Universidade Federal de Juiz de Fora

Departamento de Ciência da Computação

Primeiro Trabalho
Teoria dos Compiladores (DCC045)(2021/1)

Implementação de um Analisador Léxico para a Linguagem Lang

Lucas Diniz da Costa - 201465524C

Junho 2021

Implementação de um Analisador Léxico para a Linguagem Lang

Lucas Diniz da Costa - 201465524C

1 Introdução

Este é um trabalho elaborado pelo aluno Lucas Diniz da Costa para a disciplina de Teoria dos Compiladores(2021/1) ministrada pelo professor Leonardo. O relatório a seguir visa salientar as observações relevantes sobre o primeiro trabalho que consiste na implementação de um analisador léxico para a linguagem Lang.

2 Desenvolvimento

O projeto foi desenvolvido utilizando a linguagem de programação Java, conforme as especificações do enunciado, e foi executado em uma máquina com as seguintes configurações de Hardware e Software:

- *Sistema Operacional*: Windows 10;
- *Memória RAM*: 16 Gb CORSAIR DDR4 2400 Mhz ;
- *Processador*: AMD Ryzen 5 3400G;

Contudo, na mesma máquina também foi testado o projeto no sistema operacional Linux Mint como solicitado pelo enunciado.

2.1 Geração do Analisador Léxico

Para a criação do analisador léxico foi considerada a opção de realizar uma implementação do zero, de acordo com as vídeo-aulas apresentadas na disciplina, mas para agilizar o procedimento e evitar potenciais erros, foi utilizada a ferramenta JFLEX conforme apresentado no vídeo *[DCC045] Geradores de Analisadores Léxicos* pelo professor Elton.

2.2 Scripts para a execução do projeto

Visando facilitar a execução tanto para os testes durante o desenvolvimento do projeto quanto para a execução nos dois sistemas, foram elaborados dois scripts para rodar o projeto tanto nos sistemas Linux quanto nos sistemas Windows:

- **`build.bat`**: Script para executar o projeto nos sistemas Windows;
- **`build.sh`**: Script para executar o projeto nos sistemas Linux;

Como o projeto deve rodar no Linux, conforme a especificação do trabalho, a seguir serão listadas as linhas de comando presente no script e seu significado para compilar e executar o projeto:

- **`rm *.class`**: Remove os arquivos *.class* presentes no diretório do projeto, ou seja, limpa e remove os arquivos compilados das classes java visando evitar erros durante a próxima execução;
- **`rm Lexer.*`**: Remove a classe e arquivos compilados antigos do analisador léxico;
- **`java -jar jflex-full-1.8.2.jar Lang.jflex`**: Inicializa o JFlex com o arquivo *Lang.flex* e a partir da estrutura do analisador léxico montada no projeto e presente no arquivo *Lang.flex*, o JFlex monta o analisador léxico em uma nova classe Java chamada de Lexer.
- **`javac -cp .  Lang.java`**: Compila a classe principal do projeto e todas as outras classes relacionadas.
- **`java Lang input.txt`**: Executa o projeto compilado passando como parâmetro o nome do arquivo de texto que o analisador léxico irá analisar e gerar os *tokens* correspondentes a linguagem Lang.

Portanto, para executar o script no Windows, basta apenas executar normalmente o arquivo **build.bat**. Contudo, para executar no Linux é necessário utilizar o terminal, se atentando de o terminal estar no diretório correto, e também usar o reconhecedor de arquivos *Shell Script* com o comando: **`bash build.sh`** ou **`sh build.sh`**.

2.3 Estrutura dos Arquivos

Dentro do arquivo enviado estão presentes todos os arquivos do projeto. Segue abaixo uma descrição de cada arquivo e sua funcionalidade no projeto:

- **`build.bat`**: Arquivo em batch. É um script utilizado para limpar a pasta do projeto, compilar, construir e executar o projeto do analisador léxico em máquinas com o sistema operacional Windows. Vale ressaltar que é utilizado o arquivo **input.txt** como arquivo de entrada para ser verificado pelo analisador léxico;

- **`build.sh`**: Arquivo em shell script. É utilizado para limpar a pasta do projeto, compilar, construir e executar o projeto do analisador léxico em máquinas com o sistema operacional Linux. Vale ressaltar que é utilizado o arquivo **input.txt** como arquivo de entrada para ser verificado pelo analisador léxico;

- **`input.txt`**: Arquivo de texto que contém códigos e expressões que serão verificadas pelo analisador léxico, ou seja, se o código passado pertence a estrutura da linguagem Lang. Este arquivo pode ser editado e modificado para testar como a ferramenta irá se comportar;

- **`jflex-full-1.8.2.jar`**: Arquivo executável em Java da ferramenta JFlex. Através dele será gerada a classe base(*Lexer.java*) do analisador léxico por meio do arquivo *Lang.flex* que apresenta uma estrutura que irá comportar a linguagem Lang.

- **`Lang.java`**: Classe auxiliar do projeto. Vincula o analisador léxico com o arquivo *input.txt* que será analisado e retornado ao usuário os tokens formados, e caso o arquivo não estiver no padrão da linguagem Lang, com símbolos não identificados, será retornado um erro que houve algum caractere inválido ou inconsistência no arquivo passado.

- **`Lang.jflex`**: Arquivo com as especificações da estrutura léxica da linguagem Lang. É utilizado pelo JFlex para gerar o analisador léxico da linguagem.

- **`Lexer.java`**: Classe Java gerada automaticamente pelo JFlex. Representa a estrutura principal do analisador léxico com base nas especificações passadas através do arquivo *Lang.jflex*.

- **`Token.java`**: Classe Java que representa a estrutura de um Token da linguagem. Através dela, o analisador léxico verifica o tipo de Token definido, linha, coluna e o lexema que mostra qual expressão que o representa.

- **`TOKEN_TYPE.java`**: Arquivo que apresenta uma enumeração para todos os Tokens que a linguagem irá aceitar, esta enumeração é utilizada na definição de um Token e na classe *Lexer* do analisador léxico, que ao coincidir com uma expressão, retorna o código do Token correspondente.

2.4 Expressões Regulares

Com o uso do JFlex, foi necessário adicionar algumas expressões regulares em macros para mapear o comportamento de palavras reservadas, símbolos reservados e ao final retornando o código correspondente ao seu Token. Por padrão, o JFlex já apresenta algumas expressões regulares para facilitar:

- **`[:digit:]`** ⇒ Agrupa os algarismos numéricos de zero a nove, sendo a mesma ideia expressa por **`[0-9]`**;
- **`[:uppercase:]`** ⇒ Envolve todas as letras maiúsculas, também pode ser representado por **`[A-Z]`**;
- **`[:lowercase:]`** ⇒ Cobre todas as letras minúsculas, também pode ser representado por **`[a-z]`**;
- **`[:letter:]`** ⇒ Engloba todas as letras maiúsculas e minúsculas, sendo a mesma ideia expressa por **`[a-zA-Z]`**;

Abaixo estão descritos os macros e suas respectivas expressões regulares:

- **`EOL = "\r|\n|\r\n"`** ⇒ Identifica o final de linha. Cobre todas as condições de final de linha independente do editor de texto ou sistema operacional;
- **`WS = "{EOL} | [\ \t] | [ \t\f]"`** ⇒ Identifica espaços em branco, englobando tabulações e final de linha;
- **`lineCmt = "−−".* {EOL}`** ⇒ Assimila os comentários de linha única para descarte nas etapas posteriores do compilador. De "−−" e qualquer outro símbolo até o final de linha;
- **`numberInteger = "[:digit:]+"`** ⇒ Identifica os números inteiros, com a obrigatoriedade de conter 1 digito no mínimo;
- **`numberFloat = ([:digit:]* "."([:digit:] [:digit:]*))`** ⇒ Identifica os números reais(float), podendo ser representados como: 123.23, 1.0, $.12345$;
- **`letter = "[:letter:]"`** ⇒ Simplifica a notação do macro que captura todas as letras, sendo elas maiúsculas ou minúsculas;
- **`characterLitteral = "'"("\\'"|"\\n"|"\\t"|"\\b"|"\\r"|"\\\\"| [^\n \r])"'"`** ⇒ Identifica os caracteres comuns e os especiais que apresentam mais de um simbolo;

- **id** = **[:lowercase:]({letter}|"_"|[:digit:])*** ⇒ Reconhece os identificadores, neste caso sempre começarão com letras minúsculas e posteriormente podem variar entre números, letras minúsculas, letras maiúsculas e "_";

- **nameType** = **[:uppercase:]({letter}|"_"|[:digit:])*** ⇒ Apesar de o nome de tipo também ser um identificador, seu formato é diferente, neste caso, se inicia obrigatoriamente com letras maiúsculas e depois segue mesmo padrão dos identificadores;

Além disso, ainda foram mapeadas as palavras e simbolos reservados pela linguagem. Para as palavras, os próprios caracteres já a tornam um filtro, e símbolos, apenas o símbolo já é suficiente para que o analisador léxico consiga identificar. No caso de comentários com mais de uma linha, optou-se por identificar a expressão "{-"e tratar em um estado separado se encerrando ao encontrar um "−}".

2.5 Decisões de Projeto

- 01: Para obter um identificador foi definido que a primeira letra deve ser minúscula, dado que foi especificado no enunciado do trabalho que apenas seria uma letra e portanto, se fosse maiúscula poderia ocorrer um problema de coincidir com um **NAME_TYPE** de modo que ambos começariam com uma letra maiúscula. Portanto, como decisão de projeto, foi separado os identificadores do nomes de tipo.

- 02: No tratamento dos tokens foi decidido tratar palavras reservas e símbolos de maneira separada, ou seja, tratando caso a caso de modo que cada mapeamento de token tenha um índice correspondente associado a enumeração **TOKEN_TYPE**.

- 03: Para não coincidir nomes de palavras reservadas com o seu literal, foi decidido utilizar o prefixo **VALUE** antes do nome do token.

- 04: Durante o tratamento no estado **MULTI_LINE_COMMENT** dos comentários de múltiplas linhas, notou-se que quando era utilizado '−' ou '}' dentro do comentário, ocorria o erro de encontrar um caractere ilegal **<−>** ou **<}>**. Este erro ocorre pois ele só sairá do estado se encontrar **<−}>**, ou seja, os dois símbolos, e quando encontrava somente um deles, não havia um tratamento, logo gerava um erro. Para solucionar este problema, foi adicionado dois casos especiais de quando encontrar isoladamente qualquer um destes símbolos, não faz nada e continua no comentário, assim, estes símbolos agora podem ser utilizados no meio dos comentários de múltiplas linhas.

- 05: Na condição para obter os caracteres literais, no enunciado foi solicitado ainda receber os símbolos especiais (quebra de linha, tabulação, ...), ou seja, símbolos que

apresentam uma barra em seu início. Para tratar esta condição, primeiramente foi feito um OR (OU) lógico entre todos os símbolos especiais, e depois mais um OR (OU) lógico para qualquer captar outro símbolo. Assim, pega-se os símbolos especiais e qualquer outro tipo de caractere, contudo, é necessário fazer esta verificação na ordem dita pois se invertida, a '\' será retornada já como um caractere sem verificar a condição dos especiais, portanto, a condição deles devem ser analisadas primeiro para depois observar os símbolos individuais.

Destaca-se ainda que caracteres que o delimitador ' ' começa em uma linha e termina em outra linha e caracteres vazios(''), não serão considerados válidos.