UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO

Análise Léxica

Relatório 1: Análise Léxica
Construção de Compiladores 2
Profª. Drª. Helena de Medeiros Caseli

César Roberto de Souza 298603
Leonardo Sameshima Taba 298816

São Carlos, 16 de setembro de 2009

1. Introdução

A análise léxica é a primeira etapa na elaboração de um compilador. Neste trabalho, foi desenvolvido um analisador léxico para a linguagem hipotética *ALG*, implementado inteiramente em **Client-Side JavaScript**. A tecnologia JavaScript permite que nosso compilador seja executado em qualquer plataforma, desde que para tal plataforma exista um navegador web que ofereça suporte a esta linguagem.

A linguagem JavaScript é particularmente interessante pois é uma linguagem dinâmica, e, como tal, permite uma maior liberdade em implementação ao custo de uma certa perda em eficiência. Entretanto, esta desvantagem, com o advento das últimas tecnologias incorporadas aos navegadores mais recentes, tem se tornado cada vez menos relevante, tendo em vista que as novas máquinas virtuais JavaScript aceleram e otimizam código dinâmico transformando-o em código de máquina nativo antes de sua execução.

2. Desenvolvimento

2.1. Visão Geral

O analisador léxico foi implementado utilizando-se a técnica de Orientação a Objeto. Assim, projetamos uma classe **AnalisadorLexico** que fosse capaz de, a partir de uma entrada em texto qualquer, identificar tokens presentes nesta entrada na ordem em que aparecem.

Tal classe possui apenas um método principal, denominado **getToken()** que retorna o próximo token presente na entrada. Também é possível navegar nesta entrada utilizando-se **setIndex(**posicao**)** e verificar se atingimos o final da entrada através da função **eof()** ou se o token retornado for nulo.

Subseqüente a esta classe principal, definimos as classes Token e Keywords. A classe **Token** consiste basicamente num par contendo um lexema e a categoria do lexema identificado, como, por exemplo, <*42-numero_inteiro*> ou <*myVar-identificador*>. Esta classe possui um método adicional toString() que retorna o Token no formato esperado da saída determinado na especificação deste projeto.

Já a classe **Keywords** nada mais é do que uma implementação (eficiente) da tabela de palavras reservadas. Levando-se em consideração que, internamente, todo objeto em JavaScript é implementado como uma tabela hash, pudemos tomar esta característica peculiar como vantagem, implementando a tabela de palavras reservadas como um simples objeto. Desta maneira, para verificarmos se um determinado token está contido na tabela de palavras reservadas, podemos utilizar simplesmente a expressão:

if (myTokenString **in** Keywords**)**

Que retorna *true* ou *false* caso o lexema seja palavra reservada ou não. Internamente, será feita uma busca em tabela hash na tabela que armazena as propriedades do objeto.

Finalmente, mas sem menos importância, definimos o *main entrypoint* de nossa aplicação, onde o analisador léxico será utilizado. Na função **Main**, criamos uma nova instância da classe AnalisadorLexico e passamos o texto de entrada como parâmetro de seu construtor. Assim bastou-nos utilizar um laço *do-while* que é executado enquanto ainda houver tokens na entrada, pegando e exibindo todos os tokens encontrados até então.

2.2 Decisões de Projeto

A linguagem JavaScript foi escolhida por ser uma linguagem independente de plataforma, já que sua execução ocorre dentro do navegador. Outro motivo é que não é necessário realizar o download de programa algum, facilitando a distribuição da aplicação, bastando possuir uma conexão com a internet.

É importante notar que alguns autômatos que desempenham funções semelhantes (como, por exemplo, reconhecimento de números inteiros e reconhecimento de números reais) foram agrupados de forma a melhor aproveitar o código e otimizar a execução. Ao final da execução de um autômato é retornado o token reconhecido ou **null**, indicando que houve falha durante o reconhecimento. Em caso de falha, o método getToken passará a vez para outro autômato tentar reconhecer a cadeia, até que a cadeia seja finalmente reconhecida ou rejeitada por todos autômatos. Neste caso, ocorre um erro de reconhecimento.

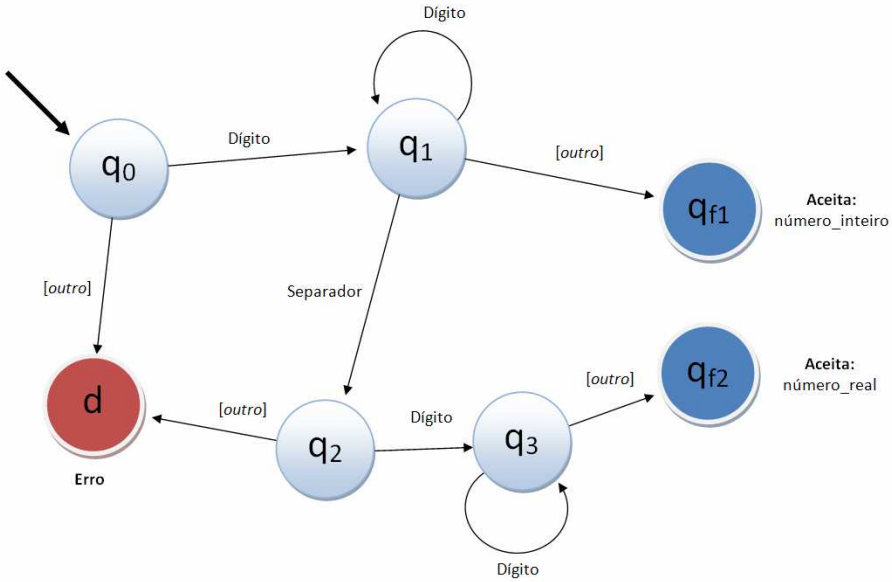

Figura 1. Autômato reconhecedor de números (inteiros e reais)

O tratamento de erros léxicos foi feito utilizando-se um único estado de erro por autômato. Quando um erro é encontrado durante o processamento, sua execução é interrompida e o controle passa para o próximo autômato. Quando nenhum autômato consegue reconhecer uma cadeia, ela é considerada um erro léxico.

No entanto, existem tipos especiais de erros que não podem ser retornados como tokens, como, por exemplo, erros de comentários. Comentários podem ter uma ou várias linhas, dependendo apenas de onde aparecem os delimitadores "{" e "}". Um erro especial "*fim de comentário não encontrado*" ocorre quando um bloco de comentário é aberto e não é fechado. Comentários, no entanto, assim como espaços em branco, são simplesmente ignorados pelo analisador léxico. Assim não é correto retornar um token contendo o erro, pois, por definição, comentários não devem ser reconhecidos como tokens.

Para gerenciar este tipo de erro, foi inserido um método adicional **error()** na classe AnalisadorLexico que indica se o analisador encontrou algum erro que não pôde ser transformado em token na chamada anterior de getToken(). O erro é retornado como um objeto da classe **Error**, criada especificamente para esse fim.

3. Compilação e execução

3.1. Compilando

Por JavaScript ser uma linguagem interpretada, não há necessidade de compilação. No entanto, alguns browsers modernos, como por exemplo o Google Chrome, realizam uma compilação Just-in-Time (JIT) do código para tornar sua execução mais ágil. Isso ocorre de maneira transparente para o usuário.

3.2. Executando

A execução do analisador é feita através de um navegador. Portanto, basta clicar duas vezes no arquivo index.html da pasta raiz do compilador para iniciá-lo.

Sua interface consiste em apenas duas caixas de texto, uma delas destinada ao código fonte de entrada e a outra dedicada a mostrar a saída, ou seja, os tokens identificados e suas respectivas categorias.

O código fonte a ser processado é colocado na caixa de texto superior. Em seguida, deve-se clicar no botão "Processar" para colocar o analisador léxico em execução. Seus resultados serão então escritos na caixa inferior.

3.2.1. Exemplos de execução

Entrada	Saída
programa exemplo; {entrada} var a: inteiro; inicio leia(a, @, 1); fim.	programa - programa exemplo - identificador ; - ; var - var a - identificador : - : inteiro - inteiro ; - ; inicio - inicio leia - identificador (- (a - identificador , - , @ - erro , - , 1 - numero_inteiro) -) ; - ; fim - fim . - .

Entrada	Saída

programa fibonacci;
{ imprime os n primeiros numeros de fibonacci }
var n1, n2, n3, qtde, i : inteiro;
inicio
    n1 := 0;
    n2 := 1;
    i := 0;
    le(qtde);
    enquanto i < qtde faca
    inicio
        escreve(n2);
        n3 := n1 + n2;
        n1 := n2;
        n2 := n3;
        i := i + 1;
    fim;
fim.
	programa - programa
	fibonacci - identificador
	; - ;
	var - var
	n1 - identificador
	, - ,
	n2 - identificador
	, - ,
	n3 - identificador
	, - ,
	qtde - identificador
	, - ,
	i - identificador
	: - :
	inteiro - inteiro
	; - ;
	inicio - inicio
	n1 - identificador
	:= - :=
	0 - numero_inteiro
	; - ;
	n2 - identificador
	:= - :=
	1 - numero_inteiro
	; - ;
	i - identificador
	:= - :=
	0 - numero_inteiro
	; - ;
	le - le
	(- (
	qtde - identificador
	) -)
	; - ;
	enquanto - enquanto
	i - identificador
	< - <
	qtde - identificador
	faca - faca
	inicio - inicio
	escreve - escreve
	(- (
	n2 - identificador
	) -)
	; - ;
	n3 - identificador
	:= - :=
	n1 - identificador
	+ - +
	n2 - identificador
	; - ;
	n1 - identificador
	:= - :=
	n2 - identificador
	; - ;
	n2 - identificador
	:= - :=
	n3 - identificador
	; - ;
	i - identificador
	:= - :=
	i - identificador
	+ - +
	1 - numero_inteiro
	; - ;
	fim - fim
	; - ;
	fim - fim
	. - .