Relatório do primeiro problema de compiladores

Siloé Bezerra Bispo[1]

Universidade Estadual de Feira de Santana (UEFS)
BA – Brazil

siloebb@gmail.com[1]

Resumo. *Este relatório procura descrever como foi feita a implementação de um analisador léxico, mostrando como aplicou-se o uso de autômatos finitos, quais foram as classes e os metódos mais importantes e como eles funcionam, mostra também os testes feitos e os resultados obtidos.*

1. Introdução

Foi pedido um software capaz de fazer a analíse léxica de um conjunto e regras de palavras passado por dois amigos que aparentemente contactaram alienígenas. O software é o primeiro passo de um compilador que será projetado para ler uma linguagem. Abaixo será mostrado o planejamento, metódos, materiais e testes feitos sobre o desenvolvimento do software e as conclusões sobre o produto finalizado.

2. Materiais e metódos

Para se criar o analisador léxico primeiro foi feito um autômato finito determinístico que fosse capaz de ler um código utilizando as expressões passadas pelos meninos que ouviram os alienígenas.

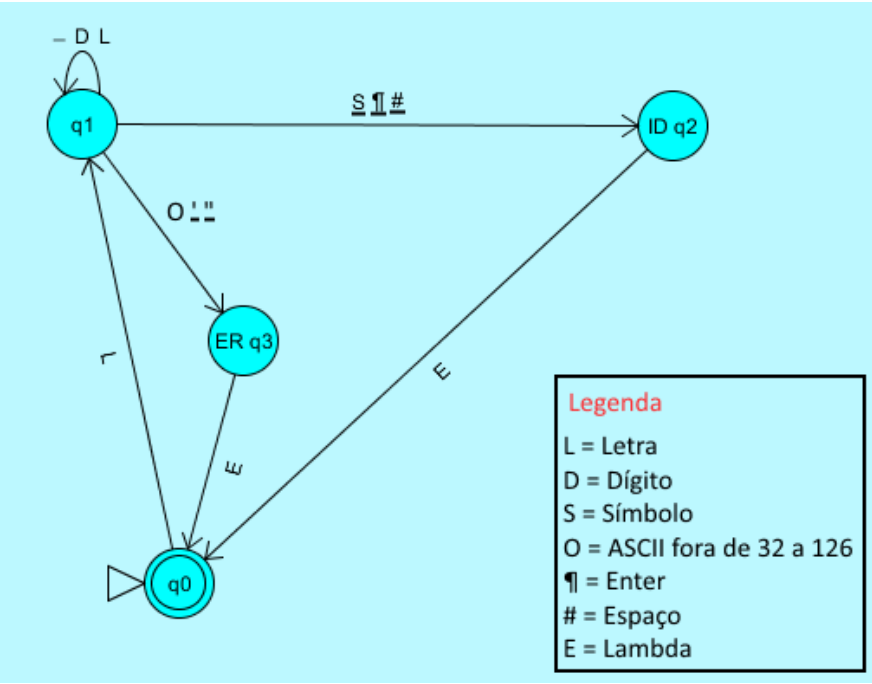

Figura 1. Parte do autômato que reconhece um identificador

Na figura 1 mostra uma parte de todo o autômato que reconhece um identificador e volta para o estado inicial onde existem outras transições e estado que está no autômato projetado, que por ser muito grande não estará por completo neste relatório, porém ele segue o mesmo padrão em todos as outras partes do autômato. Algumas adaptações foram feitas para facilitar na hora da implementação, pode-se ver na imagem que diferente de *q0* e *q1* os estados *q3* e *q2* possuem as sílabas *ER* e *ID,*essas siglas servem para saber na hora de implementar que se o autômato chegou em *q3* ele irá avisar um erro léxico na cadeia que está sendo lida, porém o autômato continuará a ser executado, quando o autômato passar por *q2* o autômato avisará que um *token* do tipo identificador foi encontrado, todas as outras partes do autômato seguem este mesmo padrão para alertar erros e os tipos de *tokens*. Outra observação é que os itens sublinhados como S, ¶ e # avisam que este símbolo será examinado mas não será consumido, continuando o autômato no próximo estado.

Depois de criado o autômato utilizou-se a linguagem *JAVA* para implementar uma simulação de como funciona um autômato finito determinístico, utilizou-se três classes principais que foram *Automata, State e Transition* onde a primeira obtém os estados e transições e simula o autômato, a segunda são os estados do autômato e a terceira as transições do autômato. Também existe uma classe chamada *Identificador* que é responsável por dizer o tipo em que se enquadra o caractere que será consumido, se é uma letra, digito, lambda, símbolo, espaço, fim de linha ou outro tipo que não seja algum destes.

```
private static int startStateLimit = 0;

private HashMap<Integer, State> states;
private HashMap<Integer, State> finalStates;
private State startState;
private HashSet<Transition> transitions;
```

Figura 2. Variavéis da classe Automata

Na classe *Automata* utilizamos coleções do tipo *HashMap* e *HashSet* para armazenar estados e transições. O *HashMap* foi usado pois pode-se passar o número do estado e ele retornará o objeto que representa o estado e o *HashSet* foi escolhido para as transições pois ele não armazena dois itens que sejam iguais, pois uma transição com mesma origem e destino para um determinado símbolo seria redundante.

A classe *State* além de possuir o número do estado também tem outras duas variáveis para armazenar um *Token* e um Erro. Estas duas variáveis são dois tipos de classes que foram criadas para que armazenem o tipo de *token* ou erro, a linha onde ocorreu e o lexema para que depois sejam exibidos ao usuário do compilador. A classe *Transition* além de armazenar o estado destino, o estado origem e o símbolo também possui uma *flag* que indicará se o símbolo será ou não consumido.

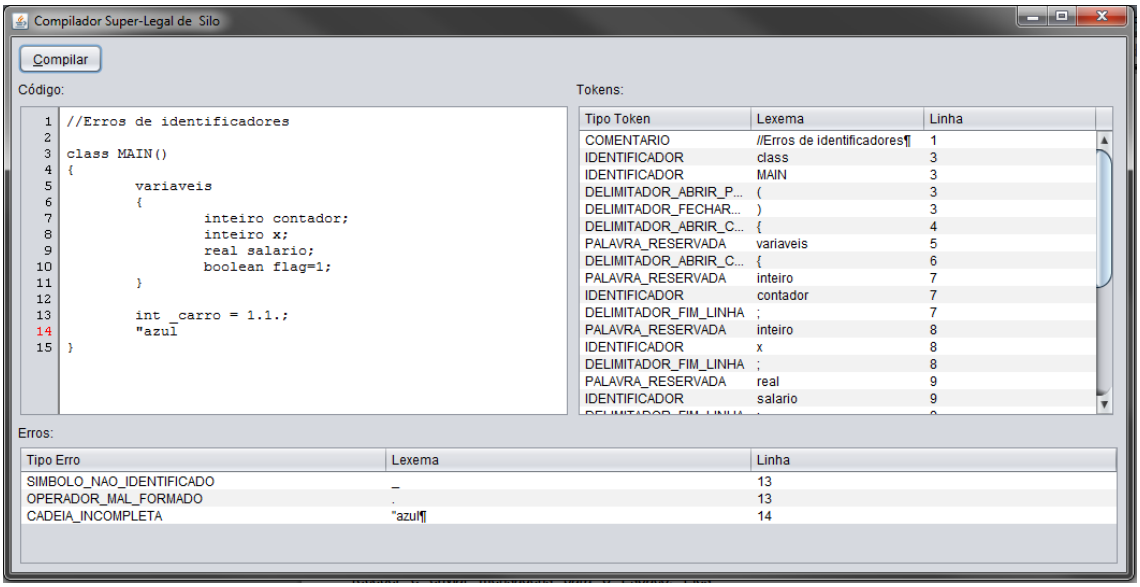

Figure 3. Tela do programa

A figura 3 mostra como é a tela do programa; onde existe um botão para compilar o programa, uma área para escrever o código do programa, uma tabela onde os *tokens* do programa são listados com o tipo do *token*, lexema e linha da ocorrência e outra tabela onde são exibidos os erros léxicos do código que descrevem o tipo de erro, o lexema e a linha onde ocorreu. O uso do *software* é simples, basta escrever o código no local apropriado e clica no botão "Compilar" que irá fazer a análise léxica e depois aparecerá uma mensagem falando se houve erros ou não.

3. Experimentos e análise de resultados

Testes foram feitos para verificar se o software era capaz de identificar os *tokens* e os erros e dar os detalhes exigidos, também foram feitos testes para verificar se o programa era capaz de se recuperar caso existisse um símbolo que não possuísse transição no estado onde se encontra.

Foram planejados testes para identificadores, números, comentários, símbolos, cadeia constante e caractere constante onde existiam representantes montados de maneira correta e incorreta no código, depois foram feitos testes com erros misturados e código sem erros.

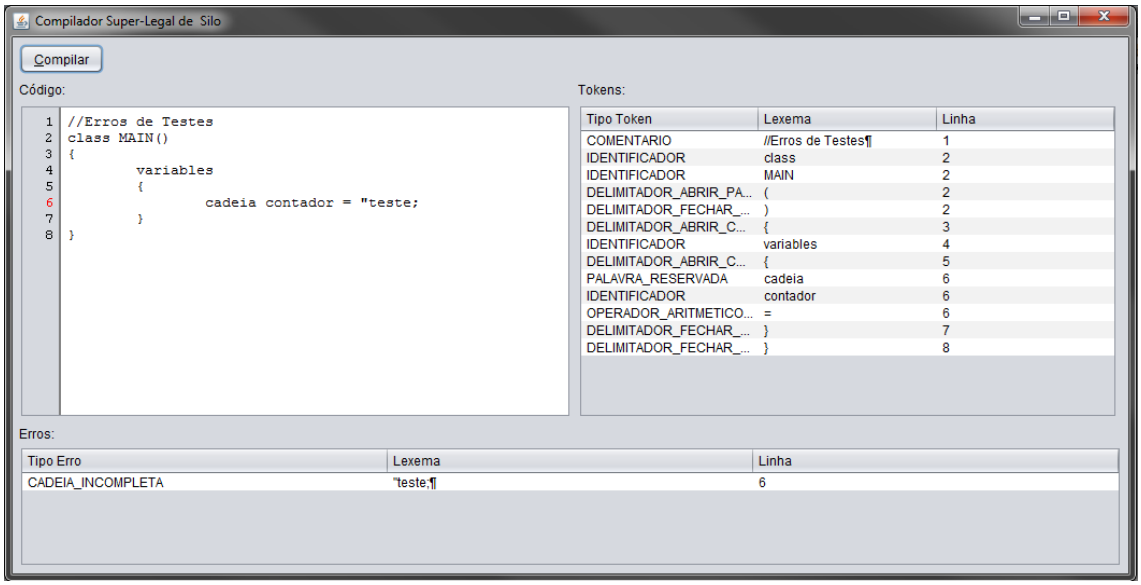

Figure 4. Teste de cadeia constante

A figura 4 mostra um teste feito para cadeia constante, onde se inicia e finaliza a cadeia constante com “ (aspas duplas), porém neste código não se encontra a aspas duplas para finalizar a cadeia constante, gerando assim um erro léxico que é reportado na tabela de erros indicando que a cadeia está incompleta e que este erro ocorreu na linha seis.

Durante os testes houveram alguns erros devido à algumas falhas no autômatos que foram rapidamente corrigidos acrescentando os estados e transições necessários. Então novamente foi feita a bateria de testes onde ocorreu tudo sem erros no software.

4.Conclusão

O software está se comportando de maneira satisfatoria apresentando o que foi pedido no problema, uma tabela de *tokens* do código e uma tabela de erros do código.

Utilizou-se uma simulador de autômatos finitos determinísticos para criar o software, o que apresentou ser uma maneira rápida e eficaz de desenvolvimento do programa, pois o trabalho se concentra em projetar o autômato e depois descrevê-lo no código, poupando assim muitos erros gerados por uma programação complexa e aninhada.