



UNIVERSIDADE FEDERAL DE OURO PRETO
DEPARTAMENTO DE COMPUTAÇÃO - DECOM
Professor: Rodrigo Ribeiro

Relatório TP1
Disciplina : BCC 328- Compiladores

Alunos:
Caio Silas de Araujo Amaro - 21.1.4111
Henrique Dantas Pighini - 21.1.4025

Ouro Preto
2023

Introdução

Neste trabalho foi proposto a criação de um analisador léxico para o compilador da linguagem lang, que foi desenvolvida pelo professor Rodrigo Ribeiro.

Primeiro vamos lembrar o que é um analisador léxico:

Sua função principal é ler o código-fonte em linguagem de programação e dividir esse código em unidades menores chamadas "tokens". Tokens são os elementos básicos da linguagem de programação, como palavras-chave, identificadores, números, operadores e símbolos especiais. Lembrando que esses Tokens são indivisíveis.

Desenvolvimento

Para gerenciar esse projeto em Haskell utilizamos o **cabal**, este possui uma funcionalidade interessante “\$ cabal init” onde abre uma interface de criação de projetos onde montamos o escopo do nosso projeto e começamos o desenvolvimento do mesmo.

Para construir o Analisador léxico em si, achamos interessante a aplicação do Alex para otimizar e automatizar a construção do mesmo.

Aproveitamos o exemplo de Alex que nos foi disponibilizado e foi usado como base para começarmos nossos testes com o Alex e com a função de análise léxica.

Analizador léxico

Como explicitado para a criação dos autómatos para os Tokens utilizamos o Alex. Para que essa ferramenta consiga criar de maneira eficiente nosso Analisador léxico é necessário que alguns campos sejam implementados:

- Wrapper deve ser definido.
- Macros devem ser definidos para auxiliar na construção das expressões regulares.
- Os tokens devem ser definidos e suas respectivas expressões regulares também.
- Deve ser criada uma função que recebe uma string e retorna uma lista de tokens que serão criados pela função “alexScanTokens”:

```
lexer :: String -> [Token]
lexer = alexScanTokens
```

Sendo assim teremos:

1) Entrada:

A entrada para o Alex é a especificação escrita em um formato próprio. Esta especificação descreve as regras para identificar tokens em um código fonte. A especificação contém informações sobre os padrões de texto a serem reconhecidos (expressões regulares) e as ações associadas a cada padrão.

```

25  ✓ tokens :-
26      $white+                ; -- removing whitespace
27      "//".*                  ; -- removing line comments
28      "Int"                   { \ _ -> TInt }
29      "Char"                  { \ _ -> TChar }
30      "Bool"                  { \ _ -> TBool }
31      "Float"                 { \ _ -> TFloat }
32      \[                       { \ _ -> TLeftBracket }
33      \]                       { \ _ -> TRightBracket }
34      \.                       { \ _ -> TDot }
35      \(                       { \ _ -> TLeftParen }
36      \)                       { \ _ -> TRightParen }
37      \!                       { \ _ -> TNot }
38      \*                       { \ _ -> TMul }
39      \/                       { \ _ -> TDiv }
40      \%                       { \ _ -> TMod }
41      \+                       { \ _ -> TAdd }
42      \-                       { \ _ -> TSub }
43      \<                       { \ _ -> TRelac }
44      "=="                     { \ _ -> TEqual }
45      "!="                     { \ _ -> TNEqual }
46      "&&"                     { \ _ -> TConj }
47      \=                       { \ _ -> TAssign }
48      "if"                     { \ _ -> TIIf }
49      "else"                   { \ _ -> TElse }
50      "read"                   { \ _ -> TRead }
51      "print"                  { \ _ -> TPrint }
52      "return"                 { \ _ -> TReturn }
53      "iterate"                { \ _ -> TIterate }
54      \{                       { \ _ -> TLeftKeys }
55      \}                       { \ _ -> TRightKeys }
56      \:                       { \ _ -> TColon }
57      "::"                     { \ _ -> TDoubleColon }
58      \,                       { \ _ -> TComma }
59      \;                       { \ _ -> TSemicolon }
60      @number                  { \ s -> TNumber (read s) }
61      @identifier              { \ s -> TId s }
62

```

Exemplo da definição do comportamento dos tokens

É interessante observar que a ordem de declaração dos Tokens tem importância. Quando fomos realizar testes percebemos que qualquer valor alfabético era reconhecido como um “Id” sendo que às vezes era uma palavra reservada como “Int” ou “Bool”.

Assim, definimos o Token “Id” por último pois ele tentaria encaixar o valor lido primeiro com todos os Tokens antes de Id.

2) Saída:

A saída do Alex é um arquivo Haskell que contém o código fonte do analisador léxico gerado. Esse código inclui definições para os tokens (geralmente representados por tipos de dados) e funções que implementam a lógica para reconhecimento de tokens. As funções geradas recebem a entrada (o código fonte) e produzem a saída correspondente (uma lista de tokens).

Após a geração do código Haskell pelo Alex, você pode incorporar esse código em seu projeto Haskell. No restante do seu projeto, você pode chamar as funções geradas pelo analisador léxico para processar o código fonte e obter a lista de tokens correspondente.

Instruções de Compilação:

versões utilizadas:

- Cabal 3.10.2.0
- ghc 4.19.0.0
- Alex 3.2.4

1) É MUITO IMPORTANTE, PARA EVITAR PROBLEMAS DE COMPILAÇÃO, QUE OS SEGUINTE SEJAM RESPEITADOS:

- Versões de todos os programas devem ser **IGUAIS** às versões utilizadas na produção do projeto
- É **NECESSÁRIO** alterar o nome do diretório que vem do github:

```
PS C:\Users\henri\Documents\Faculdade\6periodo\BCC328-CONSTRUCAODECOMPILADORESI\Praticas\Trabalhos\trabalho-pr-tico-01-analisador-l-xico-henrique_caio-main>
```

Este é um exemplo de como ficou o caminho até o projeto pelo projeto baixado pelo GitHub.

O problema é que a linguagem Haskell é sensível a caracteres especiais em nomes de diretórios. Assim, essa pasta que veio do github está acarretando em problemas de compilação:

```
PS C:\Users\henri\Documents\Faculdade\6periodo\BCC328-CONSTRUCAODECOMPILADORESI\Praticas\Trabalhos\trabalho-pr-tico-01-analisador-l-xico-henrique_caio-main> cabal new
-build
Resolving dependencies...
Build profile: -w ghc-9.6.1 -01
In order, the following will be built (use -v for more details):
- TP1-0.1.0.0 (lib) (first run)
- TP1-0.1.0.0 (exe:TP1) (first run)
Configuring library for TP1-0.1.0.0..
Preprocessing library for TP1-0.1.0.0..
Building library for TP1-0.1.0.0..
[1 of 1] Compiling Lexing.Lexer ( C:\Users\henri\Documents\Faculdade\6periodo\BCC328-CONSTRUCAODECOMPILADORESI\Praticas\Trabalhos\trabalho-pr-tico-01-analisador-
l-xico-henrique_caio-main\dist-newstyle\build\x86_64-windows\ghc-9.6.1\TP1-0.1.0.0\build\Lexing\Lexer.hs, C:\Users\henri\Documents\Faculdade\6periodo\BCC328-CONSTRUC
AODECOMPILADORESI\Praticas\Trabalhos\trabalho-pr-tico-01-analisador-l-xico-henrique_caio-main\dist-newstyle\build\x86_64-windows\ghc-9.6.1\TP1-0.1.0.0\build\Lexing\Le
xer.o )
Configuring executable 'TP1' for TP1-0.1.0.0..
ghc-pkg-9.6.1.exe: C:\Users\henri\Documents\Faculdade\6periodo\BCC328-CONSTRUCAODECOMPILADORESI\Praticas\Trabalhos\trabalho-pr-tico-01-analisador-l-xico-henr
ique_caio-main\dist-newstyle\build\x86_64-windows\ghc-9.6.1\TP1-0.1.0.0\x\TP1\package.conf.inplace: openBinaryTempFileWithDefaultPermissions: invalid argumen
t (Invalid argument)
Error: cabal-3.10.1.0.exe: Failed to build exe:TP1 from TP1-0.1.0.0.
```

Para resolver esse problema nós trocamos o nome do diretório que veio do github:

```
PS C:\Users\henri\Documents\Faculdade\6periodo\BCC328-CONSTRUCAODECOMPILADORESI\Praticas\Trabalhos\TP>
```

Com o nome agora sendo algo mais simples “TP”, podemos ver que o problema foi solucionado:

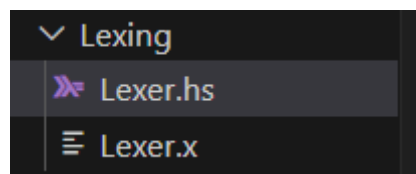
```
PS C:\Users\henri\Documents\Faculdade\6perodo\BCC328-CONSTRUCAODECOMPILADORESI\Praticas\Trabalhos\TP> cabal new-build
Resolving dependencies...
Build profile: -w ghc-9.6.1 -O1
In order, the following will be built (use -v for more details):
 - TP1-0.1.0.0 (lib) (configuration changed)
 - TP1-0.1.0.0 (exe:TP1) (configuration changed)
Configuring library for TP1-0.1.0.0..
Preprocessing library for TP1-0.1.0.0..
Building library for TP1-0.1.0.0..
[1 of 1] Compiling Lexing.Lexer      ( C:\Users\henri\Documents\Faculdade\6perodo\BCC328-CONSTRUCAODECOMPILADORESI\Praticas\Trabalhos\TP\dist-newstyle\build\x86_64-windows\ghc-9.6.1\TP1-0.1.0.0\build\Lexing\Lexer.hs, C:\Users\henri\Documents\Faculdade\6perodo\BCC328-CONSTRUCAODECOMPILADORESI\Praticas\Trabalhos\TP\dist-newstyle\build\x86_64-windows\ghc-9.6.1\TP1-0.1.0.0\build\Lexing\Lexer.o ) [Flags changed]
Configuring executable 'TP1' for TP1-0.1.0.0..
Preprocessing executable 'TP1' for TP1-0.1.0.0..
Building executable 'TP1' for TP1-0.1.0.0..
[1 of 1] Compiling Main                ( app\Main.hs, C:\Users\henri\Documents\Faculdade\6perodo\BCC328-CONSTRUCAODECOMPILADORESI\Praticas\Trabalhos\TP\dist-newstyle\build\x86_64-windows\ghc-9.6.1\TP1-0.1.0.0\x\x\TP1\build\TP1\TP1-tmp\Main.o )
[2 of 2] Linking C:\Users\henri\Documents\Faculdade\6perodo\BCC328-CONSTRUCAODECOMPILADORESI\Praticas\Trabalhos\TP\dist-newstyle\build\x86_64-windows\ghc-9.6.1\TP1-0.1.0.0\x\x\TP1\build\TP1\TP1.exe
```

2) Agora que suas versões são compatíveis e que o nome do diretório foi alterado, para compilar e testar nosso Analisador Léxico vamos seguir esse pequeno e modesto tutorial:

Para testar o analisador léxico primeiro deve-se, a partir do diretório base, rodar o Alex no arquivo “Lexer.x”:

```
\TP1> alex .\src\Lexing\Lexer.x
```

Após isso, será possível identificar que o arquivo Lexer.hs foi criado com sucesso:



A screenshot of a file explorer window showing a directory named 'Lexing'. Inside this directory, there are two files: 'Lexer.hs' and 'Lexer.x'. The 'Lexer.hs' file is highlighted with a mouse cursor.

Agora rodamos os comandos do Cabal para compilar nosso projeto:

```
\TP1> cabal new-build
```

```
\TP1> cabal new-repl
```

Com o código sendo compilado sem erros, agora, após “cabal new-repl” estamos dentro da ghci:

```
Ok, one module loaded.  
ghci> █
```

Criamos a função *testLexerFile* que irá receber um arquivo txt contendo a representação da linguagem Lang e irá executar o analisador léxico gerado automaticamente pelo alex a partir do arquivo Lexer.x que construímos:

```
1  (1+2) * !y  
2  //{  
3  1 < 2  
4  [] . () ! - * / % + - < == != &&
```

arquivo de teste “test/exp.txt”

A partir deste arquivo de entrada nos foi retornado:

```
ghci> testLexerFile "test/exp.txt"  
TLParen  
TNumber 1  
TAdd  
TNumber 2  
TRParen  
TMul  
TNot  
TId "y"  
TNumber 1  
TRelac  
TNumber 2  
TLBracket  
TRBracket  
TDot  
TLParen  
TRParen  
TNot  
TSub  
TMul  
TDiv  
TMod  
TAdd  
TSub  
TRelac  
TEqual  
TNEqual  
TConj  
ghci>
```


Para executar mais teste é interessante colocar o arquivo texto dentro da pasta “test” para que o comando de execução se mantenha sempre:

\$ testLexerFile “nomearquivo.txt”

Obs. O Alex também implementa casos de erro para o processamento de Tokens inválidos.