UNIVERSIDADE FEDERAL DE GOIÁS – UFG

CAMPUS CATALÃO – CaC

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO – DCC

Bacharelado em Ciência da Computação

Projeto Final de Curso

Estudo de um compilador sob a perspectiva de um sistema online de programação

Autor: Nélio Carneiro Júnior

Orientador: Ms. Thiago Jabur Bittar

Catalão - 2011

Nélio Carneiro Júnior

Estudo de um compilador sob a perspectiva de um sistema online de programação

Monografia apresentada ao Curso de
Bacharelado em Ciência da Computação da
Universidade Federal de Goiás Campus Catalão
como requisito parcial para obtenção do título de
Bacharel em Ciência da Computação

Área de Concentração: Compiladores
Orientador: Ms. Thiago Jabur Bittar

Catalão - 2011

Carneiro, Nélio

Estudo de um compilador sob a perspectiva de um sistema online de programação/Ms. Thiago Jabur Bittar- Catalão - 2011

Número de paginas: 38

Projeto Final de Curso (Bacharelado) Universidade Federal de Goiás, Campus Catalão, Curso de Bacharelado em Ciência da Computação, 2011.

Palavras-Chave: 1. Compilador. 2. Sistemas Web. 3. Linguagem de Programação

Nélio Carneiro Júnior

Estudo de um compilador sob a perspectiva de um sistema online de programação

Monografia apresentada e aprovada em _____ de _____
Pela Banca Examinadora constituída pelos professores.

_____

Ms. Thiago Jabur Bittar – Presidente da Banca

_____

Professor 1

_____

Professor 2

Dedico este trabalho

AGRADECIMENTOS

Por fim, agradeço a todos que de forma direta ou indireta que cotribuíram para a minha formação.

"Os desafios.

Sulamita

Sumário

Lista de Figuras

Introdução

É evidente o crescente número de usuários conectados à internet ao longo do tempo. Percebe-se também que esses usuários estão cada vez mais necessitados de que suas ferramentas, antes usadas em seus desktops, fiquem disponíveis online, de prontidão, sempre que se fizer necessário, mesmo estando conectados longe de casa. Fez-se então necessário que o desenvolvimento de softwares caminhasse neste mesmo sentido.

Um aspecto interessante desse processo a ser analisado são as aplicações que dependem de um compilador, ou seja, programas que quando instalados na máquina sempre fazem uso de um compilador ou então, são eles mesmos, compiladores. Um exemplo desse tipo de software seria um compilador para Portugol. Como iria se comportar tal compilador ao ser usado na internet? Como se daria a implementação deste compilador? Quais aspectos a serem tratados? E como seria o sistema na web que necessitasse desse compilador?

O objetivo principal deste trabalho é analisar o comportamento de um compilador feito para a linguagem Portugol, trabalhando sob um sistema de progamação na Web. A proposta é fazer um estudo detalhado da implementação desse compilador, tendo assim uma base sólida para o desenvolvimento do mesmo, visando na prática tecer conclusões à respeito das técnicas utilizadas.

Este trabalho também tem como objetivo desenvolver uma aplicação Web que irá utilizar o compilador. A aplicação é um editor o qual o usuário faz seus programas em Portugol, tendo de imediato a resposta do compilador. A intenção é analisar como irá se comportar o compilador, bem como, verificar a viabilidade de uso de tal aplicação, ao se implementar utilizando as técnicas descritas ao longo deste trabalho. A aplicação tem um caráter educativo, no qual o usuário faz programas e, ele e seus trabalhos, podem ser acompanhados por um instrutor, visando assim verificar os passos realizados pelo usuário, bem como a maneira pela qual o compilador tem sido usado.

Uma grande vantagem da criação de tal sistema será o fato dele poder ser acessado de qualquer lugar e qualquer máquina, independente do sistema operacional, bastando apenas um navegador. Como exemplo, podemos citar uma faculdade que possui bastantes máquinas e algumas inclusives com sistemas operacionais diferentes. Outro fator são as atualizações. Tendo um compilador online, basta atualizar em um só lugar e a mudança ocorrerá para todos, diferente de um compilador instalado em máquinas locais.

No capítulo 1 são abordados os conceitos de compiladores, as técnicas utilizadas na criação do compilador e como se dará o seu funcionamento diante uma aplicação Web.

No capítulo 2 é feita uma apresentação da linguagem do compilador: o Portugol. Suas características e o embasamento para a criação do compilador baseado nesta linguagem também fará parte deste capítulo

No capítulo 3 pode-se observar um levantamento sobre o estado da arte dos compiladores e como tem sido o uso destes em aplicações na Web. Neste também irei citar casos em que editores/compiladores foram colocados na Web.

No capítulo 4 é discorrido sobre a metodologia proposta para o desenvolvimento do protótipo de editor/compilador Web.

No capítulo 5 é abordado todo o processo de testes, além de ser realizado uma análise dos resultados. Por fim há uma conclusão do trabalho realizado, levantando os objetivos alcançados, pontos positivos, pontos negativos, dificuldades e trabalhos futuros.

Ainda é possível encontrar o código fonte do compilador e da aplicação Web criada, além de toda a documentação de testes elaborada no apêndice desta monografia.

Capítulo 1

Compiladores

1.1 Introdução

Criado por volta dos anos 50, o nome Compilador se refere ao processo de composição de um programa através da reunião de várias rotinas de bibliotecas. O processo de tradução (de uma linguagem fonte para uma linguagem objeto), considerado hoje a função central de um compilador, era então conhecido como programação automática [Rangel, 1999].

Definido em [Aho et al., 1995], um compilador é um programa que lê outro programa escrito em uma linguagem — a linguagem de origem — e o traduz em um programa equivalente em outra linguagem — a linguagem de destino. Como uma importante parte no processo de tradução, o compilador reporta ao seu usuário a presença de erros no programa origem.

Ao longo dos anos 50, os compiladores foram considerados programas notoriamente difíceis de escrever. O primeiro compilador Fortran, por exemplo, consumiu 18-homens ano para implementar [Backus, 1957]. Desde então, foram descobertas técnicas sistemáticas para o tratamento de muitas das mais importantes tarefas desenvolvidas por um compilador.

A variedade de compiladores nos dias de hoje é muito grande. Existem inúmeras linguagens fontes, as quais poderiam ser citadas em várias páginas deste trabalho. Isso se deve principalmente ao fato de que com o aumento do uso dos computadores, aumentou também, as necessidades de cada indivíduo, sendo essas específicas, exigindo por sua vez linguagens de programação diferentes. Este processo — juntamente com a evolução da tecnologia de desenvolvimento de compiladores — levou à criação de várias técnicas diferentes para a construção de um compilador, ou seja, passou a existir diferentes maneiras de se implementar um compilador. No entanto, a despeito dessa aparente complexidade, as tarefas básicas que qualquer compilador precisa realizar são essencialmente as mesmas.

A grande maioria dos compiladores de hoje fazem uso da técnica chamada: *tradução dirigida pela sintaxe*. Nessa técnica as regras de contrução do programa fonte são utilizadas para guiar todo o processo de compilação. Algumas das técnicas mais antigas utilizadas na contrução dos primeiros compiladores (da linguagem Fortran) podem ser obtidas em [Rosen, 1967].

1.2 Modelo de Compilação de Análise e Síntese

Ainda segundo [Rangel, 1999], existem duas tarefas triviais a serem executadas por um compilador nesse processo de tradução:

- *análise*, em que o texto de entrada (na linguagem fonte) é examinado, verificado e compreendido;
- *síntese*, ou *geração de código*, em que o texto de saída (na linguagem objeto) é gerado, de forma a corresponder ao texto de entrada.

Em [Aho et al., 1995], a *análise* é colocada como uma tarefa que divide o programa fonte nas partes constituintes e cria uma representação intermediária do mesmo. A *síntese* constrói o programa alvo desejado, a partir da representação intermediária.

Geralmente, pensamos nessas tarefas como fases que ocorram durante o processo de compilação. No entanto, não se faz totalmente necessário que a análise de todo o programa seja realizada antes que o primeiro trecho de código objeto seja gerado. Ou seja, estas duas fases podem ser intercaladas.

Pode-se ter como exemplo que o compilador pode analisar cada comando do programa de entrada e então gerar de imediato o código de saída correspondente ao respectivo comando. Ou ainda, o compilador pode esperar pelo fim da análise de cada bloco de comando — ou unidade de rotina (rotina, procedimentos, funções) — para então gerar o código correspondente ao bloco. Para aproveitar melhor a memória durante a execução, compiladores costumavam ser divididos em várias etapas, executados em sequência. Cada etapa constitui uma parte do processo de tradução, transformando assim o código fonte em alguma estrutura intermediária adequada, cada vez mais próxima do código objeto final.

É natural que a análise retorne como resultado uma representação do programa fonte que contenha informação necessária para a geração do programa objeto que o corresponda. Quase sempre, essa representação (conhecida como *representação intermediária*[Rangel, 1999]) tem como complemento tabelas que contêm informações adicionais sobre o programa fonte. Pode ter casos em que a representação intermediária toma a forma de um programa em

uma *linguagem intermediária*, deixando assim mais fácil a tradução para a linguagem objeto desejada.

Não importando a maneira pela qual se toma a representação intermediária, ela tem de conter necessariamente toda a informação para a geração do código objeto. Uma das características da representação intermediária é que as estruturas de dados implementadas devem dar garantia de acesso eficiente as informações.

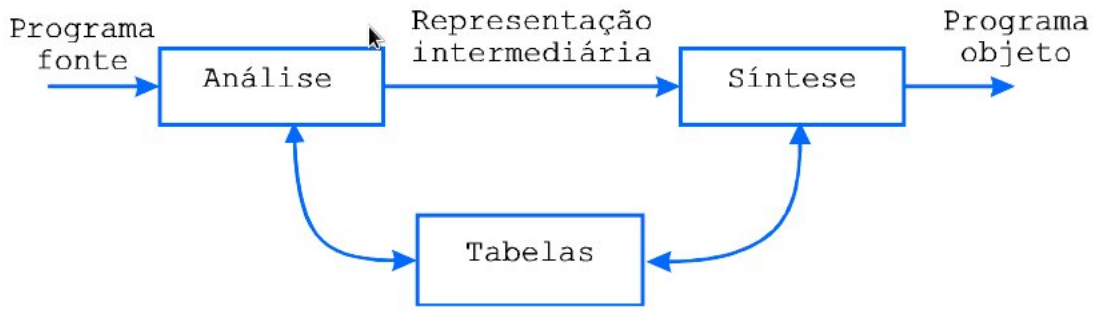

Figura 1.1: Estrutura do Compilador [Rangel, 1999]

Segundo [Rangel, 1999], uma das formas mais comuns de tabela utilizada nessa representação intermediária é a *tabela de símbolos*, em que se guarda para cada identificador(*símbolo*) usado no programa as informações correspondentes.

Há também um modelo possível em [Ullman et al., 1977], o qual se faz a separação total entre o *front-end*, encarregado da fase de análise, e o *back-end*, encarregado pela geração de código. Com isso tem-se que:

- *frontend* e *backend* se comunicam apenas da representação intermediária;
- o *frontend* depende exclusivamente da linguagem fonte
- o *backend* depende exclusivamente da linguagem objeto.

Essa idéia tem como objetivo simplificar a implementação de diferentes linguagens de programação para diferentes máquinas. Basta-se então escrever um *frontend* para cada linguagem e um *backend* para cada máquina. Ou seja, se deseja implementar x linguagens para y máquinas, precisa-se fazer x *frontends* e y *backends*. Esse esquema se torna mais fácil de aplicar quando há semelhança entre as máquinas e o mesmo acontece com as linguagens.

1.3 Análise

É normal associar a *sintaxe* a idéia de forma, em oposição a *semântica* que se associa a significado, conteúdo. Tem-se então que a sintaxe de uma linguagem de programação

deve descrever todos os aspectos relativos à forma de construção de programas corretos na linguagem, enquanto a semântica deve descrever o que acontece quando o programa é executado. Portanto, toda análise está relacionada com sintaxe, e a semântica deveria corresponder apenas à geração de código, que deve preservar o significado do programa fonte, contruindo um programa objeto com o mesmo significado[Rangel, 1999]

É necessário ressaltar uma diferença existente entre a teoria e a prática. Quando falamos em teoria, somente os programas corretos pertencem à linguagem, não havendo interesse nos programas incorretos. O fato é que um programa ou é da linguagem (está correto) ou não é da linguagem (está incorreto). No entanto, em se tratando de prática, no momento em que decide-se que um programa está incorreto, um bom compilador deve ser capaz de avisar sobre tal erro e de alguma forma, ajudar o usuário a corrigí-lo. Se faz necessário que o tratamento de erros inclua mensagens informativas e uma recuperação, para que a análise possa continuar e assim outros erros sejam sinalizados.

Em [Aho et al., 1995] vemos que a análise se constitui em 3 fases:

- *Análise Linear - Análise Léxica*, na qual um fluxo de caracteres constituindo um programa é lido da esquerda para a direita e agrupado em *tokens*, que são sequências de caracteres tendo um significado coletivo.

- *Análise Hierárquica - Análise Sintática*, na qual os caracteres ou *tokens* são agrupados hierarquicamente em coleções aninhadas com significado coletivo.

- *Análise Semântica*, na qual certas verificações são realizadas a fim de se assegurar que os componentes de um programa se combinam de forma significativa.

Sabe-se da possibilidade de total representação da sintaxe de uma linguagem de programação através de uma gramática sensível ao contexto [Wijngaarden, 1969]. No entanto, não há algoritmos práticos para tratar estas gramáticas, fazendo com que haja preferência em usar gramáticas livres de contexto. Sendo assim, fica claro que a separação entre análise sintática e análise semântica é dependente da implementação.

Sendo assim, a análise léxica tem como finalidade separar e identificar os elementos componentes do programa fonte, o qual estes geralmente, são especificados através de expressões regulares. A análise sintática deve reconhecer a estrutura global do programa, descrita através de gramáticas livre de contexto. A análise semântica se encarrega da verificação das regras restantes. Essas regras tratam quase sempre da verificação de que os objetos são usados no programa da maneira prevista em suas declarações, por exemplo verificando que não há erros de tipos [Rangel, 1999].

Não há ainda um modelo matemático que se adeque inteiramente na função de descrever o que deve ser verificado durante a análise semântica, ao contrário do que ocorre nas outras duas fases. No entanto, alguns mecanismos, como gramática de atributos, tem

sido utilizados com sucesso no processo de simplificação da construção de analisadores semânticos.

1.3.1 Tokens, Padrões e Lexemas

É necessário que antes de começar a falar de análise léxica tenha-se bem definido o significado de *tokens*, padrões e lexemas.

Tokens são símbolos terminais na gramática de uma linguagem. Falando em linguagens de programação, na maioria delas, as seguintes construções são tratadas como *tokens*: palavras-chave, operadores, identificadores, constantes, cadeias, literais(*strings*), e símbolos de pontuação como parênteses, vírgulas e pontos.

Padrão (*pattern* constitui-se em uma regra que define o conjunto de lexemas representam um *token* na linguagem. Normalmente tais regras são descritas na forma de expressões regulares.

Lexemas são sequências de caracteres descritas por um padrão de um *token*. Lexema é então o valor do *token*, aparecendo na maioria das vezes como um atributo a ser utilizado nas fases seguintes de compilação. Enquanto o analisador léxico faz a procura por *tokens*, temos que na geração de código os lexemas para produzir significado.

1.3.2 Expressões Regulares

Como já foi dito a análise léxica é responsável por gerar uma lista de tokens tendo como base o código fonte original. Dessa forma um meio de representarmos o que a linguagem aceita, ou não, pode ser neste primeiro passo de abstração representado através de Expressões Regulares.

Expressões Regulares são uma forma muito interessante de descrever padrões, especialmente aqueles que consistem em cadeias de caracteres. Através destas expressões podemos especificar que seqüências de caracteres são aceitas em um token, especificando caracteres opcionais e o número de repetições aceitos.

Segundo [Rigo, 2008] as Expressões Regulares são muito eficientes no que tange a representação ou especificação de tokens. Sendo assim neste primeiro momento o uso destas expressões são suficientemente boas para podermos representar os tokens que são aceitos pela linguagem em questão.

1.3.3 Análise Léxica

Segundo [Aho et al., 1995], o analisador léxico é a primeira fase de um compilador. Sua tarefa principal é a de ler os caracteres de entrada e produzir uma sequência de *tokens* que o *parser* utiliza para a análise sintática. Essa interação é comumente implementada

Tipo do *token*	**Valor do *token***
palavra reservada if	if
identificador x	x
operador maior	>
literal numérico	0
palavra reservada then	then
identificador	modx
operador de atribuição	:=
identificador	x
palavra reservada else	else
identificador	modx
operador de atribuição	:=
delimitador abre parêntese	(
operador menos unário	-
identificador	x
delimitador fecha parenteses	)

Tabela 1.1: Sequência de *tokens* identificados após a análise

fazendo com que o analisador léxico seja uma subrotina do *parser*. Ao receber do *parser* um comando 'obter o próximo *token*', o analisador léxico lê os caracteres de entrada até que possa identificar o próximo *token*.

Já em [Ullman et al., 1977], vemos que a análise léxica é responsável por separar e identificar os elementos componentes do código fonte. A análise léxica também elimina os elementos considerados 'decorativos' ou mesmo desnecessários para este processo, tais como espaços em branco, marcas de formatação de texto e comentários.

Em [Rangel, 1999] temos o seguinte exemplo em Pascal:

```
1  if x > 0 then          {x e' positivo}
2      modx := x
3  else                   {x e' negativo}
4      modx := (-x)
```

Após a análise léxica, a sequência de *tokens* identificadas é:

Normalmente os tipos dos *tokens*(na primeira coluna) são representados por valores de um tipo de enumeração ou por códigos numéricos apropriados.

O que se vê na grande maioria das vezes é que a implementação de um analisador léxico é baseada em um autômato finito capaz de reconhecer as diversas construções.

1.3.4 Análise Sintática

As linguagens de programação possuem regras que descrevem a estrutura sintática dos programas bem-formados. Dessa maneira, a partir da sequência de *tokens* gerada pelo analisador léxico, o analisador sintático realiza o seu trabalho, verificando se esta sequência pode ser gerada pela gramática da linguagem-fonte.

A análise sintática envolve o agrupamento dos *tokens* do programa fonte em frases gramaticais, que são usadas pelo compilador, a fim de sintetizar a saída. Usualmente, as frases gramaticais do programa fonte são representadas por uma árvore gramatical [Aho et al., 1995].

Já em [Rangel, 1999] tem-se que a análise sintática deve reconhecer a estrutura global do programa, por exemplo, verificando que programas, comandos, declarações, expressões, etc têm as regras de composição respeitadas.

Tem-se o exemplo:

```
1  se x > 0 entao
2       modx = x
3  senao
4       modx = -x
5  fimsenao
```

Caberia a análise sintática reconhecer a estrutura deste trecho, reconhecendo de que se trata de um *¡comando¿* , que no caso é um *¡comando-se¿*, composto pela palavra reserva *se*, seguido de uma *¡expressão*, seguida também de uma palavra reservada *entao*, e assim por diante.

Ainda segundo [Rangel, 1999], quase universalmente, a sintaxe das linguagens de programação é descrita por gramáticas livres de contexto, em uma notação chamada BNF (*Forma de Backus-Naur* ou ainda *Forma Normal de Backus*, ou em alguma variante ou extensão dessa notação. Essa notação foi introduzida por volta de 1960, para a descrição da linguagem Algol [Naur, 1963].

Existem três tipos gerais de analisadores sintáticos. Os métodos universais de análise sintática, tais como o algoritmo de *Cocke-Younger-Casami* e o de *Earley*, podem tratar qualquer gramática. Esses métodos, entretanto, são muito ineficientes para se usar num compilador de produção. Os métodos mais comumentes usados nos compiladores são classificados como *top-down* ou *bottom-up*. Como indicado por seus nomes, os analisadores sintáticos *top-down* constroem árvores do topo (raiz) para o fundo (folhas), enquanto que os *bottom-up* começam pelas folhas e trabalham árvore acima até a raiz. Em ambos os casos, a entrada é varrida da esquerda para a direita, um símbolo de cada vez [Aho et al., 1995].

Os métodos de análise sintática mais eficientes, tanto *top-down* quanto *bottom-up*,

trabalham somente em determinadas subclasses de gramáticas, mas várias dessas subclasses, como as das grámaticas LL e LR, são suficientemente expressivas para descrever a maioria das construções sintáticas das linguagens de programação. Os analisadores implementados manualmente trabalham frequentemente com gramáticas LL. Os da classe mais ampla das gramáticas LR são usualmente contruídos através de ferramentas automatizadas [Aho et al., 1995].

Tratamento dos Erros de Sintaxe

Seria muito mais fácil e simples se um compilador tivesse que processar apenas programas corretos. No entanto, frequentemente os programadores escrevem programas errados que necessitam ser corrigidos e, um bom compilador deve ajudar a encontrá-los. Porém, [Aho et al., 1995] cita que a maioria das linguagens de programação não descreve como um compilador deveria responder aos erros, deixando tal tarefa para o projetista do compilador. O planejamento do tratamento de erros exatamente desde o início poderia tanto simplificar a estrutura de um compilador quanto melhorar sua resposta aos erros.

Os programas podem conter erros em níveis diferentes. Por exemplo:

- Léxicos, tais como errar a grafia de um identificador ou palavra-chave;
- Sintáticos, tais como uma expressão aritmética com parênteses não fechados;
- Lógicos, tais como uma entrada em um *looping* infinito.

Em [Aho et al., 1995] temos que boa parte da detecção e recuperação de erros num compilador gira em torno da fase de análise sintática. Isto porque os erros ou são sintáticos por natureza ou são expostos quando o fluxo de *tokens* proveniente do analisador léxico desobedece às regras gramaticas que definem a linguagem de programação. Outra razão está na precisão dos modernos métodos de análise sintática, sendo estes capazes de detectar muito eficientemente a presença de erros sintáticos num programa.

Pode-se concluir que o tratamento de erros em um analisador sintático tem metas simples de serem estabelecidas:

- Relatar de maneira clara e objetiva qualquer presença de erros
- Recuperar-se o mais rápido possível de algum erro para que assim, possa detectar erros subsequentes
- E por fim, não deve atrasar de maneira significativa o processamento de programas corretos.

Realizar efetivamente tais metas não constitui em uma tarefa fácil.

Felizmente, o que se vê é que os erros frequentes são simples e na maioria das vezes basta um método de tratamento de erros relativamente direto. Em alguns casos, porém, pode acontecer de um erro ter ocorrido antes mesmo de que sua presença fosse detectada e identificar precisamente a sua natureza pode ser muito difícil. Não é raro que em alguns casos difíceis, o tratador de erros tenha que adivinhar a idéia do programador quando o programa foi escrito.

Vários métodos de análise sintática, tais como os métodos LL e LR, detectam os erros tão cedo quanto possível. Mais precisamente, possuem a *propriedade do prefixo viável*, significando que detectam que um erro ocorreu tão logo tenham examinado um prefixo da entrada que não seja o de qualquer cadeia da linguagem.

Com o intuito de se conhecer os tipos de erros que ocorrem na prática, vamos examinar os erros que [Ripley and Druseikis, 1978] encontraram em uma amostra de programa Pascal de estudantes.

[Ripley and Druseikis, 1978] descobriram que os erros não ocorrem com tanta frequência. 60% dos programas compilados estavam semântica e sintaticamente corretos. Mesmo quando os erros ocorriam de fato, eram um tanto dispersos. 80% dos enunciados contendo erros possuíam apenas um, 13% dois. Finalmente, a maioria constituia de erros triviais. 90% eram erros em um único *token*.

Ainda segundo [Ripley and Druseikis, 1978], muitos dos erros poderiam ser classificados simplificadamente. 60% eram erros de pontuação, 20% de operadores e operandos, 15% de palavras-chave e os 5% restantes de outros tipos. O grosso dos erros de pontuação girava em torno do uso incorreto do ponto e vírgula.

Gramáticas Livres de Contexto

Tradicionalmente, gramáticas livres de contexto têm sido utilizadas para realizar a análise sintática de linguagens de programação. Nem sempre é possível representar neste tipo de gramática restrições necessárias a algumas linguagens – por exemplo, exigir que todas as variáveis estejam declaradas antes de seu uso ou verificar se os tipos envolvidos em uma expressão são compatíveis. Entretanto, há mecanismos que podem ser incorporados às ações durante a análise – por exemplo, interações com tabelas de símbolos – que permitem complementar a funcionalidade da análise sintática.

A principal propriedade que distingüe uma gramática livre de contexto de uma gramática regular é a auto-incorporação. Uma gramática livre de contexto que não contenha auto-incorporação pode ser convertida em uma gramática regular.

Segundo [Wirth, 1996], o termo livre de contexto deve-se a Chomsky e indica que a substituição do símbolo à esquerda da pela seqüência derivada da direita é sempre

permitida, independente do contexto em que o símbolo foi inserido. Esta restrição de liberdade de contexto é aceitável e desejável para linguagens de programação.

Várias linguagens de programação apresentam estruturas que são por sua natureza recursivas e podem ser definidas por gramáticas livres de contexto. Podem-se dar como exemplo uma declaração condicional definida por uma regra como:

```
1  se E entao
2       S1
3  senao
4       S2
5  fimsenao
```

Tal forma de declaração condicional não pode ser escrita por uma expressão regular. No entanto, utilizando uma variável sintática *cmd* com função de atribuir a classe da declaração e *expr* para denotar a classe de expressões, pode-se então expressar a declaração condicional como:

```
1  stmt -> se expr então stmt senão stmt fimsenao
```

1.4 Síntese ou Geração de Código

Em um compilador a fase final é a geração do código alvo, que consiste em código de montagem ou código de máquina relocável. Nesta fase acontece a tradução para a linguagem de máquina da máquina alvo ou para a linguagem destino. Algumas das tarefas do gerador de código são:

- Gerenciamento de memória;
- Seleção de instruções;
- Alocação de registradores.

1.4.1 Geração de Código Intermediário

Segundo [Aho et al., 1995], no modelo de análise e síntese de um compilador, os módulos da vanguarda traduzem o programa fonte numa representação intermediária, a partir da qual os módulos da retaguarda geram o código alvo. Na medida do possível, os detalhes da linguagem alvo são confinados ao máximo nos módulos da retaguarda. Apesar de se poder traduzir o programa fonte diretamente na linguagem alvo, alguns dos benefícios em se usar uma forma intermediária independente da máquina são:

- O redirecionamento é facilitado: Um compilador para uma máquina diferente pode ser criado atrelando-se à vanguarda existente uma retaguarda para a nova máquina.

- Um otimizador de código independente da máquina pode ser aplicado à representação intermediária.

Na figura a seguir é possível ver a posição do gerador de código intermediário no processo de compilação:

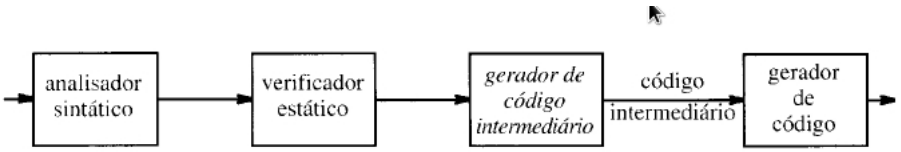

Figura 1.2: Posição do gerador de código intermediário [Aho et al., 1995]

1.4.2 Geração de Código

A fase final de um compilador é o gerador de código. Esse recebe como entrada a representação intermediária do programa fonte e produz como saída um programa alvo equivalente, como indicado na figura 1.2.

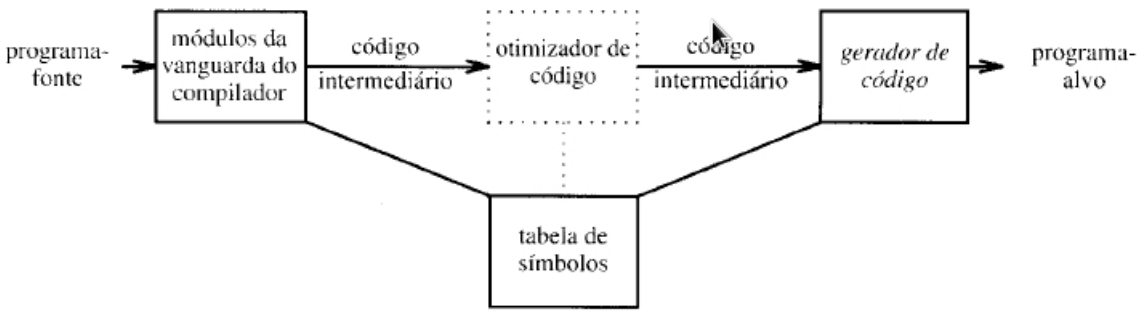

Figura 1.3: Posição do gerador de código intermediário [Aho et al., 1995]

As exigências tradicionalmente impostas a um gerador de código são severas. O código de saída precisa ser correto e de alta qualidade, significando que o mesmo deve tornar efetivo o uso dos recursos da máquina alvo. Sobretudo, o próprio gerador de código deve rodar eficientemente [Aho et al., 1995].

Matematicamente, o problema de se gerar um código ótimo não pode ser solucionado. Na prática, deve-se contentar com técnicas heurísticas que geram um código bom, mas não necessariamente ótimo. A escolha dos métodos heurísticos é importante, na medida em que um algoritmo de geração de código cuidadosamente projetado pode produzir um código que seja várias vezes mais rápido do que aquele que produzido por um algoritmo concebido às pressas [Aho et al., 1995].

Cabe ao projetista do gerador de código decidir como implementar a geração de código de maneira a fazer bom uso dos recursos disponíveis na máquina. Cabe também ao projetista decidir se a geração do código deve ser feita com cuidado, gerando diretamente código de qualidade aceitável, ou se é preferível usar um esquema mais simples de geração de código, seguido por uma 'otimização' do código depois de gerado [Rangel, 1999].

Entrada para o Gerador de Código

O gerador de código recebe como entrada a representação intermediária do programa fonte, que foi produzida anteriormente pela vanguarda do compilador, em conjunto com informações presentes na tabela de símbolos, que tem como finalidade determinar os endereços, em tempo de execução, dos objetos de dados, os quais são denotados pelos nomes na representação intermediária.

Assumi-se que a geração prévia de código, a partir da vanguarda do compilador, analisou léxica e sintaticamente o programa fonte, bem como o traduziu numa forma razoavelmente detalhada de representação intermediária, de forma que os nomes que figuram na linguagem intermediária possam ser representados por quantidades que a máquina alvo possa diretamente manipular. Também assumimos que a necessária verificação de tipos já teve lugar, de forma que os operadores de conversão de tipo já foram inseridos onde quer que fossem necessários e que os erros semânticos óbvios já foram detectados. A fase de geração de código pode, por conseguinte, prosseguir na suposição de que sua entrada está livre de erros. Em alguns compiladores, esse tipo de verificação semântica é feito junto com a geração de código [Aho et al., 1995].

Programas Alvo

A saída do gerador de código é o programa alvo. Como o código intermediário, essa saída pode assumir uma variedade de formas: linguagem absoluta de máquina, linguagem relocável de máquina ou linguagem de montagem.

Como saída, a produção de um programa em linguagem absoluta de máquina possui a vantagem do mesmo poder ser carregado numa localização fixa de memória e executado imediatamente. Um pequeno programa pode ser compilado e imediatamente executado [Aho et al., 1995].

A produção de um programa em linguagem de montagem como saída torna o processo de geração de código um tanto mais fácil. Podemos gerar instruções simbólicas e usar as facilidades de processamento de macros do montador para auxiliar a geração de código. O preço pago está no passo de montagem após a geração de código. Como a produção do código de montagem não duplica toda a tarefa do compilador, essa escolha é outra alternativa razoável, especialmente para uma máquina com uma memória pequena, onde

o compilador precisa realizar diversas passagens [Aho et al., 1995].

1.5 Conclusão

A história dos compiladores se confunde com a história da computação, visto que esta é a maneira de se comunicar com os computadores de maneira fácil e eficiente. A medida que a tecnologia vai avançando os computadores também o vão, bem como as técnicas e ferramentas utilizadas na construção do mesmo.

É fato então que os compiladores são ferramentas muito importantes para o mundo da computação e, seu estudo se faz necessário, visto a grande necessidade que temos hoje - devido ao aumento dos inúmeros dispositivos tecnológicos e das linguagens de programação - de comunicação com os computadores.

Capítulo 2

Portugol

2.1 Introdução

O Portugol como linguagem não é tão bem definida, ficando muito mais sujeita a uma linguagem de aprendizado, voltada para o ensino e não para projetos de grande porte. Algumas vezes é somente uma tradução simplificada de Pascal para o português.

O Portugol aqui definido é baseado em [Farrer, 1999]. O Livro utilizado para o estudo de algoritmos e de introdução a programação é perfeito para este trabalho. O compilador desenvolvido, bem como o sistema Web, visa estudar como o processo de compilação online se dará, sendo necessário uma linguagem simples que não tome o foco do objetivo principal.

Nas seções subsequentes é possível ver como o portugol foi proposto para este compilador. Suas características, sua sintaxe e a estrutura da linguagem.

2.2 Estrutura da Linguagem

2.2.1 Comentários

Mais de uma forma de comentar o código será aceito, estimulando o seu uso em todo o código. O // usado em linguagens como C e Java e também a # (cerquilha), utilizada em linguagens como o Ruby são aceitos como comentários de uma só linha. Os (colchetes) são a única forma aceita de comentários para várias linhas ou comentários de bloco.

2.2.2 Tipos básicos

Os tipos básicos são simples e restritos, pois visam o ensino da lógica de programação. Isso promove também a independência de linguagem e de máquina.

Numérico

O tipo numérico resume todos os tipos para o cálculo aritmético. Não existe distinção entre números inteiros ou números reais (ponto flutuante). Caso haja a necessidade de que algum algoritmo utilize uma propriedade específica de algum destes tipos, deverá esta ser obtida através de funções. A vírgula é o símbolo utilizado para a separação da parte decimal. Não se escrevem os separadores de milhar.

Literal

Este tipo é responsável por armazenar sequências de letras e símbolos — em muitas linguagens é o tipo *String*.

Lógico

Responsável pelo armazenamento das constantes verdadeiro e falso. Este tipo é muito importante em projetos de linguagens de programação, visto que ele é bastante utilizado, sendo o tipo de retorno de várias operações.

Vetores e Matrizes

É permitida a criação de vetores e matrizes de quaisquer tipos básicos. As operações realizadas com matrizes e vetores deverão ser realizadas de forma condizente com a linguagem Portugol. As operações básicas deste tipo seguem as mesmas permitidas em seus tipos básicos. A palavra-chave *matriz* é reservada para este tipo. O número inicial dos índices é sempre 1.

Exemplo:

```
1 a matriz numérico[30]  // Cria uma matriz 30x1
2 b matriz lógico[3][3]  // Cria uma matriz 3x3
```

Registros

É sempre definido pela palavra reservada **registro**. Representa um bloco composto por declarações de variáveis. Se encerra com a palavra **fimregistro**.

Exemplo:

```
1  registro
2    num1, num2 numérico
3    estaPresente lógico
4    endereco literal
5  fimregistro
```

2.2.3 Variáveis e Identificadores

As variáveis são parte importante de uma linguagem. Qualquer identificador, sendo este válido, tem como primeiro símbolo uma letra. Números e _ (sublinha) são aceitos após o primeiro caractere. O tamanho máximo válido é definido em 32 caracteres.

É necessário lembrar que alguns símbolos, normalmente ignorados ou proibidos em outras linguagens, são aceitos em Portugol. Símbolos estes presentes na língua portuguesa, como por exemplo, os acentos: á, ê, í, à, õ, ..., ç.

Vale ressaltar da diferença existente entre variáveis declaradas com acento e variáveis sem acento. Ou seja, a variável 'pé' é diferente da variável 'pe'.

2.2.4 Declaração de variáveis

Não existe um local específico para declaração de variáveis, mas fica a recomendação de que esta seja feita no início de uma seção, seja o programa principal ou um procedimento/função. A palavra-chave **declare** será utilizada para declarar variáveis em Portugol. É possível declarar variáveis de mesmo tipo em uma mesma linha, nunca variáveis de tipos diferentes.

Exemplo:

```
1  declare num1, num2, num3 numérico
2  declare palavra literal
3  declare a, b, c lógico
4  declare r registro
5          dia, mes numérico
6          nome literal
7          estaPresente lógico
8      fimregistro
```

Exemplo de declaração inválida:

```
1  declare num1 numérico, palavra literal
```

A visibilidade de uma palavra segue o mesmo padrão de outras conhecidas linguagens, ou seja, é dependente do escopo a qual foi definida.

2.2.5 Atribuição

A atribuição de valores a uma variável será denotada pelo símbolo ":=". Será permitido apenas a atribuição de um valor por vez. Uma atribuição será válida quando possuir um identificador válido do lado esquerdo e uma expressão válida e, do mesmo tipo, do lado direito.

Exemplo:

```
1  declare num1, num2 numérico
2  declare palavra literal
3
4  num1 = 3
5  num2 = 1,5
6  palavra = "Olá Mundo"
```

2.2.6 Operadores

Operadores Aritméticos

Os operadores são:

Tabela 2.1 - Operadores Aritméticos

Operação	Símbolo
Adição	+
Subtração	-
Multiplicação	*
Divisão	/

Operadores Lógicos

Os Operadores são:

Tabela 2.2 - Operadores Lógicos

Operação	Símbolo
inversão - não	não
junção	e
disjunção	ou

Operadores Relacionais

Os Operadores são:

Tabela 2.3 - Operadores Relacionais

Operação	Símbolo
Igualdade	=
Maior	>
Menor	<
Maior ou Igual	≥
Menor ou Igual	≤
Diferença	!= ou <>

Operadores Relacionais

Os Operadores são:

Tabela 2.4 - Operadores Literais

Operação	Símbolo
Concatenação	+
Igualdade	=
Maior	>
Menor	<
Maior ou Igual	≥
Menor ou Igual	≤
Diferença	!= ou <>

2.2.7 Estrutura Sequencial

O programa tem início com a palavra **programa** e tem fim com a palavra **fimprograma**. Caso queira importar outros módulos, estes devem ser feitos através de comando, após a declaração do programa.

Em Portugol não se faz necessária a utilização de algum símbolo especial ou visual para marcar o fim de linha. Basta que o "Retorno" — mais conhecido pela tecla "ENTER" — seja pressionado. Este símbolo é definido na posição 13 da tabela ASCII.

Qualquer que seja o símbolo ou comando declarado fora do escopo do programa, será considerado erro. Funções e procedimentos devem ser declarados externamente e então, importados pelo programa que necessite utilizá-los. Isto fará com que se estimule a modularização de código, bem como a criação de arquivos de menor tamanho e mais fáceis de entender.

2.2.8 Estruturas Condicionais

São definidas duas: **se** e o **caso**.

O **se** é utilizado quando se quer avaliar uma condição ou expressão lógica e retorna obrigatoriamente um valor lógico. Conforme o retorno a execução passa para o bloco **então** — caso seja verdadeiro — ou para o bloco **senão** — caso seja falso.

É definido também em Portugol o **fimentão** e o **senão**, responsáveis pelo fechamento dos blocos **então** e **senão** respectivamentes.

Exemplo:

```
1  se num1 > num2
2       então num1 = 1
3       fimentão
```

```
4       senão num2 = 2
5       fimsenão
6   fimse
```

A estrutura **caso** permite avaliar vários valores e admite váriavel dos tipos lógicos, numérico e literal.

```
1   caso tamanho
2       10: i := 10
3       11: i := 11
4       12: i := 12
5   fimcaso
```

2.2.9 Estruturas de Repetição

Três estruturas são definidas em Portugol: **faça**, **enquanto**, **repita**.

O **faça** é similar ao *for* de outras linguagens. Sua construção é demonstrada abaixo:

```
1   faça i de 1 até 10
2       <instruções>
3   fimfaça
```

O bloco acima irá executar as instruções dez vezes, incrementando o valor do "i" de 1 até 10. É recomendável como boas práticas de programação que a váriavel de controle — no bloco acima é o "i" — não tenha seu valor alterado pelas instruções internas. O valor do "i" é incrementado em 1 sempre quando se chega ao "fimfaça". O valor do "i" após o "fimfaça" é igual ao valor máximo estipulado no início, no caso 10.

O **enquanto** substitui o *while* de outras linguagens de programação. Exemplo:

```
1   enquanto num1 > num2
2       <instruções>
3   fimenquanto
```

A estrutura de repetição **enquanto** executa o bloco de código até que a condição seja falsa. Pode acontecer das instruções nem serem executadas, caso a condição seja falsa logo de início.

A estrutura **repita** tem como característica o fato de ser básica. A verificação e a interrupção do bloco de código é livre, sendo especificada pelo comando **interrompa**. Veja:

```
1   repita
2       <comandos>
3       interrompa
4   fimrepita
```

Após a leitura do interrompa, a execução do programa se dará para o trecho de código abaixo do "fimrepita".

Como pode ser observado, a maioria das estruturas definidas são fechadas sempre utilizando o "fim*estrutura*". A não utilização de tal comando acarretará em um erro de código, podendo ser este léxico ou sintático.

2.2.10 Funções

As funções aqui definidas seguem as mesmas regras de um programa, não podendo apenas importar módulos.

A declaração de uma função pode ser vista abaixo:

```
1  funcao soma(a numerico, b numerico)
2      <comandos>
3  fimfuncao
```

Assim como ocorre em outras linguagens de programação, as variáveis declaradas em uma determinada função são internas a ela, possuindo assim escopo local.

Decidi por não utilizar uma estrutura do tipo procedimento - estrutura esta existente em outras linguagens - pelo fato de simplificar o uso da linguagem, visto que ela se designa ao ensino de programação. O mesmo se faz irrelevante neste trabalho, já que a principal diferença entre uma função e um procedimento é que a função retorna um resultado.

2.3 Conclusão

A linguagem Portugol é por sua própria natureza simples. Na maioria das vezes em que é utilizada a finalidade é sempre o aprendizado de programação de computadores. O Portugol utilizado para a construção deste trabalho também tem a simplicidade em sua essência, no entanto, oferece as condições necessárias para uma introdução a programação, além de também atender ao propósito deste trabalho de final de curso.

É importante deixar claro que tomando como base este trabalho e a linguagem Portugol em si, é perfeitamente possível a criação de uma linguagem de programação mais poderosa e robusta que contenha uma estrutura muito mais complexa. Ou então é também possível, a criação de uma DSL (Domain Specific Language), mas este seria um assunto para um outro projeto.

Capítulo 3

Estado da Arte

3.1 Introdução

A contrução de compiladores, bem como o estudo do mesmo é algo antigo que acontece desde os primórdios da computação, visto que desde sempre, fez-se necessário a comunicação com as máquinas e, a evolução dos compiladores e também das linguagens de programação se baseou no estudo de técnicas que visavam melhorar tal comunicação.

Posto de forma simples, um compilador é um programa fonte que lê um programa escrito numa linguagem - a linguagem - fonte e o traduz em um programa equivalente em uma outra linguagem - a linguagem alvo (como visto na figura abaixo) [Aho et al., 1995].

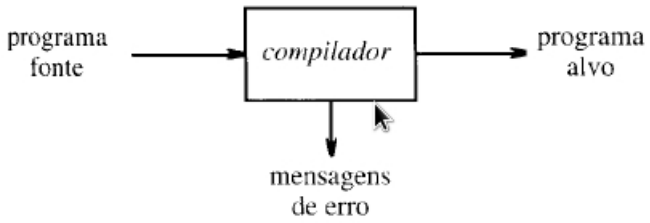

Figura 3.1: Um Compilador [Aho et al., 1995].

3.2 Compiladores

Conforme Knuth e Trabb [KP80], o termo compilador não era ainda utilizado nessa época. Na verdade falava-se sobre programação automática. No início da programação em linguagem de máquina foram desenvolvidas subrotinas de uso comum para entrada e saída, para aritmética de ponto flutuante e funções transcendentais. Junto com a idéia de um endereçamento realocável - pois tais subrotinas seriam usadas em diferentes partes de um programa - foram criadas rotinas de montagem para facilitar a tarefa de uso das subrotinas e de endereçamento relativo, idéia desenvolvida por Maurice V. Wilkes. Para

isso foi inventada uma pseudo linguagem de máquina. Uma rotina interpretativa iria processar essas instruções, emulando um computador hipotético. Esse é o sentido do termo 'compilador' até aqui usado [Filho, 2007].

Nos primórdios dos computadores, programar era uma tarefa extremamente complicada e, de certa forma, extenuante. Aos programadores era exigido um conhecimento detalhado das instruções, registos e outros aspectos ligados com a unidade de processamento central (CPU) do computador onde era escrito o código. Os programas consistiam numa série de instruções numéricas, denominadas por código binário. Posteriormente, desenvolveram-se algumas mnemónicas que resultaram no designado assembly [Henriques, 2000].

O AUTOCODE foi o primeiro 'compilador' real, que tomava uma declaração algébrica e a traduzia em linguagem de máquina. Seu desconhecido autor, Alick E. Glennie, das forças armadas da Inglaterra, declarava em Cambridge, em 1953, sua motivação para elaborá-lo: 'A dificuldade da programação tornou-se a principal dificuldade para o uso das máquinas. Aiken expressou sua opinião dizendo que a solução para esta dificuldade deveria ser buscada pela construção de uma máquina especial para codificar(...) Para tornar isso fácil deve-se elaborar um código compreensível. Tal coisa somente pode ser feita melhorando-se a notação da programação'.

John Backus discute essa distinção que Knuth faz, citando J. Halcomb Laning, Jr. e Niel Zierler como os inventores do primeiro 'compilador' algébrico, para o computador Whirlwind. Como esta, são muitas as discussões ainda hoje sobre quem foi o pioneiro no assunto. De qualquer maneira esses primeiros sistemas denominados genericamente de programação automática (acima citada) eram muito lentos e não fizeram muito sucesso, embora tivessem sido fundamentais para preparar a base do desenvolvimento que se seguiu [Filho, 2007].

Este veio com o A-0, agora sim o primeiro compilador propriamente dito, desenvolvido por Grace Murray Hopper e equipe, aprimorado para A-1 e A-2 subseqüentemente. O próximo passo seria o A-3, desenvolvido em 1955, produzido ao mesmo tempo com o tradutor algébrico AT-3, mais tarde chamado MATH-MATIC [Filho, 2007].

Em 1952 a IBM construía o computador 701 e em 1953 foi montada uma equipe liderada por John Backus para desenvolver um código automático que facilitasse a programação. O resultado foi o Speedcoding. Backus tornou-se uma das principais figuras na história da evolução das linguagens de programação, tendo um papel fundamental no desenvolvimento dos grandes compiladores que viriam a partir do ano de 1955 como o FORTRAN e o ALGOL, além do estabelecimento da moderna notação formal para a descrição sintática de linguagens de programação, denominada BNF, Backus Normal Form [Filho, 2007].

No período entre 1954-1957 uma equipa de 13 programadores liderados por John

Backus desenvolveu uma das primeiras linguagens de alto nível para o computador IBM 704, o FORTRAN (FORmula TRANslation). O objetivo deste projecto era produzir uma linguagem de fácil interpretação, mas ao mesmo tempo, com uma eficiência idêntica à linguagem assembly [Henriques, 2000].

A linguagem Fortran foi ao mesmo tempo revolucionária e inovadora. Os programadores libertaram-se assim da tarefa extenuante de usar a linguagem assembler e passaram a ter oportunidade de se concentrar mais na resolução do problema. Mas, talvez mais importante, foi o fato dos computadores passarem a ficar mais acessíveis a qualquer pessoa com vontade de despender um esforço mínimo para conhecer a linguagem Fortran. A partir dessa altura, já não era preciso ser um especialista em computadores para escrever programas para computador [Henriques, 2000].

3.3 Trabalhos Relacionados

Existem alguns trabalhos semelhantes a este. Uns mais, outro menos, mas todos com a proposta de colocar um compilador disponível de maneira online, por meio apenas de um navegador. A questão a ser discutida sempre é sobre a proposta do trabalho, ou seja, qual o motivo de tê-lo, é isto que muda a perspectiva do trabalho e maneira de implementá-lo. Com certeza o projeto que mais se parece com este, o qual tem uma ótima proposta e foi muito bem feito é o *CodeSchool*. O ponto que é o difere é a proposta de venda de informação, através de cursos online, sendo o compilador uma parte de um todo.

O grande diferencial do trabalho aqui proposto é que ainda não existe um compilador online para a linguagem de programação Portugol, sendo este o primeiro que se tem notícia.

Vejamos agora nas seções abaixo alguns trabalhos relacionados.

3.3.1 Codepad

O [codepad, 2011] é um compilador e interpretador online, além de ser uma simples ferramenta de colaboração. Tem a função de *pastebin** que executa código para o usuário. O usuário cola o código e o [codepad, 2011] o compila, dando ao usuário um URL que você pode usar para compartilhar o resultado com outros. O usuário pode também, simplesmente, colocar algum código e compilar, quando este não estiver com um compilador instalado em seu computador. O [codepad, 2011] funciona em alguns celulares e *tablets*.

O [codepad, 2011] foi escrito(e ainda é mantido) por Steven Hazel. O site [codepad, 2011] foi escrito em *Python*, usando *Pylons* e *SQLAlchemy*.

O [codepad, 2011] possui suporte a várias linguagens. São elas(com seus respectivos compiladores):

- C: gcc 4.1.2
- C++: g++ 4.1.2
- D: Digital Mars D Compiler v1.026
- Haskell: Hugs, Setembro 2006
- Lua: Lua 5.1.3
- OCaml: Objective Caml version 3.10.1
- PHP: PHP 5.2.5
- Perl: Perl v5.8.0
- Python Python 2.5.1
- Ruby: Ruby 1.8.6
- Scheme: MzScheme v372 [cgc]
- Tcl: tclsh 8.4.16

Na figura abaixo podemos ver a tela inicial do [codepad, 2011]:

Figura 3.2: Tela do compilador online CodePad. codepad.org

Nessa tela o usuário seleciona a linguagem a qual ele irá utilizar e então coloca o código no campo em branco. Após isso o usuário clica em 'submit' e tem a resposta do compilador.

3.3.2 Ideone

Este é um dos mais completos e robustos compiladores online. Possui uma grande variedade de linguagens suportadas (até a data deste trabalho, mais de 40), e inclusive, marcação de sintaxe(*sintax highlight*), funcionando como um bom editor de programação. Ele também pode ser utilizado como um *pastebin*, mas é muito mais do que isso, funcionando muitas vezes até como um debugger.

Suas características principais são:

- Compartilhamento de código;
- Compilar código diretamente do navegador em mais de 40 linguagens de programação;
- Possui uma API para que o usuário possa construir seu próprio compilador online;
- Oferece a possibilidade de gerenciar programas já feitos através de um cadastro anterior.

Abaixo a tela do Ideone:

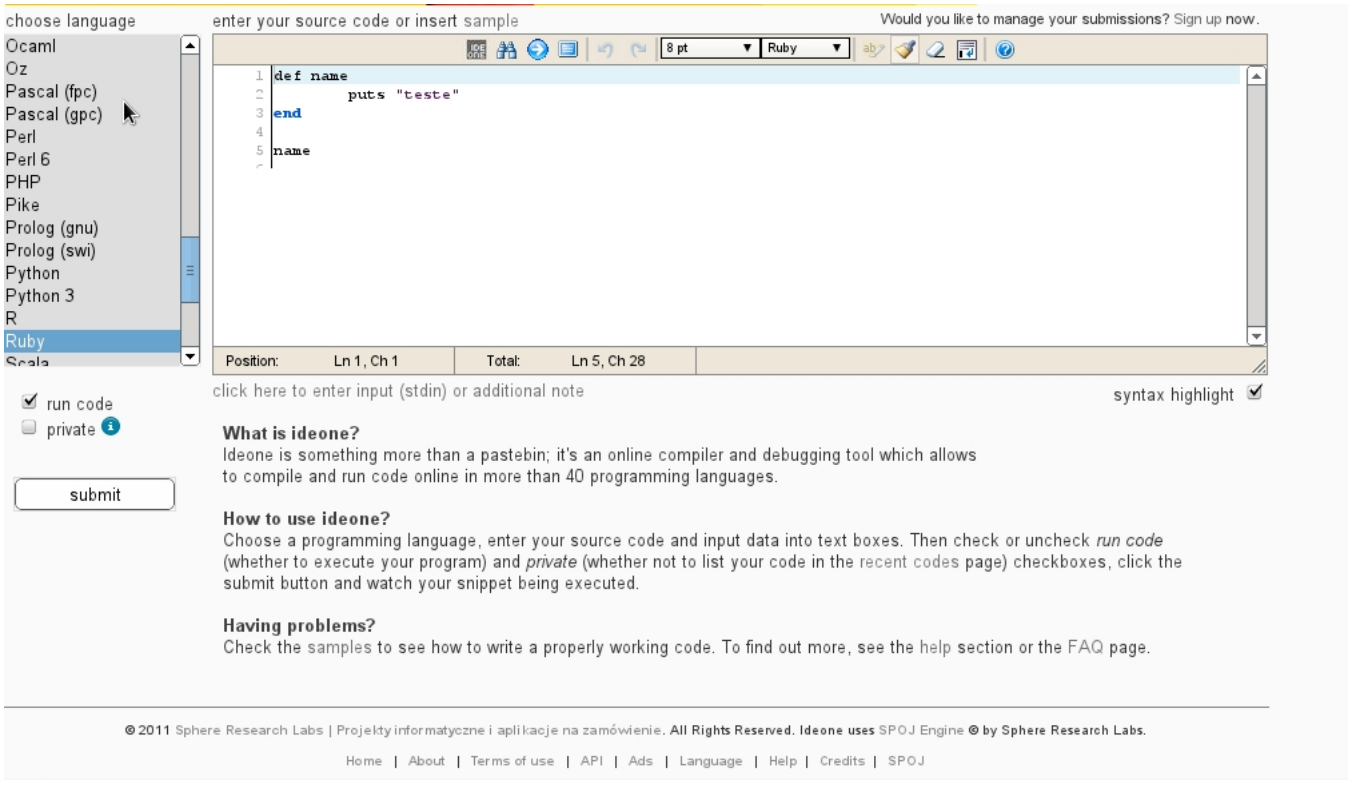

Figura 3.3: Tela do compilador online Ideone.

Essa tela é similar a tela anterior. O usuário deve selecionar a linguagem a ser utilizada e então clicar em 'submit' para que o compilador retorne com a resposta.

3.3.3 CodeSchool

Este é mais do que apenas um compilador online, o Code School tem um objetivo diferente. Ele não é apenas um campo de texto no qual o usuário digita seu código e tudo é compilado, ele é mais do que isso. O Code School é uma plataforma de ensino, no qual o usuário pode ler textos, ver vídeos e programar na prática determinada linguagem, facilitando seu aprendizado. Abaixo um trecho dos criadores sobre o Code School:

'A maioria das pessoas não aprendem programação e *design* para a web lendo livros. Aprendizados verdadeiros acontecem quando você começa a experimentar o código no navegador e usa conceitos de design em um website. Na nossa opinião, a melhor maneira de aprender é fazendo. Code School abre as portas para uma nova maneira de aprendizado, combinando vídeos, codificação no navegador e jogos para fazer o aprendizado de uma nova tecnologia divertido'.

Code School foi criado pelos cientistas da *Envy Labs*, uma equipe de desenvolvedores de software web. Abaixo uma imagem da tela do Code School:

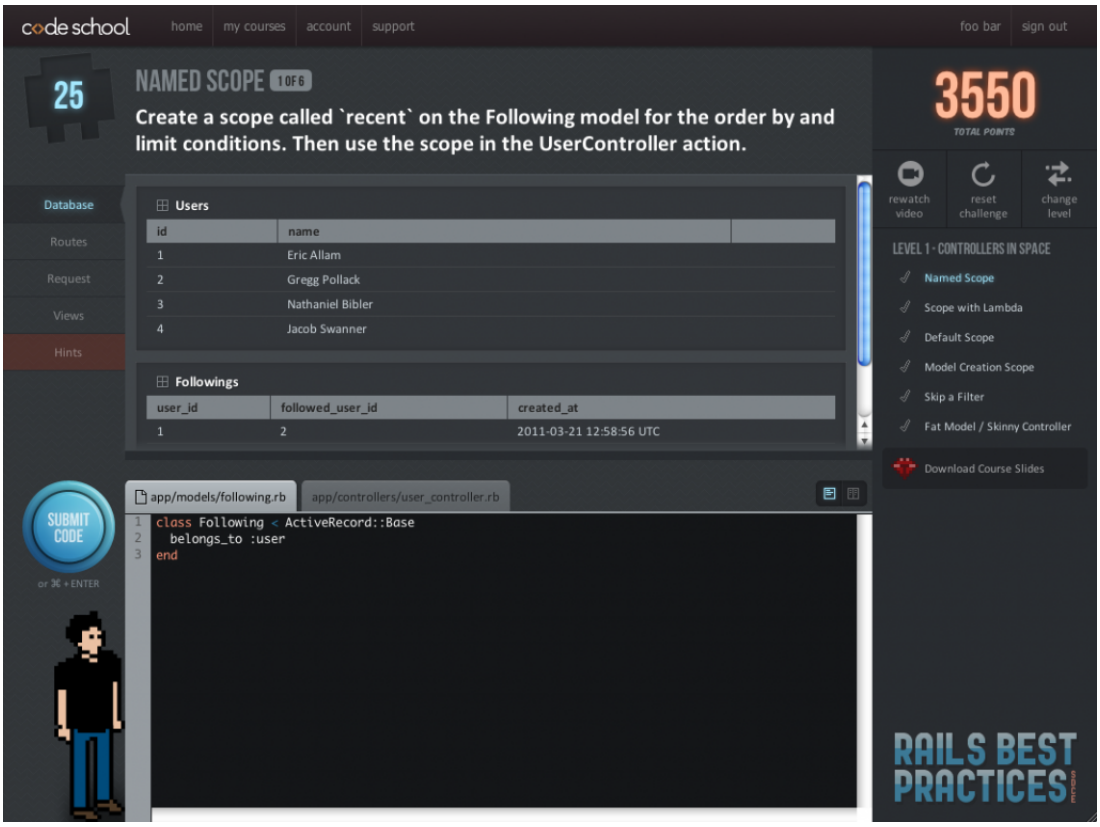

Figura 3.4: Tela de exercícios do compilador CodeSchool.

Nessa página o usuário é capaz de digitar o código e receber a resposta do compilador.

3.3.4 Rails for Zombies

Este é o mais alternativo de todos acima apresentado. Sua intenção é a de ensinar o *Ruby on Rails* por meio de uma metodologia simples e bem humorada, através de uma interface diferente, com vídeos divertidos e tudo muito bem feito. O *Rails for Zombies* foi feito para quem está começando com programação. O usuário vai passando por níveis a medida que codifica os exercícios da maneira correta. O seu editor é bem simples e fácil de usar e, o compilador, responde de maneira muito rápida.

Este website ficou muito famoso devido a sua técnica utilizada para o ensino e também por causa da velocidade que seu compilador apresentava, ficando fácil e rápido escrever o código dentro dele.

Abaixo uma imagem da tela do *Rails for Zombies*:

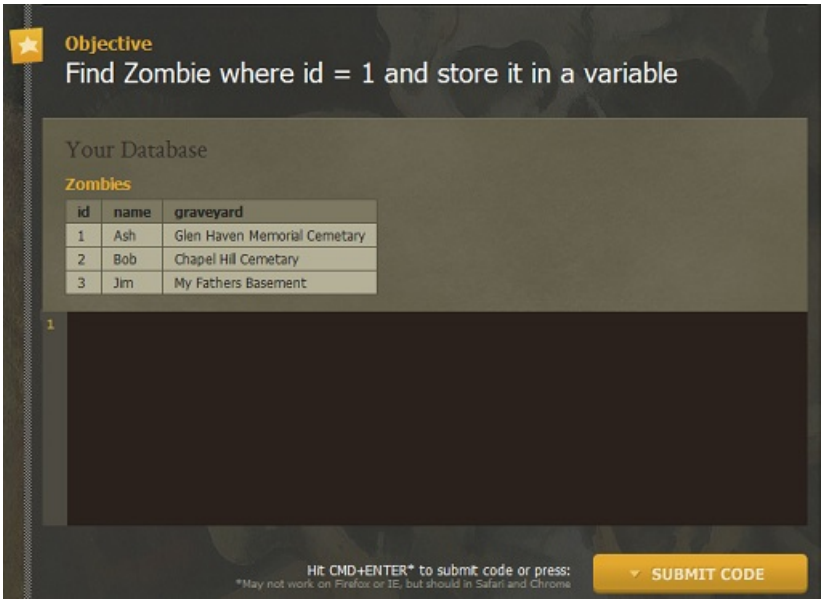

Figura 3.5: codeschool.com

3.4 Conclusão

Pode-se concluir que esta área é ainda nova, mas que se apresenta interessante e vários profissionais da área de computação vêm mostrando que ela é importante. O fato é que para vários fins, se faz necessário a presença de um compilador online, pois esse poderá ser acessado de qualquer lugar, indepedente da máquina.

Capítulo 4

Metodologia e Análise do Protótipo Proposto

4.1 Introdução

A criação de um compilador que realiza comunicação com um navegador e realiza suas operações pela internet não é tão trivial de se fazer. Os compiladores tem como característica o fato de poderem demorar em uma determinada compilação, dependendo é claro da complexidade do algoritmo criado. Já o os navegadores de internet devem sempre ter uma resposta imediata e rápida na medida do possível. Faz-se então necessário que um compilador online deva manter uma forma simples e limpa, para que facilite esse processo e seja então viável.

Este trabalho propõe uma metodologia de compilador online, ou seja, um sistema que funcione em um navegador para a internet, compilando determinado trecho de código e retornando o resultado para o usuário.

Basicamente o protótipo proposto é baseado em dois módulos principais, sendo eles o compilador e o sistema web. Entre estes módulos existe portanto uma comunicação, ou seja, o usuário acessa o sistema através da internet e insere o seu código. O sistema então enviará esse código para a compilação e então, depois de compilado, o compilador retorna a resposta para sistema web que apresentará a mesma ao usuário.

O compilador foi criado utilizando a linguagem C++, visto que tal linguagem apresentou-se fácil para a construção do mesmo, além de possuir uma documentação vasta e ser bastante robusta. Para o processo de *backend* foi utilizado o NASM (Netwide Assembler)

4.2 Compilador

Como visto no capítulo 1, o compilador apresenta fases da compilação, sendo elas a fase de análise e fase de síntese. Cada uma destas fases são dividas em outras fases menores, sendo cada uma delas responsáveis por tarefas durante todo o processo. Na fase de análise (também conhecida como *frontend*) o processo se divide em três, análise léxica, sintática e semântica. Tais fases são responsáveis por identificar e informar a grande maioria dos erros por parte do programador.

4.2.1 Análise Léxica

A analise léxica é o primeira módulo do *frontend* de um compilador e basicamente este módulo é responsável por receber um arquivo de entrada (programa fonte) e "quebrá-lo"em palavras conhecidas como tokens. Os analisadores léxicos também têm a função de descartar coisas que não terão importância para a compilação de um arquivo fonte tais como: Espaços em branco e comentários.

Como resultado um analisador léxico verifica se um determinado código é ou não válido de acordo com a gramática da linguagem regular descrita e gera uma lista de tokens que será repassada para os outros módulos do processo de compilação A implementação de um analisador léxico requer logicamente que a linguagem regular (a qual o analisador léxico ira obedecer as regras) seja descrita formalmente e para isto faz-se necessário o uso de algumas estruturas de representação desta linguagem tais como as Expressões Regulares que será a base para a construção de um NFA(Autômato Finito Não Determinístico) e a partir deste NFA é contruído um DFA (Autômato Finito Determinístico) que representa a linguagem gramaticamente e é a base para a codificação usando uma linguagem de programação do analisador léxico para uma determinado linguagem regular.

Na figura 4.1 vê-se um NFA que tem como função representar os operadores da linguagem.

Já na figura 4.2 temos um outro NFA, responsável por representar os comentários de linha.

4.2.2 Análise Sintática

Terminada a etapa de análise léxica do nosso compilador, passamos agora para a segunda etapa de construção do mesmo: A análise Sintática. A etapa de análise sintática ou em inglês *parser* recebe como entrada uma sequência de tokens do analisador léxico e determina se a string pode ser gerada através da linguagem fonte. Para este compilador foi criado um analisadore sintático do tipo *LALR(1)*.

O analisador sintático preditivo é um algoritmo simples, capaz de fazer o *parsing* de

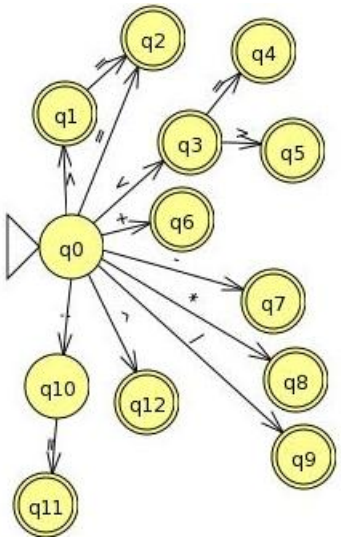

Figura 4.1: NFA que representa as palavras reservadas da linguagem

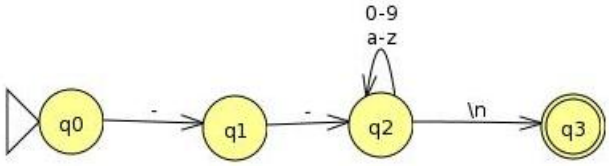

Figura 4.2: NFA que representa os comentários de linha da linguagem

algumas linguagens. Neste tipo de analisador cada produção da linguagem fonte torna-se uma cláusula em uma função recursiva, tendo-se uma função para cada não-terminal da produção. Como visto, cada função relativa a um não-terminal precisa conter um cláusula para cada produção. Desta forma faz-se necessário saber escolher qual a produção mais apropriada para tal. Esta escolha é feita baseando-se no próximo *token*. E isto é feito através da *predictive parsing table*.

A maioria das linguagens de programação é LALR(1), sendo esta técnica o tipo mais usado em geradores automáticos de parsers, foi usado então um analisador LALR(1) para a linguagem Portugol.

4.2.3 Análise Semântica

A análise semântica é o ultimo módulo do *frontend* de um compilador. Como já foi visto a análise léxica é responsável por quebrar a entrada em palavras conhecidas como *tokens*. Já a análise sintática que é o segundo módulo do *frontend*, tem o objetivo de analisar a estrutura de frases de um programa e verificar também se determinada *string* pode ser gerada pelas derivações da gramática em questão. Por fim a análise sintática calcula o "significado"do programa realizando verificações de tipos e de declarações e seus respectivos usos.o

4.2.4 Backend

Após o processo de análise, o compilador gera código em assembly para, então, usar o NASM (Netwide Assembler) como *backend* para montar e criar um executável válido. Consequentemente, não existe etapa de linkagem. A fase de otimização de código também não foi implementada.

Para usar esse recurso, é nescessário que o NASM esteja instalado no sistema. Ele pode ser encontrado em [NASM, 2011].

4.3 Sistema de Programação Web

Como segundo módulo do protótipo proposto, tem-se a criação do Sistema de Programação Web que funciona com um editor de programas e também realiza a comunicação com o compilador. O sistema tem como nome 'PortugOn'.

O sistema foi criado utilizando a linguagem de programação Ruby, juntamente com o framework Rails, pois é uma linguagem de alto nível, capaz de realizar tarefas com menos linhas de código que várias outras linguagens, sendo esta também a linguagem a qual sou mais fluente. Foi escrito também código JavaScript, para facilitar na interação com o usuário. O banco de dados utilizado para a aplicação foi o MySQL, por ser um banco popular, de fácil integração com o Ruby on Rails e também por ser de fácil manuseio.

O PortugOn é dividido em telas, as quais cada uma tem as propriedades. A princípio temos a tela inicial (como pode ser observado na figura 4.3), que tem como finalidade apresentar o sistema através de algumas de suas características e também, contém um formulário de *login* para que o usuário possa realizar sua autenticação no sistema. Caso o usuário ainda não tenha cadastro ele pode clicar em 'Cadastre-se' e aí partir para uma outra tela, a de novo cadastro.

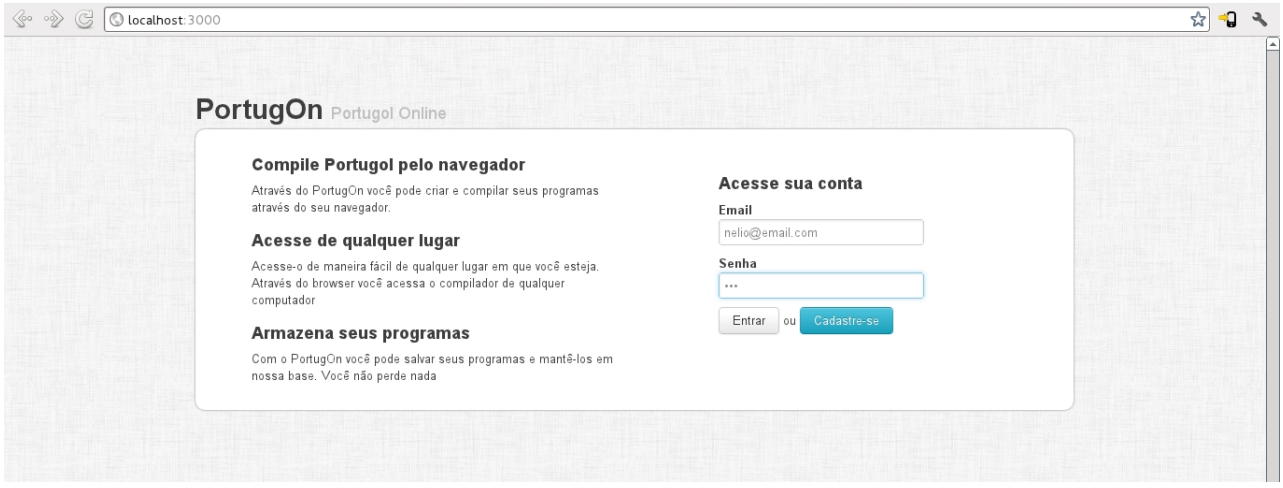

Figura 4.3: Página inicial do sistema PortugOn

Se então o usuário que acessar o sistema ainda não tiver cadastro, ele deverá realizar um novo. Ao clicar em 'Cadastre-se' na página inicial, o usuário irá para uma nova página (figura 4.4). Nessa nova página o usuário deverá preencher todos os campos pedidos e então clicar 'Cadastrar'. Após isso o usuário será cadastrado no banco de dados e poderá realizar o *login*. Caso o usuário não queira realizar seu cadastro ele pode clicar em 'Cancelar' e voltar à pagina inicial.

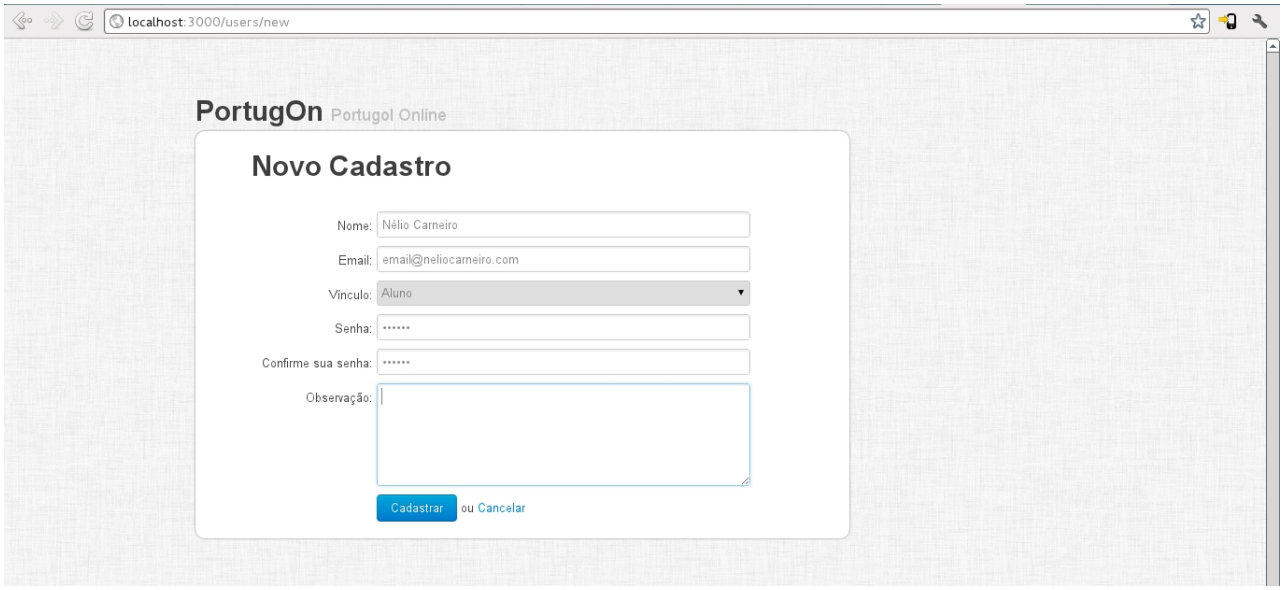

Figura 4.4: Página de cadastro de usuário do sistema PortugOn

Após o usuário efetuar o seu cadastro e entrar no sistema, ele irá para a página 'Home' (figura 4.5). Nesta página ele irá encontrar um menu no topo, podendo navegar pelos links lá visualizados. Nessa mesma página encontra-se o editor onde o usuário deve entrar com o código Portugol. Ao entrar com o código do programa e clicar em 'Submit', o código será compilado e o retorno poderá ser observado na coluna ao lado, a coluna 'Resultado' (figura 4.6). O usuário receberá a mensagem relacionada ao seu programa, se tudo correu bem, será uma mensagem de acerto, caso contrário o usuário receberá um aviso de erro e a respectiva mensagem do compilador.

Cada vez que o usuário compila determinado programa, este fica salvo no banco de dados. O usuário pode então acessar tais programas clicando no menu 'Programas' (figura 4.7). Essa tela mostra os programas já feitos com a sua respectiva, data, podendo o usuário editar o programa ou excluí-lo.

Caso o usuário ao realizar seu cadastro tenha escolhido o seu vínculo como 'Professor', este terá em sua página o menu 'Exercícios'. Ao acessar tal menu, o usuário será capaz de criar um exercício à ser aplicado e também, listar todos os seus exercícios já criados (figura 4.8). As operções de editar e excluir podem também serem utilizadas nessa seção.

Se o usuário for do tipo 'Professor', aparecerá também no seu menu, o link 'Alunos'.

Figura 4.5: Pagina Home do usuário do sistema PortugOn

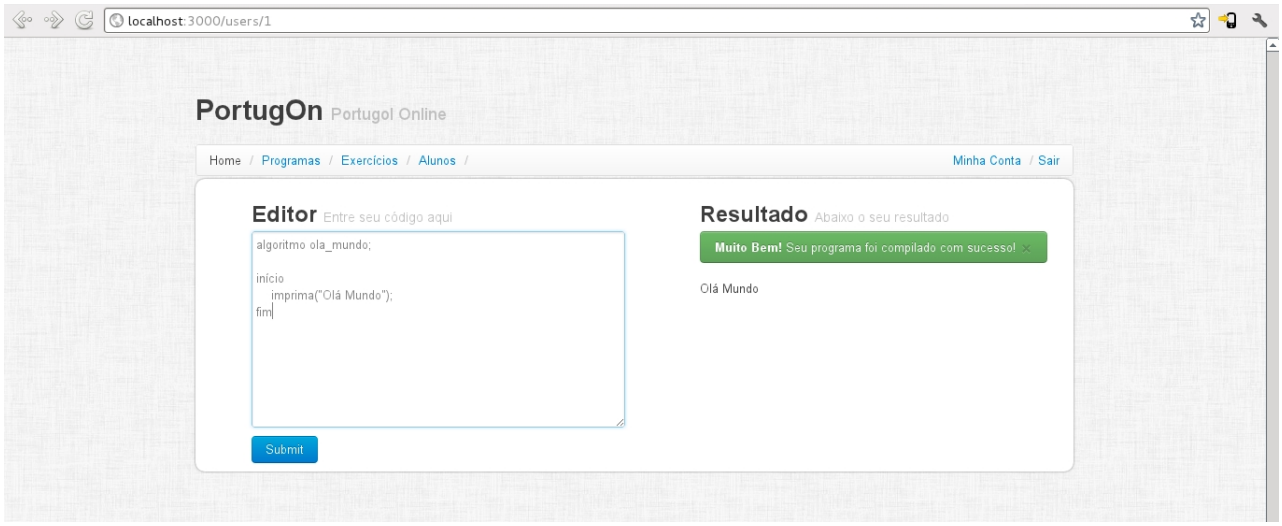

Figura 4.6: Página Home do usuário, com código compilado

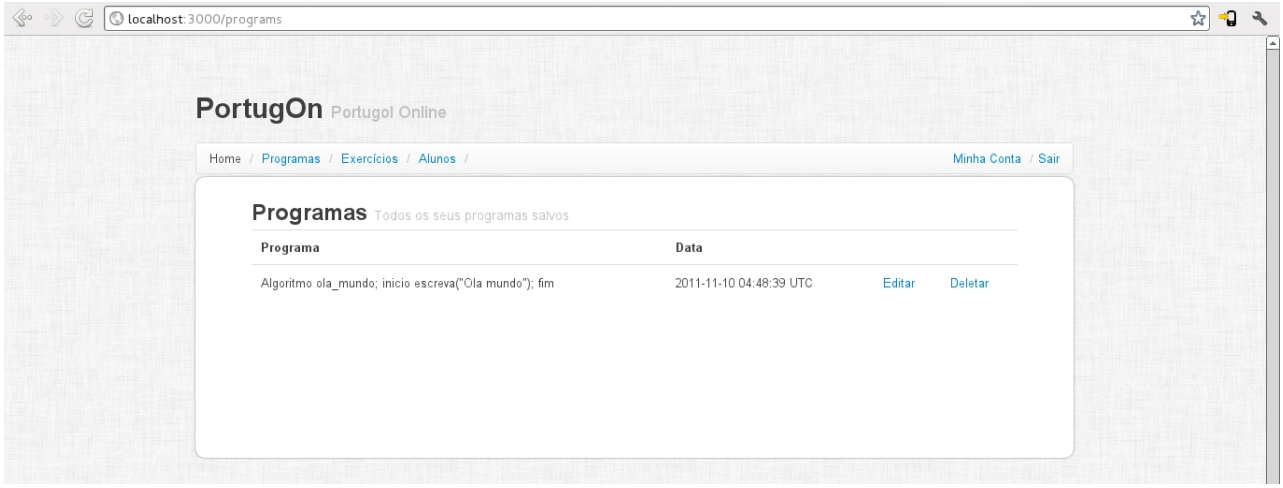

Figura 4.7: Página Programas. Lista os programas já efetuados pelo usuário

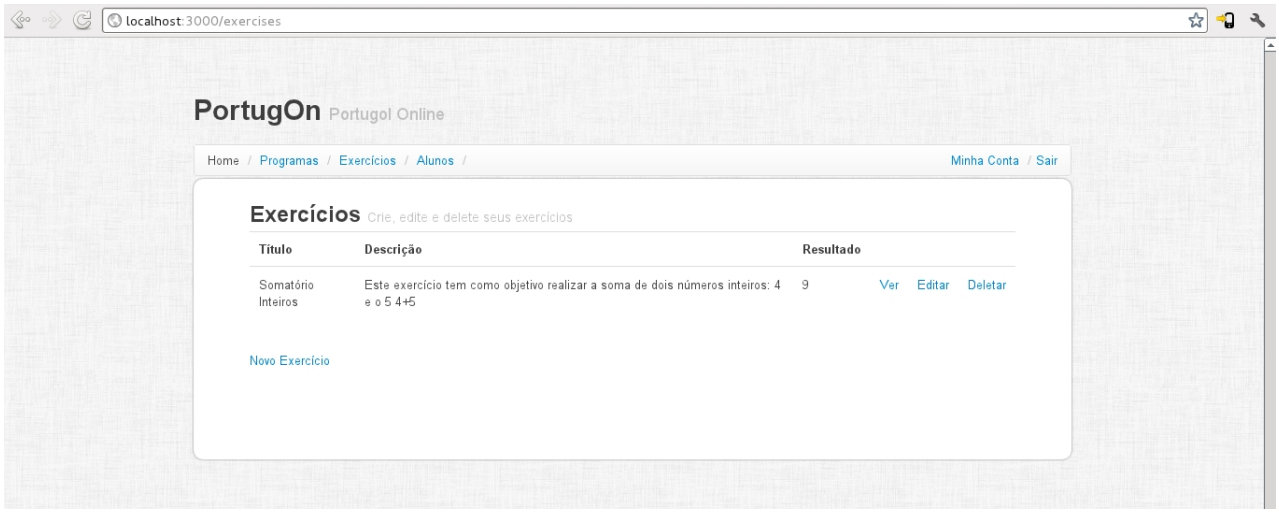

Figura 4.8: Página Exercícios. Lista os exercícios criados pelo usuário

Ao acessar este link o usuário irá para a página responsável por listar os alunos atrelados aquele professor. Será possível também cadastrar um novo aluno, bem como excluir os já existentes (figura 4.9). O usuário pode também, clicar nos exercícios de cada aluno para poder ver a resposta do aluno para o determinado exercício.

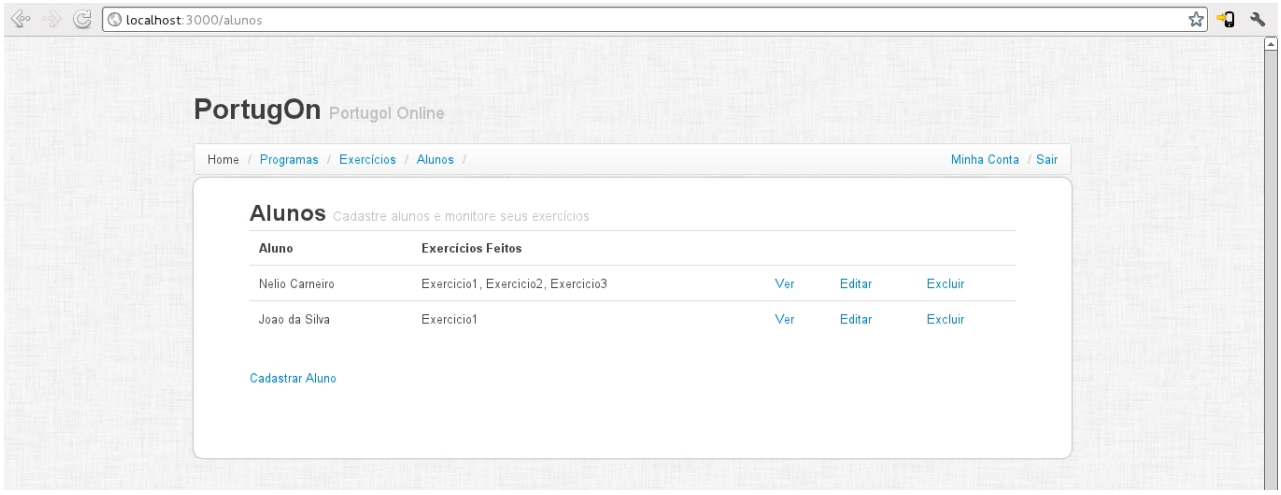

Figura 4.9: Página Alunos. Lista os alunos criados pelo usuário

Ao encerrar as atividades no sistema, o usuário deve fazer o *logout*, ou seja, encerrar sua sessão no sistema. Para isso, basta que ele clique em 'Sair', no menu ao lado direito.

Capítulo 5

Teste e Análise dos Resultados

Referências

Aho, A. V., Sethi, R., and Ullman, J. D. (1995). *Compilers, Principles, Techniques and Tools*. Company, Reading, Massachusetts, USA.

Backus, J. W. (1957). *The FORTRAN Automatic Coding System*. Western joint computer conference: Techniques for reliability, Los Angeles, California, USA.

Farrer, H. (1999). *Algoritmos Estruturados: Programação Estruturada de Computadores*. LTC - GRUPO GEN.

Filho, C. F. (2007). *História da Computação*. EDIPUCRS, Porto Alegre, RS, Brasil.

Henriques, A. A. R. (2000). *Breve história da linguagem Fortran*. Disponível em: http://paginas.fe.up.pt/ aarh/pc/PC-capitulo2.pdf.

Naur, P. (1963). *Revised report on the algorithmic language Algol 60*. Comm. ACM, USA.

Rangel, J. L. (1999). *Compiladores*. PUC-Rio, Rio de Janeiro, RJ, Brasil.

Rigo, S. (2008). *Análise Léxica*. Unicamp, Campinas, SP, Brasil.

Ripley, G. and Druseikis, F. (1978). *A Statistical Analysis of Syntax Errors*. J. Computer Languages, Vol. 3, USA.

Rosen, S. (1967). *Programming Systems and Languages*. USA.

Ullman, J. D., Aho, A. V., and Sethi, R. (1977). *Compilers, Principles, Techniques and Tools*. Company, Reading, Firs Edition, Massachusetts, USA.

Wirth, N. (1996). *A Statistical Analysis of Syntax Errors*. Addison-Wesley Pub, Zurich, Suiça.