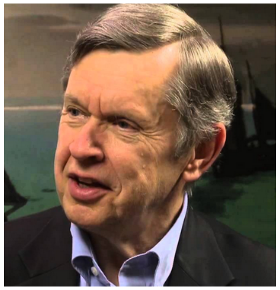
Lecture 3
AWK

Over-simplified history of programming languages

• 1940's	machine language	
• 1950's	assembly language	
• 1960's	high-level languages:	scripting languages:
	Algol, Fortran, Cobol, Basic	Snobol
• 1970's	systems programming: C	shell
• 1980's	object-oriented: C++	Awk
• 1990's	strongly-hyped: Java	Perl, Python, PHP, …
• 2000's	lookalike languages: C#	Javascript
• 2010's	retry: Go, Rust, Swift	Dart, Typescript, ...

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 }
```

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 }
END { for (i = NR; i > 0; i--) print x[i] }
```

reverse input by lines

```
{ for (i = 1; i <= NF; i++) w[$i]++ }
END { for (i in w) print i, w[i] }
```

word freq count

Associative Arrays

- array subscripts can have <u>any</u> 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

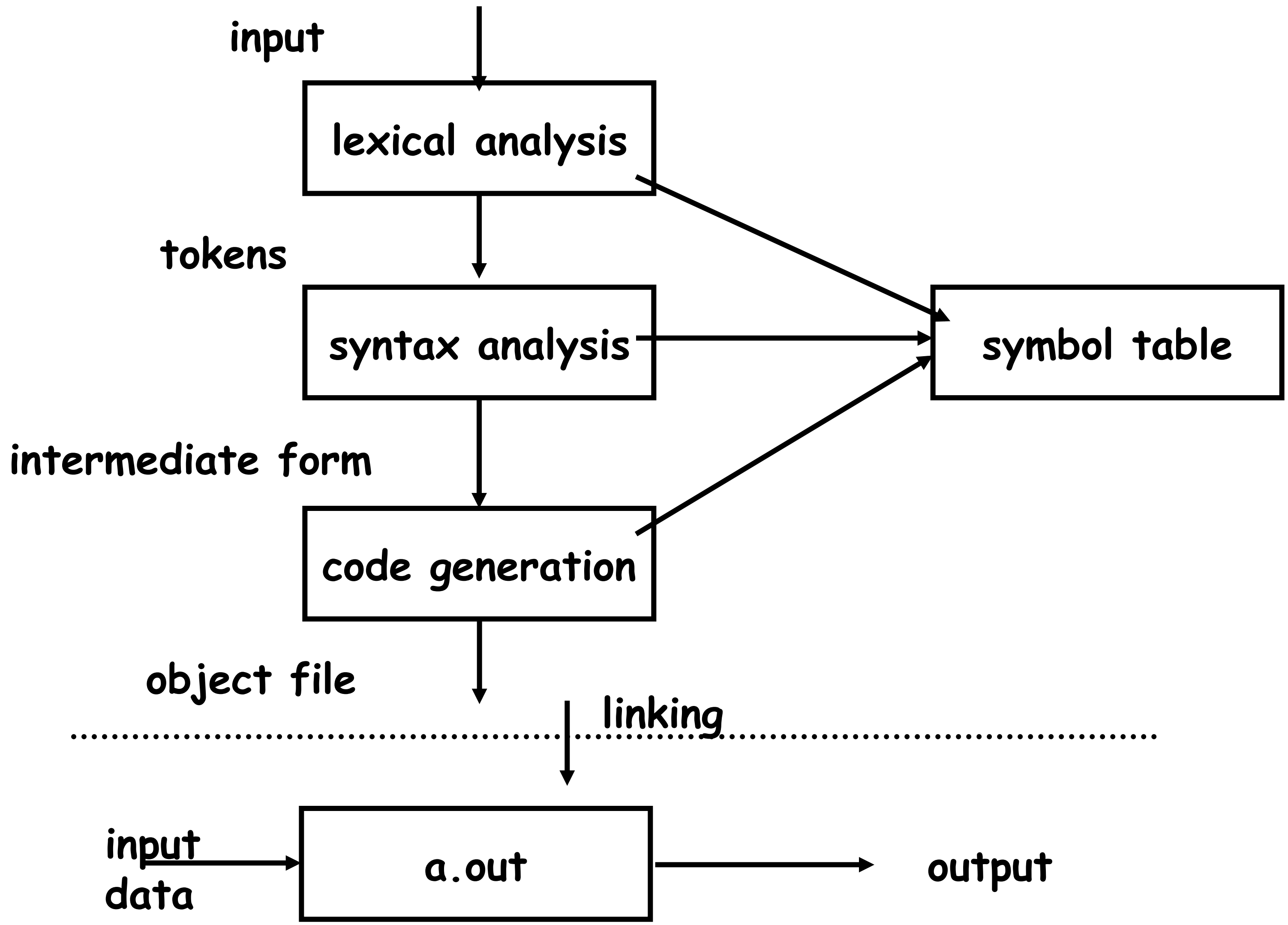
```
#!/bin/sh
# f - format text into 60-char lines

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

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

function println() {
    if (length(line) > 0)
        print line
    line = space = ""
}
' "$@"
```

COS 320 in 20 minutes

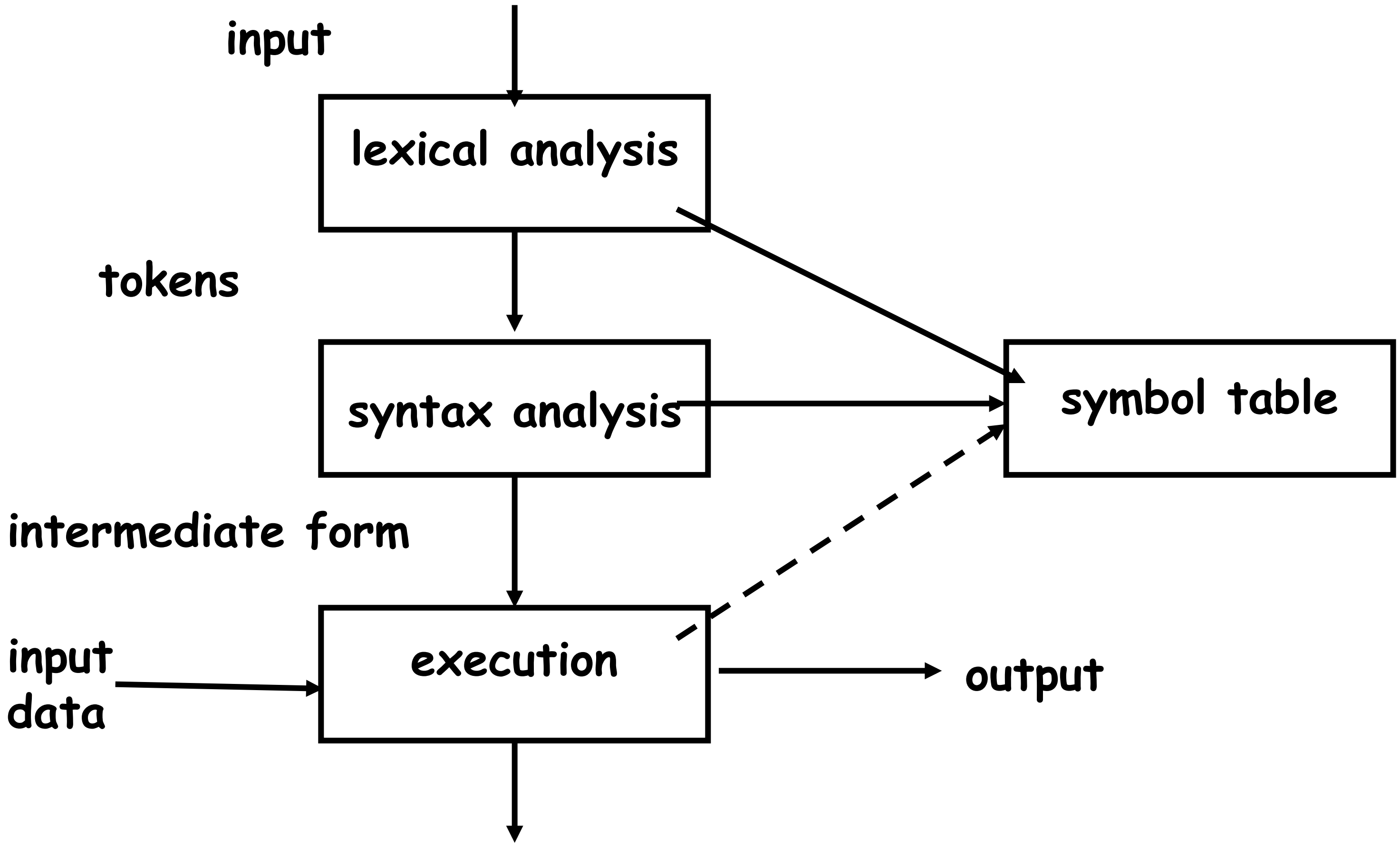
with apologies to David August and Zak Kincaid

- how does a compiler work?
- how does an interpreter work?
- syntax and semantics
- parsing and parse trees
- recursive descent parsing
- parser generator tools
- lexical analysis, generators
- testing

Anatomy of a compiler



Anatomy of an interpreter



Parsing by recursive descent

expr:	term l expr + term l expr - term
term:	factor l term * factor l term / factor
factor:	NUMBER l (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" (Steve Johnson, ~ 1972)
  - converts a grammar and semantic actions into a parser for that grammar

- LEX: lexical analyzer generator (Mike 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

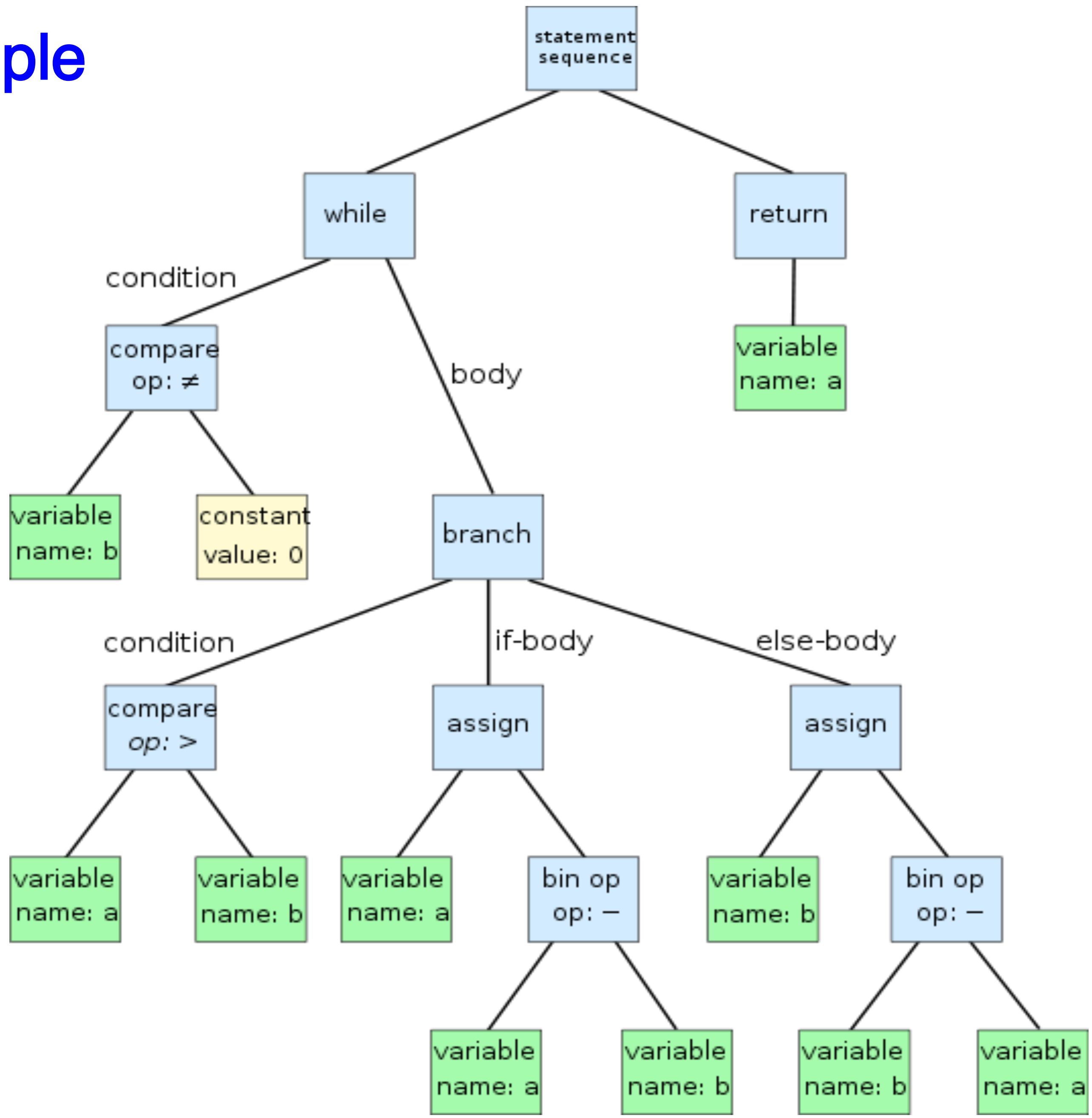
YACC overview

- 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)

Parse tree example

(from Wikipedia)

```
while b != 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* { create node(ADD, expr, expr }
  expression – *expression* { create node(SUB, expr, expr }
  ...

- 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

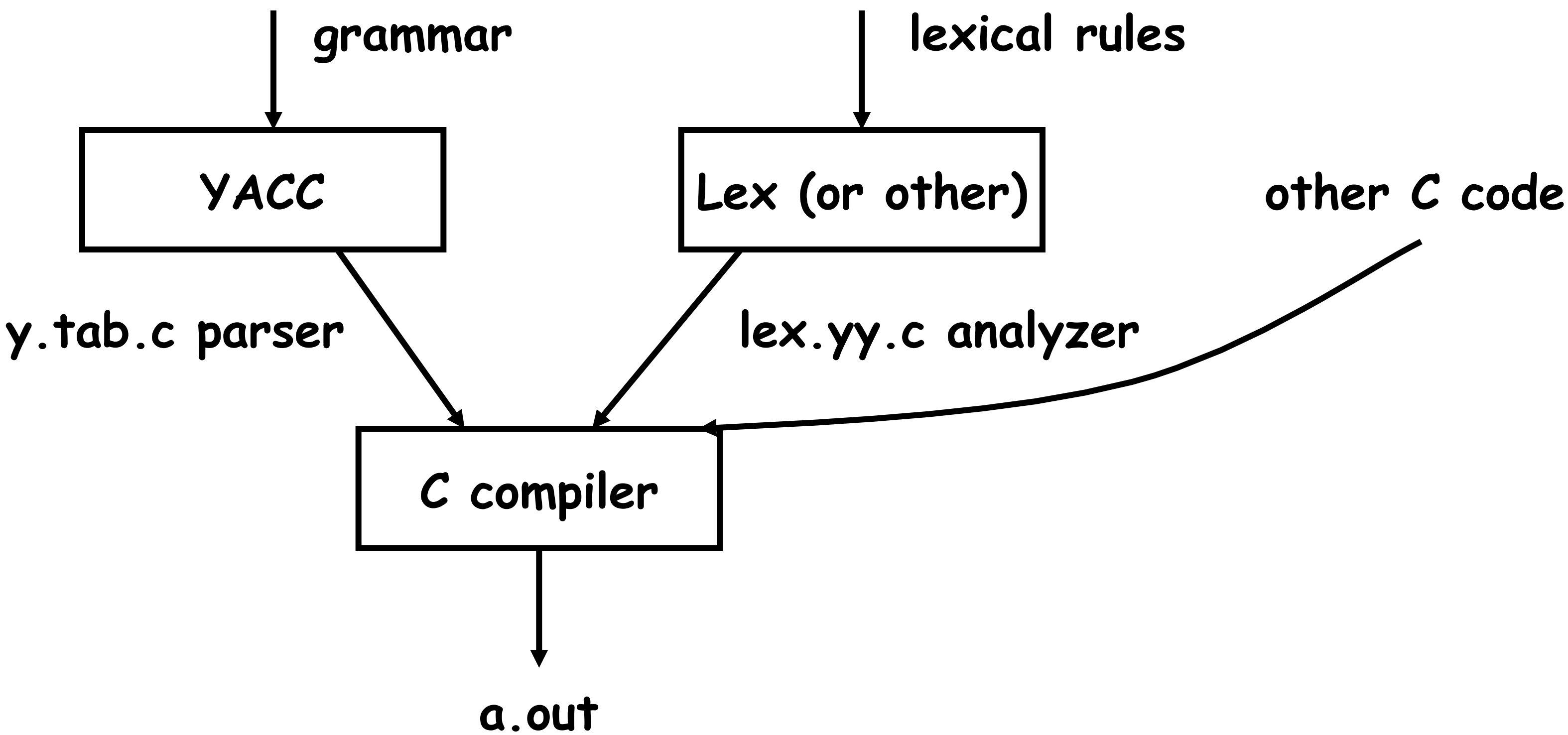
Excerpts from AWK grammar

```
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]+)([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



AWK implementation

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

- parse tree nodes (interior):

```
typedef struct Node {
    int   type;  /* ARITH, … */
    Node *next;
    Node *child[4];
} Node;
```

- leaf nodes (values):

```
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

```
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."

(C. A. R. Hoare, *Hints on Programming Language Design*, 1973)