UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

LINGUAGEM DE PROGRAMAÇÃO DE *SCRIPT* PARA ELABORAÇÃO DE CONTEÚDO DINÂMICO NA WWW

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

FABIO EDUARDO TOMAZ

BLUMENAU, NOVEMBRO/2001

2001/2-23

LINGUAGEM DE PROGRAMAÇÃO DE *SCRIPT* PARA ELABORAÇÃO DE CONTEÚDO DINÂMICO NA WWW

FABIO EDUARDO TOMAZ

ESTE TRABALHO DE CONCLUSÃO DE CURSO FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. José Roque Voltolini da Silva — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. José Roque Voltolini da Silva

_____

Prof. Joyce Martins

_____

Prof. Dalton Solano dos Reis

AGRADECIMENTO

A Deus que proporcionou minha existência de forma saudável.

Aos meus pais que ao longo da minha vida sempre incentivaram e valorizaram a busca de conhecimento.

Ao professor José Roque Voltolini da Silva pela sua sabedoria, paciência, dedicação e incentivo à pesquisa.

A meu irmão e amigos pelo carinho e compreensão prestados e a todos que me apoiaram de forma direta ou indireta para a realização deste trabalho.

Muito obrigado.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS

ASP - *Active Server Page*

CGI - *Common Gateway Interface*

DLL – Dynamic Linker Library

DTD – Document Type Definition

HTML - *Hypertext Markup Language*

HTTP - *Hypertext Transport Protocol*

FTP – *File Transfer Protocol*

ISA – Internet Server Aplication

ISAPI - *Internet Server Aplication Programming Interface*

ISO – Internation Standards Organization

IIS - *Internet Information Server*

SGML – Standart Generalized Markup Language

URL - *Uniform Resource Locator*

YACC – *Yet Another Compiler Compiler*

WWW - *World Wide Web*

WYSIWYG – *What You See Is What You Get*

RESUMO

O presente trabalho descreve a implementação de uma linguagem de programação de *script* incorporada em páginas HTML que possibilita a criação de aplicações com conteúdo dinâmico na WWW. Um protótipo de um interpretador para a linguagem foi desenvolvido usando a técnica de análise sintática *bottom-up*, gramática de atributos e geração de código intermediário. O interpretador é apresentado em uma interface WWW, disponível através de um servidor de aplicação que utiliza a especificação ISA DLL.

ABSTRACT

The present work describes the implementation of a HTML-embedded scripting language to create applications with dynamic content in the WWW. An interpreter for the language was developed using a bottom-up parsing, grammar of attributes and generation of intermediate code. The interpreter is presented in a WWW interface, available through an application server that uses specification ISA DLL.

1 INTRODUÇÃO

Na década de 50 surgiram os primeiros compiladores. Nesta época eles eram tidos como softwares complexos. Nos tempos atuais com o avanço das pesquisas e criação de técnicas provenientes das diversas áreas da computação como engenharia de software, arquitetura de computadores e teoria das linguagens formais, os compiladores passaram a ser implementados com maior rapidez, com inúmeras melhorias desde a análise léxica até a otimização do código gerado.

Com o advento da World Wide Web (WWW) as funcionalidades dos compiladores passaram a ser utilizadas nas páginas da Internet através das linguagens de *script*. As linguagens de *script* representam uma forma totalmente nova de se programar se comparadas com as linguagens de alto nível tradicionais. Nestas linguagens de *script* é assumida a existência de um conjunto de componentes já desenvolvidos em outras linguagens, de forma que o objetivo destas linguagens passa ser o de combinar estes componentes e não o de desenvolver programas partindo de estruturas de dados elementares.

Existem linguagens de *script* que são encapsuladas em páginas HTML, tais como *Javascript* e *VBScript*. Este modelo de *script* permite aos desenvolvedores de *sites* mesclar comandos de *script* com um conteúdo de um arquivo no formato HTML. Para interpretar os *scripts* é necessário desenvolver um servidor de aplicação que esteja preparado para interagir com um servidor WWW.

Dentro deste contexto encontra-se o presente trabalho que apresenta especificações para desenvolver um servidor de aplicação para Internet. Este servidor de aplicação incorpora técnicas para implementar um programa capaz de interpretar um arquivo HTML mesclado com comandos de *script*. A definição de uma linguagem de *script* com comandos em português também é parte integrante deste trabalho.

Apesar da performance das linguagens de *script* ser inferior a das linguagens tradicionais, estas linguagens vêm se popularizando cada vez mais devido a sua facilidade de aprendizado. Nunes (1999) descreve que esta facilidade é justificada por não necessitar de muito formalismo para escrever um programa. Para desenvolver um *script* não é necessário

ser um especialista em computação. Este fato explica o crescimento acelerado de utilização e também da criação de novas linguagens de *script*.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é o desenvolvimento de uma linguagem de programação encapsulada ao código da linguagem HTML que dê subsídios para que um usuário ou programador WWW possa criar conteúdos dinâmicos em seus *sites*.

Os objetivos específicos são:

a) criar uma linguagem de *script* em português que permita a manipulação de variáveis, operadores aritméticos, operadores relacionais, comandos de fluxo de controle e funções de entrada e saída;
b) criar um servidor de aplicação que funcione como um interpretador para a linguagem proposta.

1.2 ESTRUTURA DO TRABALHO

A fundamentação teórica, descrita no capítulo 2 fornece uma visão geral da linguagem HTML, o funcionamento do protocolo de transporte de arquivos hipermídia (HTTP) e especificações mais comuns para escrever um servidor de aplicação que interaja com um servidor HTTP. As principais linguagens de *script* usadas na Internet e o funcionamento do processo de solicitação e recepção de uma página WWW também são descritos. Ainda, de uma forma bem resumida, são apresentadas considerações sobre a estrutura de um compilador e técnicas para criação de uma linguagem de programação.

O capítulo 3 segue com a especificação da linguagem de programação criada e detalhes de implementação do protótipo do interpretador da linguagem. Os diagramas de classe da especificação do interpretador e a relação dos principais algoritmos usados no trabalho também são apresentados.

A conclusão e algumas sugestões para futuras implementações deste trabalho podem ser vistas no capítulo 4.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados formas de apresentação de documentos na World Wide Web definidas através da linguagem HTML e o protocolo HTTP. Na seqüência são citados componentes relacionadas aos servidores HTTP, que são a base para construção de aplicações na Internet. Uma breve descrição das linguagens de programações de *script*, conceitos de compiladores e as técnicas para sua especificação também são apresentados.

2.1 LINGUAGEM HTML

A *Hypertext Markup Languagem* (HTML) é uma linguagem usada para apresentação de documentos hipertextos. Estes documentos estão distribuídos na WWW e são conhecidos pelos leigos como “páginas da Internet”. HTML não foi projetada para ser um formato *What You See Is What You Get* (WYSIWYG) como é o caso dos formatos de arquivos no estilo editor de texto como Microsoft© Word. Essa linguagem possui um formato de representação que permite independência de plataforma, que é possível porque as informações de um arquivo HTML são organizadas através de unidades lógicas estruturadas hierarquicamente. Este modelo de representação permite enorme flexibilidade na apresentação dos documentos, não importando se o computador exibe apenas textos, como é o caso de terminais VT-100 e ambientes MS-DOS, ou sistemas operacionais com interface totalmente gráfica.

HTML foi projetada com base na *Standart Generalized Markup Language* (SGML), que é definida pelas normas da *Internation Standards Organization* (ISO). SGML é um sistema complexo para definir tipos de estrutura de documentos para linguagens de marcação com a característica de independência de *software e hardware*. Para definir a sintaxe da linguagem HTML é usado um documento SGML chamado de *Document Type Definition* (DTD). Com um arquivo DTD pode-se utilizar programas SGML para fazer a validação da sintaxe de documentos HTML.

A diferença básica entre um texto normal digitado num editor qualquer e um HTML são as marcações chamadas *tags*. Essas marcações são identificadas através de um sinal de menor que (<), seguido de um texto que indica um comando e finalizados com o sinal de maior que (>). No exemplo de um arquivo HTML, apresentado no quadro 1, percebe-se que nesta linguagem existem diferentes elementos. Alguns elementos são chamados *empty* porque

não necessitam de um *tag* finalizador, como é o caso de *<IMG>,
 e <HR>*. Os elementos que têm uma estrutura com *tags* de início e fim têm a denominação de elementos *nesting* como o *<A HREF= >...</A> e <BODY>...</BODY>*. Outra característica de alguns tipos de *tags* é definir atributos que identificam propriedades dentro das *tags*. Um exemplo é o *<IMG SCR="foto.jpg">* que identifica a localização da figura.

Quadro 1 – EXEMPLO DE UM ARQUIVO HTML

```
<HTML>
<HEAD>
<TITLE>Pagina Exemplo</TITLE>
</HEAD>

<BODY>

<H1>Este é um titulo</H1>

<BR><BR>     <!- Aqui eu pulo duas linhas -->
<IMG SCR="foto.jpg">Esta é uma foto
<HR> <!—linha horizontal -->
<A HREF="outrapagina.html">Clique aqui</A>

</BODY>
</HTML>
```

De acordo com Mcfedries (1998) e Venetianer (1996), os recursos da linguagem HTML são diversificados. Entre as principais funcionalidades para apresentação de documentos em HTML estão:

a) formatação de texto, definição de fontes, tamanho, cores e estilos;
b) inclusão de parágrafos, quebra de linha;
c) criação de tabelas, permitindo definir cores, bordas, espaçamento entre linhas, manipulação das células;
d) inclusão de imagens, definição de fundo de página;
e) inclusão de formulários, caixa de texto, listas para entrada de dados;
f) capacidade de alinhamento de quaisquer elementos;
g) criação de documentos hipertextos através de *links*.

2.2 UNIFORM RESOURCE LOCATORS

Uniform Resource Locator (URL) é uma forma de especificar um recurso disponível na Internet, seja uma imagem, vídeo, documento texto ou qualquer outro tipo de arquivo. Uma URL é formada por uma seqüência de caracteres onde cada conjunto representa informações como descrito no quadro 2. A maioria dos protocolos da Internet, incluindo FTP,

HTTP, entre outros, utiliza URL para referenciar documentos. Em documentos HTML as *tags* de *links* e imagens são representadas na forma de uma URL.

Quadro 2 – REPRESENTAÇÃO DE UMA URL

Uma URL contém as seguintes informações:

- a primeira string da URL define o protocolo de comunicação utilizado, que pode ser *HTTP*, *WAIS*, *FTP*;
- o domínio da Internet é iniciado após as duas barras (//) e representa o *site* no qual o servidor está rodando. Este pode ser também um endereço IP;
- após os dois pontos (:) tem-se o número da porta do servidor. Este item pode ser omitido, assim o *browser* assume a porta de comunicação padrão;
- a localização do recurso é apresentada da mesma maneira de uma estrutura de diretórios.

Este é um exemplo típico de URL para o protocolo HTTP:

http://www.dominio.com.br:1092/local/pagina.html

Fonte: Graham (1996)

Os caracteres usados para escrever uma URL estão disponíveis na tabela ASCII. Alguns caracteres ASCII numa URL não são representados da forma convencional, como é o caso do espaço em branco e TAB. Estes são representados com um sinal de percentual (%) seguido de um valor de 0 a 255 em hexadecimal, respectivamente, os caracteres branco e *TAB* assumem o formato *%20* e *%09*. A barra (/) é outro caractere importante que é usado para separar a hierarquia de diretórios. O sustenido (#) indica a localização de um subitem dentro de um recurso. Em HTML esses subitens são chamados de âncoras. Por fim, o sinal de interrogação (?) indica a existência de uma *query string*, que é uma cadeia de caracteres adicional a URL que é passado para o servidor, principalmente no protocolo HTTP.

2.3 PROTOCOLO HTTP

Hypertext Transport Protocol (HTTP) é um protocolo de comunicação cliente-servidor utilizado na Internet para prover a transferência de arquivos hipertexto. HTTP é um protocolo leve, que permite ao cliente fazer uma solicitação de um conjunto de elementos hipertextos ao servidor e esses pedidos serem tratados independentemente. O servidor não guarda antigas requisições, simplesmente vai despachando pacote a medida que recebe.

Uma conexão HTTP passa por quatro estágios:

a) o cliente conecta com o servidor através de um endereço da Internet e uma porta de comunicação especificada em URL;
b) o cliente transmite uma mensagem para o servidor, solicitando um serviço. Esta mensagem enviada consiste num cabeçalho HTTP que define o método em que a transação será feita e disponibiliza informações adicionais do *browser* cliente, seguida da data de envio da mensagem;
c) o servidor envia uma resposta para o cliente, que consiste em um pacote HTTP com o status de resposta seguido pelos dados solicitados pelo cliente;
d) a conexão é encerrada.

Para a comunicação entre o cliente e o servidor o protocolo HTTP utiliza mensagens. Cada mensagem deve obrigatoriamente possuir um cabeçalho e opcionalmente ter o corpo da mensagem. No HTTP existem as mensagens de solicitação que definem regras de como o *browser* deve dialogar com o servidor e as mensagens de resposta que definem uma maneira do servidor HTTP responder as solicitações.

2.3.1 MENSAGENS DE SOLICITAÇÃO HTTP

As mensagens de solicitações consistem num cabeçalho chamado *request header*, que contém um conjunto de comandos. Cada comando é formado por uma linha de texto, que sempre termina com um retorno de carro (CR) seguido de um pulo de linha (LF). Uma linha em branco apenas com o CRLF indica o final do cabeçalho HTTP. Em seguida o pacote pode conter o corpo da mensagem. No exemplo do quadro 3 tem-se um cabeçalho HTTP que não transmite nenhum dado adicional para o servidor.

Quadro 3 – MENSAGEM HTTP TRASMITIDA PARA O SERVIDOR

```
GET /Tests/file.htmtl HTTP/1.0
Accept: text/plain
Accept: application/x-html
Accept: application/html
Accept: text/x-html
Accept: text/html
Accept: audio/*
Accept: text/x-select
Accept: */*
User-Agent: NCSA Mosaic for X Windows System/2.4 libwww/2.12 modified
```

Fonte: Graham (1996)

Um cabeçalho transmitido para o servidor é composto de duas partes. A primeira parte é definida na primeira linha da mensagem HTTP e recebe o nome de *method field*. Esta linha é formada por três elementos:

a) *HTTP_Method*: o método utilizado na conexão HTTP. Este campo define a forma de como os dados serão enviados para o servidor. Se o método for tipo GET, as informações são passadas diretamente ao fim da URL. Num método tipo POST, as informações são enviadas dentro dos campos de cabeçalho da mensagem, assim o usuário não consegue visualizar estas informações;
b) *Identifier*: é o identificador do recurso. Neste identificador é passado a URL sem o nome do protocolo e o domínio da Internet;
c) *HTTP_Version*: define qual é o protocolo e versão usado pelo cliente. A versão atual é a HTTP/1.1.

A segunda parte de um cabeçalho HTTP é composta por campos chamados *request fields*. Esses campos identificam informações sobre as capacidades do *browser*. Eles podem ser formato de arquivos, sons, imagens que o *browser* pode apresentar. A forma de representação destes formatos é definida como *MIME types*. A linha *Accept: text/plain* no exemplo do quadro 3 informa ao servidor que o cliente consegue ler arquivos formato texto. Alguns campos definem variáveis de ambiente, como na última linha do quadro 3, que carrega o nome e a versão do *browser* para a variável *User-Agent*. É comum nas linguagens de *scripts* para WWW utilizar estas variáveis de ambiente (quadro 4). Essas variáveis são pré-definidas como objetos ou constantes dentro da própria linguagem.

Quadro 4 – VARIÁVEIS DE AMBIENTE HTTP

AUTH_TYPE	tipo de autenticação usada na conexão, se houver
CONTENT_LENGTH	tamanho da mensagem passada para o servidor, excluindo o tamanho do cabeçalho
CONTENT_TYPE	tipo do documento passado para o servidor em formato de *MIME types*
GATEWAY_INTERFACE	nome da tecnologia e versão para suporte à execução de aplicativos externos ao servidor, por exemplo CGI/1.1.
PATH_INFO	parte da URL após o domínio da Internet. Normalmente um caminho de diretório

PATH_TRANSLATE	caminho de um diretório traduzido a partir do PATH_INFO, se o servidor utilizar diretórios virtuais
QUERY_STRING	informação de uma URL que vem após o sinal de interrogação (?). Podem ser campos enviados a partir de um formulário HTML
REMOTE_ADDR	endereço IP do *browser* que solicitou a conexão
REMOTE_HOST	é o nome do computador do *browser* que solicitou a conexão
REMOTE_USER	contém o nome do usuário, no caso da necessidade de uma autenticação pelo servidor
REQUEST_METHOD	método HTTP da conexão
SCRIPT_NAME	nome do programa *script* sendo executado
SERVER_NAME	endereço IP ou domínio que aparece referenciada na URL
SERVER_PORT	porta TCP/IP que recebeu a solicitação
SERVER_PROTOCOL	nome e versão do protocolo; por exemplo HTTP/1.1
SERVER_SOFTWARE	nome e versão do servidor HTTP
AUTH_PASS	senha correspondente ao usuário no caso de uma autenticação

Fonte: Graham (1996)

2.3.2 MENSAGENS DE RESPOSTA HTTP

As mensagens de resposta HTTP são bem mais simples que as mensagens de solicitação. Mensagens de repostas possuem um cabeçalho informando o status da mensagem e o corpo do documento que contém a página HTML solicitada pelo *browser*.

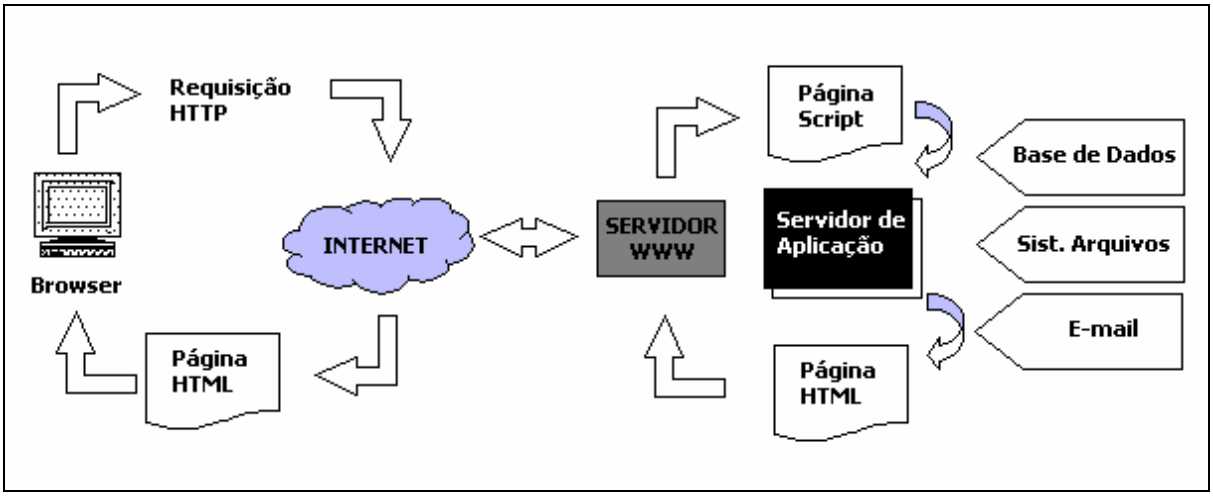
O cabeçalho de resposta é formado por *response headers fields*, campos que informam ao cliente o estado da conexão. A primeira linha deste cabeçalho é obrigatória e é formada por três elementos separados por um espaço em branco. O primeiro elemento é o *HTTP_Version*, que informa o protocolo e a versão utilizada pelo servidor. O segundo é um número que informa o estado de resposta da conexão. Este número varia de 200 a 599. Os códigos de 200 a 299 indicam sucesso na execução da transação; de 300 a 399 refletem um re-direcionamento

do documento para outra URL e códigos acima de 400 são mensagens de erro. Quando ocorre um erro, o servidor HTTP não envia todo o documento solicitado pelo cliente, apenas um pequeno documento HTML demonstrando o erro. Um exemplo de erro comum ao navegar na Internet é o erro HTTP 404, que ocorre quando o servidor não encontra o recurso solicitado. O terceiro elemento é apenas um espaço para informações adicionais.

2.4 ESPECIFICAÇÕES HTTP

Basicamente um servidor HTTP permite que sejam acessados apenas conteúdos estáticos na WWW. Para que um *site* tenha funcionalidades não disponíveis apenas com o servidor HTTP, como é o caso de acesso a banco de dados, envio de *e-mail* ou comandos de linguagem de *script* é necessário desenvolver um aplicativo externo ao servidor HTTP. Este aplicativo quando utilizado em conjunto com um servidor HTTP é chamado de servidor de aplicação para Internet. Na fig. 1 é demonstrada a seqüência de etapas percorrida para transformação de um arquivo *script* em uma página dinâmica através de um servidor de aplicação. Para escrever um servidor de aplicação para Internet é preciso seguir algumas regras que são definidas pelas diferentes especificações HTTP. A seguir são apresentadas as especificações CGI e ISA DLLS.

Figura 1 – ESQUEMA DE TRANSFERÊNCIA DE UMA PÁGINA WWW DINÂMICA



2.4.1 COMMOM GATEWAY INTERFACE (CGI)

Commom Gateway Interface (CGI) é uma especificação que define uma forma de comunicação entre o servidor HTTP e programas externos. Esta especificação permite aos

autores de páginas HTML fornecerem conteúdo dinâmico e personalizado de acordo com informações enviadas por seus usuários. Uma página dinâmica é uma página que é gerada na hora a partir de comandos de uma linguagem de programação. Isso significa que a página não precisa existir fisicamente no disco do servidor.

Quando um programa CGI é acessado através de uma URL, o servidor executa o programa que devolve como resultado um documento HTML. A forma de como é feita a passagem dos dados do servidor HTTP para o aplicativo e vice-versa são definidos pelo protocolo CGI.

A transmissão de dados do cliente para o servidor para utilização de um programa CGI pode ser feita através de uma URL, passando a localização do executável. Também é possível incluir informações adicionais diretamente para o aplicativo CGI. O exemplo do quadro 5 inclui os parâmetros *nome* e *idade* através de uma URL.

Quadro 5 – PASSAGEM DE UM CGI VIA URL

```
http://www.furb.br/cgi-bin/programa.exe?nome=Fabio&idade=22
```

Outra forma de disparar um CGI é através de um formulário HTML, utilizando a *tag* FORM. Nesta *tag* é informado o caminho do programa CGI e o método HTTP de transmissão dos dados.

A especificação CGI para enviar dados do servidor para o aplicativo é feita através de três mecanismos:

a) argumentos de linha de comando: o servidor carrega o programa passando parâmetros para o executável;
b) entrada padrão: o servidor passa dados para o programa pela entrada padrão. Num programa tradicional a entrada padrão é o teclado, no caso de um programa CGI é um arquivo texto contendo informações da mensagem de solicitação HTTP;
c) variáveis de ambiente: o servidor grava informações em variáveis especiais antes de carregar o programa, permitindo assim, o acesso durante a execução do programa.

Um CGI pode ser escrito na maioria das linguagens tradicionais (*C, Basic, Pascal, Perl*). Um programa CGI atua entre o *browser* e o servidor HTTP para produzir as informações solicitadas pelo cliente. Estas informações em conjunto com as funcionalidades

das linguagens de programação em que o programa foi escrito permitem criar rotinas não disponíveis no servidor HTTP, como enviar *e-mail*, acessar banco de dados, gravar arquivos.

A principal desvantagem do CGI é que servidor HTTP dispara um processo separado para cada solicitação, ou seja, o servidor aloca um novo processo na memória cada vez que uma URL com um programa CGI é acessada. Em servidores que possuem acessos constantes, a performance fica prejudicada devido a concorrências dos vários processos e a grande quantidade de memória RAM usada.

2.4.2 ISA DLLS

Uma forma de implementação que pode ser usada para diminuir a carga de recursos do servidor é usar DLL ao invés de executáveis. Com isso a DLL fica sempre em memória evitando o *overhead* gerado na carga de um processo.

No sistema operacional *Windows®*, a ligação dinâmica permite que um processo faça uma chamada de uma função que não faz parte do código executável. O código de máquina da função localiza-se em uma biblioteca de ligação dinâmica (DLL). A DLL pode conter uma ou várias funções que são compiladas, ligadas e armazenadas independente do programa que irá utilizá-la.

A *Microsoft Win32® Application Programming Interface* (API) utiliza bibliotecas de ligação dinâmica. Naturalmente os programas desenvolvidos para Win32 também utilizam ligação dinâmica. Existem dois métodos para chamar uma DLL:

a) *load-time dynamic linking*: ocorre quando o código de uma aplicação faz uma chamada explicita a uma função da DLL. Este tipo de ligação requer que o aplicativo importe a função externa. Deve ser informando no código fonte do aplicativo o protótipo da função a ser utilizada e o caminho onde encontrar a biblioteca quando o aplicativo é carregado;
b) *run-time dynamic linking*: ocorre quando o programa usa as chamadas da API *LoadLibrary* e *GetProcAddress* para receber o ponto inicial do código que contém a função da DLL. Este método elimina a necessidade de ligação com a importação da função externa.

Como uma alternativa aos programas CGI foi desenvolvido uma nova categoria de DLL especializa para utilização na WWW. Este tipo de DLL chama-se *Internet Server Aplications* (ISA). ISA DLL's ou ISAPI são carregadas em tempo de execução pelo servidor HTTP. Este tipo de DLL tem a característica de exportar duas funções: *GetExtensionVersion* e *HttpExtensionProc*. A função *GetExtensionVersion* recebe o número da versão da especificação usada pela DLL. *HttpExtensionProc* recebe como parâmetro uma estrutura de dados chamada *EXTENSION CONTROL BLOCK* (ECB). Um ECB disponibiliza a leitura de todas as variáveis de ambiente HTTP e os possíveis campos passados como parâmetro na solicitação de uma página.

Diferente dos *scripts* e executáveis CGI, as ISA DLL são carregadas no mesmo espaço de memória do servidor HTTP. Desta forma todos os recursos disponibilizados para o servidor HTTP estão disponíveis para a ISA DLL através da passagem de parâmetros e chamadas de função da *API*. A existência de *overheads* para novas requisições feitas ao servidor HTTP é mínima, porque o código da DLL fica ativo na memória.

2.5 LINGUAGENS DE SCRIPT NA WWW

Linguagens de s*cript* são boas para criar aplicações. Comparando com as linguagens de programação convencionais, geralmente poucas linhas de código são escritas para fazer uma tarefa. De acordo com Nunes (1999), as principais características de uma linguagem de *script* são:

a) a linguagem definida é normalmente fracamente "tipada";
b) os *scripts* da linguagem são interpretados;
c) as construções da linguagem incentivam a reutilização de código;
d) são portáveis entre diferentes plataformas.

Existem duas categorias de *scripts* utilizadas em documentos WWW: *server-side* e *client-side*. Os *scripts client-side* são interpretados pelo *browser*, que precisa ter suporte a linguagem usada. Os *scripts server-side* são interpretados por um mecanismo anexado ao servidor HTTP. Esse mecanismo pode ser um interpretador que utilize a especificação CGI ou ISA DLL.

Como os *scripts server-side* rodam no servidor, este faz todo o processamento e, na verdade, o que chega ao usuário são páginas HTML. A partir disso, pode-se chegar à conclusão que estes tipos de *scripts* são independentes dos *browsers* e que o código fonte não é visto pelo usuário solicitante da URL. Atualmente as tecnologias que utilizam linguagens de *script server-side* mais conhecidas são o *Active Server Pages* (ASP), PHP, *Cold Fusion* e *Meta-HTML*.

2.5.1 ACTIVE SERVER PAGES

Active Server Pages (ASP) foi desenvolvida pela Microsoft© para servir de apoio aos desenvolvedores de *sites* que utilizam a plataforma Windows. A tecnologia ASP é um ambiente que permite combinar HTML, *scripts* e componentes *Activex* para criar soluções WWW com conteúdo dinâmico (Weissinger, 1995).

O interpretador do ASP é independente de linguagem. Atualmente ele suporta duas das mais conhecidas linguagens de *script*: *Vbscript* e *Jscript*. Pode-se optar por qual linguagem deseja utilizar. Os *scripts* dentro de um arquivo ASP são identificados através de um delimitador especial. O delimitador de inicio é <% e o de fim é o %>. É possível declarar no interpretador ASP variáveis do tipo inteiro, ponto flutuante, moeda, data, objetos, *string* e um tipo especial chamado de tipo variável que é capaz de armazenar qualquer um dos demais tipos citados.

Para executar um arquivo ASP basta acessar uma URL de um arquivo com extensão .ASP. Com ASP é possível utilizar a maioria das funcionalidades de uma linguagem de programação tradicional. É possível manipular variáveis, utilizar comandos de fluxo de controle, comandos condicionais, declarar funções. Uma das vantagens do ASP é o seu modelo baseado em objetos, onde o desenvolvedor cria classes em qualquer linguagem de programação que utilize a especificação *ActiveX Server*, e disponibiliza para uso no ambiente ASP.

Os servidores HTTP da Microsoft disponibilizam automaticamente alguns objetos que permitem: receber informações do usuário que solicitou a página, acessar campos de um formulário HTML, acessar variáveis de ambiente, incluir texto para a página, redirecionar para outra URL, acessar base de dados e ler arquivos. O quadro 6 apresenta alguns comandos

do ASP em *VBScript*. Os servidores HTTP que suportam a tecnologia ASP são os servidores da Microsoft: *Windows NT Internet Information Server, Windows NT Workstation e Windows Personal Web Server*.

Quadro 6 – PRINCIPAIS COMANDOS ASP EM VBSCRIPT

Comando	**Descrição**	**Exemplo de sintaxe**

If	Executa um grupo de comandos dependendo do valor da condição	`If hora < 12 Then`
		`  Response.Write (“Bom dia” )`
		`ElseIf hora < 18 Then`
		`  Response.Write (“Boa tarde” )`
		`Else`
		`  Response.Write (“Boa noite”)`
		`End If`

while	Executa uma série de comandos enquanto a condição for verdadeira	`While (numero < 100)`
		`  numero = numero + 1`
		`Wend`

for	Repete um grupo de comandos um número determinado de vezes	`For i = 0 To 100 Step 2`
		`  Reponse.Write (i)`
		`Next`

do while/ until	Repete um bloco de comandos enquanto a condição for verdadeira ou até que a condição torne-se verdadeira.	`Do {While | Until} x > 0`
		`  x = x – 1`
		`Loop`

2.5.2 PHP

PHP é uma ferramenta para criar *sites* dinâmicos na WWW. PHP possui uma linguagem de *script* chamada *Zend*, que pode ser incorporada em documentos HTML. O PHP foi concebido em 1994 por Rasmus Lerdorf que distribui seu código fonte livremente na Internet e permite ser executado na maioria dos servidores HTTP disponíveis em diferentes plataformas.

No início, o PHP era um simples programa CGI escrito em *Perl*; mais tarde foi reescrito em C para melhorar seu desempenho quanto à velocidade e uso de recursos, nascia assim à versão 1.0. Até chegar a versão 4.0, o PHP passou por inúmeras melhorias deixando de ser um simples programa que substituía macros por textos e se tornar uma ferramenta para desenvolvimento de *sites* na WWW.

A sintaxe de um *script* em PHP é praticamente idêntica a da linguagem C. Quanto ao tratamento de *strings* e alocação de memória é utilizado da mesma maneira que em *Perl*. Isso foi feito devido à complexidade da linguagem C nesses itens. O tratamento da programação orientada a objetos herdou funcionalidades do Java e C++.

Em PHP, cada linha de comando é terminada com ponto-e-vírgula. As *tags* que identificam um *script* PHP num HTML são: marcador de início (<?PHP) e marcador de fechamento (?>). As variáveis no PHP são representadas por um cifrão ($) seguido pelo nome da variável. Os nomes de variáveis fazem distinção entre maiúsculas e minúsculas. O PHP suporta os seguintes tipos de variáveis: números de ponto flutuante, matrizes, inteiros, ponteiros, objetos e *strings*. O tipo da variável geralmente não é definido pelo programador; em vez disso, é decidido em tempo de execução pelo PHP, dependendo do contexto no qual a variável é usada.

Estruturas de controle num *script* PHP são construídas por blocos comandos. Um comando pode ser uma atribuição, uma chamada de função, um *loop*, um comando condicional, ou mesmo um comando que não faz nada (um comando vazio). Os comandos de fluxo de controle mais comuns são descritos no quadro 7.

Quadro 7 – PRINCIPAIS COMANDOS EM PHP

Comando	**Descrição**	**Exemplo de sintaxe**

if	A construção *if* permite a execução condicional de fragmentos de código.	`if ($hora < 12)`
		`  echo “Bom dia” ;`
		`else if ($hora > 12) Then`
		`  echo “Boa noite” ;`

while	O comando *while* é a forma mais simples de criar um *loop* em PHP.	`while ($numero < 100)` `  numero = numero + 1;`
for	O comando *for* é um laço mais complexo. Ele se comporta como o seu compatível em C. As expressões dentro do laço *for* permitem respectivamente declarar variáveis, criar uma expressão condicional e atribuir valores a variáveis.	`for ($i = 0; $i<100; $i++)` `  echo $i;`
Foreach	Permite uma maneira simples de percorrer matrizes.	`foreach (array $value)` `statement`

Switch	Semelhante ao comando *if*. Permite através de uma variável definir diferentes caminhos condicionais para a seqüência de execução do programa.	`switch ($var) {`
		`  case 0: echo “zero”; break;`
		`  case 1: echo “um”; break;`
		`}`

Uma característica significativa do PHP é seu suporte a uma ampla variedade de banco de dados. Escrever uma página que consulte um banco de dados é simples, como pode ser

visto no quadro 8. PHP tem suporte nativo aos seguintes bancos de dados: Adabas, Oracle, InterBase, PostgreSQL, FilePro (read-only), mSQL, Solid, MS-SQL, Sybase, IBM DB2, MySQL e Informix.

Quadro 8 – EXEMPLO UTILIZANDO BANCO DE DADOS EM PHP

```
$db = mysql_connect (“$host”, “$user”) or die (“Erro de Conexão”);

mysql_select_db (“$dbNome”, $db) or die (“Base de Dados não existe”);

$sql = “SELECT * FROM produto WHERE (nomProduto LIKE ‘%$trecho%’) or
(obsProduto LIKE ‘%$trecho’)”;

$resSql = mysql_query ($sql, $db);

$totProd = mysql_numrows ($resSql);
```

Fonte: Anselmo (2000)

Outras funcionalidades no PHP que podem ser citadas são a criação de funções e procedimentos, criação de classes e objetos, manipulação de ponteiros, manipulação de imagens, e suporte para comunicação em rede usando protocolos como IMAP, SNMP, NNTP, POP3 ou HTTP.

2.5.3 META-HTML

Meta-HTML é uma linguagem dinâmica para uso na WWW desenvolvida por Brian J. Fox que possui comandos de controle de fluxo e permite declarar variáveis. Como o PHP, Meta-HTML também é distribuída livremente na Internet.

Fox (1998) explica que um documento Meta-HTML consiste num texto normal, diretivas em HTML e comandos *script*. A sintaxe desta linguagem foi projetada para ser bem similar ao HTML. Meta-HTML utiliza como delimitadores os caracteres “<” e ”>”, os mesmo utilizados nas marcações em HTML.

O interpretador ao detectar a presença de um marcador, verifica se o comando lido faz parte da linguagem Meta-HTML. Se não fizer parte da sintaxe da linguagem o comando é passado como saída ao documento HTML solicitado via *browser*. Existem dois tipos de *tags* em Meta-HTML: simples e complexas. *Tags* simples iniciam com o sinal de menor que (<) e terminam com o sinal de maior que (>). O texto entre esses dois sinais é chamado de corpo do comando e contém comandos para acessar e definir valores para variáveis. *Tags* complexas

são exatamente como as *tags* simples, mas possuem uma outra *tag* simples que é responsável por fechar o bloco de comandos. O quadro 9 mostra a diferença de uma *tag* simples e uma complexa.

Quadro 9 – DIFERENÇA ENTRE TAGS SIMPLES E COMPLEXAS EM META-HTML

***tag* simples**

```
<set-var foo=bar>
```

***tag* complexa**

```
<when <get –var foo>>
  The value of foo is <get-var foo>
</when>
```

É possível inserir *tags* Meta-HTML dentro de *tags* HTML. A linguagem suporta os tipos de dados matrizes, *strings* e inteiro. Meta-HTML possui os comandos de fluxo de dados mais comuns das linguagens de *script* tradicionais (quadro 10).

Quadro 10 – PRINCIPAIS COMANDOS DE FLUXO EM META-HTML

Comando	**Descrição**	**Exemplo de sintaxe**
If	O comando *if* em Meta-HTML permite condicionar comandos. Diferente de outras linguagens, não apresenta a construção *else if*.	`<if <not <get-var name>>` `  <set-var name="all">>`

When	Permite a construção de um laço que é repetido enquanto a condição não for uma *string* vazia.	`<when TESTE>`
		`%body`
		`</when>`

foreach	Permite a construção de um laço que percorre os elementos de uma matriz.	`<foreach ELEMENTVAR ARRAYVAR`
		`[START=X] [END=X] [STEP=X]>`
		`%body`
		`</foreach>`

While	Permite executar um bloco de comandos enquanto a condição for verdadeira	`<while TEST>`
		`%body`
		`</while>`

Uma forte característica da linguagem Meta-HTML é a criação de *tags* em nível de desenvolvimento de *scripts*. Essas *tags* criadas são semelhantes às construções de procedimentos e funções numa linguagem de programação tradicional. Um exemplo da criação de uma *tag* de fatorial é apresentado no quadro 11.

Quadro 11 – CRIAÇÃO DE TAGS EM META-HTML

```
<define-function factorial num>
   <if <lt num 2> 1
      <mul num <factorial <sub num 1>>>>
</define-function>

<factorial 5> ==> 120
```

Como funcionalidades adicionais, Meta-HTML disponibiliza acesso a bancos de dados, utilização do controle de seção HTTP e criação de módulos dinâmicos que são rotinas escritas em C que podem ser utilizadas para criar *tags* que possuam tal complexidade que não possa ser desenvolvida com os *scripts* Meta-HTML.

Meta-HTML pode rodar como um CGI nos servidores *Apache Server*, *NCSA Server*, *CERN Server Plexus Server* e no *Netscape Server* através de ISA DLL.

2.5.4 COLD FUSION

Cold Fusion é um servidor de aplicação WWW que suporta múltiplas plataformas. Cold Fusion é composto por um ambiente de desenvolvimento integrado que possui um editor HTML e ferramentas para criação de *layout* de bases de dados. O ambiente Cold Fusion permite combinar páginas HTML com a linguagem de *script Cold Fusion Markup Language* (CFML).

De forma semelhante ao ASP, com Cold Fusion é possível escrever componentes que implementam rotinas com lógica de negócios e utilizar essas rotinas através de *tags* dentro do documento HTML. Cold Fusion possui cerca de 60 *tags* e 200 funções internas que fazem da CFML uma das linguagens de *script server-side* mais avançadas para criação de conteúdo dinâmico na WWW.

A sintaxe do Cold Fusion segue o padrão de uma linguagem de marcação, como o HTML. Esse tipo de linguagem permite grande produtividade para os desenvolvedores, pois com pouco esforço é possível criar poderosas aplicações. Um exemplo de um código escrito em CFML e como seu correspondente gerado em HTML é mostrado no quadro 12.

Quadro 12 –DOCUMENTO COLD FUSION ACESSANDO UM BANCO DE DADOS

Página com comandos COLD FUSION	Página HTML retornada ao browser
<CFQUERY NAME=”ProductList” DATASOURCE=”CompanyDB”> SELECT * FROM Products **</CFQUERY>** <HTML> <HEAD> <TITLE>Product List</TITLE> </HEAD> <BODY> <H1>Company X Products</H1> **<CFOUTPUT QUERY=”ProductList”>** #ProductName# $#Price# **</CFOUTPUT>** </BODY> </HTML>	<HTML> <HEAD> <TITLE>Product List</TITLE> </HEAD> <BODY> <H1>Company X Products</H1> Widgets $4.00 Diggers $5,00 Dongers $20,00 </BODY> </HTML>

2.6 LINGUAGENS FORMAIS E COMPILADORES

Dentro da área de Teoria das Linguagens Formais, encontram-se conceitos de gramática e reconhecedores, que dão base para a maioria das técnicas utilizadas hoje para se implementar compiladores. Uma gramática é um formalismo gerador, utilizado para se definir de maneira formal uma linguagem. Um reconhecedor é um dispositivo formal usado para verificar se uma determinada sentença pertence ou não à linguagem. O lingüista Noam Chomsky dividiu as linguagens em tipos dispostos em uma hierarquia, conforme abaixo:

a) irrestritas: são aquelas que nenhuma limitação é imposta. Linguagens deste tipo são aceitas por reconhecedores denominados de *Máquinas de Turing;*
b) sensíveis ao contexto: são linguagens em que as regras de substituição impõem a restrição que nenhuma substituição possa reduzir o comprimento da forma sentencial à qual é aplicada a substituição;
c) livres de contexto: conceitua-se este tipo de gramática como sendo aquelas em que é levada em consideração a substituição imposta pelas regras definidas pelas produções. Este condicionamento é eliminado impondo às produções uma restrição adicional, que restringe as produções à forma geral $A \rightarrow \alpha$, onde o lado esquerdo da produção é um não terminal isolado e α é a cadeia pela qual deve ser substituída por *A*, independente do contexto em que *A* está inserida. Este tipo de gramática pode ser reconhecido por autômatos de pilha;

d) restritas: este tipo de gramática somente admite regras de substituição de um não-terminal por uma cadeia de terminais, precedida ou não por um não terminal único. Esse tipo de linguagem pode ser expressa por uma expressão regular e são reconhecidos por autômatos finitos.

O presente trabalho enquadra-se na definição de uma gramática livre de contexto. A seguir são apresentadas as notações para especificação de uma linguagem e a descrição das fases de desenvolvimento de um compilador.

2.6.1 NOTAÇÕES PARA DEFINIÇÃO DE LINGUAGENS

As linguagens podem ser representadas através de dois mecanismos básicos: gramáticas e reconhecedores. Para ambos os mecanismos existem inúmeras formas através das quais a representação pode ser efetuada. Tais notações dá-se o nome de metalinguagens. Através de uma metalinguagem é possível descrever uma linguagem completa de uma forma bem compacta. A seguir são apresentadas as mais conhecidas formas para representação de linguagens: expressões regulares e BNF.

2.6.1.1 EXPRESSÕES REGULARES

As expressões regulares são uma das formas mais simples de representação de linguagens. As sentenças de uma expressão regular são expressas através do uso exclusivo de terminais da linguagem, sem o recurso da utilização de não-terminais. A simbologia adotada para esta metalinguagem é a seguinte:

a) concatenação (*ab):* é uma expressão regular e representa a concatenação de todos os pares ordenados obtidos pelo produto cartesiano dos conjuntos que representam as expressões *a* e *b*, nesta ordem;
b) alternância *(a | b)*: representa o conjunto união dos conjuntos representados pelas expressões regulares *a* e *b*;
c) fechamento recursivo e transitivo *(a)**: representa o conjunto obtido pela união dos conjuntos formados por todas as possíveis concatenações da expressão regular *a*, incluindo a cadeia vazia ($\in$);
d) fechamento transitivo *(a)*+: é uma expressão regular que denota o conjunto obtido eliminando-se a cadeia vazia do conjunto definido por *(a)**;

e) negação *^(a)*: utilizado por alguns autores para representar todo o alfabeto exceto o que está contido no conjunto *a*.

As expressões regulares são uma notação poderosa que pode ser utilizada para facilitar a detecção de átomos ou *tokens* na fase de análise léxica de um compilador. O quadro 13 mostra uma expressão regular que representa um número real.

Quadro 13 – EXEMPLO DE UMA EXPRESSÃO REGULAR

```
(+|-|∈) digito (.digito*)
```

2.6.1.2 BACKUS NAUR FORM (BNF)

A forma de Backus-Naur foi desenvolvida em dois esforços de pesquisa paralelos por dois homens, John Backus e Noam Chomsky. A nova notação foi ligeiramente alterada um pouco mais tarde para se descrever a linguagem ALGOL-60 por Peter Naur. Este método revisado para descrição de sintaxe passou a ser conhecido como Backus-Naur Form (BNF). BNF é uma metalinguagem para linguagens de programação.

Uma BNF é composta por um conjunto finito de regras de uma linguagem de programação. BNF usa abstrações para representar estruturas sintáticas. As abstrações na descrição BNF são freqüentemente chamadas de símbolos não-terminais. A representação para os nomes das abstrações (não-terminais) em BNF é um nome cercado pelos símbolos de menor e maior (“<” e “>”). Os itens léxicos da linguagem, também conhecidos como *lexemas* ou *tokens*, são representados por um identificador. Os itens léxicos são freqüentemente chamados terminais.

Uma regra BNF tem sempre um não-terminal em seu lado esquerdo e um conjunto composto por terminais e ou não terminais em seu lado direito. O símbolo “::=” é usado com o sentido de "é definido por" e une o lado esquerdo ao direito da regra. O símbolo | é usado com o significado de "ou" e é usado para não escrever o lado esquerdo repetidas vezes. O símbolo vazio (∈) é utilizado em BNF para representar uma cadeia vazia.

2.6.2 COMPILADORES

Como explica Aho (1995) um compilador é um software que traduz um conjunto de comandos escritos numa linguagem fonte para uma linguagem destino e que no meio do processo relata ao usuário a existência de possíveis erros no código fonte.

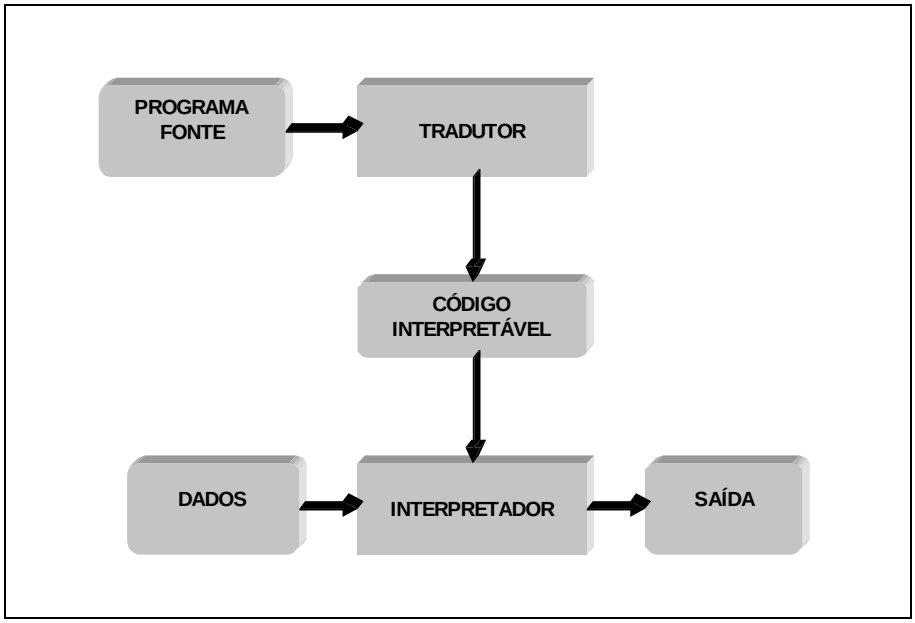
Na década de 50 surgiram os primeiros compiladores. Nesta época eles eram tidos como softwares complexos. Nos tempos atuais com o avanço das pesquisas e criação de técnicas provenientes das diversas áreas da computação como engenharia de software, arquitetura de computadores e teoria das linguagens formais, os compiladores passaram a ser implementados com maior rapidez, com inúmeras melhorias e cada vez com um nível maior de abstração para os programadores.

As primeiras linguagens introduzidas foram as linguagens de montagens, mas ainda eram muito próximas das linguagens de máquina. Num nível mais alto de abstração surgiram as linguagens de alto nível (Algol, C, Cobol, Pascal). Com a introdução de linguagens de alto nível, criou-se a necessidade de programas tradutores, isto é, sistemas que traduzissem programas escritos por programadores (*programas fonte)* para programas em linguagem de máquina (*programas objeto).*

As duas formas básicas de tradução de uma linguagem de programação são a compilação e a interpretação, que são freqüentemente usadas em combinação. Em princípio, um compilador implementa uma linguagem fonte traduzindo programas escritos nessa linguagem para a linguagem objeto da máquina alvo, onde os programas vão ser executados. Em oposição, um interpretador examina o programa fonte, e simula a execução de cada instrução ou comando de forma que o seu efeito seja reproduzido corretamente, à medida que essa execução se torna necessária.

Rangel (1999) relata que na prática, não existem compiladores ou interpretadores puros: cada compilador ou interpretador recebe o seu nome em função da forma de implementação que melhor o descreve. Muitos interpretadores efetuam uma tradução (compilação) do código fonte para uma representação interna ou código intermediário, cuja interpretação pode então ser feita com maior facilidade. A fig. 2 descreve um interpretador com tradução prévia, na qual se enquadra o protótipo deste trabalho.

Figura 2 – INTERPRETADOR COM TRADUÇÃO



Fonte: Rangel (1999)

2.6.3 ESTRUTURA DE UM COMPILADOR

De acordo com Neto (1987), os primeiros compiladores não apresentavam uniformidade na estrutura. Do ponto de vista conceitual, os compiladores apresentam-se com um número reduzido de componentes básicos, encarregados de executar funções relativamente padronizadas.

Compilar um programa é, de forma geral, ler uma seqüência de caracteres de um meio externo, reconhecer nestes caracteres as estruturas elementares da linguagem (análise léxica) e verificar se estas estruturas estão organizadas de acordo com as regras da linguagem (análise sintática). Nesta etapa, o compilador também deve ser capaz de indicar os erros de sintaxe, para que eles possam ser corrigidos posteriormente. Finalmente, o compilador deve captar o significado do texto fonte (análise semântica) para que a tradução para o programa objeto possa ser feita.

Foi pelo acúmulo de experiência, pela observação dos resultados e, principalmente, pelo desenvolvimento de teorias relacionadas à tarefa da análise e síntese de programas, que se tornou possível criar uma estruturação adequada para desenvolver compiladores. O funcionamento básico de um compilador está indicado de maneira esquemática no quadro 14.

Quadro 14 – ESTRUTURA BÁSICA DE UM COMPILADOR

ANÁLISE LÉXICA
ANÁLISE SINTÁTICA
ANÁLISE SEMÂNTICA
OTIMIZAÇÃO GLOBAL
GERAÇÃO DE CÓDIGO
OTIMIZAÇÃO LOCAL

Fonte: Neto (1987)

As funcionalidades dos módulos de análise léxica, sintática e semântica de um compilador serão apresentadas mais detalhadamente a seguir. Os demais módulos não serão aprofundados neste trabalho, que não objetiva a geração e otimização de código de máquina.

2.6.4 ANÁLISE LÉXICA

A principal função do analisador léxico é a de fragmentar o programa fonte de entrada em componentes básicos, identificando trechos elementares completos e com identidade própria, que são chamados *tokens*.

Como explica Neto (1987) para a consecução de seus objetivos, o analisador léxico executa usualmente uma série de funções, não obrigatoriamente ligadas à análise léxica, porém todas de grande importância para a correta operação das outras partes do compilador. Algumas das funções do analisador léxico são discutidas a seguir:

a) extração e classificação dos *tokens*: sua principal função é fazer um mapeamento do texto do programa fonte em outro texto, formado por *tokens*, que representam os símbolos do programa fonte;
b) eliminação de delimitadores e comentários: ainda que importante no programa fonte por razões de legibilidade para o programador, os espaços em branco, os símbolos separadores e os comentários são irrelevantes do ponto de vista de geração de código. Portanto, podem ser eliminados pelo analisador léxico;
c) identificação de palavras reservadas: sempre que for encontrado um identificador no programa fonte, é necessário verificar se ele pertence ao conjunto das palavras reservadas;

d) recuperação de erros: a ocorrência de cadeias de caracteres que não obedecem a nenhuma lei de formação das classes de *tokens* que o analisador léxico tem condição de reconhecer determina a constatação de erros léxicos no programa fonte.

2.6.5 ANÁLISE SINTÁTICA

O analisador sintático cuida exclusivamente da forma das sentenças da linguagem, baseando-se na gramática que define a linguagem. Como centralizador das atividades da compilação, o analisador sintático opera em compiladores dirigidos por sintaxe como elemento de comando da ativação dos demais módulos do compilador, efetuando decisões acerca de qual módulo deve ser ativado em cada situação da análise do programa fonte.

O analisador sintático é responsável pela recepção da seqüência de *tokens* que na etapa anterior foram extraídas do programa fonte pelo analisador léxico. A partir desta seqüência, o analisador sintático efetua uma verificação acerca da ordem de apresentação dos *tokens*, identificando em cada situação o tipo da construção sintática por eles formada, de acordo com a gramática na qual se baseia o reconhecedor. Na maioria das linguagens de alto nível, as construções sintáticas são definidas através de gramáticas livres de contexto.

De acordo com Kowaltowski (1983) os métodos da análise sintática variam bastante quanto a sua eficiência, simplicidade de implementação e aplicabilidade a linguagens específicas. Os métodos de análise sintática mais usados podem ser classificados em ascendentes (*bottom-up*) e descendentes (*top-down*). Estes termos indicam como será realizada a construção da árvore de derivação: começando pelas folhas ou pela raiz. Entende-se por árvore de derivação a representação das derivações de um gramática livre de contexto.

2.6.5.1 ANÁLISE SINTÁTICA BOTTOM-UP

O método *bottom-up* de análise sintática pode ser visto como a tentativa de construir uma árvore de derivação a partir das folhas, produzindo uma derivação mais à direita ao contrário. De acordo com Price (2001) este método é também conhecido como análise de *empilhar-reduzir*, nome dado devido ao processo que sofre a sentença de entrada, a qual é reduzida até ser atingido o símbolo inicial da gramática. Dá-se o nome de redução à operação

de substituição do lado direito de uma produção pelo não-terminal correspondente (lado esquerdo).

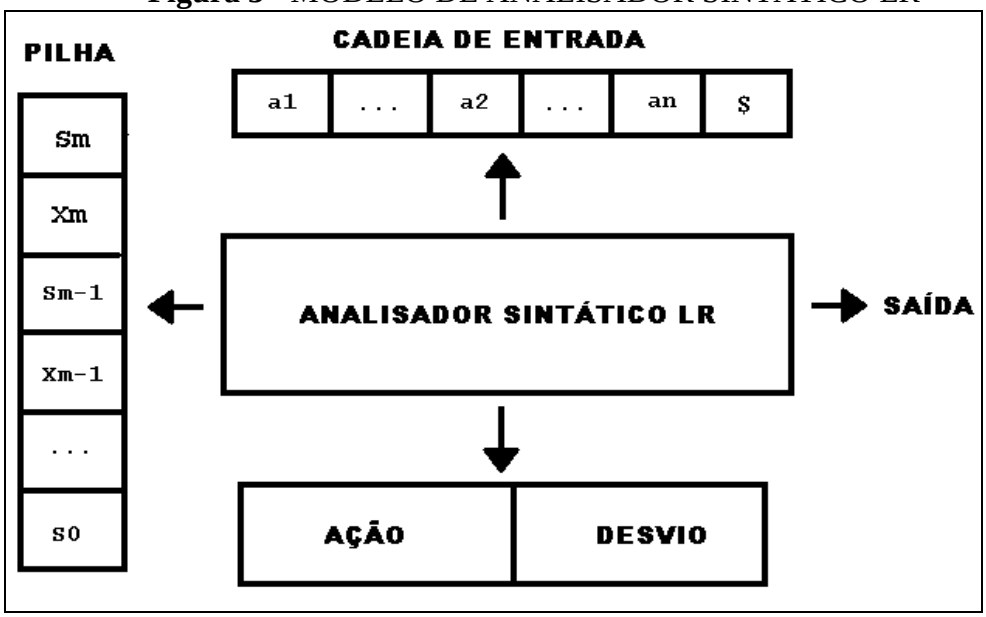
Utilizando um analisador *bottom-up,* não é necessária a eliminação prévia da recursão à esquerda na criação de uma gramática. A recursão à esquerda ocorre quando o elemento inicial da direita de uma produção é igual ao elemento definido à esquerda de uma produção.

Dentre as classes de analisadores *bottom-up* que mais reconhecem gramáticas livres de contexto, encontra-se os analisadores LR(k), onde "L" significa que a varredura da entrada é feita da esquerda para direita (*left-to-right*), o "R" significa construir uma derivação mais à direita ao contrário (*rightmost derivation*) e o "k" indica o número de símbolos de entrada que são usados para tomar decisões na análise sintática.

Um analisador LR consiste numa cadeia de entrada, uma saída, uma pilha, um programa LR e uma tabela sintática que define as possíveis ações tratadas durante a execução do analisador sintático (Figura 3).

De acordo com o símbolo de entrada e o estado identificado no topo da pilha, o analisador busca numa tabela sintática as possíveis ações a serem tomadas: empilhar um novo estado na pilha, desempilhar a quantidade de itens de uma produção, aceitar a cadeia de entrada ou notificar um erro sintático.

Figura 3 - MODELO DE ANALISADOR SINTÁTICO LR



Fonte: **Aho (1995)**

2.6.5.2 ANÁLISE SINTÁTICA TOP-DOWN

Kowaltowski (1983) relata que nos algoritmos *top-down*, a construção da árvore de derivação começa na raiz e procedem em direção as folhas. Quando todas as folhas têm seus rótulos que são terminais, a fronteira da árvore deve coincidir com a cadeia de entrada.

A análise sintática *top-down* pode ser vista como uma tentativa de se encontrar uma derivação mais à esquerda para uma cadeia de entrada. Equivalentemente, pode ser vista como uma tentativa de se construir uma árvore de derivação, para a cadeia de entrada, a partir da raiz, criando os nós da árvore gramatical em pré-ordem (Aho, 1995).

2.6.6 ANÁLISE SEMÂNTICA

O analisador semântico é responsável por detectar o restante dos erros não encontrados na análise léxica e sintática, exceto os erros de concepção e de execução do programa. A análise semântica faz uma grande quantidade de verificações, por exemplo:

a) declaração dos identificadores;
b) unicidade dos identificadores;
c) compatibilidade entre tipos numa expressão;
d) relações de herança, duplicação de métodos numa classe, no caso de linguagens orientadas a objeto.

De acordo com Monteiro (2000), a análise semântica é implementada utilizando extensões das gramáticas livres de contexto utilizadas pelos analisadores sintáticos. Estas extensões são denominadas definições dirigidas pela sintaxe e são compostas por dois tipos de extensões: atributos e regras ou ações semânticas.

Cada símbolo gramatical possui um conjunto de atributos, classificados em dois subconjuntos, chamados atributos sintetizados e atributos herdados do símbolo gramatical. Um atributo pode representar uma multiplicidade de propriedades, tais como: uma cadeia, um número, um tipo ou uma localização de memória. O valor de um atributo sintetizado é computado a partir de valores dos atributos filhos num dado nó gramatical. O valor dos atributos herdado é calculado a partir de valores dos atributos irmãos e pai de um dado nó.

As regras semânticas estabelecem dependências entre os atributos. Estas regras são cálculos ou ações executadas quando se aplica uma regra gramatical (produção) na análise sintática. O resultado da regra semântica define o valor dos atributos.

2.6.7 CÓDIGO INTERMEDIÁRIO

O código intermediário constitui uma representação do código fonte, numa seqüência de instruções, que quando executadas têm o mesmo resultado que o especificado no texto. As instruções do código intermediário são usualmente utilizadas em compiladores porque garante independência de um processador alvo.

Aho (1995) descreve que apesar de um programa fonte poder ser traduzido diretamente na linguagem alvo, existem alguns benefícios em se usar uma forma intermediária:

a) o re-direcionamento é facilitado; um compilador para uma máquina diferente pode ser escrito gerando código de máquina partindo da representação intermediária;
b) um otimizador de código independente da máquina pode ser aplicado à representação intermediária.

De acordo com Rangel (2001), a representação intermediária assume uma de duas formas: uma árvore (árvore sintática) ou uma seqüência de comandos em uma linguagem intermediária. O presente trabalho representa o código intermediário gerado através de seqüência de instruções de três endereços

Uma instrução de máquina de três endereços é composta por uma operação, dois endereços de operandos e um endereço resultado. A forma geral é x ← y op z, onde “y” e “z” representam operandos, “op’ o operador e x o resultado da instrução. Um exemplo de uma representação de um comando de atribuição (x := (a + b) * c) é mostrada no quadro 15.

Quadro 15 – REPRESENTAÇÃO INTERMEDIÁRIA DE UMA ATRIBUIÇÃO

+	a	b	temp1
*	temp1	c	temp2
:=	x	temp1	

Para a implementação deste trabalho foi criado um conjunto de instruções para gerar código intermediário. Estas instruções são geradas na execução das regras semânticas, no

momento em que uma produção é reduzida. Foram criadas instruções para manipulação de dados, instruções aritméticas, instruções de desvios condicionais e instruções de execução. O quadro 16 apresenta todas as instruções usadas para implementação do interpretador.

Quadro 16 – INSTRUÇÕES PARA GERAÇÃO DE CÓDIGO INTERMEDIÁRIO

Tipo	**Instrução**	**Descrição**
ARITMÉTICA	SOMA SUBT MULT DIVI	Soma duas variáveis na pilha Subtrai duas variáveis na pilha Multiplica duas variáveis na pilha Divide uma variável de outra na pilha
DESVIO	DSSE DSFA DSVE DSMA DSME DSIG	Desvia para uma instrução da tabela de código Desvia se o primeiro valor for falso Desvia se o primeiro valor for verdadeiro Desvia se o primeiro valor for maior que o segundo Desvia se o primeiro valor for menor que o segundo Desvia se o primeiro valor for igual o segundo
TRANFERÊNCIA	ARMZ CRVL CRCT	Armazena dados para uma variável definida Carrega dados para uma variável Carrega uma constante para pilha
EXECUÇÃO	IMPR DBEX	Imprime uma variável Manipula banco de dados ou retorna dados de campos de um banco de dados

3 DESENVOLVIMENTO DO TRABALHO

O presente trabalho é a soma da criação de três componentes fundamentais: um servidor de aplicação para Internet, uma linguagem de programação de *script* e um interpretador para a linguagem desenvolvida.

O servidor de aplicação tem a função de carregar as variáveis de ambiente HTTP e os possíveis parâmetros na solicitação de uma página WWW. A especificação deste componente do trabalho é descrita através de um diagrama de uso de caso e detalhamentos de alguns algoritmos usados na implementação.

A linguagem de programação criada utiliza algumas construções existentes em linguagens de programação bem conhecidas (Basic, Pascal). Para a especificação da linguagem são utilizados expressões regulares, a metalinguagem BNF e esquemas de tradução (utilização de atributos e regras semânticas).

Quanto ao interpretador serão apresentados as principais técnicas de construção de compiladores incorporadas no trabalho. Os algoritmos mais importantes usados no protótipo serão comentados. A especificação utiliza diagramas de classes feitos através da *Unified Modeling Languagem* (UML). Como ferramenta de apoio à modelagem do protótipo é utilizada o *Rational Rose* (Furlan, 1998).

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O trabalho consiste num servidor de aplicação para internet. Para este servidor é definida uma linguagem de programação de *script*. A interpretação de um *script* é disparada após a chamada explícita de uma URL através de um *browser*.

Na ocorrência de um erro sintático no *script* é gerada uma página HTML informando ao usuário a localização e descrição do erro.

As variáveis de ambiente e parâmetros informados nas mensagens HTTP tornam-se variáveis disponíveis para o usuário ou programador dentro da linguagem de programação criada.

A linguagem possui os tipos de dados numérico e *string*.

A linguagem de *script* permite a utilização de comandos de fluxo condicional, fluxo de controle e funções de entrada e saída. Os comandos são escritos em português.

O servidor de aplicação utiliza a especificação ISA DLL e conseqüentemente permite a execução nos servidores HTTP: *Personal Web Server* e *Internet Information Server*.

Para a implementação do interpretador é utilizada a análise sintática *botttom-up.*

3.2 ESPECIFICAÇÃO

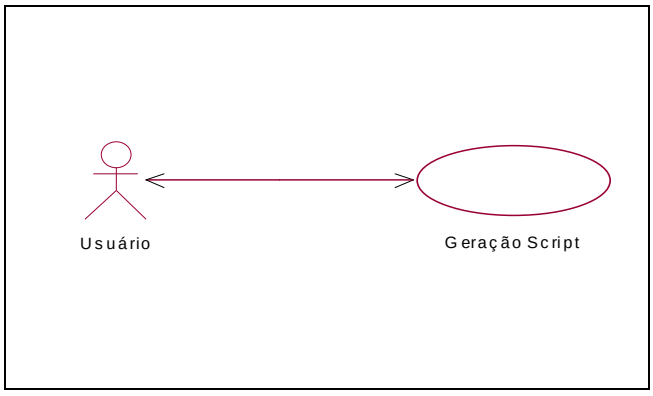
Esta seção segue com as especificações utilizadas para desenvolvimento do servidor de aplicação para internet, da linguagem de programação e do interpretador desenvolvido. Para criar os modelos e diagramas foi utilizada a ferramenta *Rational Rose*.

3.2.1 SERVIDOR DE APLICAÇÃO

O servidor de aplicação para internet que foi desenvolvido utiliza a especificação ISA DLL. Este servidor tem a particularidade de incorporar um interpretador de *script*. A função principal deste programa é gerar uma página HTML a partir de comandos de um arquivo de *script*. No caso de erros na interpretação do *script* também se gera uma página HTML com a indicação da localização do erro.

A fig. 4 apresenta o diagrama de caso de uso para o servidor de aplicação. O usuário faz a solicitação de um *script* e recebe uma página com o *script* gerado. No meio deste processo são executadas rotinas para interpretação de um arquivo *script*.

Figura 4 – DIAGRAMA DE CASO DE USO



3.2.2 LINGUAGEM DE PROGRAMAÇÃO

A especificação da linguagem de programação foi separa em três partes: análise léxica, sintática e semântica. Respectivamente foram utilizadas as técnicas de especificação através de expressões regulares, BNF e esquemas de tradução. Estas técnicas foram apresentadas na fundamentação teórica e são praticadas nesta seção.

3.2.2.1 ANÁLISE LÉXICA

A principal função do analisador léxico é agrupar os caracteres individuais do programa fonte em *tokens*, que são as unidades básicas da linguagem de programação. Os *tokens* da linguagem de programação são especificados através de expressões regulares conforme mostrado no quadro 17.

Quadro 17 – DEFINIÇÃO LÉXICA DA LINGUAGEM

```
ttIniScript    = <#
ttFimScript    =  #>
ttVirgula      = ,
ttPontoVirgula = ;
ttAtribuicao   = :=
ttSomSub       = + | -
ttMulDiv       = * | /
ttID           = ($ letra | letra) (letra | digito | _)*
ttNum          = digito (digito)*
ttString       = “ ~(“) “
ttAbrePar      = (
ttFechaPar     = )
ttOperadorE    = E
ttOperadorOU   = OU
ttOpRelacional = > | <  | =
ttComentario   = % ~(%) %
ttHtml         = ~(<#)
digito         = 0|1|2|3|4|5|6|7|8|9
letra          = A..Z | a..z
```

As palavras-chaves numa linguagem de programação são um conjunto de caracteres que não podem figurar como identificadores em uma linguagem. As palavras-chaves na linguagem são descritas em negrito na BNF apresentada no quadro 18. A declaração dos identificadores nesta linguagem é feita pelo comando de atribuição. Variáveis do tipo string iniciam com cifrão ($) e inteira sempre iniciam com uma letra.

3.2.2.2 ANÁLISE SINTÁTICA

A linguagem de programação criada é uma linguagem livre de contexto. Para especificar a linguagem foi utilizada uma gramática SLR. Este tipo de gramática não necessita utilizar mecanismos para eliminar a recursão à esquerda. No quadro 18 é apresentada em notação BNF a especificação da linguagem.

Quadro 18 – BNF DA LINGUAGEM DE SCRIPT

```
 1. <S>         ::= <HTML>
 2. <HTML>      ::= <SCRIPT>
 3. <HTML>      ::= <HTML> <SCRIPT>
 4. <HTML>      ::= ttHTML
 5. <SCRIPT>    ::= ttIniScript <LISTACMD> ttFimScript
 6. <LISTACMD>  ::= <LISTACMD> <COMANDO> ttPontoVirgula
 7. <LISTACMD>  ::= <COMANDO> ttPontoVirgula
 8. <COMANDO>   ::= ttID ttAtribuicao <EXPRE>
 9. <COMANDO>   ::= ”imprime” ttAbrePar  <EXPRE> ttFechaPar
10. <COMANDO>   ::= ”se” <CONDICAO> “então” <M> <LISTACMD> “fimse”
11. <COMANDO>   ::= ”se” <CONDICAO> “então” <M> <LISTACMD>
                    “senão” <M> <LISTACMD> “fimse”
12. <COMANDO>   ::= “enquanto” <M> <CONDICAO> “faca” <M> <LISTACMD> “fime”
13. <COMANDO>   ::= ∈
14. <EXPRE>     ::= <EXPRE> ttSomSub <TERMO>
15. <EXPRE>     ::= <TERMO>
16. <TERMO>     ::= <TERMO> ttMulDiv <FATOR>
17. <TERMO>     ::= <FATOR>
18. <FATOR>     ::= ttAbrePar  <EXPRE> ttAFechaPar
19. <FATOR>     ::= ttID
20. <FATOR>     ::= ttNum
21. <FATOR>     ::= ttString
22. <CONDICAO>  ::= <CONDICAO> ttOperadorE <M> <COND_OU>
23. <CONDICAO>  ::= <COND_OU>
24. <COND_OU>   ::= <COND_OU> ttOperadorOU <M> <COND_LOG>
25. <COND_OU>   ::= <COND_LOG>
26. <COND_LOG>  ::= <EXPRE> ttOpRelacional <EXPRE>
27. <M>         ::= ∈
28. <COMANDO>   ::= “db” ttAbrePar  ttNum <EXPRE> ttAFechaPar
29. <FATOR>     ::= “dbcampo” ttAbrePar <EXPRE> ttAFechaPar
```

Para o reconhecimento desta gramática é necessária a existência de algumas componentes e estruturas de dados: uma cadeia de entrada, uma tabela LR, uma pilha sintática e um algoritmo LR.

A cadeia de entrada é um apontador para o próximo *token.* Estes itens são reconhecidos a partir da seqüência de caracteres do arquivo fonte. Através de chamadas ao algoritmo de análise léxica este *token* é utilizado pelo analisador sintático.

De acordo com Aho (1995) são três os métodos mais conhecidos para a construção de uma tabela LR: SLR, LR canônica e LALR. A implementação utilizada no presente trabalho utiliza uma tabela SLR, onde o "S" significa simples (do inglês *simple*). O método para construção da tabela SLR é explicado na seção de implementação.

Uma tabela SLR é construída com base em uma gramática. Sua função na análise sintática é oferecer informações a fim de guiar o algoritmo LR para o reconhecimento de um arquivo fonte. Esta tabela é uma matriz bidimensional, onde as linhas são os estados e as colunas são os elementos da gramática mais o símbolo de final da cadeia. Quando as colunas desta tabela são elementos terminais as ações possíveis para o algoritmo são empilhar um novo item na pilha ou reduzir o topo da pilha através de uma produção. Quando as colunas desta tabela são elementos não-terminais é ordenado ao algoritmo desviar para outra linha. O anexo I apresenta a tabela SLR gerada para a gramática desenvolvida.

A pilha sintática dispõe de uma estrutura de dados que armazena o estado atual da tabela SLR e um número que representa um elemento da gramática. Uma observação importante é que utilizando uma pilha, ao invés de uma estrutura com toda a árvore sintática, só são mantidos os ramos da árvore que ainda não foram reconhecidos. Ao final do reconhecimento de um trecho da pilha são disparadas as regras semânticas e desempilhados os itens reconhecidos. De acordo com Pinheiro (1997) esta técnica é conhecida como "Tradução Dirigida pela Sintaxe" e é utilizada em muitos dos compiladores modernos.

Com o cruzamento do estado do topo da pilha e do *token* na cadeia de entrada, o algoritmo LR permite a execução do analisador sintático. No início do algoritmo a pilha contém o estado com valor zero e o elemento é o apontador para a cadeia de entrada. Posições

em branco na tabela SLR são considerados erros sintáticos. O quadro 19 apresenta o algoritmo para reconhecimento de uma gramática SLR.

Quadro 19 – ALGORITMO DE UM RECONHECEDOR SINTÁTICO SLR

```
Faça ip apontar para o primeiro símbolo da cadeia de entrada
Repetir
  Faça s apontar para o estado no topo da pilha e a ser apontado por ip
  Se TabelaSLR[s, a] = empilhar s' Então
    Colocar a e s' como um novo item no topo da pilha;
    Avançar ip para o próximo símbolo da cadeia de entrada;
  Senão Se TabelaSLR[s, a] = reduzir A → β Então
    Desempilhar os itens de β da pilha;
    Faça s' ser o estado atual no topo da pilha;
    Faça Tabela[s', A] ser o elemento no topo da pilha;
  Senão  Se TabelaSLR[s, a] = aceitar Então
    Retornar Ok;
  Senão
    Erro ();
  Fim
Fim
```

Fonte: **Adaptado de Aho (1995)**

3.2.2.3 ANÁLISE SEMÂNTICA

A especificação da análise semântica é composta pela definição dos atributos (quadro 20). Os atributos são armazenados na pilha sintática. Eles podem ser conhecidos na análise léxica ou no momento do reconhecimento de uma produção.

Quadro 20 – ATRIBUTOS DA LINGUAGEM DE PROGRAMAÇÃO

Atributo	**Tipo**	**Descrição**
Elemento	Inteiro	Número que identifica um t*oken* na gramática.
NomLex	String	Seqüência de caracteres associada a um identificador (passado pelo analisador léxico)
ValNum	Inteiro	Valor associado ao número (passado pelo analisador léxico)
EndVar	Inteiro	Endereço da variável declarada
IniCod	Inteiro	Armazena a localização de uma instrução
LisPrx	Lista	Lista de próximos alvos ao fim de uma bloco de comandos
LisVer	Lista	Lista de verdadeiro
LisFal	Lista	Lista de falsos

A especificação dos atributos é definida em função dos recursos que a linguagem de programação definida deve disponibilizar. Os atributos “NomLex” e “ValNum” são manipulados para o tratamento de variáveis inteiras, *strings* e constantes numéricas da linguagem. Os atributos “LisPrx”, “LisVer’ e “LisFal” são utilizados na implementação das regras semânticas no tratamento dos operadores relacionais, cálculos de desvios nas iterações e comandos condicionais definidos na linguagem.

Na análise semântica as regras são sempre disparadas no momento da redução de uma das produções da gramática. As regras semânticas constituem uma ou mais ações semânticas que manipulam os valores dos atributos para formação do código intermediário. Existe a possibilidade de na redução de uma produção não ser executada nenhuma regra gramatical. Se na execução do interpretador ocorrer à redução da primeira produção, então a análise sintática e semântica foi concluída com sucesso. No Anexo II são apresentadas as ações semânticas desenvolvidas para cada regra da gramática associada a uma produção.

3.2.3 INTERPRETADOR

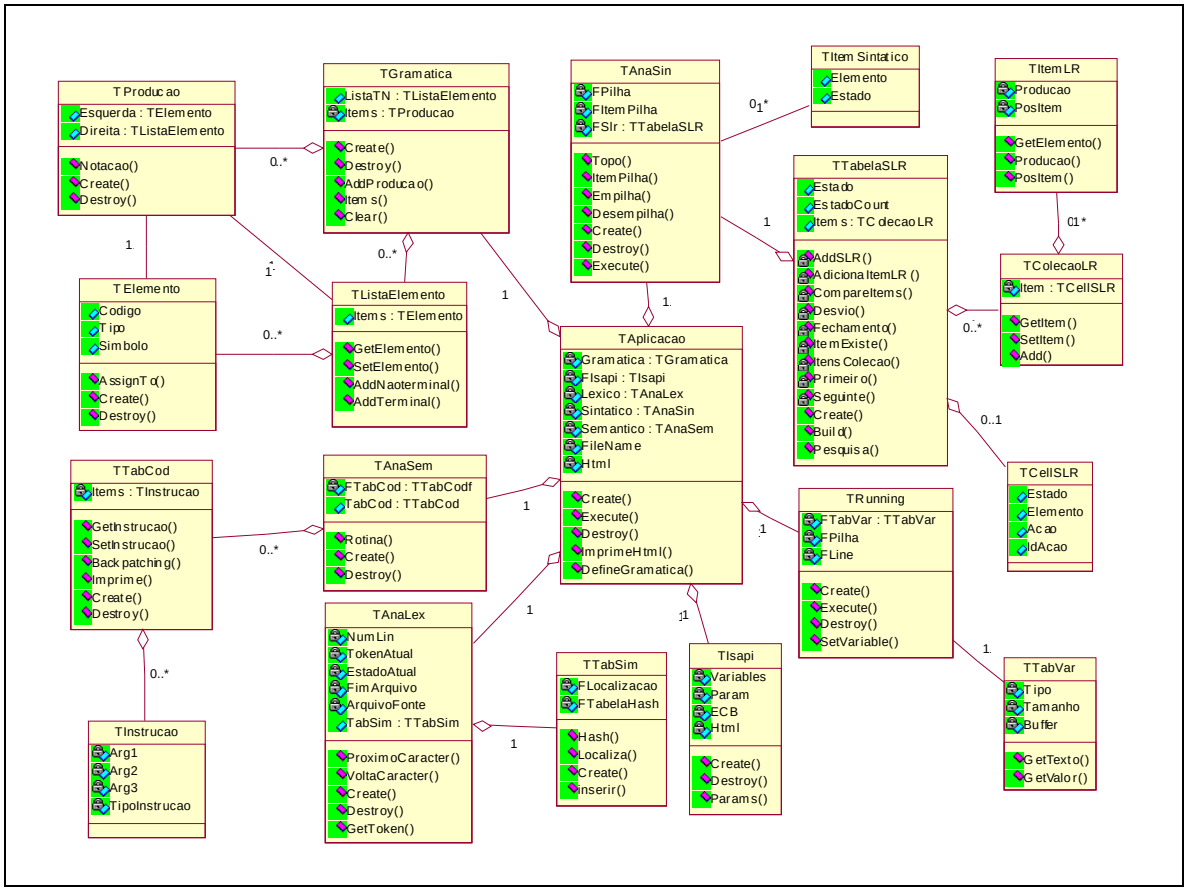
Para a especificação do interpretador é utilizada a técnica de orientação a objetos, juntamente com a UML. A principal classe do protótipo é a “TAplicacao”. Conforme apresentado na fig. 5, esta classe é responsável por instanciar os principais objetos do interpretador.

Através de uma agregação com a classe “TIsapi” o interpretador executa as principais funções do servidor de aplicação. Estas funções são imprimir informações para uma página HTML, acessar variáveis HTTP e acessar parâmetros de um formulário HTML.

A classe “TGramatica” é parte da classe “TAplicacao”. Esta classe disponibiliza propriedades que permitem acessar todas as produções definidas. Também é possível ter acesso à lista de terminais e não terminais da gramática. Uma relação do tipo “todo parte” entre as classes “TGramatica” e “TProducao” e os relacionamentos entre “TProducao” com “TListaElemento” e “TElemento” são justificados pela definição da gramática livre de contexto. Uma gramática livre de contexto contém um conjunto de produções, por sua vez cada produção possui um lado esquerdo que contém um elemento não terminal e a sua direita um conjunto de não terminais, terminais e ou o conjunto vazio.

Através da agregação de “TAplicação” com “TAnaLex” os recursos do analisador léxicos são disparados. A função “GetToken” da classe “TAnaLex” é disparada cada vez que é solicitado um novo *token* pelo analisador sintático. No caso de um *token* ser um identificador, constante numérica ou *strings* é solicitada através de uma mensagem à classe “TTabSim” a inserção de um elemento na tabela de símbolos. Esta tabela de símbolos serve como referência para localizar valores das variáveis na fase de geração de código intermediário.

Figura 5 – DIAGRAMA DE CLASSE DO INTERPRETADOR



A classe “TAnaSin” é parte da classe “TAplicacao”. Esta classe representa o analisador sintático e é responsável por direcionar o funcionamento do interpretador. A partir da chamada do construtor “Create” da classe “TAnaSin” um objeto da classe “TTabelaSLR” é instanciado. Esta classe possui toda a lógica para construção da tabela *parser* SLR que é um componente indispensável para o funcionamento do algoritmo de análise sintática.

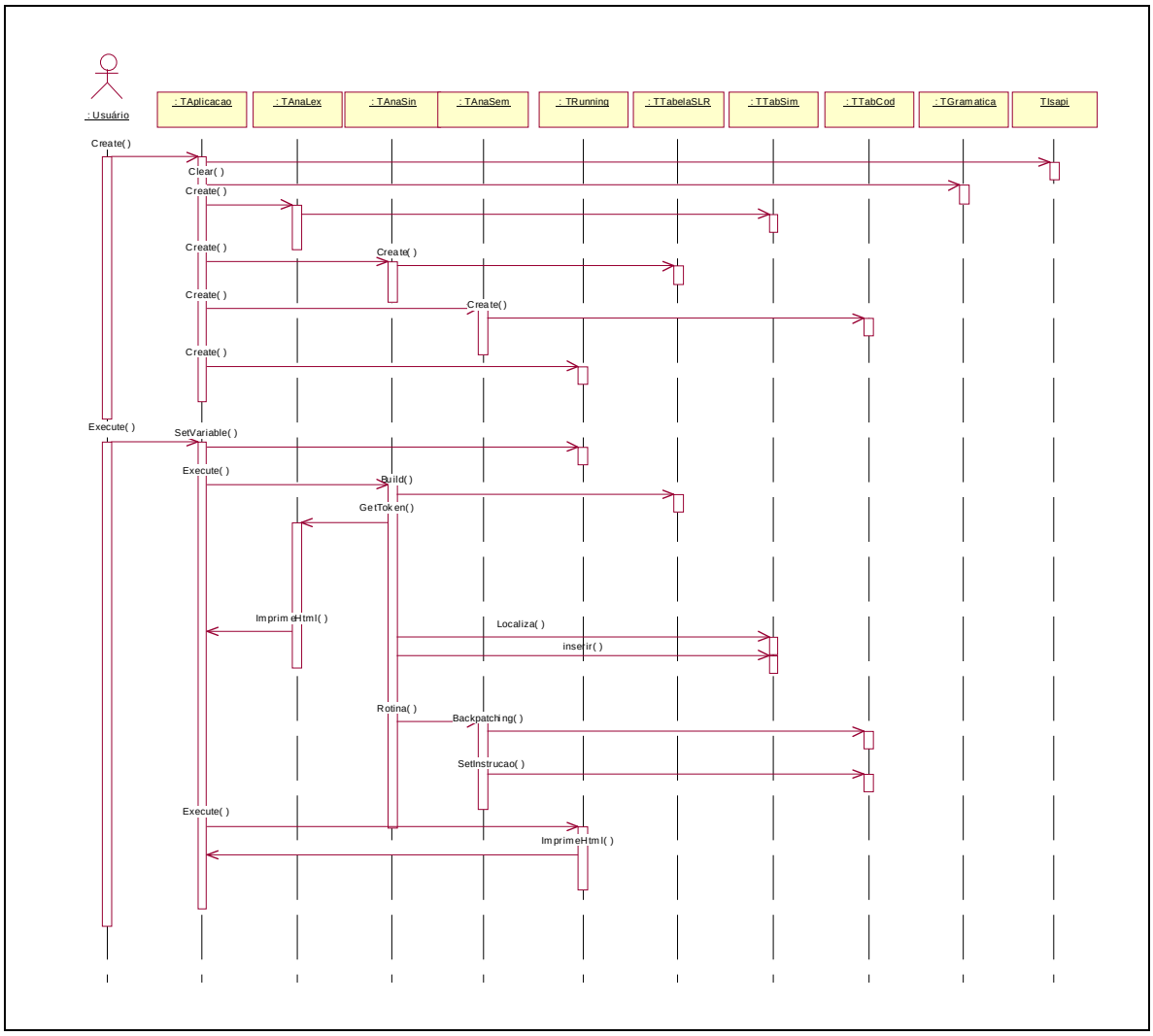
Outra importante classe que faz agregação com “TAplicacao” é a classe “TAnaSem”. Por meio desta classe são disparadas as regras semânticas. Algumas regras semânticas geram

código intermediário. A classe que armazena código intermediário é a classe “TTabCod”, onde cada instrução de código é composta por um objeto da classe “TInstrucao”.

Por fim a classe responsável pela execução do código intermediário gerado é a classe “TRunning”. Através desta classe as instruções criadas quando uma regra semântica é disparada são executadas passo a passo. As variáveis declaradas no *script* são manipuladas pela classe “TTabVar” que é um agregado de “TRunning”.

O resultado final de todo o processo de interpretação é a criação de uma página WWW a partir de um arquivo contendo comandos HTML e *script*. A interação entre os métodos das classes definidas pelo interpretador é apresentada no diagrama de seqüência da fig. 6.

Figura 6 – DIAGRAMA DE SEQÜÊNCIA DO INTERPRETADOR



3.3 IMPLEMENTAÇÃO

Informações sobre a implementação do trabalho são apresentadas nesta seção. Os principais algoritmos e técnicas são discutidos. Ao fim da seção é demonstrado um estudo de caso com base na implementação do protótipo.

3.3.1 TÉCNICAS E FERRAMENTAS UTILIZADAS

O software servidor de aplicação para Internet desenvolvido é um interpretador de *script* que permite a geração de páginas HTML dinâmicas a partir de um arquivo *script*. Este arquivo fonte contém um ou mais *scripts* (na linguagem proposta) que pode ser escrito em um simples editor de textos. Para implementação do protótipo foi utilizado o ambiente de desenvolvimento Borland Delphi (Pacheco, 2000).

A seguir são apresentadas as principais funcionalidades do protótipo, separadas em tópicos. Primeiramente será abordado o servidor de aplicação, passando pela implementação da gramática e por fim o interpretador.

3.3.1.1 SERVIDOR DE APLICAÇÃO

O servidor de aplicação para Internet desenvolvido funciona como uma interface para o interpretador. Através deste componente é possível ler o arquivo fonte a ser interpretado e retornar o arquivo alvo no fim do processo de interpretação.

Este servidor de aplicação é uma DLL que utiliza a especificação ISAPI. Um servidor HTTP que suporta esta especificação consegue disparar uma DLL porque está previsto que tanto o servidor HTTP como o servidor de aplicação, respectivamente, executa e implementa funções com nomes iguais.

A especificação ISAPI define que na implementação de uma DLL deve obrigatoriamente existir duas funções: *GetExtensionVersion* e *HttpExtensionProc.* A função *GetExtensionVersion* serve apenas para definir a versão e um nome para o servidor de aplicação. A função *HttpExtensionProc* é responsável por iniciar a execução do servidor de aplicação propriamente dito. Esta função é o ponto de partida para execução do código da DLL.

A implementação da DLL seguindo a especificação ISAPI é apresentada no quadro 21. Nesta listagem observa-se a declaração da variável “Aplicacao”. Esta variável é o principal componente do protótipo e tem a função de guiar todo o processo de interpretação.

Quadro 21 – MÓDULO PRINCIPAL DO SERVIDOR DE APLICAÇÃO

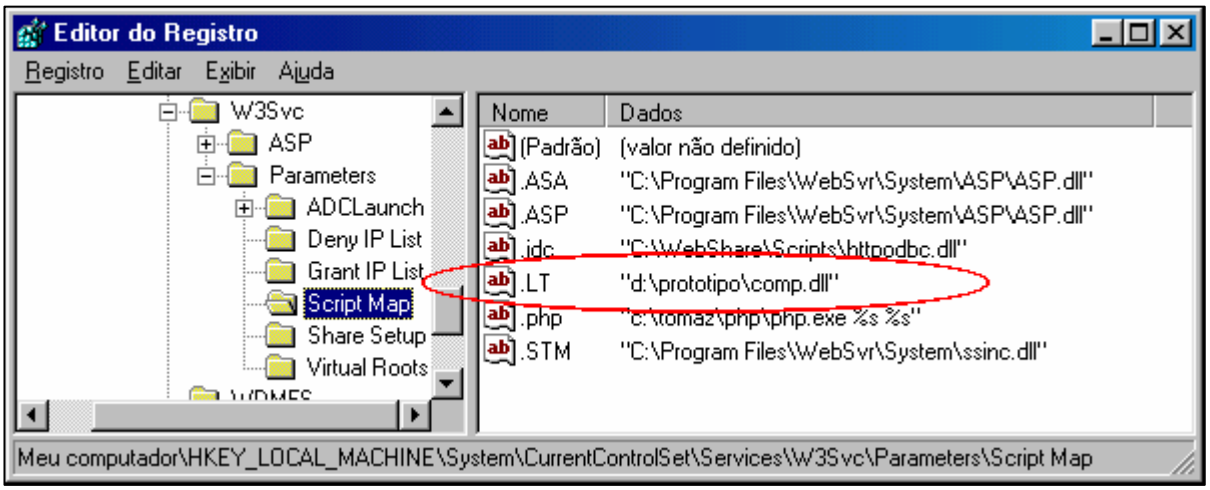
```
library comp;

uses
  SysUtils,
  Windows,
  Classes,
  Sisapi,
  Isapi,
  UAplicacao in 'UAplicacao.pas';

var
  Aplicacao: TAplicacao;

function GetExtensionVersion(var Ver: THSE_VERSION_INFO): BOOL; stdcall;
begin
  Ver.dwExtensionVersion := $00010000;  // 1.0 support
  Ver.lpszExtensionDesc := 'Protótipo TCC 2001/2 - Tomaz';
  Result := True;
end;

function HttpExtensionProc(var ECB: TEXTENSION_CONTROL_BLOCK): DWORD; stdcall;
var
  xScript: Tisapi;
begin
  try
    xScript := TIsapi.Create(ECB);
    try
      Aplicacao := TAplicacao.Create(xScript);
      try
        xScript.Clear;
        Aplicacao.Execute;
        xScript.Text := Aplicacao.Html.Text;
      finally
        Aplicacao.Destroy;
      end;
    finally
      xScript.Free;
    end;
    Result:= HSE_STATUS_SUCCESS;
  except
    Result:= HSE_STATUS_ERROR;
  end;
end;

exports
  GetExtensionVersion,
  HttpExtensionProc;
end.
```

Para esta DLL funcionar é necessário utilizar um servidor HTTP que suporte a especificação ISAPI. O servidor HTTP utilizado neste protótipo foi o *Personal Web Server*. A escolha deste software deve-se a sua simplicidade e boa documentação para permitir a depuração do servidor de aplicação.

O *Personal Web Server* dispõe de um recurso para associar uma DLL a arquivos com extensões pré-determinadas. Assim, quando for solicitada uma URL associada a tal extensão

o servidor de aplicação é carregado. No caso deste protótipo os *scripts* têm extensão ".LT" e o arquivo servidor de aplicação é denominado de "COMP.DLL". Para fazer esta associação é necessário alterar uma chave no registro do Windows (fig. 7).

Figura 7 – CONFIGURAÇÃO NO SERVIDOR HTTP



Para um *script* ser interpretado pelo servidor de aplicação é necessário que o diretório onde este arquivo está localizado permita o acesso para execução. A maioria dos servidores HTTP dispõe de uma interface para definir este tipo de configuração.

3.3.1.2 GRAMÁTICA

Com base na BNF descrita no quadro 18, foi definida a gramática da linguagem. No nível de implementação a gramática é definida através do método "DefineGramatica" da classe "TAplicacao". Neste método são criados todos os elementos terminais, não terminais e as produções que fazem parte da gramática.

O quadro 22 apresenta a implementação da gramática. A partir desta definição é criada uma estrutura que armazena as produções da gramática em uma lista de elementos que contém um lado esquerdo com um não-terminal e um lado direito com uma lista de terminais, não-terminais e ou conjunto vazio.

Quadro 22 – IMPLEMENTAÇÃO DA GRAMÁTICA

```
procedure TAplicacao.DefineGramatica;
begin
  g.Clear;
  { Não terminais }
  tnInicio   := g.ListaTN.AddNaoTerminal('<S''>');
  tnHtml     := g.ListaTN.AddNaoTerminal('<HTML>');
  tnScript   := g.ListaTN.AddNaoTerminal('<SCRIPT>');
  tnBloco    := g.ListaTN.AddNaoTerminal('<BLOCO>');
  tnListaCmd := g.ListaTN.AddNaoTerminal('<LISTACMD>');
  tnComando  := g.ListaTN.AddNaoTerminal('<COMANDO>');
```

```
  tnExpre    := g.ListaTN.AddNaoTerminal('<EXPRESS>');
  tnTermo    := g.ListaTN.AddNaoTerminal('<TERMO>');
  tnFator    := g.ListaTN.AddNaoTerminal('<FATOR>');
  tnCondicao := g.ListaTN.AddNaoTerminal('<COND_E>');
  tnCondOU   := g.ListaTN.AddNaoTerminal('<COND_OU>');
  tnCondLog  := g.ListaTN.AddNaoTerminal('<COND_LOG>');
  tnM        := g.ListaTN.AddNaoTerminal('<M>');

  { Terminais }
  ttIniScript    := g.ListaTN.AddTerminal('<#');
  ttFimScript    := g.ListaTN.AddTerminal('#>');
  ttVirgula      := g.ListaTN.AddTerminal(',');
  ttPontoVirgula := g.ListaTN.AddTerminal(';');
  ttAtribuicao   := g.ListaTN.AddTerminal(':=');
  ttIniBloco     := g.ListaTN.AddTerminal('{');
  ttFimBloco     := g.ListaTN.AddTerminal('}');
  ttSomSub       := g.ListaTN.AddTerminal('+ ou -');
  ttMulDiv       := g.ListaTN.AddTerminal('* ou /');
  ttID           := g.ListaTN.AddTerminal('id');
  ttNum          := g.ListaTN.AddTerminal('num');
  ttString       := g.ListaTN.AddTerminal('String');
  ttAbrePar      := g.ListaTN.AddTerminal('(');
  ttFechaPar     := g.ListaTN.AddTerminal(')');
  ttOperadorE    := g.ListaTN.AddTerminal('E');
  ttOperadorOU   := g.ListaTN.AddTerminal('OU');
  ttOpRelacional := g.ListaTN.AddTerminal('> = <');
  ttImprime      := g.ListaTN.AddTerminal('IMPRIME');
  ttVerdadeiro   := g.ListaTN.AddTerminal('VERDADEIRO');
  ttFalso        := g.ListaTN.AddTerminal('FALSO');
  ttSe           := g.ListaTN.AddTerminal('SE');
  ttEntao        := g.ListaTN.AddTerminal('ENTAO');
  ttSenao        := g.ListaTN.AddTerminal('SENAO');
  ttFimSe        := g.ListaTN.AddTerminal('FIMSE');
  ttEnquanto     := g.ListaTN.AddTerminal('ENQUANTO');
  ttFaca         := g.ListaTN.AddTerminal('FACA');
  ttFimEnquanto  := g.ListaTN.AddTerminal('FIME');
  ttHtml         := g.ListaTN.AddTerminal('HTML');
  ttDB           := g.ListaTN.AddTerminal('DB');
  ttDBCAMPO      := g.ListaTN.AddTerminal('DBCAMPO');
  ttEof          := g.ListaTN.AddFimArquivo;

  { Produções }
  g.AddProducao(tnInicio, [tnHtml]);
  g.AddProducao(tnHtml, [tnScript]);
  g.AddProducao(tnHtml, [tnHtml, tnScript]);
  g.AddProducao(tnScript, [ttHtml]);
  g.AddProducao(tnScript, [ttIniScript, tnListaCmd, ttFimScript]);
  g.AddProducao(tnListaCmd, [tnListaCmd, tnComando, ttPontoVirgula]);
  g.AddProducao(tnListaCmd, [tnComando, ttPontoVirgula]);
  g.AddProducao(tnComando, [ttID, ttAtribuicao, tnExpre]);
  g.AddProducao(tnComando, [ttImprime, ttAbrePar, tnExpre, ttFechaPar]);
  g.AddProducao(tnComando, [ttSe, tnCondicao, ttEntao, tnM, tnListaCmd, ttFimSe]);
  g.AddProducao(tnComando, [ttSe, tnCondicao, ttEntao, tnM, tnListaCmd, ttSenao, tnM,
tnListaCmd, ttFimSe]);
  g.AddProducao(tnComando, [ttEnquanto, tnM, tnCondicao, ttFaca, tnM, tnListaCmd,
ttFimEnquanto]);
  g.AddProducao(tnComando, []);
  g.AddProducao(tnExpre, [tnExpre, ttSomSub, tnTermo]);
  g.AddProducao(tnExpre, [tnTermo]);
  g.AddProducao(tnTermo, [tnTermo, ttMulDiv, tnFator]);
  g.AddProducao(tnTermo, [tnFator]);
  g.AddProducao(tnFator, [ttAbrePar, tnExpre, ttFechaPar]);
  g.AddProducao(tnFator, [ttID]);
  g.AddProducao(tnFator, [ttNum]);
  g.AddProducao(tnFator, [ttString]);
  g.AddProducao(tnCondicao, [tnCondicao, ttOperadorE, tnM, tnCondOU]); //21
  g.AddProducao(tnCondicao, [tnCondOU]);   //22
  g.AddProducao(tnCondOU, [tnCondOU, ttOperadorOU, tnM, tnCondLOG]); //23
  g.AddProducao(tnCondOU, [tnCondLOG]); //24
  g.AddProducao(tnCondLOG, [tnExpre, ttOpRelacional, tnExpre]); //25
  g.AddProducao(tnCondLOG, [tnExpre]); //26
  g.AddProducao(tnM, []);
  g.AddProducao(tnComando, [ttDB, ttAbrePar, ttNum, ttVirgula, tnExpre, ttFechaPar]);
  g.AddProducao(tnFator, [ttDBCampo, ttAbrePar, ttString, ttFechapar]);
end;
```

3.3.1.3 INTERPRETADOR

A partir da estrutura do servidor de aplicação e da gramática definida é necessário à compilação do *script* antes de gerar a página HTML dinâmica. As etapas apresentadas a seguir são a análise léxica, sintática, semântica e geração de código intermediário. Todas são objetos instanciados a partir da classe “TAplicacao”.

3.3.1.3.1 ANÁLISE LÉXICA

Através da classe “TAnaLex” é implementada a análise léxica do interpretador. A sua função principal é a partir de um arquivo fonte separar os *tokens* definidos pelas expressões regulares. Esta função é feita pelo método “GetToken”. O funcionamento deste método é baseado em um autômato finito, onde cada estado do autômato é um passo para o reconhecimento de um determinado *token* por uma das expressões regulares especificadas no quadro 17.

Dentro da análise léxica é feita a consulta e a inclusão de itens na tabela de símbolos. A tabela de símbolos é implementada pela classe “TTabSim”. Esta classe armazena informações como nome, tipo e a localização de identificadores declarados no programa fonte, de modo a detectar e acusar referências a variáveis não declaradas. A declaração de uma variável dentro de um programa faz com que o interpretador durante a geração de código reserve o espaço de memória necessário de acordo com seu tipo.

A localização de identificadores na tabela de símbolos é uma operação realizada inúmeras vezes durante o processo de compilação. A implementação de um algoritmo simples de pesquisa à tabela de símbolos prejudica a performance da compilação de um *script*. A fim de garantir o manuseio dos identificadores numa tabela de símbolos de forma otimizada foi utiliza neste protótipo o uso de uma tabela *Hash*.

De acordo com Aho (1995), uma tabela *hash* consiste em um *array* de tamanho fixo contendo apontadores para entradas da tabela. As entradas da tabela são organizadas em listas ligadas, chamadas de *buckets*. Um índice da tabela *hash* que não possui entradas tem seu valor nulo. No protótipo implementado a estrutura de um *bucket* armazena o nome, o tipo e a localização de um identificador.

Para determinar se existe uma entrada para uma cadeia de caracteres na tabela de símbolos, aplica-se uma *função hash* implementada na classe “TTabSim” (quadro 23). O retorno desta função é um número inteiro que representa um índice da tabela *hash*. O uso desta técnica tenta sempre encontrar um índice único para a cadeia de caracteres parametrizado na função. Caso o índice retornado for duplicado, é incluído um novo *bucket*, e este novo registro criado é apontado pelo *bucket* anterior.

Para manipular a tabela de símbolos foi implementando o método “Inserir” na classe “TTabSim”, que serve para inserir novos identificadores na tabela de símbolos. Este método é chamado pelo analisador léxico sempre que um *token* que representa um identificador é encontrado. O método “inserir” executa a função *hash* para localizar e incluir *buckets* na tabela *hash*.

Quadro 23 – FUNÇÃO HASH UTILIZADA NA TABELA DE SÍMBOLOS

```
function TTabSim.Hash(aKey: String): Integer;
var
  i: Integer;
begin
  aKey := UpperCase(aKey);
  Result := 0;
  i := 1;
  while (akey[i] <> #0) do
  begin
    Result := ((Result shl cShift) + Ord(aKey[i])) mod cTamHash;
    inc(i)
  end;
end;
```

3.3.1.3.2 ANÁLISE SINTÁTICA

A implementação da classe “TAnaSin” contém as funções de um analisador sintático, conforme especificado no quadro 19. É importante salientar que o algoritmo é sempre o mesmo independente da gramática. Uma restrição é que a gramática não seja ambígua. Para a execução deste algoritmo é necessária a construção de uma tabela *parser*.

Conforme mencionado na especificação, este protótipo implementou a construção de uma tabela *parser* (SLR) a partir da definição de uma gramática. A implementação deste algoritmo gera grande flexibilidade ao projeto que permite futuras alterações na gramática sem a necessidade de mudanças radicais no interpretador.

A construção da tabela SLR é implementada seguindo conceitos descritos em Aho (1995) e Kowaltowski (1983). Ambos autores descrevem o conceito de item na construção da

tabela SLR com sendo uma produção na qual é marcado um ponto (representado por •) em uma posição do lado direito. Para representar um item é utilizado um par de inteiro, onde o primeiro fornece o número da produção e o segundo a posição do ponto.

No quadro 24 é mostrado o conjunto dos itens derivados a partir de uma gramática simplificada. Este conjunto de itens denomina-se coleção canônica e sempre representa um conjunto finito. Os itens agrupados formam os estados (E0, E1, E2,...), que são as linhas numa tabela SLR.

Quadro 24 – COLEÇÃO CANÔNICA A PARTIR DE UMA GRAMÁTICA

Seja a gramática para expressões e sua respectiva coleção canônica:

```
 0.  E'→ E#      (# representa o fim de arquivo)
 1.  E → E + T
 2.  E → T
 3.  T → T * F
 4.  T → F
 5.  F → (E)
 6.  F → id
```

E0: E'→ •E# E → •E+T E → •T T → •T*F T → •F F → •(E) F → •id E1: E'→ E•# E → E•+T E2: E → T• E → T•+F	E3: T → F• E4: F → (•E) E → •E+T E → •T T → •E+T T → •F F → •(E) F → •id E5: F → •id E6: E → E+•T T → •T*F	T → •F F → •(E) F → •id E7: T → T*•F F → •(E) F → •id E8: F → (E•) E → E•+T E9: E → E+T• E10: T → T•*F E11: F → (E)•

O símbolo • indica quanto já foi processado de uma produção. Quando este símbolo está no fim de uma produção ($A \rightarrow \alpha\beta\bullet$), significa que esta foi reconhecida. Se um marcador estiver na forma "($A \rightarrow \alpha\bullet B\beta$)", e B é um não-terminal, é preciso ainda percorrer todos os derivados de "B" antes de chegar ao item "($A \rightarrow \alpha B\bullet\beta$)".

Alguns métodos implementados na classe "TTabelaSLR" são bastante úteis na construção de uma tabela parser: "Fechamento", "Desvio", "Seguinte". O algoritmo dessas funções é utilizado em geradores de analisadores sintáticos *bottom-up*, como é o caso do YACC (Aho, 1995). A função "Fechamento (I)" consiste em determinar a coleção de itens partindo de uma coleção de itens "I" que constitui o próximo estado da coleção canônica de uma determinada gramática. "Desvio (I, X)" consiste em determinar a coleção de itens que, em relação a uma gramática, podem figurar após um símbolo gramatical "X". Por exemplo,

encontrar todos os itens que numa dada gramática aparecerem após o sinal de atribuição (:=). A função “Seguinte (A)”, para o não terminal “A”, é definida como sendo o conjunto de elementos terminais da gramática que podem figurar imediatamente à direita de “A” em alguma forma sentencial.

A construção da coleção canônica é implementa neste protótipo através do método “ItensColecao” da classe “TTabelaSLR”. Este método cria todos os estados da coleção canônica partindo do primeiro item da primeira produção da gramática (E’→ •E#). Com o auxílio dos métodos “Fechamento” e “Desvio” é disparado um algoritmo que faz a construção da coleção canônica.

Com a coleção canônica completa é preciso preencher a tabela SLR com as possíveis ações: empilhar um elemento, desviar para outro estado, aceitar a entrada. O preenchimento desta tabela é implementado pelo método público “Build” da classe “TTabelaSLR”. O algoritmo percorre todos os itens construídos a partir da gramática partindo do estado “E0” até “Ei”. A tabela SLR é preenchida conforme as seguintes regras:

a) se [A → α•aβ] estiver em *Ei* e *desvio*(*Ei*, *a*) = *Ej*, então preencher a tabela SLR[*i*,*a*] com a ação “empilhar j”;
b) se [A → α•] estiver em *Ei*, então preencher a tabela SLR[*i*,*a*] com a ação “reduzir através da regra A → α”, para todo *a* em *Seguinte*(*A*) (aqui *A* não pode ser o estado inicial da gramática - S’)
c) se [S’ → S•] estiver em *Ei*, então preencher a tabela SLR[*i*,*a*] com a ação “aceitar a cadeia de entrada”.

Após a execução deste algoritmo e usando uma gramática de acordo com o quadro 24, é gerada uma tabela SLR como apresentado no quadro 25. Cada célula desta tabela tem um significado. Células em branco significam erros sintáticos. As demais células têm um significado conforme segue:

a) “*x*” - desvio para outro estado tabela, indicado por *x*;
b) “s*i*” - empilhar símbolo de entrada e o estado *i;*
c) “r*j*” - reduzir através da produção *j;*
d) “acc” - aceitar a cadeia.

Quadro 25 – EXEMPLO DE UMA TABELA SLR

Estados/ Elementos	E	T	F	id	+	*	(	)	#
0	1	2	3	s5			s4		
1					s6				acc
2					r2	s7		r2	r2
3					r4	r4		r4	r4
4	8	2	3	s5			s4		
5					r6	r6		r6	r6
6		9	3	s5			s4		
7			10	s5			s4		
8					s6			s11	
9					r1	s7		r1	r1
10					r3	r3		r3	r3
11					r5	r5		r5	r5

3.3.1.3.3 ANÁLISE SEMÂNTICA

Na fase de análise semântica é implementado um conjunto de métodos, definidos na classe “TAnaSem” que representam as regras semânticas. Os comandos contidos em cada um destes métodos manipulam os atributos semânticos com objetivo final de gerar código intermediário.

Uma regra semântica é executada quando o analisador sintático consegue reduzir uma produção. A redução ocorre quando todos os elementos do lado direito de uma produção são reconhecidos. Como os elementos do lado direito representam um elemento não-terminal correspondente (lado esquerdo), conclui-se que uma regra semântica está diretamente relacionada a uma produção.

Num analisador *bottom-up* só é possível executar regras semânticas quando é feita a redução de uma produção. Na gramática desenvolvida é necessário executar regras semânticas no meio das produções. Para possibilitar a implementação é utilizado um artifício. Este artifício, descrito em Aho (1995), consiste em acrescentar novos não-terminais no meio de uma produção. O quadro 26 apresenta a implementação do comando “SE/ENTAO” da linguagem proposta. O não terminal “<M>” foi incluído na gramática com objetivo de guardar o valor do atributo herdado “IniCod”. Este atributo é usado para desviar a seqüência de execução de código para determinada posição caso a condição do comando “SE/ENTAO” for verdadeira.

Quadro 26 – EXEMPLO DE UTILIZAÇÃO DE UM ARTIFÍCIO PARA EXECUTAR REGRAS NO MEIO DE UMA PRODUÇÃO

```
<COMANDO> ::= "se" <CONDICAO> "então" <M> <LISTACMD> "fimse"
  FTabCod.Backpatching(ItemPilha(4).LisVer, ItemPilha(2).IniCod);
  FTopo.LisPrx.AddLista(ItemPilha(4).LisFal);
  FTopo.LisPrx.AddLista(ItemPilha(1).LisPrx);

<M> ::= ∈
  Ftopo.IniCod := FTabCod.Count;
```

Os atributos semânticos (quadro 20) são armazenados na pilha sintática. Quando um novo item da pilha sintática é inserido alguns atributos já podem ser conhecidos. No caso de uma variável é carregado o valor do atributo "EndVar" (endereço da variável na memória) e "TipVar" (tipo da variável). No caso de uma constante do tipo *string* o atributo "ValStr" recebe um texto.

Todos os itens que guardam os atributos da pilha sintática são retirados após a execução da rotina semântica. Assim, apenas os atributos do trecho que está sendo reconhecido são armazenados. O mesmo ocorre na execução das regras gramaticais, onde na pilha sintática apenas ficam disponíveis os atributos dos terminais e não-terminais da produção que está sendo reconhecida.

O Anexo II apresenta a implementação das regras semânticas. Nesta implementação são manipulados os atributos contidos na pilha sintática e adicionados comandos para geração de código intermediário. A pilha sintática manipula o topo da pilha através da propriedade "Topo" da classe "TAnaSin". A geração de código é disparada pelas rotinas semânticas através de métodos do objeto "FTabCod" instanciado na classe "TAnaSem".

3.3.1.3.4 GERAÇÃO DE CÓDIGO INTERMEDIÁRIO

A implementação das rotinas de geração de código utiliza *quádruplas* para representar cada linha de código intermediário. Uma *quádrupla* é uma estrutura de dados contendo quatro campos, os quais são chamados de *tipins*, *arg1*, *arg2* e *arg3*. O campo *tipins* armazena o código interno de uma instrução e os demais campos são normalmente apontadores para tabela de símbolos ou valores específicos para cada tipo de instrução. Uma instrução de adição (2 + 3) gera três *quádruplas*. As duas primeiras para carregarem e empilharem as constantes numéricas 2 e 3 (*CRCT, 2, 0, 0 e* CRCT, 3, 0, 0) e a última para somar os números

através da instrução “SOMA” que irá somar os dois itens anteriormente empilhados e desempilhar um item deixando o resultado no topo da pilha.

É comum utilizar duas passagens na implementação de compiladores. Primeiramente é construída uma árvore sintática para a entrada e, em seguida, caminha-se na árvore numa ordem busca em profundidade, executando as regras semânticas dadas na definição. O principal problema de gerar código em uma única passagem é que durante a tradução de uma expressão booleana ou comando de fluxo de controle ainda não é conhecido os alvos (número da linha de código intermediário) para onde os desvio são gerados. Para solucionar este problema é utilizada neste protótipo a técnica de *backpatching*.

A técnica de *backpatching,* descrita em Aho (1995) e Price (2001), consiste em deixar as informações dos campos de uma instrução com valores indefinidos para preencher futuramente. Normalmente o momento para o preenchimento das linhas de código indefinidas ocorre na redução de regras de operações booleanas e comandos de desvios. Para implementar esta técnica são utilizadas listas de inteiros que guardam as linhas ainda não conhecidas na tradução de uma expressão booleana e ou comando de desvio. Cada item da lista aponta para uma linha do código intermediário gerado. As listas usadas neste protótipo são os atributos definidos na especificação da análise semântica:

a) *LisVer* – armazena um valor que representa as linhas de código intermediário que figuram em expressões verdadeiras;
b) *LisFal* – armazena um valor que representa as linhas de código intermediário que figuram em expressões falsas.
c) *LisPrx* – usado em comandos de desvios para armazenar os valores que indicam a linha do próximo comando a ser executado.

Para manipular os valores destas listas é utilizado o método “Concatena” que faz a junção de duas listas e o método “Backpatching” que preenche o valor do campo *arg3*, em uma instrução de desvio, com o número de uma linha especificada da tabela de código. Ambos os métodos são implementados na classe “TTabCod”.

Um conceito abordado em Rangel (1999) e utilizado neste trabalho é a geração de código de *curto-circuito* para avaliação de expressões booleanas. Neste tipo de geração de código, o interpretador ao encontrar uma expressão do tipo “a > b OU c = d E a > d”, analisa

primeiramente a expressão “a > b”, se esta for verdadeira, não há necessidade de analisar a expressão “c = d”, então a próxima expressão a ser analisada é “a > d”. Partindo deste princípio, em uma condição “ou”, deve-se gerar uma linha de código entre as duas expressões prevendo este desvio. A representação desta expressão é apresentada no quadro 27.

Quadro 27 – CÓDIGO INTERMEDIÁRIO PARA EXPRESSÃO a > b OU c = d E a > d

```
Linha   Quádupla
020  => DSMA 000 000 024 // desvia se a > b para linha 24
021  => DSSE 000 000 022 // desvia sempre para linha 22
022  => DSIG 000 000 024 // desvia se c = d para linha 24
023  => DSSE 000 000 027 // desvia sempre para linha 27   (expressão verdadeira)
024  => DSMA 000 000 026 // desvia se a > d para linha 26 (expressão falsa)
025  => DSSE 000 000 027 // desvia sempre para linha 27   (expressão verdadeira)
```

A geração de código de *curto-circuito* também é utilizada em comandos da linguagem de programação desenvolvida (SE/SENAO, ENQUANTO). Estes comandos utilizam desvios dependendo do resultado da condição do comando. Por exemplo, no comando SE/SENAO, se a condição for verdadeira executa-se lista de comandos “SE”, caso contrário executa-se os comandos abaixo do “SENAO”. O quadro 28 apresenta um código de *script* usando o comando “SE/SENAO” juntamente com o código intermediário gerado.

Quadro 28 –CÓDIGO INTERMEDIÁRIO GERADO PARA O COMANDO “SE/SENAO”

```
A VARIÁVEL "a" É
<#
  a := 5;
  SE a > 3 ENTAO
    imprime ("MAIOR ");
  SENAO
    imprime ("MENOR ");
  FIMSE;
#>
QUE 5
```

```
000  => CRVL 000 000 000
001  => IMPR 000 020 000
002  => DSSE 000 000 003
003  => CRCT 005
004  => ARMZ 001 000 000
005  => DSSE 000 000 006
006  => CRVL 001 000 000 // carrega a
007  => CRCT 003         // carrega 3
008  => DSMA 000 000 010 // se a > 3 desvia (linha 10)
009  => DSSE 000 000 013
010  => CRVL 004 000 000 // imprime (“maior ”)
011  => IMPR 004 020 000
012  => DSSE 000 000 016
013  => CRVL 005 000 000 // imprime (“menor ”)
014  => IMPR 005 020 000
015  => DSSE 000 000 016 // fim se
016  => CRVL 006 000 000
017  => IMPR 006 020 000
018  => DSSE 000 000 019
```

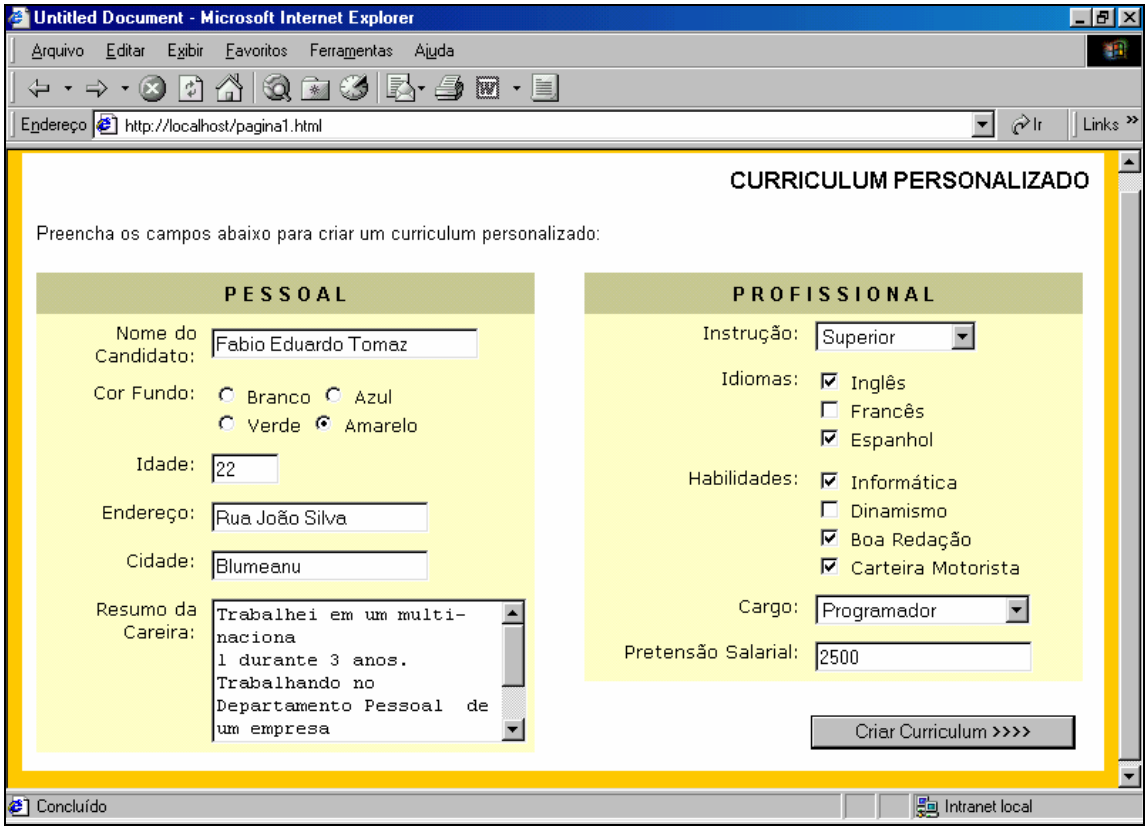
3.3.2 OPERACIONALIDADE DA IMPLEMENTAÇÃO

Com a linguagem de programação desenvolvida é possível criar diferentes aplicações para *sites* dinâmicos na WWW. Nesta seção são apresentados dois exemplos da construção de possíveis aplicações. O primeiro estudo de caso visa demonstrar as funcionalidades da linguagem quanto ao uso de comandos condicionais, comandos de repetições, operações aritméticas e a passagem de parâmetros de uma página HTML para um *script*. O segundo exemplo é um *script* que possibilita o acesso a um banco de dados.

3.3.2.1 ESTUDO DE CASO – PÁGINA DINÂMICA

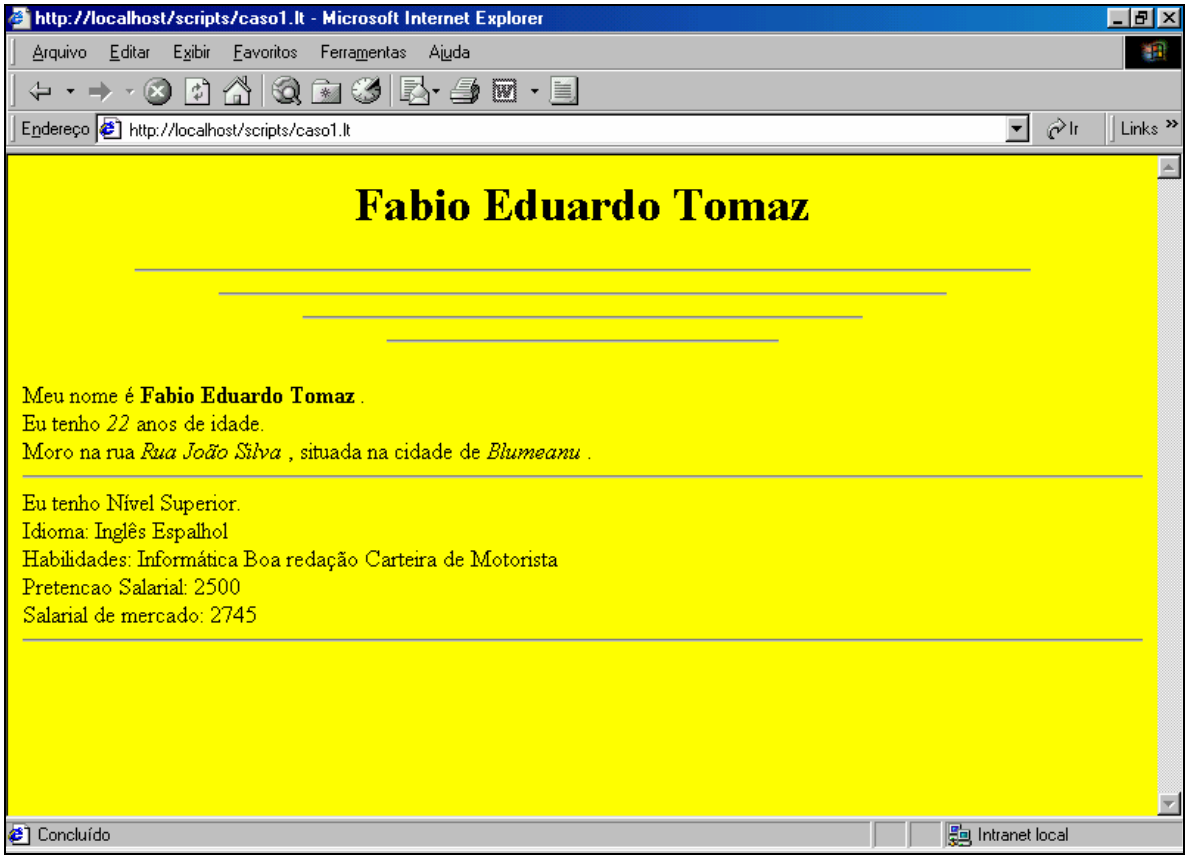
Este estudo de caso objetiva a criação de um *Curriculum Vitae* gerado a partir de um código de *script* escrito na linguagem proposta. Primeiramente o usuário acessa uma página inicial em HTML para o cadastramento de algumas informações básicas. Esta página é apresentada na fig. 8 onde o usuário preenche o seu nome, uma cor de fundo para a página, idade, endereço, cidade, um resumo da carreira e demais informações profissionais.

Figura 8 – PÁGINA HTML PARA ENTRADA DE DADOS



Após preencher as informações e confirmar o formulário o servidor de aplicação retorna ao usuário uma página HTML personalizada (fig. 9) de acordo com os parâmetros solicitados. Os parâmetros passados no formulário são transmitidos para o servidor de aplicação através de uma mensagem de solicitação HTTP. Após enviar a confirmação, o servidor WWW dispara o servidor de aplicação e inicia a interpretação do *script* “CASO1.LT”, resultando uma página HTML.

Figura 9 – PÁGINA HTML GERADA PELO SERVIDOR DE APLICAÇÃO



No quadro 29 é apresentado o código fonte do *script* “CASO1.LT”. Neste quadro são usados alguns comandos de *script* em conjunto com *tags* da linguagem HTML. Os comandos apresentados em negrito representam comandos da linguagem de *script* desenvolvida. O comando “IMPRIME” permite a impressão de expressões e constantes numéricas e ou alfanuméricas. O comando “ENQUANTO” é utilizado para imprimir linhas horizontais em seqüência iniciando com uma largura de 80 e diminuindo este valor em 10 enquanto a condição do laço for verdadeira. Na seqüência do código utilizando o comando “IMPRIME” são impressos parâmetros passados pelos campos do formulário HTML da fig. 8. No final do

script são utilizadas operações aritméticas disponível na linguagem para calcular o valor da variável “salmer”. O arquivo HTML gerado pelo “CASO1.LT” é apresentado no quadro 30.

Quadro 29 – ARQUIVO DE SCRIPT “CASO1.LT”

```
<html>
<# imprime ("<body bgcolor=" + $cor + ">"); #>
<center><H1><# imprime ($nome); #></h1></center>
<#
 x := 80;
 ENQUANTO x > 30 FACA
   imprime ("<hr width=" + x + "%>");
   x := x - 15;
 FIME;
#>
<p>Meu nome é <b><# imprime ($nome); #></b>.<br>
Eu tenho <i><# imprime ($idade); #></i> anos de idade.<br>
Moro na rua <i><# imprime ($rua); #></i>, situada na cidade de <i><# imprime ($cidade);
#></i>.
<hr>
<#
  i := 0 + $instrucao;
  SE i = 1 ENTAO
    imprime ("Eu tenho o primeiro Grau.");
  SENAO
    SE i = 2 ENTAO
      imprime ("Eu tenho o segundo Grau.");
    SENAO
      SE i = 3 ENTAO
        imprime ("Eu tenho Nível Superior.");
      FIMSE;
    FIMSE;
  FIMSE;
  imprime ("<br>Idioma: " + $idi1 + $idi2 + $idi3);
  imprime ("<br>Habilidades: " + $h1 + $h2 + $h3 + $h4);
  imprime ("<br>Pretencao Salarial: " + $salario);

  % Calcula o salario de mercado ficticio  %
  id := 0 + $idade;
  salmer := 100 + $salario;
  salmer := salmer + (25 / 5 + id) * 5 + (9 + 6 - 5);
  imprime ("<br>Salarial de mercado: ");
  imprime (salmer);
#>
<hr>
</body>
</html>
```

Quadro 30 – ARQUIVO HTML GERADOR A PARTIR DO *SCRIPT* “CASO1.LT”

```
<html>
<body bgcolor=#FFFF00>
<center><H1>
Fabio Eduardo Tomaz
</h1></center>
<hr width=80%>
<hr width=65%>
<hr width=50%>
<hr width=35%>
<p>Meu nome é <b>
Fabio Eduardo Tomaz
</b>.<br>Eu tenho <i>
22
</i> anos de idade.<br>Moro na rua <i>
Rua João Silva
</i>, situada na cidade de <i>
Blumeanu
</i>.<hr>
Eu tenho Nível Superior.
<br>Idioma: Inglês Espalhol
<br>Habilidades: Informática Boa redação Carteira de Motorista
<br>Pretencao Salarial: 2500
```

```
<br>Salarial de mercado:
2745
<hr>
</body>
</html>
```

3.3.2.2 ESTUDO DE CASO – BANCO DE DADOS

Este estudo de caso irá emitir uma listagem de clientes de uma empresa fictícia a partir de um banco de dados. O *script* criado não possui nenhum comando HTML, apenas comandos da linguagem desenvolvida. Neste estudo de caso o *script* é chamando diretamente a partir de uma URL, diferentemente do estudo de caso anterior que é disparado a partir de um formulário HTML. O arquivo de *script* utilizado neste estudo de caso é o "CASO2.LT" e é apresentado no quadro 31.

Quadro 31 – ARQUIVO DE SCRIPT "CASO2.LT"

```
<#
  imprime ("<HTML>");
  imprime ("<HEAD>");
  imprime ("<TITLE>Lista de Clientes</TITLE>");
  imprime ("</HEAD>");
  imprime ("<BODY>");
  imprime ("<H1>Lista de Clientes</H1><BR>");

  DB(10, "DBDEMOS"); % Conecta a base de dados %
  DB(30, "SELECT * FROM CLIENTS ORDER BY FIRST_NAME");

  imprime("<table border>");
  imprime("<tr>");
  imprime("  <td><b>Nome Cliente</b></td>");
  imprime("  <td><b>Cidade</b></td>");
  imprime("  <td><b>Estado</b></td>");
  imprime("  <td><b>Telefone</b></td>");
  imprime("</tr>");

  ENQUANTO DBSTATUS > 0 FACA
    imprime("<tr>");
    imprime("  <td>" + DBCAMPO("FIRST_NAME") + " " + DBCAMPO("LAST_NAME")  + "</td>");
    imprime("  <td>" + DBCAMPO("CITY")  + "</td>");
    imprime("  <td>" + DBCAMPO("STATE")  + "</td>");
    imprime("  <td>" + DBCAMPO("TELEPHONE")  + "</td>");
    imprime("</tr>");
    DB(40, 0);   % Proximo %
  FIME;

  imprime("</table>");
  imprime ("</BODY>");
  imprime ("</HTML>");
  DB(20, 0);   % Fecha base de dados %
#>
```

Neste exemplo basicamente são utilizado os comandos "DB" e "DBCAMPO". O comando "DB" é utilizado para manipular o acesso ao banco de dados. Este comando pode executar diferentes ações, conforme indica o primeiro parâmetro, o qual pode ser:

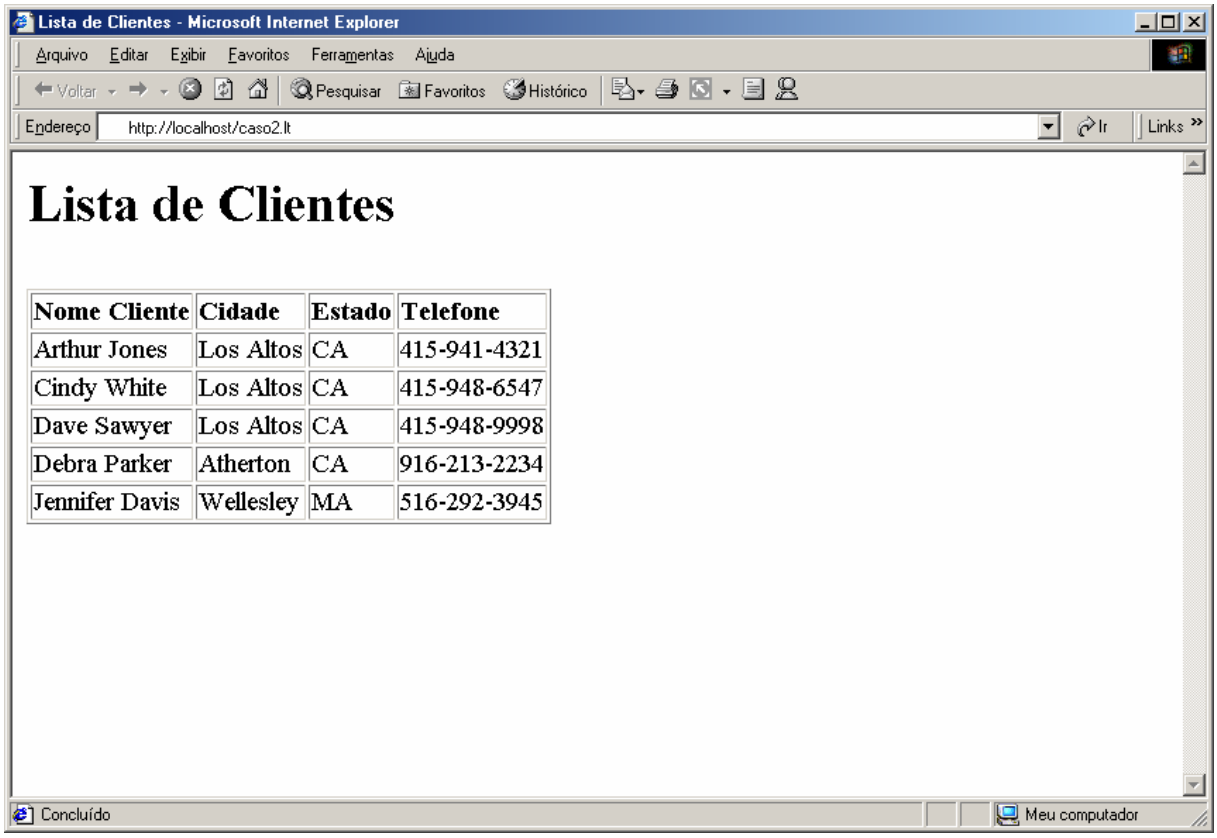
a) 10 – conecta a um banco de dados através de um alias da BDE (Borland Database Engine) indicado no segundo parâmetro da função;

b) 20 – fecha o banco de dados;
c) 30 – permite manipular tabelas de um banco de dados a partir um comando SQL informado no segundo parâmetro da função “DB”;
d) 40 – avança o cursor da seleção feita por um comando SQL em um registro;
e) 50 – retorna o cursor da seleção feita por um comando SQL em um registro.

O comando “DBCAMPO” é usado para retornar o conteúdo de um determinado campo existente num banco de dados. Para manipular as operações de avançar e retornar o cursor é definido na linguagem de *script* um identificador chamado “DBStatus”. Este identificador retorna zero se o cursor chegou ao final ou início do arquivo de banco de dados, e um valor diferente de zero se existe um registro lido.

Após a interpretação deste arquivo é retornada ao usuário uma página HTML conforme apresentado na fig 10. A listagem de cliente é apresentada em uma tabela criada usando *tags* HTML e comandos de *script* para impressão do documento. O arquivo fonte HTML gerado correspondente à listagem de clientes é apresentada no quadro 32.

Figura 10 – PÁGINA HTML COM UMA LISTAGEM DE CLIENTES GERADA PELO SERVIDOR DE APLICAÇÃO A PARTIR DE UM BANCO DE DADOS



Quadro 32 – ARQUIVO HTML GERADOR A PARTIR DO *SCRIPT* “CASO2.LT”

```
<HTML>
<HEAD>
<TITLE>Lista de Clientes</TITLE>
</HEAD>
<BODY>
<H1>Lista de Clientes</H1><BR>
<table border>
<tr>
  <td><b>Nome Cliente</b></td>
  <td><b>Cidade</b></td>
  <td><b>Estado</b></td>
  <td><b>Telefone</b></td>
</tr>
<tr>
  <td>Arthur Jones</td>
  <td>Los Altos</td>
  <td>CA</td>
  <td>415-941-4321</td>
</tr>
<tr>
  <td>Cindy White</td>
  <td>Los Altos</td>
  <td>CA</td>
  <td>415-948-6547</td>
</tr>
<tr>
  <td>Dave Sawyer</td>
  <td>Los Altos</td>
  <td>CA</td>
  <td>415-948-9998</td>
</tr>
<tr>
  <td>Debra Parker</td>
  <td>Atherton</td>
  <td>CA</td>
  <td>916-213-2234</td>
</tr>
<tr>
  <td>Jennifer Davis</td>
  <td>Wellesley</td>
  <td>MA</td>
  <td>516-292-3945</td>
</tr>
</table>
</BODY>
</HTML>
```

4 CONCLUSÕES

O objetivo de criar uma linguagem de programação encapsulada ao código da linguagem HTML foi alcançado. As técnicas de especificação da linguagem através de expressões regulares, BNF e esquemas de tradução mostraram-se adequadas para garantir a implementação da linguagem de programação de *script*.

A linguagem criada disponibiliza a utilização de operadores aritméticos, operadores condicionais, iterações, comandos condicionais e permite a entrada e saída de dados através da passagem de parâmetros em formulários HTML, banco de dados e impressão de textos. Com estas funcionalidades a linguagem desenvolvida atende aos objetivos específicos previamente formulados.

Utilizando a UML, com o auxílio da ferramenta *Rational Rose*, foi possível antes de iniciar a implementação ter uma visão geral de como o interpretador da linguagem iria funcionar e prever soluções de possíveis problemas na construção do interpretador ainda na fase de análise. O diagrama de classes desenhado na especificação foi seguido à risca na fase de implementação, garantindo um código final com mais qualidade.

A utilização do ambiente de desenvolvimento *Borland Delphi* permitiu depurar erros no interpretador com facilidades. Uma das desvantagens de utilizar esta ferramenta é o de não permitir a portabilidade para outras plataformas.

As classes para definição da gramática e o algoritmo para criação da tabela de análise sintática SLR trouxeram grande flexibilidade para criar novas funcionalidades ou futuras correções na linguagem de programação desenvolvida. Utilizando as classes desenvolvidas é possível construir uma tabela sintática para a maioria das construções de linguagens de programação. No desenvolvimento do trabalho, quando se fez necessário modificar a linguagem, bastava modificar as produções da gramática definidas na classe “TGramatica” e criar novas ou alterar as rotinas semânticas existentes para geração do código intermediário.

Comparada com as linguagens de *script* apresentadas no item 2.5, a linguagem que foi desenvolvida ainda apresenta algumas limitações, tais como a inclusão de funções, procedimentos e novos tipos de dados, os quais podem ser superadas com as extensões

propostas na seção seguinte. Apesar destas limitações com a linguagem é possível criar *sites* na WWW que necessitem de conteúdos dinâmicos.

4.1 EXTENSÕES

Para extensões deste trabalho sugere-se:

a) incluir na linguagem a manipulação de funções e procedimentos;
b) definir novos tipos de dados para a linguagem (tipo real, ponteiros, objetos e matrizes) ;
c) adaptar a rotina de criação da tabela de análise sintática a fim de implementar um gerador de analisador sintático SLR;
d) estender as funcionalidades quanto a utilização de banco de dados, incluído acesso nativo a alguns bancos de dados;
e) criar um ambiente de desenvolvimento integrado para facilitar a utilização da linguagem.

ANEXO 1 – TABELA SLR

Este anexo apresenta a tabela de análise sintática criada para o reconhecimento da gramática desenvolvida neste trabalho. As linhas da tabela, apresentadas nos quadros 33 e 34, representam os estados e as colunas os elementos da gramática. O elemento cifrão ($) na última coluna da tabela representa o símbolo de fim de arquivo. As possíveis ações dadas no cruzamento de um estado com um elemento da gramática são:

a) "d*x*": desvio para outro estado tabela, indicado por x;

b) "s*i":* empilhar símbolo de entrada e o estado i*;*

c) "r*j":* reduzir através da produção *j;*

d) *"acc":* aceitar a cadeia de entrada;

e) *""*: brancos representam erros sintáticos.

Quadro 33 – TABELA SINTÁTICA SLR (PARTE I)

S	<HTML>	<SCRIPT>	<LISTACMD>	<COMANDO>	<EXPRE>	<TERMO>	<FATOR>	<CONDICAO>	<COND_OU>	<COND_LOG>	<M>	<#	#>	,	;	:=	+ ou -	* ou /	id	num	String	(	)	E	OU	> = <	IMPRIME	SE	ENTAO	SENAO	FIMSE	ENQUANTO	FACA	FIME	HTML	DB	DBCAMPO	$
S00	d01	d02										s03																							s04			
S01		d05										s03																							s04			acc
S02												r01		r01		r01			r01	r01	r01	r01	r01				r01	r01				r01			r01	r01	r01	r01
S03			d06	d07											r12				s08								s09	s10				s11				s12		
S04												r03		r03		r03			r03	r03	r03	r03	r03				r03	r03				r03			r03	r03	r03	r03
S05												r02		r02		r02			r02	r02	r02	r02	r02				r02	r02				r02			r02	r02	r02	r02
S06				d13									s14		r12				s08								s09	s10				s11				s12		
S07															s15																							
S08																s16																						
S09																						s17																
S10					d18	d19	d20	d21	d22	d23									s24	s25	s26	s27															s28	
S11											d29			r27		r27			r27	r27	r27	r27	r27				r27	r27				r27				r27	r27	
S12																						s30																
S13															s31																							
S14												r04		r04		r04			r04	r04	r04	r04	r04				r04	r04				r04			r04	r04	r04	r04
S15													r06	r06		r06			r06	r06	r06	r06	r06				r06	r06		r06	r06	r06		r06		r06	r06	
S16					d32	d19	d20												s24	s25	s26	s27															s28	
S17					d33	d19	d20												s24	s25	s26	s27															s28	
S18																	s34							r26	r26	s35			r26				r26					
S19															r14		r14	s36					r14	r14	r14	r14			r14				r14					
S20															r16		r16	r16					r16	r16	r16	r16			r16				r16					
S21																								s37					s38									
S22																								r22	s39				r22				r22					
S23																								r24	r24				r24				r24					
S24															r18		r18	r18					r18	r18	r18	r18			r18				r18					

Quadro 34 – TABELA SINTÁTICA SLR (PARTE II)

S	<HTML>	<SCRIPT>	<LISTACMD>	<COMANDO>	<EXPRE>	<TERMO>	<FATOR>	<CONDICAO>	<COND_OU>	<COND_LOG>	<M>	<#	#>	,	;	:=	+ ou -	* ou /	id	num	String	(	)	E	OU	> = <	IMPRIME	SE	ENTAO	SENAO	FIMSE	ENQUANTO	FACA	FIME	HTML	DB	DBCAMPO	$
S25															r19		r19	r19					r19	r19	r19	r19			r19				r19						
S26															r20		r20	r20					r20	r20	r20	r20			r20				r20						
S27					d40	d19	d20												s24	s25	s26	s27																s28	
S28																						s41																
S29					d18	d19	d20	d42	d22	d23									s24	s25	s26	s27																s28	
S30																				s43																		
S31													r05	r05		r05			r05	r05	r05	r05	r05				r05	r05		r05	r05	r05		r05		r05	r05	
S32															r07		s34																					
S33																	s34						s44															
S34						d45	d20												s24	s25	s26	s27															s28	
S35					d46	d19	d20												s24	s25	s26	s27															s28	
S36							d47												s24	s25	s26	s27															s28	
S37											d48			r27		r27			r27	r27	r27	r27	r27				r27	r27				r27				r27	r27	
S38											d49			r27		r27			r27	r27	r27	r27	r27				r27	r27				r27				r27	r27	
S39											d50			r27		r27			r27	r27	r27	r27	r27				r27	r27				r27				r27	r27	
S40																	s34						s51														
S41																					s52																
S42																								s37									s53					
S43														s54																								
S44															r08																							
S45															r13		r13	s36					r13	r13	r13	r13			r13				r13					
S46																	s34							r25	r25				r25				r25					
S47															r15		r15	r15					r15	r15	r15	r15			r15				r15					
S48					d18	d19	d20		d55	d23									s24	s25	s26	s27															s28	
S49			d56	d07											r12				s08								s09	s10				s11				s12		
S50					d18	d19	d20			d57									s24	s25	s26	s27															s28	
S51															r17		r17	r17					r17	r17	r17	r17			r17				r17					
S52																							s58															
S53											d59			r27		r27			r27	r27	r27	r27	r27				r27	r27				r27				r27	r27	
S54					d60	d19	d20												s24	s25	s26	s27															s28	
S55																								r21	s39				r21				r21					
S56				d13											r12				s08								s09	s10		s61	s62	s11				s12		
S57																								r23	r23				r23				r23					
S58															r29		r29	r29					r29	r29	r29	r29			r29				r29					
S59			d63	d07											r12				s08								s09	s10				s11				s12		
S60																	s34						s64														
S61											d65			r27		r27			r27	r27	r27	r27	r27				r27	r27				r27				r27	r27	
S62															r09																							
S63				d13											r12				s08								s09	s10				s11		s66		s12		
S64															r28																							
S65			d67	d07											r12				s08								s09	s10				s11				s12		
S66															r11																							
S67				d13											r12				s08								s09	s10			s68	s11				s12		

ANEXO 2 – REGRAS SEMÂNTICAS

No quadro 35 são apresentadas as regras semânticas da linguagem proposta. Essas regras geram o código intermediário.

Quadro 35 – REGRAS SEMÂNTICAS

Regra 3
Esta regra é executada quando é encontrado um código HTML dentro do *script*. Na execução da regra é carregada uma variável *string* em memória e disparado um comando para impressão da mesma.

```
procedure TAnaSem.Regra_Html;
var
  x: String;
  i: Integer;
begin
  { Fator -> ttHtml }
  x := Format('html%4.4d', [TAplicacao(FOwner).Lexico.TabSim.LastIndex]);
  i := TAplicacao(FOwner).Lexico.TabSim.Inserir(x);
  TAplicacao(FOwner).Running.SetVariable(i, TAplicacao(FOwner).AnaSin.Pilha[0].ValStr);
  if Length(TAplicacao(FOwner).AnaSin.Pilha[0].ValStr) > 0 then
  begin
    FTabCod.AddInstrucao(tiCRVL, i, 0, 0);
    FTabCod.AddInstrucao(tiIMPR, i, tvString, 0);
    FTabCod.AddInstrucao(tiDSSE, 0, 0, FTabCod.Count + 1);
  end;
end;
```

Regra 5
Disparada após redução de uma lista de comandos.
Sua principal função é preencher a lista de próximos através da função de *backpatching*.

```
procedure TAnaSem.Regra_ListaCmd;
begin
  { tnListaCmd ->  tnListaCmd tnComando ttPontoVirgula }
  FTabCod.Backpatching(TAplicacao(FOwner).AnaSin.Pilha[1].LisPrx, FTabCod.Count);
  TAplicacao(FOwner).AnaSin.Pilha[1].LisPrx.Add(FTabCod.Count-1);
  TAplicacao(FOwner).AnaSin.Topo.LisPrx.Concatena(TAplicacao(FOwner).AnaSin.Pilha[1].LisPrx);
end;
```

Regra 6
Igual a regra 5, mas contém apenas um comando.

```
procedure TAnaSem.Regra_Cmd;
begin
  { tnListaCmd ->  tnComando ttPontoVirgula }
  FTabCod.Backpatching(TAplicacao(FOwner).AnaSin.Pilha[1].LisPrx, FTabCod.Count);
  TAplicacao(FOwner).AnaSin.Pilha[1].LisPrx.Add(FTabCod.Count-1);
  TAplicacao(FOwner).AnaSin.Topo.LisPrx.Concatena(TAplicacao(FOwner).AnaSin.Pilha[1].LisPrx);
end;
```

Regra 7
Gera código intermediário para atribuição de valores e verifica tipos de dados incompatíveis.

```
procedure TAnaSem.Regra_Atribuicao;
begin
  { Atribuição }
  if TAplicacao(FOwner).AnaSin.Pilha[2].TipVar <> TAplicacao(FOwner).AnaSin.Pilha[0].TipVar then
    raise Exception.Create('Tipos Incompatíveis na Atribuição');
  FTabCod.AddInstrucao(tiARMZ, TAplicacao(FOwner).AnaSin.Pilha[2].EndVar, 0, 0);
  FTabCod.AddInstrucao(tiDSSE, 0, 0, FTabCod.Count + 1);
end;
```

Regra 8
Gera código intermediário para imprimir dados

```
procedure TAnaSem.Regra_Imprime;
begin
```

```
  { Impressão }
  FTabCod.AddInstrucao(tiIMPR, TAplicacao(FOwner).AnaSin.Pilha[1].EndVar, TAplicacao(FOwner).AnaSin.Pilha[1].TipVar, 0);
  FTabCod.AddInstrucao(tiDSSE, 0, 0, FTabCod.Count + 1);
end;
```

Regra 9
Faz a tradução do comando condicional “SE”

```
procedure TAnaSem.Regra_Se;
begin
  { Cmd -> Se Condicao Entao M Comando Fimse }
  FTabCod.Backpatching(TAplicacao(FOwner).AnaSin.Pilha[4].LisVer, TAplicacao(FOwner).AnaSin.Pilha[2].IniCod);
  TAplicacao(FOwner).AnaSin.Topo.LisPrx.Concatena(TAplicacao(FOwner).AnaSin.Pilha[4].LisFal);
  TAplicacao(FOwner).AnaSin.Topo.LisPrx.Concatena(TAplicacao(FOwner).AnaSin.Pilha[1].LisPrx);
end;
```

Regra 10
Faz a tradução do comando condicional “SE/SENAO”

```
procedure TAnaSem.Regra_SeSenao;
begin
  { Cmd -> Se Condicao Entao M Comando Senao M Comandos FimSe }
  FTabCod.Backpatching(TAplicacao(FOwner).AnaSin.Pilha[7].LisFal, TAplicacao(FOwner).AnaSin.Pilha[2].IniCod);
  FTabCod.Backpatching(TAplicacao(FOwner).AnaSin.Pilha[7].LisVer, TAplicacao(FOwner).AnaSin.Pilha[5].IniCod);
  TAplicacao(FOwner).AnaSin.Topo.LisPrx.Concatena(TAplicacao(FOwner).AnaSin.Pilha[1].LisPrx);
  TAplicacao(FOwner).AnaSin.Topo.LisPrx.Concatena(TAplicacao(FOwner).AnaSin.Pilha[4].LisPrx);
end;
```

Regra 11
Faz a tradução do comando de iteração “ENQUANTO”

```
procedure TAnaSem.Regra_Enquanto;
begin
  { tnComando ->  [ttEnquanto tnM tnCondicao ttFaca tnM tnListaCmd ttFimEnquanto }
  FTabCod.Backpatching(TAplicacao(FOwner).AnaSin.Pilha[1].LisPrx, TAplicacao(FOwner).AnaSin.Pilha[5].IniCod);
  FTabCod.Backpatching(TAplicacao(FOwner).AnaSin.Pilha[4].LisVer, TAplicacao(FOwner).AnaSin.Pilha[2].IniCod);
  TAplicacao(FOwner).AnaSin.Topo.LisPrx.Concatena(TAplicacao(FOwner).AnaSin.Pilha[4].LisFal);
  FTabCod.AddInstrucao(tiDSSE, 0, 0, TAplicacao(FOwner).AnaSin.Pilha[5].IniCod);
end;
```

Regra 13
Traduz uma expressão contendo adição ou subtração. Gera os instruções “tiSOMA” ou “tiSUBT”.

```
procedure TAnaSem.Regra_Expr;
begin
  { Expr -> Expr + Term }
  TAplicacao(FOwner).AnaSin.Topo.EndVar := TAplicacao(FOwner).AnaSin.Pilha[2].EndVar;
  TAplicacao(FOwner).AnaSin.Topo.TipVar := TAplicacao(FOwner).AnaSin.Pilha[2].TipVar;
  case TAplicacao(FOwner).AnaSin.Pilha[1].NomLex[1] of
    '+':
      FTabCod.AddInstrucao(tiSOMA, 0, 0, 0);
    '-':
      FTabCod.AddInstrucao(tiSUBT, 0, 0, 0);
  end;
end;
```

Regra 14
Apenas carrega valores na pilha sintática
Ocorre na redução de um termo para expressão.

```
procedure TAnaSem.Regra_ExprTermo;
begin
  { Expr -> Term }
  TAplicacao(FOwner).AnaSin.Topo.EndVar := TAplicacao(FOwner).AnaSin.Pilha[0].Endvar;
  TAplicacao(FOwner).AnaSin.Topo.TipVar := TAplicacao(FOwner).AnaSin.Pilha[0].TipVar;
end;
```

Regra 15
Traduz uma expressão contendo multiplicação ou divisão.

```
procedure TAnaSem.Regra_Termo;
begin
  { Term -> Term * Fat }
  TAplicacao(FOwner).AnaSin.Topo.EndVar := TAplicacao(FOwner).AnaSin.Pilha[0].EndVar;
  TAplicacao(FOwner).AnaSin.Topo.TipVar := TAplicacao(FOwner).AnaSin.Pilha[0].TipVar;
  case TAplicacao(FOwner).AnaSin.Pilha[1].NomLex[1] of
    '*':
      FTabCod.AddInstrucao(tiMULT, 0, 0, 0);
```

```
    '/':
      FTabCod.AddInstrucao(tiDIVI, 0, 0, 0);
  end;
end;
```

Regra 16
Carrega valores na pilha sintática
Ocorre na redução de um fator para um termo.

```
procedure TAnaSem.Regra_TermFator;
begin
  { Term -> Fator  }
  TAplicacao(FOwner).AnaSin.Topo.EndVar := TAplicacao(FOwner).AnaSin.Pilha[0].Endvar;
  TAplicacao(FOwner).AnaSin.Topo.TipVar := TAplicacao(FOwner).AnaSin.Pilha[0].TipVar;
end;
```

Regra 17
Carrega valores na pilha sintática em uma expressão entre parênteses.
Ocorre na redução de uma expressão entre parênteses para um fator.

```
procedure TAnaSem.Regra_FatorExpr;
begin
  { Fator -> ( expr ) }
  TAplicacao(FOwner).AnaSin.Topo.EndVar := TAplicacao(FOwner).AnaSin.Pilha[1].Endvar;
  TAplicacao(FOwner).AnaSin.Topo.TipVar := TAplicacao(FOwner).AnaSin.Pilha[1].TipVar;
end;
```

Regra 18
Analisada quando encontrado um identificador.
Ocorre na redução de um identificador para um fator.

```
procedure TAnaSem.Regra_FatorId;
begin
  { Fator -> ID }
  TAplicacao(FOwner).AnaSin.Topo.EndVar := TAplicacao(FOwner).Lexico.TabSim.Inserir(TAplicacao(FOwner).AnaSin.Pilha[0].NomLex);
  TAplicacao(FOwner).AnaSin.Topo.TipVar := TAplicacao(FOwner).AnaSin.Pilha[0].TipVar;
  FTabCod.AddInstrucao(tiCRVL, TAplicacao(FOwner).AnaSin.Topo.EndVar, 0, 0);
end;
```

Regra 19
Analisada quando encontrada uma constante numérica.

```
procedure TAnaSem.Regra_FatorNum;
begin
  { Fator -> Num }
  TAplicacao(FOwner).AnaSin.Topo.EndVar := TAplicacao(FOwner).Lexico.TabSim.Temp;
  TAplicacao(FOwner).AnaSin.Topo.TipVar := tvInteiro;
  FTabCod.AddInstrucao(tiCRCT, TAplicacao(FOwner).AnaSin.Pilha[0].ValNum, 0, 0);
end;
```

Regra 20
Analisada quando encontrado uma constante alfanumérica.

```
procedure TAnaSem.Regra_FatorString;
var
  x: String;
begin
  { Fator -> String } // Busca da tabela de constantes alfanuméricas
  TAplicacao(FOwner).AnaSin.Topo.TipVar := tvString;
  TAplicacao(FOwner).AnaSin.Topo.EndVar := TAplicacao(FOwner).Lexico.TabSim.Inserir(TAplicacao(FOwner).AnaSin.Pilha[0].ValStr);
  x := Copy(TAplicacao(FOwner).AnaSin.Pilha[0].ValStr, 2, Length(TAplicacao(FOwner).AnaSin.Pilha[0].ValStr) - 2);
  TAplicacao(FOwner).Running.SetVariable(TAplicacao(FOwner).AnaSin.Topo.EndVar, x);
  FTabCod.AddInstrucao(tiCRVL, TAplicacao(FOwner).AnaSin.Topo.EndVar, 0, 0);
end;
```

Regra 21
Traduz uma condição E. A função de *backpaching* é utilizada para gerar desvios para M no caso da primeira condição for verdadeiro.

```
procedure TAnaSem.Regra_CondicaoE;
begin
  { tnCondicao -> tnCondicao Expr M tnCondOU }
  FTabCod.Backpatching(TAplicacao(FOwner).AnaSin.Pilha[3].LisVer, TAplicacao(FOwner).AnaSin.Pilha[1].IniCod);
  TAplicacao(FOwner).AnaSin.Topo.LisFal.Concatena(TAplicacao(FOwner).AnaSin.Pilha[3].LisFal);
  TAplicacao(FOwner).AnaSin.Topo.LisFal.Concatena(TAplicacao(FOwner).AnaSin.Pilha[0].LisFal);
  TAplicacao(FOwner).AnaSin.Topo.LisVer.Concatena(TAplicacao(FOwner).AnaSin.Pilha[0].LisVer);
end;
```

Regra 22

Carrega valores na pilha sintática

```
procedure TAnaSem.Regra_CondicaoOU;
begin
  { tnCondicao -> tnCondOU }
  TAplicacao(FOwner).AnaSin.Topo.LisVer.Concatena(TAplicacao(FOwner).AnaSin.Pilha[0].LisVer);
  TAplicacao(FOwner).AnaSin.Topo.LisFal.Concatena(TAplicacao(FOwner).AnaSin.Pilha[0].LisFal);
end;
```

Regra 23

Traduz uma condição OU. Itens falsos desviam para M através da função de *backpatching*.

```
procedure TAnaSem.Regra_CondicaoOuLog;
begin
  { tnCondicaoOU-> tnCondOU or M tnCondLog }
  FTabCod.Backpatching(TAplicacao(FOwner).AnaSin.Pilha[3].LisFal, TAplicacao(FOwner).AnaSin.Pilha[1].IniCod);
  TAplicacao(FOwner).AnaSin.Topo.LisVer.Concatena(TAplicacao(FOwner).AnaSin.Pilha[3].LisVer);
  TAplicacao(FOwner).AnaSin.Topo.LisVer.Concatena(TAplicacao(FOwner).AnaSin.Pilha[0].LisVer);
  TAplicacao(FOwner).AnaSin.Topo.LisFal.Concatena(TAplicacao(FOwner).AnaSin.Pilha[0].LisFal);
end;
```

Regra 24

Carrega valores na pilha em uma condição OU.

```
procedure TAnaSem.Regra_Ou;
begin
  { tnCondicao -> tnCondicaoOU }
  TAplicacao(FOwner).AnaSin.Topo.LisVer.Concatena(TAplicacao(FOwner).AnaSin.Pilha[0].LisVer);
  TAplicacao(FOwner).AnaSin.Topo.LisFal.Concatena(TAplicacao(FOwner).AnaSin.Pilha[0].LisFal);
end;
```

Regra 25

Traduz operações com os operadores =, > ou <.

```
procedure TAnaSem.Regra_OpLog;
begin
  { CondLog -> Expr oplog Expr }
  TAplicacao(FOwner).AnaSin.Topo.LisVer.Add(FTabCod.Count);
  case TAplicacao(FOwner).AnaSin.Pilha[1].NomLex[1] of
    '=':
      FTabCod.AddInstrucao(tiDSIG, 0, 0, -1);
    '>':
      FTabCod.AddInstrucao(tiDSMA, 0, 0, -1);
    '<':
      FTabCod.AddInstrucao(tiDSME, 0, 0, -1);
  end;
  TAplicacao(FOwner).AnaSin.Topo.LisFal.Add(FTabCod.Count);
  FTabCod.AddInstrucao(tiDSSE, 0, 0, -1);
end;
```

Regra 26

Carrega valores na pilha na redução de uma expressão para uma condição.

```
procedure TAnaSem.Regra_OpLogCond;
begin
  { tnCondicao -> Expre }
  TAplicacao(FOwner).AnaSin.Topo.LisVer.Concatena(TAplicacao(FOwner).AnaSin.Pilha[1].LisVer);
  TAplicacao(FOwner).AnaSin.Topo.LisFal.Concatena(TAplicacao(FOwner).AnaSin.Pilha[1].LisFal);
end;
```

Regra 27

Carrega o atributo “IniCod” na pilha sintática, usado na função de *backpaching*.

```
procedure TAnaSem.Regra_M;
begin
  { M -> vazio }
  TAplicacao(FOwner).AnaSin.Topo.IniCod := FTabCod.Count;
end;
```

Regra 28

Faz a tradução do comando DB.

```
procedure TAnaSem.Regra_Db;
begin
  { COMANDO -> ttDB, ttAbrePar, ttNum, ttVirgula tnDBOp, ttFechaPar }
  case TAplicacao(FOwner).AnaSin.Pilha[3].ValNum of
    10: // Conecta "Alias"
      FTabCod.AddInstrucao(tiDBEX, 10, TAplicacao(FOwner).AnaSin.Pilha[1].EndVar, 0);
    20: // Disconecta "Alias"
```

```
    FTabCod.AddInstrucao(tiDBEX, 20, 0, 0);
  30: // abre "SQL"
    FTabCod.AddInstrucao(tiDBEX, 30, TAplicacao(FOwner).AnaSin.Pilha[1].EndVar, 0);
  40: // Proximo
    FTabCod.AddInstrucao(tiDBEX, 40, 0, 0);
  50: // Anterior
    FTabCod.AddInstrucao(tiDBEX, 70, 0, 0);
  else
    raise Exception.Create('Parametro da função DB invalido ! <br>'+
     'Use 10 = conectar, 20 = desconectar, 30=abrir, '+
     'Use 40 = proximo , 50 = anterior ');
 end;
end;
```

Regra 29

Faz a tradução do comando DBCAMPO

```
procedure TAnaSem.Regra_DbCampo;
begin
  { tnFator -> ttDBCampo, ttAbrePar, ttString, ttFechapar }
  TAplicacao(FOwner).AnaSin.Topo.TipVar := tvString;
  TAplicacao(FOwner).AnaSin.Topo.EndVar := TAplicacao(FOwner).Lexico.TabSim.Inserir(TAplicacao(FOwner).AnaSin.Pilha[1].ValStr);
  x := Copy(TAplicacao(FOwner).AnaSin.Pilha[1].ValStr, 2, Length(TAplicacao(FOwner).AnaSin.Pilha[1].ValStr) - 2);
  TAplicacao(FOwner).Running.SetVariable(TAplicacao(FOwner).AnaSin.Topo.EndVar, x);
  FTabCod.AddInstrucao(tiCRVL, TAplicacao(FOwner).AnaSin.Topo.EndVar, 0, 0);
  FTabCod.AddInstrucao(tiDBEX, 0, TAplicacao(FOwner).AnaSin.Topo.EndVar, 0);
end;
```

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, Alfredo V.; SETHI, Ravi; ULMAN, Jeffrey. **Compiladores:** princípios, técnicas e ferramentas**.** Massachusetts: Addison Wesley Publisching Co., 1995.

ANSELMO, Fenando. **Php e Mysql para Windows**. Florianópolis: Visual Books, 2000.

FOX, Brian J. **The Meta-HTML language reference manual**, [S.l], [1998?]. Disponível em: < http://www.metahtml.org/documentation/manual/overview.html>. Acesso em: 03 set. 2001.

FURLAN, José Davi. **Modelagem de objetos através da UML.** São Paulo: Makron Books, 1998.

GRAHAM, Ian S. **The HTML sourcebook**. New York: John Wiley & Sons Inc, 1996.

KOWALTOWSKI, Tomasz. **Implementação de linguagem de programação**. Rio de Janeiro: Guanabara Dois S.A., 1983.

MCFEDRIES, Paul. **Guia incrível para criação de páginas *web* com HTML**. Trad. Elaine Pezzoli. São Paulo: Makron Books, 1998.

MONTEIRO, Miguel Pimenta. **Compiladores 2000/2001,** [S.l], [2000]. Disponível em: < http://www.fe.up.pt/~apm/comp/>. Acesso em: 18 out. 2001.

NETO, João José. **Introdução à compilação**. Rio de Janeiro: Livros Técnicos e Científicos, 1987.

NUNES, Marcelo Pereira. Uso de linguagens de script como ferramentas de apoio ao ensino. In: Semana Acadêmica de Engenharia de Computação, 2., 1999, Rio Grande, **Anais...** Rio Grande: FURG, 1999. p. 71-81.

PACHECO, Xavier; TEIXEIRA, Steve. **Delphi 5 developers guide.** Indianápolis: Sams Publishing, 2000.

PINHEIRO, Manuele Kirsch. **Simulando um compilador na Web.** 1997. 91 f. Monografia (Conclusão de Curso de Bacharel em Informática). Universidade Federal de Santa Maria, Santa Maria.

PRICE, Ana Maria de Alencar; TOSCANI, Simão Sirineo. **Implementação de linguagens de programação:** compiladores. Porto Alegre: Sagra Luzzatto, 2001.

RANGEL, José Lucas. **Disciplina de Compiladores**, [S.l], [1999]. Disponível em: < http://www-di.inf.puc-rio.br/~rangel/>. Acesso em: 15 out. 2001.

VENETIANER, Tomaz. **HTML:** desmistificando a linguagem da internet. São Paulo: Makron Books, 1996.

WEISSINGER, A. Keyton. **ASP** – guia completo. Rio de Janeiro: Ciência Moderna Ltda, 1995.