabeto, $\sum^*$ equivale ao conjunto de todas as palavras possíveis sobre $\sum$ e $\sum^+$ representa $\sum^* - \{\varepsilon\}$.

Exemplo: $abba$ é uma palavra sobre o alfabeto $\{a, b\}$ e seu comprimento – $|abba|$ – equivale a 4.

Prefixo: Qualquer sequência de símbolos inicial de uma palavra.

Exemplo: aba é um prefixo da palavra $abacate$.

Sufixo: Qualquer sequência de símbolos inicial de uma palavra.

Exemplo: $cate$ é um sufixo da palavra $abacate$.

Sub-palavra: Qualquer sequência contígua de símbolos de uma palavra.

Exemplo: $bacat$ é uma sub-palavra da palavra $abacate$.

Linguagem Formal: Conjunto de palavras sobre um alfabeto.

Exemplo: O conjunto $\{a, aa, ab, aaa, aba, abb, \cdots\}$ denota a linguagem que define todas as palavras que começam com o símbolo a sobre o alfabeto $\sum = \{a, b\}$

Concatenação: Se trata de uma operação binária sobre uma linguagem que resulta na justaposição dos símbolos de um par de palavras. A operação de concatenação satisfaz às seguintes propriedades: Associatividade – $a(bc) = (ab)c$, Elemento Neutro à Esquerda e à Direita – $\varepsilon a = a = a\varepsilon$. (supondo que a, b, c são palavras)

Exemplo: aba concatenado com $cate$ resulta em $abacate$.

Concatenação Sucessiva: É a concatenação de uma palavra com ela mesma um número estipulado de vezes. É representada pela forma w^n, onde w é uma palavra e n é o número de concatenações sucessivas.

Exemplo: $(ab)^3 = ababab$, $(ab)^0 = \varepsilon$, $\varepsilon^n = \varepsilon$ para $n > 0$, ε^0 é indefinida

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

- V conjunto finito de símbolos variáveis ou não-terminais;
- T conjunto finito de símbolos terminais disjuntos de V ;
- P 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 de $(V \cup T)^*$;
- S 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 $\alpha \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$

Linguagem Gerada: Seja $G = (V, T, P, S)$ uma gramática. A união de todas as palavras de 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 $L(G) = \{w \in T^* \mid S \Rightarrow^+ w\}$.

Exemplo: Dado a gramática $G = (V, T, P, S)$ onde:

- $V = \{S, V\}$
- $T = \{a, b\}$
- $P = \{S \rightarrow aV, V \rightarrow aV \mid bV \mid \varepsilon\}$

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

2.2 Estrutura Geral de Compiladores

Compilador é um programa que lê um programa escrito em uma linguagem, chamada linguagem fonte, e o traduz em um programa equivalente em uma outra linguagem, chamada linguagem alvo [Aho, 1986]. No processo de tradução são informados, ao usuário, os erros encontrados no programa fonte. Essa definição é representada pela Figura 1.

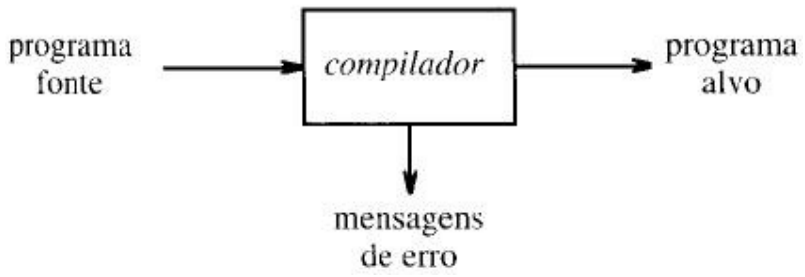

Figura 1 – Definição de um compilador

Existem duas partes na compilação: a análise e a síntese. Na análise o programa, escrito na linguagem fonte, são verificados os erros léxicos, sintáticos e semânticos e há a construção de uma representação intermediaria do programa. Na síntese a representação intermediaria obtida é convertida na linguagem alvo.

2.2.1 Fase Analítica

A seguir serão detalhadas as fases da análise e as características dos erros que cada uma reporta.

- **Análise Léxica**

O conjunto de caracteres que constituem o programa é lido da esquerda para a direita, sendo agrupados em *tokens* – sequência de caracteres que têm um significado coletivo.

A análise léxica tem como objetivo analisar todos os caracteres pertencentes ao programa retirando trechos de código – comentários, espaços em branco e caracteres especiais (tabulações e quebras de linha) – que não serão usados na geração da linguagem alvo reportando possíveis erros léxicos. Erros léxicos podem ser traduzidos como utilização de caracteres não pertencentes à linguagem fonte, malformação de lexemas, e alcance do fim de arquivo inesperado.

O agrupamento dos caracteres em *tokens* é regido por um conjunto de expressões regulares que denotam linguagem das palavras aceitas pela linguagem fonte.

O *token* ID, que representa um identificador, é definido como sendo uma letra seguida de mais letras ou dígitos. ID pode ser definido a partir das seguintes expressões:

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

$$digito \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$ID \rightarrow letra(letra \mid digito)*$$

Os caracteres “valor = sin(angulo) * 2” interpretados pelo analisador léxico do compilador apresentado nesta monografia seriam agrupados nos seguintes tokens:

- O identificador “valor”.
- O símbolo de atribuição “=”.
- O símbolo do operador seno “sin”.
- O símbolo abre parêntese “(“.

- O identificador “angulo”.
- O símbolo fecha parêntese “)”.
- O sinal de multiplicação “*”.
- O número “2”.

➢ **Análise Sintática**

Os *tokens* obtidos na analise léxica são organizados hierarquicamente, geralmente essa organização hierárquica é representada por uma árvore denominada árvore gramatical.

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. Os erros gerados nessa fase são referentes a associações incompatíveis com a linguagem.

Os *tokens* gerados a partir dos caracteres “valor = sin(angulo) * 2” interpretados pelo analisador semântico do compilador apresentado nesta monografia gerariam a árvore gramatical exposta na Figura 2:

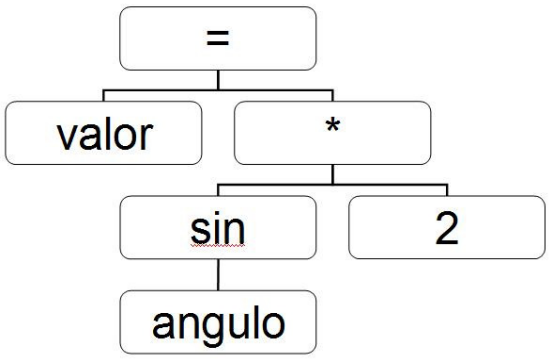

Figura 2 – Árvore Gramatical

➢ **Análise Semântica**

Percorre a estrutura montada na análise sintática verificando erros entre a associação entre os *tokens* e seus respectivos tipos.

Cada nó da árvore 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.

O que indica a corretude de uma determinada relação entre os tipos são regras semânticas empregadas na linguagem. Não cabe esta seção a exposição das regras definidas para a linguagem de geração de objetos implícitos contida neste trabalho. As regras serão expostas no Capítulo 3 onde serão expostas todas as definições da linguagem. Por hora, será apresentado como o analisador semântico interpretaria a árvore criada partir dos caracteres "valor = sin(angulo) * 2" . A ordem de varredura é especificada na Figura 3.

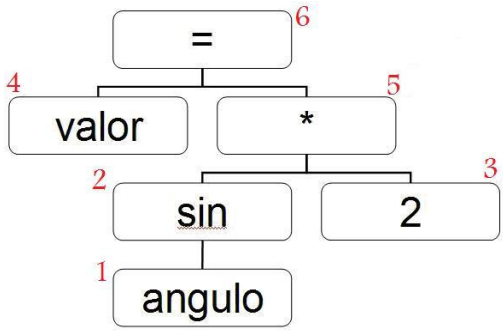

Figura 3 – Análise semântica de uma Árvore Gramatical

1. Verificar o tipo do identificador "angulo";
2. Verificar se a operação seno, representada pelo *token* "sin" pode ser efetuada com o tipo retornado em 1;
3. Retornaria o tipo definido na linguagem para o número 2. No caso *number*;
4. Verificar o tipo do identificador "valor";
5. Verificar se a multiplicação pode ser efetuada entres os tipos retornados por 2 e 3;
6. Verificar se a atribuição pode ser feita entre os tipo retornados por 4 e 5;

2.2.2 Fase Sintética

Na fase analítica há uma verificação que depende apenas da linguagem fonte. Na fase sintética o código é transformado na linguagem alvo, por tradução ou interpretação.

É iniciada na análise sintática, onde pode ser construída uma representação intermediaria do código denominado Árvore de Sintaxe Abstrata – ASA.

A ASA, depois da analise semântica, deve estar desprovida de qualquer tipo de erro e representar fielmente o código, escrito na linguagem fonte. ASA com algum tipo de erro não são transformadas na linguagem alvo.

Após a análise semântica a ASA é sintetizada na linguagem alvo. A síntese pode ser feita de uma maneira direta (a linguagem alvo é gerada a partir da ASA) ou em partes (a partir da ASA é gerado outra representação intermediária. Essa representação passa por um otimizador e a linguagem alvo é gerada).

A linguagem alvo deste trabalho deve ser facilmente convertida em uma matriz tridimensional que representa um espaço que contém as estruturas definidas na linguagem fonte.

2.3 Representação de Objetos

A geometria de um objeto (superfície ou sólido) pode ser descrita de duas formas diferentes – utilizando um modelo paramétrico ou um 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 função:

$$f(\theta) = (\cos\theta, \sin\theta), \theta \in [0, 2\pi] \quad (2.1)$$

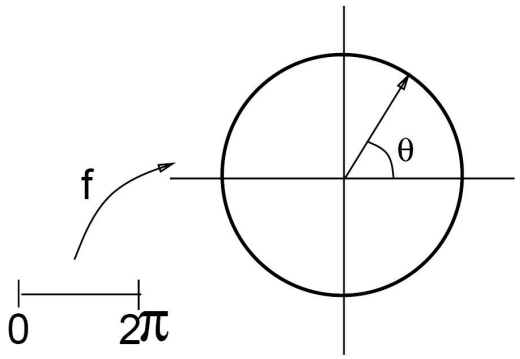

Figura 4 – Representação paramétrica do círculo de raio unitário [Velho, 2002]

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 $O = \{ p \in U : f(p) \in V \}$ [Velho, 2002].

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

$$f(x, y) = x^2 + y^2 - 1, x, y \in \Re \quad (2.2)$$

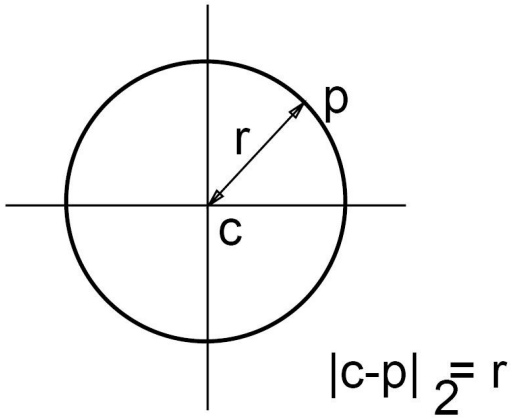

Figura 5 - Representação implícita do círculo de raio unitário [Velho, 2002]

O conjunto de pontos (x, y) que satisfazem $f(x, y) = 0$ equivalem ao círculo. A função f classifica os pontos do plano em relação à função 2.2. Quando substituir as coordenadas de um ponto $p = (x, y)$ na equação $f(x, y) = x^2 + y^2 - 1$ o sinal do valor de $f(p)$ indica a posição de p em relação ao objeto círculo definido pela mesma.

- p pertencente ao interior do circulo caso $f(p) < 0$;
- p pertencente ao círculo caso $f(p) = 0$;
- p exterior ao círculo caso $f(p) > 0$;

Vale salientar que não se pode afirmar que uma representação é melhor que a outra, tudo depende da finalidade, contexto e dos tipos de operações que serão empregadas.

Para desenhar uma aproximação de um círculo precisamos ligar linhas, retas, passando pelos pontos que compões sua borda. Isto pode ser feito facilmente usando a forma paramétrica, variando o valor de θ pelo intervalo $[0,2\pi]$, obtemos rapidamente os pontos pelos quais as linhas devem passar. Em contra partida, para determinarmos, por exemplo, uniões entre objetos precisaram testar se os pontos internos de um objeto também são internos no outro. Por outro lado, esta é uma operação simples usando a forma implícita. Basta verificar o sinal de $f(p)$.

2.3.1 Combinando Objetos Implícitos

Objetos implícitos podem se juntar para formar uma composição de objetos implícitos.

Isto é realizado através da combinação, de acordo com determinadas regras, descrevendo as funções correspondentes de um objeto individual. Operações algébricas com um conjunto de funções implícitas produzem outra função correspondente à composição das operações com os objetos implícitos.

Uma função $f : \Re^n \to \Re$ associada a um objeto implícito O e um valor regular A é dita válida 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 válidas. f^{β} seria uma função válida somente quando f for definida para valores positivos.

Se $f1$ e $f2$ são funções válidas então qualquer combinação delas com qualquer uma destas operações abaixo gera outra função fx que é válida:

- Soma: $f1 + f2$;
- Produto: $f1 * f2$;
- Composição: $f(f1, \cdots, fk)$;
- Máximo: $Max(f1, f2)$;
- Mínimo: $Min(f1, f2)$;

Vale salientar que as operações unárias, αf e f^{α} – com α inteiro e positivo – são casos particulares podendo ser expressas utilizando as operações acima.

2.3.2 Operações booleanas

Um método eficaz para 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 [Requicha, 1980].

2.3.3 Descrição funcional

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

- $Pi \subset Sj$;
- $F \in Sj \Rightarrow -F \in Sj$;

- $F1, F2 \in Sj \Rightarrow Max(F1, F2) \in Sj$;
- $F1, F2 \in Sj \Rightarrow Min(F1, F2) \in Sj$.

onde $i = 1, \cdots, k$ e $j = 1, \cdots, l$.

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

2.3.4 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 satisfaçam $F(x) \leq 0$ para algum $F \in Sj$. As operações booleanas são definidas como:

- $F1 \cup F2 = Min(F1, F2)$;
- $F1 \cap F2 = Max(F1, F2)$;
- $F1 \setminus F2 = F1 \cap \overline{F2} = Max(F1, -F2)$;

A Figura 6 representa as operações descritas acima

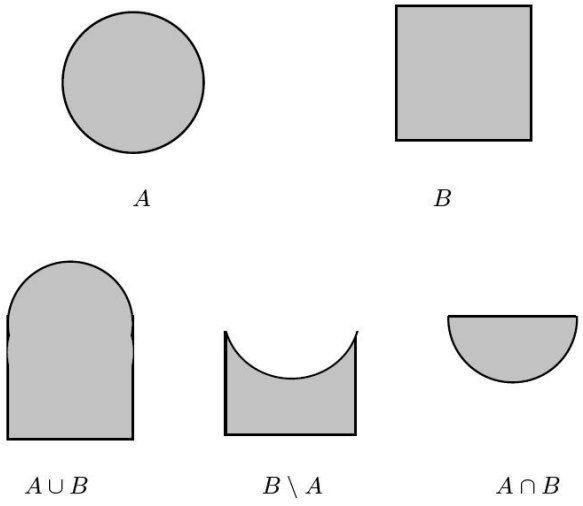

Figura 6 - Operações CSG [Velho, 2002].

Usualmente um objeto criado a partir de CSG pode ser representado por uma árvore (Figura 7) cujo nós internos representam as operações booleanas e as folhas representam os objetos primitivos utilizado. A raiz é o objeto gerado.

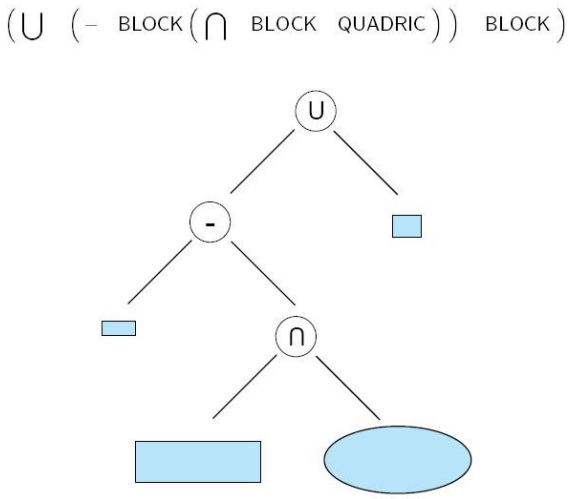

Figura 7 - Representação de geração de um objeto através de CSG [Velho, 2002].

Capítulo 3

Definição da Linguagem

Podemos definir linguagem como sendo um conjunto organizado, 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 e por quais meios, instruções, esse resultado seria obtido.

No estudo em questão verificou-se a necessidade da construção de uma linguagem que atendesse as seguintes especificações:

- Possibilitar a definição de um sólido através da união ou intersecção de uma ou mais funções características;
- Possibilitar a definição de um sólido através da composição das definições de outros sólidos já definidos;
- Realizar operações e transformações com sólidos já definidos;
- Passar parâmetros para as funções características que definem os objetos;
- Efetuar operações e funções com números;
- Prover uma estrutura de laço condicional;
- Possibilitar, ao final de sua computação, a produção de uma estrutura matricial tridimensional, definida pelo usuário, onde cada elemento dessa matriz representa uma coordenada (x, y, z) e carrega consigo possíveis propriedades – definidas também na linguagem – de um objeto da qual faça parte.
- Não ser ambígua. O resultado da computação deve gerar sempre a mesma estrutura. Dado que o código não foi alterado.

- Ser LL(1). Possibilitando assim sua análise por um analisador sintático *top-down* sem retrocesso – analisador sintático preditivo com o uso de 1 *token lookaread* [Aho, 1986].

Diante das especificações a seguinte gramática LL(1) foi criada:

Program	→	DefSolids SpinProperties **lattice** "(" Exp "**,**" Exp "**,**" Exp "**,**" Exp "**,**" Exp "**,**" Exp "**,**" Exp "**,**" Exp "**,**" Exp "**)**" **begin** Stmt_List **end**
SpinProperties	→	SpinProperty SpinProperties | Epsilon
SpinProperty	→	**spin property** Type **ID** "**=**" Value "**;**"
Type	→	**int** | **float** | **bool**
Value	→	**NUM** | **true** | **false**
DefSolids	→	DefSolid DefSolids | Epsilon
DefSolid	→	**solid ID** Arguments Exp "**;**" | **composite ID** Arguments **begin** Stmt_List **end**
Arguments	→	"(" Id_List ")" | Epsilon
Id_List	→	ID Id_list_
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 PropertyList ")" |
		while "(" Exp ")"
		begin
		Stmt_List
		end
PropertyList	→	"**,**" PropertyAssign PropertyList |
		Epsilon
PropertyAssign	→	**ID** "**=**" Value

Figura 8 - Gramática Criada

Uma gramática $G = (V, T, P, S)$ é LL(1) se e somente se:

$A \to \alpha \mid \beta \Rightarrow^{*} t$

1. $First(\alpha) \cap First(\beta) = \{\}$; ($\alpha$ e β não derivem ao mesmo tempo, cadeias começando pelo mesmo terminal a.
2. $\alpha \Rightarrow^{*} \varepsilon$ implica que $\beta \Rightarrow^{*} \varepsilon$ é falso; (Somente α ou β derivem cadeia a vazia)
3. $\alpha \Rightarrow^{*} \varepsilon$ implica que $First(\beta) \cap Follow(\alpha) = \{\}$; (Se α deriva cadeia vazia então a interseção entre $First(\beta)$ e o $Follow(\alpha)$ deve ser nula)

[Aho, 1986]

Para verificar se a que a gramática construída é LL(1) foi utilizado o conjunto *First* e *Follow* representado pela Figura 9.

CONJUNTO FIRST	
FIRST(Program)	= {**SOLID, COMPOSITE, SPIN, LATTICE**}
FIRST(SpinProperties)	= {**SPIN,** Epsilon}
FIRST(SpinProperty)	= {**SPIN**}
FIRST(Type)	= {**INT, FLOAT, BOOL**}
FIRST(Value)	= {**NUM, TRUE, FAlSE**}
FIRST(DefSolids)	= {**SOLID, COMPOSITE,** Epsilon}
FIRST(DefSolid)	= {**SOLID, COMPOSITE**}
FIRST(Arguments)	= {**"("**, Epsilon}

FIRST(Id_List) = {**ID**}
FIRST(Id_list_) = {**“,”**, Epsilon}
FIRST(Exp) = {**“-”, “(”, NUM, ID, x, y, z, 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**}
FIRST(Exp_) = {**“||”,** Epsilon }
FIRST(Exp1) = {**“-”, “(”, NUM, ID, x, y, z, 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**}
FIRST(Exp1_) = {**“&&”**, Epsilon }
FIRST(Exp2) = {**“-”, “(”, NUM, ID, x, y, z, 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**}
FIRST(Exp2_) = {**“==”, “<>”**, Epsilon }
FIRST(Exp3) = {**“-”, “(”, NUM, ID, x, y, z, 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**}
FIRST(Exp3_) = {**“<”, “<=”, “>=”, “>”,** Epsilon }
FIRST(Exp4) = {**“-”, “(”, NUM, ID, x, y, z, 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**}
FIRST(Exp4_) = {**“+”**, “**-**”, Epsilon }
FIRST(Exp5) = {**“-”, “(”, NUM, ID, x, y, z, 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**}
FIRST(Exp5_) = {**“*”, “/”**, Epsilon }
FIRST(Exp6) = {**“-”, “(”, NUM, ID, x, y, z, 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**}
FIRST(Exp7) = {**“-”, “(”, NUM, ID, x, y, z, union, intersection, difference, not, rotatex, rotatey, rotatez, scale, shear, translate**}
FIRST(Exp8) = {**“-”, “(”, NUM, ID, x, y, z**}
FIRST(Exp9) = {**“(”, NUM, ID, x, y, z**}
FIRST(Exp10) = {**NUM, ID, x, y, z**}
FIRST(ExpProperty) = {**“(”,** Epsilon}
FIRST(Expr_list) = {**“-”, “(”, NUM, ID, x, y, z, 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,** Epsilon }
FIRST(Expr_list_) = {**“,” ,** Epsilon}
FIRST(Stmt_List) = {**ID, INSERT, FOR,** Epsilon}
FIRST(Statement) = {**ID, INSERT, FOR**}
FIRST(PropertyList) = {**ID, NUM, TRUE, FALSE,** Epsilon}

CONJUNTO FOLLOW

FOLLOW(Program) = {**EOF**}
FOLLOW(SpinProperties) = {**SOLID, COMPOSITE, LATTICE**}
FOLLOW(SpinProperty) = {**SPIN, SOLID, COMPOSITE, LATTICE**}
FOLLOW(Type) = {**ID**}
FOLLOW(Value) = {“,”,“)”, “;”}

FOLLOW(DefSolids)	= {**SPIN**, **LATTICE**}
FOLLOW(DefSolid)	= {**SPIN, LATTICE**}
FOLLOW(Arguments)	= {**BEGIN, "-", "(", NUM, ID, x, y, z, 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**}
FOLLOW(Id_List)	= {**")", ","**}
FOLLOW(Id_list_)	= {**")"**}
FOLLOW(Exp)	= {**";", ")", ","**}
FOLLOW(Exp_)	= {**";", ")", ","**}
FOLLOW(Exp1)	= {"||",**";", ")", ","**}
FOLLOW(Exp1_)	= {"||",**";", ")", ","**}
FOLLOW(Exp2)	= {"||", **"&&", ";", ")", ","**}
FOLLOW(Exp2_)	= {"||", **"&&", ";", ")", ","**}
FOLLOW(Exp3)	= {"||", **"&&",** "==", "<>", **";", ")", ","**}
FOLLOW(Exp3_)	= {"||", **"&&",** "==", "<>", **";", ")", ","**}
FOLLOW(Exp4)	= {"||", **"&&",** "<", "<=",">=", ">", "==", "<>", **";", ")", ","**}
FOLLOW(Exp4_)	= {"||", **"&&",** "<", "<=",">=", ">", "==", "<>", **";", ")", ","**}
FOLLOW(Exp5)	= {"||", **"&&",** "+", "-", "<", "<=", ">=", ">", "==", "<>", **";", ")", ","**}
FOLLOW(Exp5_)	= {"||", **"&&",** "+", "-", "<", "<=", ">=", ">", "==", "<>", **";", ")", ","**}
FOLLOW(Exp6)	= {"||", **"&&",** "*", "/", "+", "-", "<", "<=", ">=", ">", "==", "<>", **";", ")", ","**}
FOLLOW(Exp7)	= {"||", **"&&",** "*", "/", "+", "-", "<", "<=", ">=", ">", "==", "<>", **";", ")", ","**}
FOLLOW(Exp8)	= {"||", **"&&",** "*", "/", "+", "-", "<", "<=", ">=", ">", "==", "<>", **";", ")", ","**}
FOLLOW(Exp9)	= {"||", **"&&",** "*", "/", "+", "-", "<", "<=", ">=", ">", "==", "<>", **";", ")", ","**}
FOLLOW(Exp10)	= {"||", **"&&",** "*", "/", "+", "-", "<", "<=", ">=", ">", "==", "<>", **";", ")", ","**}
FOLLOW(ExpProperty)	= {"||", **"&&",** "*", "/", "+", "-", "<", "<=", ">=", ">", "==", "<>", **";", ")", ","**}
FOLLOW(Expr_list)	= {")"}
FOLLOW(Expr_list_)	= {","}
FOLLOW(Stmt_List)	= {**END**}
FOLLOW(Statement)	= {";"}
FOLLOW(PropertyList)	= {")", ","}

Figura 9 - Conjunto First e Follow

Para definição dos *tokens* na análise léxica é necessário regras que definam quais são as palavras pertencentes à gramática. As expressões regulares que denotam todas as palavras da gramática estão definidas na Tabela 1.

V	**Expressão do tipo** $V \rightarrow$
SPIN	spin
PROPERTY	property
INT	int
FLOAT	float
BOOL	bool
SOLID	solid
COMPOSITE	composite
BEGIN	begin
END	end
LATTICE	lattice
FOR	for
INSERT	insert
TRUE	true
FALSE	false
ASSIGN	=

UNION	union
INTERSECTION	intersection
DIFFERENCE	difference
NOT	not
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	&&
COS	cos
SIN	sin
TAN	tan
ACOS	acos
ASIN	asin
ATAN	atan
ATAN2	atan2
COSH	cosh
SINH	sinh
TANH	tanh
EXP	exp
FREXP	frexp
LDEXP	ldexp
LOG	log
LOG10	log10
MODF	modf
POW	pow
SQRT	sqrt
CEIL	ceil
FABS	fabs
FLOOR	floor
FMOD	fmod
ROTATEX	rotatex
ROTATEY	rotatey
ROTATEZ	rotatez
SCALE	scale
TRANSLATE	translate
SHEAR	shear
Atributo	letra(letra|dígito)*
Atributo	dígito(dígito)* | dígito(dígito)*. (dígito)*
X	x
Y	y
Z	z
EOF	EOF (caractere de fim de arquivo)

Tabela 1 - Expressões Regulares referentes às palavras da Linguagem de geração de objetos implícitos.

Para a definição das regras semânticas são necessárias as especificações de algumas características da linguagem:

1. Existem dois tipos pré-definidos na linguagem – *Solid* e *Number*. O tipo *Solid* se destina ao armazenamento de estruturas de sólidos e *Number* se destina ao armazenamento de qualquer valor numérico real;

2. A linguagem é fracamente tipada, ou seja, não existem declarações de variáveis, elas recebem o valor que lhes foi atribuída independente do tipo. Uma variável que a princípio guardava um sólido – do tipo *Solid* pode receber sem problema algum um valor numérico. Passando daquele momento em diante a ser uma variável do tipo *Number*.

Regras semânticas são extremamente importantes, elas definem características peculiares de associação de termos de uma linguagem de forma sensível ao contexto em que as sentenças ocorrem.

As regras semânticas da linguagem são a seguinte:

1. A atribuição $\alpha = \beta$ é válida se somente se β já estiver envolvido em alguma atribuição do tipo $\beta = \delta$ anterior ou esta atribuição estiver dentro da definição de um sólido ("*COMPOSITE*" ou "*SOLID*") e β for parâmetro dele;

2. Operações "&&" e "||" 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 é booleano – *true* ou *false*;

3. As funções e operações matemáticas e expressões relacionais descritas na linguagem, tais como "*sin*", "*cos*", "*atan*", "+", "-", "*", "*pow*", "*sqrt*", ">", "<", "==" são validas apenas quando as expressões envolvidas resultam no tipo *Number*. As construções $\alpha + \beta$, *pow*(α, β), $\alpha > \beta$ são válidas se, e somente se o resultado da expressão α e o resultado da expressão β são numéricos;

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

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

6. As únicas expressões que podem conter as palavras reservadas *x, y* e *z* são aquelas que definem as funções características dos sólidos (definidas como sendo o "Exp" em DefSolid→**solid ID** Arguments Exp "**;**");

7. Parâmetros passados para definições de sólidos ("*COMPOSITE"* ou "*SOLID*") na construção das estruturas só podem ser do tipo *Number*;

8. A primeira Expressão atrelada à instrução *"Insert*" deve ser do tipo *Solid*;

9. A instrução "*Insert"* dentro de "*COMPOSITE"* não pode atrelar consigo passagem de características das estruturas. A operação *Insert* dentro do programa principal pode opcionalmente atrelar mudanças nas características pré-definidas em "*SpinProperty*";

10. A definição de "lattice" só pode receber expressões do tipo *Number*

Capítulo 4

Descrição do Compilador

O compilador foi construído por este autor e por [Souza, 2009]. Tem como finalidade analisar e gerar uma representação intermediaria para um código escrito na linguagem de geração de objetos definida neste trabalho.

O compilador foi projetado para receber um programa, escrito nos moldes e estruturas correspondentes a linguagem denotada pela gramática exposta no item anterior. A submissão do programa pode ser feita via console ou via um arquivo com extensão "txt".

A função do compilador é fazer toda a análise do código – reportando os possíveis erros ao usuário – e gerar uma Árvore de Sintaxe Abstrata (ASA). Os próximos tópicos expõem as partes, estruturas de dados e regras utilizadas na construção do compilador.

A estrutura geral de funcionamento do compilador esta expressa na Figura 10.

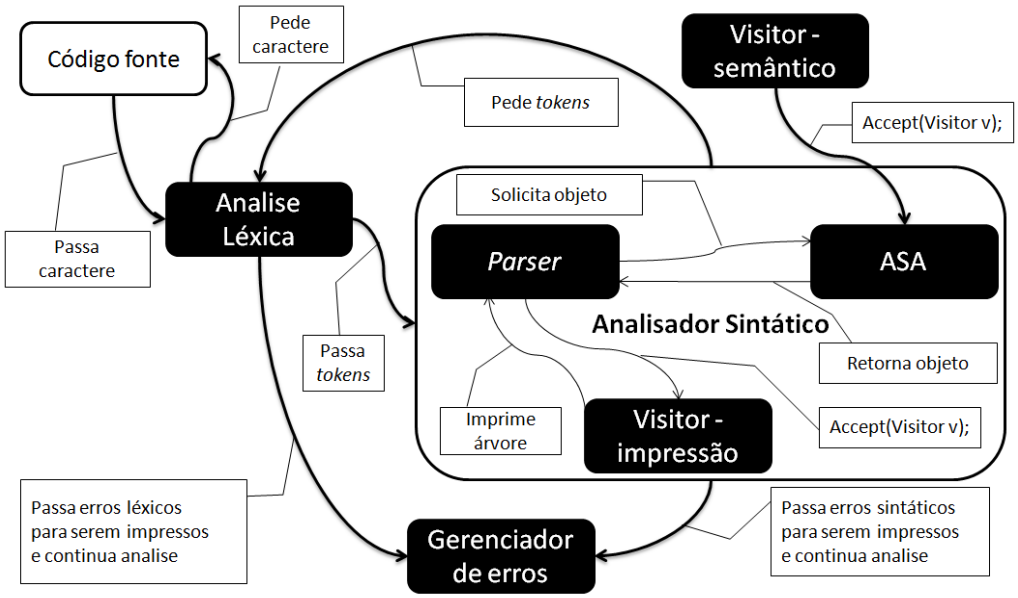

Figura 10 – Diagrama Estrutural do Compilador

O compilador foi estruturado e dividido em quatro partes:

1. **Gerenciador de Caracteres:** Responsável pela leitura, armazenamento e distribuição dos caracteres provenientes do código fonte;
2. **Analisador Léxico e Sintático:** Responsável pela análise léxica, sintática e construção da de uma ASA;
3. **Analisador Semântico:** Responsável pela análise semântica;
4. **Gerenciador de Erros:** Responsável em reportar todos os erros ocorridos na compilação;

Será apresentado a seguir o funcionamento de cada parte.

4.1 Gerenciador de Caracteres

Provê ao compilador total acesso e utilização das seguintes estruturas:

***Buffer* com sentinela:** Estrutura composta por um arranjo de duas posições onde cada posição corresponde a um *buffer*. O preenchimento é feito da seguinte maneira: É lido do arquivo e armazenado no primeiro *buffer* a quantidade de caracteres que ele comporta. Em seguida esses caracteres são usados pelo compilador até todos os caracteres sejam consumidos. Quando isso acontece é lido 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. Ocorre assim uma utilização rotacional dos *buffers*, tornando possível a utilização racional de memória. O arquivo não precisa ser carregado totalmente em memória.

Tabela de Símbolos: Estrutura complexa envolvendo um arranjo – Tabela *Hash* – de ponteiros para uma estrutura definida como entradaTab. A estrutura entradaTab é usada para guardar as informações dos *tokens* passados pela análise léxica. As inserções na Tabela *Hash* são feitas através de uma função de *hash* que calcula qual índice será usado a partir do lexema do *token*.

Extensão da Tabela de Símbolos: Estrutura utilizada para armazenar dados destinados a análise semântica. Funciona como uma extensão para a Tabela de Símbolos.

Parâmetros: É uma lista que guarda a referência, para a Tabela de Símbolos, de todos os parâmetros contidos em uma definição de sólido.

Gerenciador de *Strings*: Tem por objetivo guardar todos os lexemas dos *tokens* inseridos na tabela de símbolos. Assim a tabela de símbolos guarda apenas uma referência de onde está seu lexema.

A Figura 11 representa a utilização e a dependência entre a Tabela *Hash*, a Tabelas de Símbolos e sua extensão.

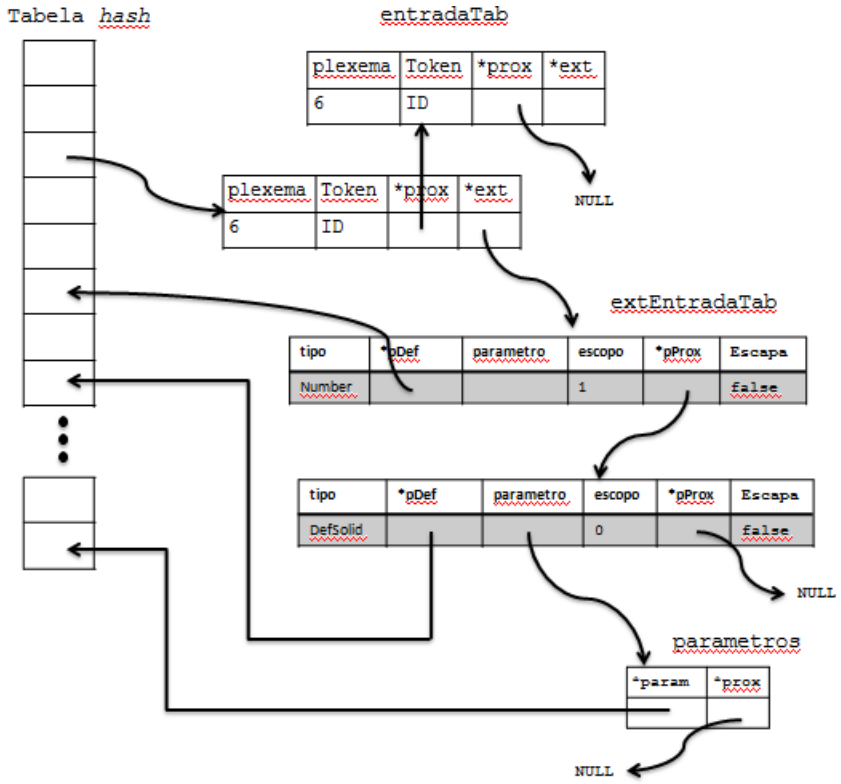

Figura 11 - Estruturas de armazenamento de informações

4.2 Analisador Léxico e Sintático

Analise Léxica

A partir das expressões regulares da linguagem fonte foi construído um autômato (Figura 12) e se baseando nele foi criada uma tabela – tabEstado – e uma função –

prox_estado – que guiam as transições de estados à medida que cada caractere do código é lido do *buffer*.

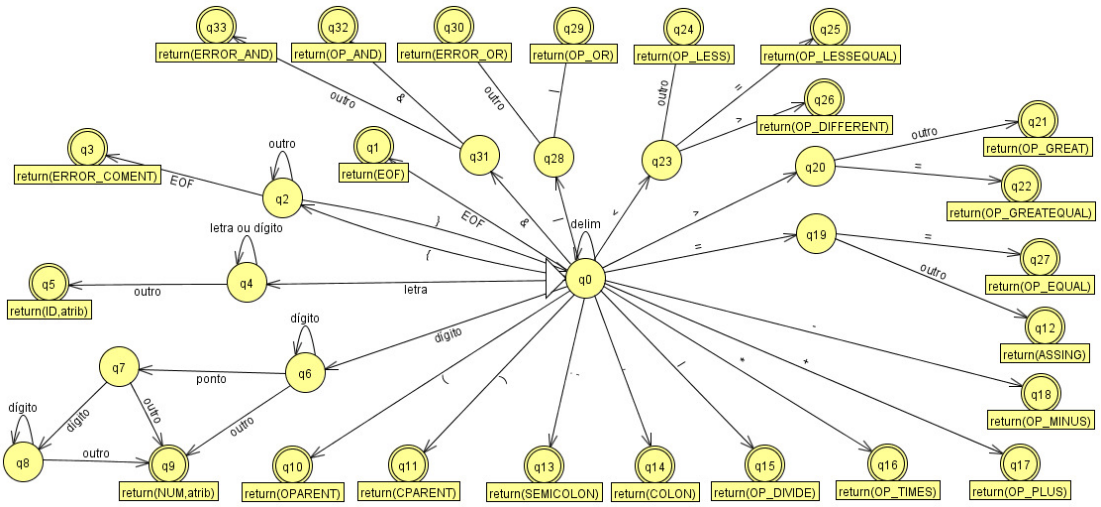

Figura 12 - Autômato que representa todas as expressões regulares da gramática

Conforme os *token*s vão sendo encontrados, os mesmos são inseridos na Tabela de Símbolos e passados para a análise sintática. Os erros encontrados são encaminhados ao gerenciador de erros.

Análise sintática

O analisador sintático utiliza um módulo denominado *Parser* para verificar cada *token* passado pela analise léxica. O *Parser* possui as seguinte funções:

- *match*: Pega o próximo *token*.
- *skipto*: Recebe como parâmetro um arranjo que contém o conjunto *Follow* da estrutura onde houve algum erro sintático. A idéia da função é desprezar a estrutura onde houve o erro e continuar a análise sintática no restante do código.

- *matchorskip*: Verifica se o *token* é o esperado. Se for utiliza a função *match*. Se não, reporta o erro ao gerenciador de erro e utiliza a função *skipto* para continuar a análise.

O analisar sintático implementa as regras de construção descritas na gramática, ou seja, lê os *token* tentando enquadrá-los em algumas das regras gramaticais. À medida que os *token* são lidos e enquadrados em alguma regra gramatical um elemento da árvore de sintaxe abstrata. Para tanto, o *Parser* utiliza as classes definidas no módulo ASA.

Caso ocorra um erro em alguma estrutura, esta será descartada, isto é, não será construída. Os filhos de uma estrutura não construída são analisados a procura de erros mas não são incorporados a ASA.

A ASA segue a estrutura expressa na Tabela 2

	Nodo	*Formas possíveis*
1	**PROGRAM**	**- (SOLID_LIST, SPIN_PROPERTIES, LATTICE, STATEMENT_LIST)**
2	**SOLID_LIST**	**- (DECL_SOLID, SOLID_LIST)** **- NULL**
3	**DECL_SOLID**	**- (ID, ARGUMENT_LIST, EXP)** **- (ID, ARGUMENT_LIST, STATEMENT_LIST)**
4	**ARGUMENT_LIST**	**- (ID, ARGUMENT_LIST)** **- NULL**
5	**SPIN_PROPERTIES**	**- (SPIN_PROPERTY, SPIN_PROPERTIES)** **- NULL**
6	**SPIN_PROPERTY**	**- (TYPE, ID, NUM)** **- (TYPE, ID, BOOL)**
7	**LATTICE**	**- (EXP, EXP, EXP, EXP, EXP, EXP, EXP, EXP, EXP)**
8	**STATEMENT_LIST**	**- (STATEMENT, STATEMENT_LIST)** **- NULL**
9	**STATEMENT**	**- (ASSIGN)** **- (INSERT)** **- (WHILE)**
10	**ASSIGN**	**- (ID, EXP)**
11	**INSERT**	**- (EXP, PROPERTY_LIST)**
12	**PROPERTY_LIST**	**- (PROPERTY_ASSIGN, PROPERTY_LIST)** **- NULL**
13	**PROPERTY_ASSIGN**	**- (ID, NUM)** **- (ID, BOOL)**
14	**WHILE**	**- (EXP, STATEMENT_LIST)**
15	**EXP**	**- (ID)** **- (NUM)** **- (x)** **- (y)** **- (z)** **- (OR_OP)** **- (AND_OP)**

		- (RELATIONAL_OP) **- (ADDITION_OP)** **- (MULTIPLICATION_OP)** **- (EQUAL_OP)** **- (UNARY_OP)** **- (FUNCTION_MATH)** **- (CSG_OP)** **- (NOT_OP)** **- (TRANSFORMATION_OP)** **- (OBJECT_CALL)**
16	**OR_OP**	**- (EXP, EXP)**
17	**AND_OP**	**- (EXP, EXP)**
18	**RELATIONAL_OP**	**- (OP_REL, EXP, EXP)**
19	**ADDITION_OP**	**- (OP_ADD, EXP, EXP)**
20	**MULTIPLICATION_OP**	**- (OP_MUL, EXP, EXP)**
21	**EQUAL_OP**	**- (OP_EQUAL, EXP, EXP)**
22	**UNARY_OP**	**- (EXP)**
23	**FUNCTION_MATH**	**- (FUNC_MATH, EXP)** **- (FUNC_MATH, EXP, EXP)**
24	**CSG_OP**	**- (CSG, EXP, EXP)**
25	**NOT_OP**	**- (EXP)**
26	**TRANSFORMATION_OP**	**- (TRANSFORMATION, EXP, EXP)** **- (TRANSFORMATION, EXP, EXP, EXP, EXP)**
27	**OBJECT_CALL**	**- (ID, EXPR_LIST)**
28	**EXPR_LIST**	**- (EXP, EXPR_LIST)** **- NULL**

Tabela 2 - Estrutura da ASA

Para imprimir a árvore gerada foi utilizado o padrão *Visitor* – padrão de projeto que possibilita aplicar uma operação em objetos de classes distintas sem precisar alterá-las.

Foi criado uma classe denominada VisitorImpresao que faz a impressão de todos as estruturas da ASA.

4.3 Analisador Semântico

Esta parte do compilador é responsável por identificar as possíveis falhas semânticas do código fonte. É implementada através de um *Visitor* implementado na classe visitorSemantico que percorre a ASA verificando possíveis falhas.

Com o auxílio das funções e estruturas e variáveis expostas abaixo as regras semânticas da linguagem são verificadas.

A seguir estão as funções e variáveis utilizadas:

- *beginScope*: Função que inicia um novo escopo salvando o atual;
- *endScope*: Função que finaliza o escopo atual e volta para o anterior;
- *put*: Função para a inserção de uma estrutura na tabela de símbolos auxiliar;
- *get*: Função para pegar uma estrutura na tabela de símbolos auxiliar;
- *estaDentrodeDEFSOLID:* Variável usada para verificar se o trecho de código em análise esta dentro de uma definição de sólido;
- *estaDentrodeDEFCOMPOSITE*: Variável usada para verificar se o trecho código em análise esta dentro de uma definição de composição de sólidos;

Na análise léxica, todo novo lexema é inserido na tabela de símbolos na posição em que a função *hash* utilizada calcular.

Na análise semântica, construímos tantas estruturas quanto necessário para cada declaração de sólidos e composições de sólidos e para cada atribuição.

4.4 Gerenciador de Erros

O gerenciador de erro provê funções que reportam os erros ocorridos ao usuário que são as seguintes:

- *imprimeErroArquivo*: Imprime erro reportado pelo compilador ao tentar abrir o arquivo que contém o código, arquivo não encontrado.
- *imprimeErroExtensao*: Imprime erro reportado pelo compilador quando a extensão do arquivo passado não é compatível com a padrão, no caso “txt”.
- *imprimeErroLexico*: Recebe um valor pré estabelecido de erro léxico e imprime o erro. Os erros léxicos são definidos pelas constantes mostradas na Tabela 3.

- *erroSintatico*: Recebe um valor pré estabelecido de erro sintático e imprime o erro. Os erros léxico são definidos pelas constantes exibidas na Tabela 3.

- *erroSemantico*: Imprime erro semântico, recebe a mensagem de qual erro semântico do analisador semântico e em qual linha ele ocorreu

- *imprimeResultErros*: Imprime o totalização dos erros encontrados divididos em: Erros Léxicos, Erros Sintáticos e Erros Semânticos.

- *totalErros*: Retorna o total de erros geral de erros presentes no código.

Constante	Tipo de Erro	Mensagem de Erro
ERROR_INVALID_CARACTER	Léxico	Caractere inválido
ERROR_COMMENT	Léxico	Fim inesperado do arquivo em comentário
ERROR_OR	Léxico	Esperado o caracter '|'
ERROR_AND	Léxico	Esperado o caracter '&'
TK_BEGIN	Sintático	Esperado um 'begin'
TK_END	Sintático	Esperado um 'end'
TK_EOF	Sintático	trecho de programa apos 'end' final
TK_PROPERTY	Sintático	Esperado um 'property'
TK_ID	Sintático	Esperado um identificador
TK_ASSIGN	Sintático	Esperado um '=' (uma atribuicao)
TK_SEMICOLON	Sintático	Esperado um ';'
TK_LATTICE	Sintático	Esperado um 'lattice'
TK_OPARENT	Sintático	Esperado um '('
TK_COLON	Sintático	Esperado um ','
TK_CPARENT	Sintático	Esperado um ')'
ERR_NOT_VALUE	Sintático	Esperado um valor (número, true ou false)
ERR_NOT_VALUE_INSERT	Sintático	Esperado um valor (identificador, número, true ou false)
ERR_INVALID_TYPE	Sintático	Esperado um tipo ('int', 'float' ou 'bool')
ERR_NOT_OPERATION	Sintático	Esperado um operador
ERR_NOT_ASSIGN	Sintático	Esperado um operador de atribuição ('=')
ERR_NOT_STMT	Sintático	Esperado um inicio de instrução (identificador, 'insert' ou 'while')

Tabela 3 - Mensagens de Erro

Capítulo 5

Resultados

Este capítulo apresenta algumas árvores de sintaxe abstrata geradas, pelo compilador, a partir de códigos propostos.

5.1 Esfera Maciça

O código apresentado na Figura 13 gera uma esfera de raio unitário, com centro na origem. O *lattice* definido como sendo um cubo, compreendido entre no intervalo $[-1,1]$ nos eixos x, y e z e com resolução de igual a 10 em cada direção.

```
{Definição de uma esfera maciça onde o raio é passado como parâmetro}
solid Sphere(raio)
    {Equação que define quais pontos pertencem ao objeto}
    pow(x, 2) + pow(y, 2) + pow(z, 2) - raio < 0;

{Propriedade booleana que deve ser incorporada as estruturas}
spin property bool color = false;

{Definicao do lattice}
lattice(-1,-1,-1,1,1,1,10,10,10)

begin

    {O identificador s1 recebe um objeto construído a partir da definição Sphere com raio = 1}
    s1 = Sphere(1);

    {É inserido no lattice o objeto contido em s1 com a propriedade color = true}
    insert(s1, color = true);
end
```

Figura 13 - Código de criação de uma esfera

A ilustração abaixo mostra a ASA criada, pelo compilador, após a análise do código descrito na Figura 13. Ao lado de cada estrutura há um comentário que descreve seu significado. Para facilitar a compreensão foi adicionado um pseudonome para a verificação e alinhado segundo a tabulação da ASA – Comentários mais à direita pertencem as estruturas descritas pelos comentários mais à esquerda.

```
-PROGRAM        {Inicio do programa p1}
    -SOLID_LIST         {Estruturas de sólidos declaradas em p1}
        -DECL_SOLID             {Declaração do primeiro objeto de O1}
            -ID.Sphere                  {Nome do objeto O1 declarado}
            -ARGUMENT_LIST              {Lista de parâmetros do objeto O1}
```

```
            -ID.raio                    {Identificador Id1 do primeiro parâmetro de O1}
        -RELATIONAL_OP          {Expressão E1 que descreve o objeto O1.(Relacional)}
            -LESS               {Operador da expressão E1. (<)}
            -ADDITION_OP        {Lado esquerdo de E1. (Expressão Adicional E2)}
                -MINUS              {Operador da expressão E2.(-)}
                -ADDITION_OP        {A esquerda da expressão E2.(Expressão Adicional E3)}
                    -PLUS               {Operador da expressão E3.(+)}
                    -ADDITION_OP        {A esquerda de E3.Expressão Adicional E4}
                        -PLUS               {Operador de E4.(+)}
                        -FUNCTION_MATH      {A esquerda de E4. Função F1}
                            -POW                {Função F1 é um pow}
                            -X.x                { pow(x, }
                            -NUMBER.2           { pow(x,2)}
                        -FUNCTION_MATH      {A direita de E4. Função F2}
                            -POW                {Função F1 é um pow}
                            -Y.y                {pow(y, }
                            -NUMBER.2           {pow(y,2) }
                    -FUNCTION_MATH      {A direita de E3. Função F3}
                        -POW                {Função F3 é um pow}
                        -Z.z                {pow(z, }
                        -NUMBER.2           {pow(z,2) }
                -ID.raio            {Lado direito de E2.  Identificador Id1}
            -NUMBER.0           {Lado direito de E1.  Numero 0}
    -SPIN_PROPERTIES        {Propriedades definidas no programa}
        -SPIN_PROPERTY          {Primeira propriedade}
            -BOOL                   {Tipo da primeira propriedade}
            -ID.color               {Identificador da primeira propriedade Id2}
            -FALSE                  {Valor atribuído a primeira propriedade}
    -LATTICE                {Definição do lattice a ser utilizado}
        -UNARY_OP               {Expressão que define o mínimo em X.(unária)E5}
            -MINUS                  {Operador da expressão E5.(-)}
            -NUMBER.1               {Operando de E5, Numero 1}
        -UNARY_OP               {Expressão que define o mínimo em Y.(unária)E6}
            -MINUS                  {Operador da expressão E6.(-)}
            -NUMBER.1               {Operando de E6, Numero 1}
        -UNARY_OP               {Expressão que define o mínimo em Z, (unária)E7}
            -MINUS                  {Operador da expressão E7.(-)}
            -NUMBER.1               {Operando de E7, Numero 1}
        -NUMBER.1               {Expressão que define o máximo em X. Numero 1}
        -NUMBER.1               {Expressão que define o máximo em Y. Numero 1}
        -NUMBER.1               {Expressão que define o máximo em Z. Numero 1}
        -NUMBER.10              {Expressão que define a resolução em X. Numero 10}
        -NUMBER.10              {Expressão que define a resolução em Y. Numero 10}
        -NUMBER.10              {Expressão que define a resolução em Z. Numero 10}
    -STATEMENT_LIST         {Instruções do programa P1}
        -ASSIGN                 {Primeira instrução I1. Instrução de Atribuição}
            -ID.s1                  {Identificador  ao qual será atribuído um valor em I1}
            -OBJECT_CALL            {Valor da atribuição I1. (Um objeto)}
                -ID.Sphere              {Nome do objeto. Se refere a O1 no caso}
                -EXPR_LIST              {Parâmetros para a construção do objeto O1}
                    -NUMBER.1           {Primeiro parâmetro passado para a construção de O1}
        -INSERT                 {Segunda instrução. Instrução de inserção}
            -ID.s1                  {Objeto a ser inserido  definido pelo identificador I1}
            -PROPERTYASSIGN         {Lista de propriedades a ser alteradas}
                -ID.color               {Primeira propriedade a ser alterada, Id2}
                -TRUE                   {Novo valor da primeira propriedade.(Id2)}
```

Figura 14 - Impressão da ASA

5.2 Cilindro Maciço

O código apresentado na Figura 15 Figura 13gera uma cilindro as seguintes características:

- Base no plano xy;
- Raio da base igual a 2;
- Comprimento em z de 10 unidades;
- Definidos para os valores de z compreendidos no intervalo $[-5,5]$;
- Definido em um *lattice* representado pelo paralelepípedo regular descrito no espaço definido pelo intervalos $[-5,5]$, $[-5,5]$ e $[-10,10]$ nos eixos x, y e z com resolução de 10 unidades em cada direção.

```
solid Cilinder
    pow(x, 2) + pow(y, 2) < 2 && z < -5 && z < 5;

lattice(-5,-5,-10,5,5,10,10,10,10)

begin
     insert(Cilinder());
end
```

Figura 15 - Código para a criação de um cilindro

O código gera a seguinte árvore:

```
-PROGRAM
    -SOLID_LIST
        -DECL_SOLID
            -ID.Cilinder
            -AND_OP
                -AND_OP
                    -RELATIONAL_OP
                        -LESS
                        -ADDITION_OP
                            -PLUS
                            -FUNCTION_MATH
                                -POW
                                -X.x
                                -NUMBER.2
                            -FUNCTION_MATH
```

```
                                    -POW
                                    -Y.y
                                    -NUMBER.2
                                -NUMBER.2
                            -RELATIONAL_OP
                                -LESS
                                -Z.z
                                -UNARY_OP
                                    -MINUS
                                    -NUMBER.5
                        -RELATIONAL_OP
                            -LESS
                            -Z.z
                            -NUMBER.5
    -LATTICE
        -UNARY_OP
            -MINUS
            -NUMBER.5
        -UNARY_OP
            -MINUS
            -NUMBER.5
        -UNARY_OP
            -MINUS
            -NUMBER.10
        -NUMBER.5
        -NUMBER.5
        -NUMBER.10
        -NUMBER.10
        -NUMBER.10
        -NUMBER.10
    -STATEMENT_LIST
        -INSERT
            -OBJECT_CALL
                -ID.Cilinder
```

Figura 16 - ASA gerada a partir do código do cilindro

5.3 Composição definida pela diferença entre o Cilindro e a Esfera

Criar uma estrutura que corresponda ao cilindro criado no item 5.2 subtraindo dos pontos pertencentes à esfera criada no item 5.1, usando o mesmo *lattice* definido para o cilindro.

Código descrito na imagem abaixo.

```
solid Sphere
    pow(x, 2) + pow(y, 2) + pow(z, 2) - 1 < 1;

solid Cilinder
    pow(x, 2) + pow(y, 2) < 2 && z < -5 && z < 5;

composite CilinderMenosSphere
    begin
          insert(diference(Cilinder(), Sphere()));
    end

lattice(-5,-5,-10,5,5,10,10,10,10)

begin
     insert(CilinderMenosSphere());
end
```

Figura 17 - Cilindro menos Esfera

A representa a árvore gerada.

```
-PROGRAM
    -SOLID_LIST
        -DECL_SOLID
            -ID.Sphere
            -RELATIONAL_OP
                -LESS
                -ADDITION_OP
                    -MINUS
                    -ADDITION_OP
                        -PLUS
                        -ADDITION_OP
                            -PLUS
                            -FUNCTION_MATH
                                -POW
                                -X.x
                                -NUMBER.2
                            -FUNCTION_MATH
                                -POW
                                -Y.y
                                -NUMBER.2
                        -FUNCTION_MATH
                            -POW
                            -Z.z
                            -NUMBER.2
                    -NUMBER.1
                -NUMBER.1
        -DECL_SOLID
            -ID.Cilinder
            -AND_OP
                -AND_OP
                    -RELATIONAL_OP
                        -LESS
                        -ADDITION_OP
                            -PLUS
                            -FUNCTION_MATH
```

```
                                    -POW
                                    -X.x
                                    -NUMBER.2
                                -FUNCTION_MATH
                                    -POW
                                    -Y.y
                                    -NUMBER.2
                            -NUMBER.2
                        -RELATIONAL_OP
                            -LESS
                            -Z.z
                            -UNARY_OP
                                -MINUS
                                -NUMBER.5
                    -RELATIONAL_OP
                        -LESS
                        -Z.z
                        -NUMBER.5
            -DECL_SOLID
                -ID.CilinderMenosSphere
                -STATEMENT_LIST
                    -INSERT
                        -CSG_OP
                            -DIFFERENCE
                            -OBJECT_CALL
                                -ID.Cilinder
                            -OBJECT_CALL
                                -ID.Sphere
    -LATTICE
        -UNARY_OP
            -MINUS
            -NUMBER.5
        -UNARY_OP
            -MINUS
            -NUMBER.5
        -UNARY_OP
            -MINUS
            -NUMBER.10
        -NUMBER.5
        -NUMBER.5
        -NUMBER.10
        -NUMBER.10
        -NUMBER.10
        -NUMBER.10
    -STATEMENT_LIST
        -INSERT
            -OBJECT_CALL
                -ID.CilinderMenosSphere
```

Figura 18 - ASA referente ao Cilindro menos a Esfera

Capítulo 6

Conclusão

Neste trabalho foi apresentada uma linguagem para se criar objetos implícitos para fins de simulação.

A linguagem separa bem a definição e a utilização de cada objeto. Isso possibilita portar as definições já criadas, basta copiar o código referente à definição que se quer portar. Objetos complexos podem ser criados facilmente através das mais variadas composições e transformações dos objetos já definidos.

O fato de a linguagem ser fracamente tipada a torna passível de expansão. Quaisquer adições de novos tipos de estruturas não causariam drásticas modificações na linguagem no compilador e no seu interpretador.

É de suma importância a construção de um interpretador e de um visualizador que possibilitem a interpretação e exibição dos objetos criados.

É necessário uma extensa documentação, descrevendo a linguagem e sua correta utilização, e um aprimoramento do compilador, tornando as mensagens de erros mais intuitivas e explicativas.

Os trabalhos futuros que podem ser desenvolvidos estão relacionados com os itens descritos anteriormente e se resumem nos tópicos abaixo:

- Construção de um interpretador e de um visualizador;
- Adaptação dos simuladores existentes para utilização da linguagem;
- Criação das definições dos objetos mais visados nas simulações e possível incorporação dos mesmos na linguagem de forma transparente ao usuário;
- Pesquisas no meio científico com o intuito de adequar a linguagem a possíveis cenários não captados por este trabalho.

Referências Bibliográficas

VELHO, L.; GOMES, J; FIGUEIREDO, L. H. **Implicit Objects in Computer Graphics.** Publicado Springer, Nova Iorque, Estados Unidos, 2002.

MENEZES, P. F. B. **Linguagens formais e autômatos.** Edição: 3 – Publicado por Editora Sagra Luzzatto, 2000.

REQUICHA, A. **Representation for rigid solids: Theory, methods, and systems.** Publicado por *ACM Computing Surveys*. Nova Iorque, Estados Unidos, 1980.

RICCI, A. **A constructive solid geometry for computer graphics.** Publicado por *The Computer Journal*. Bologna, Italia 1973.

AHO, A.V.; SETHI, R; ULLMAN, J, D. **Compilers: Principles, Techniques, and Tools.** – Publicado por Pearson Education. China, 1986.