

Compiladores - Trabalho Prático 1

Analisadores léxicos no software Lex

Mateus Pinto da Silva¹

¹Ciência da Computação – Universidade Federal de Viçosa (UFV-caf)
– Florestal – MG – Brasil

(mateus.p.silva¹)@ufv.br

Resumo. *O Lex é uma ferramenta geradora de analisador léxico feito para facilitar o trabalho da criação de um compilador. Neste trabalho, mostro a implementação com testes de execução do analisador léxico pedido e de um montador simplificado do MIPS, além de comparar este último a uma versão de um montador feito sem nenhuma ferramenta parecida por mim para outra disciplina, citando as diferenças. Termino discutindo as considerações finais, mostrando o que foi aprendido durante a elaboração deste trabalho.*

1. Como executar

Este trabalho foi dividido em duas partes, a fim de facilitar as explicações e separar os arquivos de entrada. A pasta “analisador_pedido” trata-se da implementação de analisador léxico que reconhece nomes, placas de carro, etc. A pasta “assembler_mips” trata-se da segunda parte, que foi a implementação de analisador léxico escolhida por mim.

Requisitos: para executar qualquer uma das implementações, é necessário ter o *Flex* e o *GCC*. Os testes realizados foram executados no Flex 2.6.4, GCC 9.3.0 sobre o Linux Ubuntu 22.04 LTS (Kernel Microsoft WSL2 5.10.16.3).

Foram criadas *makefiles* para facilitar a compilação e execução de ambas implementações de analisadores. Sendo assim, para executar, navegue para a pasta da implementação desejada via terminal. No linux, isso é feito através do comando *cd* seguido do nome da pasta. Depois, digite *make* para realizar a compilação do arquivo Lex em código-fonte C e ele, por sua vez, em um binário executável. Em seguida, utilize o comando *make run* para executar o analisador léxico em modo interativo. Para ler dos arquivos, basta utilizar *make run0*, utilizando um número qualquer entre 0 a 2 (inclusive). Tais números se referem aos diferentes arquivos de entrada. No caso da implementação pedida na especificação, o número 0 trata-se do arquivo de entrada padrão passado pelo professor.

Exemplo de como abrir o analisador pedido em modo interativo:

```
1 cd analisador_pedido
2 make
3 make run
```

Exemplo de como abrir o montador do mips com o arquivo de entrada 1:

```
1 cd assembler_mips
2 make
3 make run1
```

Além disso, é possível executar manualmente seguindo os comandos da especificação, desde que as pastas sejam respeitadas. Ambos arquivos `lex` tem o mesmo nome, que é “*lex.l*”, então é necessário utilizar os mesmos comandos dados da primeira especificação de analisador. Os arquivos de entrada de cada implementação estão nas suas respectivas pastas *input*, então é preciso incluir esta informação. Por exemplo, para executar o mesmo exemplo anterior explicado via Makefile (ou seja, o montador do mips com o arquivo de entrada 1):

```
1 cd assembler_mips
2 flex lex.l
3 gcc lex.yy.c
4 ./a.out < input/entrada_personalizada1.txt
```

2. Introdução

Criar compiladores pode ser uma árdua tarefa. Além da dificuldade científica de desenhar um compilador, existe a dificuldades técnica de programar um. A fim de facilitar esta última, várias ferramentas foram criadas. Uma delas é o Lex, um gerador de analisador léxico, muito útil por abstrair boa parte da necessidade de pensar em como o analisador léxico deve funcionar, permitindo que o programador foque mais no design em si. Neste trabalho, foi utilizado o Flex, uma reimplementação mais flexiva e rápida do antigo Lex, além de ser *open-source*. Duas implementações utilizando essa ferramenta foram feitas, uma a partir da especificação pedida, e a outra baseada numa implementação do montador do MIPS, que será comparada com uma versão feita totalmente a mão criada por mim há alguns semestres para a disciplina de Organização de Computadores. Ao fim, discuto as considerações finais, apresentando os pontos positivos de usar uma ferramenta de gerador de analisador léxico, os pontos positivos do próprio Flex, os problemas enfrentados e suas soluções.

3. Código comum entre as implementações

A função *main* é exatamente igual nas duas implementações. Ela não recebe nenhum argumento, ou seja, não utiliza parâmetros do terminal e apenas retorna função *yylex*. O retorno é feito dessa forma para que, caso o Lex apresente qualquer erro, esse erro pode ser informado ao sistema operacional. O Lex utiliza os mesmos códigos de erro do Linux, então não é necessário tratar esses códigos.

```
1 int main(void) {
2     return yylex();
3 }
```

4. Analisador léxico pedido

A primeira implementação segue a especificação do próprio trabalho, sendo capaz de reconhecer números de telefone, inteiros positivos e negativos, pontos flutuantes, placas de carro, nomes de pessoas e palavras, além de ignorar espaços, *tabs* e quebras de linha entre os lexemas. Acabei modificando um pouco o arquivo inicial dado no exercício para seguir melhores práticas de programação, mas falarei sobre isso um pouco mais a frente.

4.1. Definição dos lexemas

Veja, por favor, a definição dos lexemas:

```
1 delimiters [ \t\n]+
2
3 digit [0-9]
4 uppercase_letter [A-Z]
5 lowercase_letter [a-z]
```

O primeiro lexema define os delimitadores e é chamado *delimiters*, e é justamente o que modifiquei do arquivo inicial. Trata-se de um fecho positivo dos caracteres espaço, *tab* e quebra de linha. Ele é responsável por identificar a formatação e indentação do código, que não é necessária para os processos futuros, ou mesmo para tratar texto mal-formatado. Sobre a modificação, alterei o nome dele para torná-lo mais explicativo, além de incluir o seu elemento nele mesmo, visto que o mesmo elemento não foi necessário em nenhum outro lugar do código. As próximas três definições, que são *digit*, *uppercase_letter* e *lowercase_letter*, definem, respectivamente, dígitos, letras maiúsculas e letras minúsculas. São formados da mesma forma, definindo uma faixa de caracteres do mesmo tipo dentro de colchetes, que representa uma disjunção exclusiva.

```
1 positive_integer [+]?{digit}+
2 negative_integer [-]{digit}+
```

O lexema *positive_integer* define um número inteiro positivo e é formado, primeiramente, por um caractere de soma opcional. Aqui, utilizei dos colchetes novamente. Como só há a soma dentro, ela será obrigatória. Fiz isso para representar o caractere de soma explicitamente. A outra forma seria utilizando barra invertida, que julguei menos legível, visto que poderia significar a própria barra em si seguida de um sinal de soma. A interrogação representa que a sub-palavra a direita, ou seja, o sinal de soma, é opcional. Depois, as chaves representam que um lexema já definido no arquivo será utilizado. Nesse caso, o lexema *digit*, utilizando o já explicado fecho de Kleene. O lexema *negative_integer* define um número inteiro negativo e é uma simples concatenação do sinal de menos com o fecho de Kleene de *digit*, visto que aqui o sinal é obrigatório.

```
1 float_point [+]?{digit}+[.]{digit}+
2 license_plate {uppercase_letter}{uppercase_letter}{uppercase_letter}
   [-]{digit}{digit}{digit}{digit}
3 word ({uppercase_letter}|{lowercase_letter})+
4 cell_phone_number {digit}{digit}{digit}{digit}[-]{digit}{digit}{digit}{
   digit}
5 person_name {word}[ ]{word}[ ]{word}|{word}[ ]{word}[ ]{word}[ ]{word}
```

O lexema *float_point* define o ponto flutuante, e é o o fecho positivo da disjunção exclusiva dos símbolos de soma e subtração, visto que o ponto flutuante pode aparecer com o sinal de soma, subtração ou nenhum sinal antes do número em si. Depois, há outro fecho positivo para o dígito antes do ponto, a concatenação com um ponto, e outro fecho positivo para o dígito depois do ponto. O lexema *license_plate* define as placas, e é a concatenação de três letras maiúsculas, um sinal de menos, e quatro dígitos. O lexema *word* define as palavras, e é o fecho positivo da disjunção exclusiva de *uppercase_letter* e *lowercase_letter*, porém utilizando chaves e barra vertical. O lexema *cell_phone_number* define o número de telefone, e é a concatenação de quatro dígitos, um sinal de menos e mais quatro dígitos. Por fim, o lexema *person_name* é a disjunção exclusiva da

concatenação de três palavras e da de quatro palavras, todas com exatamente um espaço entre elas.

4.2. Definição do que fazer ao encontrar um determinado lexema

Veja, por favor, a definição do que fazer ao encontrar um determinado lexema:

```
1 {delimiters} {}  
2 {cell_phone_number} {printf("Foi encontrado um telefone. LEXEMA: %s\n",  
    yytext);}   
3 {float_point} {printf("Foi encontrado um numero com parte decimal.  
    LEXEMA: %s\n", yytext);}   
4 {license_plate} {printf("Foi encontrado uma placa. LEXEMA: %s\n",  
    yytext);}   
5 {person_name} {printf("Foi encontrado um nome proprio. LEXEMA: %s\n",  
    yytext);}   
6   
7 {positive_integer} {printf("Foi encontrado um numero inteiro positivo.  
    LEXEMA: %s\n", yytext);}   
8 {negative_integer} {printf("Foi encontrado um numero inteiro negativo.  
    LEXEMA: %s\n", yytext);}   
9 {word} {printf("Foi encontrado uma palavra. LEXEMA: %s\n", yytext);}
```

Por padrão, o Lex consome o lexema encontrado e, além disso, executa o que está entre chaves. A ordem aqui é muito importante, visto que a tentativa de casamento do padrão com a palavra acontece de cima pra baixo na lista. Assim, caso um padrão tenha subpadrões que também são reconhecíveis, o maior normalmente precisa ficar mais acima da lista. Isso acontece algumas vezes aqui: os padrões do primeiro bloco são formados pelos subpadrões do segundo (Veja a seção anterior). Sobre os padrões, no primeiro caso, o lexema *delimiters* apenas é consumido e nenhuma ação é tomada. Em todos os outros, os lexemas também são consumidos e uma mensagem amigável é exibida, informando do que se trata o lexema e exibindo o próprio lexema.

4.3. Testes

Evitei mostrar os testes no modo interativo pois seria muito confuso para explicar, porém funciona normalmente e é o modo interativo que o próprio Lex fornece. O primeiro teste é com o arquivo de entrada fornecido na especificação. A saída é exatamente a esperada.

```
mateus@DESKTOP-54E0GTO:~/git/lex-example/part_one$ make && make run0
flex lex.l && gcc lex.yy.c
./a.out < input/entrada_padrao.txt
Foi encontrado um numero inteiro positivo. LEXEMA: 875878
Foi encontrado um numero inteiro negativo. LEXEMA: -3355456
Foi encontrado uma palavra. LEXEMA: abc
Foi encontrado um numero inteiro positivo. LEXEMA: 5464
Foi encontrado uma palavra. LEXEMA: abc
Foi encontrado um numero inteiro negativo. LEXEMA: -5464
Foi encontrado uma placa. LEXEMA: ABC-5464
Foi encontrado um numero inteiro positivo. LEXEMA: 453
Foi encontrado um numero inteiro negativo. LEXEMA: -2345
Foi encontrado um telefone. LEXEMA: 9486-0847
Foi encontrado um nome proprio. LEXEMA: Daniel Mendes Barbosa
Foi encontrado um numero com parte decimal. LEXEMA: 32.345
Foi encontrado uma palavra. LEXEMA: Palavra
Foi encontrado uma palavra. LEXEMA: Qualquer
Foi encontrado um telefone. LEXEMA: 3567-3224
Foi encontrado um nome proprio. LEXEMA: Daniel Mendes Barbosa Daniel
Foi encontrado uma palavra. LEXEMA: Mendes
Foi encontrado uma palavra. LEXEMA: Barbosa
Foi encontrado uma palavra. LEXEMA: Menezes
Foi encontrado um numero inteiro positivo. LEXEMA: 200
```

Figura 1. Teste com o arquivo de entrada passado na especificação.

O segundo teste utiliza o arquivo de entrada personalizado 1, que foi pensado a fim de ser totalmente consumido pelo analisador léxico e testar todas as possibilidades comuns. Segue seu conteúdo e sua execução:

```
1 Mateus Pinto da Silva bonito +3.1415 3.1415 -3.1415
2 Daniel Mendes Barbosa me passa por favor
3 1234-1234 ABC-1234 -50 +32 32
```

```
mateus@DESKTOP-54E0GTO:~/git/lex-example/part_one$ make && make run1
flex lex.l && gcc lex.yy.c
./a.out < input/entrada_personalizada1.txt
Foi encontrado um nome proprio. LEXEMA: Mateus Pinto da Silva
Foi encontrado uma palavra. LEXEMA: bonito
Foi encontrado um numero com parte decimal. LEXEMA: +3.1415
Foi encontrado um numero com parte decimal. LEXEMA: 3.1415
Foi encontrado um numero com parte decimal. LEXEMA: -3.1415
Foi encontrado um nome proprio. LEXEMA: Daniel Mendes Barbosa
Foi encontrado uma palavra. LEXEMA: me
Foi encontrado uma palavra. LEXEMA: passa
Foi encontrado uma palavra. LEXEMA: por
Foi encontrado uma palavra. LEXEMA: favor
Foi encontrado um telefone. LEXEMA: 1234-1234
Foi encontrado uma placa. LEXEMA: ABC-1234
Foi encontrado um numero inteiro negativo. LEXEMA: -50
Foi encontrado um numero inteiro positivo. LEXEMA: +32
Foi encontrado um numero inteiro positivo. LEXEMA: 32
```

Figura 2. Teste com o arquivo de entrada personalizado 1.

Como pode ser observado, toda a execução segue comum, e abrange dois casos que a entrada da especificação não o faz, que são os números decimais negativos e explicitamente positivos (ou seja, que tem sinal de menos e mais, respectivamente).

O último teste é feito a fim de mostrar problemas do analisador léxico. Segue seu conteúdo e sua execução:

```

1 PrimeiroNome SegundoNome TerceiroNome QuartoNome QuintoNome
2 +-3.1415      -+3.1415   .50    556.
3 AB-1234      ABCD-1234    ABC-123    ABC-12345
4 123-1234      1234-123    12345-1234    1234-12345
5
6
7
8
9
10 arquivo
11      feito
12      para testes
13      complexos

```

Na primeira linha, são testados nomes com cinco palavras e eles não são reconhecidos como apenas um nome próprio, mas sim um nome próprio e uma palavra. Logo depois, existem quatro pontos flutuantes mal formados. Os dois primeiros tem o seu primeiro sinal não consumido. Os dois últimos são reconhecidos como números inteiros positivos. Na terceira linha, há placas mal formadas. A primeira, segunda e terceira são reconhecidos como uma palavra seguida de um número negativo. O quinto é reconhecida como uma placa de carro seguida de um número inteiro. Na quarta linha, há números de telefone mal formados. Os três primeiros são reconhecidos como números inteiros positivos seguidos de negativos, e o último como um telefone seguido de um número positivo. Nas últimas linhas, há um aviso sobre o arquivo ser complexo que é corretamente reconhecido pelo analisador. Em toda a execução do arquivo de testes problemático, o código faz exatamente o que era esperado que ele fizesse.

```

mateus@DESKTOP-54E0GT0:~/git/lex-example/part_one$ make && make run2
flex lex.l && gcc lex.yy.c
./a.out < input/entrada_personalizada2.txt
Foi encontrado um nome proprio. LEXEMA: PrimeiroNome SegundoNome TerceiroNome QuartoNome
Foi encontrado uma palavra. LEXEMA: QuintoNome
+Foi encontrado um numero com parte decimal. LEXEMA: -3.1415
-Foi encontrado um numero com parte decimal. LEXEMA: +3.1415
.Foi encontrado um numero inteiro positivo. LEXEMA: 50
Foi encontrado um numero inteiro positivo. LEXEMA: 556
.Foi encontrado uma palavra. LEXEMA: AB
Foi encontrado um numero inteiro negativo. LEXEMA: -1234
Foi encontrado uma palavra. LEXEMA: ABCD
Foi encontrado um numero inteiro negativo. LEXEMA: -1234
Foi encontrado uma palavra. LEXEMA: ABC
Foi encontrado um numero inteiro negativo. LEXEMA: -123
Foi encontrado uma placa. LEXEMA: ABC-1234
Foi encontrado um numero inteiro positivo. LEXEMA: 5
Foi encontrado um numero inteiro positivo. LEXEMA: 123
Foi encontrado um numero inteiro negativo. LEXEMA: -1234
Foi encontrado um numero inteiro positivo. LEXEMA: 1234
Foi encontrado um numero inteiro negativo. LEXEMA: -123
Foi encontrado um numero inteiro positivo. LEXEMA: 12345
Foi encontrado um numero inteiro negativo. LEXEMA: -1234
Foi encontrado um telefone. LEXEMA: 1234-1234
Foi encontrado um numero inteiro positivo. LEXEMA: 5
Foi encontrado uma palavra. LEXEMA: arquivo
Foi encontrado uma palavra. LEXEMA: feito
Foi encontrado uma palavra. LEXEMA: para
Foi encontrado uma palavra. LEXEMA: testes
Foi encontrado uma palavra. LEXEMA: complexos

```

Figura 3. Teste com o arquivo de entrada personalizado 2.

5. Analisador léxico para o montador do MIPS (simplificado)

O analisador léxico escolhido por mim reconhece o assembly do MIPS, ou seja, é a primeira parte de um montador. Escolhi isso para poder comparar com uma versão desse

mesmo montador escrita sem o auxílio de geradores de analisadores léxicos que fiz para a disciplina de Organização de Computadores. Deixo essa comparação como última subseção desta parte.

5.1. Definição dos lexemas

Veja, por favor, a definição dos lexemas:

```
1 delimiters [ \t\n; ]+
2
3 reg_operation add|sub|and|or|xor|nand|nor|xnor
4 imm_operation addi|subi|andi|ori|xori|nandi|nori|xnori|beq|bne
5 float_operation fadd|fsub
6 jump_operation j|jr
```

Todas as definições aqui utilizam apenas operações do Lex que o outro analisador também utiliza, então me atentarei a apenas explicar o que são os lexemas para evitar redundância e um texto muito grande. Primeiro, há a definição do lexema *delimiters* bem parecida com a do analisador anterior, porém com a adição do caractere ponto e vírgula, pois é um caractere ignorado por padrão no assembly do MIPS. No próximo bloco, há a definição dos lexemas *reg_operation*, *imm_operation*, *float_operation* e *jump_operation* definem, respectivamente, as operações que podem ser efetuadas de instruções do tipo registrador, imediato, ponto flutuante e saltos incondicionais (ou jumps).

```
1 int_register zero|v[0-1]|a[0-3]|t[0-9]|s[0-7]|k[0-1]|gp|sp|fp|ra
2 float_register f([0-9]|([1-2][0-9]|[3][0-1]))
3
4 positive_dec [+]?[0-9]+
5 negative_dec [-][0-9]+
6 dec_imm {positive_dec}|{negative_dec}
7 imm {dec_imm}
8 label [a-z]+
```

Aqui, são definidos subpadrões úteis para os padrões reconhecíveis, a fim de tornar o código menos complexo e mais legível. No primeiro bloco, há a definição dos lexemas *int_register* e *float_register* que definem, respectivamente, os registradores de inteiro e de ponto flutuante. No segundo bloco, são definidos os lexemas de *positive_dec*, *negative_dec*, *dec_imm*, *imm* e *label* que definem, respectivamente, um número de base dez positiva, negativa, um número de base dez qualquer, um imediato qualquer e um *label*, que é uma posição de código para saltos.

```
1 label_definition {label}[:]
2 reg_instruction {reg_operation}[ ]{int_register}[,][ ]{int_register}
   [,][ ]{int_register}
3 imm_instruction {imm_operation}[ ]{int_register}[,][ ]{int_register}
   [,][ ]{imm}
4 float_instruction {float_operation}[ ]{float_register}[,][ ]{float_register}
   [,][ ]{float_register}
5 jump_instruction {jump_operation}[ ]{label}
```

Aqui, são definidos os padrões reconhecidos pelo analisador léxico. São definidos, respectivamente, os lexemas de *label_definition*, *reg_instruction*, *imm_instruction*, *float_instruction* e *jump_instruction* que definem, respectivamente, a declaração do novo *label*, uma instrução do tipo registrador, imediato, ponto flutuante e de saltos incondicionais.

5.2. Definição do que fazer ao encontrar um determinado lexema

Veja, por favor, a definição do que fazer ao encontrar um determinado lexema:

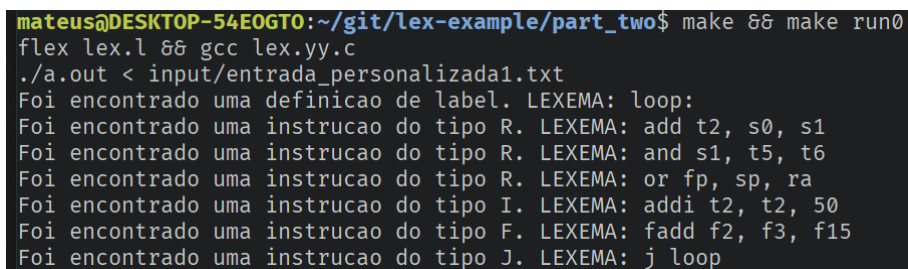
```
1 {delimiters} {}
2 {reg_instruction} {printf("Foi encontrado uma instrucao do tipo R.
    LEXEMA: %s\n", yytext);}
3 {imm_instruction} {printf("Foi encontrado uma instrucao do tipo I.
    LEXEMA: %s\n", yytext);}
4 {float_instruction} {printf("Foi encontrado uma instrucao do tipo F.
    LEXEMA: %s\n", yytext);}
5 {jump_instruction} {printf("Foi encontrado uma instrucao do tipo J.
    LEXEMA: %s\n", yytext);}
6 {label_definition} {printf("Foi encontrado uma definicao de label.
    LEXEMA: %s\n", yytext);}
```

Aqui, há basicamente a mesma coisa do analisador anterior. Caso haja um lexema *delimiter*, ele é consumido e mais nada acontece. Caso exista uma instrução, ela é consumida e uma mensagem amigável aparece.

5.3. Testes

O primeiro teste é um assembly MIPS bem feito. Segue seu conteúdo e sua execução:

```
1 loop:
2 add t2, s0, s1;
3 and s1, t5, t6;
4 or fp, sp, ra;
5 addi t2, t2, 50;
6 fadd f2, f3, f15;
7 j loop;
```



```
mateus@DESKTOP-54EOGT0:~/git/lex-example/part_two$ make && make run0
flex lex.l && gcc lex.yy.c
./a.out < input/entrada_personalizada1.txt
Foi encontrado uma definicao de label. LEXEMA: loop:
Foi encontrado uma instrucao do tipo R. LEXEMA: add t2, s0, s1
Foi encontrado uma instrucao do tipo R. LEXEMA: and s1, t5, t6
Foi encontrado uma instrucao do tipo R. LEXEMA: or fp, sp, ra
Foi encontrado uma instrucao do tipo I. LEXEMA: addi t2, t2, 50
Foi encontrado uma instrucao do tipo F. LEXEMA: fadd f2, f3, f15
Foi encontrado uma instrucao do tipo J. LEXEMA: j loop
```

Figura 4. Teste com o arquivo de entrada personalizado 1.

A execução, aqui, reconhece todas as instruções que estão corretas e apresenta seus tipos. Todos os caracteres são consumidos.

O segundo teste é também um assembly bem feito, porém com a adição de erros de formatação, com quebras de linha e pontos e vírgulas desnecessários. Segue seu conteúdo e sua execução:

```
1 add t2, s0, s1;
2
3
4
5 ;
```



```

6
7 ;
8
9 subi s0, s0, 5000
10 ;;;
11
12 add t1, t1, t2;
13
14 mateus:
15
16 fadd f20, f15, f9;
17 fsub f1, f1, f2;
18 j mateus;

```

```

mateus@DESKTOP-54E0GT0:~/git/lex-example/part_two$ make && make run1
flex lex.l && gcc lex.yy.c
./a.out < input/entrada_personalizada2.txt
Foi encontrado uma instrucao do tipo R. LEXEMA: add t2, s0, s1
Foi encontrado uma instrucao do tipo I. LEXEMA: subi s0, s0, 5000
Foi encontrado uma instrucao do tipo R. LEXEMA: add t1, t1, t2
Foi encontrado uma definicao de label. LEXEMA: mateus:
Foi encontrado uma instrucao do tipo F. LEXEMA: fadd f20, f15, f9
Foi encontrado uma instrucao do tipo F. LEXEMA: fsub f1, f1, f2
Foi encontrado uma instrucao do tipo J. LEXEMA: j mateus

```

Figura 5. Teste com o arquivo de entrada personalizado 2.

A execução, aqui, reconhece todas as instruções que estão corretas e apresenta seus tipos. Todos os caracteres são consumidos.

O último teste tem a mesma intenção do último teste do analisador anterior: levar o analisador léxico a seu limite e verificar inconsistências. Segue seu conteúdo e sua execução:

```

1 add t2, t2, 50;
2 and f2, f3, f15;
3 addi t2, s0, s1;
4 fadd s1, t5, t6;

```

```

mateus@DESKTOP-54E0GT0:~/git/lex-example/part_two$ make && make run2
flex lex.l && gcc lex.yy.c
./a.out < input/entrada_personalizada3.txt
addt2,t2,50andf2,f3,f15addit2,s0,s1fFoi encontrado uma instrucao do tipo R. LEXEMA: add s1, t5, t6

```

Figura 6. Teste com o arquivo de entrada personalizado 3.

Aqui, todas as instruções foram escritas com a operação errada para os operandos. A execução esperada era de que nenhuma instrução fosse reconhecida e todos os caracteres permanecessem não consumidos. Entretanto, o analisador léxico consome uma parte da última instrução e a interpreta erroneamente. A instrução “fadd s1, t5, t6;”, uma instrução do tipo F mal formada é interpretada como “add s1, t5, t6;”, uma instrução do tipo R bem formada. Isso ocorre pois o caractere “f” é ignorado. Para evitar esse tipo de problema, seria necessário abortar o programa ao identificar um caractere não consumido. Caso fosse necessário gerar código binário, ou seja, continuar o processo de codificação

do montador MIPS usando o LEX, seria necessário implementar isso. Para os próximos trabalhos (ou seja, construir um compilador), terei que fazer isso.

5.4. Comparação com um montador escrito sem nenhum gerador de analisador léxico

Veja, por favor, um trecho de código de montador escrito sem nenhum gerador de analisador léxico para a disciplina de Organização de Computadores feito para tratar espaços, comentários e quebras de linha.

```
1 with open(self.inputFilename, "r") as input_filename:
2     for line in input_filename:
3         if line.startswith("#") or line.isspace():
4             pass
5
6         elif line.strip().endswith(":"):
7             labelPreviousLine = line.strip()
8
9         else:
10            line = (line.split("#", 1))[0]
11
12            swap = line.split(":")
13            if len(swap) == 2:
14                label = swap[0].strip() + ":"
15                swap = swap[1]
16            elif len(swap) == 1:
17                label = ""
18                swap = swap[0]
19            else:
20                MipsMounter.__line_error(line)
21
22            while swap.startswith(" "):
23                swap = swap[1:]
24
25            swap = swap.split(" ", 1)
26            instruction = swap[0].lower().strip()
27            parameters = swap[1].split(",")
28
29            for i in range(len(parameters)):
30                parameters[i] = parameters[i].strip()
31
32            if instruction in self.instructions:
33                self.mounted.append(label)
34
35                if self.instructions[instruction]["type"] == "r":
```

Perceba que, através deste pequeno trecho do código, a complexidade de se programar aumenta consideravelmente, pois no mesmo código existem as regras de negócio do design do analisador léxico e as regras de execução, que são criadas pelo próprio Lex. Além disso, perceba a dificuldade de manter um código criado sem gerador de analisador léxico. É bastante difícil modificar alguma coisa. Conclui-se, então, há necessidade de tais geradores.

6. Considerações finais

De fato, geradores de analisador léxico ajudam muito. A diferença em construir um analisador léxico utilizando um ou não é muito grande. Entretanto, o conhecimento de design de compiladores, de expressões regulares, continua obrigatório, mesmo com auxílio dessas ferramentas. Sobre o Lex, fiquei um pouco decepcionado com suas mensagens de erro que não são nada amigáveis ou explicativas. Através deste trabalho, creio que estou mais apto a construir o analisador léxico dos trabalhos futuros, pois muitos erros encontrados e (alguns deles) descritos aqui com certeza apareceriam, e agora já sei como evitá-los ou tratá-los.