Lecture 3
AWK
EVOLUTION OF LANGUAGE THROUGH THE AGES.

6000 B.C.

UGH.

GRRF.

BOOGA.

2000 A.D.

GREP.

AWK.

SED.

COPYRIGHT (C) 1999 ILLIAD

HTTP://WWW.USERFRIENDLY.ORG/
Over-simplified history of programming languages

• 1940's machine language
• 1950's assembly language
• 1960's high-level languages: Algol, Fortran, Cobol, Basic
• 1970's systems programming: C
• 1980's object-oriented: C++
• 1990's strongly-hyped: Java
• 2000's lookalike languages: C#
• 2010's retry: Go, Rust, Swift

scripting languages:

Snobol
shell
Awk
Perl, Python, PHP, ...
Javascript
Dart, Typescript, Elm,
Coffeescript, ...?
What's a scripting language?

"Scripting is a lot like obscenity. I can't define it, but I'll know it when I see it."

Larry Wall, creator of Perl

- characteristics
  - text as a basic data type
  - regular expressions for text searching and manipulation
  - associative arrays as a basic aggregate type
  - minimal use of types, declarations, initialization, etc.
  - usually interpreted instead of compiled
AWK

• a language for pattern scanning and processing
  – Al Aho, Brian Kernighan, Peter Weinberger, at Bell Labs, ~1977
• intended for simple data processing:

• selection, validation:
  "Print all lines longer than 80 characters"
  length > 80

• transforming, rearranging:
  "Print first two fields in the opposite order"
  { print $2, $1 }

• report generation:
  "Add up the numbers in the first field,
  then print the sum and average"
  { sum += $1 }
  END { print sum, sum/NR }
Structure of an AWK program:

- a sequence of pattern-action statements

  pattern       { action }
  pattern       { action }
  ...

- "pattern" is a regular expression, numeric expression, string expression
  or combination of these
- "action" is executable code, similar to C

- usage:

  awk 'program' [ file1 file2 ... ]
  awk -f progfile [ file1 file2 ... ]

- operation:

  for each file
  for each input line
    for each pattern
      if pattern matches input line
      do the action
AWK features:

• input is read automatically across multiple files
  – lines are split into fields ($1, ..., $NF; $0 for whole line)
• variables contain string or numeric values (or both)
  – no declarations: type determined by context and use
  – initialized to 0 and empty string
  – built-in variables for frequently-used values
• operators work on strings or numbers
  – coerce type / value according to context
• associative arrays (arbitrary subscripts)
• regular expressions (like egrep)
• control flow statements similar to C: if-else, while, for, do
• built-in and user-defined functions
  – arithmetic, string, regular expression, text edit, ...
• printf for formatted output
• getline for input from files or processes
Some small useful AWK programs

NF > 0 print non-empty lines

{ sum += $1
END { print sum }
print sum of values in first field

{ x[NR] = $0 }
reverse input by lines
END { for (i = NR; i > 0; i--) print x[i] }

{ for (i = 1; i <= NF; i++) w[$i]++ }
word freq count
END { for (i in w) print i, w[i] }
Associative Arrays

• array subscripts can have any value, not just integers
• canonical example: adding up name-value pairs

• input:
  pizza 200
  beer 100
  pizza 500
  beer 50

• output:
  pizza 700
  beer 150

• program:

    { amount[$1] += $2 }
    END { for (name in amount)
        print name, amount[name] | "sort -k1 -nr"
    }

Awk text formatter

#!/bin/sh
# f - format text into 60-char lines

awk ' 
/./ { for (i = 1; i <= NF; i++)
    addword($i) }
/^$/ { printline(); print "" } 
END { printline() }

function addword(w) {
    if (length(line) + length(w) > 60)
        printline()
    line = line space w
    space = " "
}

function printline() {
    if (length(line) > 0)
        print line
    line = space = ""
}
' "$@"
Anatomy of a compiler

- Input
  - lexical analysis
  - syntax analysis
  - code generation
  - symbol table

- Intermediate form
  - tokens
  - object file

- Linking
  - a.out

- Output
  - a.out
  - data
Anatomy of an interpreter

input tokens

lexical analysis

syntax analysis

symbol table

intermediate form

execution

output

input data
Parsing by recursive descent

expr: term | expr + term | expr - term
term: factor | term * factor | term / factor
factor: NUMBER | ( expr )

NF > 0 {
  f = 1
  e = expr()
  if (f <= NF) printf("error at \%s\n", $f)
  else printf("\t%.8g\n", e)
}

function expr( e) {      # term | term [+-] term
  e = term()
  while ($f == "+'" || $f == "-'")
    e = $(f++) == "+'" ? e + term() : e - term()
  return e
}

function term( e) {      # factor | factor [*]/ factor
  e = factor()
  while ($f == "*" || $f == "/")
    e = $(f++) == "*" ? e * factor() : e / factor()
  return e
}

function factor( e) {      # number | ( expr)
  if ($f ~ ^([-+])?([0-9]+[.]?[0-9]*|[.]?[0-9]+)$/) {
    return $(f++)
  } else if ($f == "(") {
    f++
    e = expr()
    if ($(f++) != ")")
      printf("error: missing ) at \%s\n", $f)
    return e
  } else {
    printf("error: expected number or ( at \%s\n", $f)
    return 0
  }
}
YACC and LEX

• languages/tools for building [parts of] compilers and interpreters

• YACC: "yet another compiler compiler" (S. C. Johnson, ~ 1972)
  – converts a grammar and semantic actions into a parser for that grammar

• LEX: lexical analyzer generator (M. E. Lesk, ~ 1974)
  – converts regular expressions for tokens into a lexical analyzer that recognizes those tokens

• parser calls lexer each time it needs another input token
• lexer returns a token and its lexical type

• when to think of using them:
  – real grammatical structures (e.g., recursively defined)
  – complicated lexical structures
  – rapid development time is important
  – language design might change
YACC overview

• YACC converts grammar rules & semantic actions into parsing fcn yyparse()
  – yyparse parses programs written in that grammar, performs semantic actions as grammatical constructs are recognized

• semantic actions usually build a parse tree
  – each node represents a particular syntactic type, children are components

• code generator walks the tree to generate code
  – may rewrite tree as part of optimization

• an interpreter could
  – run directly from the program (TCL, shells)
  – interpret directly from the tree (AWK, Perl?):
    at each node, interpret children (recursion), do operation of node itself, return result
  – generate byte code output to run elsewhere (Java)
  – generate byte code (Python, …)
  – generate C to be compiled later

• compiled code runs faster
  – but compilation takes longer, needs object files, less portable, …

• interpreters start faster, but run slower
  – for 1- or 2-line programs, interpreter is better
  – on the fly / just in time compilers merge these (e.g., C#, Java)
while \( b \neq 0 \)
  if \( a > b \)
    \( a = a - b \)
  else
    \( b = b - a \)
return \( a \)
Grammar specified in YACC

- grammar rules specify the syntax
- the action part of a rule specifies the semantics
  - usually used to build a parse tree for subsequent processing

statement:
- `IF ( expression ) statement` { create node(IF, expr, stmt, 0) }
- `IF ( expression ) statement ELSE statement` { create node(IF, expr, stmt1, stmt2) }
- `WHILE (expression) statement` { create node(WHILE, expr, stmt) }
- `variable = expression` { create node(ASSIGN, var, expr) }

expression:
- `expression + expression`
- `expression - expression`

- YACC creates a parser from this
- when the parser runs, it creates a parse tree
- a compiler walks the tree to generate code
- an interpreter walks the tree to execute it
Excerpts from AWK grammar

term:
  | term '+' term          { $$ = op2(ADD, $1, $3); }  
  | term '-' term          { $$ = op2(MINUS, $1, $3); }  
  | term '*' term          { $$ = op2(MULT, $1, $3); }  
  | term '/' term          { $$ = op2(DIVIDE, $1, $3); }  
  | term '%' term          { $$ = op2(MOD, $1, $3); }  
  | '-' term %prec UMINUS  { $$ = op1(UMINUS, $2); }  
  | INCR var               { $$ = op1(PREINCR, $2); }  
  | var INCR               { $$ = op1(POSTINCR, $1); }

stmt:
  | while {inloop++;} stmt  {--inloop; $$ = stat2(WHILE,$1,$3);}  
  | if stmt else stmt       { $$ = stat3(IF, $1, $2, $4); }  
  | if stmt                 { $$ = stat3(IF, $1, $2, NIL); }  
  | lbrace stmtlist rbrace  { $$ = $2; }

while:
  WHILE '(' pattern rparen   { $$ = notnull($3); }
Excerpts from a LEX analyzer

"++"
  { yylval.i = INCR; RET(INCR); } 
"--"
  { yylval.i = DECR; RET(DECR); } 

([0-9]+(\.|\.[0-9]*)\.[0-9]*|\.[0-9]+)([eE]([+-])([0-9]+)\)? {  
    yylval.cp = setsymtab(yytext, tostring(yytext),
                           atof(yytext), CON|NUM, symtab);
    RET(NUMBER); } 

while  { RET(WHILE); } 
for    { RET(FOR); } 
do     { RET(DO); } 
if     { RET(IF); } 
else   { RET(ELSE); } 
return { if (!infunc)
       ERROR "return not in function" SYNTAX;
       RET(RETURN); }

• 
  { RET(yylval.i = yytext[0]); /* everything else */ }
The whole process

- Grammar
  - YACC
  - y.tab.c parser
- Lexical rules
  - Lex (or other)
  - lex.yy.c analyzer
- C compiler
- Other C code
- a.out
AWK implementation

- source code is about 6500 lines of C and YACC
- compiles (almost) without change on Unix/Linux, Windows, Mac

- parse tree nodes (interior):
  ```c
  typedef struct Node {
      int type;  /* ARITH, ... */
      Node *next;
      Node *child[4];
  } Node;
  ```

- leaf nodes (values):
  ```c
  typedef struct Cell {
      int type;  /* VAR, FLD, ... */
      Cell *next;
      char *name;
      char *sval;  /* string value */
      double fval; /* numeric value */
      int state;  /* STR | NUM | ARR ... */
  } Cell;
  ```
Using Awk for testing RE code

- regular expression tests are described in a very small specialized language:

  ^a.$ ~ ax
  aa
  !~ xa
  aaa
  axy

- each test is converted into a command that exercises awk:

  echo 'ax' | awk '!/^a.$/' { print "bad" }

- illustrates
  - little languages
  - programs that write programs
  - mechanization
Unit testing

- code that exercises/tests small area of functionality
  - single method, function, ...
- helps make sure that code works and stays working
  - make sure small local things work so can build larger things on top
- very often used in "the real world"
  - e.g., can't check in code unless has tests and passes them
- often have tools to help write tests, run them automatically
  - e.g., JUnit

```c
struct {
    int yesno; char *re; char *text;
} tests[100] = {
    1, "x", "x",
    0, "x", "y",
    0, 0, 0
};

main() {
    for (int i = 0; tests[i].re != 0; i++) {
        if (match(tests[i].re, tests[i].text) != tests[i].yesno)
            printf("%d failed: %d [%s] [%s]\n", i,
                   tests[i].yesno, tests[i].re, tests[i].text);
    }
}
```
Lessons

• people use tools in unexpected, perverse ways
  – compiler writing: implementing languages and other tools
  – object language (programs generate Awk)
  – first programming language

• existence of a language encourages programs to generate it
  – machine generated inputs stress differently than people do

• mistakes are inevitable and hard to change
  – concatenation syntax
  – ambiguities, especially with >
  – function syntax
  – creeping featurism from user pressure
  – difficulty of changing a "standard"

• bugs last forever

"One thing [the language designer] should not do is to include untried ideas of his own."