Tutorial 1:
A Toy Language and Its Interpreter/Compiler

Based on *Lex and Yacc Tutorial* by T Niemann
[https://www.epaperpress.com/lexandyacc/index.html]

Niemann’s Files

LexAndYaccCode.zip

- calc1, calc2, calc3
  - The first two are calculators
  - calc3 is a toy language, with while, if-then-else, print, etc.
  - Use calc3 as the template for your compiler (if you will use C/C++)
- Three versions of calc3:
  - calc3a – the interpreter
  - calc3b – the compiler; the output is assembly code for a simple stack machine which Niemann didn’t implement, which I will provide
  - calc3g – to display the parse trees graphically
- I expanded the language a little bit:
  - c4
- The following slides are based on calc3

Example input

```
x = 0;
while (x < 3) {
     print x;
     x = x + 1;
}
```

Output of calc3a

```
0
1
2
```

Output of calc3b

```
     push     0
     pop      x
L000:
     push     x
     push     3
     compLT
     jz       L001
     push     x
     print
     push     x
     push     1
     add
     pop      x
     jmp      L000
L001:
```

They all share the same scanner and parser files:

calc3.h

calc3.l

calc3.y

But each has its own backend (calc3a.c, calc3b.c, calc3g.c)

Output of calc3g

```
Graph 0:

    [=]
     |
  |----|
  |    |
id(X) c(0)

Graph 1:

                  while
                    |
    |-----------------|
    |                 |
   [<]               [;]
    |                 |
  |----|       |-----------|
  |    |       |           |
id(X) c(3)   print        [=]
               |           |
               |     |-------|
               |     |       |
             id(X) id(X)    [+]
                             |
                          |----|
                          |    |
                        id(X) c(1)
```

A statement (stmt)

opr

id

con

3 types of tree nodes: con, id, opr (of type nodeType)
con and id are leaf nodes;
opr is non-leaf node

The following slides describe the common files:

calc3.h (header file)

calc3.l (scanner)

calc3.y (parser)

calc3.h

```
typedef enum { typeCon, typeId, typeOpr } nodeEnum;

/* constants */
typedef struct {
    int value;                  /* value of constant */
} conNodeType;

/* identifiers */
typedef struct {
    int i;                      /* subscript to sym array */
} idNodeType;

/* operators */
typedef struct {
    int oper;                   /* operator */
    int nops;                   /* number of operands */
    struct nodeTypeTag *op[1];  /* operands (expandable) */
} oprNodeType;

typedef struct nodeTypeTag {
    nodeEnum type;              /* type of node */

    /* union must be last entry in nodeType */
    /* because operNodeType may dynamically increase */
    union {
        conNodeType con;        /* constants */
        idNodeType id;          /* identifiers */
        oprNodeType opr;        /* operators */
    };
} nodeType;

extern int sym[26];
```

= INTEGERs

= VARIABLEs

Note: not the value contained in the variable, but one of a, b, c, …, z

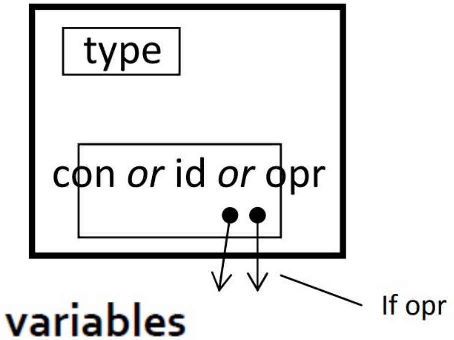
Non-leaf node of parse tree

An array of pointers to subtree nodes; only one is defined here; it will be expanded to as many subtree nodes as needed later on (via C's variadic function)

= **nodeType** which hasn't been declared

A tree node

(of type nodeType)



The symbol table for the single-letter variables

Only used by the interpreter; the compiler generates a, b, c, …, z which are mapped to the stack machine's registers with the same names

calc3.l

Two types of tokens (terminals) that have a value (yylval): VARIABLE (or id) and INTEGER (or con)

Non-zero numbers starting with 0 not allowed

```
%{
#include <stdlib.h>
#include "calc3.h"
#include "y.tab.h"
void yyerror(char *);
%}

%%

[a-z]       {
                yylval.sIndex = *yytext - 'a';
                return VARIABLE;
            }

0           {
                yylval.iValue = atoi(yytext);
                return INTEGER;
            }

[1-9][0-9]* {
                yylval.iValue = atoi(yytext);
                return INTEGER;
            }

[-()<>=+*/;{}.] {
                return *yytext;
             }
```

```
">="            return GE;
"<="            return LE;
"=="            return EQ;
"!="            return NE;
"while"         return WHILE;
"if"            return IF;
"else"          return ELSE;
"print"         return PRINT;

[ \t\n]+        ; /* ignore whitespace */

.               yyerror("Unknown character");
%%
int yywrap(void) {
    return 1;
}
```

calc3.y (Section 1)

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include "calc3.h"

/* prototypes */
nodeType *opr(int oper, int nops, ...);
nodeType *id(int i);
nodeType *con(int value);
void freeNode(nodeType *p);
int ex(nodeType *p);
int yylex(void);

void yyerror(char *s);
int sym[26];                    /* symbol table */
%}

%union {
    int iValue;                 /* integer value */
    char sIndex;                /* symbol table index */
    nodeType *nPtr;             /* node pointer */
};
```

A C variadic function!

Functions to construct tree nodes (3 different types)

A bison command for declaring yylval (default is int)

Accessed like C's union: yylval.iValue, yylval.sIndex, …; nPtr is a pointer to an inner tree node, an address

To inform bison that INTEGER is of type iValue, etc.

```
%token <iValue> INTEGER
%token <sIndex> VARIABLE
%token WHILE IF PRINT
%nonassoc IFX
%nonassoc ELSE

%left GE LE EQ NE '>' '<'
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS

%type <nPtr> stmt expr stmt_list
```

low

precedence

high

Solving dangling else (previous lectures refer)

These are non-terminals

calc3.y (Section 2)

In the outermost scope, every stmt is a tree

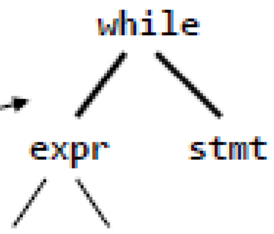
To "walk" the tree rooted at $2 (stmt); walk means execute (interpreter) or generate-code-for (compiler)

arguments following (=
branches

ELSE has higher precedence than "IF … then"

```
%%
program:
        function                { exit(0); }
        ;

function:
          function stmt         { ex($2); freeNode($2); }
        | /* NULL */
        ;

stmt:
          ';'                            { $$ = opr(';', 2, NULL, NULL); }
        | expr ';'                       { $$ = $1; }
        | PRINT expr ';'                 { $$ = opr(PRINT, 1, $2); }
        | VARIABLE '=' expr ';'          { $$ = opr('=', 2, id($1), $3); }
        | WHILE '(' expr ')' stmt        { $$ = opr(WHILE, 2, $3, $5); }
        | IF '(' expr ')' stmt %prec IFX { $$ = opr(IF, 2, $3, $5); }
        | IF '(' expr ')' stmt ELSE stmt { $$ = opr(IF, 3, $3, $5, $7); }
        | '{' stmt_list '}'              { $$ = $2; }
        ;

stmt_list:
          stmt                  { $$ = $1; }
        | stmt_list stmt        { $$ = opr(';', 2, $1, $2); }
        ;

expr:
          INTEGER               { $$ = con($1); }
        | VARIABLE              { $$ = id($1); }
        | '-' expr %prec UMINUS { $$ = opr(UMINUS, 1, $2); }
        | expr '+' expr         { $$ = opr('+', 2, $1, $3); }
        ...
```

Null statement



The "value" returned is actually a pointer to a newly created tree node (nodeType *)

constant

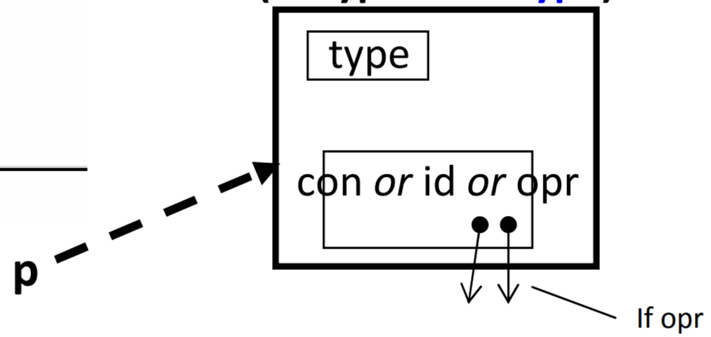
variable

calc3.y (Section 3)$_1$

Defining the 3 node construction functions:

```
%%

#define SIZEOF_NODETYPE ((char *)&p->con - (char *)p)

nodeType *con(int value) {
    nodeType *p;
    size_t nodeSize;

    /* allocate node */
    nodeSize = SIZEOF_NODETYPE + sizeof(conNodeType);
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");

    /* copy information */
    p->type = typeCon;
    p->con.value = value;

    return p;
}

nodeType *id(int i) { ... }
```

A tree node
(of type nodeType)



calc3.h:

```
typedef enum { typeCon, typeI

/* constants */
typedef struct {
    int value;
} conNodeType;

/* identifiers */
typedef struct {
    int i;
} idNodeType;

/* operators */
typedef struct {
    int oper;
    int nops;
    struct nodeTypeTag *op[1]
} oprNodeType;

typedef struct nodeTypeTag {
    nodeEnum type;

    /* union must be last ent
    /* because operNodeType m
    union {
        conNodeType con;
        idNodeType id;
        oprNodeType opr;
    };
} nodeType;

extern int sym[26];
```

calc3.y (Section 3)$_2$

```c
nodeType *opr(int oper, int nops, ...) {
    va_list ap;
    nodeType *p;
    size_t nodeSize;
    int i;

    /* allocate node */
    nodeSize = SIZEOF_NODETYPE + sizeof(oprNodeType) +
        (nops - 1) * sizeof(nodeType*);
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");

    /* copy information */
    p->type = typeOpr;
    p->opr.oper = oper;
    p->opr.nops = nops;
    va_start(ap, nops);
    for (i = 0; i < nops; i++)
        p->opr.op[i] = va_arg(ap, nodeType*);
    va_end(ap);
    return p;
}

void freeNode(nodeType *p) { ... }
void yyerror(char *s) { ... }

int main(void) {
    yyparse();
    return 0;
}
```

One can use an array or a linked list instead of a variadic function

oprNodeType
oper = WHILE
nops = 2
op[0] = &(expr)
op[1] = &(stmt)

```
Graph 0:

    [=]
     |
   |----|
   |    |
 id(X) c(0)

Graph 1:

                 while
                   |
     |----------------|
     |                |
    [<]              [;]
     |                |
   |----|     |----------|
   |    |     |          |
 id(X) c(3) print       [=]
              |          |
              |      |-------|
              |      |       |
            id(X)  id(X)    [+]
                             |
                          |----|
                          |    |
                        id(X) c(1)
```

```c
x = 0;
while (x < 3) {
    print x;
    x = x + 1;
}
```

An Example of C Variadic Function

```
#include <stdarg.h>
#include <stdio.h>

void sum(char *, int, ...);

int main(void)
{
  sum("The sum of 10+15+13 is %d.\n", 3, 10, 15, 13);
  return 0;
}

void sum(char *string, int num_args, ...)
{
  int sum = 0;
  va_list ap;
  int loop;

  va_start(ap, num_args);

  for (loop=0; loop < num_args; loop++)
    sum += va_arg(ap, int);

  printf(string, sum);

  va_end(ap);
}
```

Caller

callee

Variadic functions are convenient for the caller, but a bit clumsy for the function implementer (callee)

The following slides describe the backends:

calc3a.c (interpreter)

calc3b.c (compiler)

calc3g.c (parse tree visualizer) - skipped

Essentially, they implement the tree walking function, ex(); for the interpreter, it executes every stmt on the spot; for the compiler, it generates assembly code

calc3a.c – the Interpreter

Walk and execute a (sub)tree rooted at *p

```
#include <stdio.h>
#include "calc3.h"
#include "y.tab.h"

int ex(nodeType *p) {
    if (!p) return 0;
    switch(p->type) {
    case typeCon:       return p->con.value;
    case typeId:        return sym[p->id.i];
    case typeOpr:
        switch(p->opr.oper) {
        case WHILE:     while(ex(p->opr.op[0]))
                            ex(p->opr.op[1]); return 0;
        case IF:        if (ex(p->opr.op[0]))
                            ex(p->opr.op[1]);
                        else if (p->opr.nops > 2)
                            ex(p->opr.op[2]);
                        return 0;
        case PRINT:     printf("%d\n", ex(p->opr.op[0])); return 0;
        case ';':       ex(p->opr.op[0]); return ex(p->opr.op[1]);
        case '=':       return sym[p->opr.op[0]->id.i] = ex(p->opr.op[1]);
        case UMINUS:    return -ex(p->opr.op[0]);
        case '+':       return ex(p->opr.op[0]) + ex(p->opr.op[1]);
        case '-':       return ex(p->opr.op[0]) - ex(p->opr.op[1]);
        case '*':       return ex(p->opr.op[0]) * ex(p->opr.op[1]);
        case '/':       return ex(p->opr.op[0]) / ex(p->opr.op[1]);
        ... < > GE LE NE EQ
        }
    }
    return 0;
}
```

Recall our discussion of interpreter vs. compiler in the first 2 lectures:

- Depth-first traversal of a subtree
- Recursive function calls based on the nodes encountered during the traversal

```
        push    0
        pop     x
L000:
        push    x
        push    3
        compLT
        jz      L001
        push    x
        print
        push    x
        push    1
        add
        pop     x
        jmp     L000
L001:
```

Comparison: The while loop by the compiler version

calc3b.c – the Compiler

```
int ex(nodeType *p) {
    int lbl1, lbl2;

    if (!p) return 0;
    switch(p->type) {
    case typeCon:
        printf("\tpush\t%d\n", p->con.value);
        break;
    case typeId:
        printf("\tpush\t%c\n", p->id.i + 'a');
        break;
    case typeOpr:
        switch(p->opr.oper) {
        case WHILE:
            printf("L%03d:\n", lbl1 = lbl++);
            ex(p->opr.op[0]);
            printf("\tjz\tL%03d\n", lbl2 = lbl++);
            ex(p->opr.op[1]);
            printf("\tjmp\tL%03d\n", lbl1);
            printf("L%03d:\n", lbl2);
            break;
        case IF:
                ...
        case PRINT:
            ex(p->opr.op[0]);
            printf("\tprint\n");
                ...
```

static variable initialized to 0

```
x = 0;
while (x < 3) {
  print x;
  x = x + 1;
}
```

```
        push    0
        pop     x
L000:
        push    x
        push    3
        compLT
        jz      L001
        push    x
        print
        push    x
        push    1
        add
        pop     x
        jmp     L000
L001:
```

sas

A very simple stack machine and its assembly instructors

Feel free to change "sas.y"

The machine has a stack (the size of which is up to you), 26 registers: a .. z, and there can be up to 1000 labels (L000 .. L999) in a program.

N: an integer variable (one of a .. z) or an integer literal

R: a register name (one of a .. z)

push N stack[++top] = N

pop R R = stack[top--]

compLT if stack[top-1] < stack[top] then stack[--top] = 1 else 0

compGT if stack[top-1] > stack[top] then stack[--top] = 1 else 0

compGE if stack[top-1] >= stack[top] then stack[--top] = 1 else 0

compLE if stack[top-1] <= stack[top] then stack[--top] = 1 else 0

compNE if stack[top-1] != stack[top] then stack[--top] = 1 else 0

compEQ if stack[top-1] == stack[top] then stack[--top] = 1 else 0

print print stack[top]

read stack[++top] = input an integer

add X = stack[top-1] + stack[top]; stack[--top] = X

sub X = stack[top-1] - stack[top]; stack[--top] = X

mul X = stack[top-1] * stack[top]; stack[--top] = X

div X = stack[top-1] / stack[top]; stack[--top] = X

neg stack[top] = -stack[top]

and X = stack[top-1] && stack[top]; stack[--top] = X

or X = stack[top-1] || stack[top]; stack[--top] = X

jz Lxxx if (stack[top--] == 0) then jump to label **Lxxx**

jmp Lxxx jump to label **Lxxx**

top: the stack pointer, pointing at the topmost element of the stack

Implemented by sas.l + sas.y (sas = simple assembler)

Note: these instructions are "destructive" – the two operands are replaced by the result; similarly, for the add, sub, …

"and" "or" are implemented in the stack machine in the same fashion as the arithmetic operators, which makes complex logical expressions very easy to compile into assembly code; otherwise, it would be quite difficult (we'll see in the near future)

This one is destructive too

c4

- Features I added:
  - Comments (// …)
  - Input operator (read x)
  - Logical operators (&& for AND, and || for OR)
  - FOR-loop
- Common files: calc3.h (the same one), c4.l (scanner), and c4.y (parser)
- Interpreter: c4i.c
- Compiler: c4c.c
- Assembler and simulator (which is an interpreter): sas.l, sas.y

You might want to call it a “virtual machine”

An Example & it's A.L. Output

```
// FACTORIAL(x)

read x;
if (x == 0)
  print 1;
else
  if (x >= 1 && x <= 12) { // 13! is too much for int
    f = 1;
    while (x > 1) {
      f = f * + x;
      x = x - 1;
    }
    print f;
  }
```

$ c4c fact.sc >fact.sas
$ sas fact.sas

```
        read
        pop     x
        push    x
        push    0
        compEQ
        jz      L000
        push    1
        print
        jmp     L001
L000:
        push    x
        push    1
        compGE
        push    x
        push    12
        compLE
        and
        jz      L002
        push    1
        pop     f
L003:
        push    x
        push    1
        compGT
        jz      L004
        push    f
        push    x
        mul
        pop     f
        push    x
        push    1
        sub
        pop     x
        jmp     L003
L004:
        push    f
        print
L002:
L001:
```

The readme File for c4.zip

```
A simple interactive calculator:
================================

cal - integers only [to make: make cal]
calx - real numbers, power (^) [make calx]

A simple calculator language (sc) - interpreter and compiler
============================================================

c4i - interpreter [make c4i]
c4c - compiler -> [make c4c]
sas - the assembler for a simulated stack machine [make sas]

Example programs:

for.sc - a double for loop
fact.sc - factorial

To run:

c4i fact.sc
c4c fact.sc >fact.sas
sas fact.sas

c4i for.sc
c4c for.sc >for.sas
sas for.sas
```