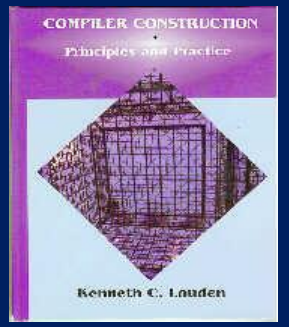
CS 445: Assignment 1

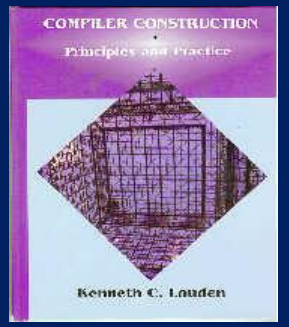
Just enough information to be dangerous!

Dr. BC



- https://webpages.uidaho.edu/drbc/
- https://canvas.uidaho.edu/
- Textbook: http://grsotudeh.ir/compiler/compiler_ebooks/compiler-construction-principles-and-practice-k-c-louden-pws-1997-cmp-2002-592s.pdf
- Unix server: cs-445.cs.uidaho.edu
- Unix shared directory: /y/shared/Engineering/cs-drbc/cs445

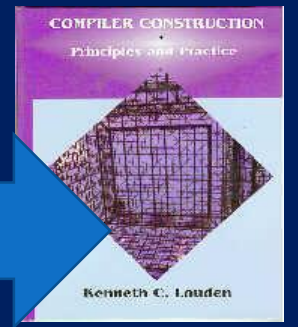
Assignment



Build a compiler for C-

(C- is a bit different each semester)

Process of Compiling



We will focus here

Preprocessor

Scanner → Tokens (Regular Expressions)

Parser → Abstract Syntax Tree (AST)/ Context Free Grammar (CFG)

Semantic analyzer → Annotated AST

Intermediate code generator → Intermediate Representation (IR)

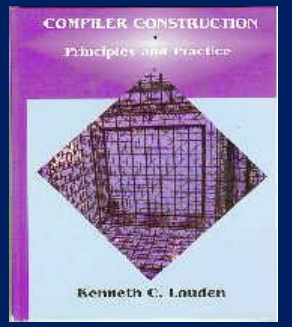
Code optimization

Code generator

Code optimization

Assignment

2,000-5,000 lines of code
(Plan on spending LOTS of time programing)



Note: Each assignment must be complete before you move to the next one.

Assignment 1
Use a combination of Flex and Bison code to build and drive a scanner for the C- programming language.

Assignment 2
Modify the "driver" of the last assignment to produce a parser for C-. Your parser will construct the abstract syntax tree (AST) corresponding to the input C- program.
1. Build a C- recognizer
2. Extend your C- recognizer to add the AST
3. Add user interface options

Assignment 3
Type expressions in the abstract syntax tree (AST) and start to perform semantic analysis and error generation.

Assignment 4
Continue semantic analysis and error generation.
Also add I/O runtime library support.

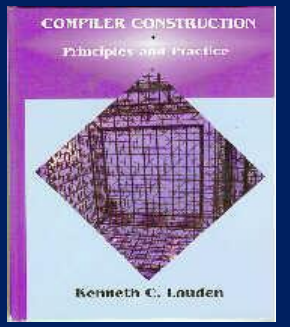
Assignment 5
Make all error and warning messages have the same format and add modifications to keep syntax errors from halting syntactic analysis in the compiler

Assignment 6
Preparing to generate code. Your compiler will need to compute the scope, size, and location of each symbol in an input program. You will add a new -M option to your compiler to print this information that your compiler has computed.

Note: Ensure that your code runs on the Uidaho server.

Assignment

Appendix B: Tiny Compiler Listing

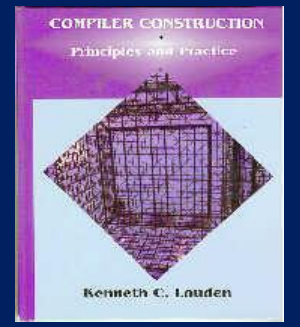


Assignment 1
Use a combination of Flex and Bison code to build and drive a scanner for the C- programming language.

a compiler for C-

(C- is a bit different each semester)

Process of Compiling



Preprocessor

flex (Compiled Second) → Scanner → tiny.l → lex.yy.c

includes

bison (Compiled first) → Parser → tiny.y → tiny.tab.h & tiny.tab.c

Semantic analyzer

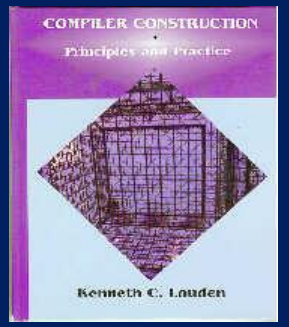
Intermediate code generator

Code optimization

Code generator

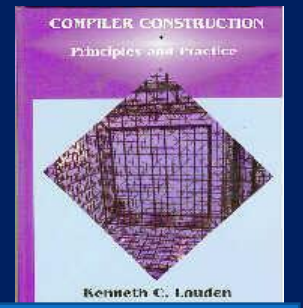
Code optimization

TINY C



- Much smaller than C-
- So we have a lot more to do than this than this tiny C compiler.
- But it shows you some of the organization and some of your options.
  - In here they do a hand constructed lexer
  - They also use a yak like flexor. So here it's all laid out for you.

Tiny Machine (TM)

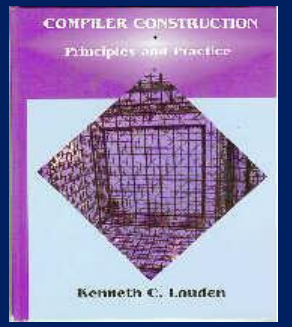


8 registers			
0	**AC**	**Accumulator**	**15**
1	AC1	Accumulator	29
2	R2		0
3	R3		0
4	R4		0
5	FP	Frame Pointer	26
6	GP	Global Area Pointer	32
7	PC	Program Counter	16

Data Memory			
32	**0**	**main**	**ret-addr**
31	6	main	st-val
30	10	main	st-val
29	15	main	st-addr
28	29	main	ctrl-link
27	32	main	ctrl-link
26	32	main	ctrl-link
25	78	gcd	ret-addr
24	15	main	arg
…			

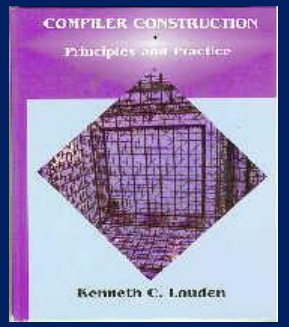
Instruction Memory					
…					
3	LDA	5	0	5	set new frame
4	LDA	0	1	7	load AC with return address
5	LDA	7	45	7	jump to function main
6	HALT	0	0	0	
7	ST	0	-1	5	Store return –in-ret-addr-
8	IN	0	0	0	input
.. .					

2 The Grammar



1. *program* → *stmt-seq*
2. *stmt-seq* → *stmt-seq* ; *stmt* | *stmt*
3. *stmt* → *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*
4. *if-stmt* → **if** *exp* **then** *stmt-seq* **end** | **if** *exp* **then** *stmt-seq* **else** *stmt-seq* **end**
5. *repeat-stmt* → **repeat** *stmt-seq* **until** *exp*
6. *assign-stmt* → **ID** := *exp*
7. *read-stmt* → **read ID**
8. *write-stmt* → **write** *exp*
9. *exp* → *simple-exp* < *simple-exp* | *simple-exp* = *simple-exp* | *simple-exp*
10. *simple-exp* → *simple-exp* + *term* | *simple-exp* − *term* | *term*
11. *term* → *term* ∗ *factor* | *term* / *factor* | *factor*
12. *factor* → (*exp*) | **NUM** | **ID**

Flex: The four rules for matching tokens:



1. Characters are only matched once. That is, each character is matched by only one pattern.
2. Longest matching string gets matched first. That is, if one pattern matches "zin" and a later pattern matches "zinjanthropus" the second pattern is the one that matches.
3. If same length of matching string then first rule matches.
4. If no pattern matches then the character is printed to standard output.

Tiny 1 Introduction

For the grammar that follows, here are the types of the various elements by type font:

- **Keywords are in this type font**
- **TOKEN CLASSES ARE IN THIS TYPE FONT.**
- <Nonterminals are in this type font>.

1.1 Some Token Definitions

letter = a | . . . | z | A | . . . | Z

digit = 0 | . . . | 9

ID = letter+

NUM = digit+

Also note that white space is ignored except that it must separate **ID**'s, **NUM**'s, and keywords.

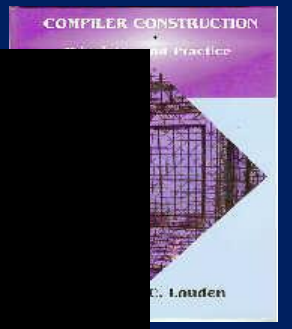
Tiny.l


```
/****************************************/
/* File: tiny.l                         */
/* Lex specification for TINY           */
/* Compiler Construction: Principles and
Practice   */
/* Kenneth C. Louden                    */
/****************************************/

%{
#include "globals.h"
#include "util.h"
#include "scan.h"
/* lexeme of identifier or reserved word */
char tokenString[MAXTOKENLEN+1];
%}

digit       [0-9]
number      {digit}+
letter      [a-zA-Z]
identifier  {letter}+
newline     \n
whitespace  [ \t]+

%%
```

Tiny 2 The Grammar

1. <program> ::= <stmt-seq>
2. <stmt-seq> ::= <stmt-seq> ; <stmt>
   | <stmt>
3. <stmt> ::= <if-stmt> | <repeat-stmt>
   | <assign-stmt> | <read-stmt>
   | <write-stmt>
4. <if-stmt> ::= **if** <exp> **then** <stmt-seq> **end**
   | **if** <exp> **then** <stmt-seq>
   else <stmt-seq> **end**
5. <repeat-stmt> ::= **repeat** <stmt-seq> **until** <exp>
6. <assign-stmt> ::= **ID** := <exp>
7. <read-stmt> ::= **read ID**
8. <write-stmt> ::= **write** <exp>
9. <exp> ::= <simple-exp> < <simple-exp>
   | <simple-exp> = <simple-exp>
   | <simple-exp>
10. <simple-exp> ::= <simple-exp> + <term>
    | <simple-exp> - <term>
    | <term>
11. <term> ::= <term> * <factor>
    | <term> / <factor>
    | <factor>
12. <factor> ::= (<exp>) | **NUM** | **ID**

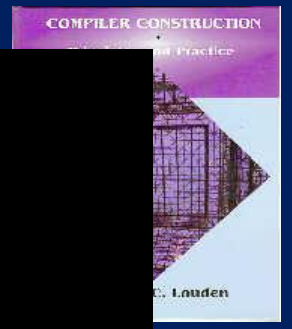
Tiny.l




```
"if"          {return IF;}
"then"        {return THEN;}
"else"        {return ELSE;}
"end"         {return END;}
"repeat"      {return REPEAT;}
"until"       {return UNTIL;}
"read"        {return READ;}
"write"       {return WRITE;}
":="          {return ASSIGN;}
"="           {return EQ;}
"<"           {return LT;}
"+"           {return PLUS;}
"-"           {return MINUS;}
"*"           {return TIMES;}
"/"           {return OVER;}
"("           {return LPAREN;}
")"           {return RPAREN;}
";"           {return SEMI;}
{number}              {return NUM;}
{identifier}          {return ID;}
{newline}             {lineno++;}
{whitespace}          {/* skip whitespace */}
"{"           { char c;
              do
              { c = input();
                if (c == EOF) break;
                if (c == '\n') lineno++;
              } while (c != '}');
            }
.             {return ERROR;}

%%
```

Tiny

Tiny.l




```
{newline}     {lineno++;}
{whitespace}                {/* skip whitespace */}
"{"           { char c;
                do
                { c = input();
                  if (c == EOF) break;
                  if (c == '\n') lineno++;
                } while (c != '}');
              }
.             {return ERROR;}

%%
TokenType getToken(void)
{ static int firstTime = TRUE;
  TokenType currentToken;
  if (firstTime)
  { firstTime = FALSE;
    lineno++;
    yyin = source;
    yyout = listing;
  }
  currentToken = yylex();
  strncpy(tokenString,yytext,MAXTOKENLEN);
  if (TraceScan) {
    fprintf(listing,"\t%d: ",lineno);
    printToken(currentToken,tokenString);
  }
  return currentToken;
}
```

Tiny.l

```
"if"          {return IF;}
"then"        {return THEN;}
"else"        {return ELSE;}
"end"         {return END;}
"repeat"      {return REPEAT;}
"until"       {return UNTIL;}
"read"        {return READ;}
"write"       {return WRITE;}
":="          {return ASSIGN;}
"="           {return EQ;}
"<"           {return LT;}
"+"           {return PLUS;}
"-"           {return MINUS;}
"*"           {return TIMES;}
"/"           {return OVER;}
"("           {return LPAREN;}
")"           {return RPAREN;}
";"           {return SEMI;}
{number}              {return NUM;}
{identifier}          {return ID;}
{newline}             {lineno++;}
{whitespace}          {/* skip whitespace */}
"{"           { char c;
              do
              { c = input();
                if (c == EOF) break;
                if (c == '\n') lineno++;
              } while (c != '}');
            }
.             {return ERROR;}

%%
```

Tiny.y


```
/****************************************/
/* File: tiny.y                         */
/* The TINY Yacc/Bison specification file        */
/* Compiler Construction: Principles and Practice   */
/* Kenneth C. Louden                            */
/****************************************/
%{
#define YYPARSER /* distinguishes Yacc output from
other code files */

#include "globals.h"
#include "util.h"
#include "scan.h"
#include "parse.h"

#define YYSTYPE TreeNode *
static char * savedName; /* for use in assignments */
static int savedLineNo;  /* ditto */
static TreeNode * savedTree; /* stores syntax tree for
later return */

%}

%token IF THEN ELSE END REPEAT UNTIL READ WRITE
%token ID NUM
%token ASSIGN EQ LT PLUS MINUS TIMES OVER
LPAREN RPAREN SEMI
%token ERROR
```

Tiny 2 The Grammar

1. <program> ::= <stmt-seq>
2. <stmt-seq> ::= <stmt-seq> ; <stmt>
   | <stmt>
3. <stmt> ::= <if-stmt> | <repeat-stmt>
   | <assign-stmt> | <read-stmt>
   | <write-stmt>
4. <if-stmt> ::= **if** <exp> **then** <stmt-seq> **end**
   | **if** <exp> **then** <stmt-seq>
   else <stmt-seq> **end**
5. <repeat-stmt> ::= **repeat** <stmt-seq> **until** <exp>
6. <assign-stmt> ::= **ID** := <exp>
7. <read-stmt> ::= **read ID**
8. <write-stmt> ::= **write** <exp>
9. <exp> ::= <simple-exp> < <simple-exp>
   | <simple-exp> = <simple-exp>
   | <simple-exp>
10. <simple-exp> ::= <simple-exp> + <term>
    | <simple-exp> - <term>
    | <term>
11. <term> ::= <term> * <factor>
    | <term> / <factor>
    | <factor>
12. <factor> ::= (<exp>) | **NUM** | **ID**

Tiny.y

COMPILER CONSTRUCTION

```
%% /* Grammar for TINY */

program     : stmt_seq
            { savedTree = $1;}
          ;
stmt_seq    : stmt_seq SEMI stmt
            { YYSTYPE t = $1;
              if (t != NULL) {
                 while (t->sibling != NULL) t = t->sibling;
                 t->sibling = $3;
                 $$ = $1;
              }
              else {
                 $$ = $3;
              }
            }
          | stmt  { $$ = $1; }
          ;
```

Tiny 2 The Grammar

1. <program> ::= <stmt-seq>
2. <stmt-seq> ::= <stmt-seq> ; <stmt>
   | <stmt>
3. <stmt> ::= <if-stmt> | <repeat-stmt>
   | <assign-stmt> | <read-stmt>
   | <write-stmt>
4. <if-stmt> ::= **if** <exp> **then** <stmt-seq> **end**
   | **if** <exp> **then** <stmt-seq>
   else <stmt-seq> **end**
5. <repeat-stmt> ::= **repeat** <stmt-seq> **until** <exp>
6. <assign-stmt> ::= **ID** := <exp>
7. <read-stmt> ::= **read ID**
8. <write-stmt> ::= **write** <exp>
9. <exp> ::= <simple-exp> < <simple-exp>
   | <simple-exp> = <simple-exp>
   | <simple-exp>
10. <simple-exp> ::= <simple-exp> + <term>
    | <simple-exp> - <term>
    | <term>
11. <term> ::= <term> * <factor>
    | <term> / <factor>
    | <factor>
12. <factor> ::= (<exp>) | **NUM** | **ID**

Tiny.y

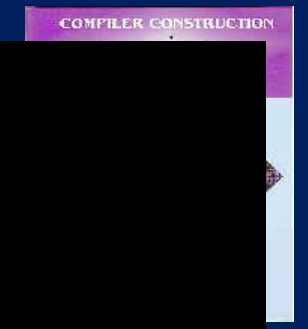
COMPILER CONSTRUCTION

```
stmt      : if_stmt { $$ = $1; }
          | repeat_stmt { $$ = $1; }
          | assign_stmt { $$ = $1; }
          | read_stmt { $$ = $1; }
          | write_stmt { $$ = $1; }
          | error  { $$ = NULL; }
          ;
```

Tiny 2 The Grammar

1. <program> ::= <stmt-seq>
2. <stmt-seq> ::= <stmt-seq> ; <stmt>
   | <stmt>
3. <stmt> ::= <if-stmt> | <repeat-stmt>
   | <assign-stmt> | <read-stmt>
   | <write-stmt>
4. <if-stmt> ::= **if** <exp> **then** <stmt-seq> **end**
   | **if** <exp> **then** <stmt-seq>
   else <stmt-seq> **end**
5. <repeat-stmt> ::= **repeat** <stmt-seq> **until** <exp>
6. <assign-stmt> ::= **ID** := <exp>
7. <read-stmt> ::= **read ID**
8. <write-stmt> ::= **write** <exp>
9. <exp> ::= <simple-exp> < <simple-exp>
   | <simple-exp> = <simple-exp>
   | <simple-exp>
10. <simple-exp> ::= <simple-exp> + <term>
    | <simple-exp> - <term>
    | <term>
11. <term> ::= <term> * <factor>
    | <term> / <factor>
    | <factor>
12. <factor> ::= (<exp>) | **NUM** | **ID**

Tiny.y




```
if_stmt    : IF exp THEN stmt_seq END
             { $$ = newStmtNode(IfK);
               $$->child[0] = $2;
               $$->child[1] = $4;
             }
         | IF exp THEN stmt_seq ELSE stmt_seq END
             { $$ = newStmtNode(IfK);
               $$->child[0] = $2;
               $$->child[1] = $4;
               $$->child[2] = $6;
             }
         ;
repeat_stmt : REPEAT stmt_seq UNTIL exp
             { $$ = newStmtNode(RepeatK);
               $$->child[0] = $2;
               $$->child[1] = $4;
             }
         ;
```

Tiny 2 The Grammar

1. <program> ::= <stmt-seq>
2. <stmt-seq> ::= <stmt-seq> ; <stmt>
   | <stmt>
3. <stmt> ::= <if-stmt> | <repeat-stmt>
   | <assign-stmt> | <read-stmt>
   | <write-stmt>
4. <if-stmt> ::= **if** <exp> **then** <stmt-seq> **end**
   | **if** <exp> **then** <stmt-seq>
   else <stmt-seq> **end**
5. <repeat-stmt> ::= **repeat** <stmt-seq> **until** <exp>
6. <assign-stmt> ::= **ID** := <exp>
7. <read-stmt> ::= **read ID**
8. <write-stmt> ::= **write** <exp>
9. <exp> ::= <simple-exp> < <simple-exp>
   | <simple-exp> = <simple-exp>
   | <simple-exp>
10. <simple-exp> ::= <simple-exp> + <term>
    | <simple-exp> - <term>
    | <term>
11. <term> ::= <term> * <factor>
    | <term> / <factor>
    | <factor>
12. <factor> ::= (<exp>) | **NUM** | **ID**

Tiny.y

COMPILER CONSTRUCTION

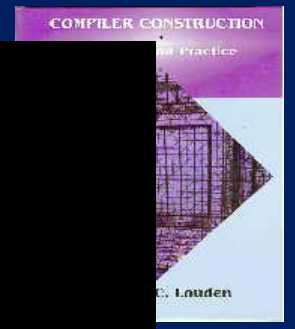
```
assign_stmt : ID { savedName = copyString(tokenString);
                   savedLineNo = lineno; }
            ASSIGN exp
              { $$ = newStmtNode(AssignK);
                $$->child[0] = $4;
                $$->attr.name = savedName;
                $$->lineno = savedLineNo;
              }
          ;
read_stmt   : READ ID
              { $$ = newStmtNode(ReadK);
                $$->attr.name =
                  copyString(tokenString);
              }
          ;
write_stmt  : WRITE exp
              { $$ = newStmtNode(WriteK);
                $$->child[0] = $2;
              }
          ;
```

Tiny 2 The Grammar

1. <program> ::= <stmt-seq>
2. <stmt-seq> ::= <stmt-seq> ; <stmt>
   | <stmt>
3. <stmt> ::= <if-stmt> | <repeat-stmt>
   | <assign-stmt> | <read-stmt>
   | <write-stmt>
4. <if-stmt> ::= **if** <exp> **then** <stmt-seq> **end**
   | **if** <exp> **then** <stmt-seq>
   else <stmt-seq> **end**
5. <repeat-stmt> ::= **repeat** <stmt-seq> **until** <exp>
6. <assign-stmt> ::= **ID** := <exp>
7. <read-stmt> ::= **read ID**
8. <write-stmt> ::= **write** <exp>
9. <exp> ::= <simple-exp> < <simple-exp>
   | <simple-exp> = <simple-exp>
   | <simple-exp>
10. <simple-exp> ::= <simple-exp> + <term>
    | <simple-exp> - <term>
    | <term>
11. <term> ::= <term> * <factor>
    | <term> / <factor>
    | <factor>
12. <factor> ::= (<exp>) | **NUM** | **ID**

Tiny.y

```
exp       : simple_exp LT simple_exp
            { $$ = newExpNode(OpK);
              $$->child[0] = $1;
              $$->child[1] = $3;
              $$->attr.op = LT;
            }
          | simple_exp EQ simple_exp
            { $$ = newExpNode(OpK);
              $$->child[0] = $1;
              $$->child[1] = $3;
              $$->attr.op = EQ;
            }
          | simple_exp { $$ = $1; }
          ;
```

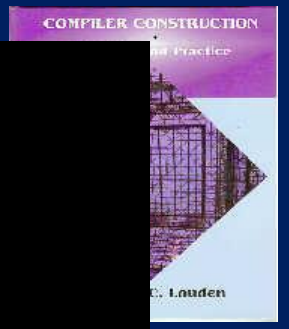


Tiny 2 The Grammar

1. <program> ::= <stmt-seq>
2. <stmt-seq> ::= <stmt-seq> ; <stmt>
   | <stmt>
3. <stmt> ::= <if-stmt> | <repeat-stmt>
   | <assign-stmt> | <read-stmt>
   | <write-stmt>
4. <if-stmt> ::= **if** <exp> **then** <stmt-seq> **end**
   | **if** <exp> **then** <stmt-seq>
   else <stmt-seq> **end**
5. <repeat-stmt> ::= **repeat** <stmt-seq> **until** <exp>
6. <assign-stmt> ::= **ID** := <exp>
7. <read-stmt> ::= **read ID**
8. <write-stmt> ::= **write** <exp>
9. <exp> ::= <simple-exp> < <simple-exp>
   | <simple-exp> = <simple-exp>
   | <simple-exp>
10. <simple-exp> ::= <simple-exp> + <term>
    | <simple-exp> - <term>
    | <term>
11. <term> ::= <term> * <factor>
    | <term> / <factor>
    | <factor>
12. <factor> ::= (<exp>) | **NUM** | **ID**

Tiny.y

```
simple_exp  : simple_exp PLUS term
              { $$ = newExpNode(OpK);
                $$->child[0] = $1;
                $$->child[1] = $3;
                $$->attr.op = PLUS;
              }
            | simple_exp MINUS term
              { $$ = newExpNode(OpK);
                $$->child[0] = $1;
                $$->child[1] = $3;
                $$->attr.op = MINUS;
              }
            | term { $$ = $1; }
            ;
```

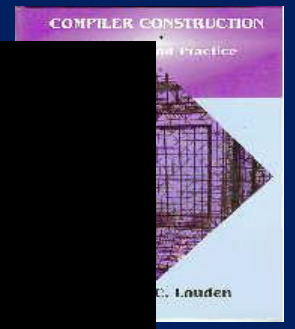


Tiny 2 The Grammar

1. <program> ::= <stmt-seq>
2. <stmt-seq> ::= <stmt-seq> ; <stmt>
   | <stmt>
3. <stmt> ::= <if-stmt> | <repeat-stmt>
   | <assign-stmt> | <read-stmt>
   | <write-stmt>
4. <if-stmt> ::= **if** <exp> **then** <stmt-seq> **end**
   | **if** <exp> **then** <stmt-seq>
   else <stmt-seq> **end**
5. <repeat-stmt> ::= **repeat** <stmt-seq> **until** <exp>
6. <assign-stmt> ::= **ID** := <exp>
7. <read-stmt> ::= **read ID**
8. <write-stmt> ::= **write** <exp>
9. <exp> ::= <simple-exp> < <simple-exp>
   | <simple-exp> = <simple-exp>
   | <simple-exp>
10. <simple-exp> ::= <simple-exp> + <term>
    | <simple-exp> - <term>
    | <term>
11. <term> ::= <term> * <factor>
    | <term> / <factor>
    | <factor>
12. <factor> ::= (<exp>) | **NUM** | **ID**

Tiny.y

```
term      : term TIMES factor
            { $$ = newExpNode(OpK);
              $$->child[0] = $1;
              $$->child[1] = $3;
              $$->attr.op = TIMES;
            }
          | term OVER factor
            { $$ = newExpNode(OpK);
              $$->child[0] = $1;
              $$->child[1] = $3;
              $$->attr.op = OVER;
            }
          | factor { $$ = $1; }
          ;
```

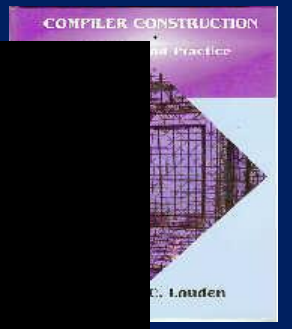


Tiny 2 The Grammar

1. <program> ::= <stmt-seq>
2. <stmt-seq> ::= <stmt-seq> ; <stmt>
   | <stmt>
3. <stmt> ::= <if-stmt> | <repeat-stmt>
   | <assign-stmt> | <read-stmt>
   | <write-stmt>
4. <if-stmt> ::= **if** <exp> **then** <stmt-seq> **end**
   | **if** <exp> **then** <stmt-seq>
   else <stmt-seq> **end**
5. <repeat-stmt> ::= **repeat** <stmt-seq> **until** <exp>
6. <assign-stmt> ::= **ID** := <exp>
7. <read-stmt> ::= **read ID**
8. <write-stmt> ::= **write** <exp>
9. <exp> ::= <simple-exp> < <simple-exp>
   | <simple-exp> = <simple-exp>
   | <simple-exp>
10. <simple-exp> ::= <simple-exp> + <term>
    | <simple-exp> - <term>
    | <term>
11. <term> ::= <term> * <factor>
    | <term> / <factor>
    | <factor>
12. <factor> ::= (<exp>) | **NUM** | **ID**

Tiny.y

```
factor     : LPAREN exp RPAREN
             { $$ = $2; }
           | NUM
             { $$ = newExpNode(ConstK);
               $$->attr.val = atoi(tokenString);
             }
           | ID { $$ = newExpNode(IdK);
               $$->attr.name =
                    copyString(tokenString);
             }
           | error { $$ = NULL; }
           ;

%%
```

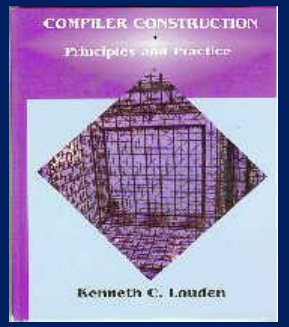


Tiny 2 The Grammar

Tiny.y


```
int yyerror(char * message)
{ fprintf(listing,"Syntax error at line %d:
%s\n",lineno,message);
  fprintf(listing,"Current token: ");
  printToken(yychar,tokenString);
  Error = TRUE;
  return 0;
}

/* yylex calls getToken to make Yacc/Bison output
 * compatible with ealier versions of the TINY scanner
 */
static int yylex(void)
{ return getToken(); }

TreeNode * parse(void)
{ yyparse();
  return savedTree;
}
```

Your code:



Any of these will run your code:

c- {filename}

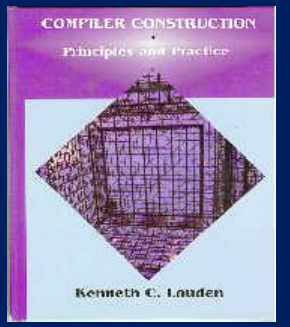
cat filename | c-

c- < filename

You will have at least these files:

- **parser.l** that contains the flex code
- **parser.y** that contains the bison code
- **scanType.h** that contains the declaration of either a struct or class that is used to pass your token information back from the scanner. This file will be included in the right place both the .l and .y files.
- **makefile** (note the all lowercase) that I will execute to build your c-.

ASN_1 parser.l



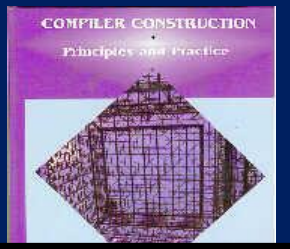

```
%{
#include "scanType.h"
#include "parser.tab.h"

using namespace std;

#define YY_DECL extern "C" int yylex()
#define YYSTYPE int

int line=1;

int setValue(int linenum, int tokenClass, char *svalue) {
   yylval.tinfo.tokenclass = tokenClass;
   yylval.tinfo.linenum = linenum;
   yylval.tinfo.tokenstr = strdup(svalue);
   yylval.tinfo.cvalue = '@';
   yylval.tinfo.nvalue = 777;
   yylval.tinfo.svalue = NULL;
   return tokenClass;
}
%}
%option noyywrap
```

%{ C Code %}
Copied “as is” to
lex.yy.c

Below we will increment line each time we see a \n

```
cp –r /y/shared/Engineering/cs-drbc/cs445/ASN_1 .
```

ASN_1 parser.l



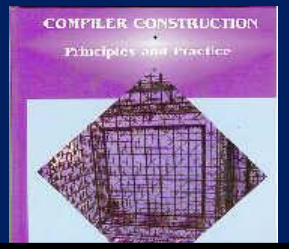

```
%{
#include "scanType.h"
#include "parser.tab.h"

using namespace std;

#define YY_DECL extern "C" int yylex()
#define YYSTYPE int

int line=1;

int setValue(int linenum, int tokenClass, char *svalue) {
   yylval.tinfo.tokenclass = tokenClass;
   yylval.tinfo.linenum = linenum;
   yylval.tinfo.tokenstr = strdup(svalue);
   yylval.tinfo.cvalue = '@';
   yylval.tinfo.nvalue = 777;
   yylval.tinfo.svalue = NULL;
   return tokenClass;
}
%}
%option noyywrap
```

yylval is a global variable used to pass info to parser.y.
The %union declaration in parser.y modifies the type of yylval .

```
#ifndef _SCANTYPE_H_
#define _SCANTYPE_H_
//
//  SCANNER TOKENDATA
//

struct TokenData
{
   int  tokenclass;       // token class
   int  linenum;          // line where found
   char *tokenstr;        // what string was read (pointer)
   char cvalue;           // any character value
   int  nvalue;           // any numeric or Boolean value
                          // string e.g. an id (pointer)
};
#endif
```

We will define tinfo to be a struct of type TokenData in parser.y

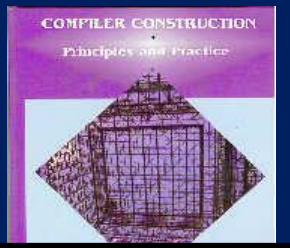
ASN_1 parser.l




```
%{
#include "scanType.h"
#include "parser.tab.h"

using namespace std;

#define YY_DECL extern "C" int yylex()
#define YYSTYPE int

int line=1;

int setValue(int linenum, int tokenClass, char *svalue) {
   yylval.tinfo.tokenclass = tokenClass;
   yylval.tinfo.linenum = linenum;
   yylval.tinfo.tokenstr = strdup(svalue);
   yylval.tinfo.cvalue = '@';
   yylval.tinfo.nvalue = 777;
   yylval.tinfo.svalue = NULL;
   return tokenClass;
}
%}
%option noyywrap
```

```
#ifndef _SCANTYPE_H_
#define _SCANTYPE_H_
//
//  SCANNER TOKENDATA
//

struct TokenData
{
   int  tokenclass;      // token class
   int  linenum;         // line where found
   char *tokenstr;       // what string was read (pointer)
   char cvalue;          // any character value
   int  nvalue;          // any numeric or Boolean value
   char *svalue;         // any string e.g. an id (pointer)

};
#endif
```

ASN_1 parser.l



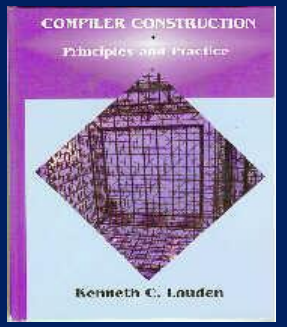

```
%{
#include "scanType.h"
#include "parser.tab.h"

using namespace std;

#define YY_DECL extern "C" int yylex()
#define YYSTYPE int
  char *lastToken=(char *)"";
int line=1;

int setValue(int linenum, int tokenClass, char *svalue) {
    yylval.tinfo.tokenclass = tokenClass;
    yylval.tinfo.linenum = linenum;
    yylval.tinfo.tokenstr = strdup(svalue);
    yylval.tinfo.cvalue = '@';
    yylval.tinfo.nvalue = 777;
    yylval.tinfo.svalue = NULL;
    return tokenClass;
}
%}
%option noyywrap
```

```
#ifndef _SCANTYPE_H_
#define _SCANTYPE_H_
//
//  SCANNER TOKENDATA
//

struct TokenData
{
    int  tokenclass;      // token class
    int  linenum;         // line where found
    char *tokenstr;       // what string was read (pointer)
    char cvalue;          // any character value
    int  nvalue;          // any numeric or Boolean value
    char *svalue;         // any string e.g. an id (pointer)
```

```
switch (tokenClass) {
   case BOOLCONST:
        yylval.tokenData->nvalue = ((lastToken[0]=='t') ? 1 : 0);
        break;
....
```

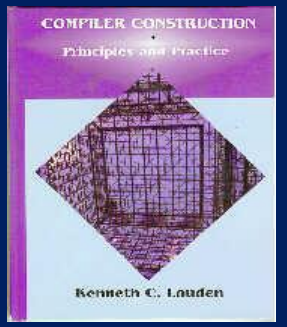
future assignment

ASN_1 parser.l




```
%{
#include "scanType.h"
#include "parser.tab.h"

using namespace std;

#define YY_DECL extern "C" int yylex()
#define YYSTYPE int

int line=1;

int setValue(int linenum, int tokenClass, char *svalue) {
   yylval.tinfo.tokenclass = tokenClass;
   yylval.tinfo.linenum = linenum;
   yylval.tinfo.tokenstr = strdup(svalue);
   yylval.tinfo.cvalue = '@';
   yylval.tinfo.nvalue = 777;
   yylval.tinfo.svalue = NULL;
   return tokenClass;
}
%}
%option noyywrap
```

ASN_1 parser.l




```
int setValue(int linenum, int tokenClass, char *svalue) {
   yylval.tinfo.tokenclass = tokenClass;
   yylval.tinfo.linenum = linenum;
   yylval.tinfo.tokenstr = strdup(svalue);
   yylval.tinfo.cvalue = '@';
   yylval.tinfo.nvalue = 777;
   yylval.tinfo.svalue = NULL;
   return tokenClass;
}
%}
%option noyywrap
letter      [a-zA-Z]
digit    [0-9]
quote    [\']
underbar [_]
letdig      {digit}|{letter}|{underbar}
limitters   [\;\,\(\)\{\}\[\]]
op    [\-\+\*\/\%]
whitespace  [\ \t]
%%
"<"      { return setValue(line, LT, yytext); }
">"      { return setValue(line, GT, yytext); }
\n       { line++; }
```

```
[\(\)\*\+\,\-\/\:\;\<\=\>\?\[\]\%\{\}]      future assignment
                                             { return setValue(line, yytext[0], yytext); }
```

```
[\(\)\*\+\,\-\/\:\;\<\=\>\?\[\]\%\{\}]      { return setValue(line, OP, yytext); }
```

```
":<:"                          { return setValue(line, MIN, yytext); }
...
"and"                           { return setValue(line, AND, yytext); }
...
```

```
int setValue(int linenum, int tokenClass, char *svalue)
   yylval.tinfo.tokenclass = tokenClass;
   yylval.tinfo.linenum = linenum;
   yylval.tinfo.tokenstr = strdup(svalue);
   yylval.tinfo.cvalue = '@';
   yylval.tinfo.nvalue = 777;
   yylval.tinfo.svalue = NULL;
   return tokenClass;
}
%}
%option noyywrap
letter     [a-zA-Z]
digit    [0-9]
quote    [\']
underbar [_]
letdig     {digit}|{letter}|{underbar}
limitters   [\;\,\(\)\{\}\[\]]
op    [\-\+\*\/\%]
whitespace  [\ \t]
%%
"<"     { return setValue(line, LT, yytext);  }
">"     { return setValue(line, GT, yytext);  }
\n      {  line++; }
```

The printToken function used below to make the output consistent. For now, I am using type to indicate if it is the char, string or number.

```
%{
#include <cstdio>
#include <iostream>
#include <unistd.h>
#include "scanType.h"
using namespace std;

extern "C" int yylex();
extern "C" int yyparse();
extern "C" FILE *yyin;

void yyerror(const char *msg);

void printToken(TokenData myData, string tokenName, int type = 0) {
  cout << "Line: " << myData.linenum << " Type: " << tokenName;
  if(type==0)
    cout << " Token: " << myData.tokenstr;
  if(type==1)
    cout << " Token: " << myData.nvalue;
  if(type==2)
    cout << " Token: " << myData.cvalue;
  cout << endl;
}
%}
```

The %{ C code %} will be copied “as is” to parser.tab.c

Note: This is just enough parser.y to get your parser.l to compile.
This file will change dramatically in future assignments.

```
int setValue(int linenum, int tokenClass, char *svalue)
   yylval.tinfo.tokenclass = tokenClass;
   yylval.tinfo.linenum = linenum;
   yylval.tinfo.tokenstr = strdup(svalue);
   yylval.tinfo.cvalue = '@';
   yylval.tinfo.nvalue = 777;
   yylval.tinfo.svalue = NULL;
   return tokenClass;
}
%}
%option noyywrap
letter      [a-zA-Z]
digit    [0-9]
quote    [\']
underbar [_]
letdig      {digit}|{letter}|{underbar}
limitters   [\;\,\(\)\{\}\[\]]
op    [\-\+\*\/\%]
whitespace  [\ \t]
%%
"<"      { return setValue(line, LT, yytext);  }
">"      { return setValue(line, GT, yytext);  }
\n       {  line++; }
```

```
%{
#include <cstdio>
#include <iostream>
#include <unistd.h>
#include "scanType.h"
using namespace std;

extern "C" int yylex();
extern "C" int yyparse();
extern "C" FILE *yyin;

void yyerror(const char *msg);

void printToken(TokenData myData, string tokenName, int type = 0) {
  cout << "Line: " << myData.linenum << " Type: " << tokenName;
  if(type==0)
    cout << " Token: " << myData.tokenstr;
  if(type==1)
    cout << " Token: " << myData.nvalue;
  if(type==2)
    cout << " Token: " << myData.cvalue;
  cout << endl;
}

%}
```

```
int setValue(int linenum, int tokenClass, char *svalue)
   yylval.tinfo.tokenclass = tokenClass;
   yylval.tinfo.linenum = linenum;
   yylval.tinfo.tokenstr = strdup(svalue);
   yylval.tinfo.cvalue = '@';
   yylval.tinfo.nvalue = 777;
   yylval.tinfo.svalue = NULL;
   return tokenClass;
}
%}
%option noyywrap
letter      [a-zA-Z]
digit    [0-9]
quote    [\']
underbar [_]
letdig      {digit}|{letter}|{underbar}
limitters   [\;\,\(\)\{\}\[\]]
op    [\-\+\*\/\%]
whitespace  [\ \t]
%%
"<"     { return setValue(line, LT, yytext);  }
">"     { return setValue(line, GT, yytext);  }
\n      {  line++; }
```

```
%union
{
  struct   TokenData tinfo ;
}
%token   <tinfo>  LT
%token   <tinfo>  GT
%token   <tinfo>  ERROR
%type <tinfo>  term program
%%
program  :  program term
   |  term  {$$=$1;}
   ;
term  :
    LT {printToken(yylval.tinfo, "LT");}
   |  GT {printToken(yylval.tinfo, "GT");}
   |  ERROR    {cout << "ERROR(SCANNER Line " <<
yylval.tinfo.linenum << "): Invalid input character " <<
yylval.tinfo.tokenstr << endl; }
   ;
%%
void yyerror (const char *msg)
{
  cout << "Error: " <<  msg << endl;
}
int main(int argc, char **argv) {
```

Remember in parser.l we did:

```
[\(\)\*\+\,\-\/\:\;\<\=\>\?\[\]\%\{\}]
                        { return setValue(line, OP, yytext); }
```

So here it will be:

```
 |  OP    {printToken(yylval.tinfo, "OP", 2); }
```

```
 | MIN  {printToken(yylval.tinfo, "MIN");}
...
 |  AND   {printToken(yylval.tinfo, "AND");}
...
```

```
%union
{
  struct   TokenData tinfo ;
}
%token   <tinfo>  LT
%token   <tinfo>  GT
%token   <tinfo>  ERROR
%type <tinfo>  term program
%%
program  :  program term
  |  term  {$$=$1;}
  ;
term  :
     LT {printToken(yylval.tinfo, "LT");}
  |  GT {printToken(yylval.tinfo, "GT");}
  |  ERROR    {cout << "ERROR(SCANNER Line " <<
yylval.tinfo.linenum << "): Invalid input character " <<
yylval.tinfo.tokenstr << endl; }
  ;
%%
void yyerror (const char *msg)
{
  cout << "Error: " <<  msg << endl;
}
int main(int argc, char **argv) {
```

This is completely unrealistic. It is just enough to get your .l file to compile.
We are saying that a program is just a list of terms (in ANY order), so fishy.bC is a valid program.

```
%union
{
  struct   TokenData tinfo ;
}
%token   <tinfo>  LT
%token   <tinfo>  GT
%token   <tinfo>  ERROR
%type <tinfo>  term program
%%
program  :  program term
  |  term  {$$=$1;}
  ;
term  :
     LT {printToken(yylval.tinfo, "LT");}
  |  GT {printToken(yylval.tinfo, "GT");}
  |  ERROR    {cout << "ERROR(SCANNER Line " <<
yylval.tinfo.linenum << "): Invalid input character " <<
yylval.tinfo.tokenstr << endl; }
  ;
%%
void yyerror (const char *msg)
{
  cout << "Error: " <<  msg << endl;
}
int main(int argc, char **argv) {
```

The %union declaration specifies the entire collection of possible data types for semantic values. (Remember the yylval from parser.l?)

future assignment

```
%union {
   TokenData *tokenData;
   TreeNode *tree;
   ExpType type;   // for passing type spec up the tree
}
```

future assignment

```
%type <tokenData> assignop minmaxop
...
%type <tree> andExp argList
...
%type <type> typeSpec
```

```
%union
{
  struct   TokenData tinfo ;
}
%token   <tinfo>  LT
%token   <tinfo>  GT
%token   <tinfo>  ERROR
%type    <tinfo>  term program
%%
program  :  program term
  |  term  {$$=$1;}
  ;
term  :
     LT {printToken(yylval.tinfo, "LT");}
  |  GT {printToken(yylval.tinfo, "GT");}
  |  ERROR    {cout << "ERROR(SCANNER Line " <<
yylval.tinfo.linenum << "): Invalid input character " <<
yylval.tinfo.tokenstr << endl; }
  ;
%%
void yyerror (const char *msg)
{
  cout << "Error: " <<  msg << endl;
}
int main(int argc, char **argv) {
```

% token *this assignment*

- Identifies the token names that the yacc command accepts.
- They must be declared here to be used in parser.l
- You can put several on the same line:
  %token <tinfo> LT GT MIN AND

% type *future assignment*

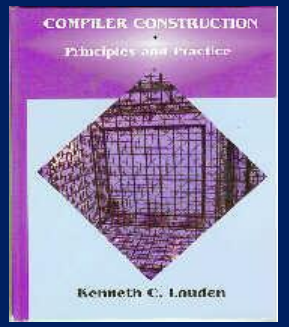
- These are the left side of our CFG
- For now the only types are program and term

```
%union
{
  struct  TokenData tinfo ;
}
%token  <tinfo>  LT
%token  <tinfo>  GT
%token  <tinfo>  ERROR
%type   <tinfo>  term program
%%
program  :  program term
  |  term  {$$=$1;}
  ;
term  :
    LT {printToken(yylval.tinfo, "LT");}
  |  GT {printToken(yylval.tinfo, "GT");}
  |  ERROR   {cout << "ERROR(SCANNER Line " <<
yylval.tinfo.linenum << "): Invalid input character " <<
yylval.tinfo.tokenstr << endl; }
  ;
%%
void yyerror (const char *msg)
{
  cout << "Error: " <<  msg << endl;
}
int main(int argc, char **argv) {
```

After the %% is more C code. Ignore this for now. We need a main function to compile, but this will be moved to other files as our project gets bigger.

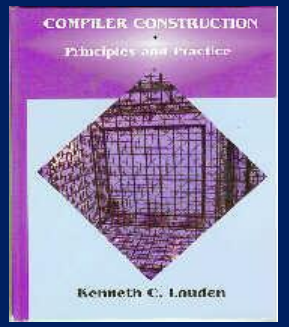
```
%union
{
  struct   TokenData tinfo ;
}
%token   <tinfo>  LT
%token   <tinfo>  GT
%token   <tinfo>  ERROR
%type <tinfo>  term program
%%
program  :  program term
  |  term  {$$=$1;}
  ;
term  :
     LT {printToken(yylval.tinfo, "LT");}
  |  GT {printToken(yylval.tinfo, "GT");}
  |  ERROR    {cout << "ERROR(SCANNER Line " <<
yylval.tinfo.linenum << "): Invalid input character " <<
yylval.tinfo.tokenstr << endl; }
  ;
%%
void yyerror (const char *msg)
{
  cout << "Error: " <<  msg << endl;
}
int main(int argc, char **argv) {
```

Importance of Compilers



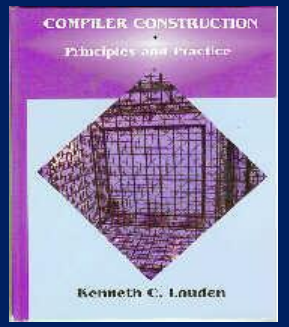
- You can create your own language
- Know better how languages work
- We expect compilers to be flawless and reliable

Importance of This course: Advisory Board



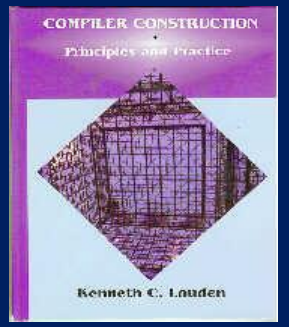
- So every department has an industrial Advisory Board: companies in the northwest that advise us on, on what we should have in our curriculum, what they like to see.
- We were going to get rid of this course at one point.
  - We have a limited staff.
- The Advisory board said keep it.
  - It is an individual project in which the student are able to compose a large working program in one big block all on their own.
  - Several of them said they look when they were looking at Idaho students. They looked at the compilers course to see how good a job they did in making decisions for hiring.
- So we put it back because they screamed loudly
  - They come back once a year and they go out of their way to say we're very happy you have the compilers course.
  - We like to cater to our industrial partners by giving them this course.

Goals

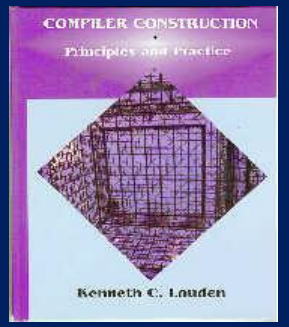


- To understand the basics of translation, compilation, interpolation.
- Understand the theory behind each of those.
- Understand why languages are built the way they are and compilers behave the way they do.
  - Because the compilers do strange things sometimes, and I didn't understand it until I took a compiler's course.
- Gain hands on experience in the construction of a compiler.

Tie in from prerequisites



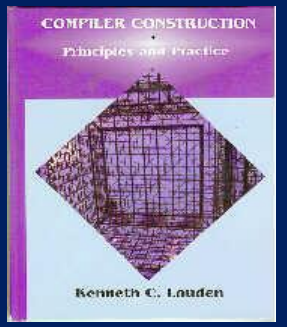
- Programing languages
  - Scoping rules
  - Static variables
- Theory of computation
  - State Machines (Used in this class for parsers)



- You probably write maybe as much as 5000 lines of code.
- Preferred language is C++.
- I'll be supplying you, for instance, a symbol table live object in C++ that you can call.
  - You don't have to use it, but it has a bunch of debugging stuff in it so you can help debug your code.

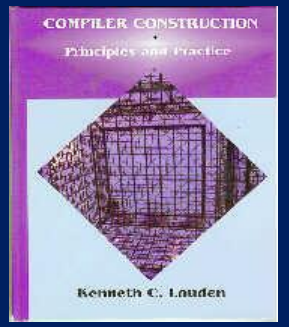
Note: Each assignment must be complete before you move to the next one.

Tests



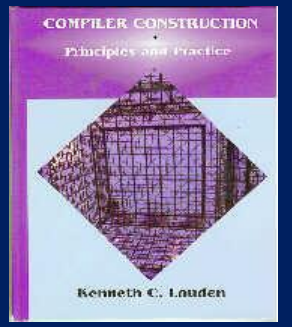
- Tests: They're simple hour long tests.
  - The first one will be on grammar stuff because people need it.
  - The second one is on parsers because we got to have it.

Necessary Skills for CS445



- Here is a short list of the necessary skills (C/C++) for being successful at building a compiler in CS445.
  - Understand the difference between declare and define.
  - Understand command line argument processing with getopt.
  - Understand the use old "C style" strings which are done with char *. No, really. This includes things like strlen, strdup, and printf formatting.
  - Also understanding std::strings might be of help.
  - A very basic template library stuff might be useful.
  - Understanding state machines.
  - Understand pointers, pointers, pointers, pointers, and more pointers ...
  - Understand memory allocation with "new" and "delete".

Necessary Skills for CS445



- Here is a short list of the necessary skills (C/C++) for being successful at building a compiler in CS445.
  - Being skilled at recursion esp. with respect to tree construction and traversal.
  - Very important to understand construction of n-ary trees in C/C++ using pointers. Know how to traverse the tree. Again... understand recursion, understand pointers.
  - Understanding of objects might not be needed but this is a large program and some parts lend themselves to thoughts of objects (although the sample compiler will not use them).
  - You must understand debugging of seg faults! Get used to gdb or other debugger that can track down pointer corruption.
  - You need to understand how to build programs from multiple files using make. In particular you must understand how a makefile works and how to create a makefile.
  - The proper use of header files and code files. Do not put executable code in header files! Know why you don't do that.
  - Understand the meaning of extern and static.