Universidade Federal de Juiz de Fora

Departamento de Ciência da Computação

Teoria dos Compiladores

Etapa 01: Analisador Léxico

Daniel Augusto Machado Baeta - 201965122C

Thiago do Vale Cabral - 201965220AC

Relatório do trabalho prático apresentado ao prof. Leonardo Vieira dos Santos Reis como requisito de avaliação da Disciplina DCC045 - Teoria dos Compiladores

Juiz de Fora

Maio de 2022

Relatório Técnico

Daniel Augusto Machado Baeta [1]
Thiago do Vale Cabral [1]

1 Introdução

O presente trabalho tem como temática central descrever uma etapa de análise léxica em um processo de compilação de uma linguagem de programação arbitrária *lang*. Para tal, propõe-se um algoritmo para a análise léxica utilizando a linguagem de programação Java e uma classe geradora de autômatos, JFlex 1.8.2 (2020). Os resultadores obtidos pelos testes, assim como a estratégia de resolução utilizada, são alvo de discussão neste texto.

A realização desse experimento se deu como forma de consolidar e aplicar os conceitos teóricos vistos durante a disciplina DCC045 - Teoria dos Compiladores durante o primeiro semestre de 2022. Além de expor os discentes a problemas que requerem abstração e a busca por soluções otimizadas e eficientes, tais conceitos abordados são cruciais para o entendimento da funcionalidade de um compilador e boas práticas de programação.

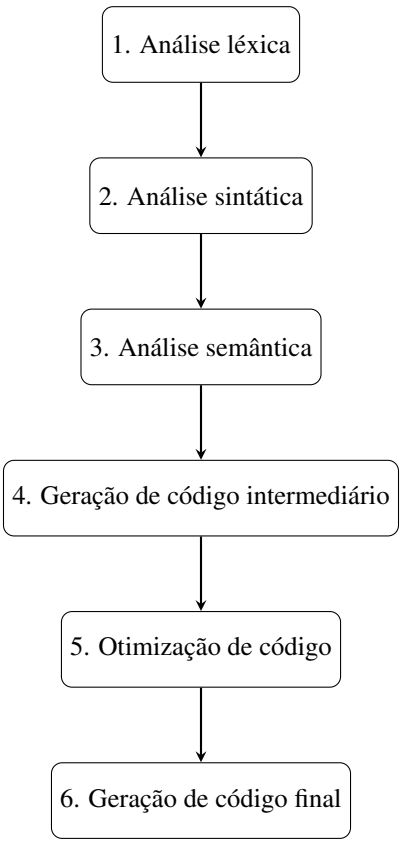
Diante disso, na seção 2 descreve-se o contexto do problema; na seção 3 mostra-se a abordagem e estratégia utilizada, enquanto a seção 4 apresenta o experimento realizado juntamente com análise dos resultados obtidos. Por último, a seção 5 apresenta as conclusões do trabalho.

[1]Departamento de Ciência da Computação – Instituto de Ciências Exatas
Universidade Federal de Juiz de Fora (UFJF)
Rua José Kelmer, s/n, Campus Universitário, Bairro São Pedro
CEP: 36036-900 – Juiz de Fora/MG – Brasil
{daniel.baeta, thiago.cabral}@ice.ufjf.br

2 Descrição do problema

De forma objetiva, um compilador é um programa que lê um programa escrito em uma linguagem *fonte* e o traduz em um programa equivalente em uma outra linguagem *alvo*. Além disso, compõe-se como uma etapa crucial desse processo de tradução a interação com redator do programador, informando-o de eventuais erros no programa *fonte*.

Compõe-se como etapas da compilação de uma linguagem *fonte*: (1) Análise léxica, (2) Análise sintática, (3) Análise semântica, (4) Geração de código intermediário, (5) Otimização de código e (6) Geração de código final:



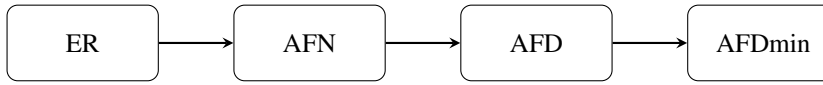
Para o presente experimento, apenas a etapa de análise léxica será discutida e desenvolvida.

2.1 Análise Léxica

Etapa inicial de um processo de compilação, a análise léxica também é conhecida como *scanner*. Sua função é ler o programa fonte, caractere a caractere, realizar o agrupamento de caracteres lidos em lexemas e produzir uma sequência de símbolos léxicos. Estes símbolos produzidos, por sua vez, são denominados *tokens*.

É importante reforçar o que objetivo desta etapa não é analisar se a sequência de *tokens* emitidos faz sentido - semântico ou sintático - mas identificar todos os *tokens* válidos seguindo uma lista de prioridades. Caso um lexema identificado não seja compatível com nenhum *token* mapeado, o processo de compilação retorna um erro léxico para o autor do programa fonte.

Cada *token* possui uma regra de identificação, definida por uma Expressão Regular (ER). Esta expressão regular é posteriormente convertida em um Autômato Finito Não Determinístico (AFN), que por sua vez é traduzido para um Autômato Finito Determinístico (AFD) correspondente. Por fim, este autômato pode ser simplificado para a uma forma mínima equivalente (AFDmin).



A identificação de *tokens* válidos é feita por intermédio deste autômato mínimo resultante, que pode ser representado por um algoritmo de *scanner* utilizando da implementação direta ou até mesmo através da representação em uma estrutura de dados de matriz.

3 Abordagens para o problema

Frente ao desafio de construir um compilador para a linguagem de programação *lang*, buscou-se elaborar um programa para identificar a sequência de *tokens* presentes nos arquivos de teste. Para tal, foi elaborada uma tabela com os *tokens* válidos seguindo a especificação da linguagem. Os *tokens* foram mapeados em ordem de prioridade, sendo que a tabela 1 tem maior prioridade que a tabela 2, que por sua vez possui maior prioridade que a tabela 3. Além disso, os *tokens* foram organizados em ordem de prioridade em cada tabela, sendo o primeiro registro o de maior prioridade, e o último de menor prioridade.

Na tabela 1 abaixo, descreve-se todas as palavras reservadas pela linguagem fonte, e por isso é de extrema importância que essas palavras não sejam categorizadas como identificadores.

Tabela 1 – Mapeamento de palavras-chave

Código	Nome	Expressão Regular
DATA_KEYWORD	Comando data	[data]
INT_KEYWORD	Comando Int	[Int]
CHAR_KEYWORD	Comando Char	[Char]
FLOAT_KEYWORD	Comando Float	[Float]
BOOL_KEYWORD	Comando Bool	[Bool]
IF_KEYWORD	Comando if	[if]
ELSE_KEYWORD	Comando else	[else]
ITERATE_KEYWORD	Comando iterate	[iterate]
READ_KEYWORD	Comando read	[read]
PRINT_KEYWORD	Comando print	[print]
RETURN_KEYWORD	Comando return	[return]
NEW_KEYWORD	Comando new	[new]

Na tabela 2 abaixo, descreve-se literais em geral, juntamente com a definição de tipo abstrato de dados e identificador. Uma particularidade da linguagem *lang* é que pode-se definir que um *token* é um tipo abstrato de dado devido à regra de que todo identificador começa com letra minúscula, enquanto todo tipo é definido como maiúscula. Assim, tem-se um ganho em conseguir categorizar esses *tokens* antes da análise sintática.

Tabela 2 – Mapeamento de literais e identificadores

Código	Nome	Expressão Regular
TYPE	Tipo abstrato de dado	[A-Z][a-zA-Z_0-9]*
FLOAT_LITERAL	Literal de ponto flutuante	[:digit:] [:digit:]* ['.'] [:digit:]* | ['.'] [:digit:] [:digit:]*
CHAR_LITERAL	Literal de caractere	['']{qualquer caractere}['] | ['']{caractere especial}[']
INT_LITERAL	Literal inteiro	[:digit:] [:digit:]*
BOOL_LITERAL	Literal booleano	[true] | [false]
NULL_LITERAL	Literal nulo	[null]
IDENTIFIER	Identificador	[a-z][a-zA-Z_0-9]*

Por fim, define-se na tabela 3 todos os operadores e símbolos. Neste caso, a estratégia utilizada foi de categorizar ao máximo os *tokens*.

Tabela 3 – Mapeamento de operadores e outros símbolos

Código	Nome	Expressão Regular
EQUALITY	Operador de igualdade	[==]
INEQUALITY	Operador de desigualdade	[!=]
GREATER_EQUAL	Operador de maior ou igual	[>=]
LESS_EQUAL	Operador de menor ou igual	[<=]
GREATER_THAN	Operador de maior	[>]
LESS_THAN	Operador de menor	[<]
ASSIGNMENT	Operador de atribuição	[=]
ADDITION	Operador de adição	[+]
SUBTRACTION	Operador de subtração	[-]
MULTIPLICATION	Operador de multiplicação	[*]
DIVISION	Operador de divisão	[/]
MODULUS	Operador de módulo	[%]
AND	Operador lógico "e"	[&&]
OR	Operador lógico "ou"	[||]
NEGATION	Operador lógico "não"	[!]
OPEN_PARENTHESIS	Abre parênteses	[(]
CLOSE_PARENTHESIS	Fecha Parênteses	[)]
OPEN_BRACE	Abre Chaves	[{]
CLOSE_BRACE	Fecha Chaves	[}]
OPEN_BRACKET	Abre Colchetes	[[]
CLOSE_BRACKET	Fecha Colchetes	[]]
COMMA	Separador	[,]
SEMICOLON	Ponto e vírgula	[;]
TYPE_ASSIGMENT	Atribuição de Tipo	[::]
COLON	Dois Pontos	[:]
DOT	Ponto	[.]

Unificando as regras descritas acima, com auxílio da classe geradora JFlex 1.8.2 (2020), gerou-se uma classe Java responsável pela execução do AFD mínimo que identifica e instancia os *tokens* de acordo com o tipo, desconsiderando completamente comentários de linha e bloco, assim como os seus conteúdos.

4 Resultados

Apesar de ser uma linguagem extremamente enxuta, a mesma teve uma variedade significativa de 44 *tokens* distintos. Tendo em vista este resultado, foi necessário validar as respectivas definições com uma variedade significativa de testes. Validando a seguinte entrada abaixo:

```
f(x :: Int) : Int, Float{
   y = 2*x  + 1;
   return y, 1.5;
}


main(v :: Int[], f :: Float){
  z = 10;
  f(z)< x, w >;
}
```

Temos a seguinte saída:

```
ID:f
(
ID:x
::
INT
)
:
INT
,
FLOAT
{
```

```
ID:y
=
INT:2
*
ID:x
+
INT:1
;
RETURN
ID:y
,
FLOAT:1.5
;
}
ID:main
(
ID:v
::
INT
[
]
,
ID:f
::
FLOAT
)
{
```

```
ID:z
=
INT:10
;
ID:f
(
ID:z
)
<
ID:x
,
ID:w
>
;
}
Total de tokens lidos 53
```

Foi possível observar pela impressão acima que a identificação dos *tokens* foi bem sucedida. Entretanto, é interessante mencionar que há casos onde a análise léxica não retornará erro, como por exemplo o caso abaixo:

```
if (+variable) {
  print variable;
}
```

Neste caso, temos a seguinte saída:

```
IF
(
```

```
+
ID:variable
)
{
PRINT
ID:variable
;
}
```

Apesar do sinal de "+"ser claramente um erro, o código ainda sim é considerado correto no ponto de vista léxico. Esta validação deverá ser feita posteriormente!

Outro caso interessante identificado foi a interpretação de inteiros e decimais negativos. O exemplo abaixo:

```
variable = -0.1
```

Produz:

```
ID:variable
=
-
FLOAT:0.1
Total de tokens lidos 4
```

Neste caso, instintivamente esperava-se que o literal de ponto flutuante tivesse valor -0.1. Entretanto, do ponto de vista léxico esta saída ainda está correta, pois pode ser interpretada como uma operação 0 - 0.1 em uma análise posterior. Tendo em vista esta interpretação, foi conveniente manter as expressões regulares o mais objetivas e simples possível.

5 Conclusão

De modo geral, ao observarmos os resultados obtidos, podemos afirmar que a etapa de análise léxica alcançou os resultados desejados. Ademais, foi possível concluir que a definição de linguagem de *lang* tornou oportuna a maximização de categorizações de lexemas, o que permitiu identificações antecipadas de *tokens*, o que poderá simplificar consideravelmente as etapas subsequentes.

As regras definidas para cada *token*, entretanto, não garantem uma coerência de linguagem. A ordem em que os *token* são apresentados ainda precisa ser analisada.

Findando, espera-se realizar o aperfeiçoamento das técnicas e estratégias utilizadas, além de conhecer outras metodologias presentes na literatura. Com isso, será possível realizar trabalhos futuros com o fito de buscar propostas de soluções mais eficientes.