

[MAC0211] Laboratório de Programação I
Aula 20
AWK
Geradores de Analisadores Léxicos – Flex

Kelly Rosa Braghetto

DCC-IME-USP

21 de maio de 2013

Na aula passada...

- ▶ Expressões regulares
- ▶ Awk (Parte 1)

[Aula passada] Awk

- ▶ O *Awk* é uma ferramenta para tratamento de textos baseada em expressões regulares
- ▶ Com ela, podemos fazer operações tais como:
 - ▶ Processamento de arquivos texto e criação de relatórios a partir dos resultados
 - ▶ Tradução de arquivos de um formato para outro
 - ▶ Criação de pequenos bancos de dados
 - ▶ Realização de operações matemáticas em arquivos de dados numéricos
- ▶ O *Awk* pode ser usado para realizar tarefas simples de processamento de texto via linha de comando, ou como uma linguagem de programação para a criação de scripts para processamentos mais complexos
- ▶ *Awk* vem do nome dos seus criadores: Aho, Weinberger & Kernighan

[Aula passada] Estrutura de um script Awk

- ▶ Formato mais simples:

```
awk '<padrão de busca> <ações do programa>' arq_entrada
```

- ▶ Formato mais geral:

```
awk [-F<sc>] 'prog' | -f <arq_prog> [<vars>] [-|<arq_entrada>]
```

`sc` caracter separador de campo

`prog` programa Awk de linha de comando

`arq_prog` arquivo contendo um programa Awk

`vars` inicialização de variáveis

- lê da entrada padrão

`arq_entrada` arquivo texto de entrada

onde `arq_prog` é um script que possui a seguinte estrutura

```
BEGIN           {<inicializações>}  
<padrão de busca> {<ações do programa>}  
<padrão de busca> {<ações do programa>}  
...  
END             {<ações finais>}
```

[Aula passada] Awk – entradas

- ▶ O programa `awk` lê um script e o aplica na sua entrada
- ▶ O script pode ser passado como o primeiro parâmetro do programa (entre ' ') ou então através da opção `-f nome_do_script`
- ▶ A entrada é quebrada em registros (normalmente, linhas)
- ▶ Os registros são quebrados em campos (normalmente, nos espaços em branco) que podem ser referenciados por meio das variáveis posicionais `$1`, `$2`, ..., `$n`
- ▶ `$0` se refere ao registro inteiro
- ▶ `NR` é uma variável “embutida” que contém o número de registros já processados
- ▶ `NF` é a variável que contém o número de campos do registro atual
- ▶ `length` é a variável que contém o número de caracteres da linha atual
- ▶ Para definir um outro separador de campos, basta usar a opção `-F` no `awk` para determinar o separador.

[Aula passada] Expressões regulares do Awk

Semelhantes às do UNIX:

`^` casa com o começo da string (p.ex., começo da linha)

`$` casa com o fim da string (p.ex., fim da linha)

`\` é o metacaracter de *escape*, que pode ser usado para remover o significado especial de um metacaracter

`.` casa com qualquer caractere, inclusive o newline

`[xyz]` casa com 1 caractere do conjunto xyz

`[^xyz]` casa com 1 caractere qualquer que não esteja no conjunto xyz

`|` para indicar alternativas (ou)

`*` casa com a expressão anterior repetida 0 ou mais vezes

`+` casa com a expressão anterior repetida 1 ou mais vezes

`BEGIN` casa com o início do arquivo

`END` casa com o final do arquivo

[Aula passada] Awk – exemplos simples

- ▶ Move todos os arquivos cujo nome inicia com “junk” para o diretório “lixo”, renomeando-os com uma extensão “.lix”
`ls junk* | awk '{print "mv \"$0\" ../lixo/"$0".lix"}' | bash`

- ▶ Calcula a soma e a média dos números armazenado em um arquivo (um número por linha)

```
BEGIN { s = 0 }  
      { s += $1 }  
END   { print "Soma: ", s, "Media: ", s/NR }
```

- ▶ Calcula o tamanho médio das linhas de um arquivo

```
      { s += length }  
END { print "Tamanho medio das linhas: ", s/NR }
```

[Aula passada] Outras características importantes do Awk

- ▶ As variáveis não são declaradas. Elas passam a existir na primeira vez em que são usadas
- ▶ Não existe tipo de variável; uma mesma variável pode conter num dado momento uma string e, num outro, um número
- ▶ Não é preciso inicializar variáveis; o valor inicial é sempre string vazia (o que é convertido em 0 numa operação aritmética)
- ▶ É possível a utilização de vetores. Os índices de vetores podem ser tanto números quanto strings (ver o exemplo anterior)

[Aula passada] Awk – exemplos envolvendo vetores

- ▶ Contador de ocorrências de palavras

```
{
    for (i = 1; i <= NF; i++)
        freq[$i]++
}
END {
    for (palavra in freq)
        printf "%s\t%d\n", palavra, freq[palavra]
}
```

- ▶ Ordena um arquivo considerando como chave os valores numéricos armazenados em seu primeiro campo

```
{
    if ($1 > max)
        max = $1
    vet[$1] = $0
}
END {
    for (x = 1; x <= max; x++)
        if (x in vet)
            print vet[x]
}
```

Awk – exemplos envolvendo busca de padrões

- ▶ Imprime todas as linhas que começam com uma vogal
`awk '/^[AEIOUaeiou]/'`
- ▶ Imprime as linhas cuja segunda palavra é maior (numérica ou lexicograficamente) do que a primeira. A comparação será numérica se ambas palavras tiverem representação numérica
`awk '$2 > $1'`
- ▶ Resultados de jogos onde o primeiro time venceu (o nome do time não pode ter espaços)
`awk '/^[a-zA-Z]+ +[0-9]+ +x +[0-9]+ +[a-zA-Z]+/ && $2>$4'`
- ▶ Linhas começando com dígito ou com o 2º campo maior do que o 3º
`awk '/^[0-9]/ || $2 > $3'`

Awk – exemplos envolvendo busca de padrões

- ▶ O operador `~` é chamado de *match* e diz se uma expressão casa com um determinado padrão especificado através de uma expressão regular
- ▶ Os operadores `~` e `!~` podem ser usados como nos exemplos abaixo ou dentro de comandos `if`, `while` e `for`

Exemplos:

- ▶ linhas cujo primeiro campo casa com a expressão regular Janeiro.
`awk '$1 ~ /Janeiro/'`
- ▶ Linhas cujo primeiro campo não casa a expressão regular Janeiro.
`awk '$1 !~ /Janeiro/'`

Awk – outro exemplo

- ▶ Soma dos tamanhos dos arquivos cujo mês da última alteração é Maio:

```
ls -lg | awk '$6 == "Mai" { sum += $5 }  
            END { print sum }'
```

Obs.: A saída do comando `ls -lg` é algo como:

```
-rw-r--r--  1 arnold  user   1933 Nov  7 13:05 Makefile  
-rw-r--r--  1 arnold  user  10809 Nov  7 13:03 gawk.h  
-rw-r--r--  1 arnold  user    983 Apr 13 12:14 gawk.tab.h  
-rw-r--r--  1 arnold  user  31869 Jun 15 12:20 gawk.y  
-rw-r--r--  1 arnold  user  22414 Nov  7 13:03 gawk1.c  
-rw-r--r--  1 arnold  user  37455 Nov  7 13:03 gawk2.c  
-rw-r--r--  1 arnold  user  27511 Dec  9 13:07 gawk3.c  
-rw-r--r--  1 arnold  user   7989 Nov  7 13:03 gawk4.c
```

Awk – um exemplo mais completo ¹

- ▶ Entrada: arquivo `coins.txt`, contendo a descrição de uma coleção de moedas
- ▶ Formato do arquivo de entrada: 5 colunas
metal | peso | ano de cunhagem | país de origem | nome
- ▶ Exemplo:

```
gold      1      1986  USA      American Eagle
gold      1      1908  Austria-Hungary  Franz Josef 100 Korona
silver    10      1981  USA      ingot
gold      1      1984  Switzerland  ingot
gold      0.5    1981  RSA      Krugerrand
gold      0.1    1986  PRC      Panda
silver    1      1986  USA      Liberty dollar
gold      0.25   1986  USA      Liberty 5-dollar piece
silver    1      1987  USA      Constitution dollar
gold      0.25   1987  USA      Constitution 5-dollar piece
gold      1      1988  Canada   Maple Leaf
```

- ▶ Saída desejada: dados sumarizados da coleção de moedas

¹Extraído de: *An Awk Primer* – <http://www.vectorsite.net/tsawk.html>

Awk – um exemplo mais completo

- ▶ Lista todas as moedas de ouro:

```
awk '/gold/' coins.txt
```

```
awk '/gold/ {print}' coins.txt
```

```
awk '/gold/ {print $0}' coins.txt
```

- ▶ Lista o nome de todas as moedas de prata:

```
awk '/silver/ {print $5,$6,$7,$8}' coins.txt
```

Note que algumas moedas possuem nomes compostos (com até 4 palavras); por essa razão, o comando acima imprime os valores dos campos de 5 a 8. Se uma dada linha não possui todos esses campos, o valor da variável correspondente ao campo “faltante” é string vazia

- ▶ Considerando que o valor do ouro por unidade de peso é \$425, calcula o valor total das moedas de ouro:

```
awk '/gold/ {weight += $2} END {print "value = $"425*weight}' coins.txt
```

Awk – um exemplo mais completo

```
# This is an awk program that summarizes a coin collection.
#
/gold/    { num_gold++; wt_gold += $2 }    # Get weight of gold.
/silver/  { num_silver++; wt_silver += $2 } # Get weight of silver.
END { val_gold = 485 * wt_gold;          # Compute value of gold.
      val_silver = 16 * wt_silver;       # Compute value of silver.
      total = val_gold + val_silver;
      print "Summary data for coin collection:"; # Print results.
      printf ("\n");
      printf ("    Gold pieces:                %2d\n", num_gold);
      printf ("    Weight of gold pieces:         %5.2f\n", wt_gold);
      printf ("    Value of gold pieces:             %7.2f\n", val_gold);
      printf ("\n");
      printf ("    Silver pieces:                       %2d\n", num_silver);
      printf ("    Weight of silver pieces:              %5.2f\n", wt_silver);
      printf ("    Value of silver pieces:               %7.2f\n", val_silver);
      printf ("\n");
      printf ("    Total number of pieces:                %2d\n", NR);
      printf ("    Value of collection:                  %7.2f\n", total); }
```

Awk – outras funcionalidades

- ▶ `/(<string1>)|(<string2>)/` – define padrões de busca alternativos
- ▶ `/<string1>/,/<string2>/` – define uma faixa de linhas, delimitadas por 2 strings
- ▶ `==, !=, <, >, <=, >=` – operadores que podem ser usados na definição de condições de busca
- ▶ `&&` (AND) e `||` (OR) – operadores que conectam duas condições
- ▶ `+, -, *, /, %, ++, --` – operadores aritméticos
- ▶ `sqrt, log, exp, int` – funções aritméticas
- ▶ `length, substr, split, index` – funções para strings
- ▶ `while, if, for` (sintaxe semelhante à da linguagem C) – estruturas de controle de fluxo

Awk – quer mais exemplos???

Consulte a seção de exemplos do manual do Awk:

`http://www.gnu.org/software/gawk/manual/gawk.html#
Sample-Programs`

[Relembrando] O processo de compilação

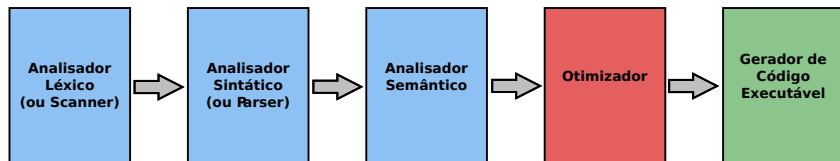
Um compilador ou interpretador para uma linguagem de programação é geralmente decomposto em duas subtarefas:

1. Ler o código fonte e descobrir sua estrutura
2. Processar essa estrutura, geralmente para gerar um código executável (programa)

A primeira tarefa também pode ser dividida em duas:

1. Dividir o arquivo de entrada em *tokens* (itens léxicos) ⇒ **análise léxica**
2. Encontrar a estrutura hierárquica do programa ⇒ **análise sintática**

O processo de compilação



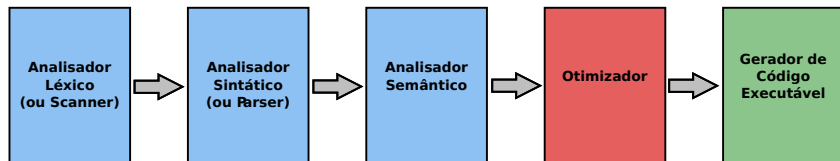
Análise Léxica

var1 = 1 - 3**2



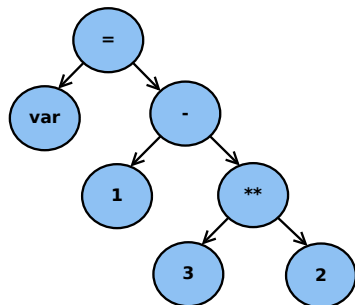
Lexema	Tipo do <i>token</i>
var1	Variável
=	Operador de Atribuição
1	Número
-	Operador de Subtração
3	Número
**	Operador de Potência
2	Número

O processo de compilação



Análise Sintática

Lexema	Tipo do <i>token</i>
var1	Variável
=	Operador de Atribuição
1	Número
-	Operador de Subtração
3	Número
**	Operador de Potência
2	Número



Geradores de Analisadores

- ▶ São programas que recebem como entrada um arquivo com uma descrição em alto do que o analisador deve reconhecer e que geram como saída um código que, depois de compilado/interpretado, vira um programa analisador
- ▶ Exemplos:
 - ▶ **geradores de analisadores léxicos**
 - ▶ para C: Lex, Flex
 - ▶ para Java: JFlex
 - ▶ **geradores de analisadores sintáticos**
 - ▶ para C: Yacc, Bison
 - ▶ para Java: JavaCC, ANTLR

Geralmente, são usados em conjunto: Lex + YACC, Flex + Bison

Flex – *Fast Lexical Analyzer Generator*

- ▶ Implementação gratuita e de código aberto do Lex
- ▶ Distribuído pelo projeto GNU, mas não é parte dele
- ▶ **Entrada do programa:** arquivo contendo uma tabela de expressões regulares e suas respectivas ações associações
- ▶ **Saída do programa:** o código fonte em C de um analisador léxico que reconhece as expressões regulares especificadas no arquivo de entrada

Formato de um arquivo de entrada para o Flex

O arquivo é dividido em três partes, separadas por linhas que começam com `%%`:

```
[Definições]
```

```
%%
```

```
[Regras]
```

```
%%
```

```
[Código do usuário]
```

Flex – Definições [parte do início]

A seção de *Definições* pode conter:

- ▶ código “**literal**” que deve aparecer no analisador fora de qualquer função. Esse código deve ser delimitado por `%{ %}` ou então deve aparecer indentado
- ▶ declarações de **definições de nomes simples**. São feitas no formato:

`[nome] [definição]`

Exemplos:

```
DIGITO          [0-9]
```

```
IDENTIFICADOR  [a-z][a-z0-9]*
```

- ▶ declarações de **condições de início** para regras adicionais ou regras exclusivas. São feitas nos seguintes formatos:
 - ▶ condição para regras adicionais: `%s [nome]`
 - ▶ condição para regras exclusivas: `%x [nome]`

(Veremos mais detalhes sobre o seu uso mais adiante)

Flex – ...

Continua na próxima aula...

Bibliografia e materiais recomendados

- ▶ GNU AWK – Guia do Usuário
<http://www.gnu.org/software/gawk/manual/gawk.html>
- ▶ *An Awk primer* (tutorial de Awk bastante interessante)
<http://www.vectorsite.net/tsawk.html>
- ▶ Manual do Flex
<http://flex.sourceforge.net/manual/>
- ▶ Notas das aulas de MAC0211 de 2010, feitas pelo Prof. Kon
<http://www.ime.usp.br/~kon/MAC211>

Cenas dos próximos capítulos...

Na próxima aula:

- ▶ Flex (continuação)
- ▶ Geradores de analisadores sintáticos
- ▶ GNU Bison