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

# Desenvolvimento de um compilador C cliente-servidor para auxiliar o aprendizado em computação através de navegadores web

**Claudson da Silva Oliveira**

JUIZ DE FORA
MARÇO, 2013

# Desenvolvimento de um compilador C cliente-servidor para auxiliar o aprendizado em computação através de navegadores web

Claudson da Silva Oliveira

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: Marcelo Bernardes Vieira

JUIZ DE FORA

MARÇO, 2013

# Desenvolvimento de um compilador C cliente-servidor para auxiliar o aprendizado em computação através de navegadores web

Claudson da Silva Oliveira

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

Aprovada por:

---

Marcelo Bernardes Vieira
D. Sc. em Ciência da Computação

---

Rubens de Oliveira
D. Sc em Engenharia Civil

---

Luciana Conceição Dias Campos
D. Sc em Ciência da Computação

JUIZ DE FORA

22 DE MARÇO, 2013

*Aos amigos mais próximos pelos importantes momentos de descanso. Sem eles, a caminhada seria muito mais árdua.*

# Resumo

O ensino de Computação básica torna-se muitas vezes desafiador: os institutos precisam dar manutenção para manter os ambientes de compilação nos laboratórios, e os alunos apresentam dificuldades para utilizar esses ambientes. O objetivo desse trabalho é reduzir esses problemas definindo e fornecendo um compilador que leia códigos na linguagem C e execute-os usando um navegador web, dessa forma todo o controle do ambiente e a dinâmica no ensino poderia ser facilitada. Além disso, é interessante fornecer informações como o tempo de execução do código gerado, uma ferramenta para análise de memória passo a passo e análises de algoritmos submetidos ao servidor, para salientar dúvidas dos alunos e acompanhar a evolução em uma aula.


**Palavras-chave:** compilador, linguagem c, aprendizado de computaçao, cliente-servidor, php, javascript, navegador.

# Abstract

The basic computer learning is hard for many times: the institutes need give maintenence to preserve the compiler environment in labs, and students show difficulty to use that environments. The goal of this work is reduce those problems defining and serving a compiler that reads sources code in language C and executes them using a web browser, then all control the environment and the dinamic in learning could be facilitated. Furthermore, generate informations as execution time, a tool to analysis of memory step by step and analysis of algorithms sent to server to answer some questions of the students is interesting.

**Keywords:** compiler, C language, server client, computational learning, php, javascript.

# Agradecimentos

A toda a família por apostar em um sonho.

Aos professores do Departamento de Ciência da Computação por todos os anos de ensinamento, dedicação e apoio. Em especial ao professor Marcelo Bernardes Vieira pela paciência e contribuição para o crescimento profissional e pessoal.

*"Cada sonho que você deixa para trás é um futuro que deixa de existir".*

*Steve Jobs*

# Sumário


**Lista de Figuras** 8

**Lista de Tabelas** 9

**Lista de Abreviações** 10

**1 Introdução** 11
- 1.1 Motivação . . . . . 12
- 1.2 Definição do problema . . . . . 12
- 1.3 Objetivos . . . . . 12

**2 Arquitetura Cliente-servidor e a web** 14
- 2.1 A Linguagem de marcação de hipertextos . . . . . 14
- 2.2 O Protocolo de transferência de hipertexto . . . . . 15
- 2.3 Javascript - uma linguagem do lado do cliente . . . . . 16
- 2.4 PHP - uma linguagem do lado do servidor . . . . . 17

**3 Linguagem C** 18
- 3.1 Diretivas de pré-compilação . . . . . 18
- 3.2 Tipos de dados . . . . . 19
  - 3.2.1 Estruturas . . . . . 19
  - 3.2.2 Enumeradores . . . . . 19
  - 3.2.3 Ponteiros . . . . . 20
  - 3.2.4 Vetores . . . . . 20
- 3.3 Tipagem . . . . . 21
- 3.4 Considerações da linguagem C com o Javascript . . . . . 22

**4 Proposta de arquitetura e desenvolvimento** 23
- 4.1 Limitações e características gerais da linguagem . . . . . 23
- 4.2 O navegador como máquina alvo . . . . . 24
  - 4.2.1 Desafios e ferramentas utilizadas . . . . . 24
  - 4.2.2 Requisições assíncronas . . . . . 25
- 4.3 PHP como um roteador de requisições . . . . . 26
- 4.4 O compilador . . . . . 27
  - 4.4.1 Analisador léxico . . . . . 27
  - 4.4.2 Analisador sintático . . . . . 28
  - 4.4.3 Analisador semântico . . . . . 28
  - 4.4.4 Gerenciador de erros . . . . . 29
  - 4.4.5 Gerador de código Javascript . . . . . 29

**5 Conclusão** 32
- 5.1 Trabalhos futuros . . . . . 34

**A Tokens** 35

**B Gramática** 37

# Lista de Figuras

4.1 Ferramenta que transforma o html em um editor de código. . . . . . . . . . . 24
4.2 Visão geral do uso do console e editor de código. . . . . . . . . . . . . . . . . . 26
4.3 diagrama de módulos da arquitetura proposta. . . . . . . . . . . . . . . . . . . 27

# Lista de Tabelas


4.1 Palavras reservadas da linguagem C proposta . . . . . . . . . . . . . . . . . 23
4.2 Lista de erros semânticos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

A.1 Lista de tokens e seus lexemas . . . . . . . . . . . . . . . . . . . . . . . . . . 35
A.2 Lista de tokens e seus lexemas . . . . . . . . . . . . . . . . . . . . . . . . . . 36

B.1 Gramática LL(1) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
B.2 Gramática LL(1) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
B.3 Gramática LL(1) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
B.4 Lista de tokens e seus lexemas . . . . . . . . . . . . . . . . . . . . . . . . . . 40

# Lista de Abreviações

| | |
|---|---|
| DCC | Departamento de Ciência da Computação |
| UFJF | Universidade Federal de Juiz de Fora |
| PHP | Preprocessor Hipertext PHP |
| HTTP | Hypertext Tranfer Protocol |
| HTML | Hpertext Marked Language |
| DOM | Document Object Model |

# 1 Introdução

Um compilador é um programa de computador capaz de realizar a leitura de algoritmos escritos numa linguagem fonte definida, gerando um código - muitas vezes de mais baixo nível - em outra linguagem, objeto, sem alterar a semântica entre as duas. O estudo de compiladores é fundamental na Ciência da Computação tendo em vista que o principal objetivo de um profissional da área é solucionar problemas através de algoritmos escritos numa linguagem de programação. As evoluções de *hardware* em sistemas computacionais causam uma necessidade de evoluir essas linguagens e consequentemente as máquinas (compiladores) que interpretam esses conjuntos de símbolos.

A linguagem de programação C surgiu na década de 1970 com o objetivo de servir de base para o desenvolvimento do sistema operacional UNIX. Foi lançada junto ao livro de referência (DM RITCHIE, 1973) que define as estruturas e funcionalidades da linguagem.

Em paralelo, a constante evolução da linguagem C, a web cresceu exponencialmente em suas primeiras décadas. Consequentemente, as tecnologias ao redor dela acompanharam essa evolução, assim, o que antes era somente um hipertexto (TED NELSON, 1987) onde a única interatividade era a navegação através das ligações, hoje se tornou uma forma de disponibilizar vídeos, áudios e animações vetorias. Surgiram linguagens do lado do cliente e do servidor, que acabaram dando base ao nascimento do termo computação em nuvem, que define a internet como um repositório de documentos e aplicações. Dessa forma, cada vez menos esses dados e aplicações estariam nos computadores pessoais e sim, na web.

Nesse trabalho, foi desenvolvido um compilador da linguagem C a ser disponibilizado como um serviço em nuvem, que se insere no contexto onde há um grande número de alunos com dificuldades em manipular ferramentas de compilação e onde existe um instituto interessado em reduzir custos em manutenção de laboratórios.

## 1.1 Motivação

Nos últimos anos a estrutura de ensino superior mudou, agora muitos alunos, de exatas por exemplo, compartilham várias disciplinas em comum antes de chegarem as específicas de cada curso. Dessa forma, alunos de química, estatística e matemática passam pelas disciplinas básicas de Computação sem o mesmo interesse daqueles que são do curso. Surge-se uma dificuldade em ensinar para esses alunos, diversas vezes desmotivados ou não familiarizados com o tipo de ferramenta usado no curso. O professor acaba desfocando-se da ementa do curso solucionando esses problemas de manuseio de ferramenta.

De outro lado temos os problemas em manter vários laboratórios com o ambiente necessário para ministrar as aulas. Por muitas vezes encontra-se máquinas em laboratórios que não podem ser utilizadas por falta de algum software.

O trabalho proposto se apoia nessas motivações.

## 1.2 Definição do problema

O problema proposto nesse trabalho é definir e prover um compilador que traduza códigos na linguagem C para uma linguagem que rode em navegadores comumente encontrados em quaisquer sistemas operacionais.

## 1.3 Objetivos

O objetivo principal é fornecer um compilador simples de ser usado, que rode no navegador e consiga gerar informações úteis que auxiliem no estudo de disciplinas de Computação. Outros objetivos secundários surgem a partir disso:

- Diminuir a curva de aprendizagem;
- Prover um depurador de código passo a passo;
- Fornecer um disco virtual para usuários salvarem seus arquivos na nuvem;
- Gerar estatísticas sobre tempo de execução;

- Apresentar estado atual da memória.

# 2 Arquitetura Cliente-servidor e a web

Para entender os problemas e soluções no desenvolvimento da ferramenta apresentada nesse trabalho é preciso entender como a web funciona, este capítulo é dedicado aos pilares da arquitetura base da ferramenta que será aprensentada adiante.

Quando há intenção de disponibilizar recursos interligando computadores por uma rede, pode-se utilizar a arquitetura Cliente-servidor. Nela, é definido que esses computadores são participantes do tipo servidor que fornecem recursos - e geralmente são máquinas mais robustas e disponíveis para uma solução específica - ou participantes do tipo cliente, máquinas que interagem com o servidor através de regras definidas.

## 2.1 A Linguagem de marcação de hipertextos

Foi definido em (TED NELSON, 1987) que hipertexto é o texto digital unido a outras informações de mídia como imagens e sons através de ligações (*links*). Em (T. BERNERS-LEE, 1995) a linguagem de marcação que produz hipertextos para a web, o HTML, foi proposta. É uma linguagem baseada no XML, entendida por navegadores web para apresentar os conteúdos das páginas. Abaixo é apresentado um exemplo simples de código na linguagem. O HTML sempre começa com uma primeira marcação html que define um escopo. Dessa forma é possivel definir um modelo que afirma que todo código na linguagem pode ser lido como um caminhamento em árvore, o modelo DOM (*Document Object Model* (L. WOOD, 2000)).

```
1 <html>
2   <head>
3     <title>Compilador web</title>
4   </head>
5   <body>
6     <a href="http://www.ufjf.com/">link para outra pagina</a>
7   </body>
8 </html>
```

## 2.2 O Protocolo de transferência de hipertexto

O protocolo adotado para comunicação cliente-servidor na internet é o HTTP (Hypertext Transfer Protocol), definido em (T. BERNERS-LEE, 1989). A comunicação é feita sob os métodos definidos na especificação do protocolo. Os mais utilizados são:

- GET: Usado quando se tem intenção de solicitar um recurso;
- POST: Método que envia dados, ativando um recurso que causará um processamento no servidor;
- DELETE: Intenção de deletar um recurso;
- PUT: A fim de criar recursos ou modificá-los.

Através desses métodos, o cliente manda uma mensagem para o servidor informando a versão do HTTP, o método utilizado e a localização do recurso necessário, como observa-se no exemplo abaixo.

```
GET /index.html HTTP/1.1
Host: www.ufjf.br
```

O servidor responde a requisição com outra mensagem, mais detalhada, contendo o objeto pedido em caso de sucesso. No exemplo abaixo um código hipertexto simples de resposta é apresentado.

```
HTTP/1.1 200 OK
Date: Mon, 14 May 2013 22:38:34 GMT
Server: Apache/1.3.27 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Etag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Content-Length: 438
Connection: close
Content-Type: text/html; charset=UTF-8

<html>
  <head>
    <title>Test</title>
  </head>
</html>
```

Um sucesso ou uma falha são identificados através de um campo na mensagem de resposta com um código de status. Por exemplo, se o cliente envia uma requisição solicitando um recurso inexistente, uma mensagem de resposta com código 404 é retornada.

Esses códigos se encontram em categorias:

- 1xx Informação
- 2xx Sucesso
- 3xx Redirecionamento
- 4xx Erro de cliente
- 5xx Erro de servidor

Um navegador web conhece o protocolo e encapsula seu funcionamento para que o usuário só veja o conteúdo de resposta das mensagens.
Neste trabalho, o HTTP é o protocolo utilizado para que o código escrito no navegador seja enviado ao servidor, seja compilado e retornado ao cliente como um código interpretável.

## 2.3 Javascript - uma linguagem do lado do cliente

Aos poucos, houve a necessidade de tornar as páginas html mais dinâmicas e interativas. O navegador mais usado na década de 1990, o Netscape, implementou uma linguagem que seria interpretada por ele para que desenvolvedores conseguissem manipular a árvore DOM após o carregamento da página, através de eventos. Era possível ligar um conjunto de ações a serem executadas caso o usuário clicasse em certo local da página, por exemplo. Em alguns anos a linguagem se tornou um padrão para desenvolvimento no navegador e passou a se chamar Javascript.
Todos os navegadores possuem interpretadores Javascript, isso faz com que a linguagem tenha papel fundamental neste trabalho, sendo aquela resultante do compilador e executada no navegador.

## 2.4 PHP - uma linguagem do lado do servidor

Surgia-se também a necessidade de guardar e processar dados em um servidor antes de se apresentar uma página. Logo a arquitetura de aplicações web começou a ganhar novas camadas.
Ao invés da requisição HTTP somente requisitar um arquivo a ser entregue, foram surgindo linguagens que tratam essa requisição, executam um código no servidor para por fim devolver um conteúdo ao cliente. A linguagem PHP (*PHP Hypertext Preprocessor*) trabalha no lado do servidor como um preprocessador que resulta em arquivos html. A linguagem se tornou muito popular pela facilidade de aprendizado. No exemplo abaixo, um arquivo php que será processado em outro arquivo, puramente html, a ser entregue para o navegador do cliente.

```
<html>
  <head>
    <title>Compilador web</title>
  </head>
  <body>
    <?php
         echo "Hello <br />";
    ?>
    <a href="http://www.ufjf.com/">link para outra pagina</a>
  </body>
</html>
```

```
<html>
  <head>
    <title>Compilador web</title>
  </head>
  <body>
     Hello <br />
    <a href="http://www.ufjf.com/">link para outra pagina</a>
  </body>
</html>
```

Neste exemplo, o PHP foi utilizado como ”linguagem de cola”já que pode-se observar que ele trabalhou dentro de marcações HTML. Nada impede que um arquivo puramente PHP seja escrito para manipular requisições HTTP.

Em nosso contexto existe um programa nesta linguagem que funciona como uma camada intermediária, esperando os códigos do cliente (enviados via HTTP) e passando-os para o compilador que se encontra no servidor. Foi a escolhida (dentre milhares de linguagens do lado do servidor) por ser livre e fácil de manipular.

# 3 Linguagem C

A linguagem C surgiu na década de 1970 para o desenvolvimento do sistema operacional Unix, sendo assim uma linguagem de baixo nível altamente indicada a cursos de graduação em Computação.

Neste capítulo serão discutidas as principais características da linguagem, que é aquela utilizada neste trabalho.

## 3.1 Diretivas de pré-compilação

A linguagem C oferece o uso de diretivas para que estruturas e dados sejam acessíveis de forma mais rápida para o compilador, já que antes de compilar esses dados já ficam disponíveis.

```
1 #include <stdio.h>
2 #include "parser.h"
3 #define PI  3.1415926
```

Cria-se arquivos *header* para representar um módulo no programa, nesses arquivos (comumente usando extensão .h) apresenta-se informações públicas para outros módulos como definições de estruturas, constantes e assinatura de funções. Nas linhas 1 e 2 vemos a diretiva "include"que importa um módulo em outro usando o arquivo *header*, a diferença é que na primeira usou-se os operadores de maior e menor pois o módulo representado por "stdio.h"é fornecido pela própria linguagem, e será buscado em diretórios padrão (em sistemas UNIX, será buscado em /usr/include/ ). Na segunda linha, usou-se aspas pois o módulo é encontrado no diretório atual do arquivo que está importando.

A biblioteca "stdio.h"contém funções básicas usadas em qualquer código na linguagem C como comandos de impressão e leitura.

Na linha 3, o uso de "define"faz com que PI seja um valor constante para o código. Da mesma forma é possivel criar rotinas com diretivas, chamadas de macros, nesse contexto. Por questões de tempo, nesse trabalho não houve condições de se trabalhar com um

pré-processador, mas em um trabalho completo essa ferramenta é indispensável.

## 3.2 Tipos de dados

A linguagem C possui como tipos primitivos os inteiros (int), caracteres (char), pontos flutuantes (float) e valores de precisão dupla (double). Algumas palavras reservadas funcionam como modificadores, estendendo esses tipos, como a palavra "long"que tornaria um inteiro de 4 bytes (numa máquina de 32 bits) em um inteiro com domínio maior, utilizando 8 bytes. Neste trabalho há suporte a inteiros, caracteres e pontos flutuantes, exceto utilizando modificadores como "unsigned"e "short".

Além dos tipos primitivos, é possível definir tipos derivados (abstratos) na linguagem

### 3.2.1 Estruturas

Com estruturas (*struct*) é possivel construir tipos de dados que modelem melhor problemas do mundo real, isso é feito definindo campos que caracterizam a estrutura.

```
1 struct Pessoa {
2     int idade;
3     char primeira_letra_nome;
4 };
5 struct Pessoa foo;
```

Na linha 1 apresenta-se uma estrutura Pessoa e na linha 5 uma variável desse tipo é declarada. Neste momento a memória irá alocar 5 bytes, 4 para o variável inteira e 1 para o caractere.

O manuseio de estruturas foi implementado neste trabalho.

### 3.2.2 Enumeradores

Com um enumerador também conseguimos abstrair tipos de dados

```
1 enum Estudantil {
2     int numero_matricula;
3     float numero_profissional;
4 };
```

```
5 enum Estudantil foo;
```

Quando um enumerador é declarado, tem-se que seu maior campo define o tamanho máximo que uma variável consegue conter em sua alocação. No caso acima tem-se que foo pode assumir um dos dois valores, ou ele possui numero_matricula ou numero_profissional, mas nunca os dois.

Neste trabalho não foi implementado o uso de enumeradores por fugir do escopo do problema.

### 3.2.3 Ponteiros

Por ser de baixo nível, a linguagem C provê formas nativas de manipular memória. Além dos tipos já mencionados é possível criar variáveis ponteiro, que pesam apenas uma palavra computacional e guardam outro endereço.

```
1 int numero = 22;
2 int* poteiro;
3 numero = &numero;
```

Na linha 2 uma declaração de um ponteiro é feita para um "lugar"inteiro, ou seja, o ponteiro guarda um endereço de uma variável inteira. Na linha 3 tem-se que esse endereço é o da variável "numero". Abaixo será apresentado um exemplo que declara uma variável ponteiro para uma estrutura, na segunda linha acontecerá uma alocação dinâmica, em tempo de execução uma instrução de baixo nível irá requisitar o tamanho da estrutura em questao.

```
1 struct Pessoa * aluno;
2 aluno = (struct Pessoa*) malloc( sizeof(struct Pessoa) );
```

### 3.2.4 Vetores

Utilizando vetores é possivel que uma variável contenha um conjunto de dados do mesmo tipo. Um dos casos mais importantes sobre, é o de manipulação de vetor de caracteres que também podem ser interpretados como uma String.

```
1 char nome[20];
2  // ....
3 printf("%s", nome );
```

Na linha 1 foi alocado 20 posições contíguas na memória, cada uma para um char. Em seguida, na linha 3, usou-se o comando de impressão para imprimir todas os valores do array, em sequência.

## 3.3 Tipagem

Sobre tipagem de variáveis pode-se encaixar linguagens em duas categorias, uma delas é sobre sua força. Uma linguagem possui tipagem forte se o compilador gera um erro em tempo de compilação ao tentar operar sobre valores de domínios diferentes. No caso da linguagem C, a tipagem é fraca para os tipos primitivos, ou seja, implicitamente a linguagem converte (coerção ou *casting* implícito) tipos para conseguir realizar operações nativas. Abaixo um exemplo da linguagem realizando coeções é apresentado.

```
1 int foo = -2;
2 char bar = 'c';
3
4 printf("%c",  bar + foo);
```

Nas linhas 1 e 2 tem-se a declaração de duas varíaveis de tipos diferentes, uma representa um número inteiro, e a outra um caracter na tabela ASCII. Na linha 4, porém, acontece uma operação (aritimética, de soma) com os dois valores. Na linguagem C assim como espera-se de uma linguagem de tipagem fraca, não há lançamento de erro, ela converte os valores para o tipo primitivo mais básico encontrado, um inteiro, e realiza a operação. Assim 'c' possui o valor inteiro 99 (código ASCII), a operação resulta em 97 e outra coerção acontece (dessa vez explícita), o comando printf pretende imprimir o resultado 97 na saída padrão (*stdout*) do usuário, porém o "%c"indica que a saída se dará como um caractere, e não como um inteiro. A conversão inversa é feita, 97 na tabela ASCII é 'a' . Ainda sobre coerção de dados, considera-se uma equivalência de tipos derivados como estruturas e enumeradores. Nestes casos, a comparação de tipos é feita por nome (e não por estrutura), ou seja, uma variável só pode ser atribuída por outra se as duas

compartilham o mesmo nome de tipo.

Outra categoria sobre tipagens de linguagens as separa em dinâmicas e estáticas. Uma linguagem com tipagem dinâmica define que suas declarações de variáveis não amarram a alocação de espaço a um tipo de dado pré-definido, ou seja, uma variável é declarada e pode assumir valores de domínios diferentes ao longo do código, o tipo estão associado ao tempo de execução. O PHP é uma linguagem de tipagem dinâmica, como observa-se no código abaixo.

```
1 $foo = 22;
2 // ... codigos depois
3
4 $foo = "Cloud";
```

Nas linhas 1 e 4 tem-se que a declaração da variável "$foo"define o tipo apenas pela atribuição, e que é possível sobreescrever um valor da variavel com outro tipo.
Na linguagem C, isso não ocorre, ela possui tipagem estática, logo é preciso definir os tipos com palavras reservadas.

## 3.4 Considerações da linguagem C com o Javascript

Encontra-se vários problemas ao se converter códigos escritos em linguagem C para Javascript. A linguagem C, como foi discutido, trabalha com o conceito de módulos com arquivos *headers* e todas as funções utilizadas na linguagem são encontradas nesses arquivos, importados pelo pré-processador. É dficíl dar suporte a todos os *headers* convertendo-os para Javascript, logo neste trabalho as únicas funções nativas encontradas são aquelas de impressão e leitura em console ("printf"e "scanf").
Em Javascript, não existe nenhum tipo análogo a estruturas ou classes, existindo o problema de converter as estruturas em C para algum tipo rotulável em javascript.

# 4 Proposta de arquitetura e desenvolvimento

Neste capítulo será discutido a proposta de desenvolvimento de um compilador que soluciona o problema apresentado anteriormente.

## 4.1 Limitações e características gerais da linguagem

O desenvolvimento de compiladores está ligado a teoria da computação e linguagens formais. O primeiro passo foi definir o que a linguagem aceitará ou não. Por ser um compilador focado em atender estudantes de Computação, algumas caraterísticas foram abandonadas para dar mais ênfase a outras, abaixo encontram-se essas características e em seguida as palavras reservadas da linguagem implementada.

- Ponteiros e alocações dinâmicas;
- Pré-processador (sintaticamente é possível usar #include );
- Uso de múltiplos arquivos;
- Enumeradores;
- Modificadores de tipos primitivos (como *long*, *short*, *unsigned* );
- Apenas erros fatais são gerados;
- Uso de *static*, *volatile*, *union*, *goto* e *const*.

Tabela 4.1: Palavras reservadas da linguagem C proposta

| int | float | char | struct |
|---|---|---|---|
| if | else | while | for |
| switch | typedef | return | |

## 4.2 O navegador como máquina alvo

Atualmente vários navegadores web estão disponíveis no mercado, cada um com seus núcleos de renderização de páginas e interpretadores de HTML e Javascript. Desenvolvedores web se preocupam com problemas de compatilibidade de código cliente entre essas máquinas. Este trabalho foi testado no navegador Mozilla Firefox 19[1], mas suas funcionalidades são acessíveis por qualquer navegador moderno. O objetivo é que o código javascript gerado no servidor seja executado no navegador cliente, reduzindo uma possível sobrecarga do servidor.

### 4.2.1 Desafios e ferramentas utilizadas

Um dos primeiros desafios encontrados no lado do cliente foi o de simular um editor de código no navegador, algumas ferramentas foram pesquisadas e a escolhida foi a Ace [2] [3], uma bilbioteca javascript que transforma um container HTML em um editor, com marcação de código e atalhos, como podemos ver na figura 4.1

```
1  #include <stdio.h>
2  int main()
3  {
4      printf("Formei!");
5      return 0;
6  }
```

Figura 4.1: Ferramenta que transforma o html em um editor de código.

Outro desafio era de como apresentar os dados do código gerado e executado no navagedor, via HTML. A melhor forma, era simular um console na prória página.

[1]http://firefox.com - acessado dia 19/03/2013 às 16:44
[2]http://ace.ajax.org - acessado em 20/03/2013 às 09:50
[3]https://github.com/ajaxorg/ace - acessado em 20/03/2013 às 09:52

Utilizamos uma ferramenta que provê uma interface para criar e manipular elementos html como se fosse um console. Essa ferramenta é chamada de jqconsole [4], um dos maiores ganhos de utilizá-la é que com ela é possível emitir erros no console, assim como impressões simples (traduzidas de comandos *printf* na linguagem C).

Abaixo uma geração de código para o problema de impressão de dados é apresentada.

```
1 // codigo em C: printf ("Voce tem %d anos de idade", 22);
2 jqconsole.Write(
3      sprintf("Voce tem %d anos de idade",22),
4 'jqconsole-output');
```

Perceba outra ferramenta [5] foi utilizada (função sprintf em javascript não é nativa) para formatar strings em javascript, já que precisávamos fazê-lo para os comandos de impressão e leitura. Além de poder exibir dados no console, outro grande desafio era o de usá-lo como *stdin*, ou seja, o console precisava esperar dados de entrada para que a execução continuasse.

Veja o código convertido abaixo.

```
1 // codigo em C:
2 // scanf("%d",&numero);
3 // return numero + 1;
4
5 jqconsole.Input(function(input) {
6     numero = input;
7     return numero + 1;
8 });
```

Percebe-se que o código javascript gerado usa a função Input da biblioteca jqconsole e esta espera uma função anônima que será executada assim que o usuário digitar algo no console. O problema é que tudo que vier depois de um scanf precisa ser convertido para instruções dentro dessa função de retorno.

Na imagem 4.2 encontra-se um navegador acessando a ferramenta proposta.

### 4.2.2 Requisições assíncronas

Quando o usuário digita o código a ser compilado e clica no botão de executar, um evento javascript é ativado, nele pega-se o código e encapsula-o numa requisição para o servidor.

[4]https://github.com/replit/jq-console - acessado em 20/03/2013 às 13:11
[5]http://www.diveintojavascript.com/projects/javascript-sprintf acessado em 21/03/2013 às 17:07

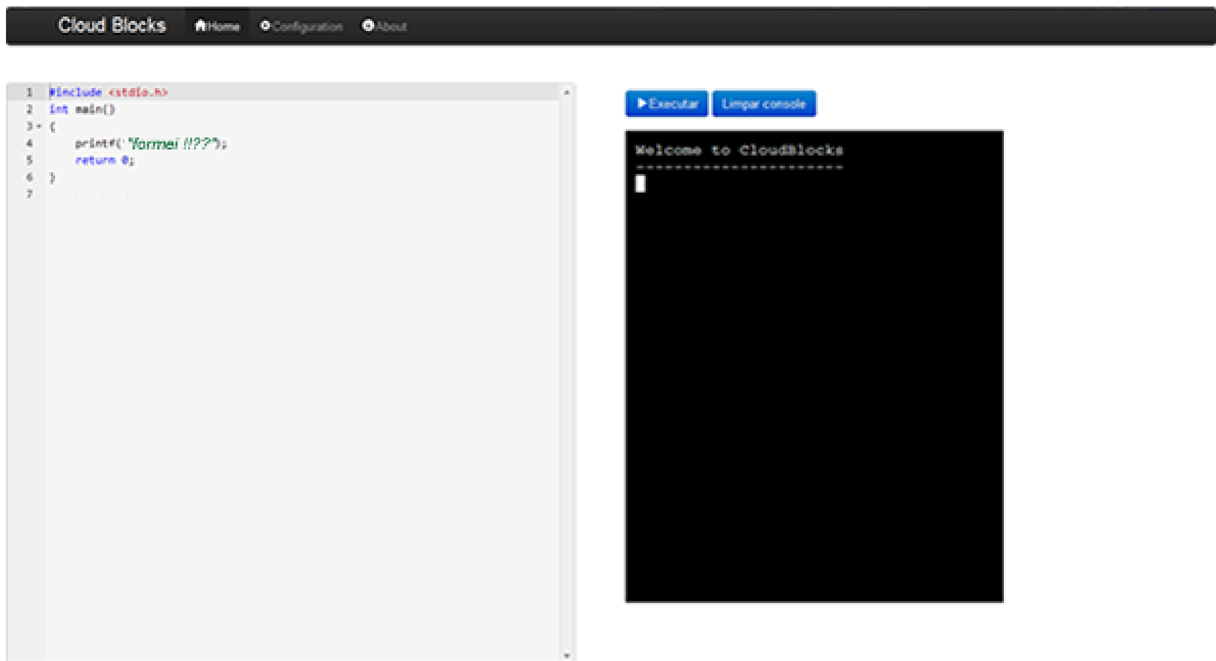


Figura 4.2: Visão geral do uso do console e editor de código.

Essa requisição utiliza um objeto javascript chamado *XMLHttpRequest*, em um método nomeado AJAX. Neste, o navegador consegue fazer requisições ao servidor sem recarregar a página. Quando a resposta HTTP chega no navegador ativamos uma função de retorno (*callback*) que tratará essa resposta, executando o código compilado em caso de sucesso, ou emitindo erros, caso contrário.

## 4.3 PHP como um roteador de requisições

Quando o navegador envia a requisição assíncrona (usando método POST) com o código fonte do usuário, na linguagem C, espera-se que o servidor processe esse código gerando a linguagem alvo (Javascript) a ser executada pelo próprio navegador.

No servidor um programa em php acaba agindo como um roteador, que atende a requisição da melhor forma. É possível utilizar esse código como um balanceador de carga em uma rede de computadores ou mesmo salvar em um banco de dados um histórico de compilação dos usuários.

No caso de nossa proposta, utiliza-se uma máquina servidor e apenas compila-se o código. Então o php pega a requisição, gera um arquivo local com o código do usuário (esse código só pode ser acessado novamente pelo próprio usuário por uma questão de segurança, graças

a um identificar único) e inicia-se um processo do compilador passando o arquivo gerado. Por fim, ele lê o código de saída da execução (todo programado rodado em linha de comando possui um código de saída, como um *magic number*) e identifica se ocorreu um erro sintático, semântico ou se o código pode ser executado sem problemas.

Por fim, dados impressos na saída padrão do console são encapsulados num formato JSON (D. Crockford, 2006) e inseridos na mensagem de resposta do HTTP.

## 4.4 O compilador

O desenvolvimento do compilador é divido em módulos, apresentados na figura 4.3.

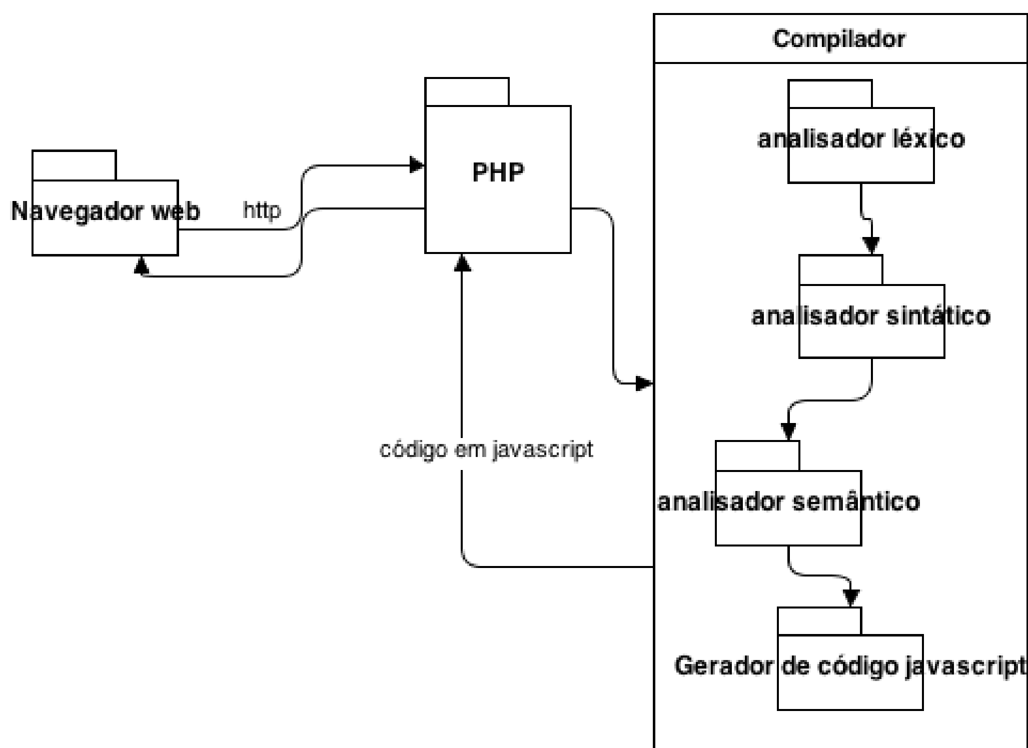


Figura 4.3: diagrama de módulos da arquitetura proposta.

### 4.4.1 Analisador léxico

A primeira fase no desenvolvimento de um compilador é a análise léxica. Uma instância desse analisador tem como objetivo ler o arquivo fonte e possuir um método que identifique e retorne um *token* (um identificador único) que classifica o lexema atual a ser lido. Um lexema nada mais é que uma cadeia de caracteres pertencente a grupos como o das palavras reservadas, dos símbolos de sintaxe ou dos valores literais.

No apêndice desse documento encontra-se a lista de tokens utilizados no trabalho.
Encontra-se abaixo a tabela com os mais importantes

Para agilizar o processo utilizamos o LEX (ME Lesk, 1975), uma ferramenta que gera analisadores léxicos facilmente.

### 4.4.2 Analisador sintático

Baseado nas funcionalidades dispensadas para a proposta de linguagem, foi definida uma gramática BNF (LM Garsho, 2003) que representa a sintaxe da linguagem.
O analisador sintático (*parser*) foi implementado de forma LL(1), isso quer dizer que ele faz uma leitura de código da esquerda para a direita e realiza derivação mais a esquerda utilizando um token para a tomada de decisão sobre qual caminho escolher.
Na gramática BNF (encontrada no apêndice desse documento) Nesta fase do compilador o objetivo é gerar uma estrutura conhecida como árvore de sintaxe abstrata, uma representação da estrutura de código através de objetos interligados. A partir da gramática foram implementadas funções que pedem os próximos tokens e baseado nisso chamam outras funções e retornam partes da árvore.
É importante salientar que analisadores LR(1) são mais indicados e performáticos, assim como uma abordagem bottom-up (onde o caminhamento na árvore é feito a partir das folhas), mas essas teorias não foi utilizada.

### 4.4.3 Analisador semântico

A intenção dessa fase é de buscar os últimos erros na árvore de sintaxe abstrata, percorre-se a árvore através do padrão de projeto (*design patterns*) Visitor (E Gamma, 1995).
Na tabela 4.2 encontra-se todos os erros semânticos.
Uma importante teoria para definir erros semânticos gira em torno de escopos para discernir, por exemplo, se uma variável existe em um determinado contexto. Foi implementada uma classe Escopo, que guarda uma estrutura de dados que contém todas as declarações de variáveis no contexto. Além disso ponteiros para Escopo foram criados para criar uma pilha.
Abaixo, observa-se um exemplo de declaração eu dois escopos diferentes.

```
1 // ...
2 int numero;
3 int main() {
4     Estrutura_conhecida numero;
5     numero = 5;
6 }
```

Semanticamente, observa-se dois níveis de escopo, sempre que o programa começa, um escopo (nível zero) é criado, neste caso uma variável inteira "numero"é associada a este escopo. Quando é encontrada uma declaração de função, outro escopo (nível um) é criado, neste momento o escopo anterior aponta para o escopo criado e o analisador léxico (sempre) guarda o escopo atual, montando uma estrutura de pilha.

Agora, "numero"representa uma variável do tipo Estrutura_conhecida, sendo guardada no escopo de nível atual.

Quando, na linha 5, a variável número é acessada, ela é buscada via caminhamento na pilha de escopos, começando do nível atual até o nível zero. Ela será encontrada no nível atual e por ser uma estrutura, um erro semântico irá ser lançado na atribuição entre estrutura e inteiro.

Quando saímos de uma função, desempilhamos o escopo atual da pilha.

A representação de variáveis no código, é feito pos classes chamadas Símbolos, apesar dos símbolos serem guardados no escopo, que serão descartados em algum momento, eles também são apontados pela árvore de sintaxe abstrata para serem usados na geração de código final.

### 4.4.4 Gerenciador de erros

Durante todas as fases de compilação, erros são identificados ou gerados, os analisadores sintático e semântico possuem um atributo/instância do gerenciador de erro.

Se o gerenciador de erro emitiu qualquer erro durante a compilação, a árvore não tem condições de ser convertida para javascript, e a compilação para logo após a fase corrente.

No analisador sintático todo erro tem a forma "Esperado X encontrado Y na linha Z". Enquanto que os erros semânticos são mais detalhados.

Tabela 4.2: Lista de erros semânticos

| Constante de erro | Mensagem |
|---|---|
| 1 | Redeclarando variável %s na linha %d |
| 2 | Esperado um tipo %s e encontrado um tipo %s em uma atribuição |
| 3 | Variável %s não declarada na linha %d |
| 4 | Atributo %s não declarado no tipo %s na linha %d |
| 5 | Redeclarando função %s na linha %d |
| 6 | Retorno inválido, esperado %s e encontrado %s na linha %d |
| 7 | Redeclarando tipo %s na linha %d |
| 8 | Tipo %s não declarado na linha %d |
| 9 | Função %s não declarado na linha %d |
| 10 | Returno do tipo %s esperado na linha %d |
| 11 | Esperada expressão em sentença if na linha %d |
| 12 | Assinatura de função inválida, o candidato é %s na linha %d |
| 13 | Variável %s é do tipo %s ao invés de um vetor, na linha %d |
| 14 | Atributo %s inválido em tipo %s na linha %d |
| 15 | Vetor %s usando dimensão inválida na linha %d |
| 16 | Formato inválido em scanf na linha %d |
| 17 | Acesso inválido com variável %s' na linha %d |

### 4.4.5 Gerador de código Javascript

Nesta fase outro visitor foi gerado com o objetivo de transformar a árvore em código javascript.

Esse visitor, chamado Converter na implementação, é representado por uma classe, que ao percorrer a árvore, imprime na saída padrão o código objeto. Abaixo observa-se um exemplo de uma função que representa a visita do conversor ao nó da árvore que ilustra uma declaração de *while*.

```
1  void Converter::visit(node_while* node){
2       char* instr = new char[200];
3       sprintf(instr,"while(");
4       this->addJs(instr);
5       if (node->attr_expr) node->attr_expr->accept(this);
6       sprintf(instr,"){\n");
7       this->addJs(instr);
8       if (node->attr_stmt) node->attr_stmt->accept(this);
9       sprintf(instr,"}\n");
10      this->addJs(instr);
11 }
```

Algumas diferenças entre C e Javascript foram encontradas, como a falta de classes ou estruturas. Para solucionar esse problema, converteu-se estruturas C em funções Javascript, já que nesta linguagem, funções são estruturas de primeira ordem, podendo

ser associadas a variáveis ou passadas por parâmetro.

```
// codigo em c
struct Pessoa {
    int idade;
};
struct Pessoa joao;

//codigo javascript gerado
function str_Pessoa{
     this.idade = 0;
}
var joao = new str_Pessoa();
```

# 5 Conclusão

Conclui-se que a proposta de trabalho é válida, possível de ser implementada e que o objetivo proposto foi alcançado.

O compilador [6] demonstrou-se capaz de resolver exemplos comuns no ensino básico de Computação, manipulando variáveis primitivas, estruturas, laços e condicionais. Abaixo encontra-se um exemplo que demonstra todas essas funcionalidades importantes e sua conversão em código javascript.

```
1  #include "stdio.h"
2  int main(){
3      int numbers[10];
4
5      int i = 9;
6      while(i >= 0){
7          numbers[i] = i;
8          printf("%d\n",numbers[i]);
9          i = i - 1;
10     }
11
12     i = 0;
13     while(i < 10){
14         int j = 0;
15         while (j < 10 ) {
16             if (numbers[i] < numbers[j]) {
17                 int aux = numbers[i];
18                 numbers[i] = numbers[j];
19                 numbers[j] = aux;
20             }
21             j = j + 1;
22         }
23         i = i + 1;
24     }
25     printf("============ Ordernado? =============\n");
26     i = 0;
27     while(i < 10){
28         printf("%d\n",numbers[i]);
29         i = i + 1;
30     }
31     return 5;
32 }
```

[6]http://www.gcg.ufjf.br/cloudblocks

```
var return_var = null;
function main(){
    var numbers;
    numbers =  { } ;
    var i;
    i = 9;
    while(i >= 0){
        numbers[i] = i;
        var print_6691676  = numbers[i];
        jqconsole.Write(sprintf("%d\n",print_6691676), 'jqconsole
            -output');
        i = i - 1;
    }
    i = 0;
    while(i < 10){
        var j;
        j = 0;
        while(j < 10){
            if(numbers[i] < numbers[j]){
                var aux;
                aux = numbers[i];
                numbers[i] = numbers[j];
                numbers[j] = aux;
            }
            j = j + 1;
        }
        i = i + 1;
    }
    jqconsole.Write("=========== Ordernado? ============\n", '
        jqconsole-output');
    i = 0;
    while(i < 10){
        var print_6691677  = numbers[i];
        jqconsole.Write(sprintf("%d\n",print_6691677), 'jqconsole
            -output');
        i = i + 1;
    }
    return false;
}

main();
```

Além da dificuldade de gerar um analisador sintático completo, os maiores esforços ficaram ao redor do gerador de código final, sabido que as diferenças entre a linguagem fonte e objeto muitas vezes tornou-se um problema.

Nesta implementação , um depurador de código não foi concluído por questões de tempo e dificuldade, ficando como proposta de trabalho futuro. O depurador daria uma interface para que usuários pudessem acompanhar o estado de memória passo a passo, imagina-se

que neste modo *debug*, cada sentença javascript deve ser encapsulada em uma função. A partir disso, chama-se cada função a medida que o usuário desejar seguir para o próximo passo.

## 5.1 Trabalhos futuros

São várias as possibilidades de trabalhos futuros envolvendo a ferramenta proposta.

É possível integrá-la com outros sistemas da universidade, fornecendo contas para que os alunos possam, por exemplo, usar a ferramenta como um repositório de código e acessá-lo de qualquer lugar.

Durante a aula os professores poderiam ter acesso ao desenvolvimento dos alunos, solucionando dúvidas constantes. Outra possibilidade para professores seria o uso em provas de laboratório. O aluno escreveria seus códigos e para concluir o teste submetendo os algoritmos, marcaria um campo de conclusão. Usar a ferramenta nesse caso ainda pode solucionar problemas de comunicação entre alunos durante a prova, estatísticas podem ser usadas para saber quantas vezes o aluno tentou solucinar certas questões ou a semelhança entre resultados de alunos.

Monitores das disciplinas envolvidas também poderiam usar a ferramenta e incentivar os alunos durante a monitoria, assim seria possível descobrir o intervalo de tempo em que a ferramenta ficou aberta e o número de códigos que o aluno, com dúvidas, rodou.

Um outro grande acréscimo seria o de utilizar a ferramenta no ensino de educação a distância, esses alunos sentem-se muitas vezes carentes de como acompanhar as aulas, já que não possuem o suporte presencial dos professores.

# A Tokens

Tabela A.1: Lista de tokens e seus lexemas

| T_EOF | carateter 'end of file' |
|---|---|
| T_BREAK | break |
| T_CASE | case |
| T_CHAR | char |
| T_CONTINUE | continue |
| T_DEFAULT | default |
| T_DOUBLE | double |
| T_ELSE | else |
| T_FLOAT | float |
| T_FOR | for |
| T_IF | if |
| T_INT | int |
| T_RETURN | return |
| T_SIZEOF | sizeof |
| T_SWITCH | switch |
| T_STRUCT | struct |
| T_ADD_ASSIGN | += |
| T_ADD | + |
| T_SUB_ASSIGN | -= |
| T_SUB | - |
| T_MULT_ASSIGN | *= |
| T_MULT | * |
| T_DIV | / |
| T_DIV_ASSIGN | /= |
| T_MOD | % |
| T_MOD_ASSIGN | %= |
| T_INC | ++ |
| T_DEC | – |
| T_PTR | -≫ |
| T_AND | && |
| T_OR | \|\| |
| T_BEGIN | { |
| T_END | } |
| T_COMMA | , |
| T_SEMICOLON | ; |

Tabela A.2: Lista de tokens e seus lexemas

| T_QUEST | ? |
|---|---|
| T_LESS | ≪ |
| T_LESS_EQ | ≪= |
| T_GREATER | ≫ |
| T_GREATER_EQ | ≫= |
| T_EQUAL | == |
| T_NOT_EQUAL | != |
| T_DOT | ! |
| T_BRACKET_L | [ |
| T_BRACKET_R | ] |
| T_PARENT_L | ( |
| T_PARENT_R | ) |
| T_WHILE | while |
| T_VOID | void |
| T_LITERAL | ”.*” |
| T_CONSTANT | [0-9]+e0,1.[0-9]* |
| T_ID | ”[a-zA-Z_]+” |
| T_STRING_LITERAL | ”string” |
| T_LONG | ”long” |
| T_STATIC | ”static” |
| T_TYPEDEF | ”typedef” |
| T_UNION | ”union” |
| T_DO | ”do” |
| T_ADDRESS | ”&” |
| T_DOT_DOUBLE | ”:” |
| T_NEG | ”!” |
| T_ASSIGN | ”=” |
| T_PIPE | ”\|” |
| T_XOR | ”xor” |
| T_INCLUDE | ”#include” |
| T_PRINTF | ”printf” |
| T_SCANF | ”scanf” |

# B Gramática

Tabela B.1: Gramática LL(1)

| Não terminal | produções |
|---|---|
| G_PROGRAM | G_INCLUDE G_MAIN |
| G_INCLUDE | #include G_INCLUDE_INT |
| G_INCLUDE | vazio |
| G_INCLUDE_INT | literal G_INCLUDE |
| G_INCLUDE_INT | $\ll$ T_ID G_INCLUDE_ROOT $\gg$ G_INCLUDE |
| G_INCLUDE_ROOT | . T_ID |
| G_INCLUDE_ROOT | . vazio |
| G_MAIN | G_DECL G_MAIN |
| G_MAIN | vazio |
| G_DECL | G_TYPE G_POINTER T_ID G_FUNC_OR_VAR |
| G_TYPE | int |
| G_TYPE | char |
| G_TYPE | float |
| G_TYPE | void |
| G_TYPE | T_ID |
| G_POINTER | * G_POINTER |
| G_POINTER | vazio |
| G_FUNC_OR_VAR | ( G_PARAMS ) G_SCOPE |
| G_FUNC_OR_VAR | G_RVAR |
| G_RVAR | = G_EXPR G_RVAR |
| G_RVAR | , G_TYPE T_ID G_RVAR |
| G_RVAR | ; |
| G_SCOPE | G_STMT_LIST |
| G_STMT_LIST | G_STMT G_STMT_LIST |
| G_STMT_LIST | vazio |

Tabela B.2: Gramática LL(1)

| | |
|---|---|
| G_STMT | if( G_EXPR ) G_SCOPE G_ELSE |
| G_STMT | while( G_EXPR ) G_SCOPE |
| G_STMT | scanf( literal , & T_ID ) ; |
| G_STMT | printf( literal, G_EXPR ) ; |
| G_STMT | return G_EXPR ; |
| G_STMT | for ( G_FOR_INIT ; G_EXPR; G_FOR_INT_INC ) G_SCOPE |
| G_STMT | switch ( G_EXPR ) G_CASES |
| G_STMT | T_ID G_DECL_ID |
| G_STMT | G_TYPE G_POINTER T_ID G_RVAR |
| G_DECL_ID | G_POINTER T_ID G_RVAR |
| G_DECL_ID | G_ID |
| G_ELSE | else G_SCOPE |
| G_ELSE | vazio |
| G_PRINTF_INT | T_ID |
| G_PRINTF_INT | literal |
| G_PRINTF_INT | number |
| G_FOR_INT | G_FOR_INT_TYPE T_ID G_FOR_INT_ASSIGN |
| G_FOR_INT_TYPE | G_TYPE |
| G_FOR_INT_TYPE | vazio |
| G_FOR_INT_ASSIGN | = G_EXPR |
| G_FOR_INT_INC | T_ID G_FOR_INT_INC_INT |
| G_FOR_INT_INC_INT | ++ |
| G_FOR_INT_INC_INT | – |

Tabela B.3: Gramática LL(1)

| G_CASES | case G_CASE_VALUE : G_CASE_SCOPE |
|---|---|
| G_CASE_VALUE | literal |
| G_CASE_VALUE | number |
| G_PARAMS | G_TYPE G_POINTER T_ID G_ARRAY G_PARAMS_INT |
| G_PARAMS | vazio |
| G_PARAMS_INT | , G_PARAMS |
| G_PARAMS_INT | vazio |
| G_ARRAY | [ ] G_ARRAY |
| G_ARRAY | vazio |
| G_ID | ( G_EXPR_LIST ) ; |
| G_ID | G_ARRAY_ACCESS_OPTIONAL = G_EXPR; |
| G_ID | . T_ID G_ID_OR_SEMICOLON |
| G_ID | -≫ T_ID G_ID_OR_SEMICOLON |
| G_ID_OR_SEMICOLON | G_ID |
| G_ID_OR_SEMICOLON | ; |
| G_ARRAY_ACCESS_OPTIONAL | G_ARRAY_ACCESS |
| G_ARRAY_ACCESS_OPTIONAL | vazio |
| G_ARRAY_ACCESS | [ G_EXPR ] G_ARRAY_ACCESS_OPTIONAL |
| G_EXPR_LIST | G_EXPR G_EXPR_LIST |
| G_EXPR | B2 \| |
| G_EXPR | G_CAST |
| G_CAST | (G_TYPE G_POINTER) G_EXPR G_CAST_INT |
| G_CAST_INT | G_ARRAY_ACCESS |

Tabela B.4: Lista de tokens e seus lexemas

<table>
<tr><td>EXPR_</td><td>= B2 EXPR_</td></tr>
<tr><td>EXPR_</td><td>vazio</td></tr>
<tr><td>B2</td><td>B3 B2_</td></tr>
<tr><td>B2_</td><td>| | B3 B2_</td></tr>
<tr><td>B2_</td><td>vazio</td></tr>
<tr><td>B3</td><td>B4 B3_</td></tr>
<tr><td>B3_</td><td>&& B4 B3_</td></tr>
<tr><td>B3_</td><td>vazio</td></tr>
<tr><td>B4</td><td>B5 B4_</td></tr>
<tr><td>B4_</td><td>== B5 B4_</td></tr>
<tr><td>B4_</td><td>!= B5 B4_</td></tr>
<tr><td>B4_</td><td>vazio</td></tr>
<tr><td>B5</td><td>B6 B5_</td></tr>
<tr><td>B5_</td><td>≪ B6 B5_</td></tr>
<tr><td>B5_</td><td>≪= B6 B5_</td></tr>
<tr><td>B5_</td><td>≫ B6 B5_</td></tr>
<tr><td>B5_</td><td>≫= B6 B5_</td></tr>
<tr><td>B5_</td><td>vazio</td></tr>
<tr><td>B6</td><td>B7 B6_</td></tr>
<tr><td>B6_</td><td>+ B7 B6_</td></tr>
<tr><td>B6_</td><td>- B7 B6_</td></tr>
<tr><td>B6_</td><td>—B7 B6_</td></tr>
<tr><td>B6_</td><td>vazio</td></tr>
<tr><td>B7</td><td>B8 B7_</td></tr>
<tr><td>B7_</td><td>* B8 B7_</td></tr>
<tr><td>B7_</td><td>/ B8 B7_</td></tr>
<tr><td>B7_</td><td>% B8 B7_</td></tr>
<tr><td>B7_</td><td>& B8 B7_ —</td></tr>
<tr><td>B7_</td><td>vazio</td></tr>
<tr><td>B8</td><td>+G_PRIMARY</td></tr>
<tr><td>B8</td><td>- G_PRIMARY</td></tr>
<tr><td>B8</td><td>! G_PRIMARY</td></tr>
<tr><td>B8</td><td>G_PRIMARY</td></tr>
<tr><td>G_PRIMARY</td><td>( G_EXPR_OR_CAST )</td></tr>
<tr><td>G_PRIMARY</td><td>T_ID G_ID</td></tr>
<tr><td>G_PRIMARY</td><td>literal</td></tr>
<tr><td>G_PRIMARY</td><td>number</td></tr>
<tr><td>G_EXPR_OR_CAST</td><td>G_CAST</td></tr>
<tr><td>G_EXPR_OR_CAST</td><td>G_EXPR</td></tr>
</table>

# Referências Bibliográficas

Crockford, D. In: The application/json Media Type for JavaScript Object Notation (JSON), 2006.

Kernighan, B. W.; Ritchie, D. M. In: The C Programming Language, 1973.

Gamma, E.; Helm, R.; Johnson, R. ; Vlissides, J. In: Design patterns: Elements of reusable object-oriented design, 1995.

Lauren Wood, Arnaud Le Hors, V. A. S. B. M. C. I. J. S. I. G. N. J. R. R. S. C. W. In: Document object model (DOM) level 1 specification, 2000.

Garshol, L. M. In: BNF and EBNF: What are they and how do they work, 2003.

Lesk, M. E.; Schmidt, E. In: Lex: A lexical analyzer generator, 1975.

Tim Berners-Lee, D. C. In: HTTP RFC 1866, 1989.

Tim Berners-Lee, H. Frystyk, J. G. J. M. R. F. In: HTTP RFC 2616, 1989.

Nelson, T. In: Computer Lib/Dream Machines, 1987.