Implementação de Analisador Léxico para Linguagem *Lang*

Diego Paiva - 201565516C
Thaynara Ferreira - 201565254C

Outubro, 2020

Resumo

O presente documento tem como objetivo esclarecer as principais decisões de projeto tomadas na implementação do analisador léxico para a linguagem fictícia *Lang*, referente ao trabalho prático da disciplina *DCC045 - Teoria dos Compiladores*.

1 Contextualização do Problema

Um analisador léxico constitui a primeira etapa do projeto de um compilador. Seu funcionamento se dá a partir da leitura da sequência de caracteres do programa fonte, e, após isso, os agrupa de modo a formar *tokens*, que são unidades lexicais da linguagem. Cada *token* é constituído por um lexema e um tipo, que é um símbolo abstrato que é utilizado durante a etapa de análise sintática [1].

O objetivo deste trabalho é apresentar uma implementação de analisador léxico para a linguagem *Lang*, bem como o projeto do autômato utilizado para os reconhecimentos dos *tokens* possíveis.

2 Estrutura léxica de *Lang*

A linguagem fictícia *Lang* é uma linguagem com propósitos meramente educacionais. Ela possui construções simples como tipos de dados, declarações, funções, comandos e expressões. A saber, na implementação deste trabalho, *tokens* em *Lang* seguem um dos tipos a seguir.

- `IDENTIFIER`: Sequência de letras, dígitos e sobrescritos (underscore) que, obrigatoriamente, começa com uma letra. Ex: `var`, `var1`, `fun10`, etc.
- `INT`: Sequência de um ou mais dígitos. Ex: `0`, `1`, `10`, `128`, etc.
- `FLOAT`: Sequência de zero ou mais dígitos, seguido por um ponto e uma sequência de um ou mais dígitos. Ex: `3.141526535`, `1.0`, `.12345`, etc.
- `CHAR`: Um único caractere delimitado por aspas simples. Os caracteres especiais quebra-de-linha, tabulação, *backspace* e *carriage return* são definidos usando os caracteres de escape `\n`, `\t`, `\b` e `\r`, respectivamente. Para especificar um caractere \ , é usado `\\` e para a aspas simples o `\'`. Ex: `'a'`, `'\n'`, `'\t'`, `'\\'` e `'\''`, etc.
- `BOOL`: Literal lógico. Ex: `true` e `false`.
- `NULL`: Literal nulo `null`.
- `TYPE_NAME`: Identificador cuja primeira letra é maiúscula. Ex: `Racional`, `Point`, etc.
- `TYPE_INT`: Palavra reservada pela linguagem para nomear o tipo `Int`.
- `TYPE_FLOAT`: Palavra reservada pela linguagem para nomear o tipo `Float`.
- `TYPE_BOOL`: Palavra reservada pela linguagem para nomear o tipo `Bool`.

- `TYPE_CHAR`: Palavra reservada pela linguagem para nomear o tipo `Char`.
- `IF`: Palavra reservada pela linguagem para nomear o comando de seleção `if`.
- `ELSE`: Palavra reservada pela linguagem para nomear o comando de seleção `else`.
- `ITERATE`: Palavra reservada pela linguagem para nomear o comando de iteração `iterate`.
- `READ`: Palavra reservada pela linguagem para nomear o comando de entrada `read`.
- `PRINT`: Palavra reservada pela linguagem para nomear o comando de saída `print`.
- `RETURN`: Palavra reservada pela linguagem para nomear o comando de retorno `return`.
- `DATA`: Palavra reservada pela linguagem para nomear o comando de definição de tipos de dados `data`.
- `SEPARATOR`: Separadores da linguagem. Ex: `(`, `)`, `[`, `]`, {, }, `:`, `::`, `;`, `.` e `,`.
- `OPERATOR`: Operadores da linguagem. Ex: `=`, `<`, `==`, `!=`, `+`, `-`, `*`, `/`, `%`, `&&` e `!`.

Além dos *tokens* enumerados, há também comentários na linguagem *Lang*, podendo estes ser de dois tipos: comentário de uma linha e de múltiplas linhas. O comentário de uma linha começa com o lexema `--` e se estende até a próxima quebra de linha, enquanto que o comentário de múltiplas linhas começa com {`-` e se estende até os caracteres `-`}. Comentários são ignorados pelo analisador léxico. Um exemplo de código na linguagem *Lang* pode ser visto na Seção 6.1.

3 Autômato para Reconhecimentos de *Tokens*

Na etapa de análise léxica costuma-se utilizar autômatos e/ou expressões regulares para reconhecimento de *tokens* do programa de entrada codificado na linguagem alvo. A figura 1 mostra o autômato finito determinístico (AFD) que foi projetado para reconhecer os *tokens* descritos anteriormente.

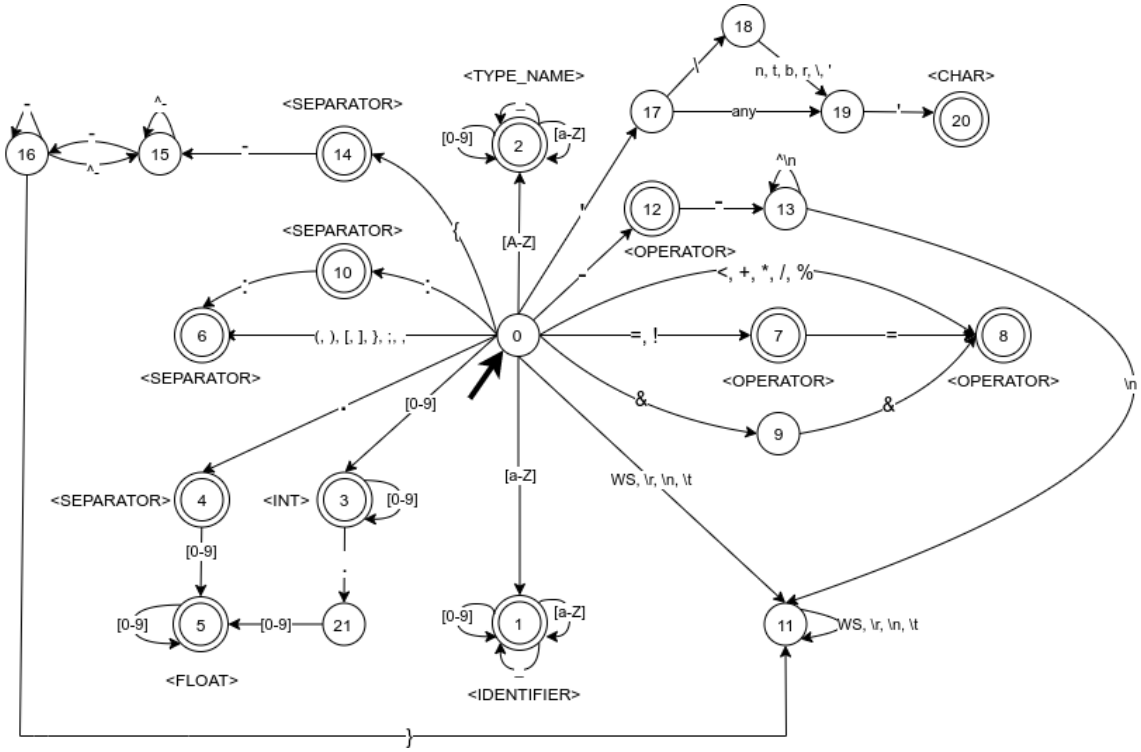

Figura 1: AFD reconhecedor de *tokens* da linguagem *Lang*.

Este AFD possui um total de 22 estados, nos quais 12 são estados finais. Para uma dada sequência de caracteres, o processamento é iniciado a partir do estado 0, e o estado resultante do processamento da cadeia estabelecerá qual o tipo de *token* ela determina. Se o estado resultante não for final, então um *token* inválido foi identificado. É importante observar que caracteres de espaçamento (WS - espaço em branco, `\r`, `\n` e `\t`) são ignorados, bem como qualquer caractere que se encontrar após o lexema `--` ou entre os lexemas `{-` e `-}`. O estado 11 desempenha o papel de estado *skip*, ou seja, responsável por ignorar o processamento dos espaçamentos.

4 Modelagem e Estruturas de Dados

Para melhor estruturação do analisador léxico, optou-se por criar as seguintes classes: `Dfa.java`, `State.java`, `Token.java`, `LangScanner.java` e `App.java`. Cada uma dessas classes será detalhada a seguir.

4.1 Dfa

Primeiramente foi codificado o AFD na classe `Dfa.java`. Esta classe constrói um objeto `Dfa` a partir de um arquivo passado como parâmetro, que contêm todos os estados e suas transições. Este arquivo deve estar especificado no seguinte formato:

```
1  NºEstados NºTransições NºEstadosFinais IdEstadoSkip
2
3  IdEstadoOrigem X IdEstadoDestino
4  ...
5
6  IdEstadoFinal TipoToken
7  ...
```

A linha 1 é o cabeçalho do arquivo, contendo os metadados. A partir da linha 3 até a próxima linha em branco, são mapeadas todas as transições do autômato, onde `X` é um caractere ou um dos seguintes termos: `lowerLetter, upperLetter, digit, whitespace, any, ^\n` ou `^-`. Os termos `^\n e ^-`, denotam, respectivamente, quaisquer caracteres que não sejam `\n` e `-`. Finalmente, no terceiro e último bloco, são definidos todos os estados finais e o tipo de *token* que eles representam. Internamente, `Dfa.java` utiliza uma estrutura `HashMap` para mapear as transições de estado.

4.2 State

A classe `State` representa um estado de um autômato, e possui apenas três atributos: um inteiro denotando o seu *id*, o tipo de *token* que o estado define e uma *flag* que indica se é o estado de *skip*. Além disso, a classe conta com métodos para verificar se o estado é final e/ou de erro.

4.3 Token

A classe `Token` possui apenas o valor do lexema e o seu tipo, que é umas da opções definidas no enumerador interno `Type`.

4.4 LangScanner

A classe `LangScanner` é o analisador léxico em si, e contém o algoritmo principal para obtenção dos *tokens*. Sempre que é feita uma chamada ao método `nextToken()` é retornado o próximo *token* do programa de entrada. Além disso, `LangScanner` contém a tabela de palavras reservadas, que constitui um mecanismo de desambiguação e que é abordado na Seção 5.

4.5 App

Esta classe contém o método `main()`, que é responsável por imprimir, iterativamente, todos os *tokens* do programa de entrada passado como argumento pela linha de comando.

5 Mecanismo de Desambiguação

Em determinados casos, acontece de alguns tokens serem reconhecidos como sendo do tipo `IDENTIFIER` ou `TYPE_NAME`, quando na verdade são palavras reservadas da linguagem para representar comandos ou nomes de tipo de variáveis. Por exemplo, há o lexema `Int`, que é a palavra reservada para representar o tipo de variável inteiro, e é reconhecida pelo autômato como `TYPE_NAME` quando, na verdade, deveria ser do tipo `TYPE_INT`. Para o tratamento desses casos, foi feito na classe `LangScanner` um mecanismo para desambiguação.

Esse mecanismo consiste em utilizar uma estrutura `HashMap` para mapear quais são as palavras reservadas pela linguagem e a qual tipo de *token* elas pertencem. Com isso, quando um *token* é encontrado, testamos se o seu lexema é algumas dessas palavras e, caso seja, retornamos o tipo de *token* que a define.

6 *Building*

Para realizar o *build* da aplicação, recomenda-se a utilização da ferramenta de gerenciamento **Maven** [2].

Com o Maven instalado, navegue até o diretório do projeto que contém o arquivo `pom.xml`. Após isso, rode o comando:

```
mvn package
```

Este comando irá compilar o código e o empacotar em um formato distribuível `jar` dentro da pasta `target`. Para executar o programa principal, rode o seguinte comando:

```
java -cp target/lang-compiler-1.0-SNAPSHOT.jar app.App test_prog.txt
```

Onde `test_prog.txt` é o arquivo texto que contém o programa em *Lang* a ser analisado.

6.1 Exemplo de Execução

Considere o seguinte código em *Lang*:

```
main() {
    print fat(10);
}

fat(num :: Int) : Int {
    {-
    multiline
    comment
    -}
    if (num < 1)
        return 1;
    else
        return num * fat(num - 1); -- One-line comment
}
```

Após execução conforme mostrado na Seção 6, a saída exibirá todos o *tokens* e seus respectivos tipos:

```
<main, IDENTIFIER>
<(, SEPARATOR>
<), SEPARATOR>
<{, SEPARATOR>
<print, PRINT>
```

```
<fat, IDENTIFIER>
<(, SEPARATOR>
<10, INT>
<), SEPARATOR>
<;, SEPARATOR>
<}, SEPARATOR>
<fat, IDENTIFIER>
<(, SEPARATOR>
<num, IDENTIFIER>
<::, SEPARATOR>
<Int, TYPE_INT>
<), SEPARATOR>
<:, SEPARATOR>
<Int, TYPE_INT>
<{, SEPARATOR>
<if, IF>
<(, SEPARATOR>
<num, IDENTIFIER>
<<, OPERATOR>
<1, INT>
<), SEPARATOR>
<return, RETURN>
<1, INT>
<;, SEPARATOR>
<else, ELSE>
<return, RETURN>
<num, IDENTIFIER>
<*, OPERATOR>
<fat, IDENTIFIER>
<(, SEPARATOR>
<num, IDENTIFIER>
<-, OPERATOR>
<1, INT>
<), SEPARATOR>
<;, SEPARATOR>
<}, SEPARATOR>
```

Referências

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman, "Compiladores: princípios, técnicas e ferramentas, 2ª edição," 2007.

[2] "Apache maven project." `http://maven.apache.org/`. Accesso em: 04/10/2020.