**COP3402 PROJECT Kool Compiler Fall 2011**

**Due: Monday, December 5 at 11:59PM (no lateness allowed)**

The major project for this course is the creation of a language translator for a made up language that we shall call Kool. The language is defined, to a large extent, by the grammar that is displayed later in this document.

**THE SOURCE LANGUAGE:**

Before discussing the intermediate code, I need to describe some features of Kool you will be translating. Some of this is evident from the grammar. Some is not.

1. The program starts with a block of int declarations (scalars and vectors) that are global (available to all code within the program). These are stored in a randomly accessible memory area.
2. Unlike our previous assignments, Kool does not have a main class, just a main body of code in braces, which has access to global variables and appears after all classes (of which there may be none). The simplest program is just { } – no declarations; no body; implied return. A slightly more interesting, but still silly, one is

int x; /\* global x \*/

int y; /\* local x \*/

{

x = 1; y = 2;

x = x+y;

}

1. The notion of a class is core to this language, but these classes are fairly lame (so the assignment is manageable). A class has a name that becomes available immediately as a class variable. A class then has just one method in it (a class method), which starts with the keyword METHOD followed by a set of zero or more formal parameters enclosed in parentheses. Actual parameters are limited to int scalar variables, which are passed by value (this is done by copying the values to a stack). This is then followed by local declarations (also on the stack) and a body of code, which is enclosed in braces. The class is immediately available for us and yas a pseudonym called “self.” However, classes never can be used in expressions or as arguments to calls. Thus, they only serve as gateways to their class’s code. Recursion is usually done using self, but you can also explicitly refer to your own class name.

int y;

int fact;

class factorial {

method (n)

int n\_minus\_1;

{

if (n<=1) fact = 1;

else {

n\_minus\_1 = n-1; self(n\_minus\_1); fact = n \* fact;

}

}

}

{

y = 5;

factorial(y);

}

1. Classes cannot be nested. Classes do not have instance or class variables, nor can instances of a class be decalred. Moreover, only one method exists per class and it is unnamed.
2. Arguments to a class’s method are simple variables. They may not include class names and/or constants and/or expressions with operators. Formal parameters are always by value, so results can only be returned by altering global variables. There are no functions. Note: A nice extension would be to add a return expression to the RETURN statement and allow class methods to return int results.
3. Upper and lower case letters are indistinguishable in identifiers and keywords. Underscores may appear inside identifiers, but only serve to improve readability. However, they should not be embedded in keywords. Also, identifiers are truncated to at most 8 characters, after underscore removal.

**OPCODES**

**Constant term**

CON op1 0 // op1 is a constant value (positive or zero)

op used for following code is offset in main memory, offset from stack mark or a –triple

Offsets for main memory should be preceded by ‘M’; stack should be preceded by ‘S’. In your recursive descent just return index to symbol table and emit should take care of rest using storetype (see symbol table description).

**Assignment operators**

:= op1 op2 // assign op2 to op1

[]= op1 op2 // assigns the value from op3 to vector component op1[op2]

\_ op3 0

**Accessor for subscripted variable**

=[] op1 op2 // evaluates to op1[op2] (used in rhs references)

**Arithmetic operations**

+ op1 op2 // operands can be to symbol table (positive)

- op1 op2 // operands can be references to other triples (negative)

\* op1 op2 // no operand can be zero

/ op1 op2

++ op1 0

-- op1 0

**Unconditional jump uses first field as destination triple**

JUMP -loc 0 // jumps to triple at loc

**Conditional jumps use first field to reference triple where test value was computed**

**Second field is destination triple**

JZ -loc1 -loc2 // conditional jump on zero (=0)

JNZ -loc1 -loc2 // conditional jump on not zero (<>0)

JP -loc1 -loc2 // conditional jump on positive (>0)

JNP -loc1 -loc2 // conditional jump on not positive (<=0)

JM -loc1 -loc2 // conditional jump on minus (<0)

JNM -loc1 -loc2 // conditional jump on not minus (>=0)

**Calls and returns (assumes a stack for procedure calls)**

ARG entry 0 // pushes value associated with argument variable on stack

CALL #args #sz // adds #sz (size of local storage) to stack top;

\_ -loc1 0 // pushes return and old mark on stack; sets stack mark

// to stack location just above first ARG; calls code at loc1

RET 0 0 // pops return address; cleans up stack to current mark and returns

// also works to exit main

**Miscellaneous**

ERROR 0 0 // error (can extend to put error codes in triple)

NOP 0 0 // no operation (used when instruction deleted

// might use operands to save useful information)

**LIMITATIONS**

I have limited formal parameters to be an untyped list of variable names. That’s acceptable in that all arguments must be int variables. It does limit us some, but it makes your life easier. You, in fact, only need to know the number of arguments required and be sure there is a match; associations of names to stack offsets will still need to be done when you compile the routine’s code. Realize that methods must use globals to return results.

There are no logical operators. That makes life a bit easier. In fact, all tests are of the form expression RELOP expression, where the expressions are numeric. There are no new control structures or arithmetic operations beyond what you did in Assignment#3. Of course, you now need to do everything in a recursive descent parser.

**SYMBOL TABLE STRUCTURE**

**name nesting type cls**

**name** is obvious; it’s the lexeme that resulted in the identifier token

**nesting** for us is static (memory) or auto (stack). Use code of 1 for memory and 2 for stack. Note: real language also uses heap.

**type** is not the data type but the kind of identifier. The choices are class, var and formal argument.

**cls** is the class of the identifier. We treat int as a pseudo class, so this is really a data type. A class actually points to itself.

There are other symbol table fields that depend on the type of identifier.

For a var or formal argument, we need

**base** for main memory is the base of the structure; really an offset from some point. We assume 1, 2, etc. Of course arrays affect this. Offset for stack is relative to latest mark. 1 is first arg, …, k is k-th arg; k+1 is first local, etc.

**size** is number of elements in area allocated for variable (0 for scalar, m for array of length m).

For a class, we need

**base**  – as above. This is used for the start of code in code space.

**num\_args** is the number of arguments (0 or more) expected fro its method call.

**local\_space** is the number of stack slots needed for local variables.

**OPTIMIZATIONS**

**Easy:**

Identities: x+0 = 0+x = x\*1 = 1\*x = x

Increment/Decrement; x+1 = 1+x = ++value(x); x-1 = --value(x); works even if x is a triple reference

Note: ++ and -- are not part of language, just of intermediate code and they do not alter any variable

Strength reduction: x\*2 = 2\*x = x+x

Constant folding: Examples: 2+7 = 9; 2 \* 7 = 14;

Useless: a = a can be replaced by NOP but be careful about references to this triple by other triples

**Harder:**

Common subexpressions: Be careful – must have no intervening changes to operands unless CON entries

Code compression (getting rid of NOPs and adjusting all references correctly)

Goto chasing: jumps to jumps (e.g., unconditional jump to conditional); jumps to returns; jumps to next triple

Other ones that you think about

**Note:** I will provide other suggestions over the next week for extra credit.

**WHAT YOU MUST TURN IN**

1. A C or C++ program that includes a lexical analyzer, recursive descent parser, intermediate code generator (triples are required) and some code optimizations (at least two simple optimizations; challenging optimizations can results in extra credit or can offset other less well done parts of the assignment) for the Kool language. This program must report syntactic and unit-based semantic errors. It must also recover gracefully from errors. As in Assign#3, code is shown incrementally and at end. Symbol table must be dumped at start of each block of code for a procedure or the main (after local variables are inserted for method block and at start of main block where symbol table has just globals which include all class method prototypes).
2. A log book with at least three entries per week from now until the due date (December 5, 2011). This log book should report progress made, problems encountered and solutions found. Each entry should be dated and time stamped. Always bring your log when you come to me or Steven or Wenhui.
3. A summary of your project that highlights its successes and shortcomings, as you perceive them. This is just a page long.
4. A collection of two to five sample programs that highlight the strengths (error recovery, code improvement) and shortcomings of your compiler.

%{

/\*////////////////////////////////////////////////////////////////////////

// LEX for a Kool language //

// //

// FILENAME: KoolProject.l //

// //

// Author(s): Charles E. Hughes //

// //

////////////////////////////////////////////////////////////////////////\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

#include <ctype.h>

#include <limits.h>

%}

LETTER [A-Za-z]

DIGIT [0-9]

ID {LETTER}({LETTER}|{DIGIT}|\_)\*

NUMBER {DIGIT}+

WHITE [ \t\f]

A [aA]

B [bB]

C [cC]

D [dD]

E [eE]

F [fF]

G [gG]

H [hH]

I [iI]

J [jJ]

K [kK]

L [lL]

M [mM]

N [nN]

O [oO]

P [pP]

Q [qQ]

R [rR]

S [sS]

T [tT]

U [uU]

V [vV]

W [wW]

X [xX]

Y [yY]

Z [zZ]

%%

"/\*" {ECHO; comment();}

"//".\* {ECHO;}

"==" {ECHO;return(EQ);}

"#" {ECHO;return(NE);}

"<" {ECHO;return(LT);}

"<=" {ECHO;return(LE);}

">" {ECHO;return(GT);}

">=" {ECHO;return(GE);}

"+" {ECHO; return(PLUS);}

"-" {ECHO; return(MINUS);}

"\*" {ECHO; return(TIMES);}

"/" {ECHO; return(DIVIDE);}

"=" {ECHO; return(ASSIGN);}

"{" {ECHO; return(LBRACE);}

"}" {ECHO; return(RBRACE);}

"(" {ECHO; return(LPAREN);}

")" {ECHO; return(RPAREN);}

"[" {ECHO; return(LBRACK);}

"]" {ECHO; return(RBRACK);}

";" {ECHO; return(SEMICOLON);}

"," {ECHO; return(COMMA);}

{D}{O} {ECHO; return(DO);}

{D}{O}{W}{N} {ECHO; return(DOWN);}

{E}{L}{S}{E} {ECHO; return(ELSE);}

{F}{O}{R} {ECHO; return(FOR);}

{I}{F} {ECHO; return(IF);}

{R}{E}{T}{U}{R}{N} {ECHO; return(RETURN);}

{T}{H}{E}{N} {ECHO; return(THEN);}

{U}{P} {ECHO; return(UP);}

{W}{H}{I}{L}{E} {ECHO; return(WHILE);}

{C}{L}{A}{S}{S} {ECHO; return(CLASS);}

{I}{N}{T} {ECHO; return(INT);}

{M}{E}{T}{H}{O}{D} {ECHO; return(METHOD);}

{S}{E}{L}{F} {ECHO; return(SELF);}

{ID} {ECHO; setid(); return(IDENT);}

{NUMBER} {ECHO; setnum(); return(NUMBER);}

{WHITE} {ECHO;}

[\n]+ {ECHO; dumpcode(0);}

. {ECHO; return(BAD);}

%%

void setnum(void) {

int i;

yylval.ival = 0;

i = 0;

while (yytext[i]) /\* convert string to int \*/

yylval.ival = yylval.ival\*10 + ( (int) yytext[i++] - (int) '0' );

}

void setid(void) {

int i = 0;

int j = 0;

while (yytext[i] && j<IDLENGTH-1) {

if (yytext[i]!='\_') idname[j++] = tolower(yytext[i]); i++;

}

idname[j] = '\0';

strcpy(symtab[0].name, idname); /\* copy new symbol into table \*/

i = symsize+1; /\* search bottom up -- small table so linear okay \*/

while ( strcmp(symtab[--i].name, idname) ); /\* always succeeds \*/

yylval.ival = i;

}

void comment(void) {

int c;

c = input(); putchar(c);

do {

while ((c != '\*') && (c != EOF)) {

c = input();

putchar(c);

}

c = input();

putchar(c);

} while ((c != '/') && (c!=EOF));

}

int yywrap(void) {

return(1);

}

%{

/\*////////////////////////////////////////////////////////////////////////

// YACC for a Kool language //

// //

// FILENAME: KoolProject.y //

// //

// Author(s): Charles E. Hughes //

// //

// Revision History: //

// 11/4/2011: Basic grammar and symbol table entries mostly completed //

// 11/5/2011: Semantic checks and code generation added //

// 11/6/2011: Simple optimization added (constant pool) //

// Symbol table print compressed //

// Code cleaned up for consistency //

// References to local (stack) vs global memory added //

// 11/8/2011: Added interpreter //

// 11/9/2011: Made minor changes removing class types //

// //

////////////////////////////////////////////////////////////////////////\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

#include <ctype.h>

#include <limits.h>

/\* Constants \*/

#define IDLENGTH 9 /\* one more (for \0) than max ident length \*/

#define SYM\_MAX 200 /\* size of symbol table \*/

#define CODE\_MAX 300 /\* size of code table \*/

#define CONST\_MAX 50 /\* size of constant pool \*/

#define DEFAULT\_STACK 300 /\* size of stack \*/

#define MAINSLOT 1

#define INTSLOT 2

#define NONE 0

#define GLOBAL 1

#define LOCAL 2

#define CLASS\_TYPE 1

#define VAR\_TYPE 2

#define FORMAL\_TYPE 3

char\* type\_names[] = {"NONE ", "CLASS ", "VAR ", "FORMAL" };

char\* nest\_names[] = {"NONE ", "GLOBAL", "LOCAL " };

struct class\_type {

int base; /\* first triple for code \*/

int num\_args; /\* space on stack for arguments \*/

int local\_space; /\* space on stack for local variables \*/

} ;

struct var\_type {

int base; /\* offset into storage (memory or stack ) \*/

int size; /\* non-zero if vector (only ints are allowed to be vectors) \*/

} ;

union symtab\_entry { /\* create object with fields for various symbol table entries \*/

struct class\_type clazz;

struct var\_type var;

};

struct symtab\_type {

char name[IDLENGTH];

int nesting; /\* 0 = none (no actual entry); 1 = global (static); 2 = local (auto); \*/

int type; /\* 0 = none (no actual type); 1=class; 2=var; 3=formal arg; \*/

int cls; /\* points to class entry (can be self or even int) \*/

union symtab\_entry sym;

} symtab[SYM\_MAX];

struct code\_type {

char opcode[IDLENGTH]; /\* mnemonic of operator \*/

char prefix1, prefix2; /\* used to differentiate Memory from Stack \*/

int first,second; /\* first and second operand \*/

} code[CODE\_MAX];

struct const\_pool\_type {

int value; /\* int value of constant \*/

int triple; /\* triple where constant is computed \*/

} const\_pool[CONST\_MAX];

/\* Other variables \*/

char idname[IDLENGTH]; /\* Ident just found -- first eight characters are meaningful \*/

int triple; /\* Next available code slot \*/

int decl\_size; /\* space for declared variables \*/

int globals; /\* space needed for globals \*/

int showStart; /\* start of code area yet to be displayed \*/

int symsize; /\* next available symbol table slot \*/

int constSize; /\* next available cons-Pool slot \*/

int nesting; /\* current nesting \*/

int currclass; /\* current class def \*/

int var\_start; /\* first symbol table entry in class's var \*/

int i; /\* counter \*/

int errorNo; /\* error number counter \*/

int warnNo; /\* warning number counter \*/

FILE \*interp = NULL; /\* output file of triples for interpretation \*/

int forwardTest; /\* flag indicating cond jump forward or backward \*/

int errorNo; /\* error number counter \*/

int warnNo; /\* warning number counter \*/

/\* lexical helpers \*/

void setid(void);

void comment(void);

void setnum(void);

/\* initialize all \*/

void basicinit(void);

/\* symbol table helpers \*/

void store\_cons(void);

void store\_class(void);

void store\_var(int size);

void store\_formal(void);

void enter(void);

/\* intermediate code helpers \*/

void backpatch(int trip, int val);

void backpatch1(int trip, int val);

int redundant(char \*opcode, int first, int second);

int emit(char \*opcode, char prefix1, int first, char prefix2, int second);

/\* report routines \*/

void dump\_sym(void);

void dumpcode(int final);

/\* error message routines \*/

void fatal(char const \*s);

void error(char const \*s);

void warning(char const \*s);

void yyerror(char const \*s);

/\* parser attributes \*/

typedef struct { int ival; int cval; char prefix; char op[3]; } YYSTYPE;

extern YYSTYPE yylval;

#define YYSTYPE\_IS\_DECLARED 1

%}

%token SEMICOLON COMMA LBRACE RBRACE LBRACK RBRACK RPAREN LPAREN DOT

%token CLASS INT IDENT NUMBER

%token DO DOWN ELSE FOR IF RETURN THEN UP WHILE

%token ASSIGN METHOD SELF BAD

%nonassoc EQ NE LT LE GT GE

%left PLUS MINUS

%left TIMES DIVIDE

%%

program:

startup

decl.section

classes

body

;

startup:

/\* EMPTY \*/

;

decl.section:

decl.section decl

|

/\* EMPTY \*/

;

decl:

type var.list SEMICOLON

|

error SEMICOLON

;

type:

INT

;

var.list:

var.list COMMA var.def

|

var.def

;

var.def:

IDENT LBRACK NUMBER RBRACK

|

IDENT

;

classes:

classes class.def

|

/\* EMPTY \*/

;

class.def:

class.header

LBRACE

method.section

RBRACE

;

class.header:

CLASS IDENT

;

method.section:

METHOD

decl.section

formal.args

body

;

formal.args:

LPAREN formal.list RPAREN

|

LPAREN RPAREN

;

formal.list:

formal.list COMMA formal.arg

|

formal.arg

;

formal.arg:

IDENT

;

body:

LBRACE statement.list RBRACE

;

statement.list:

statement.list statement

|

/\* EMPTY \*/

;

statement:

assign SEMICOLON

|

method.call SEMICOLON

|

return SEMICOLON

|

for

|

while

|

do SEMICOLON

|

if

|

LBRACE statement.list RBRACE

|

SEMICOLON

;

assign:

IDENT sub ASSIGN exp

;

sub:

LBRACK exp RBRACK

|

/\* EMPTY \*/

;

variable:

IDENT sub

;

exp:

exp PLUS exp

|

exp MINUS exp

|

exp TIMES exp

|

exp DIVIDE exp

|

LPAREN exp RPAREN

|

NUMBER

|

variable

;

method.call:

class.name LPAREN arguments RPAREN

;

class.name:

IDENT

|

SELF

;

arguments:

call.list

|

/\* EMPTY \*/

call.list:

call.list COMMA actual.argument

|

actual.argument

;

actual.argument:

IDENT

;

return:

RETURN

;

while:

while.start

;

while.start:

WHILE

;

optionalDo:

DO

|

/\* EMPTY \*/

;

for:

FOR for.direction LPAREN for.init test.start test RPAREN optionalDo statement

;

for.init:

IDENT ASSIGN exp SEMICOLON

;

for.direction:

UP

|

DOWN

|

/\* EMPTY \*/

;

test.start:

/\* EMPTY \*/

;

test:

exp EQ exp

|

exp NE exp

|

exp LT exp

|

exp LE exp

|

exp GT exp

|

exp GE exp

;

do:

do.start LBRACE statement.list RBRACE WHILE LPAREN test RPAREN

;

do.start:

DO

;

if:

IF LPAREN test RPAREN optionalThen statement optionalElse

;

optionalThen:

THEN

|

/\* EMPTY \*/

;

optionalElse:

ELSE

|

/\* EMPTY \*/

;

%%

void basicinit(void) {

symtab[0].nesting=NONE; /\* fake to not match current level \*/

nesting = GLOBAL;

strcpy(symtab[MAINSLOT].name, "\*MAIN\*");

symtab[MAINSLOT].nesting = nesting; /\* global \*/

symtab[MAINSLOT].type = CLASS\_TYPE; /\* pseudo class \*/

symtab[MAINSLOT].cls = MAINSLOT; /\* class is self \*/

symtab[MAINSLOT].sym.clazz.num\_args = 0;

symtab[MAINSLOT].sym.clazz.base = 1;

strcpy(symtab[INTSLOT].name, "INT");

symtab[INTSLOT].nesting = nesting; /\* global \*/

symtab[INTSLOT].type = CLASS\_TYPE; /\* pseudo class \*/

symtab[INTSLOT].cls = INTSLOT; /\* INT class is self \*/

symtab[INTSLOT].sym.clazz.num\_args = 0;

symtab[INTSLOT].sym.clazz.base = 1;

symsize = INTSLOT; /\* Index of last symbol table slot (some reserved) \*/

code[0].opcode[0] = '\0';

showStart = 2; /\* Index of next code slot that has yet to be dumped \*/

triple = 1; /\* Index of first available code slot \*/

constSize = 0; /\* index of next available constant pool slot \*/

decl\_size = 0; /\* space for variable section \*/

errorNo = 0; /\* Number of errors detected \*/

warnNo = 0; /\* Number of warnings given \*/

}

void enter(void) {

if (++symsize < SYM\_MAX) {

strcpy(symtab[symsize].name, idname);

symtab[symsize].nesting = nesting;

symtab[symsize].cls = INTSLOT;

yylval.ival = symsize;

}

else fatal("Symbol table overflow");

}

void store\_class(void) {

enter();

symtab[symsize].type = CLASS\_TYPE;

symtab[symsize].cls = symsize;

symtab[symsize].sym.clazz.num\_args = 0;

symtab[symsize].sym.clazz.base = triple;

currclass = symsize;

decl\_size = 0;

}

void store\_var(int size) {

enter();

symtab[symsize].type = VAR\_TYPE;

symtab[symsize].nesting = nesting;

symtab[symsize].sym.var.base = ++decl\_size;

symtab[symsize].sym.var.size = size;

if (size>1) decl\_size += size-1;

}

void store\_formal(void) {

enter();

symtab[symsize].type = FORMAL\_TYPE;

symtab[symsize].cls = INTSLOT;

symtab[currclass].sym.clazz.num\_args++;

symtab[symsize].sym.var.base = ++decl\_size;

symtab[symsize].sym.var.size = 0;

}

void backpatch(int trip, int val) {

code[trip].second = val;

}

void backpatch1(int trip, int val) { /\* for backpatch the first field \*/

code[trip].first = val;

}

int redundant(char \*opcode, int first, int second) { /\* checks if triple already available \*/

int i;

if (strcmp(opcode, "CON") == 0) { /\* just doing CON as an example \*/

for (i=0; i<constSize; i++)

if (first == const\_pool[i].value) return(const\_pool[i].triple);

if (constSize<CONST\_MAX) {

const\_pool[constSize].value = first;

const\_pool[constSize].triple = triple;

constSize++;

}

return(0);

}

return(0);

}

int emit(char \*opcode, char prefix1, int first, char prefix2, int second) { /\* generates code and handles var count \*/

int old\_triple;

old\_triple = redundant(opcode, first, second);

if (old\_triple>0) return(old\_triple);

if (triple<CODE\_MAX) {

code[triple].prefix1 = prefix1;

code[triple].first = first;

code[triple].prefix2 = prefix2;

code[triple].second = second;

strcpy(code[triple].opcode, opcode);

return(triple++);

}

else {

fatal("code table is too large\n");

return(NONE);

}

}

void ok(int extra) {

while ((yychar!=WHILE) && (yychar!=FOR) && (yychar!=IF) && (yychar!=SEMICOLON) && (yychar!=extra) &&

~YYEOF) yychar = yylex();

}

void dump\_sym(void) {

int i;

printf("\n\nSYMBOL TABLE (size = %d)", symsize);

for(i=1; i<=symsize;i++) {

printf("\n%3d: ", i);

printf("\t%8s",symtab[i].name);

printf("\t%s ",type\_names[symtab[i].type]);

printf("\t%s ",nest\_names[symtab[i].nesting]);

if (symtab[i].type == NONE)

printf("\t\tBase=%d", symtab[i].sym.clazz.base);

else if (symtab[i].type == CLASS\_TYPE) {

printf("\t\tBase=%d", symtab[i].sym.clazz.base);

printf(", Args=%d", symtab[i].sym.clazz.num\_args);

printf(", Locals=%d", symtab[i].sym.clazz.local\_space);

}

else { /\* var or parm \*/

printf("\tClass=%s",symtab[symtab[i].cls].name);

printf("\tBase=%d",symtab[i].sym.var.base);

printf(", Size=%d",symtab[i].sym.var.size);

}

}

printf("\n\n");

}

void dumpcode(int final) {

int i, p1, p2;

final = final && (interp != NULL);

/\* header record is first instruction, number of instructions, size of globals, size of stack \*/

if (final)

fprintf(interp, "%d %d %d %d\n", symtab[MAINSLOT].sym.clazz.base, triple-1, globals, DEFAULT\_STACK);

printf("\n");

for (i=showStart;i<triple;i++) {

printf("%4d: %-8s %c%-3d %c%-3d\n",i,code[i].opcode,p1=code[i].prefix1,code[i].first,p2=code[i].prefix2,code[i].second);

if (final)

fprintf(interp, "%-8s %c%-3d %c%-3d\n",code[i].opcode,p1==' ' ? '#':p1,code[i].first,p2==' ' ? '#':p2,code[i].second);

}

showStart = triple;

}

void fatal(char const \*s) {

printf("nFatal Error: %s\n",s);

dumpcode(0);

}

void error(char const \*s) {

printf("\nError: %s\n", s);

errorNo++;

}

void warning(char const \*s) {

printf("\nWarning: %s\n", s);

warnNo++;

}

void yyerror(char const \*s) {

printf("%s\n", s);

}

#include "KoolProject.lex.c"

int main(int argc, char \*\*argv ) {

int result;

++argv, --argc; /\* skip over program name \*/

if ( argc > 0 ) { yyin = fopen(argv[0], "r"); interp = fopen(strcat(argv[0],".interp"), "w"); }

else yyin = stdin;

result = yyparse();

system("pause()");

return(result);

}