Este documento apresenta manipulação de arquivos e criação de tabela de símbolos através da ferramenta de compilação flex, respectivamente nas Seções 1 e 2.

A referência principal para elaboração desse documento é:

LEVINE, John. **Flex & Bison**. Editora O'Reilly. 2009. 271 p.

1 Arquivos no Flex

Analisadores léxicos no flex vão ler da entrada padrão, a não ser que seja determinada o contrário. Na prática, a maior parte dos analisadores fazem leitura de arquivos. Aqui será apresentado um exemplo de um programa para contagem de palavras, o qual será implementado com o uso de arquivos. Especialmente serão apresentadas duas versões, uma com leitura de um único arquivo e outra com a leitura de múltiplos arquivos.

1.1 Único Arquivo

Existem diferentes formas de manipular arquivos, entrada e saída no flex. A forma padrão define que o analisador léxico faz a leitura do `stdio FILE` chamado `yyin`. Assim para ler um único arquivo, basta configurar a leitura do arquivo antes da primeira chamada de `yylex`. No Exemplo 1, apresenta-se um programa simples para contagem de palavras, onde a entrada é especificada para um arquivo de entrada.

```
1  /* Exemplo de Arquivo no Flex (contagem de palavras) */
   %option noyywrap
3  %{
   int chars = 0;
5  int words = 0;
   int lines = 0;
7  %}

9  %%

11 [a-zA-Z]+    { words++; chars += strlen(yytext); }
   \n      { chars++; lines++; }
13 .    { chars++; }

15 %%

17 main(argc, argv)
   int argc;
19 char **argv;
   {
21   if (argc > 1) {
       if (!(yyin = fopen(argv[1], "r"))) {
23       perror(argv[1]);
         return(1);
25     }
     }
27   yylex();
     printf("%8d%8d%8d\n", lines, words, chars);
29 }
```

Listing 1: Exemplo 1 – Arquivo com Flex

O programa apresentado no Código 1 ilustra definições regulares para contagem de palavras, caracteres, linhas. Na terceira seção do código flex tem-se uma rotina principal encarregada de abrir um nome de arquivo passado por linha de comando, caso o usuário tenha especificado um nome de arquivo. Assim, o arquivo é atribuído a `yyin`.

Para compilar os seguintes comandos podem ser usados:

```
flex file-wc-1.l
cc lex.yy.c -lfl
./a.out arquivo-teste.txt
```

1.2 Múltiplos Arquivos

Agora será vista uma extensão do exemplo visto no Código 1, de forma que múltiplos arquivos podem ser lidos. Essa implementação é apresentada no Código 2.

O flex provê a rotina `yyrestart(f)`, a qual é usada para ler cada arquivo do início ao fim. Para cada arquivo, o mesmo é aberto, a função `yyrestart()` é utilizada para determinar os arquivos como a entrada para o analisador. Após, a função `yylex()` é chamada para fazer o escaneamento. Caso tenha mais de um arquivo na entrada, os valores totais são impressos na saída.

```
1  /* Exemplo de Multiplos Arquivos no Flex (contagem de palavras) */
   %option noyywrap
3  %{
   int chars = 0;
5  int words = 0;
   int lines = 0;
7
   int totchars = 0;
9  int totwords = 0;
   int totlines = 0;
11 %}

13 %%

15 [a-zA-Z]+    { words++; chars += strlen(yytext); }
   \n      { chars++; lines++; }
17 .    { chars++; }

19 %%

21 main(argc, argv)
   int argc;
23 char **argv;
   {
25   int i;

27   if (argc < 2) {  /* stdin eh lida */
       yylex();
29     printf("%8d%8d%8d\n", lines, words, chars);
       return(0);
31   }

33   for(i = 1; i < argc; i++) {
```

```
    FILE *f = fopen(argv[i], "r");
35
    if (!f) {
37    perror(argv[i]);
      return(1);
39  }
    yyrestart(f);
41  yylex();
    fclose(f);
43  printf("%8d%8d%8d %s\n", lines, words, chars, argv[i]);
    totchars += chars; chars = 0;
45  totwords += words; words = 0;
    totlines += lines; lines = 0;
47 }
   if(argc > 2) /* imprime valores totais, no caso de mais de um arquivo */
49   printf("%8d%8d%8d total\n", totlines, totwords, totchars);
  return(0);
51}
```

Listing 2: Exemplo 2 – Arquivos com Flex

2 Tabela de Símbolos

Programas escritos nas ferramentas flex e bison geralmente fazem uso de uma **tabela de símbolos** para manter o controle de nomes utilizados na entrada. Aqui será visto um exemplo simples de criação e utilização de tabela de símbolos em conjunto com um analisador léxico. Note que esse exemplo é fundamental para a utilização de tabela de símbolos em um analisador sintático que será visto adiante na disciplina de Compiladores.

2.1 Gerenciamento de Tabelas de Símbolos

O exemplo visto aqui tem a finalidade de manter o registro de cada palavra encontrada em arquivos, além do respectivo número da linha em que uma dada palavra aparece num arquivo. No Exemplo 3, apresentam-se as definições do analisador léxico, bem como as declarações referente a tabela de símbolos.

```
/* Exemplo - 01 - Flex, Files e Tab. Simbolos */
2 %option noyywrap nodefault yylineno case-insensitive

4 /* Tabela de Simbolo */
  %{
6   struct symbol {
      char *name;
8     struct ref *reflist;
    };
10
    struct ref {
12    struct ref *next;
      char *filename;
14    int flags;
      int lineno;
16  };

18  /* tabela de tam. fixo */
   #define NHASH 9997
20 struct symbol symtab[NHASH];
```

```
22|4 struct symbol *lookup(char*);
   void addref(int, char*, char*, int);
24
   char *curfilename;    /* nome do arquivo corrente */
26 %}
   %%
28 an |           /* Regras do Analisador Lexico */
   and |   /* artigos, preposicoes ignorados */
30 are |
   as |
32 at |
   be |
34 but |
   for |
36 in |
   is |
38 it |
   of |
40 on |
   or |
42 that |
   the |
44 this |
   to |
46 [A-Z]       /* palavras formadas por unica letra maiuscula/minuscula ignoradas
      */

48 [a-z]+(\'(s|t))?  { addref(yylineno, curfilename, yytext, 0); }
   .|\n      /* qualquer outro caracter eh ignorado */
50 %%
   /* funcoes em C para TS */
52 /* funcao hashing */
   static unsigned symhash(char *sym)
54 {
     unsigned int hash = 0;
56   unsigned c;

58   while(c = *sym++)
       hash = hash*9 ^ c;
60
     return hash;
62 }

64 struct symbol *lookup(char* sym)
   {
66   struct symbol *sp = &symtab[symhash(sym)%NHASH];
     int scount = NHASH;
68
     while(--scount >= 0) {
70     if (sp->name && !strcasecmp(sp->name, sym))
         return sp;
72
       if (!sp->name) {  /* nova entrada na TS */
74       sp->name = strdup(sym);
         sp->reflist = 0;
76       return sp;
       }
78
       if (++sp >= symtab+NHASH)
80       sp = symtab;  /* tenta a prox. entrada */
```

```
  }
  fputs("overflow na tab. simbolos\n", stderr);
  abort(); /* tabela estah cheia */
}

void addref(int lineno, char *filename, char *word, int flags)
{
  struct ref *r;
  struct symbol *sp = lookup(word);

  /* verificar se eh mesma linha e arquivo */
  if (sp->reflist &&
    sp->reflist->lineno == lineno &&
    sp->reflist->filename == filename) return;

  r = malloc(sizeof(struct ref));
  if (!r) {
    fputs("sem espaco\n", stderr);
    abort();
  }

  r->next = sp->reflist;
  r->filename = filename;
  r->lineno = lineno;
  r->flags = flags;
  sp->reflist = r;
}

/* Impressao das referencias, ordenadas alfabeticamente */

/* funcao auxiliar para ordenacao */
static int symcompare(const void *xa, const void *xb)
{
  const struct symbol *a = xa;
  const struct symbol *b = xb;

  if(!a->name) {
    if(!b->name) return 0;  /* ambos estao vazios */
    return 1;   /* coloca vazios no final */
  }
  if(!b->name) return -1;
  return strcmp(a->name, b->name);
}

void printrefs()
{
  struct symbol *sp, *sp_aux;

  /* ordenacao da TS */

  qsort(symtab, NHASH, sizeof(struct symbol), symcompare);

  for(sp = symtab; sp->name && sp < symtab+NHASH; sp++) {
    char *prevfn = NULL;  /* ultimo arquivo impresso, pra evitar duplicatas */

    /* reverte a lista de referencias */
    struct ref *rp = sp->reflist;   /* referencia atual */
    struct ref *rpp = 0;    /* referencia previa */
    struct ref *rpn;    /* prox. referencia */

```

```
6     do {
142     rpn = rp->next;
        rp->next = rpp;
144     rpp = rp;
        rp = rpn;
146   } while(rp);

148
      /* agora imprime a palavra e suas referencias */
150   printf("%10s", sp->name);
      for (rp = rpp; rp; rp = rp->next) {
152     if(rp->filename == prevfn) {
          printf(" %d", rp->lineno);
154     } else {
          printf(" %s:%d", rp->filename, rp->lineno);
156       prevfn = rp->filename;
        }
158   }
      printf("\n");
160 }
}
162
/* rotina principal */
164 main(argc, argv)
int argc;
166 char **argv;
{
168   int i;

170   if (argc < 2) { /* stdin eh lida */
        curfilename = "(stdin)";
172     yylineno = 1;
        yylex();
174   } else
      for (i = 1; i < argc; i++) {
176     FILE *f = fopen(argv[i], "r");

178     if(!f) {
          perror(argv[i]);
180       return(1);
        }
182     curfilename = argv[i];

184     yyrestart(f);
        yylineno = 1;
186     yylex();
        fclose(f);
188     }

190   printrefs();
}
```

Listing 3: Exemplo – Analisador Léxico & Tabela de Símbolos

A linha %`option` apresenta o uso de algumas opções específicas. A opção %`yylineno` indica ao flex para definir uma variável inteira chamada `yylineno` e manter o número da linha atual que está sendo escaneada. Assim, toda vez que o analisador léxico lê um caracter de nova linha, a variável `yylineno` é incrementada. Note que ainda faz-se necessário inicializar `yylineno` no início do escaneamento de cada arquivo, naturalmente no caso de manipular vários arquivos.

Outra opção selecionada é `case-insentive`, a qual indica ao flex para construir um analisador que não faz distinção entre caracteres maiúsculos e minúsculos. Assim, um padrão `abc` irá casar com: *abc*, *Abc*, *ABc*, *AbC*, etc.

A tabela de símbolos é definida como uma matriz de símbolos (conforme definição da estrutura *symbol*), cada elemento da tabela contém um ponteiro para um nome e uma lista de referências. As referências são definidas através de uma lista encadeada com números de linha e ponteiros para nome de arquivos. A Fig. 1 ilustra uma representação da Tabela de Símbolos utilizada neste exemplo. Adicionalmente, define-se `curfilename`, um ponteiro estático para o nome do arquivo corrente, o qual será usado quando referências são adicionadas.

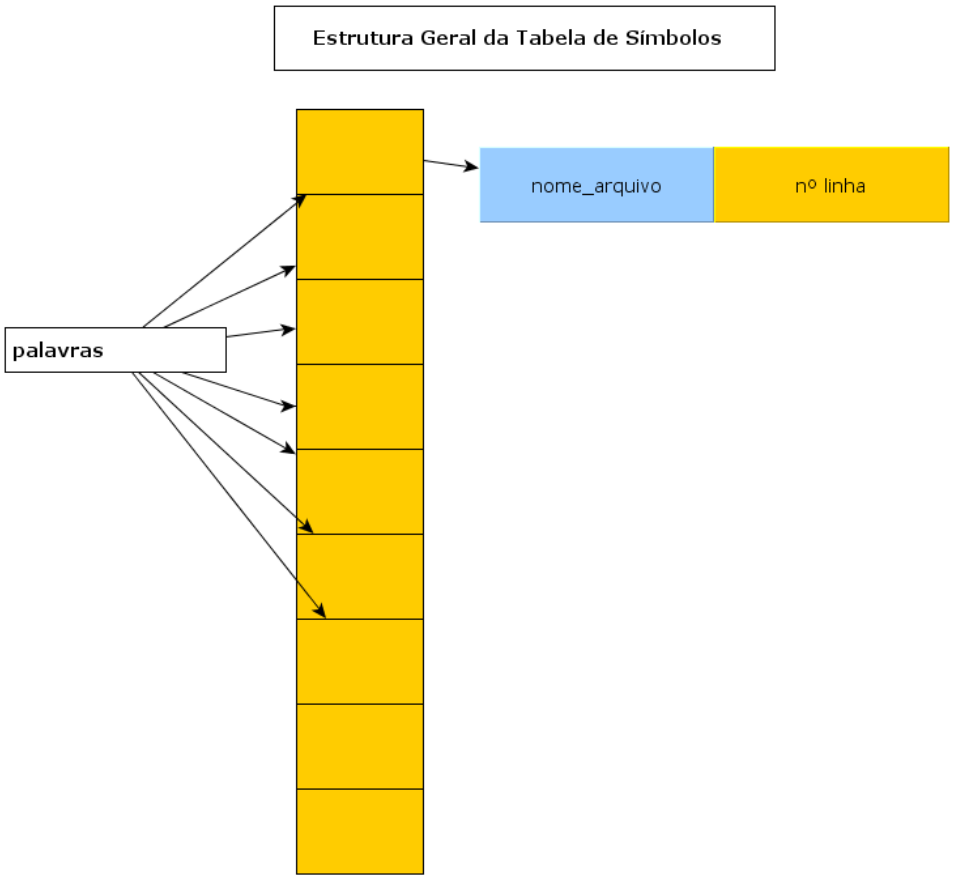

Figura 1: Representação da Tabela de Símbolos

No exemplo aqui apresentado as palavras do arquivo escaneado são indexadas. Nesse tipo de aplicação, geralmente palavras curtas (como, artigos e preposições) são ignoradas para regras de expressões regulares. Assim, as regras vistas no Código 3 não possuem nenhuma ação associada a identificação dessas palavras curtas.

A próxima expressão regular é definida para "casar" com a grande maioria de palavras da língua Inglesa. Note que a regra inclusive pode "casar" palavras como: *owner's* e *can't*. Cada palavra "casada" é passada para função `addref` (que será descrita adiante), em conjunto com a identificação do arquivo corrente e número de linha. O padrão (ou regra) final "casa" como qualquer outra entrada que não tenha sido filtrada pelas regras anteriores.

A rotina principal do programa é similar ao exemplo visto anteriormente para contagem de palavras, linhas e caracteres em arquivos. Ou seja, a rotina é responsável por abrir cada arquivo, depois utilizar a função `restart` para gerenciar a leitura do mesmo e então chamar a função `yylex`. Ademais, tem-se o uso de `curfilename` para indicar o nome do arquivo corrente, o qual será necessário quando a lista de referências for construída. Ainda a variável `yylineno` é

setada" com valor **1** para cada arquivo. Caso contrário, a numeração das linhas iria continuar de um arquivo para outro, o que naturalmente pode ser desejável dependendo da aplicação. Por fim, a função `printrefs` é responsável por ordenar alafabeticamente a tabela de símbolos e imprimir as respectivas referências.

2.2 Utilizando a Tabela de Símbolos

Continuando com o exemplo de uso da Tabela de Símbolos, agora destaca-se o código do analisador léxico, o qual inclui rotinas para gerenciar a Tabela de Símbolos. Uma rotina adiciona uma palavra a Tabela de Símbolos, outra rotina é usada para imprimir as palavras indexadas.

A rotina `lookup` tem a finalidade de a partir de uma cadeia (*string* referente a um nome) retornar o respectivo endereço na tabela de símbolos para o nome, caso não tenha uma entrada na tabela para o nome buscado, cria-se uma nova entrada. A técnica da rotina `lookup` é conhecida como *hashing com tentativa linear*. Utiliza-se uma função *hash* para transformar uma cadeia em um número de entrada na tabela, então verifica-se a entrada, caso a entrada já esteje ocupada por um outro símbolo, a tabela será escaneada linearmente até que uma nova entrada seja encontrada.

A função *hash* é relativamente simples. Para cada caracter, o resultado anterior da função *hash* é multiplicado por nove, após é feita uma operação `xor` com o caracter. A função `lookup` calcula a entrada na tabela de símbolos conforme o valor calculado pela técnica de hashing.

Outra rotina apresentada no Código 3 é `addref`, essa rotina é chamada a partir do analisador léxico para adicionar uma referência a uma palavra específica. A implementação é feita através de uma lista encadeada de referências a partir das estruturas de símbolos. Para evitar que seja gerado um relatório desnecessariamente grande, não adiciona-se uma referência se o símbolo já possuir uma referência para a mesma linha e nome de arquivo. A variável `flag` poderia ser utilizada para outras aplicações baseadas nesse exemplo.

A última rotina é responsável por ordenar e imprimir a tabela de símbolos. A tabela de símbolos é criada em uma ordem que depende da função hash, a qual não é adequada para leitores humanos. Assim, a tabela de símbolos é ordenada alfabeticamente através da função `qsort`.

Depois, a função `printrefs` percorre a tabela de símbolos e imprime as referências para cada palavra. As referências estão em uma lista encadeada, a qual será percorrida para impressão dos respectivos valores. Ainda para fazer a geração do relatório "amigável", o nome de arquivo é impresso apenas se ele é diferente do nome do arquivo anterior.

Por fim, a Fig. 2 ilustra alguns exemplos de teste da implementação do analisador léxico com tabela de símbolos.

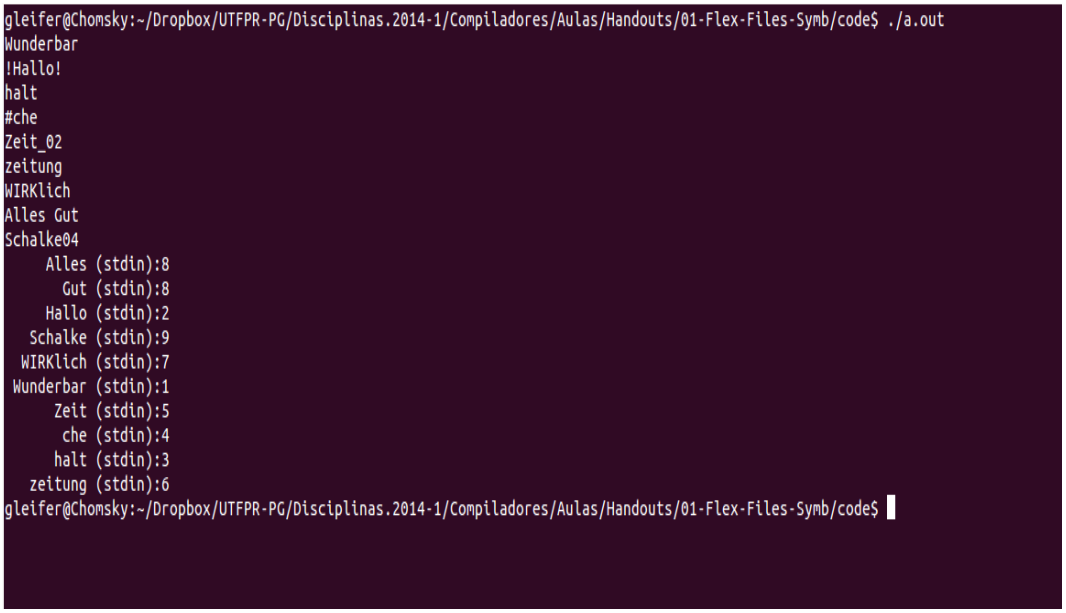

Figura 2: Testes – Exemplo Tabela de Símbolos