Anima Algo - *software* de animação de algoritmos para auxílio ao ensino e entendimento da programação[1]

Frederico de Almeida Martins[2]

Resumo

O desenvolvimento de um *software* que permita acompanhar a execução de um algoritmo de forma visual com intuito de auxiliar no ensino e entendimento da programação, parte da elaboração de uma linguagem que seja completa e de fácil entendimento para a escrita desse algoritmo, passa pelos projetos do compilador dessa linguagem e do executor do código ou árvore sintática gerada pelo compilador e culmina na implementação propriamente dita. Este artigo descreve a implementação de um protótipo que permite escrever, compilar e visualizar a árvore sintática gerada a partir do algoritmo escrito, visando dar suporte à implementação do executor.

Palavras-Chave

Algoritmo; Animação; Linguagem de Programação; Compilador; Lex; Yacc.

Anima Algo - algorithm animation software to assist the education and understanding of the programming

Frederico de Almeida Martins

Abstract

The development of a software that allows to follow the execution of an algorithm in a visual way with the intention of assisting in the education and understanding of the programming, starts with the elaboration of a language that is complete and easy understanding for the writing of this algorithm, passes for the projects of the compiler of this language and the executer of the code or syntactic tree generated by the compiler and culminates in the implementation. This article discrible the implementation of a prototype that allows write, compile and visualize the syntactic tree generated from the algorithm wrote, with the purpose of giving support to the implementation of the executer.

Keywords

Algorihtm; Animation; Programming Language; Compiler; Lex; Yacc.

[1] Trabalho desenvolvido na Disciplina de Tópicos Especiais VII - Universidade de Uberaba.
[2] Aluno do Curso de Bacharelado em Sistemas de Informação – Universidade de Uberaba.

1. Introdução

De acordo com [VARANDA E HENRIQUES], a animação de um algoritmo é definida como sendo um tipo de visualização dinâmica das principais abstrações expressas por esse algoritmo.

Segundo [REZENDE] e [MENDES], uma grande dificuldade encontrada pelos estudantes - e conseqüentemente pelos professores - é a falta da capacidade de abstração do processo de execução de um programa. São estudantes que, na sua maioria, nunca tiveram contato com a matéria e têm dificuldade em abstrair o problema para instruções de uma linguagem de programação.

A animação de algoritmos vem dar uma contribuição nesse sentido à medida que torna visual o processo de execução, facilitando o entendimento do que ocorre quando uma dada instrução é executada [VARANDA E HENRIQUES]. Além disso, segundo [VARANDA E HENRIQUES], pode-se transpor os limites do ensino da programação e aplicar a animação de algoritmos em qualquer área onde algoritmos são utilizados, como por exemplo no ensino da matemática.

Surge nesse ponto, um imenso leque de aplicações para ferramentas de animação de algoritmos, que vai desde o ensino médio até o ensino superior, e também para os autodidatas.

Propõe-se assim, o desenvolvimento de um *software* que permita escrever, compilar e executar de forma animada um determinado algoritmo escrito numa linguagem de fácil entendimento.

O projeto para o desenvolvimento deste *software* foi dividido em duas fases. A primeira fase compreende o desenvolvimento de um protótipo funcional que permite escrever e compilar um algoritmo gerando uma estrutura de dados - árvore de sintaxe decorada - que será, na segunda fase, utilizada para execução do algoritmo. Nesta primeira fase, com base nos requisitos de *software*, serão feitas as definições sobre a linguagem a ser utilizada na escrita dos algoritmos, a arquitetura do *software*, o modo de funcionamento do compilador para a linguagem, a estrutura de dados da árvore de sintaxe decorada e a interface com o usuário. A segunda fase então, compreende o desenvolvimento do módulo que irá interagir com a árvore de sintaxe decorada gerada na compilação, realizando o processo de execução animada do algoritmo.

Será tratada neste artigo, a primeira fase do projeto.

2. Requisitos

Como descrito em [VARANDA E HENRIQUES], o objetivo da animação de algoritmos é perceber o funcionamento de aplicações cuja animação vai ocorrendo progressivamente como resposta a ações do utilizador, tornando a interação mais realista.

Em [MENDES], pode-se encontrar mais alguns requisitos importantes para uma ferramenta que se propões ao auxílio do ensino e entendimento da programação. São eles:

I) Ser Interativa, gerando feedback adequado para o aluno permitindo-lhe interatuar, controlar e desempenhar um papel ativo no processo de aprendizagem.

II) Ser Configurável, a fim de permitir a adição, remoção e alteração dos exemplos, exercícios propostos e soluções apresentadas, sem que tal acarrete quaisquer alterações no seu código.

III) Permitir Representações alternativas, de forma a possibilitar ao aluno diferentes pontos de vista, estilos de raciocínio e soluções para os mesmos problemas.

IV) Ser Animada, para melhor expressar as idéias, transmitindo a dinâmica envolvida e a informação patente no problema com mais realismo, facilitando a compreensão de conceitos, sem contudo gerar confusão visual.

V) Ser Simples, óbvia e intuitiva, exigindo pouco tempo de aprendizagem, mas persuasiva, sofisticada e atrativa.

VI) Ser Portável, para que seja facilmente transportável entre diversas arquiteturas e sistemas operacionais.

VII) Ser Econômica, envolvendo baixos custos de aquisição, para alcançar um elevado número de alunos.

Seguindo as idéias expostas acima, pode-se definir os requisitos básicos para o *software* a ser desenvolvido.

Como o *software* é proposto para ser utilizado no aprendizado da programação, deve-se desenvolver uma linguagem que seja completa - possuir as funcionalidades básicas de qualquer linguagem de programação - e de simples entendimento - com sintaxe intuitiva. A interface para a escrita do algoritmo deve ser amigável contando com recursos que permitam destacar com cores diferentes as palavras reservadas da linguagem, os símbolos, os números, os comentários e os identificadores.

Antes da execução do algoritmo, o mesmo deve ser analisado para verificar se os códigos correspondem às especificações da linguagem, buscando possíveis erros léxicos, sintáticos e semânticos. Os prováveis erros deverão ser mostrados indicando o local onde ocorreram e o motivo pelo qual ocorreram para que se possa corrigi-los.

A execução deverá mostrar passo a passo quais, e como, as instruções escritas no algoritmo estão sendo executadas. Por ser uma forma bem intuitiva de se entender um processo, neste caso, o processo de execução, deve-se utilizar um fluxograma, no qual as instruções (representadas por retângulos, losangos, círculos e etc) mudam de cor à medida que estão sendo executadas. O comportamento da memória: variáveis alocadas e valores assumidos, também deverá ser mostrado na execução de forma animada, com mudanças de cores e outros recursos para visualização das declarações, atribuições e comparações feitas pelas instruções.

O *software* deve ser flexível a ponto de permitir outras formas de execução (como execução simples no modo tipo *prompt*, por exemplo) e futuras alterações (como no caso da implementação de novas funções, alterações na linguagem e transporte para outras linguagens, por exemplo).

3. Arquitetura Proposta

No sistema ALMA, [VARANDA E HENRIQUES] propõem uma arquitetura baseada nos seguintes itens:

I) Criação de um *front-end* (interface da frente) para uma determinada linguagem, a partir da respectiva gramática (para diferentes linguagens, o *front-end* deverá ser gerado de novo de acordo com as suas gramáticas). O *front-end* é primeira interface com o usuário. É a interface que permite escrever e/ou editar os algoritmos, bem como compilar o código fonte escrito, garantindo que o mesmo está corretamente escrito com base nas definições da linguagem.

II) Criação de um *back-end* (interface final) que, a partir da informação extraída do resultado produzido pelo *front-end*, forneça a necessária interface para programação da animação e realize a visualização pretendida (independente da linguagem fonte). Este

back-end será implementado com base no algoritmo de um *Tree-Walker Evaluator*[3], que percorre a Árvore de Sintaxe Decorada (ASD)[4] aplicando as operações de visualização, filtragem e animação adequadas. O desenvolvimento do *back-end* envolve a criação das seguintes interfaces:

A) Uma interface que permita programar a animação, ou seja, indicar os subprogramas (dos quais se quer visualizar a seqüência de instruções) e as variáveis (das quais se quer visualizar o conteúdo); essa interface terá ainda de permitir a definição do aspecto com que os objetos selecionados serão apresentados.

B) Uma interface para visualizar a seqüência de instruções e o conteúdo das variáveis, de acordo com os requisitos expressos no item anterior.

Com base nestes conceitos, define-se a arquitetura do *software* conforme mostra a figura abaixo.

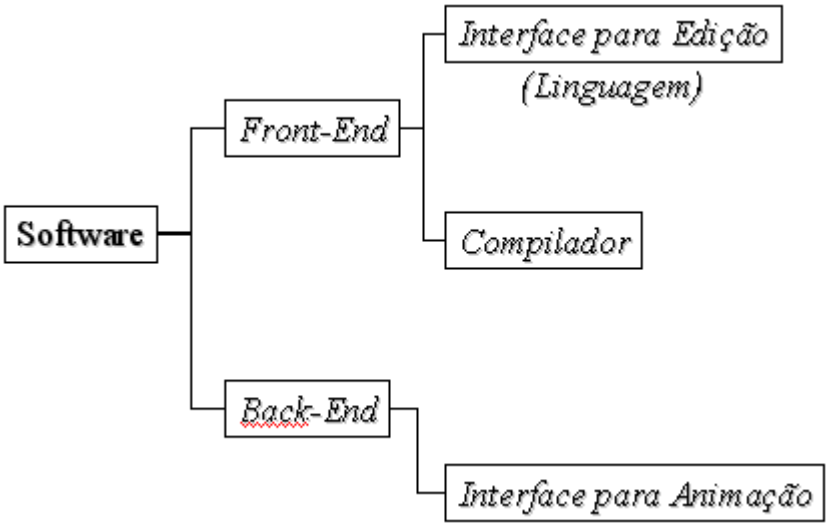

Figura 1 - Arquitetura Básica

Sendo assim, pode-se visualizar dois módulos principais do *software*:

I) *Front-End*:

- Interface: Contém o Editor para a escrita do algoritmo na linguagem definida, além de outras funcionalidades secundárias como *help*, menu de comandos e etc.
- Compilador: Encarrega-se de realizar a análise léxica, a análise sintática, a análise semântica e montar a ASD com base no código escrito no editor.

II) *Back-End*:

- Executor: Percorre a ASD de forma a gerar a animação e permitir a interação com o usuário.

Uma análise mais detalhada da arquitetura proposta acima mostra a interação entre os componentes conforme mostra a figura abaixo.

[3] Pode ser traduzido simplesmente como Avaliador de Árvore. Um algoritmo que possui as instruções necessárias para avaliar cada nó de uma árvore a medida em que a percorre segundo um percurso pé-definido.

[4] Ver tópico 4.2.1. Árvore de Sintaxe Decorada.

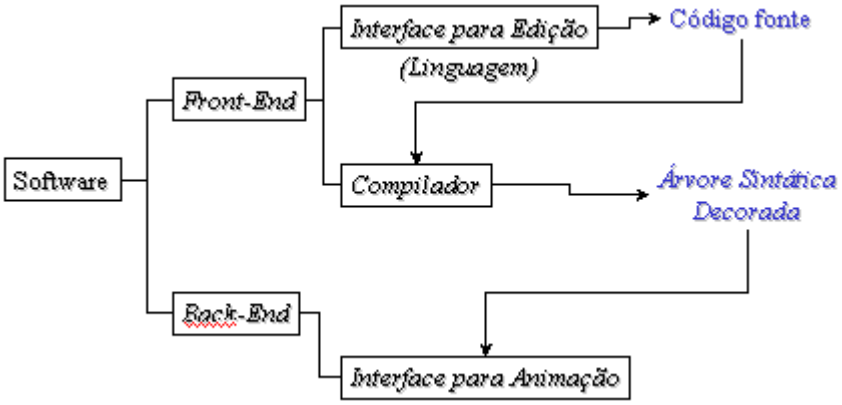

Figura 2 - Interação dos componentes

O editor fornece o código fonte como entrada para o compilador. Este retorna para a interface, através do editor, os erros que por ventura existirem neste código fonte. Ao terminar a compilação do código sem erros, o compilador terá gerado a ASD, que fica disponível na memória para que o executor realize a animação.

Como pode ser notado na Figura 2, o *back-end* necessita apenas da ASD como entrada, não importando como a mesma foi gerada, nem em qual linguagem foi escrito o algoritmo. Isso faz com que o executor seja independente do *front-end* e também permite que o *back-end* seja escrito para utilizar a ASD para a geração de código objeto e outras formas de execução diferente da execução animada.

4. Projeto do *Software*

O projeto do *software* envolve o detalhamento de cada um dos componentes da arquitetura do sistema descrita no item anterior.

Inicialmente, porém, deve-se definir qual linguagem e IDE serão utilizadas no desenvolvimento, pois algumas linguagens possuem limitações que implicam no projeto do compilador.

O Delphi 5 foi escolhido como IDE para o desenvolvimento do *software*, devido ao fato de possuir uma linguagem muito versátil, permitindo criação de tipos, uso de ponteiros na memória, recursividade e disponibilizar vários componentes (proprietários e de terceiros) para construção da interface, do compilador e do executor.

A única limitação para o Delphi é o fato de não gerar código portável.

4.1. *Front-End*

4.1.1. Linguagem

O projeto do *front-end* inicia-se com a definição da linguagem a ser utilizada na escrita dos algoritmos.

Tem-se na linguagem Pascal, amplamente utilizada e consagrada, uma linguagem completa e de fácil entendimento. Por esse motivo, o Pascal foi utilizado como base para se definir a linguagem a ser utilizada no projeto. A BNF[5] do Pascal Simplificado pode ser encontrada em [KOWALTOWSKI] e [NEPOMUCENO].

[5] Bakus-Naur From (BNF) é uma metalinguagem, uma linguagem utilizada para descrever outra linguagem. Cada sentença desta metalinguagem é chamada de regra de produção. [MAK]

Existem ainda alguns pontos que também devem ser considerados na definição da linguagem.

Primeiramente, o objetivo do projeto não é criar uma nova linguagem que contemple a resolução de qualquer problema, mas sim possibilitar o entendimento da execução utilizando animação. Também, deve-se considerar que a linguagem não necessita prover todas as funcionalidades do Pascal. Desta forma, não será implementado o uso de procedimentos e funções e somente serão utilizados os tipos de variável inteiro e real. Estes recursos poderão ser agregados ao projeto futuramente, caso haja necessidade.

Em segundo lugar, a BNF deve ser construída de forma que facilite a implementação da estrutura de dados que representa o programa[6].

Outro ponto a ser considerado é a sintaxe da linguagem, que deve ser simples e permitir a estruturação do programa. Para facilitar ainda mais, a linguagem não deve diferenciar letras maiúsculas de minúsculas (não é *case sensitive*).

Por último, deve-se utilizar identificadores de palavras reservadas em português, permitindo entendimento mais intuitivo do algoritmo pelos usuários.

Assim, tem-se, de forma simplificada, a definição da linguagem:

PROGRAMA NomeDoPrograma
VARIAVEIS
    ListaDeVariáveis **COMO INTEIRO**
    ListaDeVariáveis **COMO REAL**
INICIO
    ListaDeComandos
FIM

Em ListaDeComandos podem ser encontrados comandos simples, comando condicional e/ou comandos repetitivos.

Os comandos simples são:

Variável **:=** Expressão
LEIA (ListaDeVariáveis**)**
MOSTRE (ListaDeVariáveis**)**

O comando condicional é:

SE Expressão **ENTAO**
    ListaDeComandos
SENAO
    ListaDeComandos
FIM SE

Onde não é obrigatório que haja a condição representada pelo “SENAO”.

Os comandos repetitivos são:

REPITA
    ListaDeComandos
ATE QUE Expressão

[6] Pode-se adequar a BNF durante a construção do compilador de forma que as regras de produção permitam gerar a ASD corretamente.

ENQUANTO Expressão **FACA**
    ListaDeComandos
FIM ENQUANTO

PARA Variável **DE** Expressão1 **ATE** Expressão2 **FACA**
    ListaDeComandos
FIM PARA

Onde o incremento da variável no comando “PARA” pode ser positivo ou negativo. Para o incremento positivo, utiliza-se a palavra “ATE”, caso contrário, utiliza-se a palavra “ATE-DECRESCENDO”.

A especificação completa da BNF definida para a linguagem, pode ser encontrada em anexo [Anexo B – BNF da Linguagem].

4.1.2. Árvore de Sintaxe Decorada

Antes de prosseguir com o projeto do software, é necessário também definir as estruturas de dados que irão constituir a ASD. Esta definição é básica para que se possa construir o compilador e o executor.

A Árvore de Sintaxe Decorada (ASD) [VARANDA E HENRIQUES] ou Árvore Sintática Anotada [PRICE] é uma estrutura de dados que representa a Árvore Sintática[7] de forma que possa ser implementada na memória.

A Árvore Sintática, por sua vez, é a representação gráfica da hierarquia das instruções escritas em um algoritmo ou programa.

Na figura abaixo, vê-se a Árvore Sintática e a ASD de forma simplificada para a expressão X + 5 - Y.

[7] Árvore Gramatical [AHO].

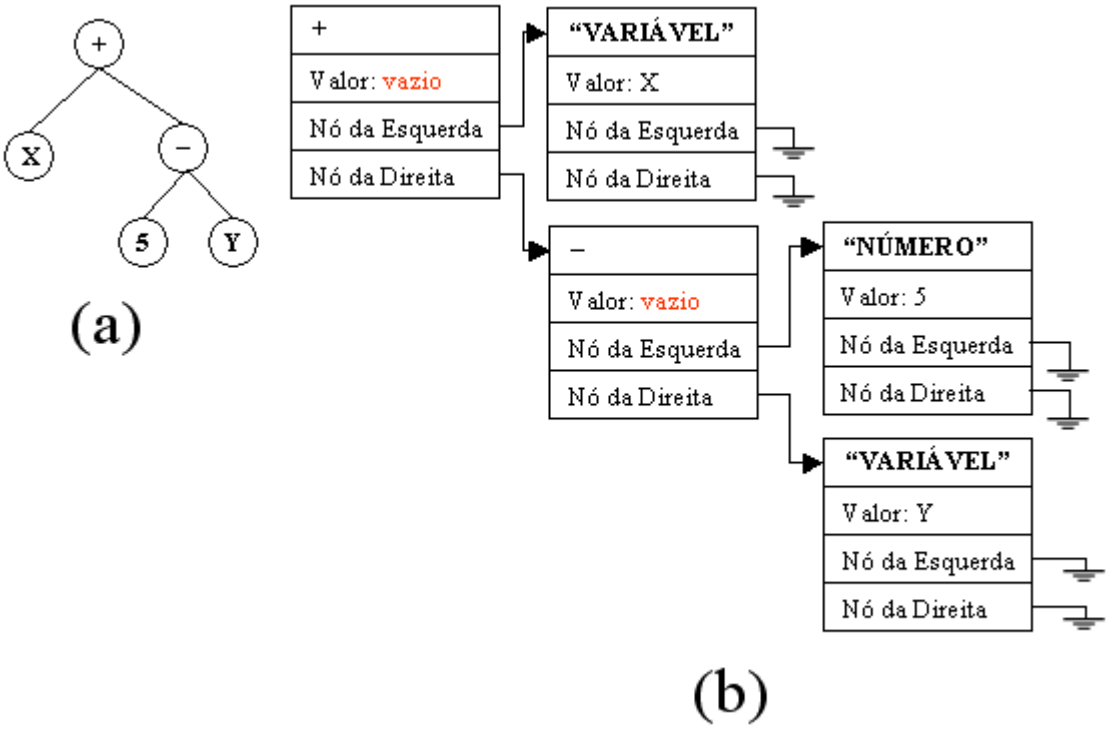

Figura 3 – Árvore Sintática (a) e ASD (b)

Na figura abaixo, é apresentada a ASD para o comando
Se A=1 Entao
Mostre(a)
Fim Se.

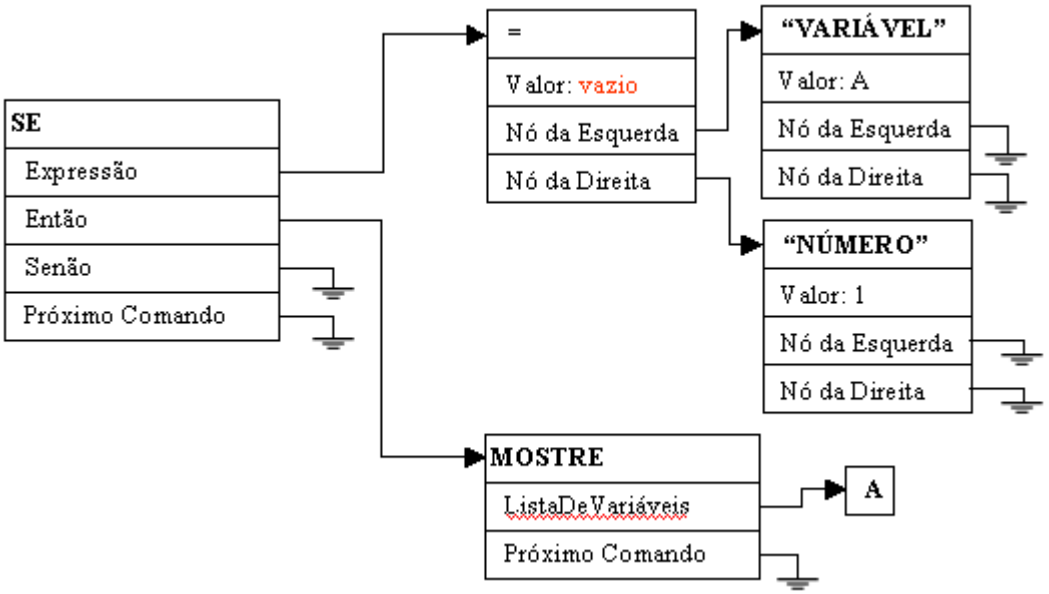

Figura 4 - ASD para um comando SE

É possível verificar então, que a ASD é composta de registros (ou estruturas) para descrever cada nó da árvore. Os registros são compostos de campos representados, nas figuras acima, pelos retângulos.

No caso de uma linguagem, deve-se diferenciar os comandos das expressões, já que cada um deles necessita de registros com campos específicos para representá-los.

Sendo assim, são definidas as estruturas para as expressões e comandos da linguagem do projeto conforme se segue.

4.1.2.1. Expressões

As expressões possuem os seguintes campos em sua estrutura:

- Operador
- Valor
- NóFilhoDaEsqueda
- NóFilhoDaDireita

Sendo que, há distinções nos campos para os nós e para as folhas (extremidades). Estas distinções podem ser verificadas no anexo [Anexo B – Árvore de Sintaxe Decorada].

4.1.2.2. Comandos

Os comandos possuem os seguintes campos em sua estrutura:

- Comando
- PróximoComando
- ListaDeVariáveis
- Variável
- Expressão
- PrimeiroComandoInterno
- PrimeiroComandoSenao
- Expressão1
- ContadorDoPara
- Expressão2

Sendo que, para cada comando, existem algumas distinções nos campos. Estas distinções podem ser verificadas no anexo [Anexo B – Árvore de Sintaxe Decorada].

4.1.3. Interface

Algumas especificações são necessárias para a interface. Ela deve conter algumas funcionalidades básicas comuns à maioria dos softwares de edição. São elas:

I) Comandos para criar novo, abrir, fechar e salvar os arquivos escritos.

II) Comandos para organizar as janelas abertas dentro da aplicação (considera-se que vários arquivos podem ser abertos para edição ao mesmo tempo).

III) Ajuda ou *Help* com a sintaxe da linguagem como referência.

No entanto, a parte central da interface é o editor, que como já foi dito, é onde o algoritmo é escrito.

O ponto crucial do editor é a marcação das palavras reservadas, símbolos e comentários da linguagem na edição. Para isso, é necessário que seja implementado um analisador léxico embutido no editor, que permita reconhecer cada palavra da entrada e

formatá-la (colorir, destacar em negrito, etc) de acordo com uma tabela de definições de formatação pré-carregada na memória. Esse analisador léxico deve ser chamado toda vez que um evento de teclado ou de mouse (copiar/colar/recortar) for interceptado, para que a marcação possa ser percebida como em tempo real.

Foi comprovado de forma empírica, com a implementação de um editor simplificado [Anexo A - Editor], que as especificações para a marcação do código são extensivas e fogem do escopo do projeto. Como já existem componentes disponíveis no mercado que realizam esta tarefa [Anexo A - SynEdit], fica a critério do desenvolvedor implementar ou não a marcação do código de forma original. Optou-se neste projeto por utilizar o SynEdit na construção do editor para diminuir o tempo de implementação.

4.1.4. Compilador

O compilador por sua vez é a parte central e mais crítica do projeto do *software*, pois é ele que permite gerar a ASD utilizada pelo executor, além de verificar a corretude do algoritmo escrito, retornando os erros para o usuário.

[AHO], [PRICE] e [MAK] descrevem os conceitos e técnicas para projetar e implementar compiladores. Estes três autores indicam que a construção de um compilador pode ser facilitada utilizando-se geradores de código para compiladores. Estes geradores de código interpretam arquivos com as especificações da linguagem e geram o código fonte correspondente ao analisador léxico e ao analisador sintático do compilador, reduzindo boa parte do trabalho braçal de programação.

No entanto, a maior vantagem de se utilizar geradores de código para compiladores é que eles facilitam a manutenção do compilador, pois permitem facilmente alterar a linguagem.

Sendo assim, foi utilizado neste projeto o *software* Glyd [Anexo A - Glyd], que é um gerador de código para compiladores compatível com o Delphi (leia Pascal).

A escolha do Glyd se deve ao fato de possuir uma interface mais amigável para escrever e compilar os arquivos de especificação necessários à geração do código do compilador. Na verdade, o Glyd funciona utilizando dois outros programas: o Lex e o Yacc, [Anexo A - LexYacc] que podem ser adquiridos separadamente e gratuitamente na Internet (assim como o Glyd). O Lex lê um arquivo contendo as expressões regulares que definem os identificadores e símbolos da linguagem, gerando o código fonte do analisador léxico. O Yacc lê um outro arquivo contendo a gramática da linguagem e as ações semânticas (utilizadas na análise semântica e na geração da ASD ou código intermediário) gerando o código fonte do analisador sintático. Para a construção do compilador, utiliza-se o código gerado pelo Lex e pelo Yacc incluindo-os no programa principal e fazendo-se chamadas a eles de forma apropriada.

4.2. *Back-End*

O *Back-End* do *software* é o executor. Conforme foi dito anteriormente, a partir da ASD gerada no *front-end*, é possível escrever um *Tree-Walker Evaluator* (TWE) capaz de fazer as animações necessárias para a visualização da execução do algoritmo e interagir com o usuário durante essa execução. Da mesma forma, o TWE pode também ser escrito para gerar um código intermediário como o MEPA [KOWALTOWSKI], por exemplo. Em fim, a escolha de como será a "interface da execução" depende apenas da aplicação e não do *front-end*.

Em se tratando de animação de algoritmos, o funcionamento do executor pode ser definido em duas etapas: Carregar a ASD e Executar a ASD.

Carregar a ASD significa percorrê-la montando o fluxograma que representa o algoritmo escrito.

Executar a ASD significa voltar ao primeiro comando (raiz da árvore) e executar passo a passo os comandos interagindo com o usuário.

Este artigo não contempla a implementação do executor animado. Entretanto, foi implementado um TWE que permite pelo menos mostrar a ASD, que seria a primeira etapa do executor. Ao carregar a ASD, os comandos são aninhados em uma árvore, e não em um fluxograma, para simplificar o projeto.

O TWE implementado percorre cada nó da árvore utilizando um procedimento que contém as instruções necessárias para mostrar o comando da forma desejada. Este procedimento é chamado recursivamente até encontrar um nó aterrado. Uma função auxiliar é utilizada para percorrer as árvores binárias das expressões quando necessário, retornando uma cadeia de caracteres (*string*). Veja abaixo, de forma resumida, como o procedimento para carregar a ASD e a função para avaliar as expressões foram implementados.

Tree-Walker Evaluator para Carregar a ASD

```
Procedimento ProcessaComando(Cmd)
inicio procedimento
        se (Cmd.Comando NãoAterrado) então
                se (Cmd.Comando = 'ATRIBUIÇÃO') então
                        Mostre('Atribuir em ')
                        Mostre(Comando.Variavel)
                        Mostre('o valor de ')
                        Mostre(ProcessaExpressao(Comando.Expressao))
                fim se
                se (Cmd.Comando = 'SE') então
                        Mostre('Se')
                        Mostre(ProcessaExpressao(Comando.Expressao))
                        Mostre('Então')
                        ProcessaComando(Comando.PrimeiroComandoInterno)
                        se (Comando.PrimeiroComandoSenao NãoAterrado)
                                Mostre('Senao')
                                ProcessaComando(Comando.PrimeiroComandoSenao)
                        Fim se
                fim se
                //Segue mesma lógica para os demais comandos.
                ProcessaComando(Cmd.PróximoComando)
        fim se
fim procedimento
```

Tree-Walker Evaluator para Avaliar Expressões

```
Função ProcessaExpressao(Expressão) como String
inicio função
        se (Expressão.Operador = 'NUMERO') entao
                RetornaComoString (Expressão.Valor)
        senao se (Expressão.Operador = 'VARIAVEL') entao
                RetornaComoString (Expressão.Valor)
        senao se (Expressão.NóDaDireita EstáAterrado) entao
```

```
            RetornaComoString (ProcessaExpressao (Expressão.NóDaEsquerda))
        senão se (Expressão.Operador = 'SOMA') entao
            RetornaComoString (
            ProcessaExpressao(Expressão.NóDaEsquerda) + '+' +
            ProcessaExpressao(Expressão.NóDaDireita)
            )
        //Segue mesma lógica para os demais operadores.
        fim se
fim função
```

5. Protótipo

O protótipo se encontra em anexo [Anexo A - AnimaAlgo].

6. Conclusão

Os objetivos iniciais foram atingidos satisfatoriamente, já que o protótipo está estável e permite trabalhar bem com algoritmos simples.

A implementação do compilador do protótipo foi bastante facilitada com o uso do Glyd (Lex/Yacc). Com ele, é possível a adição de novas funcionalidades à linguagem de forma mais simples. Inclusive, a principal funcionalidade que deve ser adicionada à linguagem é a utilização de vetores e matrizes.

Outro ponto a ser destacado é a facilidade que foi obtida com a estratégia de trabalhar com a ASD. Claramente, se verifica que a ASD gerada pode ser utilizada de várias formas: para gerar código intermediário, para executar o programa, para mostrar as instruções e para depuração (*debuging*). E mais ainda, pode-se percorrê-la no sentido do topo para baixo (*top-down*) ou de baixo para cima (*botton-up*).

O projeto deixa uma imensa base para a próxima fase, que é a construção do executor para a animação do algoritmo. Para isso, ainda é necessário definir como serão os blocos representativos de cada instrução do algoritmo no fluxograma, definir como será a interface com o usuário, como deve ser mostrado o conteúdo das variáveis e como será implementado a parte gráfica (desenhos e animações).

7. Referências

[AHO] - AHO, Alfred V., et al.; Compiladores: Princípios, Técnicas e Ferramentas. Rio de Janeiro: Editora Guanabara Koogan S.A., 1995.

[KOWALTOWSKI] - KOWALTOWSKI, Tomasz; Implementação de Linguagens de Programação. Rio de Janeiro: Editora Guanabara Dois, 1983.

[MAK] - MAK, Ronald; Writing Campilers adn Interpreters. Estados Unidos da América: Editora: Wiley Computer Publishing, 1996.

[MENDES] - MENDES, António José Nunes; Software educativo para apoio à aprendizagem de programação. http://virtual.inesc.pt/virtual/CGME99/actas/c2/, 27/05/2003.

[NEPOMUCENO] - NEPOMUCENO, Rogério Melo; BNF do Pascal Simplificado. Disciplina de Compiladores, Universidade de Uberaba. Uberaba-MG, 2002.

[PRICE] - PRICE, Ana Maria Alencar, et al.; Implementação de Linguagens de Programação. Editora Sagra-Luzzatto, 2000.

[REZENDE] - REZENDE, Pedro J. De; ASTRAL - Animation of data Structures and Algorithms. http://www.dcc.unicamp.br/~rezende/ASTRAL/, 27/05/2003.

[VARANDA E HENRIQUES] - VARANDA, Maria João, HENRIQUES, Pedro Rangel; Animação de Algoritmos Tornada Sistemática. http://virtual.inesc.pt/virtual/CGME99/actas/c2/, 27/05/2003.

ANEXO A

Implementações e Componentes utilizados no projeto

AnimaAlgo
http://www.nucleoti.com/Frederico/Compiladores/AnimaAlgo.zip

Editor
http://www.nucleoti.com/Frederico/Programas/Editor.zip

SynEdit
http://synedit.sourceforge.net/

Glyd
http://www.musikwissenschaft.uni-mainz.de/~ag/tply/Glyd_2_0.zip

LexYacc
http://www.musikwissenschaft.uni-mainz.de/~ag/tply/tply41a.zip

ANEXO B

Especificações da BNF e ASD

BNF da Linguagem

<codigo_fonte> ::= **programa** <identificador> <bloco>

<bloco> ::= <definicao_das_variáveis> **inicio** <lista_de_comandos> **fim**

<tipo> ::=
<identificador_de_tipo>

<identificador_de_tipo> ::= **inteiro** | **real**

<definicao_das_variáveis> ::=
variaveis <declaracao_de_variáveis> <mais_variáveis>
| l

<declaracao_de_variaveis> ::=
<lista_de_identificadores> **como** <tipo>

<mais_variaveis> ::=
<declaracao_de_variaveis> <mais_variaveis>
| l

<lista_de_identificadores> ::= <identificador> <mais_identificadores>

<mais_identificadores> ::= **,** <identificador> <mais_identificadores> | l

<lista_de_comandos> ::= <comando> <mais_comandos>

<mais_comandos> ::= <comando> <mais_comandos> | l

<comando> ::=
<comando_simples>
| <comando_condicional>
| <comando_repetitivo>

<comando_simples> ::=
<atribuicao>
| **leia (**<variaveis_leia>**)**
| **mostre (**<variaveis_mostre>**)**

<atribuicao ::= <variavel> **:=** <expressao>

<variaveis_leia> ::= <variavel> <mais_variaveis_leia>

<mais_variaveis_leia> ::= **,** <variaveis_leia> | l

<variaveis_mostre> ::= <variavel> <mais_variaveis_mostre>

<mais_variaveis_mostre> ::= **,** <variaveis_mostre> | l

<comando_condicional> ::=

se <expressao> **entao** <lista_de_comandos> <parte_senao> **fim se**

<parte_senao> ::= **senao** <lista_de_comandos> | l

<comando_repetitivo> ::=
enquanto <expressao> **faca** <lista_de_comandos> **fim enquanto**
| **repita** <lista_de_comandos> **ate que** <expressao>
| **para** <variavel> **de** <expressao_simples> <tipo_contador> <expressao> **faca** <lista_de_comandos> **fim para**

<tipo_contador> ::= **ate** | **ate-decrescendo**

<expressao> ::=
<expressao_simples>
| <expressao_simples> <relacao> <expressao_simples>

<relacao> ::= **<>** | **<** | **<=** | **>=** | **>** | **=**

<expressao_simples> ::=
<expressao_simples> **+** <expressao_simples>
| <expressao_simples> **-** <expressao_simplesv
| <expressao_simples> ***** <expressao_simples>
| <expressao_simples> **/** <expressao_simples>
| <fator>

<fator> ::=
<variavel>
| <numero>
| **(**<expressao>**)**

<variavel> ::= <identificador>

<numero> ::= <numero_inteiro> | <numero_real>

<numero_inteiro> ::= <digito>+

<numero_real> ::= <digito>+(**.**<digito>+)?

<identificador> ::= <letra>(<letra>|<digito>|_<letra>|_<digito>)*

<digito> ::= [**0-9**]
<letra> ::= [**a-zA-Z**]

Árvore de Sintaxe Decorada

Expressões

Campos:

- Operador
- Valor
- NóFilhoDaEsqueda

- NóFilhoDaDireita

Nos Nós

Operador: é igual a "+", "-" , "*", "/", "=", "<", ">", "<>", ">=" ou "<="
Valor: é inválido (não é utilizado).
NóFilhoDaEsqueda: aponta para o nó da esquerda.
NóFilhoDaDireita: aponta para o nó da direita.

Nas Folhas (Extremidades)

Operador: é igual a "VARIÁVEL" ou "NÚMERO"
Valor: é o valor do número ou o endereço da variável na tabela de símbolos.
NóFilhoDaEsqueda: aterrado.
NóFilhoDaDireita: aterrado.

Comandos

Campos:

- Comando
- PróximoComando
- ListaDeVariáveis
- Variável
- Expressão
- PrimeiroComandoInterno
- PrimeiroComandoSenao
- Expressão1
- ContadorDoPara
- Expressão2

Comando Atribuição (:=)

Comando: "ATRIBUIÇÃO".
PróximoComando: ponteiro para o próximo comando ou aterrado.
ListaDeVariáveis: inválido.
Variável: endereço da variável na tabela de símbolo.
Expressão: ponteiro para a expressão cujo valor deve ser atribuído à variável.
PrimeiroComandoInterno: inválido.
PrimeiroComandoSenao: inválido.
Expressão1: inválido.
ContadorDoPara: inválido.
Expressão2: inválido.

Comando Leia

Comando: "LEIA".
PróximoComando: ponteiro para o próximo comando ou aterrado.
ListaDeVariáveis: vetor com o endereço das variáveis na tabela de símbolos.
Variável: inválido.
Expressão: inválido.
PrimeiroComandoInterno: inválido.
PrimeiroComandoSenao: inválido.
Expressão1: inválido.
ContadorDoPara: inválido.

Expressão2: inválido.

Comando Mostre

Comando: "MOSTRE".
PróximoComando: ponteiro para o próximo comando ou aterrado.
ListaDeVariáveis: vetor com o endereço das variáveis na tabela de símbolos.
Variável: inválido.
Expressão: inválido.
PrimeiroComandoInterno: inválido.
PrimeiroComandoSenao: inválido.
Expressão1: inválido.
ContadorDoPara: inválido.
Expressão2: inválido.

Comando Se

Comando: "SE".
PróximoComando: ponteiro para o próximo comando ou aterrado.
ListaDeVariáveis: inválido.
Variável: inválido.
Expressão: ponteiro para a expressão a ser avaliada.
PrimeiroComandoInterno: ponteiro para o primeiro comando caso a expressão seja verdadeira.
PrimeiroComandoSenao: ponteiro para o primeiro comando caso a expressão seja falsa.
Expressão1: inválido.
ContadorDoPara: inválido.
Expressão2: inválido.

Comando Enquanto

Comando: "ENQUANTO".
PróximoComando: ponteiro para o próximo comando ou aterrado.
ListaDeVariáveis: inválido.
Variável: inválido.
Expressão: ponteiro para a expressão a ser avaliada no início do passo do *loop*.
PrimeiroComandoInterno: ponteiro para o primeiro comando caso a expressão seja verdadeira.
PrimeiroComandoSenao: inválido.
Expressão1: inválido.
ContadorDoPara: inválido.
Expressão2: inválido.

Comando Repita

Comando: "REPITA".
PróximoComando: ponteiro para o próximo comando ou aterrado.
ListaDeVariáveis: inválido.
Variável: inválido.
Expressão: ponteiro para a expressão a ser avaliada no final do passo do *loop*.
PrimeiroComandoInterno: ponteiro para o primeiro comando do *loop*.
PrimeiroComandoSenao: inválido.
Expressão1: inválido.
ContadorDoPara: inválido.

Expressão2: inválido.

Comando Para

Comando: "PARA".
PróximoComando: ponteiro para o próximo comando ou aterrado.
ListaDeVariáveis: inválido.
Variável: endereço da variável na tabela de símbolo a ser utilizada como contador do *loop*.
Expressão: inválido.
PrimeiroComandoInterno: ponteiro para o primeiro comando do *loop*.
PrimeiroComandoSenao: inválido.
Expressão1: ponteiro para a expressão que inicializa o contador.
ContadorDoPara: valor do incremento ou decremento para o contador.
Expressão2: ponteiro para a expressão a ser avaliada no início do passo do *loop*.