UNIVERSIDADE FEDERAL DE SANTA CATARINA - UFSC
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO
INE5426 - Construção de compiladores
Prof. Alvaro Junio Pereira Franco

Compilador da linguagem CC-2021-2: Analisador Léxico

Leonardo Schlüter Leite - 13200658

Florianópolis, 25 de novembro 2021

1. Identificação dos Tokens e Definições Regulares

Para identificar todos os tokens da linguagem foram produzidos todos os símbolos terminais possíveis da linguagem alvo (CC-2021-2). A primeira coluna é o nome que dei para o Token; a segunda é a expressão regular deste token; a terceira é a definição regular do token ; a quarta é o nome da regra no arquivo ./compiler/src/main/antlr4/CC_2021_2.g4

Tokens	**ER**	**Def. Reg.**	**ANTLR**
DEFINITION	def	DEFINITION -> def	DEF
IDENT	([a-z][a-zA-Z0-9])	LetterOrDigit -> [a-zA-Z0-9] Letter -> [a-zA-Z] IDENT -> Letter LetterOrDigit*	Identifier
OPEN_PAR	(	OPEN_PAR -> (	OPEN_PAR
CLOSE_PAR	)	CLOSE_PAR ->)	CLOSE_PAR
OPEN_CHAVE	{	OPEN_CHAVE -> {	OPEN_CHAVE
CLOSE_CHAVE	}	CLOSE_CHAVE -> }	CLOSE_CHAVE
INT	int	INT -> int	INT
FLOAT	float	FLOAT -> float	FLOAT
STRING	string	STRING -> string	STRING
SEMI_COLON	;	SEMI_COLON -> ;	SEMI_COLON
BREAK	break	BREAK -> break	BREAK
OPEN_COL	[	OPEN_COL -> [	OPEN_COL
CLOSE_COL	]	CLOSE_COL ->]	CLOSE_COL
ATR	\=	ATR -> =	ASSIGN
INT_CONSTANT	([0-9])	NonZeroDigit -> [1-9] Digit -> '0' | NonZeroDigit Digits -> Digit* Numeral -> '0' | NonZeroDigit (Digits?) INT_CONSTANT -> Numeral*	IntegerConstant
FLOAT_CONSTANT	([0-9](.[0-9])?)	NonZeroDigit -> [1-9] Digit -> '0' | NonZeroDigit Digits -> Digit* FLOAT_CONSTANT -> Digits '.' Digits?	FloatConstant
STRING_CONSTANT	[a-zA-Z\u00C0-\u00FF]+	StringCharacter -> [a-zA-Z\u00C0-\u00FF]+ StringCharacters -> StringCharacter+ STRING_CONSTANT -> "" StringCharacters? ""	StringConstant
COLON	,	COLON -> ,	COLON

PRINT	print	PRINT -> print	PRINT
READ	read	READ -> read	READ
RETURN	return	RETURN -> return	RETURN
IF	if	IF -> if	IF
ELSE	else	ELSE -> else	ELSE
FOR	for	FOR -> for	FOR
NEW	new	NEW -> new	NEW
LESS	<	LESS -> <	LESS
GREATER	>	GREATER -> >	GREATER
LESSEQUAL	<=	LESSEQUAL -> <=	LESS_EQUAL
GREATEREQUAL	>=	GREATEREQUAL -> >=	GREATER_EQUAL
EQUAL	\==	EQUAL -> ==	EQUAL
DIFFERENT	!=	DIFFERENT -> !=	DIFFERENT
PLUS	\+	PLUS -> +	PLUS
SUBTR	\-	SUBTR -> -	SUBTR
NULL	null	NULL -> null	NULL
MULT	*	MULT -> *	MULT
DIV	/	DIV -> /	DIV
REMINDER	%	REMINDER -> %	REMINDER

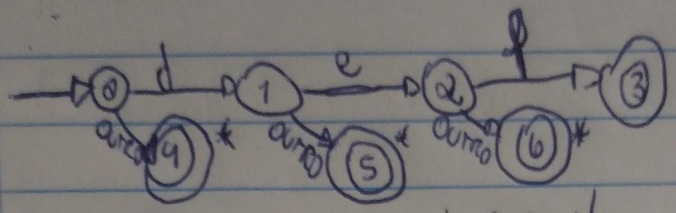
Além disso, tive que adicionar mais um tipo de token devido a forma que o ANTLR-4 funciona. Depois no capítulo de utilização do ANTLR-4 será especificado melhor o motivo das alterações.

2. Diagramas de transição

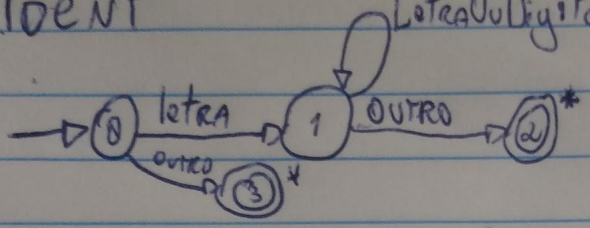
Para fazer os diagramas utilizei de começo papel e caneta, mas depois percebi que seria mais fácil e rápido utilizar a ferramenta diagram do google. Segue as Imagens:

data

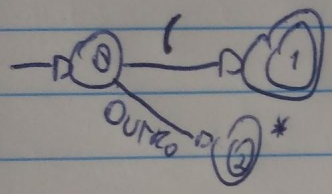
DEF



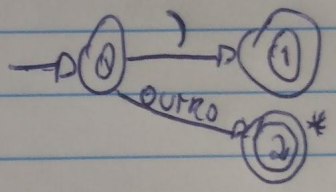
IDENT



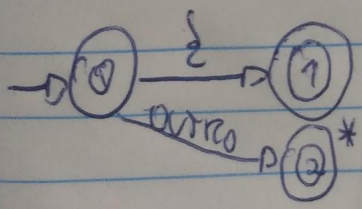
OPEN_PAR



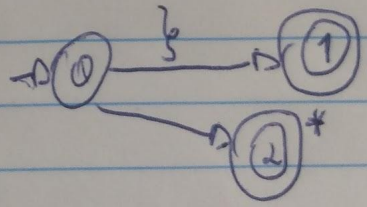
CLOSE_PAR



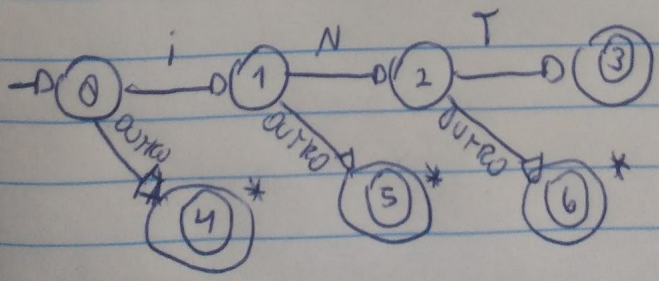
OPEN_CHAVE



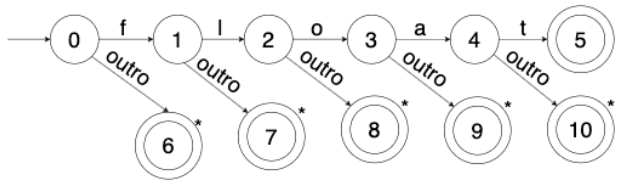
CLOSE_CHAVE



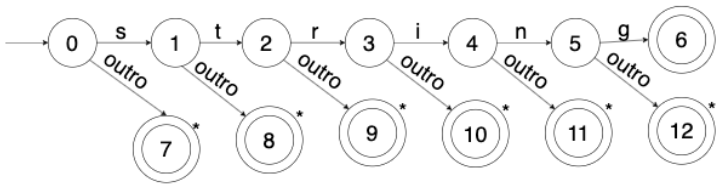
INT



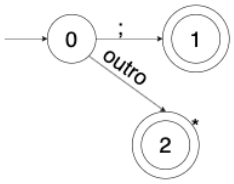
FLOAT



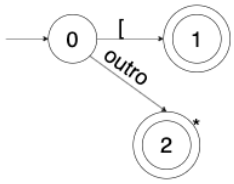
STRING



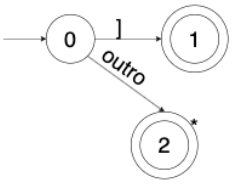
SEMICOLON



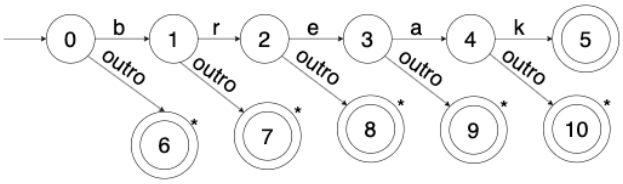
OPEN_COL



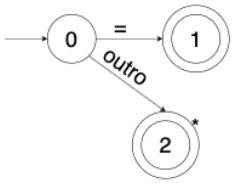
CLOSE_COL



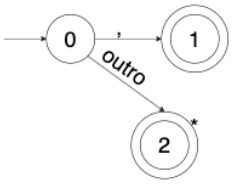
BREAK



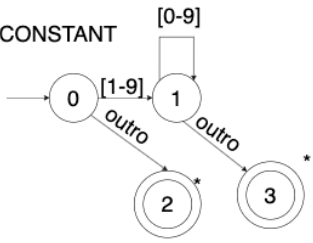
ATR



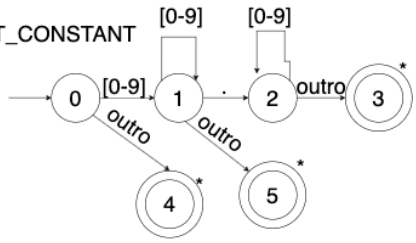
COLON



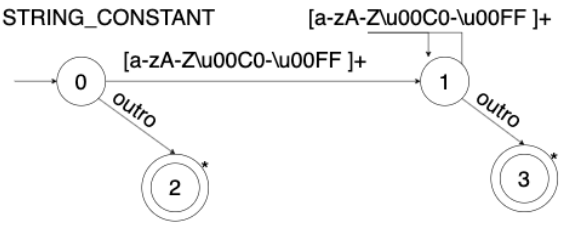
INT_CONSTANT



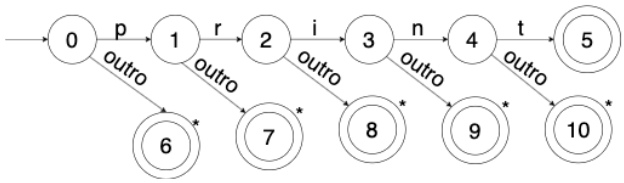
FLOAT_CONSTANT



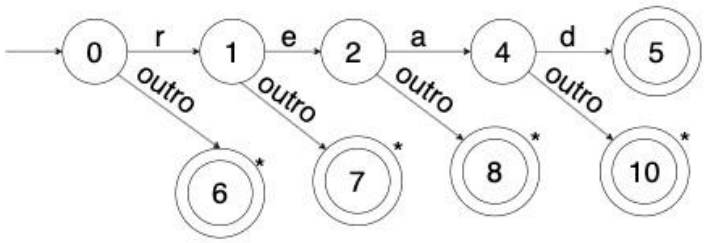
STRING_CONSTANT



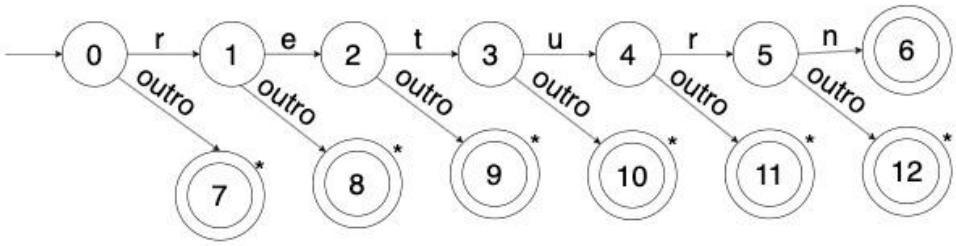
PRINT



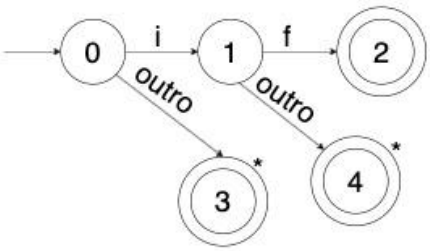
READ



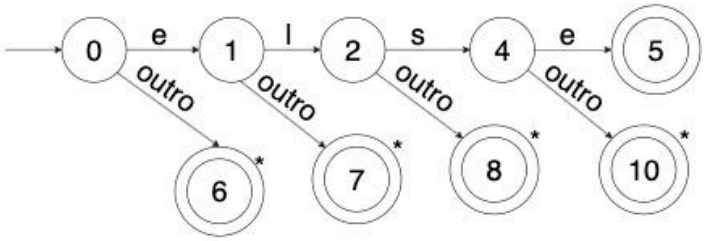
RETURN



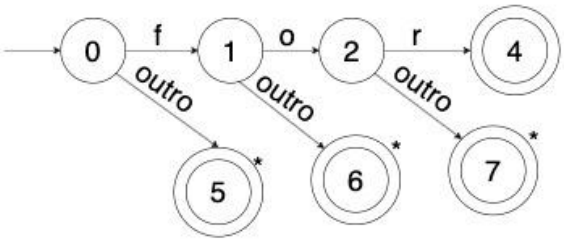
IF



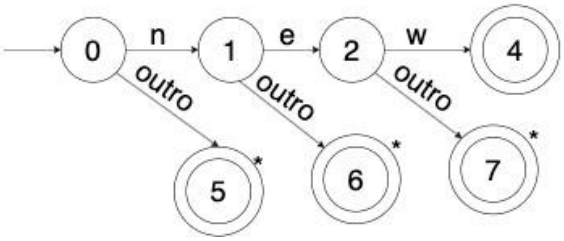
ELSE



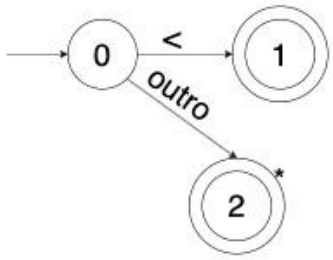
FOR



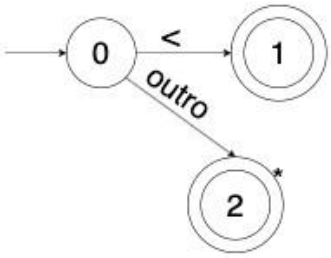
NEW



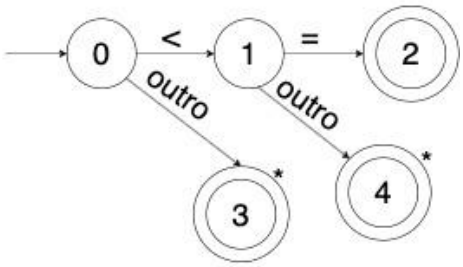
LESS



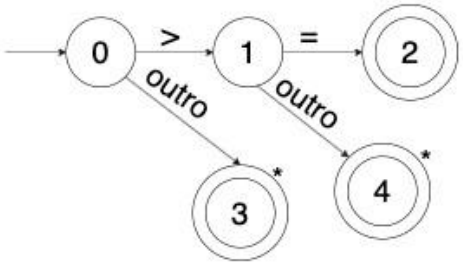
GREATER



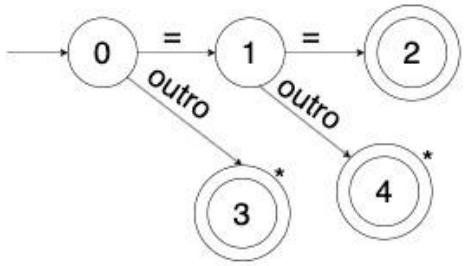
LESSEQUAL



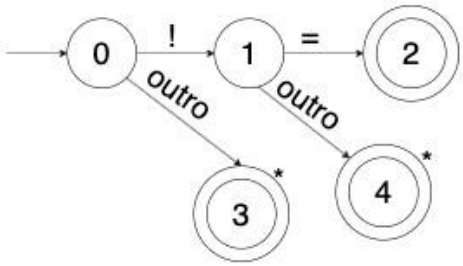
GREATER_EQUAL



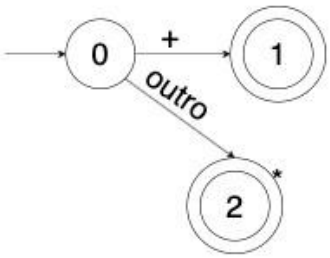
EQUAL



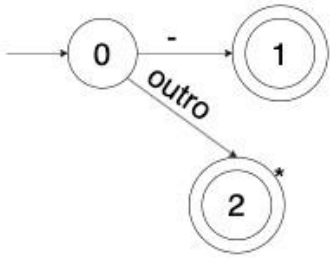
DIFFERENT



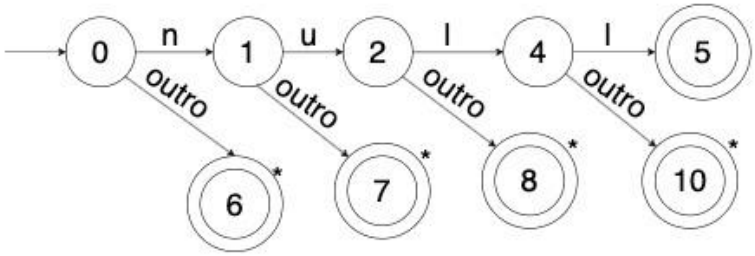
PLUS



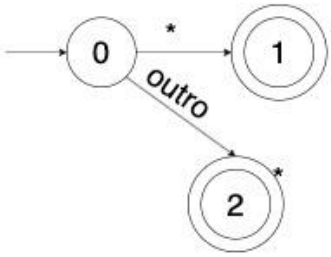
SUBTR



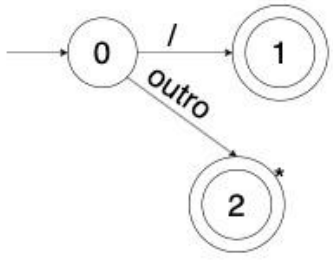
NULL



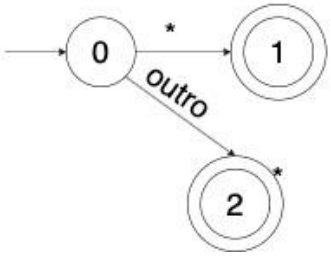
MULT



DIV



REMINDER



3. Tabela de Símbolos

Como estou utilizando o ANTLR-4 a tabela de símbolos é independente. Até onde entendi o próprio framework se resolve com isso, mas analisando o código e pensando que a tabela de símbolos guarda informações sobre os tokens importantes para os passos, cheguei a uma classes do ANTLR-4 que acredito responderem a pergunta.

A classe CommonToken contém os seguintes parâmetros: type, line, charPositionInLine, channel (é uma categorização do token que instrui como o compilador deve tratar este token), source, start, stop.

4. Utilização do ANTLR-4

Como utilizei o ANTLR-4, precisei descrever a gramática num arquivo .g4 que está localizado em `src/main/antlr4/CC_2021_2.g4`. Tive que descrever a gramática alvo como um todo, não apenas os símbolos terminais, se não o ANTLR-4 não consegue gerar corretamente os arquivos necessários. De qualquer forma, vale ressaltar que o ANTLR entende dois tipos de regras: regras léxicas e regras sintáticas. Regras sintáticas e léxicas seguem o mesmo padrão na definição da regra em si, mas o nome da regra que define como o gerador irá tratar aquela regra. Se começar com o Letra Maiúscula, será uma regra léxica. Defini uma convenção vendo exemplos: para regras léxicas que dependem de outra regra léxica, utilizar camelCase. Para regras léxicas puras, utilizar CAPS. Regras sintáticas começam com letras minúsculas.

Fiz uma gramática exemplo apenas com uma regra léxica que aceita a string "ab":

AB: 'ab'

Se eu aplicar a ferramenta de geração em cima dessa gramática, serão gerados apenas 3 arquivos (nome da gramática é example): exampleLexer.java, exampleLexer.interp, exampleLexer.tokens, com apenas esses 3 arquivos já é possível implementar um analisador léxico utilizando os métodos nextToken() ou getAllTokens() do exampleLexer. Porém o gerador reclama que a gramática é inválida e até onde consegui entender, é porque falta regras sintáticas. Então já deixei definido a gramática CC-2021-2 como um todo. Por exemplo, a seguinte gramática já é livre de erros ao gerar código:

```
grammar example;
program : AB+;
AB: 'ab' ;
```

Esta gramática gera todos os arquivos: exampleLexer.java, exampleLexer.interp, exampleLexer.tokens, example.inter, example.tokens, exampleBaseListener.java, exampleBaseVisitor.java, exampleListener.java, exampleParser.java, exampleVisitor. Porém

não estudei a parte de análise sintática do ANTLR-4 pois não é o foco deste trabalho, quando formos fazer o analisador sintático trarei uma explicação para cada um desses arquivos. Como por hora preciso apenas do analisador léxico, foquei apenas nele.

Após, utilizei o plugin do ANTLR-4 do maven para gerar os arquivos que estão localizados em `src/main/java/org/schluter/compiler/gen`. Detalhe aqui é que tive que alterar o arquivo `CC_2021_2Lexer.java` para modificar a ação tomada pelo analizador ao encontrar com o token do tipo Invalid. Este tipo de token é responsável por pegar todos os caracteres que não se encaixam em nenhum outro tipo de token. No código explico a modificação, mas para deixar explícito, eu criei uma lista de erros, e quaisquer itens que sejam reconhecidos como tipo Invalid são adicionados à lista de erros.

O plugin maven do ANTLR-4 tem um warning:

[WARNING] Failed to retrieve plugin descriptor for org.antlr:antlr4-runtime:4.9.2: Failed to parse plugin descriptor for org.antlr:antlr4-runtime:4.9.2 (/Users/x266483/.m2/repository/org/antlr/antlr4-runtime/4.9.2/antlr4-runtime-4.9.2.jar): No plugin descriptor found at META-INF/maven/plugin.xml

Este warning deve ser resolvido por quem implementou a biblioteca, provendo um descriptor.

5. Utilizando o analisador léxico

Utilizar o CC_2021_2Lexer foi relativamente tranquilo, pois estende a classe Lexer, que por sua vez tem uma interface intuitiva, tendo um método chamado nextToken() e outro chamado getAllTokens(). Criei uma classe App que contém o main e recebe uma string de argumento como o caminho do arquivo a passar pelo analisador léxico. Então apenas utilizo o getAllTokens, verifico se tem algum erro e sigo o fluxo conforme requisitado na descrição do trabalho.

Referências

https://github.com/antlr/antlr4

https://www.antlr.org/api/Java/org/antlr/v4/runtime/package-summary.html