Instructor

David R. Hanson
UCC 318
621-6610
Office Hours by appointment

Required Texts

D. R. Hanson, *DECSystem-10 C Programmer's Manual*, TR 83-12, Dept. of Computer Science, Univ. of Arizona, Aug. 1983.

Optional Texts

D. R. Hanson, *Software Tools User's Manual*, TR 81-20, Dept. of Computer Science, Univ. of Arizona, Dec. 1981.

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, 1976.

R. M. Graham, *Principles of Systems Programming*, John Wiley & Sons, 1975.

P. Calingaert, *Assemblers, Compilers, and Program Translation*, Computer Science Press, 1979.

D. W. Barron, *Assemblers and Loaders*, 3rd edition, Elsevier North-Holland, 1978.

N. Wirth, *Algorithms + Data = Programs*, Prentice-Hall, 1976, chap. 5.

Grading

1/2 programming assignments; 1/6 each of 3 exams (including the final). These weights are subject to minor variation.

Attendance

Attendance is expected but will not be recorded. Students are, however, fully responsible for all material presented in class. Since much of the material is not available in textbooks, class attendance is strongly recommended.

Rules

Exams will be scheduled in advance; unless prior arrangements are made, a grade of zero will be recorded for missed exams.

Students are encouraged to discuss programs in general terms and to help one another find bugs in existing programs. Writing code for use by another or using another's code is cheating, however.

Except in extraordinary circumstances, late programs are *not* accepted.

Csc 453 Translators and System Software, Fall 1983
Schedule

date	*topic*	*assignments*	*due*
Aug 23	introduction; compiling		
25	compiling: grammars	parser	
30	compiling: declarations		
Sep 1	compiling: procedures		
6	compiling: statements		
8	compiling: expressions		
13	compiling: expressions		
15	compiling: expressions		
20	code generation	code generator	parser
22	code generation		
27	code generation		
29	code generation		
Oct 4	exam		
6	linking	link [– 450, due 11/1]	
11	exam review/linking		code generator
13	linking		
18	linking		
20	no class		
25	linking		
27	linking		
Nov 1	linking	library searching	link [150]
3	linking		
8	exam		
10	debugging	db [300]	
15	exam review/linking		library searching
17	debugging		
22	debugging	cdb	
24	Thanksgiving		
29	debugging		db
Dec 1	debugging		
6	debugging		cdb
16	final exam (10:30 am)		

Csc 453 Translators and System Software, Fall 1983
Exam 1

October 4, 1983

Name: Bill Mitchell

There are four problems on this exam. Each problem is worth 25 points. This is *open book* exam.

Please attempt to solve all problems in the space provided. Show your work in arriving at solutions; partial credit will be given. There is an extra page at the back of the exam. Be brief.

Some of the problems ask for DEC-10 instruction sequences. The following instruction examples provide all of the necessary details.

instruction	*effect*
move r,x	load the contents of the word at address 'x' into register 'r'
movem r,(r1)	store the contents of 'r' at the address contained in 'r1'
movem r,-5(14)	store the contents of 'r' at the address 5 plus the contents of 14
movei r,x	load the address 'x' into 'r'
movei r,$4	load the address of the label '$4' into 'r'
jrst $4	jump to label '$4'
jrst (r)	jump to the address given by the contents of 'r'
adjsp r,n	increment the stack pointer in 'r' by 'n' words
adjsp r,-n	decrement the stack pointer in 'r' by 'n' words
add r1,r2	add the contents of 'r2' to the contents of 'r1'

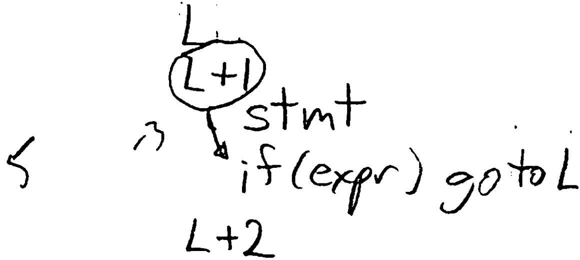
Good luck.

22
20
10
20
72

1. In addition to the **while** and **for** loops, C also has a **do** loop in which the test is at the bottom of the loop:

do *stmt* **while** (*expr*)

(a) (8 pts) Give the 'code template' for the **do** statement, showing the correct placement of the necessary labels.



5

(b) (12 pts) Write the code necessary for parsing this statement, including the correct semantics (i.e. calls to *walk*, *deflabel*, *jump*, etc.). Make sure your code interacts properly with **break** and **continue** statements.

```
dostat()
{
    int lab;

    lab = genlabel(3)
    t = stok();          deflabel(lab)
    stmt(lab);           deflabel(lab+1)
    if (t != WHILE)
        err("while' expected", "");
    else
        t = stok();
    mustbe('(');
    walk(cnode(expr()), lab, 0);
    deflabel(lab+2)
    mustbe(')');
}
```

12

22

(c) (5 pts) How does adding this statement affect the *first* and *follow* sets?

"do" is added to the First + Follow sets for stmt.

5

widen is called for all ints

2. C permits expressions like

```
a = b < c;
```

which sets *a* to 1 if *b* < *c* and to 0 otherwise. The type of the result for the comparison operators, &&, ||, and unary ! is integer. This feature can be implemented by 'converting' conditional expressions to integer expressions using the code template

attempt at a "big hint"

```
        if (e) goto L
        c.cc = 0
        goto L+1
L       c.cc = 1
L+1
```

where *c.cc* is a global variable allocated elsewhere. Describe the changes necessary to implement this feature in our compiler. Be specific; give the relevant details concerning type checking and expression tree traversal, using code fragments or pseudo-code where necessary.

If widen (actually, do in rvalue) sets a T_COND mode, it adds a '#' node, later intercepted by walk that generates the code shown above.

good

In walk;

case '#':

r1 = walk(p->e_left, 0, 0); — via additional nodes (best added in encode)

vague about template.

(Generate code to check value of r1 and produce a 0 or 1 value as result of exprcode in r.)
This value is later used as an integer in any computations.

10

3. Generating code for the DEC-10 is simplified because it has many registers. Suppose, however, that it had only 3 registers. Describe how you would handle this situation, especially when you run out of registers. Your scheme should affect only the bottom half. Be specific about details concerning types, if any, and what changes to the generated code must be made to use your scheme. Keep your scheme simple, but be sure it works for recursive procedures.

In brief, use words in the procedure frame as temporary storage and use one of the three registers as a scratch register. More specifically, the procedure frame will be:

-1
move

unnecessary

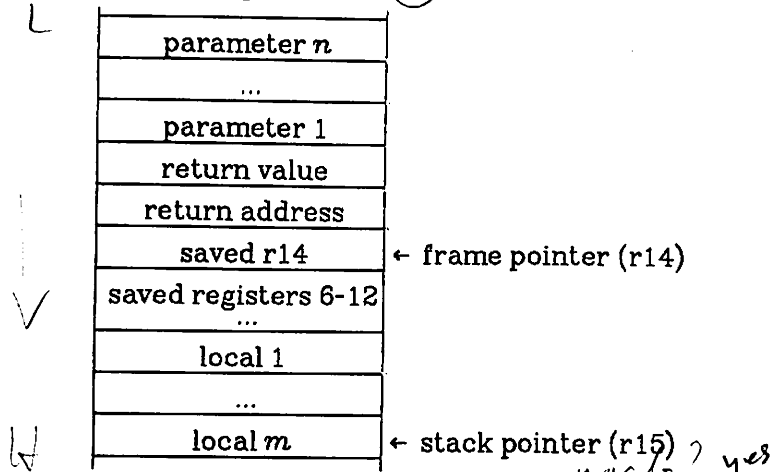
movei 5,X
move 6,(5)

how is this space allocated? fixed #?

In the new scheme, ralloc manages two real registers and some suitably large number of mem. registers. If a real register is available ralloc returns 1 or 2, else it returns a negative number corresponding to a memory register. When the bottom half encounters a read reference neg. reg. number it generates code to get the right value out of the stack frame and puts it into the scratch register which is then used in a computation. Write references do the reverse, calculating the result in the scratch register + then storing it back in the frame.

20

4. Suppose the frame layout for procedures was [is?] as follows.



Note that the parameters are in the *reverse* order. Assuming that the DEC-10 does *not* have subroutine calling instructions, give appropriate instruction sequences for a call to an integer function f with n parameters and m locals, the entry sequence generated by *procbeg* for f and the exit sequence generated by *retcode* for f. As suggested by the diagram, your sequences must preserve the contents of registers 6-12, but may use registers 2-5 as desired. You do *not* have to write any C code; just give the relevant instruction sequences. If you are unsure of specific details of the DEC-10 instruction set, use English as necessary to make the intent of your sequences clear.

Calling sequence:
The value for each parm is calculated and put on the stack via: (in correct order)

```
move   r, (sp)      ← 15, parm value
adjsp  15,1
```

after parms are on stack

```
      adjsp  15,2
      movei  2,$1+1
      movem  2,(15)
      movei  3,f
$1:   jrst   (3)
```

Entry Sequence:

```
adjsp  15,7
movem  6,-7(15)
       7,-6(15)
       :
movem  12,0(15)
adjsp  m,15
```

(you can combine moves & move adjsp) (where do you set reg. 14? -5)

Return:
Assume ret val is in r1 (use frame pointer)

```
movei  1,-(m+7+1)(15)
move   2,(1)          # ret addr in 1
adjsp  -(m+7+2+n),15
jrst   (2)
```

(restore 14?)

assumes sub is called w/ correct # of args. (can do in caller.)

(20)

1. (a) The code template is

```
L       stmt
L+1     if (expr) goto L
L+2
```

(b) The case

```
case DO:
        dostat(loop = genlabel(3));
        break;
```

is added to *stmt*, where *loop* is the current loop handle. *dostat* is

```
dostat(lab)
int lab;
{
        t = gtok();
        deflabel(lab);
        stmt();
        deflabel(lab + 1);
        mustbe(WHILE);
        mustbe('(');
        walk(cnode(expr()), lab, 0);
        mustbe(')');
        deflabel(lab + 2);
}
```

(c) **do** affects the *first* and *follow* sets in just like **while** and **for**. **do** must be added to *first*(stmt) and to *follow*(stmt).

2. 'Converting' a conditional expression to an integer expression can be done in *rvalue*:

```
struct node *rvalue(p)
struct node *p;
{
        ...
        if (p-> e_type == T_COND)
                p = node('?', T_INT, p, NULL);
        return (p);
}
```

Could use overloaded CVI

When the ? operator is encountered in *walk*, the appropriate code is generated for the left operand, constructing the trees as necessary, and the result of computing the rvalue of *c.cc* is returned:

```
        ...
        case '?':
                walk(p-> e_left, lab = genlabel(2), 0);
                p1 = node(O_ID, T_INT|T_ADDR, lookup("c.cc"), NULL);
                walk(node(O_ASGN, T_INT, p1, constant("0", T_INT)), 0, 0);
                jump(lab + 1);
                deflabel(lab);
                walk(node(O_ASGN, T_INT, p1, constant("1", T_INT)), 0, 0);
                deflabel(lab + 1);
                r = walk(rvalue(p1), 0, 0);
                break;
```

3. A simple approach to dealing with a limited supply of registers is to provide a fixed number of temporaries and allocate them as if they were registers. When emitting an

instruction, it may be necessary to load temporary values into real registers and to store the previous values. Thus, *exprgen* should return an index into an array of resources, which may either registers or temporaries. This is necessary so that when a register is 'spilled' into a temporary the change affects all uses of that register. In order to work properly for recursive procedures, the temporaries must be located in the frame. Either a fixed number of temporary locations can be allocated in every frame (above the locals, for example), or only enough space for those temporaries actually used can be allocated. In the latter case, the size of the local frame is not known until the end of the procedure, which may require that the entry point be at the bottom of the procedure or that a constant containing the frame size be generated and referenced by the entry sequence.

4. The 'frame operations' are

```
procedure call:
        adjsp   15,n+3          ; allocate space for arguments & linkage info
        ...
        movem r,-i-2(15)        ; argument i
        ...
        movei   2,L             ; get return address
        jrst    f               ; transfer to procedure
L:      move    r,-2(15)        ; get return value
        adjsp   15,-n-3         ; remove arguments

procedure entry:
f:      movem 2,-1(15)          ; save return address
        movem 14,0(15)          ; save frame pointer
        move    14,15           ; establish new frame pointer
        adjsp   15,m+7          ; allocate space for locals & registers
        movem 6,1(14)           ; save register 6
        ...
        movem 12,7(14)          ; save register 12

procedure exit:
        movem r,-2(14)          ; set return value
        move    6,1(14)         ; restore register 6
        ...
        move    12,7(14)        ; restore register 12
        move    15,14           ; deallocate local portion of the frame
        move    14,0(15)        ; restore frame pointer
        move    2,-1(15)        ; get return address
        jrst    (2)             ; return to caller
```

Csc 453 Translators and System Software, Fall 1983
Exam 2

November 10, 1983

Name: Bill Mitchell

There are four problems on this exam. Each problem is worth 25 points. This is *closed book* exam.

Please attempt to solve all problems in the space provided. Show your work in arriving at solutions; partial credit will be given. There is an extra page at the back of the exam. Be brief.

Good luck.

Problem 4 asks about changes to the code generator, *cgen.c*. The routines in *cgen.c* and their current output are as follows.

routine	*output*
defconst(p)	.seg lit .def -s $*L* lit *offset* + char, int, float constants: bit pattern char * constants: 0331100000001 lit *offset* + 9-bit bytes, 4/word
deflabel(lab)	.def -s $*lab* *segment* *offset* +
exprcode(op,r1,r2,lab,p)	code for *op*
global(p)	.def *p->s_name* data *offset* +
jump(lab)	jrst $*lab*
procbeg(p)	.seg code 0331100000000 lit *offset* + .seg lit *p->s_name* given in 9-bit bytes, 4/word .seg code .def *p->s_name* code *offset* + jsp 4,c.savr adjsp 15,*size of locals* (*procbeg* must also compute *s_offset* values for local variables)
procend(x)	no output
progbeg()	no output
progend(x)	.len code *length* .len data *length* .len lit *length*
retcode(r)	move 1,*r* jsp 4,c.retr
rfree(r)	no output free resource *r*

25
25
17
19
86

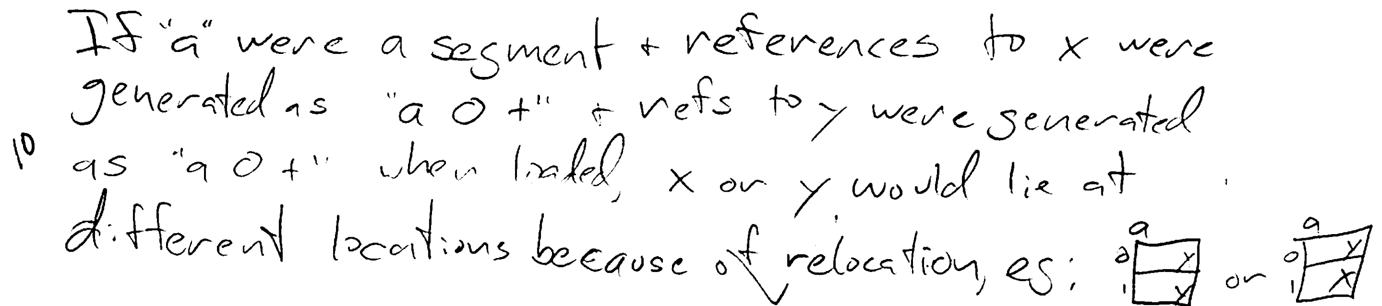
good job!

1. Named (and unnamed) common in Fortran permits separately compiled subroutines to access the same storage, e.g. in

```
subroutine A        subroutine B
common /a/ x        common /a/ y
 ...                 ...
end                 end
```

x in A and y in B refer to the same cell, which is resolved during linking.

(a) (10 points) Named common *cannot* be handled by simply making the common block name (a in the example above) a segment. Explain why not.



If "a" were a segment + references to x were generated as "a 0 +" + refs to y were generated as "a 0 +" when loaded, x or y would lie at different locations because of relocation, eg:

10

common.name.len

(b) (15 points) Can you devise a way to handle common using the *existing* facilities of link? Explain. Feel free to assume and state compiler and loader conventions to support your scheme, if that is necessary.

15/ yup!

The basic problem is to avoid relocation. It almost seems that having the compiler use a separate segment for each common and then outputting .len w/ 0 will do the trick. The problem w/ this is that the loader won't know how long the common blocks are. To solve this, modify the compiler to make common segment names of the form "common.name.len" (actual name of common), generate refs to these segments for vars which lie in it, and let the loader sort out everything. (Basically, the loader must find the longest length and use it.)

good

Note that the arbitrary limitation on segment names length must be removed if source-lang. limitations are to be avoided.

25

2. Suppose you have an object file, *table.o*, that contains routines and data for a general-purpose symbol table manager. *table.o* defines the symbols **lookup**, **install**, and **initialize**, which are the routines for accessing the table. You are building a system in which separate symbol tables are needed in each of two modules, *command.o* and *lex.o*. Describe how you could use **link** to construct the final object file, named **prog.out**. You *may* add new command line options to **link**, but you *may not* change any of the object files or link object code in any way.

link -s "lookup, install, initialize" command.o table.o -o command.o.o
link -s "lookup, install, initialize" lex.o table.o -o lex.o.o
link command.o.o lex.o.o -o prog.out

-s "idlist" causes the SUPRESS bit to be set for each identifier in idlist. (Same as described in paper, I think)

P?

25

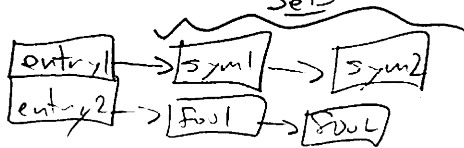
3. Suppose the entries in a library are *not* in topological order, but the index at the beginning of the library contains lines of the form

> **# *symbol name***
> **– *name1 name2***

Lines of the first form indicate that *symbol* is defined in entry *name*; lines of the second form indicate that entry *name1* requires symbols defined in entry *name2*. Give *pseudo-code* for *lib(file)*, which completely searches the library in *file* in *one* sequential pass over *file*. Be sure to describe the data structure that results from reading the index; do *not* give the details of reading the index itself, however.

Reading the index produces ① a topologically sorted list (called dlist) of symbol dependencies built from the "name1 name2" (symbol_name) information, and ② a table mapping entry-definition information (sets), e.g.:

overkill — how does –name1 name2 lines give this list? (by {unioning / concatenating} the respective lists, I suspect) −5



which is called etab.

Alg:

R − (DUS)

```
for each entry in index {
    if etab[entry] ∩ undefined symbols ≠ ∅ then
        pass1(entry)
    else skip this entry   -3
}
```

{ Find first symbol in dlist in (R−(DUS)) and add it + all following symbols to R.

(17)

4. Typically, each entry in a library is compiled separately and then inserted into the library. Thus, for example, a collection of functions is usually kept one function per source file, and each one is compiled and inserted into the library individually. A more convenient approach is to have a compiler option that, given a source file containing only functions (no global data), causes the compiler to *generate* a library whose entries are the link object code for those functions. Describe what changes to the code generator (*cgen.c*) and to link are necessary to implement this strategy. Give enough detail so that a person familiar with *cgen.c* and link object code would understand how to make the necessary changes.

Overview: Have the compiler generate code for each function into a separate file and then ~~merge~~ concatenate the files into a library.

More Specifically,

Add a data structure (ctab) that maintains information about the routines called by a particular routine.

procbeg causes an entry to be made in ctab for the current procedure and establish a global ptr to the entry. When walk encounters a CALL node, it adds (but not redundantly) the routine being called to the current ctab list. When progend is called, it outputs the index entry by scanning ctab + traversing the call lists. Then, ctab is scanned again + each temp file is appended to the library.

Note that it is a simple matter to produce a top. sorted library.

BTW - No link mods at all. (!)

unnecessary for index.

don't have to wait till prgm.

details you forgot: 1) literals -3 2) reset locations ctrs: -3

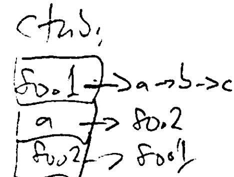
good

(19)

Ex:

foo.1()
{
a()
b()
c();
}
a()
{
foo.2()
}
foo.2()
{
foo.1()
}

ctab:



1. (a) Segments alone cannot be used for labeled common because **link** concatenates segments of the same name from different files and relocates references to segments. In the example given, if **a** is made a segment, linking the two files would result in **a** being two words in length and **y** being incorrectly defined as '**a 1 +**'. A common block is more like a segment that may be defined in many files but is 'allocated' only once. That is, subsequent occurrences of a common block should simply refer to the first one.

(b) Clearly, **link** itself cannot handle common. It is, however, possible to establish compiler and loader conventions so that the compiler can produce segments that are handled as usual by **link** but are recognized by the loader as common blocks and treated accordingly. Perhaps the simplest scheme is to represent a common block **x** as a segment named **comm_x_n**, where **n** is the length of the common block, and to specify **0** on the associated **len** command. Specifying a length of 0 avoids relocation. The loader can obtain the correct length from the segment name. In example in the problem, the output from the common declaration in **A** would be

```
.seg comm_a_1
.def x comm_a_1 0 +
.len comm_a_1 0
```

Could pass length via unrecog command, but wouldn't know where length will come out

and the output from **B** would be

```
.seg comm_a_1
.def y comm_a_1 0 +
.len comm_a_1 0
```

Another, similar, approach to is represent common block **x** as a segment named **comm_sub_x**, where **sub** is the subroutine name in which **x** is referenced. In the example in the problem, subroutine **A** would define a segment named **comm_A_a** and **B** would define a segment named **comm_B_a**. Using different names keeps **link** from concatenating the segments. The loader, however, can treat segment names of the form **comm_sub_x** properly by removing the **comm_sub** prefix. Thus, **comm_A_a** and **comm_B_a** will be assigned the same base address. Note that their lengths could also be checked (they should be equal).

2. The solution to this problem is an application of 'link-time binding' described on page 361 of the **link** paper. A copy of *table.o* can be linked with *command.o* and with *lex.o*, providing the symbols **initialize**, **lookup**, and **install** are suppressed, i.e. defined with the **-s** option. The resulting two object files can then be linked to form *prog.out*. Since we cannot alter the object files, the command line option

```
-s symbol,symbol,...
```

which sets the **SUPPRESS** bit for the given symbols is added. Then the object files for *command.o* and *table.o* and for *lex.o* and *table.o* are constructed as by the commands.

```
link command.o table.o -s initialize,lookup,install -o command.out
link lex.o table.o -s initialize,lookup,install -o lex.out
```

The final object file is constructed by linking *command.out* and *lex.out*:

```
link lex.out command.out -o prog.out
```

3. The index lines '- *name1 name2*' can be used to augment the entry structure with a list of other entries that are needed if the entry of interest is linked. The relevant C declarations are

```
struct entry {
        char *e_name;
        int e_needed;
        struct enode *e_family;
        struct entry *e_link;
}
```

If name1 must be loaded, then name2 must be loaded

```
struct enode {
        struct entry *f_entry;
        struct enode *f_link;
}
```

e_family is the head of a list of pointers to entry structures for the other entries in the same 'family', i.e. those needed if this entry is linked. This list is constructed as index lines of the form given above are read. **lib(file)** can then be written as follows.

```
lib(file)
char *file;
{
        struct symbol *p;
        struct entry *q;
        struct enode *f;

        read index and build entry structure;
        for each (symbol p in (L∩R)-D) {
                p->s_entry->e_needed++;
                for (f = p->s_entry->e_family; f; f = f->f_link)
                        if (q = f->f_entry)
                                q->e_needed++;
                }
        for each (entry in the library) {
                q = associated entry structure;
                if (q->e_needed)
                        pass1(this entry);
                else
                        skip this entry;
                }
}
```

4. The code generator can generate libraries by emitting code for each function and for the index into temporary files and then concatenating the temporary files at the end of compilation. In doing so, the compiler must insure that each temporary file is complete, e.g. that each includes the necessary literals. Specifically, the major modifications to the routines in *cgen.c* are as follows.

progbeg opens a temporary file for the index and initializes a list of structures that describe the temporary files.

procbeg opens a new temporary and adds a node to the list initialized by *progbeg*. It also writes an index entry for the procedure to the index temporary file.

procend must insure that the output is complete. To do so, it executes

```
foreach(defconst, CONST, 0);
```

to include the necessary literals, and emits the appropriate len commands. It also must *reset* the location counters to 0, since each library entry is a separate object file.

progend copies the temporary files, beginning with the index, to the standard output, prefacing each one with the appropriate archive header, and deletes them.

defconst should set the **p->s_offset** field to 0 so that future references to the constant in other functions will cause its inclusion ~~in the object file~~. (See *cgen.c*). *cgen.c[1055,21]* includes the 60 or so lines needed to implement this feature, which is activated by the -l option to the compiler.

~~Global data has one entry file~~
Library entry for each global data item
Var name lengths

Csc 453 Translators and System Software, Fall 1983
Exam 3

December 16, 1983

Name: Bill Mitchell

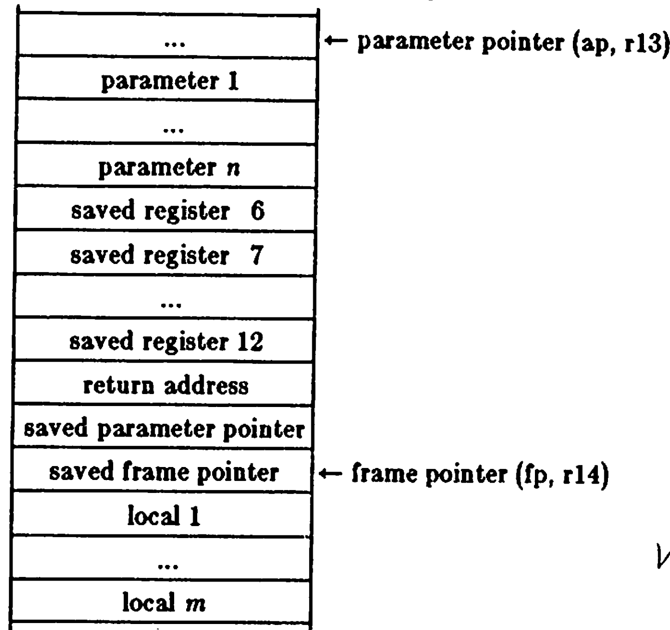
There are four problems on this exam. Each problem is worth 25 points. This is *closed book* exam.

Please attempt to solve all problems in the space provided. Show your work in arriving at solutions; partial credit will be given. There is an extra page at the back of the exam. Be brief.

Good luck.

Useful Information

The layout of an activation record for a procedure with *n* parameters and *m* locals is as follows.



Parameters and locals are addressed by constant offsets from the 'parameter pointer' (register 13) and the 'frame pointer' (register 14), respectively.

findframe(n) positions a global frame pointer, *cursor* (a *frame* structure), to the *n*th frame from the top of the stack. It returns 0 if the positioning fails and 1 otherwise.

Possibly useful definitions are as follows.

```
struct frame {                      /* frame variables */
        struct symbol *f_func;      /* function */
        int f_depth;                /* depth from the top */
        int f_nargs;                /* number of arguments */
        int f_ap;                   /* argument pointer */
        int f_fp;                   /* frame (local) pointer */
        int f_sp;                   /* stack pointer */
};
struct breakpt {                    /* breakpoint information */
        ADDRESS b_addr;                     /* breakpoint address */
        int b_inst;                 /* instruction at breakpoint */
        int b_jumps[2];             /* jumps back into main code */
        struct symbol *b_func;      /* function in which breakpoint is set */
        int b_epoint;               /* execution point in function of breakpoint */
};
```

```
struct state {                          /* machine execution state */
        ADDRESS x_pc;                   /* location counter */
        int x_regs[16];                 /* registers */
        int x_level;                    /* call level */
        int x_trace;                    /* trace counter */
};
struct symbol {                         /* symbol table entries */
        char *s_name;                   /* entry name */
        int s_flags;                    /* flags (see below) */
        int s_type;                     /* data type (see below) */
        int s_size;                     /* size of object in bits */
        int s_offset;                   /* activation frame offset in words */
        struct symbol *s_link;          /* link to next table entry */
#ifdef CDB
        int *s_points[1];               /* execution points for functions */
#endif
};
struct node {                           /* expression nodes */
        int e_op;                       /* operator */
        int e_type;                     /* type of result */
        struct node *e_left;            /* left operand */
        struct node *e_right;           /* right operand */
        int e_result;                   /* execution result */
        int e_count;                    /* reference count */
};

/* symbol table entry flags */
#define DEF     0001                    /* symbol is defined */
#define REF     0002                    /* symbol is referenced */
#define GLOBAL  0004                    /* symbol is a global */
#define LOCAL   0010                    /* symbol is a local */
#define PARAM   0020                    /* symbol is a formal parameter */
#define CONST   0040                    /* symbol is a constant */
#define KEYWORD 0100                    /* symbol is a keyword */
#define ARRAY   0200                    /* symbol is an array */
```

1. Suppose the breakpoint command was extended to be

   b *name n* { **cdb** *commands...* }

where the braces enclose a sequence of **cdb** commands, separated by semicolons, to be executed when the breakpoint is encountered. If the {...} specification, is omitted, a **c** command is executed and commands are accepted from the user, as usual. For example,

   b strcmp 3 { –;C;j }

would set a breakpoint at the 4th execution point in *strcmp* and execute the –, **C**, and **j** commands upon encountering the breakpoint. Describe what changes or additions you would make to **cdb** to implement this extension. Be specific.

ptr

Add something to the breakpt structure that allows a string to be associated w/ a breakpoint. When {–} is specified, b_cmd is pointed at a copy of the string sans the brackets, otherwise, b_cmd = NULL.

i.e. char *b_cmd
allocation

Next, restructure the command loop of cdb so that the user can type strings like –;C;j at the prompt. This is fairly trivial, just split the line at ;'s (watch out for quoted strings!) and make a queue of these lines. Input lines are added to the queue and cdb takes lines from the queue (as if the user had typed them in) and executes them.

With the new command parsing in hand, modify the 'j' case (after the resume) to check the b_cmd field of the breakpoint hit; if NULL, just do a 'c', otherwise, point the input line pointer at the b_cmd string and jump to the point just after where commands are read in. Thus, it looks like the user typed in the commands associated w/ the breakpoint.

√

2. It is useful to be able to call user-defined C functions during the evaluation of expressions typed to **cdb**. For example, typing

p lookup("int", &root)   f(x1,

at a breakpoint in *main* in *test6.c* would print the index of the entry for *int*. During execution of a user-defined function in response to a cdb expression, breakpoints are *not* set. Describe the changes you would make to cdb to implement this feature. Assume user-defined functions must be executed at full speed, i.e. you cannot interpret the code. Be specific about the modules in which changes are necessary and the details concerning stack usage.

Add two global variables;
int argno;   ← Max value for # of paramters, the 10
int fargs[NPARAM];   seems to limit this to about 7.

In eval;

argno = 0;

In exprcode(op, r1, r2, p)

Case O_ARG;

fargs[argno++] = results[r1].w-value;
break;

Case O_CALL; {
int (*f)()
f = (int) p->f_func->s_offset;
rval = (*f)(fargs[0], fargs[1], ... fargs[NPARAM-1])
results[r] = rval
break;
}

allocated   decrement argno.

Problems; runs on cdb stack; could do switch, but this seems like a reasonable solution. Also, could craft stack by hand to avoid (fargs[0] ...), but again, has bad win/lose ratio. Also limited to # of args, but since the 10 doesn't seem to like more than 7, that doesn't seem like a problem. (Hand crafting stack avoids max args problem.) Over.

25/

This doesn't allow for $f(x(1), y(2))$ ← it's very close, however.

but I can't figure it out. Perhaps something w/ allocating parameter values in the results array and maintaining a list of the elements w/ the values and then filling in fargs at the point of call.

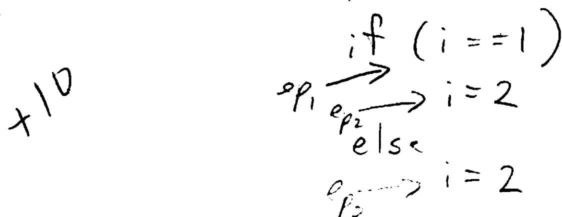
3. Suppose cdb had a k command, which caused the next expression to be executed as if a breakpoint was set at the next execution point. Successive k commands could then be used to 'single step' the program.

(a) (10 points) Consider the following implementation of the k command.

> **"If execution is currently halted at execution point *n* in function *f*, set a breakpoint at execution point *n+1*. Add a *b_flags* field to the breakpoint structure and set a flag indicating the breakpoint at *n+1* is a 'single-step' breakpoint. The k command is then identical to the J command, except that 'single-step' breakpoints are removed after they are encountered."**

This implementation does *not* work; explain why.

Execution points are non sequentially reached, because the flow of control is not necessarily sequential.
Consider:



If i==0, the flow is 1, 3; ep2 is never reached. (Hence, a k at ep1 might go for a good ways.)

(b) (15 points) Ignoring the complications implied by the return statement, describe what modifications to the implementation outlined in part (a) are necessary to correctly implement the k command. Be specific about any changes you make to the compiler's *output* or to cdb. Do *not* describe any changes to the compiler itself; concentrate on the changes in its output.

The compiler needs to tell the debugger about execution points (epi) where the code might cause epi+1 to not be reached "next", and for such eps, which eps might be next. For example, in an if stmt, the first ep in either arm might follow the clause; In a while stmt, the body might be skipped. The for stmt is also a problem, and break + continue are also trouble.

So, modify the compiler so that for each ep it outputs the address of the first instruction and follows the address w/ negative addresses of instructions that begin execution points that might be reached next.

↓ Over

25

inst. addrs

For example:

```
ep0, 000 -> i = f();
ep1, 010 -> if (i == 1)
ep2, 014 ->     i = 2;
            else
ep3, 017 ->     i = 3;
ep4, 021    j = i*2
```

ep array:

```
ep0 ->   1
ep1 ->  010
        -014
        -017
ep2 ->  014
        -021
        017
ep3 ->
ep4 ->  021
```

No neg #'s if next ep is sequential.

The k command finds the current ep & determines addresses of all the possible next eps. It makes a list of them, sets breakpoints at all of them. global It does a switch, a breakpoint is hit, it cleans up the points using the list & returns to the cdb command level.

sym table output only

4. C permits 'register' declarations, e.g.

register int n; only locals

which instruct the compiler to use machine registers (instead of allocating space in the frame) for the indicated local variable. Suppose our compiler used registers 6-12 as necessary for such variables, and set the REGISTER flag in the symbol table entry field, *s_flags*, to indicate that *s_offset* contained a register number instead of a frame offset. Describe the steps necessary to compute the *lvalue* of a register variable during the evaluation of an expression, i.e. the steps in the O_ID case in *exprcode*. Be careful!

```
O_ID;
    if (ISREG(p))          <- ptr to symbol
        lvalue = &(vp->x_regs[p->s_offset]);
                   ^ ptr to use-state
```

This seems to be enough. If the value is required, it is gotten from the saved register value. If it is modified, the saved value is written over and subsequently restored by resume.

for frame 0, not for frame i, i>0

,20

5/

1. Breakpoint commands can be handled by adding a *b_cmd* field to the breakpoint structure that points to the command string given in braces. This field is set on a **b** command, and, upon encountering a breakpoint, its contents are appended to an input buffer, substituting newlines for semicolons. The main command loop is modified to take commands from the command buffer until it is empty, and then resume accepting commands from the terminal. This approach preserves the current method for handling errors and permits the breakpoint commands to set and remove breakpoints correctly.

2. User-defined functions can be called by simply calling them from **cdb**, i.e. using **cdb**'s stack. This is possible because the locations of the arguments and locals, given by the frame offsets, are independent of which stack is used, providing the arguments are pushed onto the right stack. The values of the actual arguments can be saved in an array during evaluation in *exprcode*:

```
union words args[MAXARGS];
int argptr = 0;
 ...
case O_ARG:
        args[argptr++] = w1p->w_int;
        break;
```

Calling the function amounts to passing the correct number of arguments and getting the address of the function:

```
union word *ap;
int (*f)();
 ...
case O_CALL:
        ap = &args[argptr-r2]; /* r2 = argument count */
        f = p->e_left->e_left->s_offset;
        switch (r2) {
                case 0:  wp->w_int = (*f)(); break;
                case 1:  wp->w_int = (*f)(*ap); break;
                case 2:  wp->w_int = (*f)(*ap, *(ap+1)); break;
                ...
                case MAXARGS:
                        wp->w_int = (*f)(*ap, *(ap+1), ..., *(ap+MAXARGS-1));
                        break;
                }
        argptr -= r2;
        break;
```

This code takes advantage of the architectural property that all scalar types occupy 1 word of storage.

3. (a) The proposed implementation of the **k** command does not work because it assumes that the order in which execution points are encountered during *execution* is the same as during *compilation*, which is incorrect. Consider, for example, the following fragment and its execution points.

```
while ( i < 10)
        ↑0
        f(i++);
        ↑1
g();
↑2
```

Stepping through the loop with the **k** command implemented as described will only step through the first iteration of the loop because the execution point following point 1 is number 2. During *execution*, however, control flows to the execution points in the sequence 0, 1, 0, 1, ..., 0, 2.

(b) The flaw in the implementation strategy is that the notion of "successor" execution point is incorrect. This can be corrected by having the compiler generate the list of execution points and their *execution-time* successors. For example, the *e_points* array could be organized so that it contains the address of the

corresponding execution point followed by a list of the addresses of its successor execution points, terminated by -1. (Currently, this list will have 0, 1, or 2 entries). The entire list is terminated by a 0 execution point address, as before. For the code fragment given above, the output for the *e_points* array might be as follows.

```
code 056 +
code 065 +
code 075 +
-1
code 065 +
code 056 +
-1
code 074 +
-1
0
```

As shown, execution point 0 is at **code 056 +** and its execution-time successors are points 1 and 2, which are at **code 065 +** and **code 075 +**, respectively. Likewise, point 1 is at **code 065 +** and its successor is point 0, which is at **code 056 +**, and point 2 is at **code 074 +** and it has no successors. Given this output, the strategy described in part (a) can be generalized to set 'single-step' breakpoints at all of the *successors* of the current breakpoint. This approach can also be extended to handle **return** statements and flowing off the end of a function.

4. The basic question of this problem is "where is register *r* for frame *i*", where frames are numbered beginning at 0 from the top of the stack. The answer depends on the value of *i*. If *i* is 0, register *r* is stored in the *x_regs* fields of the user's program state. Because registers are saved by the callee during the function entry sequence, if *i* is non-zero, register *r* is stored in the frame for the function called by the function associated with frame *i*, i.e. register *r* is stored at offset *fp*-15+*r* in frame *i*-1. The appropriate code for the O_ID case in *exprcode* is as follows.

```
case O_ID:
        q = (struct symbol *) p->e_left;
        if (q->s_flags&GLOBAL)
                wp->w_addr = q->s_offset;
        else if (q->sflags&REGISTER) {
                if ((i = cfrp->f_depth) == 0)
                        wp->w_addr = &up->x_regs[q->s_offset];
                else {
                        findframe(i - 1);
                        wp->w_addr = cursor.f_fp - 15 + q->s_offset;
                        findframe(i);
                        }
        ...
        break;
```

C Parser

Design and implement a recursive descent parser for the subset of C defined by the grammar given below and the attached syntax diagrams. Divide your parser into modules defined by the following files.

name	purpose
* *cc.h*	definitions
cexpr.c	expression parsing – 530
* *cgen1.c*	symbolic code generation
* *clex.c*	lexical analysis
* *cmain.c*	main program
cparse.c	declaration parsing – 360
cstat.c	statement parsing 180
* *csym.c*	symbol table management

Both the source and object files (*.rel* files) are available on [1055,21] for those files indicated by an asterisk. You are free to copy and examine the source code as desired, but you should load your program with the object files.

Syntax

Your parser must accept a subset of C defined by the following extended BNF grammar in which alternatives are separated by vertical bars, i.e. *a*|*b* stands *a* or *b*, optionality is indicated by square brackets, i.e. [*a*] stands for *a*|ε, repetition is indicated by curly braces, i.e. {*a*} stands for ε|*a*|*aa*|*aaa*|..., and parentheses are used for grouping, i.e. (*a*|*b*)*c* stands *ac*|*bc*. Identifiers for which there is no production and operators denote terminal symbols. When braces, brackets, vertical bars, or parentheses are used as terminal symbols, they are enclosed in quotes to distinguish such usage from their grammatical use.

```
prog   : { dcl ; | func }

dcl    : type dclr { , dclr }

dclr   : ptr id [ '[' [ con ] ']' | '(' ')' ]

type   : char | float | int

ptr    : { * }

func   : [ type ptr ] id '(' [ id { , id } ] ')' { dcl ; } '{' { dcl ; } { stmt } '}'

stmt   : expr ;
       | if '(' expr ')' stmt [ else stmt ]
       | while '(' expr ')' stmt
       | for '(' [ expr ] ; [ expr ] ; [ expr ] ')' stmt
       | return [ expr ] ;
       | break ;
       | continue ;
       | '{' { stmt } '}'
       | ;

expr   : expr [ binop ] = expr
       | expr ( relop | binop ) expr
       | term
```

ptr – returns # of *s for use in symbol table.

Assume untyped fcns

do stmt while(expr)


```
term   : term ( ++ | -- )
       | ( ! | * | & | - | ~ | ++ | -- ) term
       | term '[' expr ']'
       | term '(' [ expr { , expr } ] ')'
       | id
       | con
       | fcon
       | ccon
       | scon
       | '(' expr ')'

relop  : == | != | <= | >= | < | > | && | ||

binop  : ^ | & | '|' | << | >> | + | - | * | / | %
```

Syntax diagrams corresponding to each of the productions in the above grammar are attached and should be used to guide the implementation of your parser. The corresponding *first* and *follow* sets for the above grammar are as follows.

```
first(prog)    = first(dcl) ∪ first(func) ∪ { ε EOF }
               = { char int float id EOF }

first(dcl)     = first(type)
               = { char int float }

first(dclr)    = first(ptr) - { ε } ∪ { id }
               = { * id }

first(type)    = { char int float }

first(ptr)     = { * ε }

first(func)    = first(type) ∪ { id }
               = { char int float id }

first(stmt)    = first(expr) ∪ { if while for return break continue '{' ; }
               = { ! * & - ~ ++ -- id con fcon ccon scon ( if while for
                   return break continue '{' ; }

first(expr)    = first(term)
               = { ! * & - ~ ++ -- id con fcon ccon scon ( }

first(term)    = { ! * & - ~ ++ -- id con fcon ccon scon ( }

follow(prog)   = { EOF }

follow(dcl)    = { ; }

follow(dclr)   = { , } ∪ follow(dcl)
               = { , ; }

follow(type)   = first(dclr) ∪ first(ptr) - { ε } ∪ { id }
               = { * id }

follow(ptr)    = { id }
```

do

```
follow(func)   = first(prog) ∪ follow(prog)
               = { char int float id EOF }

follow(stmt)   = first(stmt) ∪ { '}' else }
               = { ! * & - ~ ++ -- id con fcon ccon scon ( if while for
                   return break continue '{' ; '}' else }

follow(expr)   = { ; ) = binop relop ] , }

follow(term)   = follow(expr) ∪ { ++ -- [ ( }
               = { ; ) = binop relop ] , ++ -- [ ( }
```

Code Generation

cgen1.c contains a symbolic code generator that must be loaded with your program to obtain output. The code generation interface consists of a set of functions that are called at appropriate points during parsing as follows.

x = progbeg(); progend(x)
progbeg should be called before any parsing is initiated, and *progend* should be called after all parsing is completed, passing it the integer returned by *progbeg*.

deflabel(lab)
should be called at the point of definition of label number *lab*. Label numbers should be generated sequentially by the parser beginning at 1.

global(p)
should be called with the symbol table pointer for each global identifier, except functions, as they are declared.

defconst(p)
After parsing all of the input, *defconst* should be called with the symbol table pointer for each constant referenced in the input.

x = procbeg(p); procend(x)
procbeg should be called when parsing for a procedure body is initiated with the symbol table pointer for the procedure. *procend* should be called after parsing a procedure body, but before the local variables are expunged from the symbol table, passing it the integer returned by *procbeg*.

jump(lab)
should be called to generate a jump to label number *lab*.

r = exprcode(op, r1, r2, lab, p)
should be called for each node in an expression tree in the correct evaluation order. *op* is the operator (see *cc.h*), *r1* and *r2* are the results from calling *exprcode* for the descendents of the node, *lab* is the associated label (used only for conditional operators), and *p* is a pointer to the node. *exprcode* should return a integer describing the result of executing the operator, such as a temporary number. For the operator O_ARG, the argument number should be passed as *r2*, and for the operator O_CALL, *r2* should be the number of arguments. (See *cgen1.c*).

retcode(r)
should be called to generate code for a *return* statement. It is passed the result of *exprcode* for the root of the expression tree for the return expression, or 0 if the return expression is omitted.

Semantics

Your parser must implement the correct semantics for C. The following code templates define the semantics of statements. In the following, L stands for a label number. For loops, L labels the beginning of the loop body, L+1 is the 'continue' label, and L+2 is the 'break label'.

```
if ( expr ) stmt1 else stmt2
                 if ( ~expr ) goto L
                 stmt1
                 goto L+1
        L        stmt2
        L+1

while ( expr ) stmt
        L
        L+1      if ( ~expr ) goto L+2
                 stmt
                 goto L
        L+2

for ( expr1 ; expr2 ; expr3 ) stmt
                 expr1
        L        if ( ~expr2 ) goto L+2
                 stmt
        L+1      expr3
                 goto L
        L+2

break
                 goto L+2(L is current loop label)

continue
                 goto L+1(L is current loop label)
```

Type checking and the order of evaluation are the important semantic issues for expressions. The following tables summarize the type behavior of the operators. In all cases, characters are 'widened' to integers, if necessary. Consult the *C Reference Manual* at the back of *The C Programming Language* for further details.

expression	*equivalent expression*
e1[e2]	*(e1 + e2)
e1 op= e2	e1 = e1 op e2 (e1 evaluated once)
++e	e += 1
--e	e -= 1
e++	(t = e, e += 1, t) (e evaluated once)
e--	(t = e, e -= 1, t) (e evaluated once)
e	e != 0 (e used in conditional)

<table>
<tr><th>expression</th><th>result</th></tr>
<tr><td>!conditional</td><td>conditional</td></tr>
<tr><td>*(ptr T)</td><td>lvalue T</td></tr>
<tr><td>&(lvalue T)</td><td>ptr T</td></tr>
<tr><td>-int</td><td>int</td></tr>
<tr><td>-float</td><td>float</td></tr>
<tr><td>~int</td><td>int</td></tr>
<tr><td>(ptr T) (+ | -) int</td><td>ptr T</td></tr>
<tr><td>T(...)</td><td>T</td></tr>
<tr><td>int (+ | - | * | /) int</td><td>int</td></tr>
<tr><td>(int | float) (+ | - | * | /) (int | float)</td><td>float</td></tr>
<tr><td>int % int</td><td>int</td></tr>
<tr><td>(int | float) (== | != | < = | > = | < | >) (int | float)</td><td>conditional</td></tr>
<tr><td>(ptr T) (== | !=) int</td><td>conditional</td></tr>
<tr><td>(ptr T) (== | != | < = | > = | < | >) (ptr T)</td><td>conditional</td></tr>
<tr><td>int (< < | > > | & | '|' | ^) int</td><td>int</td></tr>
<tr><td>conditional (&& | ||) conditional</td><td>conditional</td></tr>
<tr><td>lvalue T = T</td><td>T</td></tr>
<tr><td>lvalue int = float</td><td>int</td></tr>
<tr><td>lvalue float = int</td><td>float</td></tr>
<tr><td>lvalue (ptr T) = int</td><td>ptr T</td></tr>
</table>

Evaluating expressions in the correct order amounts to making the correct sequence of calls to *exprcode*. For binary operators, including assignment, the left operand must be evaluated first. Evaluating conditional operators (! && || == etc.) is similar except that evaluation causes transfer of control instead of computing a value. In addition, 'short circuit' evaluation is used: the expression is evaluated only as far as is necessary to determine the truth value. Conditional expressions are always evaluated with a 'true' or 'false' label (but not both) to which control should be transferred. Assume *walk(p, tlab, flab)* traverses the tree rooted at *p* with true and false labels *tlab* and *flab*, respectively. The following table shows the correct sequence of recursive traversals depending on *p*'s operator, *tlab*, and *flab*; *left* and *right* denote the left and right operands of *p*, respectively, and L denotes a generated label.

<table>
<tr><th>op, tlab, flab</th><th>sequence</th></tr>
<tr><td>||, lab, 0</td><td>walk(left, lab, 0)
walk(right, lab, 0)</td></tr>
<tr><td>||, 0, lab</td><td>walk(left, L, 0)
walk(right, 0, lab)</td></tr>
<tr><td>L:</td><td></td></tr>
<tr><td>&&, lab, 0</td><td>walk(left, 0, L)
walk(right, lab, 0)</td></tr>
<tr><td>L:</td><td></td></tr>
<tr><td>&&, 0, lab</td><td>walk(left, 0, lab)
walk(right, 0, lab)</td></tr>
<tr><td>< , lab, 0</td><td>r1 = walk(left, 0, 0)
r2 = walk(right, 0, 0)
exprcode(< , r1, r2, lab, p)</td></tr>
<tr><td>< , 0, lab</td><td>r1 = walk(left, 0, 0)
r2 = walk(right, 0, 0)
exprcode(> =, r1, r2, lab, p)</td></tr>
<tr><td>!, tlab, flab</td><td>walk(left, flab, tlab)</td></tr>
<tr><td>binaryop, 0, 0</td><td>r1 = walk(left, 0, 0)
r2 = walk(right, 0, 0)
r = exprcode(binaryop, r1, r2, 0, p)</td></tr>
<tr><td>unaryop, 0, 0</td><td>r1 = walk(left, 0, 0)
r = exprcode(unaryop, r1, 0, 0, p)</td></tr>
</table>

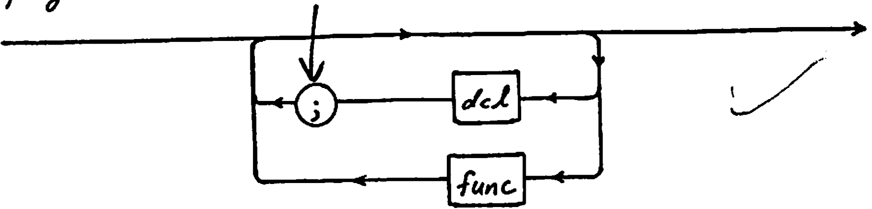
As shown, evaluation of the comparison operators requires both the comparisons and their complements.

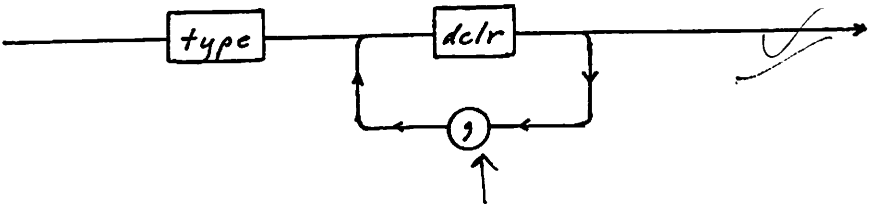
c0.exe[1055,21] is a working version of the parser. In addition, the files *test?.c*[1055,21] contain test programs of increasing complexity; using these files as input, your parser should produce output that is *identical* to that produced by *c0*.

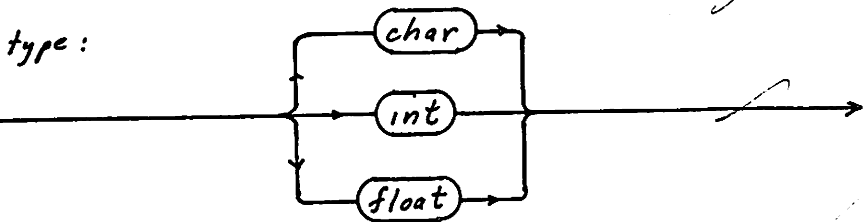
This is a moderately large program; the following table gives the line counts for my version of each of modules in this program.

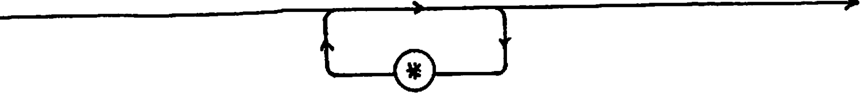
521	*cexpr.c*
137	*cgen1.c*
251	*clex.c*
32	*cmain.c*
360	*cparse.c*
173	*cstat.c*
189	*csym.c*
1663	*total*

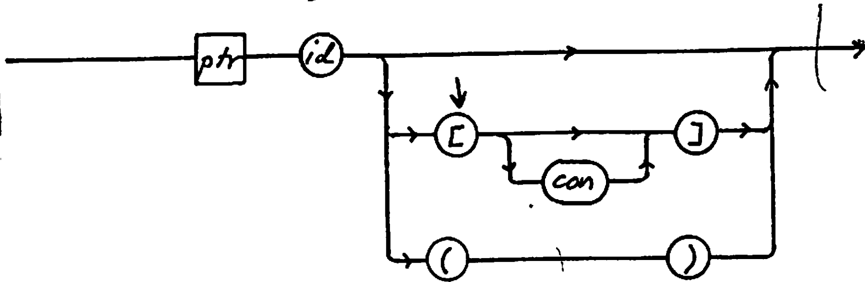
For further information on recursive descent parsing, see Chap. 5 of N. Wirth, *Algorithms + Data = Programs*, Prentice-Hall, 1976. Consult B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978 for details concerning C. Further information (alot of it!) will be given in class; don't miss it.

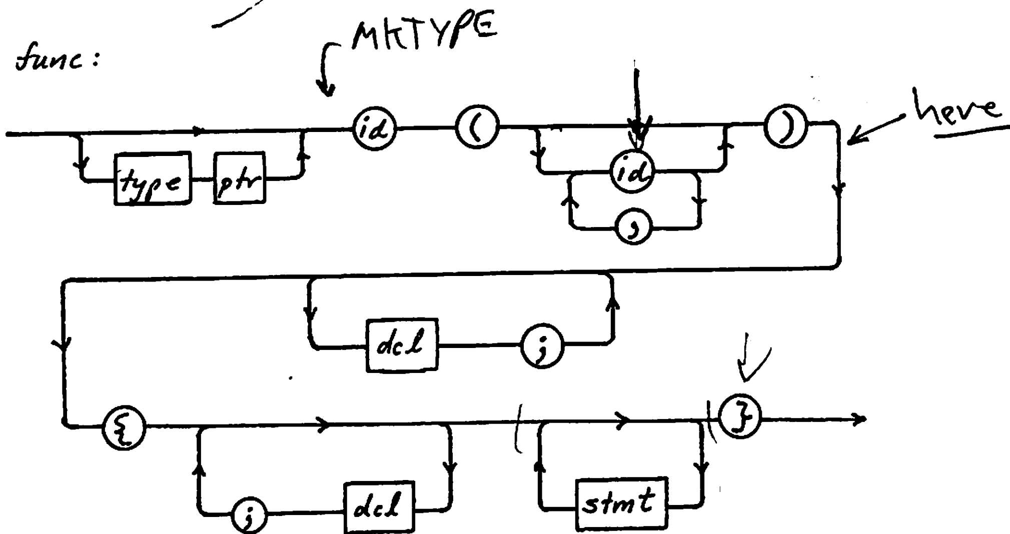

prog :
dcl
;
func


dcl :
type
dclr
,

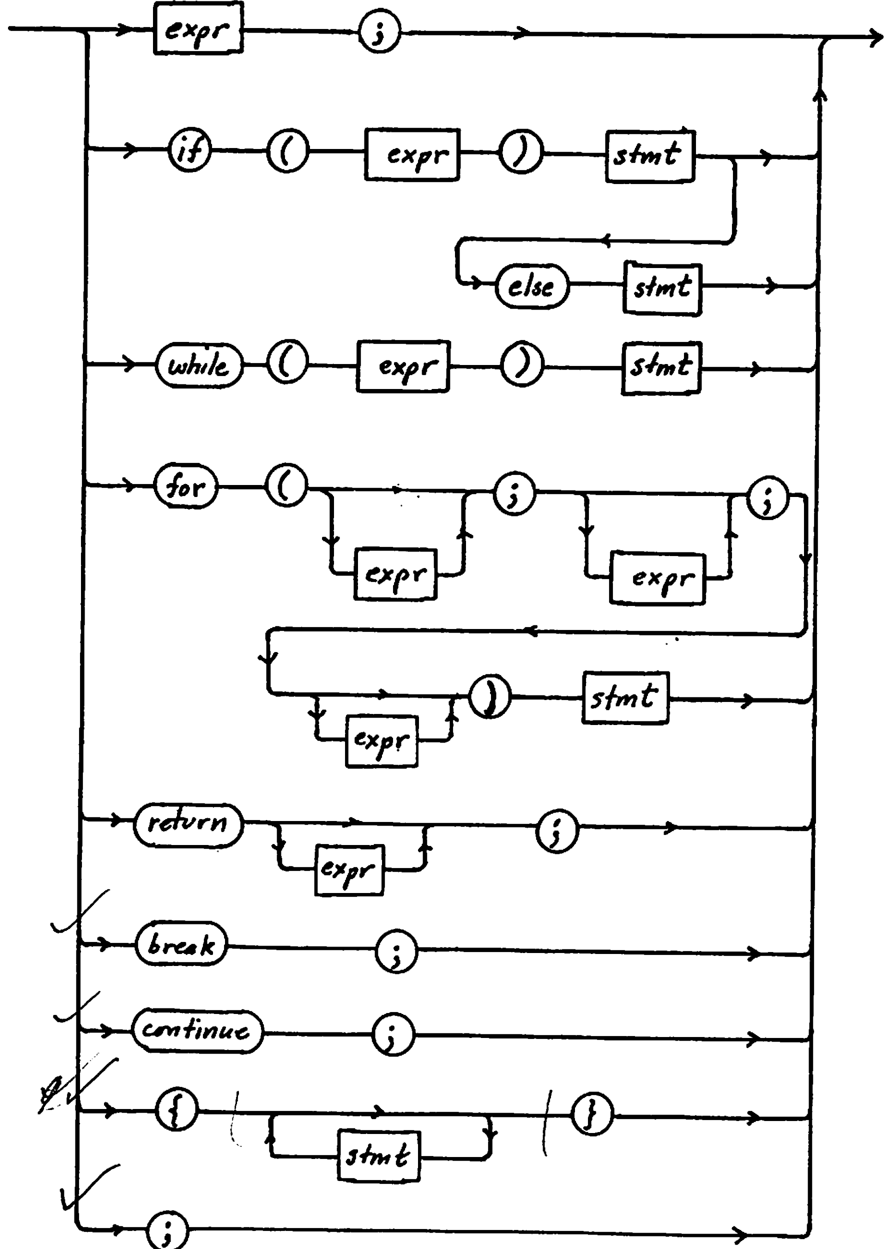

type :
char
int
float


ptr :
*

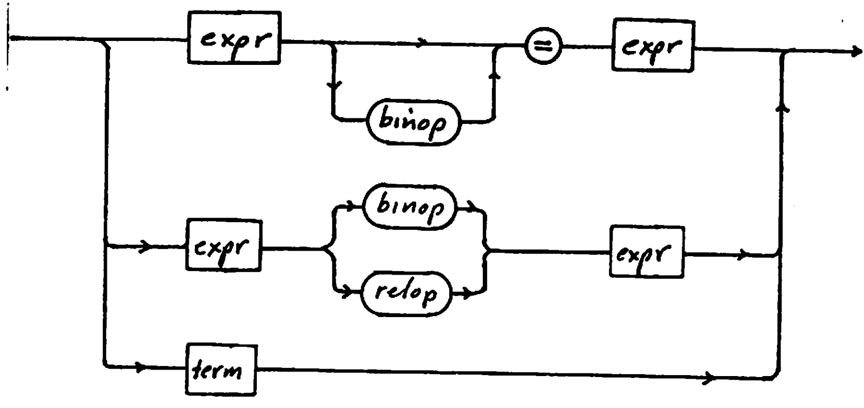

dclr :
ptr
id
[
con
]
(
)

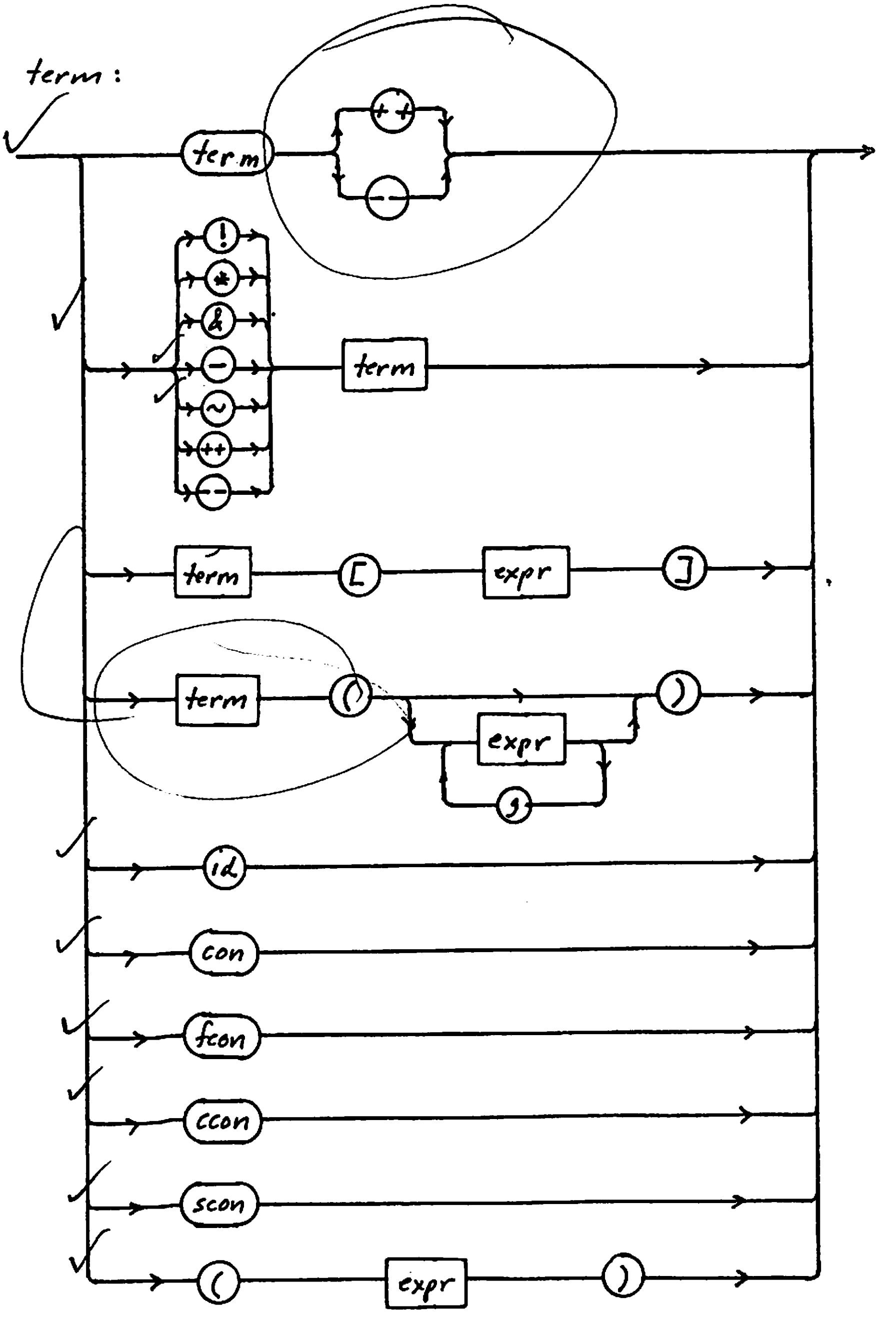

func :
MKTYPE
here
type
ptr
id
(
id
,
)
dcl
;
{
;
dcl
stmt
}

stmt:



expr:




term:
term
++
--
!
*
&
-
~
++
--
term
term
[
expr
]
term
(
expr
,
)
id
con
fcon
ccon
scon
(
expr
)

C Code Generation

The object of this assignment is to replace the routines in *cgen1.c* by routines that generate DEC-10 instructions in **link** object code format. Templates of the expected output for each of the routines and for expressions are given below. As before, the relevant files are as follows.

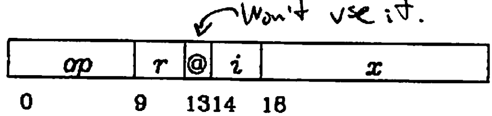
name	*purpose*
* *cc.h*	definitions
* *opcode.h*	instruction definitions
cexpr.c	expression parsing
cgen.c	DEC-10 code generation
* *clex.c*	lexical analysis
* *cmain.c*	main program
cparse.c	declaration parsing
cstat.c	statement parsing
* *csym.c*	symbol table management

Both the source and object files (*.rel* files) are available on [1055,21] for those files indicated by an asterisk. Use your versions of *cparse.c*, *cstat.c*, and *cexpr.c* and place your code generator in *cgen.c*.

The general assembly language form of DEC-10 instructions is

op *r*,@*x*(*i*)

where *op* is a 9-bit instruction code, *r* is a 4-bit register number, @ is a 1-bit indirection indicator, *x* is an 18-bit address expression or small constant, and *i* is a 4-bit index register number. Any of *r*, @, *x*, or *c*, can be omitted. These fields appear in the 36-bit word as follows.



Object code consists of ***commands***, which are introduced by a period in column 1, and ***object text***, which contains symbolic 36-bit polish suffix expressions whose values occupy some memory cell. Object code is defined by the following grammar.

file : { *cmd* | *expr* }

cmd : **.def** [**-s**] *id* *expr*
| **.seg** *id*
| **.len** *id* *con*

expr : *expr* *expr* (+ | - | < | >) | *id* | *con*

Code is emitted into *segments*, which are blocks of object text that are loaded in contiguous blocks of memory. There are three segments: **code**, which contains executable program code, **data**, which contains global data, and **lit**, which contains constants (e.g. program literals). **seg** commands are used to start or to resume emitting code into the named segment. **def** commands are used to define symbols, e.g.

.def rows data 56 +
.def -s $4 code 100 +

define the address of global variable **rows** to be the 57th word in the **data** segment and the address of label **$4** to be the 101st word in the **code** segment. The **-s** indicates that **$4** is a temporary symbol. **len** commands give the length, in words, of each segment.

The DEC-10 has 16 registers, which are to be used as follows.

register	*use*
0	not used
1	function return values
2-5	scratch, used in entry/exit sequence
6-12	scratch, saved across calls
13	parameter pointer
14	frame pointer
15	stack pointer

Expressions should be evaluated using registers 6-12.

Global variables are addressed by simply referring to their names, e.g. loading the address of the global variable **rows** into register 6 is accomplished by **movei 6,rows** in assembly language and by

```
0201300000000 rows +
```

in **link** object code. Constants are addressed by generating a label for them and placing the actual constant in the **lit** segment, e.g. loading the constant **45** into register 6 is accomplished by

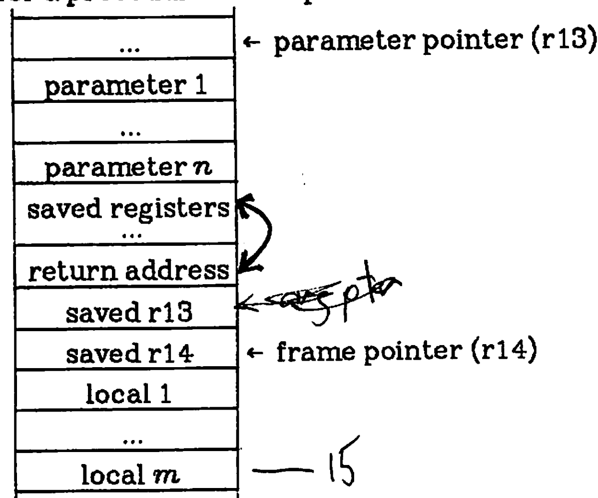
```
        move   6,$15
        ...
$15:    exp    45
```

in assembly language and by

```
0200300000000 $15 +
...
.seg lit
.def -s $15 lit 1 +
45
```

in **link** object code.

Addressing local variables and parameters depends on the run-time representation of procedure activation records. Locals and parameters are located in the current activation record or 'frame' for the procedure in which they are declared. The layout of an activation record for a procedure with n parameters and m locals is as follows.



Parameters and locals are addressed by constant offsets from registers 13 (the 'parameter pointer') and 14 (the 'frame pointer'), respectively. These offsets should be computed by *procbeg* stored in the *s_offset* fields of the symbol table entries for relevant variables. As shown, the offset for the kth parameter is k. For example, if **a** is the third

parameter, loading the address of **a** into register 6 is accomplished by

movei 6,3(13)

in assembly language and by

0201315000003

in **link** object code. Offsets for locals begin at 1 and increase according to the width of each local.

The output for each routine in the bottom half is as follows. For explanatory purposes, Assembly language output is given in some cases for DEC-10 instructions; **link** object text should be emitted for the actual output, however.

routine	*output*
defconst(p)	.seg lit .def $*L* lit *offset* + char, int, float constants: bit pattern char * constants: 0331100000001 lit *offset* + 9-bit bytes, 4/word
deflabel(lab)	.def -s $*lab segment offset* +
exprcode(op,r1,r2,lab,p)	see below
global(p)	.def *p-> s_name* data *offset* +
jump(lab)	jrst $*lab*
procbeg(p)	.seg code 0331100000000 lit *offset* + .seg lit *p-> s_name* given in 9-bit bytes, 4/word .seg code .def *p-> s_name* code *offset* + jsp 4,c.savr adjsp 15,*size of locals* (*procbeg* must also compute *s_offset* values for local variables)
procend(x)	no output
progbeg()	no output
progend(x)	.len code *length* .len data *length* .len lit *length*
retcode(r)	move 1,*r* jsp 4,c.retr
rfree(r)	no output free resource *r*

In generating code for expressions, each operator computes its result and leaves it in a register (6..12) for use as an operand by other operators. In the following table, unless otherwise specified, the operands of each operator are given by *r1* and *r2* and the result is given in *r*, and *r1* and *r2* should be freed. Generated code is given for each operator and combination of operand types. Omitted type information indicates the default code for all types not explicitly mentioned.

operator	*operand type/scope*	*code*
O_ID	global	movei r,*s_name*
O_ID	local	movei r,*s_offset*(14)

O_ID	parameter	movei r,*s_offset*(13)
		if *id* is a char, also do
		hrli r,0001100
		else if *p->e_type* is char *, also do
		hrli r,0331100

O_CON	char	movei r,*constant*
O_CON	int < = 18 bits	movei r,*constant*
O_CON		move r,$*L* (*L* in lit segment)
O_CVI	float	fix r,r1
O_CVF		fltr r,r1
O_NEG		movn r,r1
O_BCOM		setcm r,r1
O_INDIR	lvalue char	ldb r,r1
O_INDIR		move r,(r1)
O_ARG		push 15,r1
O_CALL		movni 5,r2 (r2 = number of args) pushj 15,*p->e_left->e_left->s_name* move r,1
O_ASGN	lvalue char, char	dpb r2,r1
O_ASGN		movem r2,(r1) (result in r2)
O_LT	char *, char *	push 15,r1 push 15,r2 pushj 15,c.scp adjsp 15,-2 jumpl 0,*lab*
O_LT		camge r1,r2 jrst *lab*

O_ADD	char *, int	adjbp r2,r1 (result in r2)
O_ADD	int, int	add r1,r2 (result in r1)
O_ADD	float, float	fadr r1,r2 (result in r1)
O_SUB	char *, int	movn r2,r2 adjbp r2,r1 (result in r2)
O_SUB	int, int	sub r1,r2 (result in r1)
O_SUB	float, float	fsbr r1,r2 (result in r1)
O_MUL	int, int	imul r1,r2 (result in r1)
O_MUL	float, float	fmpr r1,r2 (result in r1)
O_DIV	int, int	idiv r1,r2 (r1+1 must be free) (result in r1)
O_DIV	float, float	fdvr r1,r2 (result in r1)
O_MOD	int, int	idiv r1,r2 (r1+1 must be free) (result in r1+1, free r1)
O_BXOR		xor r1,r2 (result in r1)
O_BOR		ior r1,r2 (result in r1)
O_BAND		and r1,r2 (result in r1)
O_LSH		lsh r1,(r2) (result in r1)
O_RSH		movn r2,r2 lsh r1,(r2) (result in r1)

The remaining comparison instructions are as follows.

operator	*jump?*	*cam?*
O_LT	jumpl	camge
O_LE	jumple	camg
O_EQ	jumpe	camn
O_NE	jumpn	came
O_GE	jumpge	caml
O_GT	jumpg	camle

c1.exe[1055,21] is a working version of the compiler. When run with a *-d* option, *c1*

prints instructions in assembly language format; without the *-d*, **link** object text is printed. In addition, the files *test[2356]*.c[1055,21] are meaningful programs, for which your compiler should be capable of producing executable code. To execute the test programs, the **link** object text must be converted to a DEC-10 relocatable file. This is accomplished by running **link** (which is on the Computer Science library) and **casm**[1055,21] and loading the resulting assembly language file and the skeleton run-time library *rtlib.rel*[1055,21] with **ccom**; for example, to compile and execute *test2.c* use the commands

> *c1 < test2.c | link | casm > test2.mac*
> *ccom test2.mac rtlib.rel[1055,21]*
> *a*

My version of *cgen.c* is ~375 lines long.

163

Csc 453 Translators and System Software, Fall 1983
Programming Assignment

Due Oct 25

Linking

Write the two-pass link editor **link** for the object code described in the paper, 'A Machine Independent Linker'. **link** is invoked by the command

link [*file* | **−o** *ofile* | **−u**]...

Input is taken from *files*, in the order given. Standard input is read in the absence of *files* or if the file '-' is given. Output is written to *ofile*; if *ofile* is omitted, the output is written to the standard output. The **−u** option causes a list of undefined symbols to be printed on standard error.

Object text consists of command lines and polish suffix expressions. The following grammar defines the format of the input files.

```
file   : { cmd | expr }

cmd    : .def [ -s ] id expr
       | .seg id
       | .len id con
       | .org con

expr   : expr expr ( + | - | < | > ) | id | con
```

Identifiers are sequences of letters, digits, percent signs, underscores, or periods, beginning with any of these except a digit. Constants are decimal or octal integers; a leading 0 indicates an octal constant. The semantics of each command and of expressions are described in the paper. Unrecognized commands should be treated as comments and copied to the output.

The version of **link** described in the paper is available in the Computer Science library and is invoked by the command given above. *c1.exe*[1055,21] is a working version of the C compiler that emits **link** object code in 36-bit units. Giving the **−18** option causes *c1* to emit object code in 18-bit units. Compile the programs, *test*?.*c*[1055,21] to produce object code for testing your linker. Additional test data will be provided later.

Use the following files in constructing your program.

name	*purpose*
* *link.h*	definitions
lexpr.c	expression evaluation
lmain.c	main program
lpass1.c	pass one
lpass2.c	pass two
* *lsym.c*	symbol table

Both the source and object files (*.rel* files) are available on [1055,21] for those files indicated by an asterisk.

The approximate lengths (in lines) of the files comprising my version of **link** are

152	*lexpr.c*
76	*lmain.c*
141	*lpass1.c*
114	*lpass2.c*
103	*lsym.c*
586	*total*

Library Searching

Extend **link** to search a library for undefined references. This revised version of **link** is invoked by the command

link [*file* | **-o** *ofile* | **-l** *lfile* | **-u**]...

As before, input is take from *files*, in the order given; standard input is read in the absence of *files* or if the file '-' is given. Output is written to *ofile*; if *ofile* is omitted, the output is written to the standard output.

During pass one, files are read in the order given. The **-l** option indicates that the library *lfile* should be searched in an attempt to resolve any undefined references. The search must be made immediately upon encountering the **-l** option using the set of undefined references at that point.

Libraries are simply archive files (see **ar** and *Software Tools*). Each entry in an archive begins with a header of the form

#-h- name size year date time

The *name* is the file name of the entry and *size* is its size in characters (not including the header). The remaining fields indicate the date and time of insertion into the archive. The first entry in the archive, named 'index', is an index that contains lines of the form

symbol name

where *name* is the name of the entry in which *symbol* is defined. Note that the presence of the index makes it unnecessary to keep the library topologically sorted. Portions of the library may need to be read twice, however. Your program should read the library sequentially constructing an appropriate data structure to represent the library contents. If, after reading the library once, it still contains entries that define undefined but referenced symbols, use random access (see *fseek* and *ftell* in the C Programmer's Manual) to link the necessary library entries. There are several test libraries, named *lib?.lib*, and short test C programs, named *l?.c*, in [1055,21].

You must use your other **link** modules the revised version of *lsym.c*; place the library searching code in *llib.c*. My version of *llib.c* is about 180 lines long.

entry > library entry
> library member

l2.lc + lib1
uses - lxl1.lc + lib3
+ lib3
l1.lc + lib1
+ lib2
l3.lc + lib3
l4.lc + lib4
+ lib3

mylink.h — not modified
lexpr.c
llib.c
lmain.c
lpass1.c
lpass2.c

three weeks

Low-Level Interactive Debugging

Write **db**, a low-level interactive debugger that allows examination, modification and execution of an executing program. The relocatable binary file for **db** is loaded with the program to be debugged using **ccom** and execution begins in **db**. For example, the command

ccom test2.rel rtlib.rel db.rel

loads **db** with the test program and creates the executable file *a.exe*. Execution of *a.exe* begins by calling the function *_db* in *db.rel* instead of *main* in *test2.rel*.

db issues the prompt "*db*>" and accepts the following edit-like commands.

addr	**b**	set a breakpoint at *addr*
addr	**c** *x*	change word at *addr* to *x*
addr	**d**	remove the breakpoint at *addr*
	j	begin or resume execution of user's code
addr	**p** *fmt*	print word at *addr*
addr	**=** *fmt*	print *addr*
addr	**-** *fmt*	print word at --*addr*
addr	*newline*	print word at ++*addr*

The *addr* argument, which is an octal address, is optional; if it is omitted, the most recently specified *addr* is used.

p, **=**, and **-** commands can be followed by a *fprintf* format string that is used in printing the relevant value. This format is remembered and reused in subsequent **p**, **-**, **newline**, and **=** commands until it is replaced. Initialize it to "0%o\n". A simplified version of *fprintf* in *fprintf.c[1055,21]* should be included in your program. In addition to the usual formats, it supports the "%i" format for printing words as instructions, assuming the array *opc* is initialized properly (see *casm.c[1055,21]*). The "newline" command increments *addr* and prints the word at the resulting address; the **-** decrements *addr* and prints the word at the resulting address; thus repeated newline and **-** commands can be used to step through memory. The output of **p**, **-**, and newline commands should be preceeded by the associated address (in octal).

The **c** command changes the word at *addr* to *x*, where *x* is a signed decimal or octal constant. If *x* is omitted, the word is set to 0.

b and **d** commands are used to set and removed breakpoints at *addr*, respectively. **db** should support 5 breakpoints. The **j** command begins execution of the user's program or resumes its execution after stopping at a breakpoint.

Further details concerning the operation of **db** can be obtained by loading the relocatable binary file for my version, which is in *db.rel[1055,21]*, and experimenting with the test programs in *test[2356].rel[1055,21]*.

db.h[1055,21] contains the relevant definitions for **db**. *db.c[1055,21]* contains a skeleton version of **db** that can be used as a starting point; place your debugger in a file named *db.c*. You should "include" *fprintf.c[1055,21]*, which is a version of *fprintf*, and *swtch.c[1055,21]*, which includes the context switching routines, in your version (see *db.c*). Your version of **db** must define the following two routines. *get(adr)* which returns the contents of the word at *adr*, and *put(adr,x)*, which replaces the contents of the word at *adr* with *x* and returns the replaced value. Both of these routines must check the legality of *adr* to avoid addressing errors. In addition, *put* must clear the 'user write protect bit' if *adr* is a legal address in the high segment. This is accomplished by *_setuwp(bit)*, which sets the write bit to *bit* and returns the previous value so that it can be reset after replacing the word at *adr*. *_setuwp* is in *swtch.c[1055,21]*.

j commands are executed by calling *_resume(xp,&dbstate,brks)*, where *xp* is a pointer to the state of the user's program (a *struct state* *, see *db.h*), *dbstate* is the structure that holds the state of **db** during the execution of the user's program, and *brks* is an array of pointers to breakpoint structures (see *db.h*). If *_resume* returns, a breakpoint was encountered. Set the "most recently specified *addr*" to the address of the breakpoint and accept **db** commands.

My version of **db** is 140 lines long.

High-Level Interactive Debugging

Write **cdb**, a high-level interactive debugger that allows examination, modification and execution of an executing C program. The relocatable binary files comprising **cdb** are loaded with the program to be debugged using **ccom** and execution begins in begins by calling the function *_db* in *cdb.rel* instead of *main*.

cdb issues the prompt "*cdb>* " and accepts the following commands.

b		print a list of breakpoints
b	*name n*	set a breakpoint at execution point *n* in *name*
c		print the call to the focus
C		print the call to the focus including locals
d		delete all breakpoints
d	*name n*	delete the breakpoint at execution point *n* in *name*
J		begin or resume execution of user's program
p	*expr*	evaluate C expression *expr* and print the result
s		print the symbols for the focus
S		print the symbols for the focus and all globals
T	*n*	set the user's *_trace* to *n*
–		move the focus "down" one activation
+		move the focus "up" one activation
n		move the focus to the *n*th activation

Unrecognized commands are treated as **p** commands.

Debugging focuses on a "currently active procedure", or *focus*. Initially, there is no focus; after encountering a breakpoint, the function containing the breakpoint, whose activation record is on the top of the stack, becomes the focus. The runtime symbol table is organized so that the symbols for the focus and global symbols are available, in that order.

– and + commands change the focus by moving it to the function associated with the next activation record "down" or "up" the stack, where "up" means towards the top of the stack. Thus, for example, a series of – commands can be used to walk down the stack. A number, *n*, can be given as a command to set the focus to the function associated with the *n*th activation record from the top of the stack. Thus, **0** sets the focus to the function whose activation is at the top of the stack.

The **c** and **C** commands print the call to the focus. For example, if the focus is set to an activation of *queens*, **c** prints

```
queens(c=6)
```

C prints the call and lists the names and values of the locals:

```
queens(c=6)
        r=7
```

Whenever the focus is changed, by a command or by encountering a breakpoint, a **c** command is executed.

The **s** and **S** commands print lists of currently available symbols. The **s** command prints the name of the focus and its local symbols. For example, for the eight queens program, **s** prints

```
int queens()
        int r
        int c
```

S print all of the global symbols, including the locals for the focus:

```
int up[]
int down[]
int rows[]
int x[]
int main()
int queens()
```

```
            int r
            int c
    int print()
```

p commands evaluate C expressions and print the results. Variables appearing in expressions are interpreted according to C scope rules assuming the current function is given by the focus. Variables can be changed by assignment expressions.

b and **d** commands are used to set and remove breakpoints. Breakpoint locations are given as "execution points" within a function. An execution point is the address of the first instruction in the instruction sequence for an expression. Expressions are numbered sequentially beginning at 0 in the order of appearance in the function. The compiler emits a list of execution points immediately following the runtime symbol table entry for functions (see *db.h[1055,21]*). The list is terminated by a 0. **cdb** should support 5 breakpoints. The **j** command begins execution of the user's program or resumes its execution after stopping at a breakpoint.

Further details concerning the operation of **cdb** can be obtained by loading the relocatable binary file for my version, which is in *cdb.rel[1055,21]*, and experimenting with the test programs in *test[2356].rel[1055,21]*.

The relevant files for **cdb** are as follows.

	name	*purpose*
*	*db.h*	**db/cdb** definitions
*	*cc.h*	compiler definitions
	cdb.c	main program
*	*eval.c*	expression evaluation
*	*cexpr.c*	expression parsing
*	*clex.c*	lexical analysis
*	*csym.c*	symbol table management
*	*swtch.c*	context switching primitives

Both the source and object files (*.rel* files) are available on [1055,21] for those files indicated by an asterisk. *cdb.c[1055,21]* contains a skeleton version of **cdb** that can be used as a starting point; place your debugger in a file named *cdb.c*.

The symbol table routine *lookup(str)* searches for the identifier given in *str* by first examining the locals and parameters for the function whose symbol table entry is pointed to by *_focus* following by the globals. Thus, maintaining the focus is accomplished by setting *_focus* correctly. Likewise, *foreach* calls its argument function for each global symbol and each local symbol in the focus. *ptype(p)* prints symbol table entry **p** in the format shown above; it should be used in the **s** and **S** commands.

result *eval(str,frp) (in *eval.c*) parses and evaluates the expression in *str* using the activation record represented by the *frame* structure pointed to by *frp* for accessing locals and parameters. *eval* results a pointer to a *result* structure. *frame* and *result* are defined in *db.h*. *eval* should be used for the **p** command.

findframe(n) (in *cdb.c*) positions a global frame pointer, *cursor*, to the *n*th frame from the top of the stack. It returns 0 if the positioning fails and 1 otherwise. *findframe* should be used for the −, +, and number commands.

As in **db**, **j** commands are executed by calling *_resume(up,&dbstate,brks)*, where *up* is a pointer to the state of the user's program, *dbstate* is the structure that holds the state of **cdb** during the execution of the user's program, and *brks* is an array of pointers to breakpoint structures (see *db.h*). If *_resume* returns, a breakpoint was encountered. Issue a message giving the execution point of the breakpoint, set the focus (*cursor* and *_focus*) to the function associated with the frame at the top of the stack, execute a **c** command, and accept **cdb** commands.

My version of that portion of **cdb** not provided in *cdb.c* is 275 lines long.

Bill Mitchell
Q&C 327

Computer Science 453
Fall 1983

Class Notes, Set 1

August 25, 1983

Software Layers

"applications" software
e.g. software tools, editors,
DBMS, etc.

CS 430

"basic" systems software
e.g. compilers, linkers, editors,
performance monitoring, etc.
i.e. essential tools

CS 453

operating system
system calls: { i/o services, etc.
processes

CS 452

"bare" machine

Essential Systems Software

- in presence of virtual machine presented by the hardware + operating system, i.e. instructions + system calls.

- essential tools are those necessary to construct other tools, i.e. translators + associated systems software.

- Editors for program preparation,

CS 453:

- → Compilers for language translation;
- → Linkers for combining program modules,
- → Debuggers for finding program errors
- → Monitors for understanding program behavior + cost
- Command language interpreters (e.g. a shell) for program invocation.

[Note absence of assembler?]

Editors

- program preparation
- file manipulation/creation
- provide model of general user interface
- examples:
  - line-oriented, e.g. ed (UNIX), sos (Dec10)
  - display-oriented, e.g. Emacs, vi, edt, Rand editor, etc.
  - syntax-directed editors for editing program text, other "objects" Maintain correct form at all times; speed program preparation; bypass some compilation tasks. e.g. Cornell Program Synthesizer, Z, Emacs (sorta), sds.
  - display-editor interface to everything, i.e. editing as the computing metaphor.

Compilers

- for systems programming language, i.e. language for writing operating system and other tools, e.g. BCPL, C, Pascal (sorta), Mesa (Xerox), Bliss, etc.
- must produce reasonable code, and be relatively simple; more importantly, must produce correct code!

For CS 453:

- write a recursive-descent compiler for a subset of C in (full) C, on the Dec-10.
- 1-pass, emitting link object code (will also write the linker!).
- machine-independent parser ("top half"); machine-dependent code generator ("bottom half").
- classical design.

Linkers

- machine-independent <u>link</u> ed<u>itor</u>
- links textual object language
  - machine-dependent <u>content</u>
  - machine-independent <u>form</u>
- permits "iterative linking" i.e. output can be used as input.
- requires simple, machine-dependent loader
- includes library searching (libraries are <u>archive</u> files)
- replaces traditional use of an assembler as compiler's "last pass".

Debuggers

- low-level debugging, i.e. at the instruction level
- use to built high-level debugger, i.e. at the C statement level.
- features
  - editor style interface
  - breakpoints
  - "single stepping"
  - procedure tracing
- requires modification of Compiler.
- programmer-defined debugging functions.

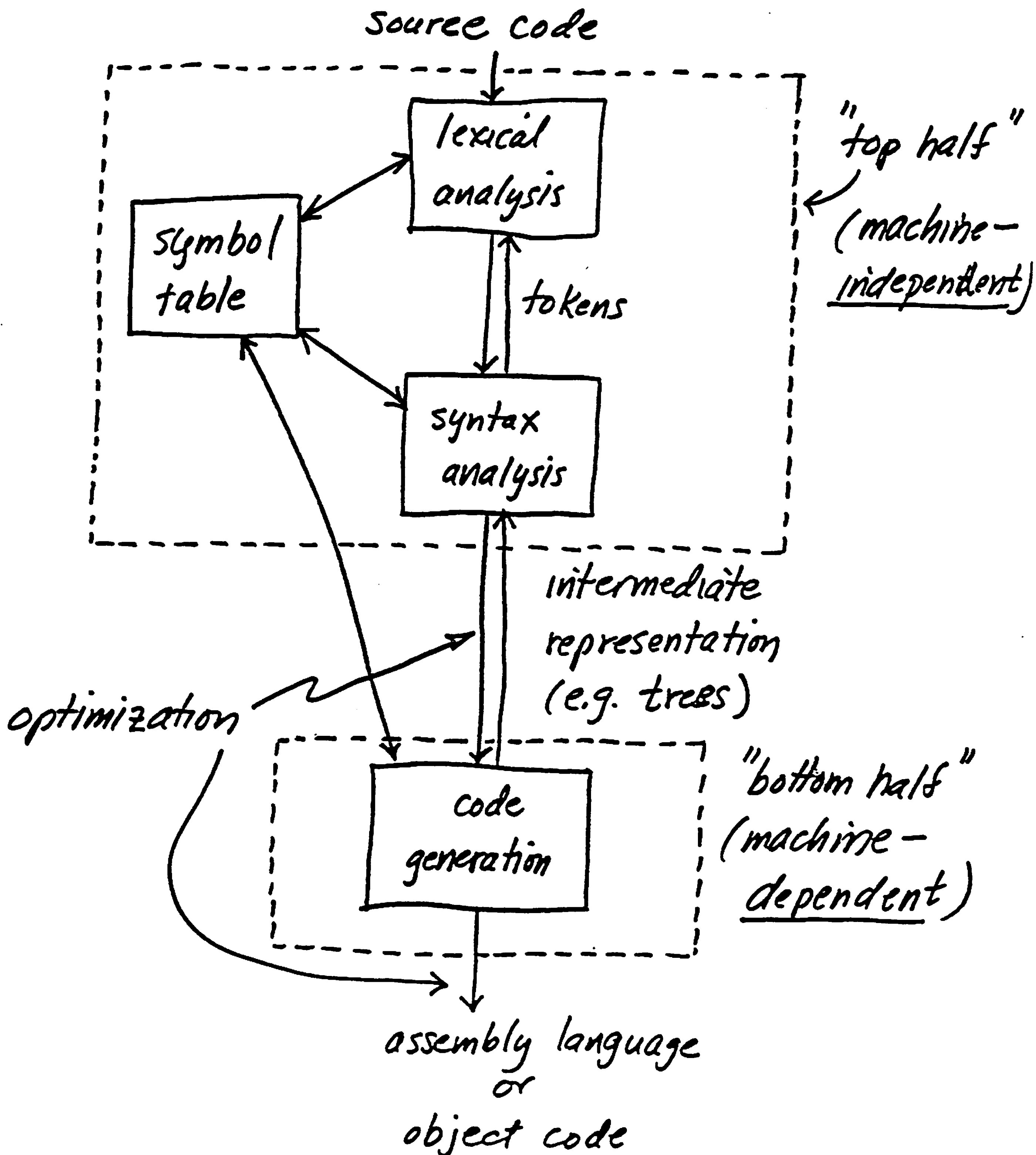
Monitoring

- statement level performance information
- execution frequencies { statements / procedures
- time histogram; sampling techniques.
- data presentation techniques; program "slices"; call graphs.

Command Languages

- UNIX shell; shell programming
- display shells; "point & execute"
- interactiveness vs. "batch"
  (the UNIX shell is batch-oriented!).
- language design / binding issues / data type issues.
- unifying programming & command languages; EZ and related systems.

Compiling: Classical Design



Syntax Analysis

[cparse.c
cexpr.c
cstat.c]

- read tokens
- parse token stream according to a grammar that defines the language.
- perform semantic processing, eg. type checking, execution sequencing
- produce intermediate representation (IR), and interface with code generator, e.g. generate calls to emit jumps and define labels.
- error handing (syntactic & semantics errors).
- machine-independent processing

Lexical Analysis [clex.c]

- read source code,
  return "tokens", e.g.

  - single chars : '(' → '('
  - operators : '>=' ⇒ GE
    (↖ symbolic constant)
  - constants:
    - 65 ⇒ CON, "65"
    - 5.7 ⇒ FCON, "5.7"
    - 'a' ⇒ CCON, "97"
    - "hi there" ⇒ SCON, "hi there"
  - identifiers : putc ⇒ ID, putc
  - keywords : else ⇒ ELSE

- also return "associated" value, e.g.
  symbol table pointer for constant,
  identifier string,
  value of integer constant.

Code Generation [cgen1.c cgen.c]

- read IR, e.g. abstract syntax trees, misc. directives ("generate a jump instruction").
- manage machine resources, e.g. registers
- assign other machine-dependent parameters, e.g. procedure frame layout.
- emit output, e.g. assembly language or object code

Symbol Table [csym.c]

- maintain semantic information on identifiers, constants, e.g. type, address, size, etc.

- maintain multiple instances of same identifier, e.g. global function x, local integer x.

- maintain single-copy of all strings. two or more occurrences of "x" result in entry of only one instance of "x", e.g. like unique software tool.

  - keep all unique strings in the input!
  - permits fast string comparison
  - permits "constant pooling"

[see csym.c, function slookup]

Grammars

grammar: precise specification of syntax

language: set (possibly ∞) of sentences
of terminal symbols (e.g. "a", 56).

$$L = L(T, N, P, S)$$

T = vocabulary of terminal symbols

N = set of non-terminal symbols
("grammatical categories")

P = set of productions
("syntactic rules")

S = start symbol from N.

Example: grammar given in Backus-Naur Form:

```
E : E + E
E : E * E
E : ( E )
E : - E
E : id
```

} productions

$$E \;:\; E + E \;|\; E * E \;|\; (E) \;|\; -E \;|\; id$$

("|" = "or")

$$L = L(\{+ \; * \; (\;) \; - \; id \; EOF\},$$
$$\{E\},$$
$$\{E : E+E,\; E : E*E,\; E : (E),\; E : -E,\; E : id\},$$
$$E).$$

<u>Derivation</u>: repeated replacements (according to productions) to yield sentence from start symbol, e.g.

$$
\begin{aligned}
E &\Rightarrow -E \\
&\Rightarrow -(E) \\
&\Rightarrow -(E + E) \\
&\Rightarrow -(id + E) \\
&\Rightarrow -(id + id).
\end{aligned}
$$

(this is a <u>leftmost</u> derivation)

<u>Sentence recognition</u> ("parsing"): reconstruct the derivation given the sentence.

Top-down Parsing ("Recursive Descent")

- using the next input symbol and current derivation to properly "guess" the next derivation step.

- implement using a recursive procedure for each non-terminal.

Pitfalls

1. left recursion, e.g. $E : E + E$ causes parser to loop.

2. backtracking, e.g. $A : xB \mid xC$ might have to "back-up"; which production to apply?

3. ambiguity : $E : E + E \mid E * E$ "precedence" of + and * ?

4. error handling : what happens when we get "stuck"?

Solutions

3. ambiguity: add more productions to enforce precedence, associativity, etc. e.g.

$$E : E + E \mid E * E \mid (E) \mid id$$

becomes

$$E : E + T \mid T$$
$$T : T * F \mid F$$
$$F : (E) \mid id$$

1. left recursion: replace $A : A\alpha \mid \beta$ ("β followed by 0 or more α's") with $A : \beta A'$ and $A' : \alpha A' \mid \epsilon$ (i.e. <u>right</u> recursion).

$$E : TE'$$
$$E' : +TE' \mid \epsilon$$
$$T : FT'$$
$$T' : *FT' \mid \epsilon$$
$$F : (E) \mid id$$

Solutions, cont'd

2. backtracking: in $A : \alpha \mid \beta$, must be able to determine which alternative, α or β, to choose based on current input symbol.

   Let: first(α) = terminals that begin sentences derived from α.

   Then first(α) $\cap$ first(β) = ϕ

Computing first(X): continue until no change.

a) if X is a terminal, first(X) = {X}

b) if X : aα, where a is a terminal, add a to first(X)

c) if X : ϵ, add ϵ to first(X)

d) if $X : \underbrace{Y_1 Y_2 \cdots Y_{i-1}}_{\stackrel{*}{\Rightarrow} \epsilon} Y_i \cdots Y_k$, add

   first(Y_1) $\cup$ first(Y_2) $\cdots$ $- \{\epsilon\}$ to first(X)
   and add first(Y_i) to first(X).
   i.e. can "look thru" via ϵ-productions $Y_1 \cdots Y_{i-1}$.

<u>Let</u> follow(A) = terminals that can <u>follow</u> non-terminal A.

computing follow : repeat until no change

a) if S is the start symbol, add EOF to follow(S).

b) if $A : \alpha B \beta$ and $\beta \neq \epsilon$, add $first(\beta) - \{\epsilon\}$ to follow(B).

c) if $A : \alpha B$ or $A : \alpha B \beta$ and $\epsilon \in first(\beta)$, add follow(A) to follow(B).

<u>Example</u>:

rule	first	follow
E : TE'	{ (id }	{) EOF }
E' : +TE' | ε	{ + ε }	{) EOF }
T : FT'	{ (id }	{ +) EOF }
T' : * FT' | ε	{ * ε }	{ +) EOF }
F : (E) | id	{ (id }	{ * +) EOF }

(see Asgn #1 handout)

Extended BNF

- recursion and ϵ-productions are troublesome
- extend BNF to include explicit repetition and optional constructs
- use syntax diagrams.

Extensions

[a]	stands for	$a \mid \epsilon$
{ a }	" "	$\epsilon \mid a \mid aa \mid \cdots$
(a | b) c	" "	$ac \mid ab$

Example

$E' : \epsilon \mid +T \mid +T +T \mid \cdots \Rightarrow \{+T\}$

$E : T \; \{+T\}$

$T : F \; \{*F\}$

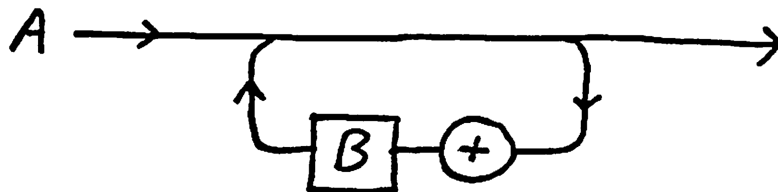
$F : (E) \mid id$

first & follow Sets

- first – computed by initial branches in diagram

  – includes ϵ if diagram can be traversed without encountering □ or ○.

- follow – computed from examining where □ is used.



$first(A) = \{ + \ \epsilon \}$

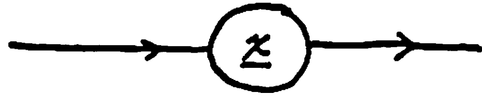
$follow(B)$ includes $follow(A) \cup \{+\}$

<u>Want</u> :

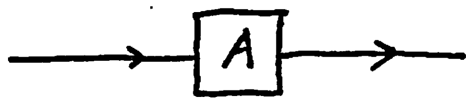
1. every fork to be selectable by looking only at next i.e. $first(\alpha) \cap first(\beta) = \emptyset$
2. if any graph can be traversed without reading input, label its exit path with follow set (which affects #1).

Translating EBNF → Diagrams

1. terminal $\underline{x}$



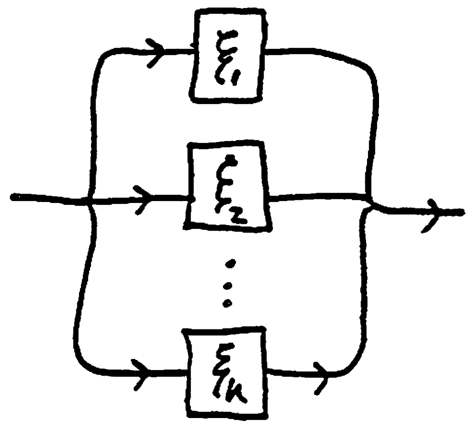
2. non-terminal A



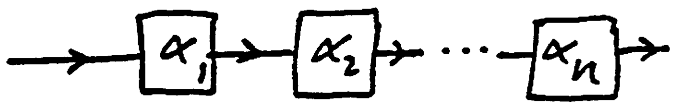
3. $A : \xi_1 \mid \xi_2 \mid \cdots \mid \xi_n$

   where $\boxed{\xi_k}$ is

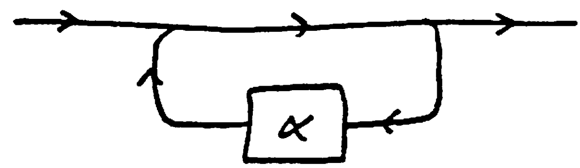
   syntax diagram for ξ_k



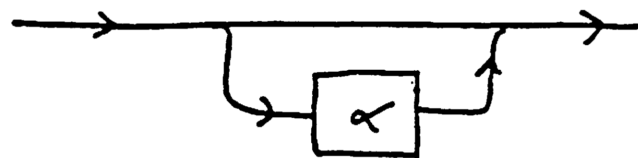
4. $A : \alpha_1 \alpha_2 \cdots \alpha_n$



5. $\{\alpha\}$



6. $[\alpha]$



Translating Diagrams → C Code

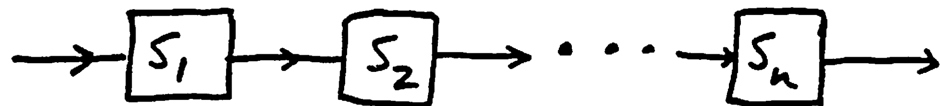
input tokens: `t = gtok();`

↑ "current token"

1. reduce graphs to few manageable graphs via substitution.

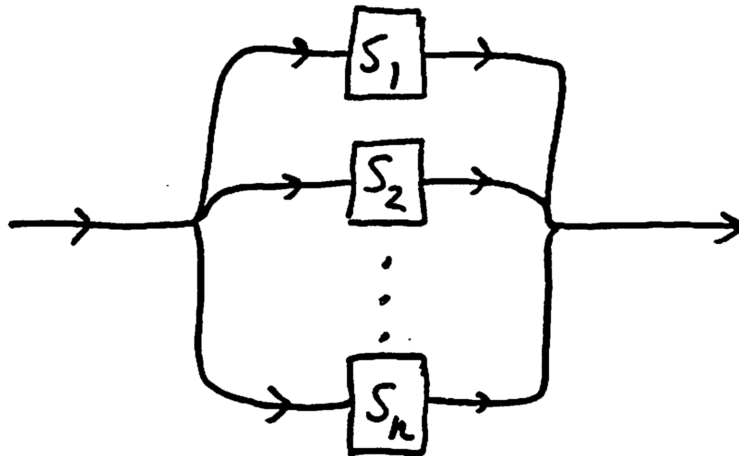
2. for each graph S, write a procedure T(S) that "traverses" the graph guided by the input.

Use following templates:



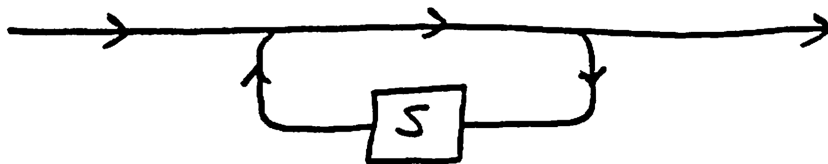

```
{ T(S1);
  T(S2);
   :
  T(Sn); }
```

i.e. non-terminals ⇒ procedure calls




```
if ( t ∈ first(S1) ) T(S1);
else if ( t ∈ first(S2) ) T(S2);
    ⋮
else if ( t ∈ first(Sn) ) T(Sn);
else err( "syntax error");
```

(or use equivalent switch statement).

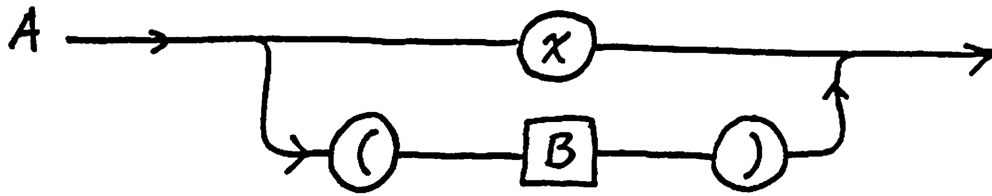


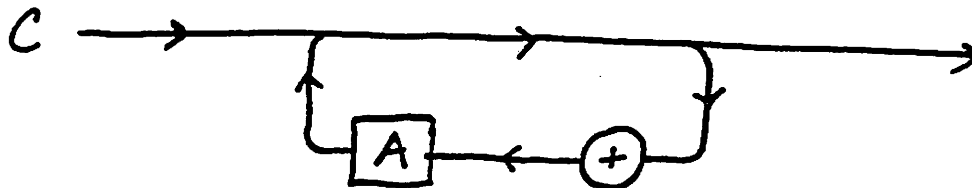

```
while ( t ∈ first(S))
        T(S);
```

Example :

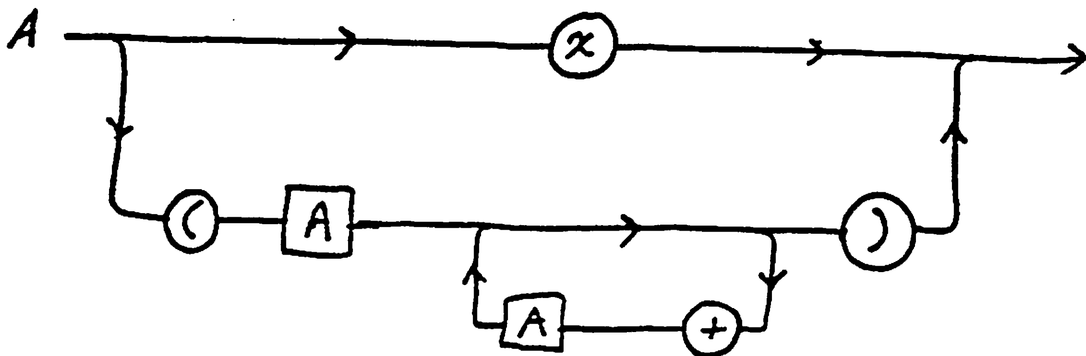
A : x | '(' B ')'
B : A C
C : { + A }

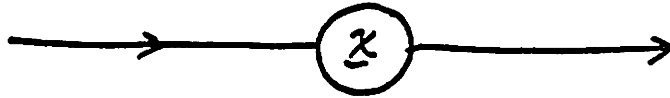
sentences are x, (x), (x+x), ((x)), ...





Could draw one diagram for A :






```
if ( t == x )
        t = gtok();
else  err( "missing x", "");
```

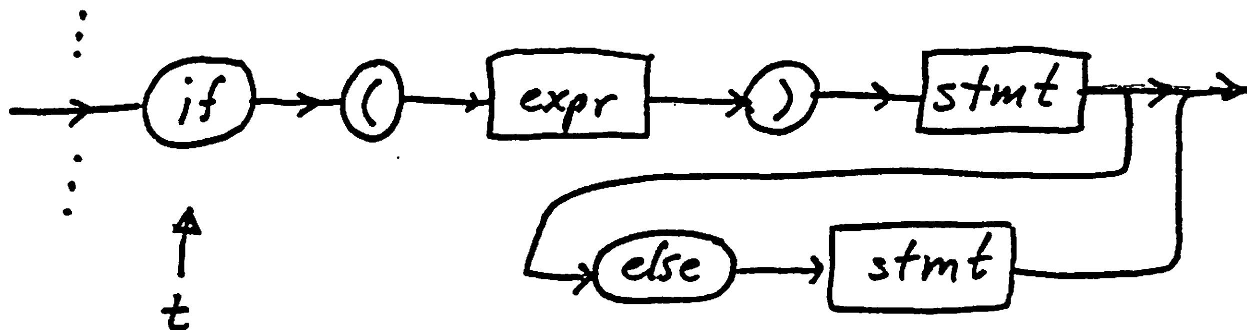
Example : syntax diagram for A

```
int t;
main()
{
    t = gtok();
    A();
    if (t != EOF)
        err("syntax error");
}
```

(after some refinement)

```
A()
{
        if(t == 'x')
                t = gtok();
        else if(t == '(') {
                t = gtok();
                A();
                while (t == '+') {
                        t = gtok();
                        A();
                        }
                if (t == ')')
                        t = gtok();
                else  err("missing )");
                }
        else  err("syntax error");
}
```

Example: C if statement




```
stmt()
{
    switch (t) {
        :
        case IF:
            ifstat();
            break;
        :
        default: err("unrecognized statement")
    }
}
```

```
ifstat()          /* without semantics */
{                      (5 more lines!)
        t = gtok();
        mustbe('(');
        expr();
        mustbe(')');
        stmt();
        if (t == ELSE) {
                t = gtok();
                stmt();
                }
}

mustbe(c)
{
        static char missing[] = "missing X";
        if (t == c)
            t = gtok();
        else {
            missing[8] = c;
            err(missing, "");
           }
}
```

Compiler Organization

(see grammar)

cparse.c { prog, dcl, dclr, type, ptr, func

cstat.c { stmt

cexpr.c { expr, term

Advice

1. write a procedure (or several) for <u>each</u> of the above non-terminals.

   Don't take short cuts!

2. resolve ambiguities locally; i.e. ignore them, then revise code to fix them.

Compiling Declarations

- syntactic specification for global declarations & functions is ambiguous, e.g.

int f ( ) ;	global ("forward") declaration
int f ( ) { ...	function declaration
int f (a ...	function declaration

jump to start of param list parsing

in all cases, f is a function; parse as usual until after (, make symbol table entry, then make global decl. / function decl decision.

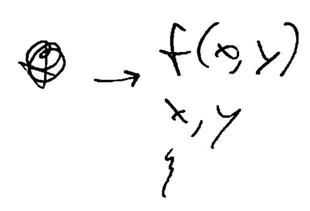
- grammar accepts invalid array decls; use semantic checks:
  - globals : array size must be given;
  - parameters: array size should be ignored;
  - locals : array size must be given.

- grammar accepts invalid function decls; use semantic checks
  - globals: install as usual, but don't turn on DEF (establishes type for forward references).
  - parameters: illegal.
  - locals: install as global, as above.

Strategy for writing dclr & dcl:

```
dclr(type, f)
int type;     /* type of this list */
struct symbol *(*f)();
{
    ... gather up one declaration
        item ... call appropriate routine
  (*f)(name, type, size, ...)
         ↑
       symbol
}
```

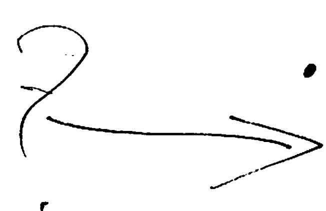
f is a function; one for each of global, parameter, local, which installs symbol & performs semantic checks.

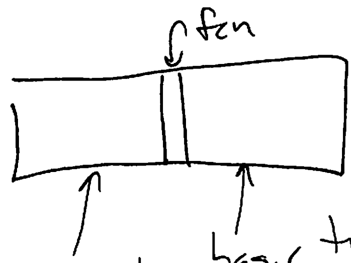



```
dcl(type, f)
int type;
struct symbol *(*f)();
{
        while (t == ID || t == '*') {
                dclr(type, f);
                if(t == ';')
                        t = gtok();
                }
}
.
.
.
/* parse locals */
while (t == CHAR || t == INT || t == FLOAT) {
     dcl(type(), decllocal);
     mustbe(';');
     }
```

installs local identifiers

<u>Note</u>:
- must always check for erroneous redeclaration, or for missing or extraneous declarations for parameters.
- need to call global(p) for global variables (will allocate storage).





Symbol table entries

Be sure to use the macros

```
struct symbol {
    char  *s_name;           /* name */
    int   s_flags;           /* scope & other flags */
    int   s_type;
    int   s_size;            /* size in bits */
    int   s_offset;  for use by bottom half, arg numbers, etc.
    struct symbol *s_link;
};
```

flags: DEF REF
GLOBAL · LOCAL PARAM CONST
KEYWORD

Examples

input	s_type	s_size
int a;	int	36
char s[8];	char *	72
float *y[10];	float **	360

Compiling Procedures

- straightforward traversal of syntax diagrams (once ambiguity is resolved).

```
func()
{
    install function name as global;
    parse parameter list, setting
       s_offset of each parameter
       to parameter number;
    parse parameter declarations;
    check for semantic errors in params.;
    mustbe('{');
    parse locals;
    x = procbeg(symbol table entry
                  for function name);
    stmtlist();
    retcode(0);   /* gratis return */
    mustbe('}');
    procend(x);   } order is important!
    expunge();
}
```

Compiling Statements

- make calls to deflabel, jump, and expression code generator at appropriate points in syntax diagram.

- use three contiguous labels for loops, e.g. L, L+1 ("continue" label), L+2 ("break" label).

- suggested organization:
  - stmtlist
  - stmt
    - ifstat
    - whilestat
    - forstat
    - (others-inline code)

- write a label generator; e.g. genlabel(n) generates n labels, returns first one.
  Uses integers for labels first call - ret 1

- generate code for expressions via walk(p, tlab, flab); where
  - p - pointer to root of expression tree
  - tlab - "true" label
  - flab - "false" label

```
stmt()
{   switch (t) {
        case first(expr):    ↙ Version of expr that eats ident + rets id node
            walk(expr(), 0, 0);
            mustbe (';');
            break;
        case IF:
            ifstat(genlabel(2));
            break;
        case WHILE:
            whilestat(genlabel(3));
            break;
        •
        •
        •
        default:
            err("unrecognized stmt", "");
    }
    if(t ∉ follow(stmt)) {
        err("illegal stmt termination", "");
        skip to t ∈ follow(stmt);
    }
}
```

if (expr) $stmt_1$ else $stmt_2$

```
       if (~ expr) go to L      <- "false" label
       stmt1
       go to L+1   ]
  L:   stmt2       }  omit if no else clause
  L+1:
```

```
ifstat (lab)
int lab;
{
        t = gtok();
        mustbe ('(');
        walk(expr(), 0, lab);
        mustbe (')');
        stmt();
        if(t == ELSE) {
                t = gtok();
                jump (lab + 1);
                deflabel (lab);
                stmt();
                }
        else deflabel(lab);
        deflabel(lab+1);
}
```

Loops

while (expr) stmt

```
L
L+1     if (~expr) goto L+2
        stmt
        goto L
L+2
```

L+2 is always break label, if chk is ; no expr, etc.

for ($expr_1$; $expr_2$; $expr_3$) stmt

($expr_1$, $expr_2$, $expr_3$) optional!

```
        expr1
L       if (~expr2) goto L+2
        stmt
L+1     expr3
        goto L
L+2
```

---

break

goto L+2

continue

goto L+1

L is current loop "handle"
error is there is no current loop!

Example

```
for(i=1; i<10; i++) {
        sum += x[i];
        if(sum > 100)
            break;
        }
```

```
        i = 1
L1:     if( i >= 10) go to L3
        sum += x[i]

        if (sum <= 100) go to L4
        go to L3
L4: L5: L2:
        i ++
        go to L1
L3:
```

L1, L2, L3 - for loop labels

L4, L5 - if labels

Error Handling

- lexical errors (" ....)
- syntactic errors
  - extra symbols
  - missing symbols
  - use of incorrect symbol
  - transposition
- semantic errors
  - undeclared identifier — uses "?" and int type + uses it on demand
  - break/continue outside a lopp
  - duplicate declarations
  - missing declarations (e.g. parameter)
  - illegal array size
  - type conflict

- strategy: continue parsing, skipping input if necessary; <u>must</u> make forward progress.

- maintain correct parsing state according to first/follow sets.

- do something "reasonable" for semantic errors, e.g. enter undeclared identifiers, assume <u>int</u> for type errors.
- avoid "cascading"

Errors

- missing symbols, e.g. ')' – assume it was there.

```
mustbe(c)
int c;
{
        if (t == c)
            t = gtok();
        else {
            err("missing c", "");     <-- do not advance input
        }
}
```

- $t \notin first(X)$ – skip input until $t \in first(X)$ or $t \in follow(X)$ or $t == EOF$.

  alternative: skip one input symbol (useful in expression parsing).

- $t \notin follow(X)$ – skip input until $t \in follow(X)$ or $t == EOF$

- use more restricted sets in some cases, e.g. stmts.

Semantic Errors

- maintain correct expression trees, i.e.
  - operators must have proper number of arguments
  - assume type information in presence of type errors, e.g. T_INT.
- install undeclared identifiers as globals with type int.
- ignore duplicate declarations (but issue msg!)
- assume 1 for illegal array sizes
- ignore extraneous break/continue's.
- bottom line: make sure code generator gets "reasonable" parameters, even if code is incorrect.

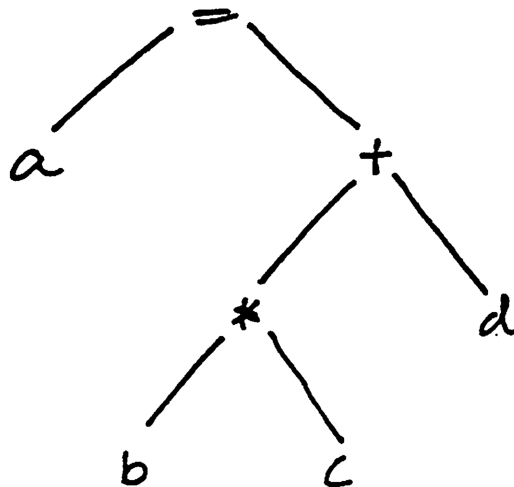
#-h- name ○

Compiling Expressions

- parse expression syntax.
- construct trees to represent expressions.
- perform semantic checks (i.e. type checking) as trees are built.
- traverse trees calling exprcode to generate code, in correct order. (i.e. postorder traversal).

Example: int a, b, c, d;

a = b*c + d;



a b c * d + =

Expression Nodes

```
struct node {
        int e_op;        /* operator O_xxx */
        int e_type;      /* type of result */
        struct node *e_left;
        struct node *e_right;   /* descendants */
        int e_result;
}       int e_count;     /* ref count */
```

Operators:

- O_ASGN
  O_NEQ
  ⋮
  O_CON
  } operators expected by exprcode

- may use other operator codes to guide traversal, e.g. ||

  if (a < 10 || b == 6) . . .

  ↑ never passed to exprcode, but exists in tree.

addr val

Lvalues / Rvalues

- Rvalue : value associated with a name.
- Lvalue : location associated with a name.

a = b

need a's lvalue (arrow to a); need b's rvalue (arrow to b)

- must distinguish between pointers (which are rvalues) and lvalues, e.g.

```
int a, *p;

*p = a;
```

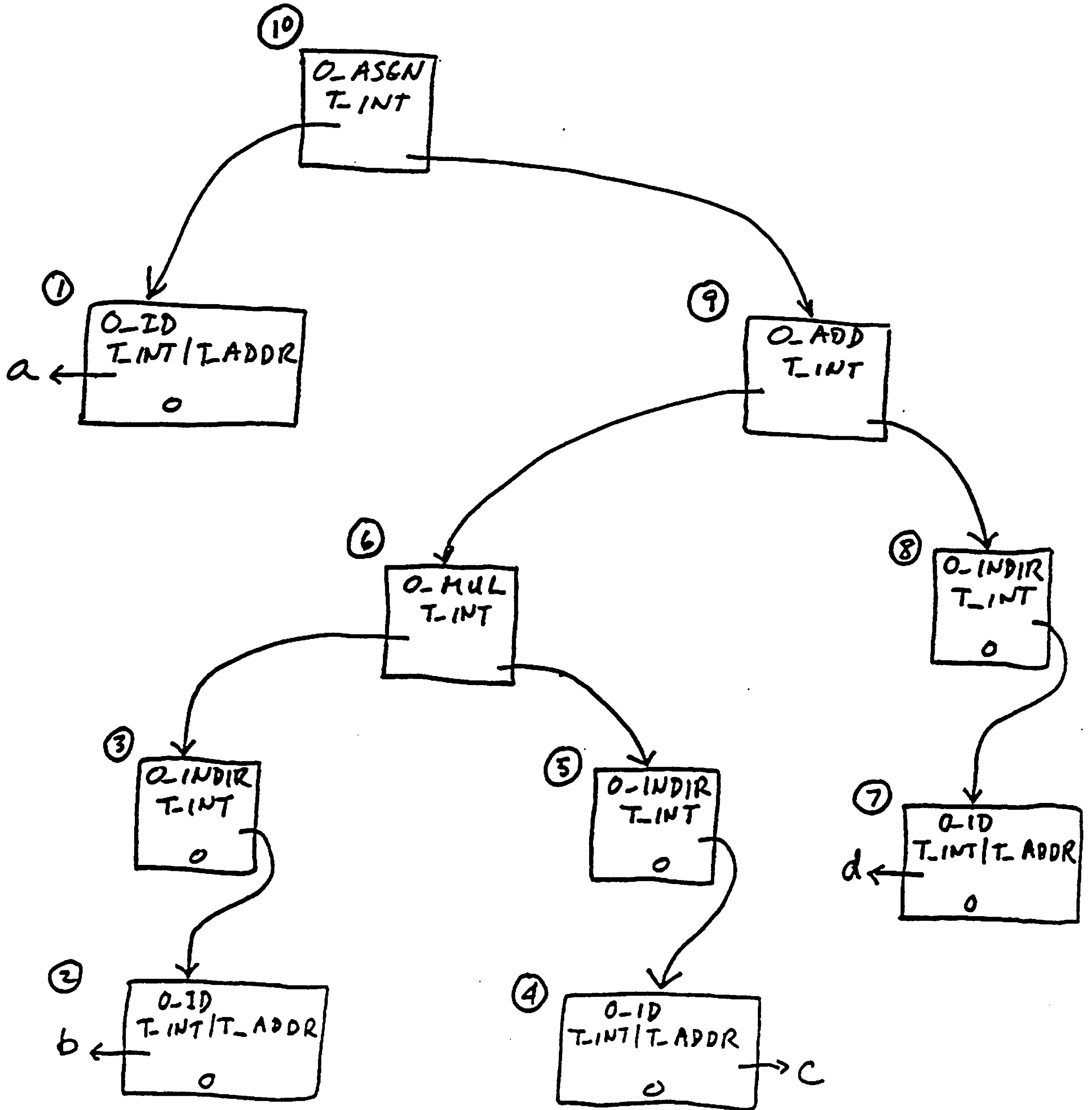
takes rvalue & rets it as lvalue

need rvalue (arrow to p)

returns lvalue : the location referred to by the value of p. (arrow to *)

- use T_ADDR type flag to denote lvalue.
- use O_INDIR (unary *) to "convert" lvalue → rvalue.

$a = b * c + d$



- (i) denotes order of calls to exprcode.

```
a = b*c + d

r1 = exprcode(O_ID, 0, 0, (1));
        T1 = &a          (r1 is assigned 1)

r2 = exprcode(O_ID, 0, 0, (2));
        T2 = &b

r3 = exprcode(O_INDIR, r2, 0, (3));
        T3 = *T2

r4 = exprcode(O_ID, 0, 0, (4));
        T4 = &c
r5 = exprcode(O_INDIR, r4, 0, (5));
        T5 = *T4
r6 = exprcode(O_MUL, r3, r5, (6));
        T6 = T3 * T5
r7 = exprcode(O_ID, 0, 0, (7));
        T7 = &d
r8 = exprcode(O_INDIR, r7, 0, (8));
        T8 = *T7
r9 = exprcode(O_ADD, r6, r8, (9));
        T9 = T6 + T8
r10 = exprcode(O_ASGN, r1, r9, (10));
        *T1 = T9
```

Parsing Expressions

- consider subset of C expressions:

```
expr  : expr '=' expr
      | expr ('+' | '-' | '*' | '/') expr
      | term

term  : '++' term
      | term '++'
      | id
```

++ ++ id ++ ++

t -> t ++
t -> t ++ t ++
id id ++ ++

- grammar is a problem:
  - left recursion
  - ambiguous (no precedence/assoc.)
- revised grammar:

gives right assoc. (points to: { = expr })

```
expr  : expr1 { = expr }

expr1 : expr2 { ( + | - ) expr2 }

expr2 : expr3 { ( * | / ) expr3 }

expr3 : term

term  : ( ++ term | id ) { ++ }
```

```
struct node *expr()
{
        struct node *p;
        p = expr1();
        while ( t == '=' ) {
                t = gtok();
                p = enode (O_ASGN, p, expr());
                }
        return (p);
}
```

check follow set → return (p);

performs type-checking, allocates nodes → enode

"No point in checking follow set in other routines."

```
struct node *expr1()
{
        int op;
        struct node *p;
        p = expr2();
        while ( t == '+' || t == '-' ) {
                op = t == '+' ? O_ADD : O_SUB;
                t = gtok();
                p = enode ( op, p, expr2());
                }
        return (p);
}
```

ife(c, i, e)
((c) ? i : e)

y*z+p;

```
struct node *expr2()
{
        int op;
        struct node *p;
        p = expr3();
        while (t == '*' || t == '/') {
                op = t == '*' ? O_MUL : O_DIV;
                t = gtok();
                p = enode(op, p, expr3());
                }
        return (p);
}

struct node *expr3()
{
        return (term());
}
```

```
struct node *term()
{
        struct node *p;
        if (t == INCR) {
                t = gtok();
                p = enode (—, term(), 0);
                }
        else if(t == ID) {
                p = enode (O-ID, lookup( lexval.l-str),0);
                t = gtok();          dcldident
                }
        else err( — );
        while ( t == INCR ) {
                p = enode( —, p, 0);
                t = gtok();
                }
        return (p);
}
```

- idea: parameterize expr1, expr2 sequence to handle arbitrary number of precedence levels.

```
int optab[][3] = {      /* tokens for each precedence level */
        { '+', '-', 0 },
        { '*', '/', 0 },
        0
};

struct node *expr()
{
        struct node *p;

        p = expr1(0);
        while (t == '=') {
                t = gtok();
                p = enode(O_ASGN, p, expr());
        }
        return (p)
}
```

tokens for each precedence level

asgn doesn't fit model

0 — & run off end, call term

Dealing w/ lower bin ops.

|| binop)

"This is about the only neat part of the program."

Row of optab

```
struct node *expr1(n)
int n;
{
        int op;
        struct node *p;

        if (*optab[n] == 0)
                return (term());
        p = expr1(n+1);
        while (t ⊂ optab[n]) {
                op = "correct operator"
                t = gtok();
                p = enode(op, p, expr1(n+1));
        }
        return (p);
}
```

if at lowest level, call term

expr1 calls expr2

while(optab(inset2()) != '=')

don't have arg. assignment

expr1 calls expr2

csym - alloc

p = (struct node *) alloc(sizeof (struct node));

```
        ...
        while (op = inset(t, optab[n])) {
                t = gtok();
                p = enode(op, p, expr1(n+1));
                }
        ...
```

Bill Mitchell

Computer Science 453
Fall 1983

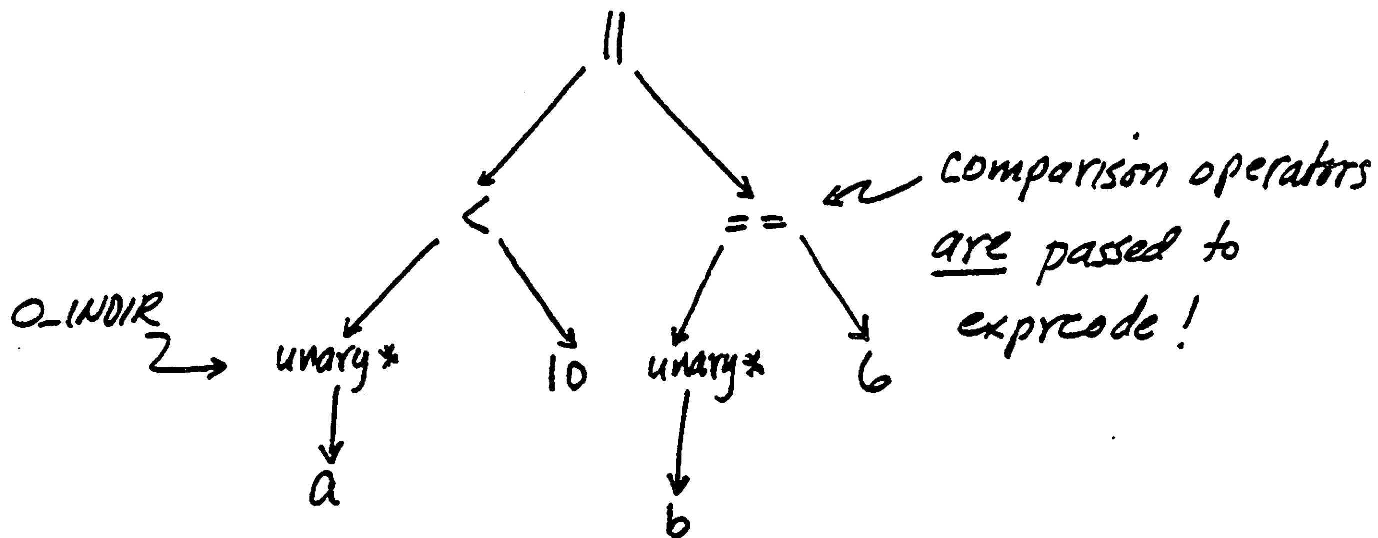
Class Notes, Set 2

August 29, 1983

Conditional Expressions

- unary !, ||, && alter flow of control; do not produce values.
- included in trees to guide traversal; not passed to exprcode.

Example: a < 10 || b == 6

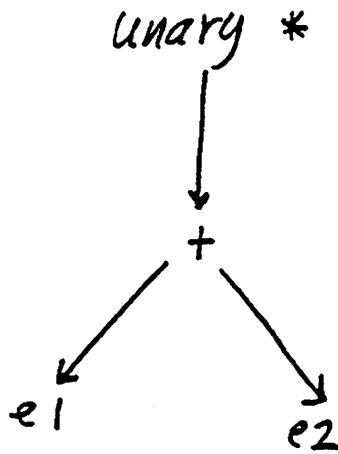


- general technique: use additional operator as necessary to guide traversal or to simplify code generation, e.g. x[e]

e1[e2]

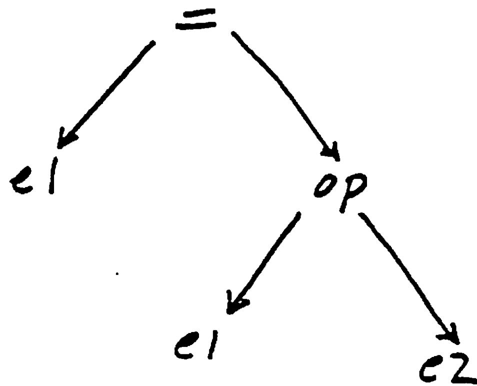
⋮
p = enode('[', p, expr());

enode returns *(e1+e2) :



---

e1 op= e2

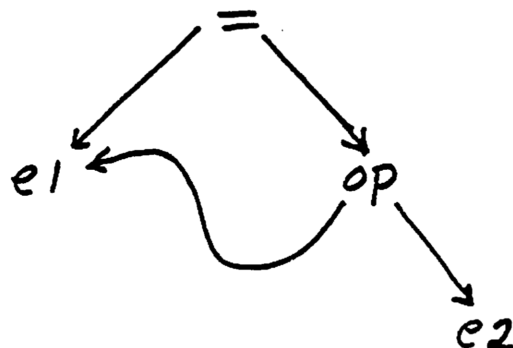


Examples: a += 5;

*f() += a; ← oops! f() is called twice!

∴ must evaluate e1 only once.

e1 op= e2



- need to "remember" result of exprcode for 1st evaluation of e1, and re-use it for 2nd evaluation.

- store "result" for each node in e_result field

res #

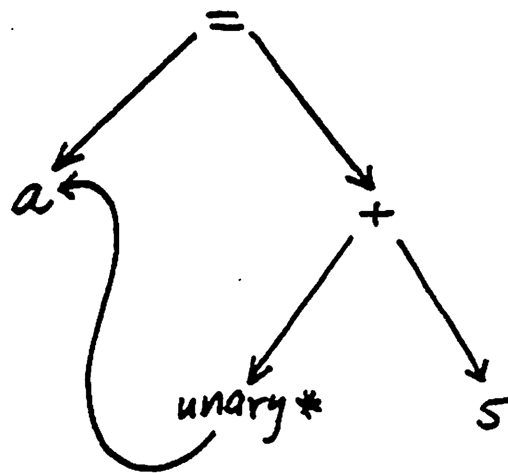
- walk(...) returns e_result field <u>instead</u> of traversing the subtree

- technique simplifies type-checking, e.g. can compile e1 += e2 as e1 = e1 + e2; special type-checking for += is unnecessary.

e1 op= e2

- left operand of op must be an rvalue; e1 must be an lvalue.

Example: a += 5;

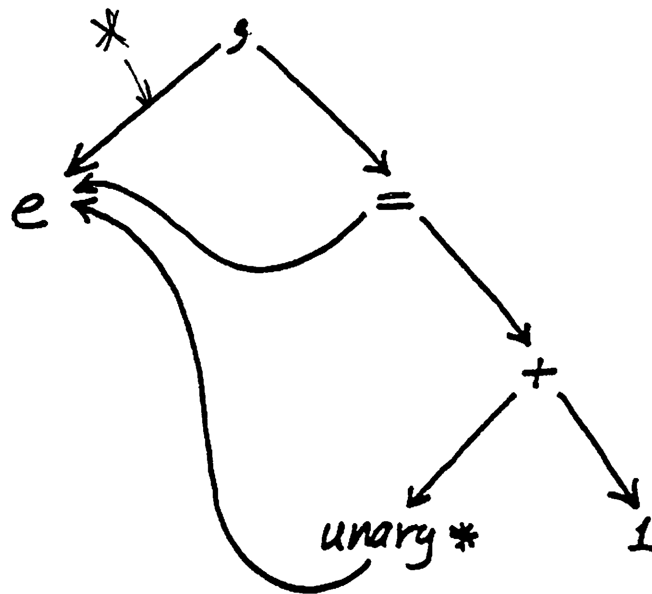


- ++e ⟺ e += 1
- --e ⟺ e -= 1
- e++ ⟺ (t' = e, e += 1, t')
- e-- ⟺ (t' = e, e -= 1, t')

(e++ and e--:) need to fetch value, perform assignment, return value.

e++

- introduce new operator that evaluates both operands, but returns result of left operand :



- e-- is similar.

Tree Traversal

r = walk(p, tlab, flab)

- traverses tree rooted at p, calling exprcode as necessary.
- tlab ("true" label), flab ("false" label) are used for conditional expressions.
- returns result of calling exprcode.

```
int walk(p, tlab, flab)
struct node *p;
int tlab, flab;
{
        int r, r1, r2, op;

        if (p == NULL)
                return (0);
        if (r = p->e_result)      result already cacl[c]ulated?
                return (r);
        r = r1 = r2 = 0;
        switch (op = p->e_op) {
                :
                case O_ADD:
                        :  (set r)
                        break;
                default:
                        fprintf(stderr, "compiler error");
                        exit(1);
                }

        p->e_result = r;
        return (r);
}
```

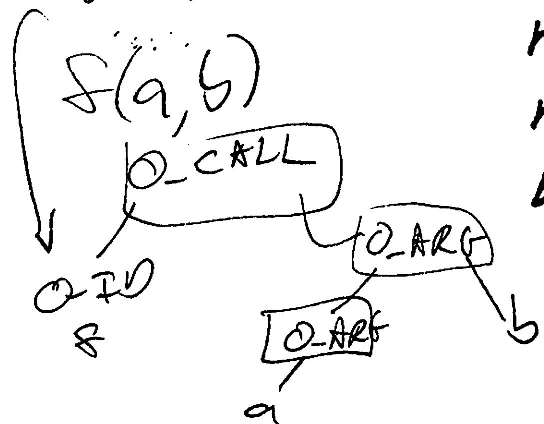
```
case O_ID: case O_CON:
        r = exprcode(op, 0, 0, 0, p);
        break;
```

op is + comes from p

```
case O_ADD: case O_SUB: case O_MUL:
case O_DIV: case O_MOD: case O_LSH:
case O_RSH: case O_BXOR: case O_BOR:
case O_BAND:

case O_NEG: case O_INDIR:
case O_BCOM: case O_CVI: case O_CVF:

case O_ASGN:

        r1 = walk(p->e_left, 0, 0);
        r2 = walk(p->e_right, 0, 0);
        r  = exprcode(op, r1, r2, 0, p);
        break;

case O_CALL:
        r2 = walk(p->e_right, 0, 0);
        r  = exprcode(op, 0, r2, 0, p);
        break;
```

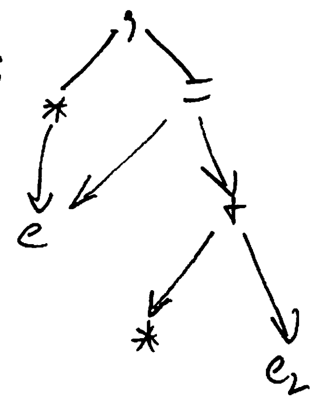
O_ARG

full [illegible] could be expr.



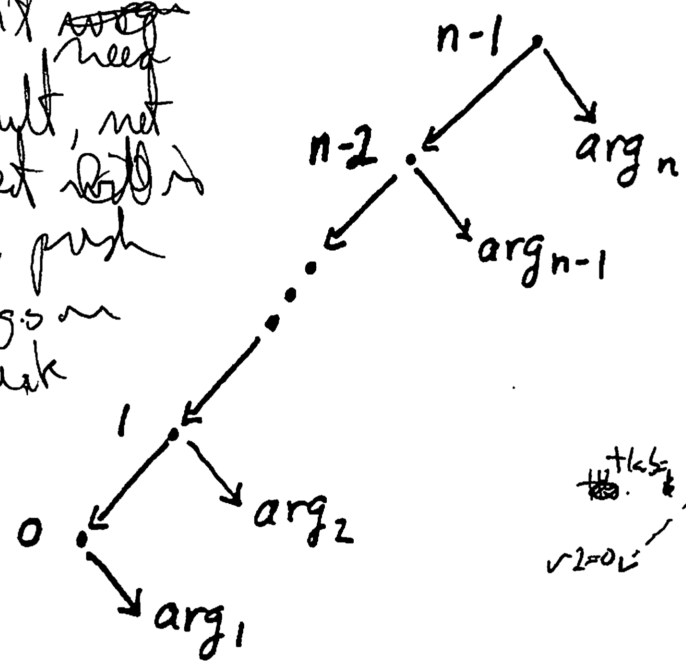
solely for e++ e++s

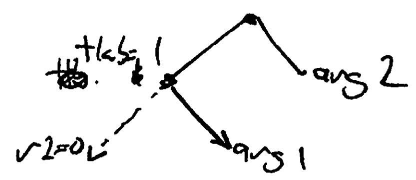
```
case ',':
    r = walk(p->e_left, 0, 0);
    walk(p->e_right, 0, 0);
    break;
```



```
case O_ARG:
    r2 = walk(p->e_left, ++tlab, 0);
    r1 = walk(p->e_right, 0, 0);
    exprcode(op, r1, r = r2+1, 0, p);
    break;
```

don't need result, not effect ... push args on stack





Short-Circuit Evaluation

- evaluate conditional only as far as is necessary to determine truth value.

```
if (a < 10 || b == 6)
        a = 1;
```

semantically equivalent to

```
    if (a < 10) goto L3
    if (b != 6) goto L1
L3: a = 1
L1:L2:
```

- obtain short-circuit evaluation by correct traversal of trees for conditional expressions.

```
ifstat(lab)
int lab;
{
    ...
        walk(expr(), 0, lab);
        :
}
```

Is never called w/ both tlab & flab

- must consider || && ! comparisons with both "true" and "false" labels.

- walk(e1 || e2 , tlab, 0) "fall thru on false" (if)

want:

```
if(e1) goto tlab
if (e2) goto tlab
   ↓
"false"
```

use:

```
walk(p->e_left, tlab, 0);
walk(p->e_right, tlab, 0);
```

- walk(e1 || e2 , 0, flab) "fall thru on true" (while)

want:

```
   if(e1) goto L
   if (~e2) goto flab
L:    ↓
   "true"
```

use:

```
walk(p->e_left, tlab = genlabel(1), 0);
walk(p->e_right, 0, flab);
deflabel(tlab);
```

• walk(e1 && e2 , tlab, 0) "fall thru on false"

want:

```
        if (~e1) goto L
        if (e2) goto tlab
   L:   "false"
```

use:

```
walk(p->e_left, 0, flab = genlabel(1));
walk(p->e_right, tlab, 0);
deflabel(flab);
```

• walk(e1 && e2, 0, flab) "fall thru on true"

want:

```
if (~e1) goto flab
if (~e2) goto flab
"true"
```

use:

```
walk(p->e_left, 0, flab);
walk(p->e_right, 0, flab);
```

or, and, not, never [illegible] to expand

• walk (!e , tlab, flab)

use:

```
walk(p->e_left, flab, tlab)
```

- walk(e1 < e2, tlab, o) "fall thru on false"

want: if (e1 < e2) goto tlab

use:

```
r1 = walk(p->e_left, tlab, 0);
r2 = walk(p->e_right, tlab, 0);
exprcode(O_LT, r1, r2, tlab, p);
```

↑ used in code generator

- walk(e1 < e2, o, flab) "fall thru on true"

want: if (e1 >= e2) goto flab

↑ complement operator

use:

```
r1 = walk(p->e_left, 0, flab);
r2 = walk(p->e_right, 0, flab);
exprcode(O_GE, r1, r2, flab, p);
```

if (a) → if (a != 0)

cnode(node) → if not a cmp op or result != COND add != 0 node

Do this [not]!

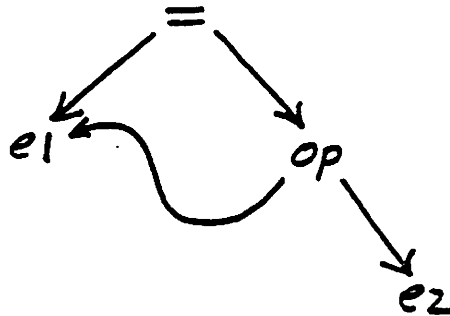
Storage Management

- **must** avoid using storage (for nodes) proportional to program size.
- want to reach "steady state".
- maintain free list of available nodes as in csym.c for symbol table entries.
- add nodes (after traversal) to free list:

```
int walk(p, tlab, flab)
struct node *p;
int tlab, flab;
{
        :
        switch (op = p->e_op) {
            :
        }
        if (op != O_ID && op != O_CON) {
        freenode(p->e_left);
        freenode(p->e_right);
        }
        p->e_result = r;
        return (r);
```

O_ID, O_CON need special handling { don't call freenode for symbol ptrs }

- problem: what about e1 op= e2 ?



- must avoid putting e1 on free list twice.
- maintain <u>reference</u> <u>count</u> in e_count field; freenode decrements e_count, frees node if e_count == 0.

  (margin note: # of pointers points to node)

  (margin note: bump count of l & r trees)

- e_count is number of references to the node; computed during tree construction

```
freenode(p)
struct node *p;
{
        if(p && --p->e_count == 0) {
                p->e_left = freelist
                freelist = p;
                }
}
```

- useful during <u>code generation</u> for resource management.

enode does semantics while expr does syntax

Type Checking

- p = enode(op, left, right)
  - check types of left & right
  - allocate node – and others if necessary
  - fill in e_type field
  - return pointer to new node
- enode embodies <u>expression</u> <u>semantics</u>
- may need to <u>add</u> "conversion nodes", e.g. a_CVF, a_CVI
- may need to "convert" lvalues to rvalues.
- may need to "widen" chars to ints.
- need to do "type algebra", e.g. if p is *T, then *p is a T; if a is an int, then &a is a *T (int); etc.
- may need to add "!= 0" to expressions to yield conditional expressions.

```
struct node *enode(op, left, right)
int op;
struct node *left, *right;
{
        int ty;    /* type of result */

        ty = T_VOID;
        switch (op) {
            case O_ID:
                :

            case AND: case OR:
                :

            default:
                fprintf(stderr, "compiler error");
                exit(1)
            }
        return (node(op, ty, left, right))
}
```

where <u>node</u> allocates a struct node and initializes its fields as given; also initializes e.result (0), e_count (0); and <u>increments</u> reference counts of left & right nodes (watch out!) id + const

```
struct symbol *q;
   :
case O_ID:
     q = (struct symbol *) left;
     ty = q -> s_type;
     if((q -> s_flags & ARRAY) == 0)
           ty |= T_ADDR;
     break;
```

"this node is an lvalue"

```
case O_CON:
     q = (struct symbol *) left;
     ty = q -> s_type;
```

- O_ID & O_CON <u>could</u> be handled directly by calling node in term(), since there is no type checking.
- same for O_ARG; others that do not require checking, such as ';'.

```
case O_NEG:
        left = widen(left);
        ty = left -> e_type;
        if (ty != T_INT && ty != T_FLOAT)
            err("type conflict for unary -", "");
ty=T_INT;
        break;
        :

struct node *widen(p)
struct node *p;
{
        p = rvalue(p);
        if (p->e_type == T_CHAR)
            p = node(O_CVI, T_INT, p, NULL);
        return (p);
}

struct node *rvalue(p)
struct node *p;
{
        if (ISADDR(p->e_type))
            p = node(O_INDIR, p->e_type& ~T_ADDR,
                        p, NULL);
        return (p);
}
```

```
case O_ADD: case O_SUB: case O_MUL: case O_DIV:
        left  = widen(left);
        right = widen(right);
        tl = left->e_type;
        tr = right->e_type;
        if (tl == T_INT && tr == T_INT) {
                ty = T_INT;
                break;
                }
        if (tl == T_FLOAT && tr == T_FLOAT) {
                ty = T_FLOAT;
                break;
                }
        if (tl == T_INT && tr == T_FLOAT) {
                ty = T_FLOAT
                left = node(O_CVF, T_FLOAT, left,
                                        NULL);
                break;
                }
        if (tl == T_FLOAT && tr == T_INT) {
                ty = T_FLOAT;
                right = ...
                break;
                }
```

hint: make this mess a function!

p++ → p+1
tptr

```
if (ISPTR(tl) && tr == T_INT &&
    (op == O_ADD || op == O_SUB)) {

        ty = tl;
        break;

err( "type conflict for ", opnames[op]);
ty = T_INT;
break;
```

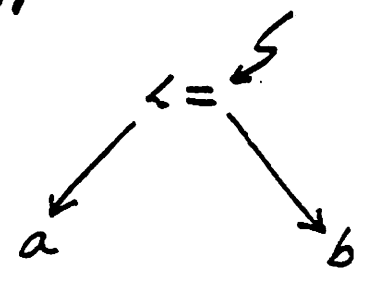
avoids further errors.

```
case O_LSH: case O_RSH: case BXOR:
case O_BOR: case O_BAND: case O_MOD:
    left = widen(left);
    right = widen(right);
    if (left->e_type != T_INT ||
        right->e_type != T_INT)
            err( "type conflict for ", opnames[o
    ty = T_INT;
    break;
```

```
case B_BCOM:
    as above, omitting right tree
```

Conditional Expressions

- have "type" T_COND

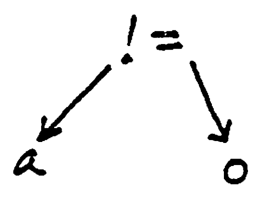


- "conversion" of expression to T_COND involves adding comparison with 0, with associated type-checking ⇒ use enode.

```
if (a) ...
```

is equivalent to

```
if ( a != 0) ...
```




```
struct node *cnode(p)
struct node *p;
{
        if (p->e_type != T_COND)
            p = enode(O_NE, p, node(O_CON,
                T_INT, constant("0", T_INT),
                NULL));
        return (p);
}
```

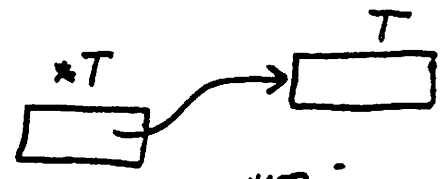
use enode to get type checking.

enode cont'd

```
case AND: case OR:
        left = enode(left);
        right = enode(right);
        ty = T_COND;
        break;

case O_LT: case O_LE: case O_EQ:
case O_NE: case O_GE: case O_GT:
        ty = T_COND;
        +--------------------------+
        | same as O_ADD, etc.      |
        | for T_INT, T_FLOAT,      |
        | but don't set ty         |
        +--------------------------+

 ? {    if (ISPTR(tl) && tr == T_INT &&
            (op == O_EQ || op == O_NE))
                break;
        if (ISPTR(tl) && tl == tr)
                break;
        err("type conflict for ", opnames[op]);
        break;
```

enode cont'd

Type Algebra!




```
case O_INDIR:                                   *T is an lvalue.
        left = rvalue(left);
        if ((n = ISPTR(left→e_type)) == 0)
            err("type conflict for unary *", "");
        ty = MKTYPE(n-1, BASETYPE(left→e_type));
        left→e_type = ty | T_ADDR;
        return (left);
```

- Note: node for O_INDIR constructed by rvalue, if T_ADDR set.
- assume you want lvalue, let rvalue handle other case.

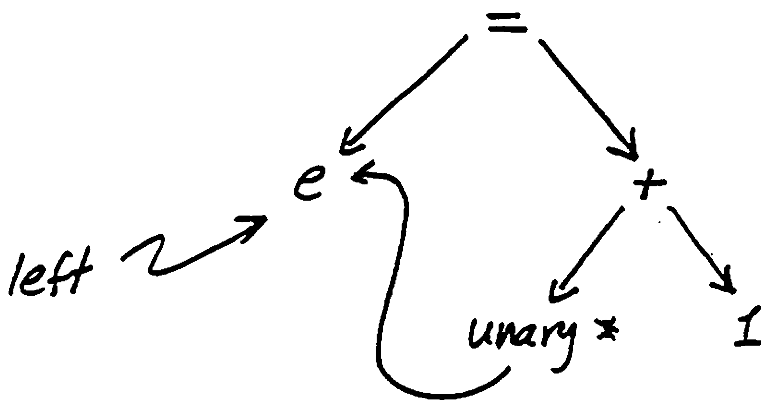
```
case UNARY '&':      { define UNARY to be something to distinguish operators, e.g. 010001 }
        if (!ISADDR(left→e_type))
            err("type conflict for unary &", "");
        ty = MKPTR MAKETYPE(1+ISPTR(left→e_type),
                left→e_type & ~T_ADDR);
        left→e_type = ty;
        return (left);
```

only "real" effect of unary &!!

$++e \equiv e += 1$

(see previous slide #49)

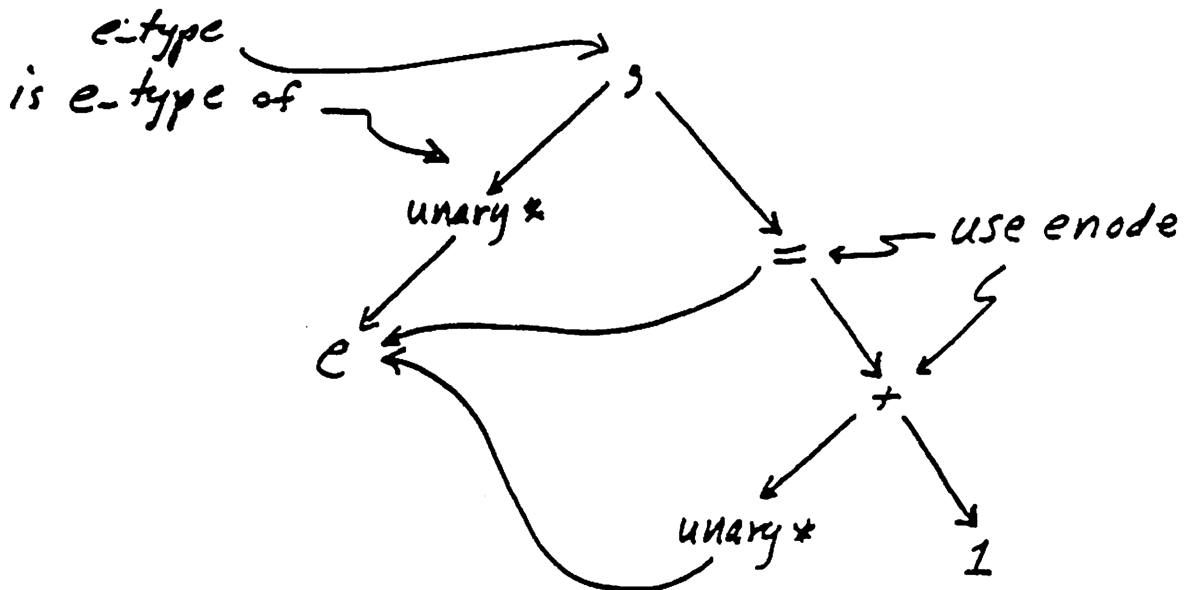
(likewise for --e)

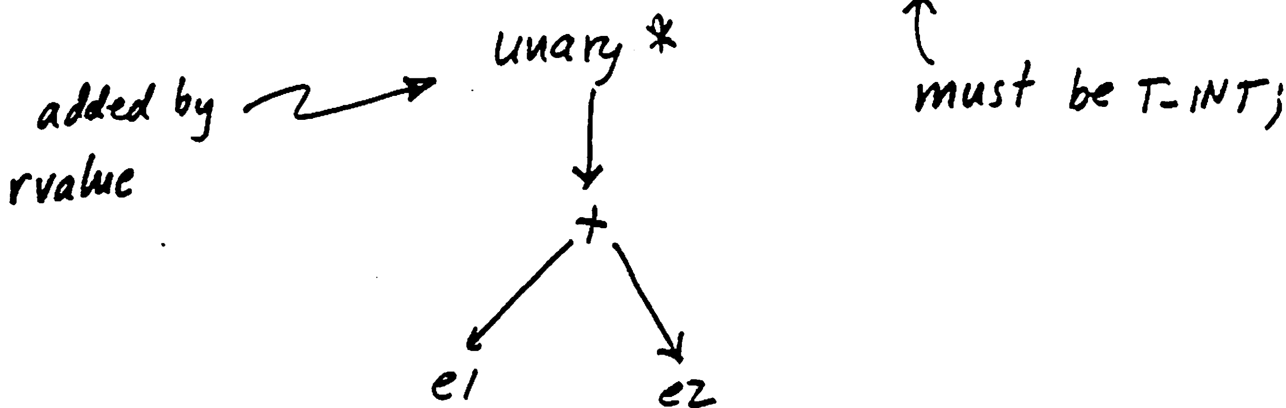



```
case UNARY INCR:
    left = lvalue(left);
    right = node(O_CON, T_INT,
                 constant("1", T_INT), NULL);
    right = enode(O_ADD, left, right);
    return(enode(O_ASGN, left, right));
```

enode adds unary * →

$e++ \equiv (t' = e,\ e += 1,\ t')$






```
case '[':
    left = rvalue(left);
    right = widen(right);
    if (right->e_type != T_INT)
        err("type conflict for subscript", "");
    if ((n = ISPTR(left->e_type)) == 0)
        err("type conflict for array name", "");
    ty = MKTYPE(n-1, BASETYPE(left->e_type));
    return(node(O_ADD, ty|T_ADDR,
                left, right));
```

like unary * (applies to the last three statements)

don't need enode because dirty work is done. (refers to `node`)

e1 = e2

lvalue T = T

lvalue int = float (convert to int)

lvalue float = int (convert to float)

lvalue (ptr T) = int

↑ result type

```
case O_ASGN:
    left = lvalue(left);
    right = widen(right)
    ty = left->e_type & ~T_ADDR;
    if (ty == T_CHAR && right->e_type == T_INT)
        ;
    else if (ty == T_INT && right->e_type == T_FLOAT)
        right = node(O_CVI, T_INT, right, NULL);
    else if (ty == T_FLOAT && right->e_type == T_INT)
        right = node(O_CVF, T_FLOAT, right, NULL)
    else if (ISPTR(ty) && right->e_type == T_INT)
        ;
    else if (ty != right->e_type)
        err("type conflict for =", "");
    break;
```

Only call O_INDIR for ~~[illegible]~~ source of "*".

f(...)

(f)(a, b, c)

```
case O_CALL:
        if (!ISFUNC(left->e_type)) {
                err("function expected", "");
                left = node(O_ID, T_INT,
                        lookup("?"), NULL);
        }
        ty = left->etype & ~(T_FUNC|T_ADDR);
        break;
```

semantic error recovery → (points to `left = node(...)`)

- enter "?" in symbol table during initialization
- use "?" whenever an ID is needed but you don't have one!
- give code generator well-formed, but maybe semantically incorrect, trees.

Computer Science 453
Fall 1983

Class Notes, Set 3

September 19, 1983

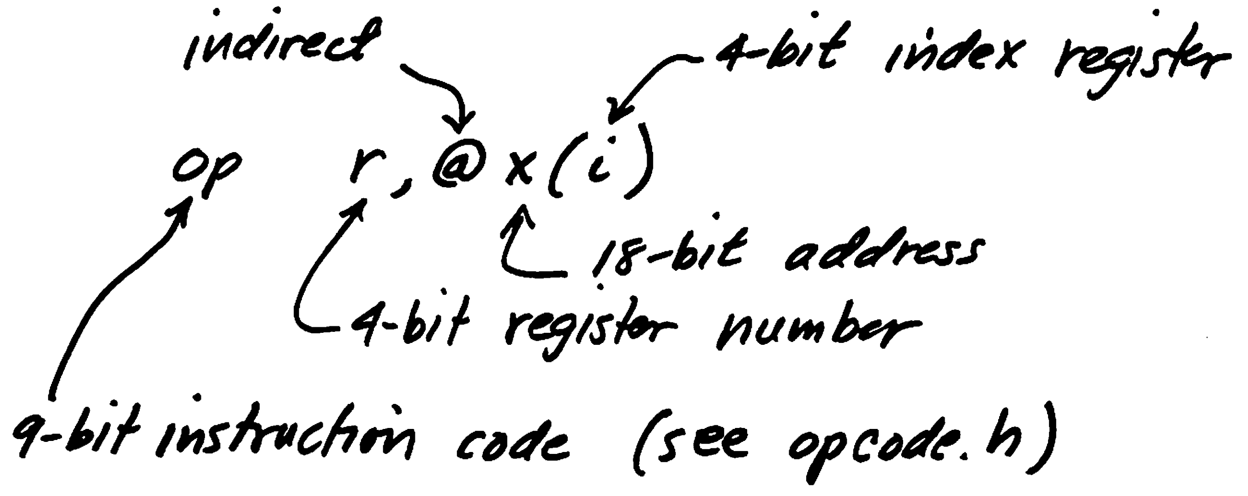
Code Generation

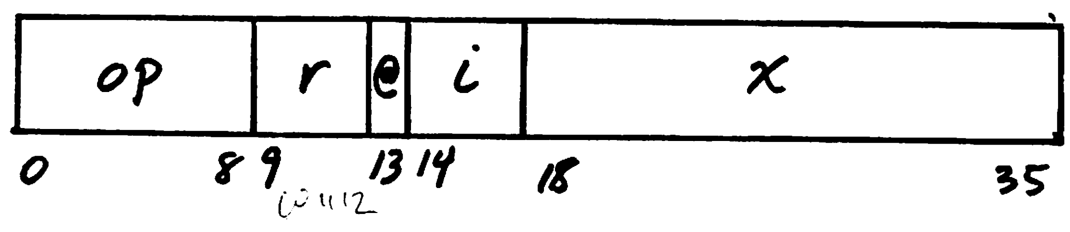
- replace bottom half routines by set for DEC-10
- generate link object code
- generate poor, but correct code
- manage machine resources, e.g. registers
- design run-time representation for programs, e.g. memory allocation for code, data; activation record layout for procedures.

Machine Model

- 2-address machine w/16 registers
- operators leave results in registers
- 2^{18} words (36-bit) of memory; 0-15 address the registers
- three logical segments
  - "code" - executable program code
  - "data" - global data
  - "lit" - constants
- code + lit are loaded into read-only memory.
- stack (in the data segment) for procedure activation records. ("frames").
- fixed-width (36-bits) instructions

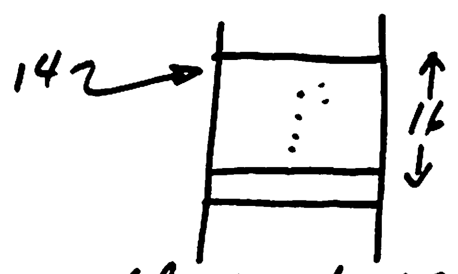
DEC-10 Instruction Set





Examples: (assembly language) [see opcode.h]

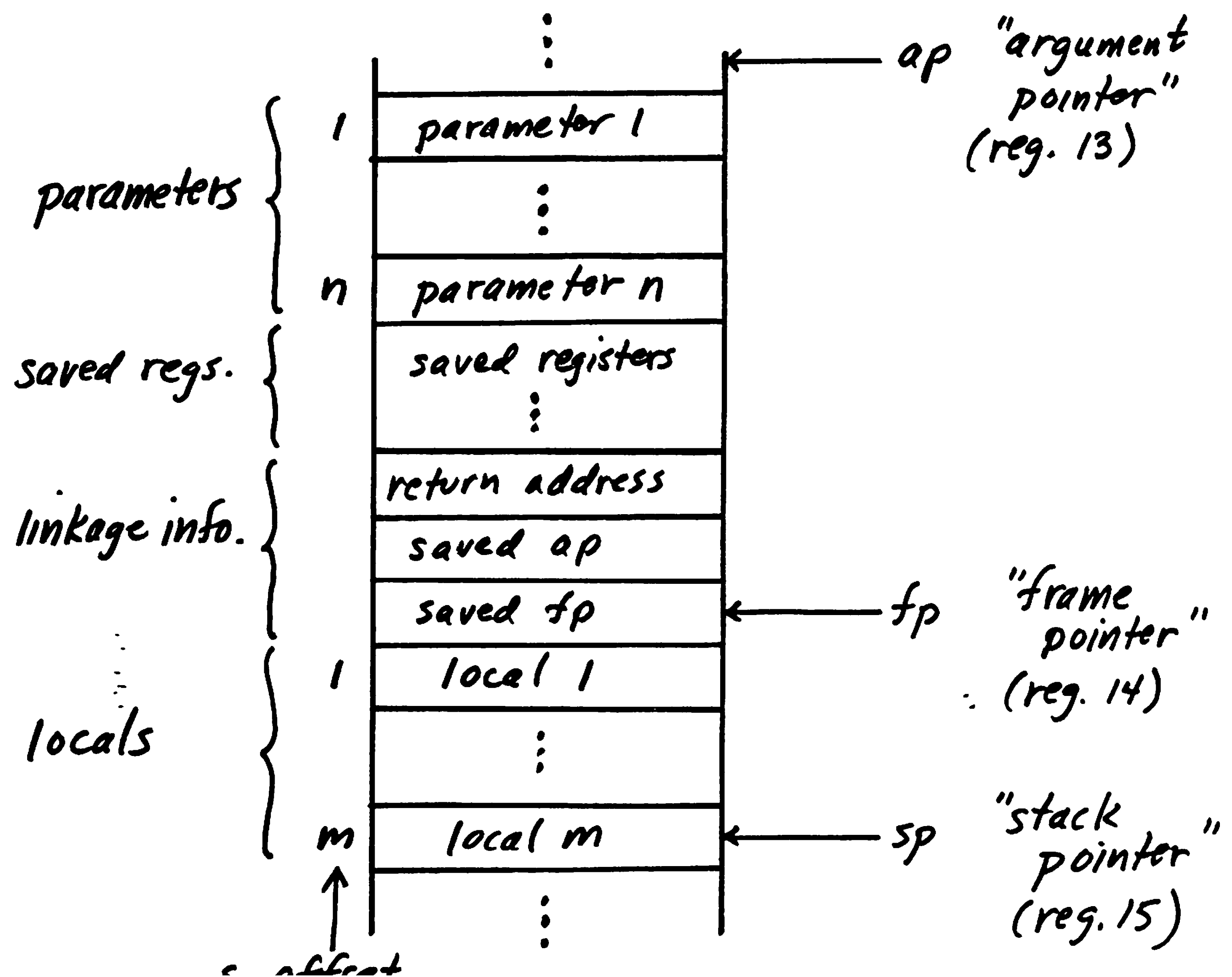
```
move   2, x        ; load value of x
add    6,7         ; add register 7 to 6
movem  13,16(14)   ; store register 13
```




```
addi   6, 40       ; add 40 to reg. 6
```

Activation Records

- allocated & initialized by calling sequence & entry code
- deallocated by exit code
- contains parameters, locals, linkage info, saved registers



Object Code

- commands & "object text"

- commands:

```
.def id expr
.seg id
.len id n
```

- expressions: polish suffix

```
expr : id
     | con
     | expr expr op

op   : + - < >
           ︸
         shifts
```

Example: movei 6, rows — "load address of rows into r6."

(movei = op, 6 = r, rows = x)

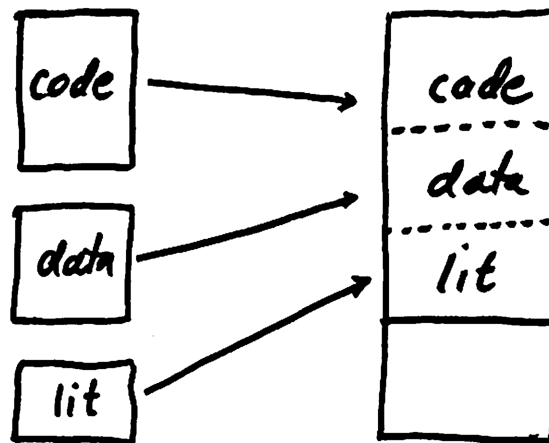
```
0201 27 < 6 23 < + rows +
└─── op ─┘ └─ r ─┘    └ x ┘

020130000000 rows +
```

op << 27
↑ unsigned
output two 16 bit #'s

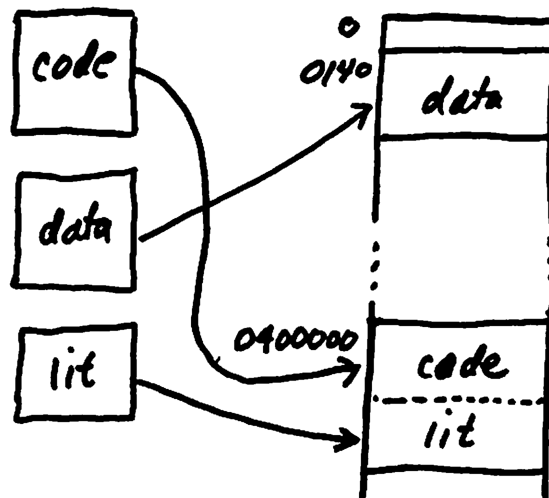
Segments

- code : executable program code (read-only)
- data : global data & the stack
- lit : constants (read-only)

- map 3 logical segments onto target machine:



CDC Cyber: single address space, not protected



DEC-10 :

"low" segment

"high" segment (protected)

(actually, DEC-10 is paged machine)

Segments cont'd

- during *execution*, segments are *contiguous*
- during compilation, segments are *interleaved*
- *link* converts *interleaved* to *contiguous* segments
- during compilation, use .seg commands to switch segments, e.g.

```
int a;
int g()
{
    a = g("hi there");
      :
}

.seg data
  (definition for a)
.seg code
  (code for g)
.seg lit
  (constant "hi there")
.seg code
  (more code for g)
```

Segments contd

- .def commands define symbols in terms of segment locations
- expressions refer to symbols

```
int a;
int f()
{
    :
    a = 1;
    :
}
```

```
.seg data
.def a data 0 +          <- "data"'s location counter
   {
.seg code
.def f code 014 +        <- "code"'s location counter
     :
 020130000000 a +        (load address of a)
     :
.len data 1    } length of each segment
.len code 36   } (in words)
```

Example: Local labels

```
for(i = 1; i < 10; i++) {
        sum += x[i];
        if (sum > 100)
            break;
        }
```

```
        i = 1
 L1:    if (i >= 10) go to L3
        sum += x[i];
        if (sum <= 100) goto L4
        go to L3
L4:L5:L2:   i ++
        go to L1
L3:
```

```
        .seg code
        i = 1
        .def $1 code 5 +
        if ( i >= 10) go to $3
         :
        .def $4 code 25 +
        .def $5 code 25 +
        .def $2 code 25 +
         :
         0254000000000 $1 +        (go to L1)
        .def $3 code 33 +
```

Local Labels cont'd

```
.def -s $n code loc +
```

(↑ the `-s` indicates "local" label (deleted from link output))

- must maintain "location counter" for each segment, which is used in definitions.
- location counter incremented for each word of object text emitted or allocated.

Curseg Curloc

```
deflabel(lab)
int lab;
{
        output ".def -s $lab name loc +"
}
```

- `$lab`: value of "lab"
- `name`: name of current segment
- `loc`: location counter for current segment

Segment Lengths

- at end of object file, issue .len commands giving length (in words) of each segment: e.g.

```
.seg data
   }
.seg code
   }
.seg lit
   }
.seg code
   }
.len data 44      <- total # words emitted or allocated
.len code 136
.len lit  7
```

```
progend(x)
int x;
{
    for each segment X
        if (X > 0 length)
            output ".len X n"      <- n: length of X
}
```

Data Segment

- contains definitions only – nothing to emit ("uninitialized data")

- len command must take allocation into account, e.g.

```
int a;
float x[10];
char s[8];

.seg data
.def a data 0 +
.def x data 1 +
.def s data 11 +
.len data 13
```

```
global(p)
struct symbol *p;                    if necessary!
{
        switch to data segment;
        output .def command;
        increment data location counter
          according to p -> s_size;
}
```

Bottom Half Routines

	defconst(p)	define a constant
✓	deflabel(lab)	define a label
	exprcode(op, r1, r2, lab, p)	expression operator
✓	global(p)	define a global
	jump(lab)	unconditional jump
	procbeg(p)	begin procedure
*	procend(x)	end procedure
*	progbeg()	begin program
✓	progend(x)	end program
	retcode(r)	return statement
*	rfree(r)	release resource r

✓ - done!

* - no output; perhaps initialization or other processing

Jumps

- unconditional jump:

  jrst label

- link object code:

  254

  0254000000000 label +

```
jump(lab)
int lab;
{
    emit(JRST, 0, NULL, lab, 0);
}
```

see opcode.h

DEC-10 Insts. only

"x"

- hint: write emit(op, r, name, off, i) to emit

  char   int

  op r, name ± off (i)

  [permit 0 args for omitted parts]

  in link object text, and increment appropriate location counter.

- use -d option to print expressions in symbolic (assembly language) form

Procedures

- procbeg must
  - initialize code generator
  - generate procedure entry sequence
  - assign frame offsets for parameters and locals.

- procedure entry sequence:

```
for tracing    {        pointer to "f" in lit segment
entry          {  f:    jsp  4, c.savr
sequence       {        adjsp  15, m
                                   ↑ size (in words)
emit(jsp,4,c.savr,0,0)             of local portion
emit(adjsp,15,0,0,m)               of the frame
```

- argument pushed on the stack in calling sequence (return address, too).
- linkage information saved, r13+r14 set by c.savr.

Procedures cont'd

- assume code location counter is 30, lit location counter is 6:

0331100000000 6 + lit +

```
.seg code
033100000006 lit +
.seg lit
014600000000        ("f" in 9-bit bytes)
.seg code

.def f code 31 +
ep ox8 ---> 0265 200000000 c.savr +
            0105 74000010
                      size of locals
```

- assign offsets 1, 2, ⋯ n to parameters, set s_offset field. [Done by top half!]

- assign offsets beginning at 1 to locals, increasing by size of each local, in any order.

Procedures cont'd

- retcode generates return sequence:

```
move 1, r    (if r ≠ 0)
jsp  4, c.ret
```

link code:

```
020004000000 r +
026520000000 c.retr +
```

- omit "move 1,r" if r == 0, e.g. retcode(0)

```
retcode(r)
int r;
{
      if (r)
            emit(MOVE, 1, NULL, r, 0);
      emit(JSP, 4, "c.retr", 0, 0)
}
```

Expressions (exprcode)

- manage resources, i.e. registers
- leave result of operation in a register
- determine type-specific operation (e.g. int add, float add) given generic operation (O_ADD) and operand types.
- generate correct (but awful!) code

a = b+c (all int)

yields:

```
movei   6, a      ; load a's lvalue
movei   7, b      ; load b's lvalue
move    8, (7)    ; load b's rvalue
movei   7, c      ; load c's lvalue
move    9, (7)    ; load c's rvalue
add     8, 9      ; a+b+c
movem   8, (6)    ; store in a
```

note reuse of reg. 7 → (points to `movei 7, c`)

could be:

```
move    6, b      ; b's rvalue
add     6, c      ; add c's rvalue
```

```
int exprcode(op, r1, r2, lab, p)
int op, r1, r2, lab;
struct node *p;
{
        int r;
        struct symbol *q;

        r = 0;
        switch (op) {
                :
            case O_ADD:
                (set r)
                break;
                :
            default:
                fprintf(stderr, "compiler error\n");
                exit(1);
        }
        return (r);
}
```

Register Usage

register	use
0	not used
1	function return values
2-5	entry/exit sequence
6-12	saved across calls
13	parameter pointer
14	frame pointer
15	stack pointer

- use 6-12 for evaluating expressions
- if run out of registers, issue error "expression too complicated"
- * what if you only had 3 registers? temporaries?
- 13-15 are initialized by "start-up code" prior to calling main; (stack is also allocated).

Constants

- reference to a constant is to its rvalue :

```
move r, $L
  :
.seg lit
.def -s $L lit 1 +
constant
```

store L in s_offset field, allocate if necessary

output by defconst (the .seg lit / .def / constant lines)

```
case O_CON:
    q = (struct symbol *) p->e_left;
    if (q->s_offset == 0)
        q->s_offset = genlabel(1);
    emit(MOVE, r = ralloc(1), NULL,
            q->s_offset, 0);
    break;
```

will change labels in tryhull

"just better execute"
2,3,5,6

- ralloc(1) - allocates 1 register in 6-12. (useful to allocate lowest # first).

"Immediate Mode"

"address is value"

- immediate mode: address <u>is</u> the value instead of the address of the value.

```
move 6, $10
   :
.seg lit
.def-s $10 lit 1 +
45
```

} load 45 into reg. 6

also done by

```
movei 6, 45
```

- used to load addresses & small constants (<= 18 bits).

```
case O_CON:
    q = (struct symbol *) p->e_left;
    if ((q->s_type == T_CHAR ||
        q->s_type == T_INT ) &&
        (n = atoi (q->s_name)) <= 0777777)
            emit(MOVEI, r = ralloc(1), NULL,
                        n, 0);
  else {
    ...
  }
```

Lvalues / Addresses

- lvalues are <u>addresses</u> of variables

- lvalue representation must be coordinated with O_INDIR & O_ASGN.

- simple case: 18-bit address

- hard case: byte pointer (for character strings)

- simple lvalues: depend on scope

  global: name (defined elsewhere) by global

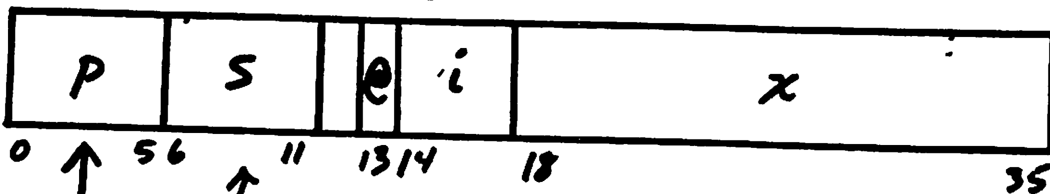
  param: s_offset(13)

  local: s_offset (14)

```
case O_ID:
    r = ralloc(1);
    q = (struct symbol *) p->e_left;
    if(q->s_flags & GLOBAL)
        emit(MOVEI, r, q->s_name, 0, 0);
    else if (q->s_flags & PARAM)
        emit(MOVEI, r, NULL, q->s_offset, 13,
    else
        emit(MOVEI, r, NULL, q->s_offset, 14);
    [handle byte pointers]

    break;
```

Byte pointers - interpreted by hardware

- 36-bit pointer specifying byte size, byte position, address

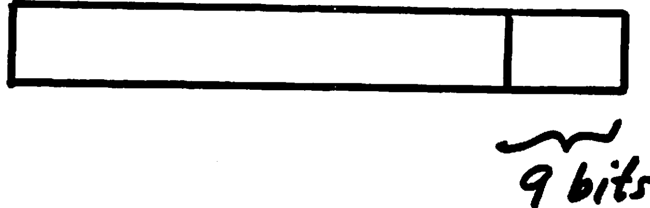


byte size (9 bits for characters)

number of bits to the right of byte.

Byte Lvalues

- identifier is a T_CHAR :



must treat s[i] and c (both T_CHAR) in same fashion; return byte pointer for both.

return byte pointer to rightmost 9 bits for characters :

in previous box {
```
if (q -> s_type == T_CHAR)
    emit(HRLI, r, NULL, 0001100, 0);
```
↑ octal

hrli 6,0001100

constructs :

(HRLI ≡ half word, right, to left, immediate!)

- more sophisticated code generator would avoid use of byte pointers here.

Byte Lvalues

- identifier is an array name: want

27	9		addr of id

i.e. pointer to first byte.

- do **not** set T_ADDR on O_ID nodes for array names (GLOBAL & LOCAL); then p->e_type == MKTYPE(1, T_CHAR)

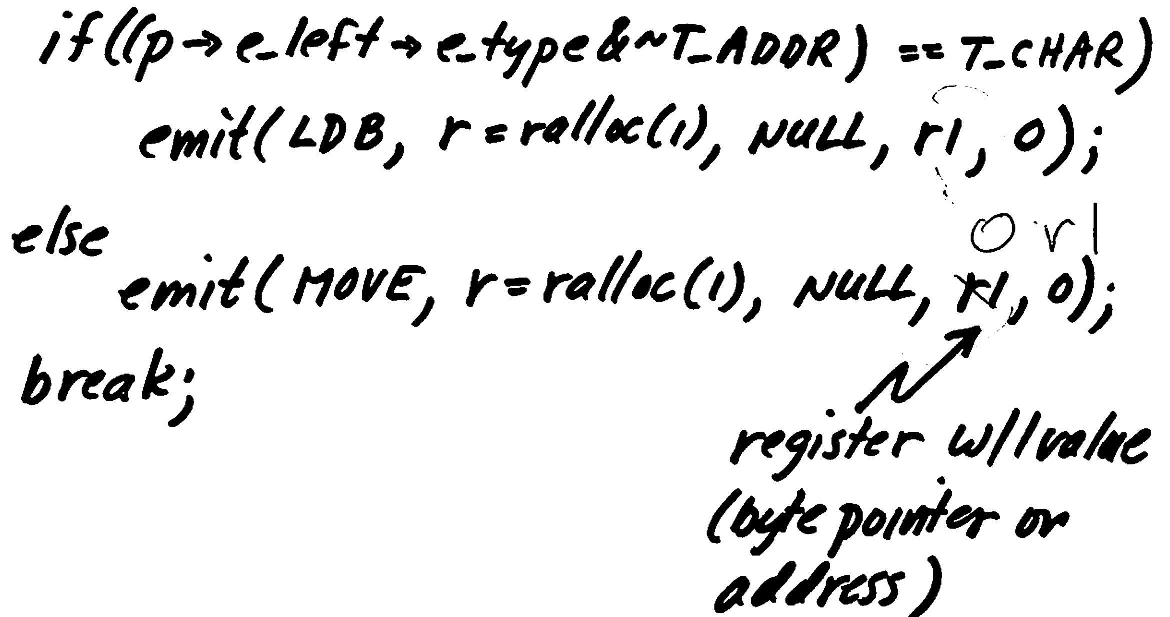
in previous box

```
    :
else if (p->e_type == MKTYPE(1,T_CHAR))
    emit(HRLI, r, NULL, 0331100, 0);
break;
```

- note type of result is T_ADDR|T_CHAR because it's an lvalue; use in O_INDIR & O_ASGN.

Loading Rvalues

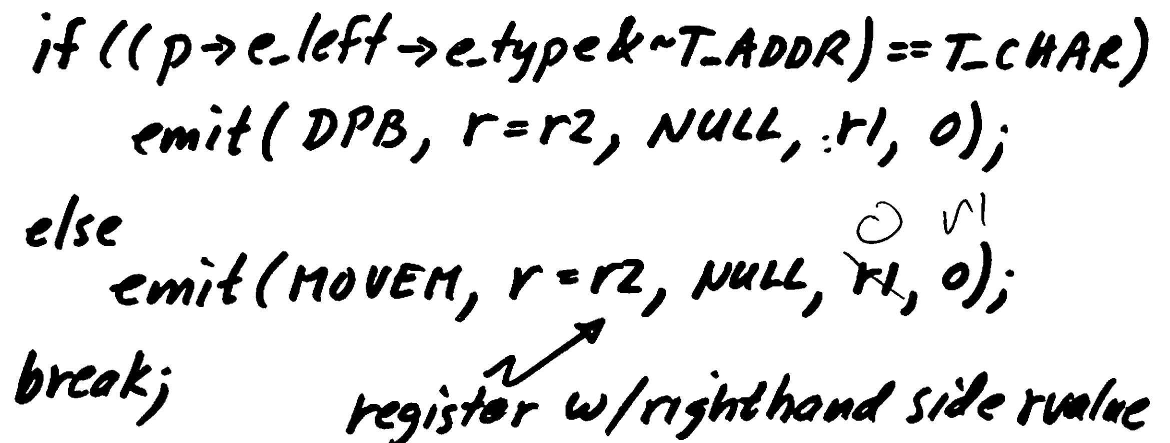
- for characters, use "load byte" instruction; for others, use "load word" instruction.



```
case O_INDIR:
    if((p->e_left->e_type&~T_ADDR) == T_CHAR)
        emit(LDB, r=ralloc(1), NULL, r1, 0);
    else
        emit(MOVE, r=ralloc(1), NULL, r1, 0);
    break;
```

register w/lvalue (byte pointer or address)

- O_ASGN is similar:



```
case O_ASGN:
    if ((p->e_left->e_type&~T_ADDR) == T_CHAR)
        emit(DPB, r=r2, NULL, r1, 0);
    else
        emit(MOVEM, r=r2, NULL, r1, 0);
    break;
```

register w/righthand side rvalue

defconst

- called at end of compilation with each constant; to be placed in lit segment.
- if referenced, will have non-zero s_offset field (see p. 21).
- output label and constant value:

T_INT, T_CHAR: print the value, e.g.

```
.def -s $16 lit 1 +
45
```

T_FLOAT: print bit pattern given by atof, e.g.

```
.def -s $16 lit 1 +
0206706314631      ← 56.8
```

byte ptr as rvalue

MKTYPE(1, T_CHAR): print byte pointer followed by string in 9-bit bytes, e.g. "hello"

```
                       .def -s $16 lit 1 +     ↙ next word
byte pointer to   →    0|331||100|000|002 lit +
next word              0150145154154      ← h e l l
                       0157000000000
                           ↑ null byte
```

don't forget null byte; e.g. "four" takes 3 words

Addition

- possible operand types are

  ptr T + int

  int + int

  float + float

- special case ptr char + int; treat ptr T + int like int + int. (i.e. addresses are added w/ normal addition *except* byte pointers)

```
case O_ADD:

    if (p->e_left->e_type == MKTYPE(1,T_CHAR))
        emit(ADJBP, r=r2, NULL, r1, 0);
    else if (p->e_left->e_type == T_FLOAT)
        emit(FADR, r=r1, NULL, r2, 0);
    else
        emit(ADD, r=r1, NULL, r2, 0);

break;
```

~~MUL, SUB, LT~~

table indexed by O_ values

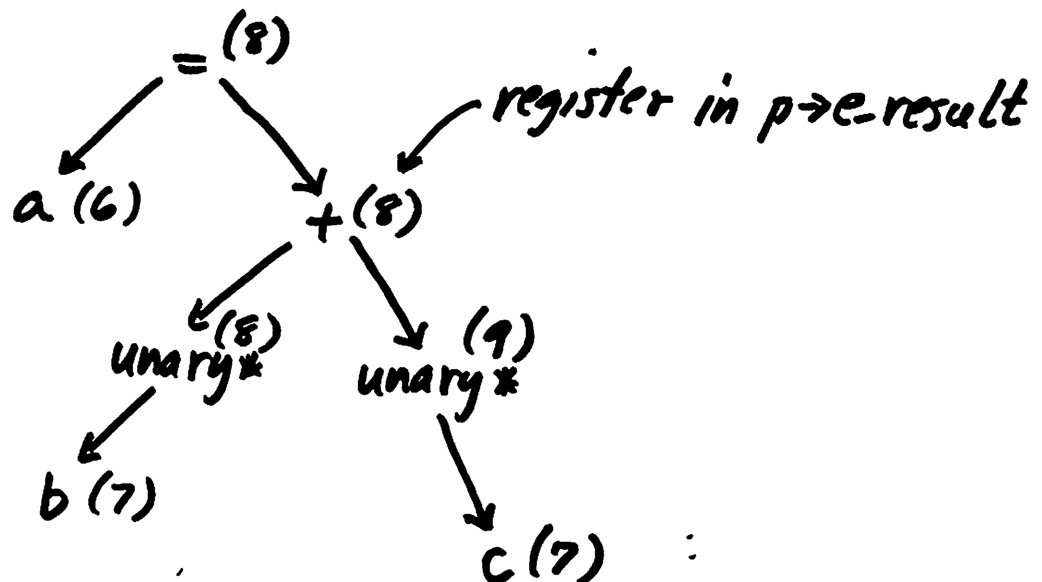
+
−
*
XOR
OR
AND

Can't do shifts

- hint: this code can handle several operators

Register Management

- allocate registers (6..12) via ralloc(1)
- free registers via rfree(r), called when node is freed during traversal by top half.
- maintain reference count in code generator indicating number of uses of each register. Be careful!
- Example: a = b + c



- note multiple occurrences of register 8, which arise from "copying" register in o_ADD & o_ASGN code (e.g. r = r1 in o_ADD).

Register Management, cont'd

- "copying" is equivalent to "allocating"
- reference count must be incremented when register is copied.

- ralloc(1) : finds a register r with rcount[r] == 0, sets rcount[r] to 1, and returns r. [issue error if none available!]

Think about this

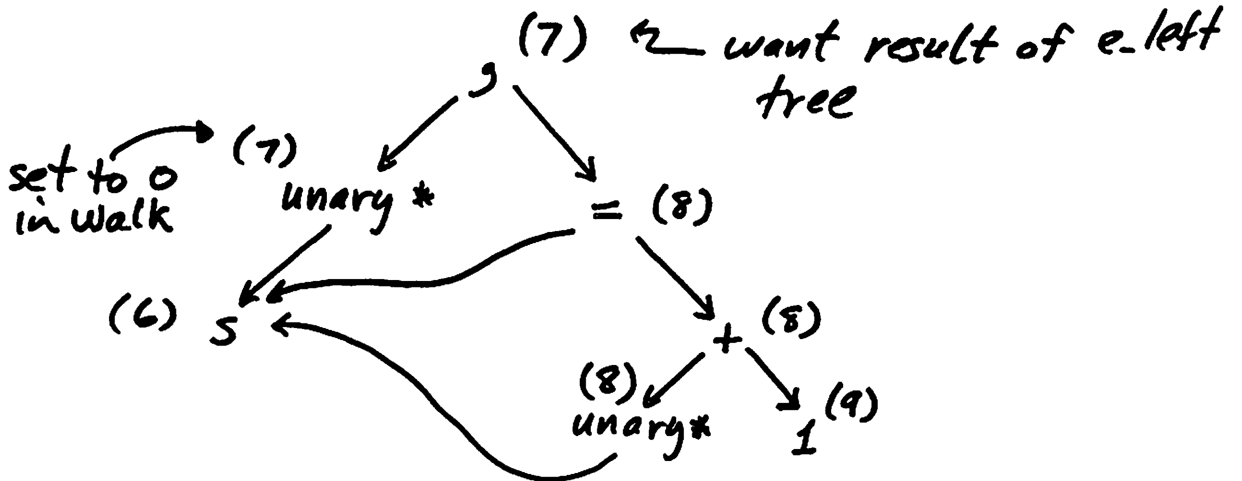
- "copying" r : increment rcount[r]
- rfree(r) : decrement rcount[r]

```
case O_ADD:        (copies r1 or r2)
    :
  else
      emit(ADD, r = r1, NULL, r2, 0);
→  rcount[r]++;
   break;
```

- also need rcount[r]++ in O_ASGN.
- rfree called almost exclusively by top half.

Register Management, cont'd

- problem: root node of s++



- "," causes a "copy" of register 7, but exprcode is not called, so reference count is wrong!

- solution: "move" register 7 to "," node in walk:

  could call exprcode directly
  Better code requires RB to copy regs

```
case ',':
        r = walk(p->e_left, 0, 0);
        walk(p->e_right, 0, 0);
        p->e_left->e_result = 0;
        break;
    ...
  }
p->e_result = r;
return(r);
```

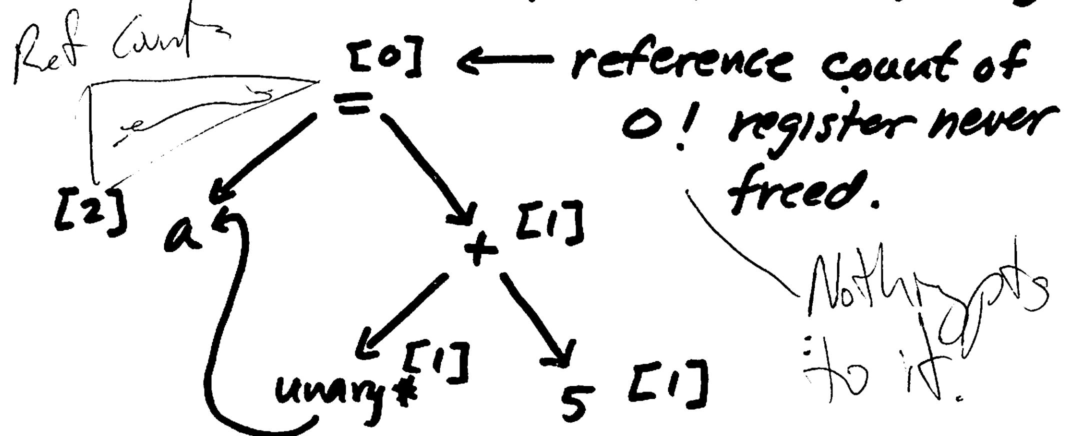
(annotation on `p->e_left`: unary '*')

freenode

```
freenode(p)
struct node *p;
{
        if(p && --p->e_count == 0) {
            if(p->e_op != O_ARG && p->e_result)
                  rfree(p->e_result);
            p->e_left = freelist;
            freelist = p;
            }
}
```

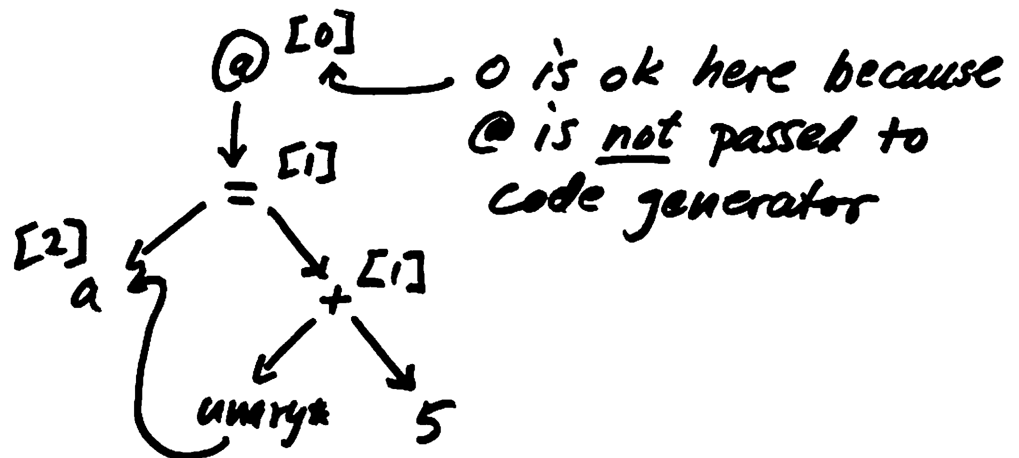
left of p can't be O_ID or O_CON

• problem: what about root node: a += 5



freenode, cont'd

- solution: add another operator at the root of the tree so reference count is 1
- used only for tree traversal to handle register deallocation properly



- in walk:

```
case '@':
        r = walk(p->e_left, tlab, flab);
        freenode(p->e_left);
        freenode(p);
        return(r);
```

p->e_result is 0. → freenode(p);

why? just retcode?

- add @ node to result of call to expr() (but in right places!)

Division / Modulus

- integer division (and modulus) requires 2 adjacent registers.

```
idiv 6, 8
```

leaves quotient in register 6 and remainder in register 7.

- if rcount[r1+1] == 0, fine otherwise must move r1
- ralloc(2) : allocate 2 adjacent registers, return first one.
- free extra register, if allocated.
- O_MOD is similar.

```
case O_DIV:

    if (p->e_left->e_type == T_FLOAT) {
        emit(FDVR, r = r1, NULL, r2, 0);
        rcount[r]++;
        break;
    }

    if (rcount[r1+1] == 0) {
        emit(IDIV, r = r1, NULL, r2, 0);
        rcount[r]++;
    }
    else {
        emit(MOVE, r = ralloc(2), NULL, r1, 0);
        emit(IDIV, r, NULL, r2, 0);
        rfree(r+1);
    }
    break;
```

ravail

rcopy

do NOT free r1 ! why?

left operand is in r1

count will be -1 as it is later freed

Comparisons

- use "compare & skip" instructions for all but char * comparisons.

  e.g. r1 < r2, use

```
camge r1,r2     "skip" if
jrst  $lab        r1 >= r2
```

- call library routine to compare char *'s, "jump" on result.

  e.g. r1 < r2, use

```
push   15,r1
push   15,r2
pushj  15,c.scp
adjsp  15,-2        (remove args)
jumpl  0,$lab
       ↑
  result (<0, =0, >0) in reg. 0.
```

Vac conv.
Derf. Trapped

Computer Science 453
Fall 1983

Class Notes, Set 4

October 6, 1983

Linking

- "standard" translation process:
  compile source programs,
  link object programs,
  load executable programs.

- linkers:
  - combine separately compiled object modules
  - "search" libraries
  - machine dependent
  - idiosyncratic syntax (e.g. DEC-10 .rel files)
  - archaic restrictions (e.g. ordering, naming) often influence language design! (e.g. 6-character names).
  - cumbersome i/o (binary)

- solution: get a new linker!

link

- machine independent linker
- language independent
- regular syntax, semi-readable
- no ordering or naming restrictions (e.g. long names permitted).
- features to simplify code generation (e.g. arbitrary # of segments)
- centralizes compiler's "last pass" functions
- does <u>not</u> include "exotic" features
  - overlays
  - report generation
  - "patching"
- Goal: elevate linkers to useful tools, functional standardization of linkers

Linking & Loading

- object code:
  - output of compilers & assemblers
  - incompletely bound addresses
    e.g. references to external symbols, forward references
  - ".rel" files on the DEC-10
    ".o" files on UNIX
    "TEXT" files on IBM under CMS
  - often includes "relocation" information indicated by flags pointing to locations that must be adjusted upon loading.
- executable code:
  - output of linkers
  - (almost) completely bound addresses, depends on operating system
  - input to operating system "execute" system call.
  - machine dependent
  - ".exe" files on the DEC-10
    "load modules" on the IBM

Linking & Loading, cont'd

- linking: "linking loader", "relocating loader"

  object code → executable code

  e.g. link-10 on the DEC-10

- linking: "link editors", "binder"

  object code → object code (arrow looping back)

  e.g. ld on UNIX, IBM "linkage editor"

- link: a link editor

  link code → link code (arrow looping back)

- translation process:

  compilation: source code → link code

  linking: link code → link code ...

  loading: link code → executable code

  ↳ machine dependent (but small)

  machine independent (contents of link code is machine dependent; functions

Link Code

- lines of text (could use binary format).

- interleaved <u>commands</u> and <u>object text</u>.

- syntax:

```
file  : { cmd | expr }

cmd   : .def [-s] id expr
      | .seg id
      | .org con
      | .len id con

expr  : expr expr ( + | - | < | > )
      | id
      | con
```

init & char?

- identifiers: letters, digits, $. % _
- constants: octal (leading 0), decimal

Expressions

- Polish postfix - simplifies evaluation
- link makes no architectural assumptions e.g. word size, addressability

wrong!

- expressions represent 1 "interesting cell" as determined by compiler & loader; i.e. "interesting cell" is machine-dependent.
- typically, "interesting cell" contains relocatable address, e.g.

machine	cell size	address size	word size
DEC-10	18 bits	18 bits	36 bits
PDP-11	16	16	16
Cyber/175	15	18	60
IBM/370	16	24	32
VAX	16	16-32	32

- loaders combine cells into words by addition to permit overflow of cell boundaries.

Expressions, cont'd

- examples: (DEC-10)

```
move  0100 + 15 +      } 2 18-bit
%x  02 -               } cells

0                      } 2 18-bit
0206706314631          } cells w/overflow
           ↳ 56.8
```

- on DEC-10, loader combines

```
a
      (18-bits each)
b
```

via (a<<18) + b (36 bits)

- similarly for addresses etc. on IBM/370, Cyber, etc.

- shifting operators (< >) can be used to place relocatable address at any position, e.g.

```
0
x 9 <
```

places "x" in middle 18 bits on DEC-10.

Expressions, cont'd

- cell size determined only by convenience
- link is independent of cell size, but cell size must be fixed for all object files.
- link simply replaces symbols by their values, and evaluates expressions.
- goal: get rid of symbols! (except segment names).

Segments

- blocks of object text to be loaded into a contiguous region of memory

- segment names & meanings coordinated among compilers & loaders; link itself is independent of the "meaning" of segment names.

- input to link: any number of segments interleaved in any way

- output: segments output as contiguous blocks of object text.

- inter- and intra-segment references made by referring to base location of a segment, which is simply the segment name.

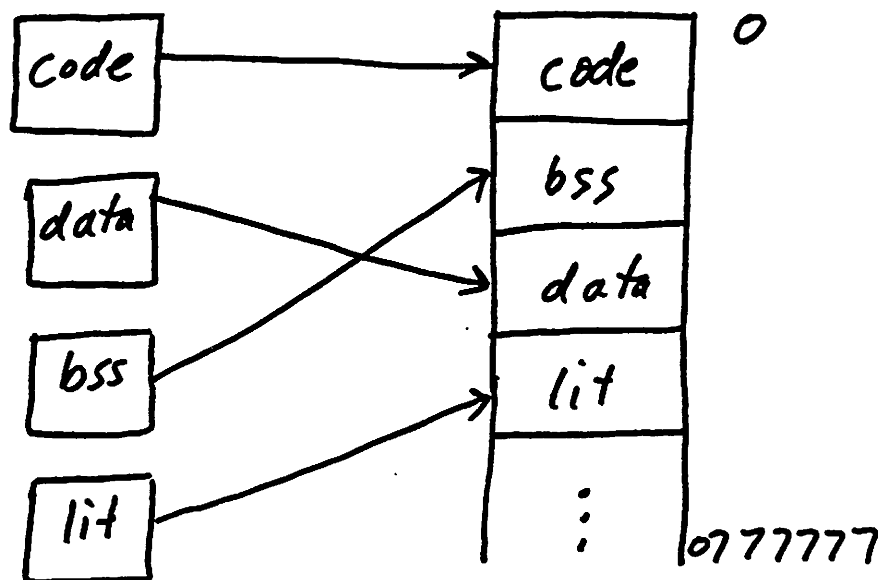
Segments cont'd

- Suppose compiler emits code for 4 segments:

  code - executable program code
  data - initialized data
  bss - uninitialized data
  lit - literals (constants)

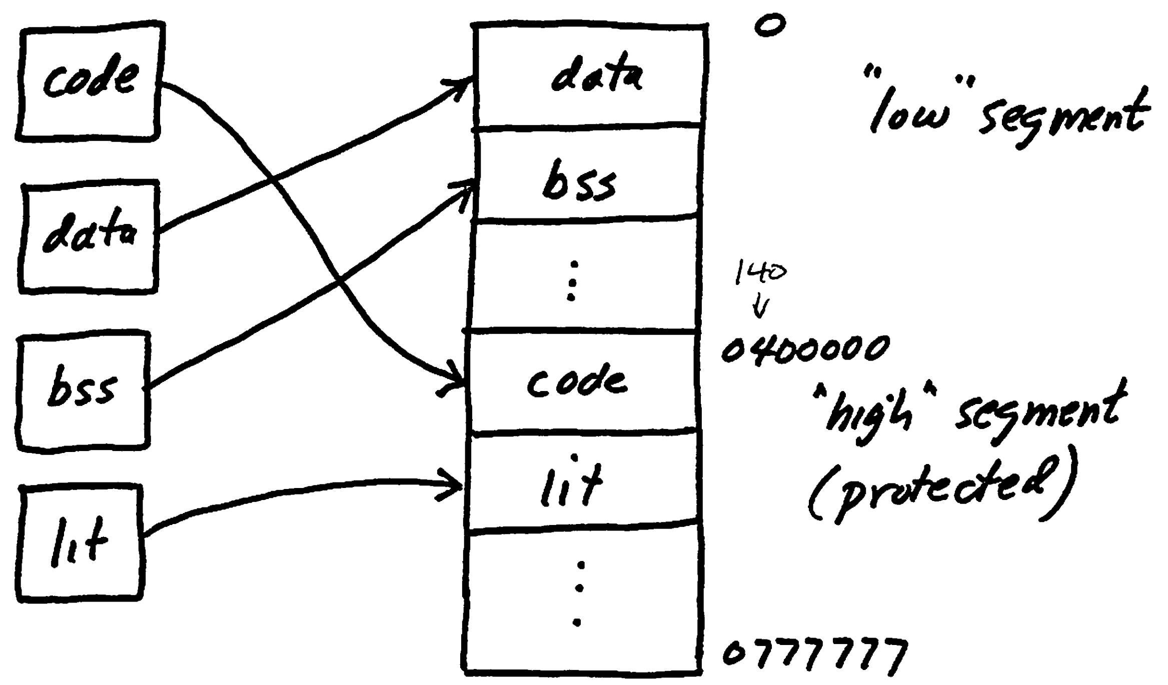
- loader maps these logical segments onto the target machine, using hardware as appropriate.

e.g. Cyber/175: single address space, no protection

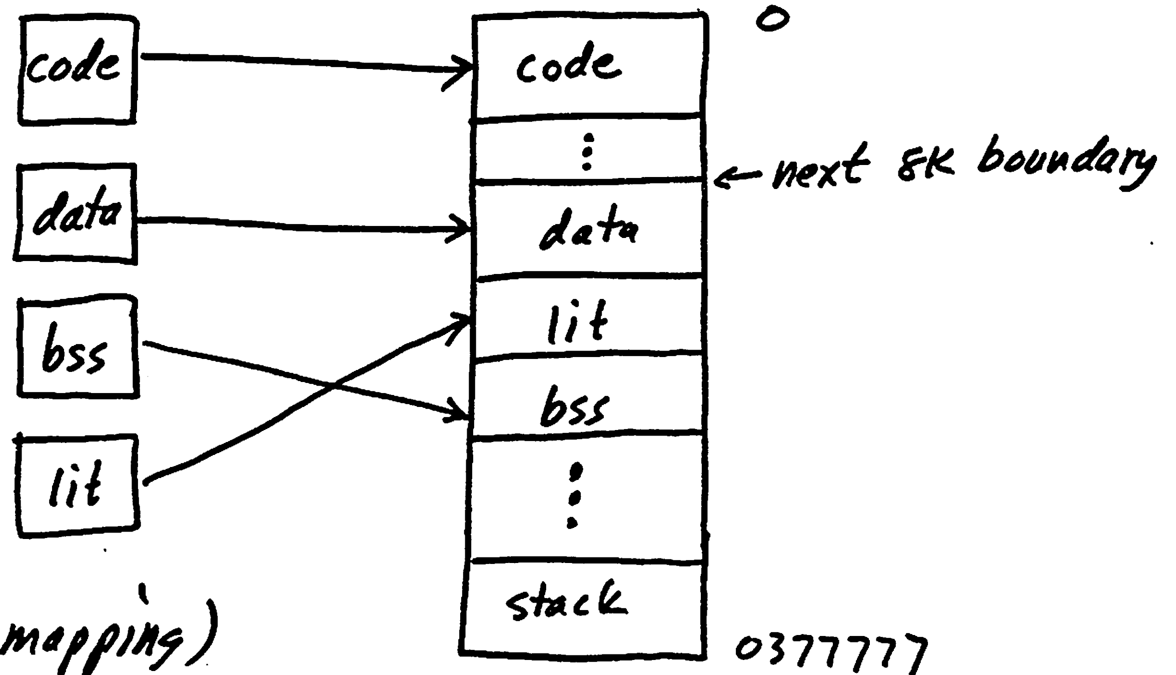


Segments cont'd

e.g. DEC-10: 2 segments, 1 protected



e.g. PDP-11/70: 8 segments, up to 8 protected



(UNIX mapping)

Segment Commands

.seg id

- start, or resume, placing text in segment <u>id</u>.
- may flip back & forth between segments as necessary to simplify code generation.

input:

```
int a;
int b = 6;
int g()
{
    ...
    a = g("hi there");
    ...
}
```

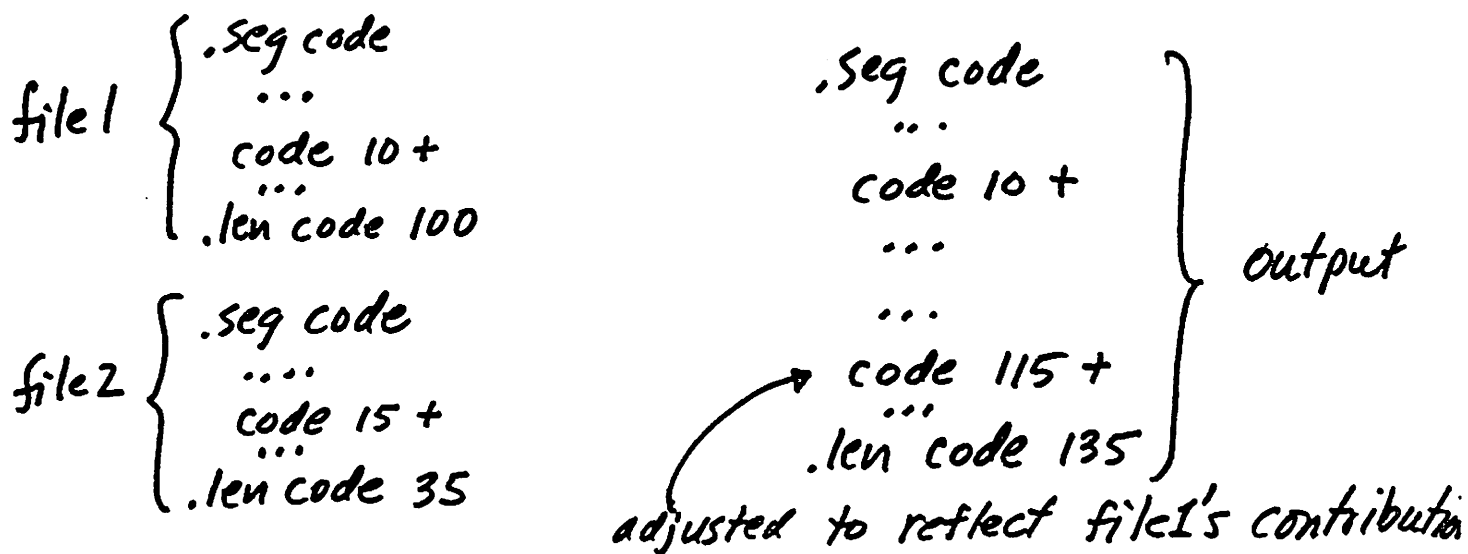
output:

```
.seg bss
 (definition for a)
.seg data
 (definition for b)
 6
.seg code
 (code for g)
.seg lit
 (constant "hi there")
.seg code
```

Segment Commands, cont'd

.len id con

- indicate length of segment *id* in units agreed upon by compilers & loader
- units *may not* be "interesting cell" e.g. on DEC-10, cell may be 18 bits, lengths in words (36 bits).
- segment lengths used by *link* to adjust base locations when several object files contribute to the same segment; used to obtain proper *relocation* of references in the output file



Segment Commands cont'd

- order of linking affects relocation (but not semantics!)

file1	file2
.seg code	.seg code
...	...
code 10 +	code 15 +
...	...
.len code 100	.len code 35

.link file1 file2	.link file2 file1
.len code 135	.len code 135
.seg code	.seg code
...	...
code 10 +	code 15 +
...	...
code 115 +	code 45 +
...	...

- len commands appear first in the output so loaders can compute segment addresses, if necessary, and allocate space.

Segment Commands, cont'd

.org con

- instructs loader to place text at different locations in current segment.

- gives "random access" capability (used for "backpatching").

- link copies org commands to the output, adjusting con if necessary

Symbols

- def commands define symbols as expressions (polish suffix)
- link evaluates expressions substituting expressions for symbols as necessary
- symbols are defined in terms of other symbols and segment names (i.e. base locations).

e.g.

.def rows data 25 +

(rows = symbol; data 25 + = "value")

subsequent line of object text

rows 4 +

becomes (after evaluation)

data 29 +

Symbols, cont'd

- symbols can be defined & referenced in any order (e.g. forward references)

- in expressions, segment names may be used as symbols.

e.g.

```
.seg code
.def a b
  ...
 a 1 +
  ...
.def b code 25 +
```

or/

```
.seg code
.def b code 25 +
.def a b
  ...
 a 1 +
  ...
```

both yield

```
.seg code
  ...
code 26 +
  ...
```

- also, local labels for control constructs, e.g., if, while, etc.

Symbol Sets

- Let
  - R be set of <u>referenced</u> symbols,
  - D be set of <u>defined</u> symbols,
  - S be set of <u>segment</u> names.

- R comes from symbols in expressions,
  D comes from .def commands (left operands)
  S comes from .seg & .len commands

- Goal: "get rid of symbols", or a <u>resolved</u> file, i.e. a file for which

$$R \subseteq D \cup S$$

- given values for symbols in S, a resolved file can be loaded.

- <u>unresolved</u> files have <u>external</u> <u>references</u>, i.e. symbols in R that are <u>not</u> in $D \cup S$.

- linking seeks to combine unresolved files into a single resolved file.

- linking is an iterative process; an <u>unresolved</u> file is <u>valid</u> output.

e.g. given 3 unresolved files that each reference symbols in all 3 files, can link in <u>1 step</u>:

```
link file1 file2 file3 -o out
```

(-o out: specifies output file)

or in <u>2 steps</u>:

```
link file1 file2 -o temp
link file3 temp -o out
```

- if $R \cup S \subseteq D$, file is <u>absolute</u>; must be loaded at locations consistent with segment definitions.

Output

- link produces output in a form that can be loaded in one pass:

```
.len commands
.def commands
.seg id_1
(text for segment id_1)
...
.seg id_k
(text for segment id_k)
```

(segments are contiguous)

- len commands are first so loaders can know lengths and hence position of all segments.
- def commands come next so symbols are defined before used in object text
- segments appear last, and are contiguous (no segment switching).
- output is valid link input!

Compiler Construction

- method 1 : use assembler to produce object code

  source → assembler lang. → object code

  e.g. C compiler

- advantages :
  - easy to generate assembler lang.
  - smaller compiler
  - readable output (easy to debug)
  - avoids "forward reference" problem
  - uses existing tool to do dirty work

- disadvantages :
  - slow
  - idiosyncratic (e.g. ordering, naming restrictions, inability to handle source constructs
  - fixed output format
  - at least 3 passes + linker
  - duplication of work (e.g. symbol table)

Compiler Construction cont'd

- method 2: produce object code directly

  source → object code

  e.g. most commercial compilers.

- advantages:
  - fast (sometimes)
  - fewer passes
  - smaller (and fewer) files

- disadvantages:
  - larger compiler
  - difficult to debug
  - idiosyncratic syntax, binary i/o
  - silly, archaic restrictions

- use link: retain advantages + disadvantages of both methods

  source → link code → executable code

may be fewer passes when linking taken into account; no worse than assembly lang.

Using link as an Assembler

- consider f(a+b, c); macro-10 assembly language code is (good code!)

```
move   2, a      ; load a
add    2, b      ; compute a+b
push   15, 2     ; push a+b
push   15, c     ; push c
pushj  15, f     ; call f
adjsp  15, -2    ; remove arguments
```

- in link code, "push 15, c" becomes (36-bit cells)

```
026174000000 c +
```

- can define symbol "push" to make this more readable:

```
.def push  0261000 18 <
...
push  15 23 < + c +
```

(15 23 < : register 15 in right place.)

- definitions such as push need be made only once.

link as an Assembler, cont'd

- can make "15 23 <" part of definition of push, if all push (and pushj, adjsp, etc.) are for same register; e.g.

```
.def push 0261740 18 <
  ...
push c +
```

which is as readable as assembly lang.

- assuming definitions for other opcodes and for registers (e.g. .def r2 2 23 <), above sequence is:

```
move r2 a + +     (or move r2+ a+)
add r2 b + +
push 2 +
push c +
pushj f +
adjsp 0777776 +
```

bogus!

link as an Assembler, cont'd

- typical usage: simply generate bit patterns in code generators, or have option to produce symbolic output for debugging.

- for assembling "small" assembly language programs, write program using symbolic form (shown above) and link with definitions file, e.g.

  link defs pgm.asm -o pgm.obj

- Experience with Y compiler:

  link code gen. is 15% shorter than macro-10 code gen.;

  link code gen. is 54% shorter than binary object code gen.

  (lines of code in output module).

Local Symbol Resolution

- "local" symbols can be hidden from outside by 3 different approaches.

Consider stack module in C:

```
static int sp;
static int stack[STKSIZE];
                    ↑ e.g. 100
push(x)
int x;
{ :
}
pop()
{ :
}
```

static also means the name won't be known outside of the file

- sp & stack are local symbols and are not accessible from outside the stack module.

Local Symbol Resolution, cont'd

- method 1: "suppress" definitions of sp & stack and *always* run output of compiler thru *link*:

```
.seg data
.def -s sp data 0 +
.def -s stack data 1 +
.seg code
.def push code 0 +
   :
.def pop code 36 +
   :
.len data 101
.len code 80
```

- -s option causes *link* to *omit* definitions for those symbols from the output; hence, sp & stack do not appear and cannot be referenced.
- disadvantage: must run link an extra time

Local Symbol Resolution, cont'd

- method 2: add string to local names (such as the file or module name) to make local names unique:

```
.seg data
.def -s stack.c.sp data 0 +
.def -s stack.c.stack data 1 +
.seg code
.def push code 0 +
  :
.def pop code 36 +
  :
.len data 101
.len code 80
```

- can still use -s to reduce # of symbols, but do not have to run link after every compilation.

Local Symbol Resolution, cont'd

- method 3: "backpatch" forward references when target is known using .org command.

- assume expression evaluation starts with "current value" of the cell on the stack; then

  code +

  means "add code" to current value.

Example:

```
while (a < 0) S

L1,L2   if (a >= 0) goto L3
        S
        goto L1
```

output:

code location counter →

```
011   movei  6, a          ; get a's lvalue
012   move   7, (6)        ; get a's rvalue
013   movei  8, 0          ; get 0
014   caml   7, 8          ; a < 0 ?
015   jrst                 ; no, goto L3
      ...
      S
      ...
021   jrst code 011 + +    ; jump to L1
022
```

Backpatching cont'd

- at location 022, L3 is defined; backpatch all references to L3 via .org commands:

```
.org 015
code 022 + +
.org 022
```

adds "code 022 +" to "jrst" at location 015.

- .org 022 puts location counter "back" for subsequent code

- backpatching specifications (i.e. locations to backpatch) maintained until label is defined, then appropriate sequence of .org commands are emitted.

- complicates compiler.

Computer Science 453
Fall 1983

Class Notes, Set 5

October 13, 1983

#1/

do stmt while (expr)

a) code template:

```
L    stmt                          "true" label
L+1  if (expr) goto L
L+2
```

b) add to stmt():

```
        case DO:                   "loop handle" L
added to    dostat (loop = genlabel(3));
first(stmt),  break;
follow(stmt)  ...

    dostat(lab)
    int lab;
    {
            t = gtok();
            deflabel((lab);
            stmt();
            deflabel(lab+1);
            mustbe(WHILE);
            mustbe('(');
            walk(cnode(expr()), lab, 0);
            mustbe(')');
            deflabel(lab + 2);
    }
```

#21 **Conditional Expressions**

- used wherever a value may be used; add "conversion" node in rvalue:

```
struct node *rvalue(p)
struct node *p;
{
        ....
        if(p->e_type == T_COND)
            p = node('?', T_INT, p, NULL);
        return(p);
}
```

- rvalue adds node with '?' operator that is used in walk to generate code according to following template:

```
              ┌─ p->e_left
        if(e) goto L
global
variable ──> c.cc = 0
        goto L+1
    L   c.cc = 1
    L+1 (return value of c.cc
```

- in walk:

```
case '?':
        walk(p->e_left, lab=genlabel(2), 0);
tree for          {  p1 = node(O_ID, T_INT|T_ADDR,
lvalue of c.cc    {                lookup("c.cc"), NULL);

tree for          {  walk(node(O_ASGN, T_INT,
c.cc = 0;         {          p1, constant("0", T_INT)), 0, 0).
                  {

        jump(lab+1);
        deflabel(lab);

tree for          {  walk(node(O_ASGN, T_INT,
c.cc = 1;         {          p1, constant("1", T_INT)), 0, 0);

        deflabel(lab+1);

tree for          {  r = walk(rvalue(p1), 0, 0);
rvalue of c.cc    {

        break;
```

#3 Temporaries

- allocate fixed number of temporaries in <u>local portion</u> of frame each large enough to hold largest datatype.

- in ralloc, when out of registers "spill" least-recently-used register into a temporary.

- exprcode returns an index into an array of "resources", which can be registers or temporaries. Level of indirection is necessary to reflect state changes due to spilling.

- emit checks for temporaries, and reloads them into registers (not necessarily the same ones!) if necessary.

Temporaries, cont'd

Example: $a = b + c$ w/ 3 registers

```
        movei   6,a       ; a's lvalue
        movei   7,b       ; b's lvalue
        move    8,(7)     ; b's rvalue
        movei   7,c       ; c's lvalue
spill → movem   6,1(14)   ; spill reg. 6
        move    6,(7)     ; c's rvalue
        add     8,6       ; b+c
reload → move   6,1(14)   ; reload a's lvalue
        movem   8,(6)     ; a = b+c
```

Implementation

```
int rtime [16];          /* LRU time        */
int rcount[16];          /* reference count */
int timer = 0;           /* the clock       */

struct resource {
        int count;       /* reference count */
        int reg;         /* register # or   */
        int temp;        /* temporary #     */
} resources[RESOURCES];
```

```
/* ralloc - allocate a register, return resource */
int ralloc()
{
        int  r, r1, min;

        min = MAXINT;
        for (r = 6; r <= 12 ; r++)
              if (rcount [r] == 0)
                    break;
              else if (rtime [r] < min) {
                    r1 = r;
                    min = rtime[r];
                    }
     if(r > 12)
           rspill( r = r1);
     rcount[r] ++;
     rtime[r] = ++timer;
     r1 = index of free resource;
     resources[r1]. count = 1;
     resources[r1]. reg = r;
     resources[r1]. temp = 0;
     return (r1);
}
```

```
/* emit - emit an instruction */
emit(op, r, name, off, idx)
int op, r, off, idx;
char *name;
{
                              resource
                              index
register     { r   = reload(r) ;
numbers  →   { idx = reload (idx) ;

        if (off is a temporary) {
            idx = 14;
            off = proper frame offset;
            }

        :

}
```

- reload ~~may~~ calls ralloc to allocate a register, which can cause a spill.

Spilling

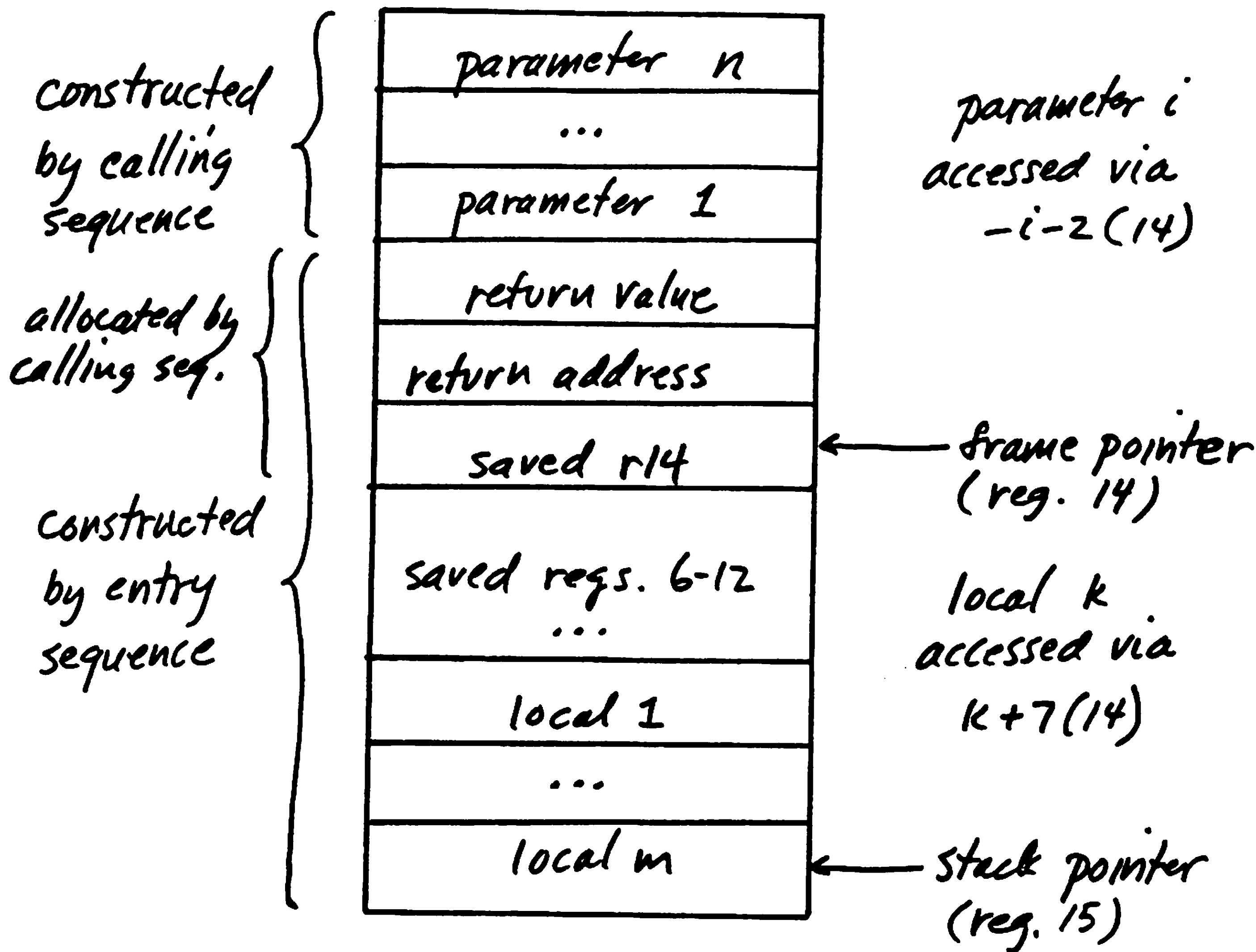
```
int temps[11];     /* non-zero if busy */
/* rspill - spill register r */
rspill(r)
int r;
{
        int  r1, t;

        t  =  index (1..10) of free temporary;
        temps[t]++;

        emit(MOVEM, r, NULL, toff+t, 14);
                              ↑
                    offset-1 to 1st temp.

      for (r1 = 0; r1 < RESOURCES; r1++)
           if(resources[r1].reg == r) {
                resources[r1].temp = t;
                resources[r1].reg = 0;
                rcount[r]--;
                }
}
```

Reloading

```
/* reload - reload resource r, if necessary */
/* returns register number */

int reload(r)
int r;
{
        int r1, r2, t;

        if (t = resources[r].temp) {
                r1 = ralloc();
                r = resources[r1].reg;
                emit(MOVE, r1, NULL, toff+t, 14);
                for (r2 = 0; r2 < RESOURCES; r2++)
                        if(resources[r2].temp == t) {
                                resources[r2].temp = 0;
                                resources[r2].reg = r;
                                rcount[r]++;
                                }
                rfree(r1);
                temps[t] = 0;
                }
        else    r = resources[r].reg;
        rtime[r] = ++timer;
        return (r)
}
```

#41

Frame Construction



- frame construction coordinated among calling, entry, exit sequence to give "best" code.
- must allocate stack space before using it! (nested calls, interrupts!).
- best to allocate/deallocate in same sequence

calling sequence: (for n arguments)

```
        adjsp   15, n+3       ; allocate space
          ⋮
        movem   r, -i-2(15)   ; argument i
          ⋮
        movei   2, $L         ; get return address
        jrst    f             ; jump to procedure
$L:     move    r, -2(15)     ; get return value
        adjsp   15, -n-3      ; deallocate space
```

could use

```
        adjsp   15, 1
        movem   r, 0(15)
```

for each argument, but code is longer.

entry sequence: (m words of locals)

```
f:      movem   2, -1(15)     ; save return address
        movem   14, 0(15)     ; save frame ptr
        move    14, 15        ; set new frame ptr
        adjsp   15, m+7       ; allocate local space
        movem   6, 1(14)      ; save reg. 6
          ⋮
        movem   12, 7(14)     ; save reg. 12
```

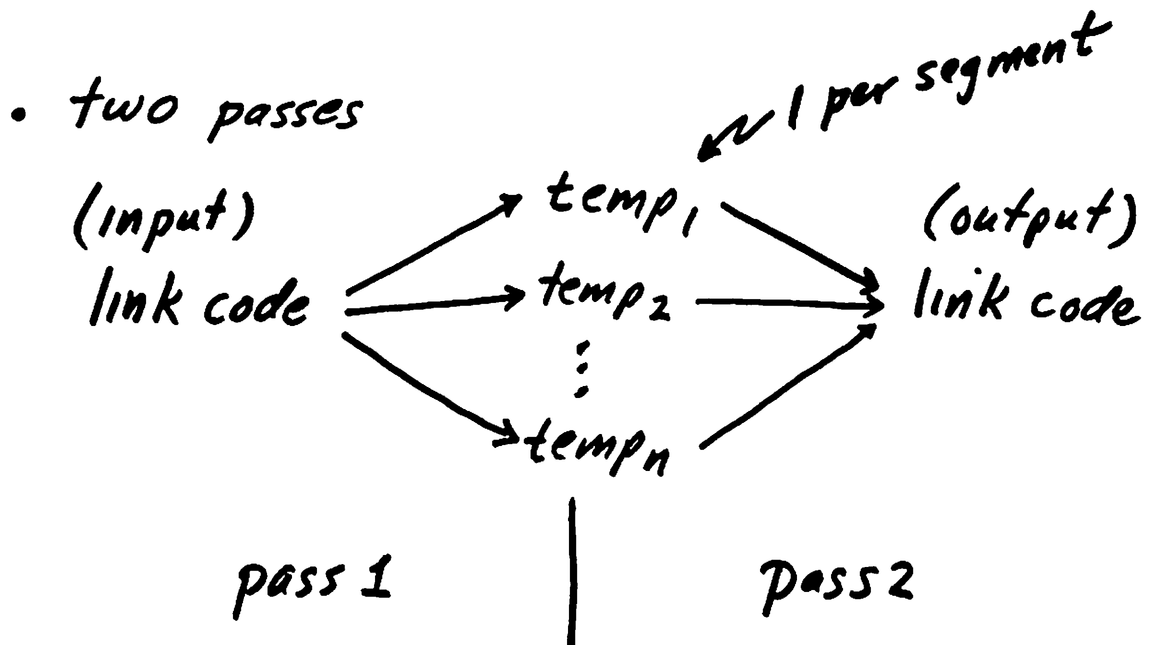
<u>exit</u> sequence: (return value in r)

```
movem  r,-2(14)   ; set return value
move   6,1(14)    ; restore reg. 6
  :
move   12,7(14)   ; restore reg. 12
move   15,14      ; deallocate locals
move   14,0(15)   ; restore frame ptr.
move   2,-1(15)   ; get return address
jrst   (2)        ; return to caller
```

- note entry/exit sequence is <u>independent</u> of number of actual parameters, and that parameter offsets are independent of number of actual parameters.
  This makes writing procedures w/varying number of arguments easier.

Implementation

- two passes



- pass one:
  - distributes text into temporary files, one per segment
  - build symbol table representing sets R, D, S
  - evaluate definitions
  - compute segment lengths
- pass two:
  - combines temporary files into output
  - evaluates expressions in object text (as far as possible).

Modules

*	link.h	definitions
*	lsym.c	symbol table, storage alloc.
	lmain.c	main program
	lpass1.c	pass one
	lpass2.c	pass two
	lexpr.c	expression evaluation
	llib.c	library searching (later!)

* - provided

- psuedo-code for pass 1 & 2 given in paper.
- major task: getting symbolic evaluation & relocation correct.

Symbol Table (lsym.c)

```
struct symbol {
        char *s_name;      /* entry name */
        int s_flags;       /* flags, see below */
        char *s_value      /* value (an expr) */
       struct symbol *s_link;
 }

/* flags */

#define   DEF   001    /* symbol is defined */
#define   REF   002    /* symbol is referenced */
#define   SEGMENT 004  /* symbol is a seg. name */
#define   SUPPRESS 010 /* symbol has -s on */
```

- idea: treat segment name like other symbols during evaluation
- routines: struct symbol *lookup (s): looks up s, installing it if necessary. also slookup, dump, foreach, alloc, initsym (see lsym.c)

Main Program (suggested organization)

```
main(argc, argv)
int argc;
char *argv[];
{
        initsym();
        init1();
        for(each argument)
            if(argument is an option)
                process option;
            else
                pass1(argument);
        if (no file arguments)
            pass1(standard input);
        end1();
        init2(output file);
        pass2(output file);
        end2(output file);
}
```

-u - uflag++
-o ofile is next args

- init1, pass1, end1 in lpass1.c
- init2, pass2, end2 in lpass2.c

Segment Information

* side effect of pass one is a segment table:

```
struct segment {
    FILE *x_fp;        /* temp file pointer */
    char *x_fname;     /* temp file name */
    int   x_length;    /* length in current file */
    int   x_base;      /* accumulated length */
    struct symbol *x_p /* symbol table
                          entry */
```

√value of segm

x_fname : construct using segment name, e.g. "name.seg"; unlink (delete) after pass 2 (unless -d).

bonus! →

name

x_length, x_base : set by .len commands; used as "value" of segment name symbol.

test for segment or
maintain value

Pass One

- distribute text into segment temporary files.
- evaluate definitions
- build R, S, D (i.e. build symbol table).
- compute segment lengths.
- implement relocation (i.e. adjust segment base locations).

init1 - init segment table
pass1 read object opened mf
exec pass1 operator
end1 - close temp files

affect of .len cmds must be indep of location

```
pass1(f)
FILE *f;
{
        for(each segment p)
            p->x_length = 0;

        curseg = NULL;  /* "current" segment */
        while (fgets(line, MAXLINE, f) != NULL)
            if (line is a command)
                ...
            else {
                add symbols in line to
                    symbol table;
                fputs(line, curseg->x_fp);
            }

        for(each segment p) {
            p->x_base += p->x_length;
            for(each segment p1)
                fprintf(p->x_fp,
                    ".len %s %d\n",
                    p1->x_p->s_name,
                    p1->x_length);
        }
}
```

Margin notes:

symbols start w/ a letter ., digit, etc.

just for -u? builds sd R

eval enters symbols in symbol table thus, evaluate

p = segments; p < segments + nsegs; p++

make segname name base +

adjust segment base location

write .len commands to temp file.

all temp files have len cmds for all segments

Pass One, cont'd

```
if (line is ".seg id") {
    p = lookup "id" in segment table;
    if("id" is a new segment) {
        p->x_fname = "id.seg";
        p->x_fp = fopen(p->x_fname, "w");
        if(p->x_fp == NULL)
            issue fatal error & exit

        p->x_base = p->x_length = 0;

        p->x_p = lookup("id");

        enter p into segment table;

        fprint(p->x_fp, ".seg %s\n",
            p->x_p->s_name);
    }
    curseg = p;
}
```

Pass One, cont'd

```
else if (line is ".len id n") {
    p = lookup "id" in segment table;
    if("id" is a new segment) {
            ... (same as in .seg command)
        }
    p->x_length += n;
}
```

.len commands can precede .seg commands

- len commands can occur anywhere (and repeatedly), but their effect is deferred until the end of pass 1 when x_base is incremented by the sum of the n's in len commands.

Pass One, cont'd

```
elseif (line is ".def [-s] id expr") {
    sp = lookup("id");
    if (sp->s_flags & DEF)
        fprintf(stderr, "multiply defined
                 symbol: %s\n", sp->s_name);
    sp->s_value = evaluate("expr");
    sp->s_flags |= DEF;
    if (-s specified)
        sp->s_flags |= SUPPRESS;
}
else
    fputs(line, curseg->x_fp);
```

use slookup

to tuck away the strings

as strings

(can also turn on SUPPRESS to avoid output in pass2)

move to 36, don't want output a lif for a seg

.org commands & "comments".

any other dot commands

Pass One: Example (from paper)

- two files, a.o & b.o:

a.o	b.o
.seg code	.seg code
.def start code	.def f code
0263 f +	0201 b_l +
.seg data	.seg data
.def -s a_l data	.def -s b_l data
data 01 +	start
.org 6	.len code 1
code	.len data 1
.len code 1	
.len data 7	

inter-segment reference

link b.o a.o { Note: b.o is linked first }

- after pass 1, symbols are

name	flags	value
f	DEF | REF	"code"
a_l	DEF | SUPPRESS	"data 1 +"
start	DEF | REF	"code 1 +"
b_l	DEF | REF | SUPPRESS	"data"
code	DEF | REF | SEGMENT	—
data	DEF | REF | SEGMENT	—

"code 0 +"

Example, cont'd

- temporary files:

```
code.seg                          data.seg

.seg code                         .seg data
 0201 b_l +  }                  { start
.len code 1  }  b.o             { .len code 01
.len data 1  }                  { .len data 1

 0263 f +    }                  { data 01 +
.len code 1  }  a.o             { .org 6
.len data 7  }                  { code
                                { .len code 1
                                { .len data 7
```

- Pass two combines code.seg & data.seg into single output file, evaluating expressions in object text.

Expression Evaluation

- problem: must avoid "double relocation" while still permitting "active substitution" of symbol values.

Example: in pass two, will evaluate

0263 f +

substituting value of "f" yields

rescan substituted value is active sub, other is passive

0263 code +

but value of "code" must not be substituted because appropriate relocation of segment names in definitions is done in pass one.

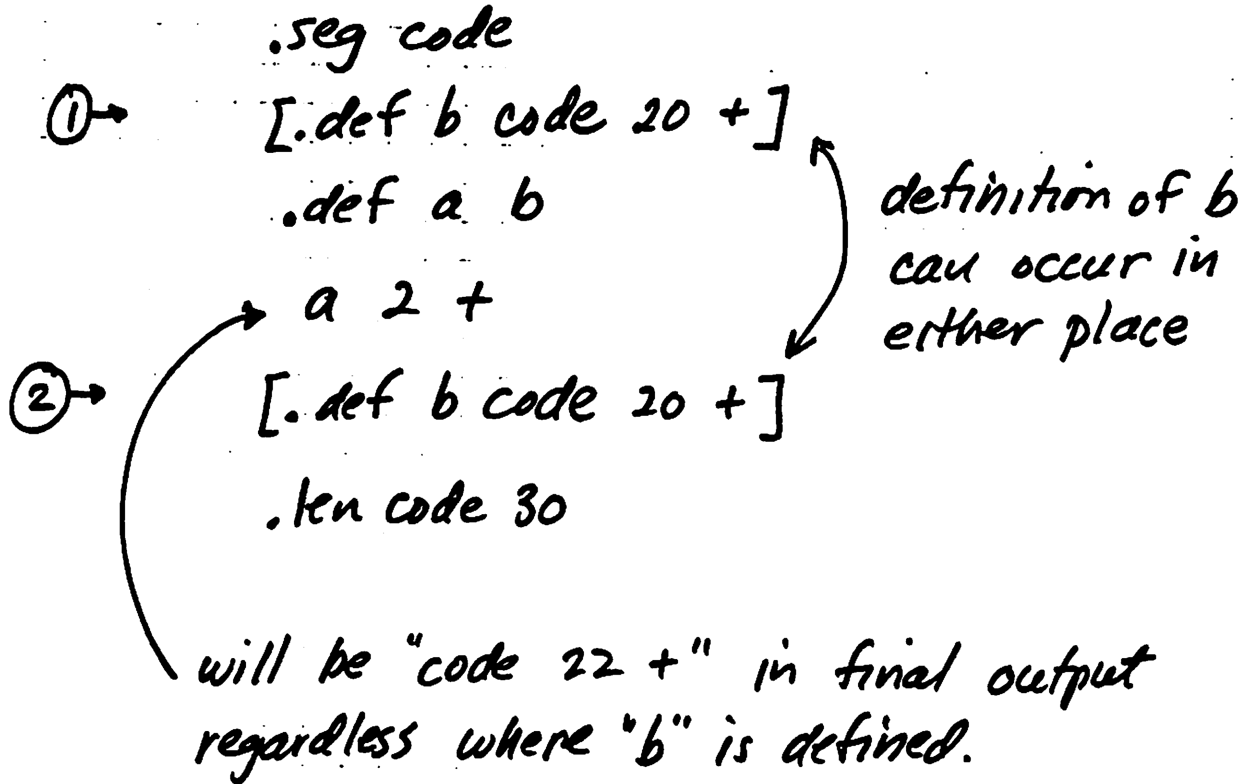
On the other hand, in expression

code

pass two must evaluate code to obtain proper relocation in object text:

code 01 +

<u>another example:</u>



- in position ①, a is defined as "code 20 +" and "a 2 +" is evaluated as

  a 2 + → code 20 + 2 +

- in position ②, a is defined as "b" and "a 2 +" is evaluated as

  a 2 + → b 2 + → code 20 + 2 +

ok, don't get

Possible Solutions

- special case segment names so their "evaluation" depends on context (i.e. definition in pass 1, object text evaluation in pass 2)

- add syntax to expressions (for use only within link) that suppresses substitution of value, e.g.

 .def f code

 evaluation of "code" yields "'code n +" where "n" is x_base field for segment "code".

 'name suppresses further evaluation & is removed when definitions are printed in pass 2.

- expression evaluation must recognize 'name and passes it through (but should turn on REF bit?).

code

Pass Two

- output final segment lengths
- output definitions (unless suppressed)
- combine segment temporary files into output
- evaluate expressions in object text

Pass 2 Routines:

- init2(f) - output segment lengths and symbol definitions to f
- pass2(f) - combine temporary files into output file f
- end2() - remove temporary files (unless -d).

Pass Two, cont'd

```
init2(f)
FILE *f;
{
        for(each segment p)
            fprintf(f, ".len %s %d\n",
                p->x_p->s_name,
                p->x_base);
        foreach(putdef, DEF, f);
}
```

- x_base field has final length for each segment
- putdef outputs definition if s_flags & (SEGMENT|SUPPRESS) == 0.

```
pass2(f)
FILE *f
{
        FILE *fp;
        for (each segment p) {
            fp = fopen(p->x_fname, "r");
            if (fp == NULL)
                issue fatal error & exit;
            for (each segment p1)
                p1->x_base = 0;
            while (fgets(line, MAXLINE, fp))
                if (line is a command)
                    ...
                else {
                    s = evaluate(line);
                 -> output s to f;
                }  \ strip \'s from line
            fclose(fp)
        }
}
```

.len commands in temp files cause x_base to be incremented

Pass Two cont'd

```
if (line is ".len id n") {
    p1 = lookup "id" in segment table;
    p1-> x_base += n;
    }
else if (line is ".org n") {
    n += p -> x_base;
    fprintf(f, ".org %d \n", n);
    }
else
    fputs(lin, f);
```

Example

- temporary files for b.o & a.o input:

code.seg

```
.seg code
 0201 b_l +
.len code 1
.len data 1
 0263 f +
.len code 1
.len data 7
```

data.seg

```
.seg data
 start
.len code 1
.len data 1
 data 01 +
.org 6
 code
.len code 1
.len data 7
```

- output:

```
.len code 2
.len data 8
.def f code
.def start code 01 +
.seg code
 0201 data +
 0263 code +
.seg data
 code 01 +
 data 02 +
.org 7
```

Library Seaching

- link object modules based on current value of $R - (D \cup S)$, i.e. symbols referenced but not defined; sort of a "data-driven include mechanisim".

- libraries are searched during pass one:

  link a.o b.o -l subs.lib

  performs pass1 on relevant object files in "subs.lib".

- more than one library can be searched, and same library may be searched more than once:

  link a.o -l lib1 b.o -l lib1 -l lib2

- function of library searching is simple: "if library member defines a symbol in $R - (D \cup S)$, link that member";

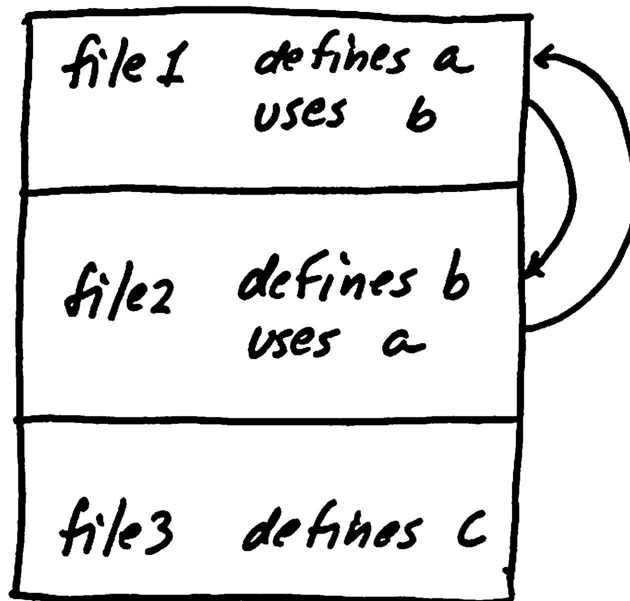
- form of library affects search

Library Representations

- simple concatenation of object files (e.g. DEC-10):

```
for each object file in library
    if this file defines an undefined symbol
        link this file
```

- object files must be concatenated in topological order:



note: library searching affects R, D, & S!

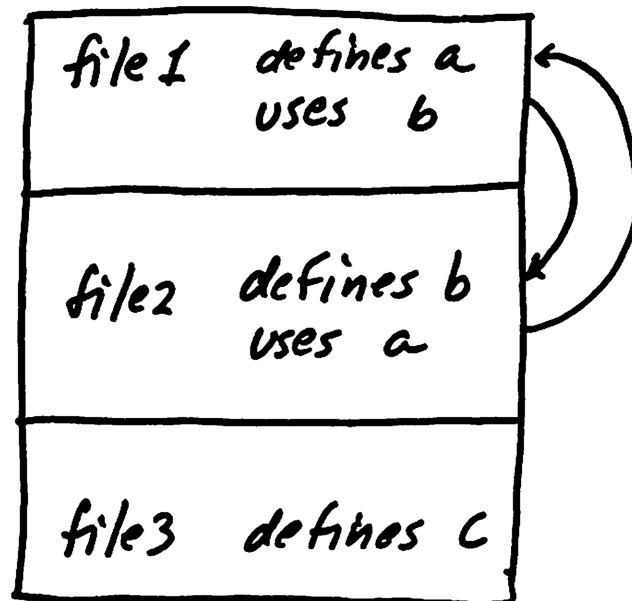
- if a is undefined, both file1 & file2 are linked; if only b is undefined, file2 is linked and a remains undefined.

Library Representations

- simple concatenation of object files (e.g. DEC-10):

  for each object file in library
      if this file defines an undefined symbol
          link this file

- object files must be concatenated in <u>topological order</u>:



note: library searching affects R, D, & S!

- if <u>a</u> is undefined, both file1 & file2 are linked; if only <u>b</u> is undefined, file2 is linked and <u>a</u> remains undefined.

Library Representations, cont'd

- simple concatenation may require several passes to get everything: at worst use

```
for each undefined symbol
    for each object file in library
        if this file defines symbol
            link it
```

- topologically sorted concatenation of object files can be searched in one pass.

- better organization: concatenation of object files preceded by index or directory of (symbol, file) pairs for all symbols defined in the library:

```
for each undefined symbol
    if symbol defined in library
        link corresponding file
```

- can use either random access or multiple passes over the library in search

Computer Science 453
Fall 1983

Class Notes, Set 6

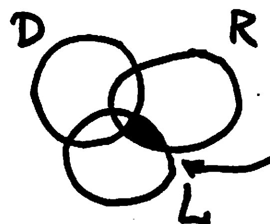
November 1, 1983

Implementation

- library contains <u>entries</u>, each of which is an object file that contains symbol definitions.

- let L = set of symbols defined by all entries in the library.

- want to load those entries that define symbols in $(L \cap R) - (D \cup S)$ [assume "$\cup S$" below].



- algorithm:

$(L \cap R) - D$

↑ undef syms in library

```
open library
load index, i.e. construct L
while ((L ∩ R) - D is non-empty)
    for(each library entry q)
        if(q defines a symbol in (L ∩ R)-D)
            pass1(q)
expunge symbols in L - (R ∪ D)
reclaim space
```

may involve random-access or multiple passes. (the while loop)

↑ symbols only in L (i.e. $L - (R \cup D)$)

Implementation, cont'd

- additional symbol table information: (see lsym.c)

```
struct symbol {
    char *s_name;
    int s_flags;
    char *s_value;
    struct entry *s_entry;
        /* library entry structure */
    struct symbol *s_link;
};

#define LIB 020   /* symbol defined
                     in a library */
```

↑ member of L

- library entry information (see link.h)

```
struct entry {
    char *e_name;   /* entry name */
    long e_pos;     /* location in library */
    int e_size;     /* length of entry */
    int e_count;    /* # symbols in (L∩R)-D */
    struct entry *e_link;  /* next entry */
};
```

? ⟶ (e_count)

Library Format

- archive files:

  header
  entry
  header
  entry
  ...

- first entry is "index":

  #-h- index size date information
  # symbol name
  ...

  (annotation, arrow to "size": length of entry in chars.)
  (annotation, arrow to "name": entry name where "symbol" is defined)

- remaining entries are object files.
- no position information; must be computed while reading library.

-1 800
call 13(800) ?

```
lib(name)
char *name;
{
        FILE *f;

        f = fopen(name, "r");
        if (f == NULL) {
                fprintf(stderr, "can't search %s\n",
                                                name);
                return;
                }
        getindex(f);    /* construct index structure */
        search(f);
        fclose(f);
        expunge();
        free index structure;
}
```

Processing the Index

```
getindex(f)
FILE *f;
{
        struct symbol *p;
        struct entry *q;
        for (each line in index) {
            q = pointer to entry structure for "name"
            if (q is a new entry) {
                fill in q's fields;
                add q to list of entries
                }
            p = lookup("symbol");
            p->s_flags |= LIB;
            p->s_entry = q;
        }
}
```

2nd field

- end up with all symbols in symbol table, and with them LIB set and s_entry pointing to entry structure, and a list of entry structures.

Searching the Library

- two phase search:

  1) read library sequentially, linking necessary modules, and computing e_pos & e_size for second phase.

  2) read library randomly, as necessary to link remaining required modules.

- "linking" an entry involves using pass1 to read the entry.

- linking an entry <u>may change</u> $(L \cap R) - D$; must recompute that set after calling pass1. (hint: use q->e_count field).

```
search(f)
FILE *f;
{
        struct entry *q;
        /* phase one */
        for(each entry ) {
            q = pointer to corresponding
                entry structure;
            q -> e_size = size from header;
            q -> e_pos  = ftell(f);
            if(q defines a symbol in (L∩R)-D)
                pass1 (this entry);
            else
                skip this entry;
        }
        /* phase two */
        while ((L∩R)-D is non-empty)
            for(each entry q in list of entries)
                if(q defines a symbol in (L∩R)-D){
                    fseek(f, q->e_pos, 0);
                    pass1 ( this entry);
                }
}
```

While there are symbols in the library that are refered but not defined

Referenced, in lib

"If I get wrong, I'm going to tell you. Now I'm going to ruin your life this term"

8/23 T ①

Intro DEC-10 - 27, 30, 31

STUM

Don't miss a class

1200 lines

OH TH - 2:00

Mail on 10 [1055, 21)

HW - ½ / ½ 3 Exams - Equal weight

A Bare machine

~~applicatn~~

"bare" machine

operating system

system calls, { i/o services 452
{ process

basic system software 453
e.g. compilers, linkers, editors
perf. monitoring, etc., i.e. essential tools

"application" software
e.g. software tools editors,
DBMS, etc.

②

8/23T

Essential Sys Soft.

· in presence of virtual machine presented by the hardware + operating system calls

· essential tools are those necessary to construct other tools, i.e.

· Editors

· Compilers — Doubles up as assembler (if not small, too much assembly language)

: Linkers ← Combining

· Debuggers

· Monitors - Understand prog behavior + cost

· command lang. interpreters

[Note absence of assembler?] Don't need them!

Impossible to write an incorrect program → No error ~~correction~~ detection/hardly needed in compiler

Ideal - editor interface to [illegible] edit as ~~[illegible]~~ ~~[illegible]~~ computing interface

8/23 T

Compiler

· for ~~systems prog~~ lan.

· must ~~produce~~ reasonable be relatively simple; more importantly must produce correct code.

For CS 453

· Write a real compiler for subset of C in (full) C on the DEC-10.

· 1-pass, emits link object code

· machine independent parser ("top half") machine dependent code gen.

· classical design

Linkers

· machine-independent link editor

· links textual object lang.

— ~~[illegible]~~
— mach. content
— ~~[illegible]~~ uni form

· permits "iterative links" i.e. output can be used as input

· ~~includes library searching~~

8/23 T

- requires symbols, and loader
- includes library searching (libraries are archive files)
- replace traditional use of an assembler

Debuggers

- low level debugger — at the instruction level
- used to build higher-level debugger i.e. at the C statement level
- features
  - edition style interface
  - breakpoints
  - single step
  - procedure trace
- requires compiler mods
- editor interface — like editing a running program
- programmer defined debugging functions (print lists etc.)

$1250

⑤

8/23 T

Monday

- stmt level perf. measurement
- execution frequency & stmts & procedures
- timing histo; sampling techniq
- data presentation, tools, program listings, call graphs.

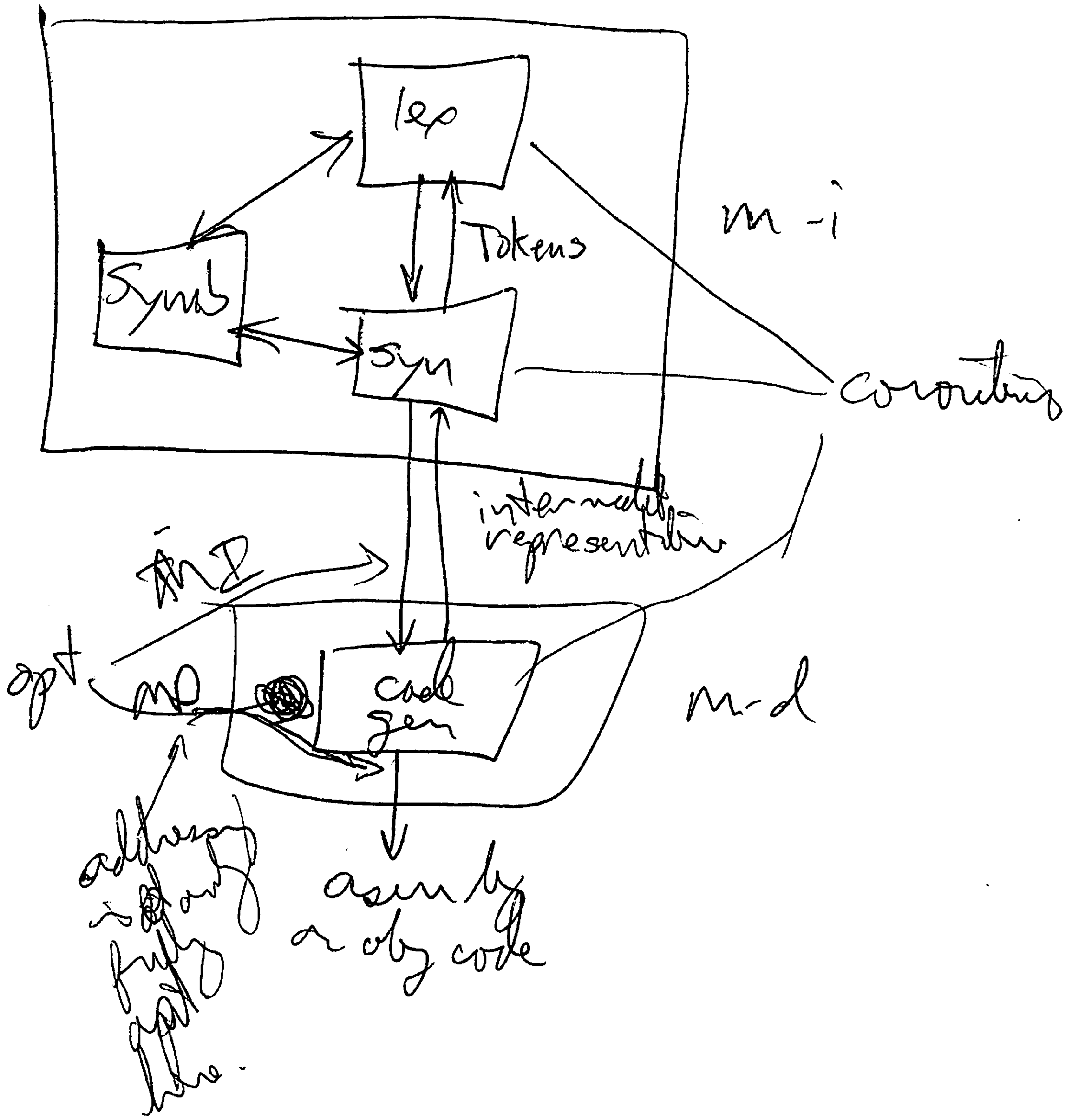
---

command langs

- Unix Shell; shell program
- display shells, point + execute
- interactive & batch command
- language design/binding issues/ data types issues
- unifying prog & command languages; EZ & related systems

8/23

Compilers Construction Design



~~we might be able~~ to tell how to go small, but all you'll be able to get is Ridge."

(1983)

8/25 H

Notes in quick copy center

switch.ini [1055,21]

Lex Analysis

pairs of values (type, value)

constants: con, "value"

↑ pointer into symbol table

"Why does the chalk always walk behind the screen?"

Returns tokens, sets global var for value

must handle errors

Trees will be built before it is emitted

IR:
quads
triples
~~postfix~~
prefix
trees

Code Gen

- read IR (abs. syn. trees), misc direct.
- manage machine resources
- assign machine depend. params e.g. frame layout
- emit output

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

Most Com., Closer spend a lot of time in parsing — parsing is "somewhings" (8/25H)

Symbol Table

- maintain semantic information on identifiers, const, e.g. type, address, size, etc.

- maintain mult. instances of same identifier (local, global)

- maintain single-copy of all strings
  - keep all unique string in input
  - permits fast string comparison
  - permits consolidated poly

Grammar: precise specification of syntax

language: set (possibly ∞) of sentences of terminal symbols

Top Down Parsing

- using the next input symbol and current derivation to properly "guess" the next derivation step

- implement using a rec. proc for each NT

Pit falls

1. Left recursion
2. Backtrack
3. ambiguity $E \to E + E \mid E * E$
4. Error Handling

8/25/?

$A \to A\alpha \mid \beta$

$A \to \beta A'$ ↑ β followed by 0 or more α's

$A' \to \alpha A' \mid \epsilon$

* first(α) = terminals that begin sentences derived from α

Thm: first(α) $\cap$ first(β) = $\emptyset$

Comp::

1. if x is a terminal, first(x) = {x}
2. if $X \to a\alpha$ when a is a terminal add a to first(X)
3. if $X \to \epsilon$, add ϵ to first(X)
4. if $X \to Y_1 Y_2 \cdots Y_{i-1} Y_i \cdots Y$

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

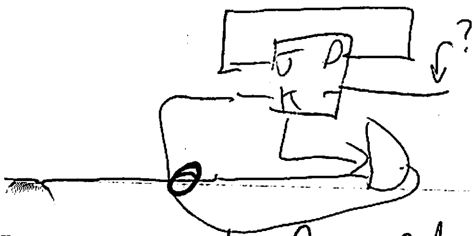
[illegible]

[illegible]

[illegible]

[illegible]

[illegible]



8/30 F

* Looks the same in decl as in use.

struct sym *p → type of *p is sym

(*p).s_name ≡ p→s_name

p→x (*p).x

Follow(A) = terminals that can follow NTA

"Don't panic, the date in this class hasn't been stuck" (due)

"Well, let's not worry about it until we figure it out"

struct symbol *(*f)();

ptr (of fcn?) to fcn that returns a ptr to a symbol

*(*f)();

pointer to (*f)();
pointer to function ret (*f)

ptr to fcn return a ptr to a symbol

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

9/14

New cc.h csym.h

O_ADDR - removed

(struct sym *)p

char *alloc();

p = (struct symbol *) alloc (sizeof(struct sym))

symbol pointer — typecast — char point — #bytes

convert to type.

"A type conversion fcn."

"Have you tried it? It works for me."

~~Don't~~

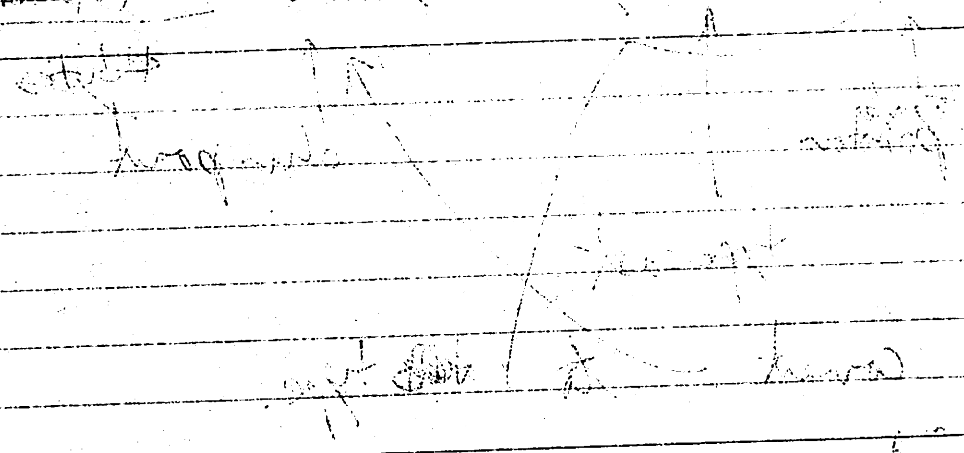
Source lang ops w/ most

global decls & funcs first,

typed funcs later

refs are ~~invokes~~ ~~...~~ globals static records

can read listings file to get global & static vars



9/21

"We need to order some chalk that doesn't have feet"

Sizes:
int a -> 36
char s[6] -> 54
not used for functions

**p is int
*p is pointer to int
p is pointer to ptr to int

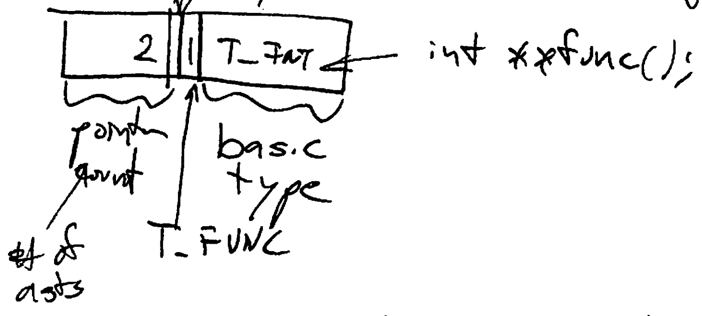
foreach - csym.c loops through symbol table. also see dump

9/6T

install

```
int f(a,b)
int a;    } lookup  T_ADDR - T_LVAL  forms an lvalue
float x;  }
{
```

← if any params undefined



MKTYPE(2, T_INT) | T_FUNC

Array is a pointer type

Size field not used for params

Size field for array param is ptr size

int *x[10]; - 10 * PTR_SIZE

char *s[7] - 7 * PTR_SIZE

REF flag - symbol is referenced

9/6T

Must stack loop heads but recursion should handle it.

Add locals any trouble because adds locals change stack frame size

[illegible]

Must stack loop heads, first
so memory should handle it

[illegible]
[illegible]
[illegible]

9/8H

x[i]
*(x+i)

*f() += a;

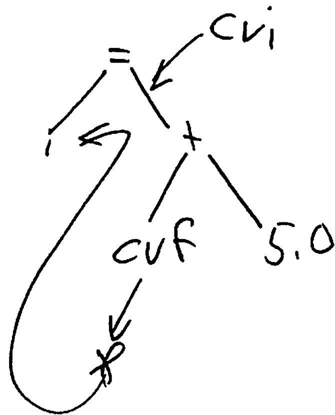
e1 op= e2

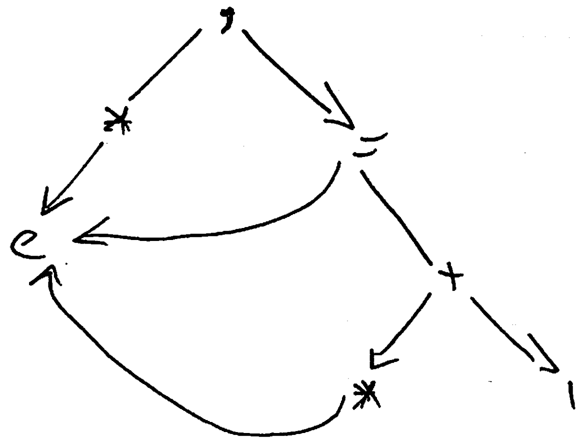
return e_result field rather than walking the tree

enode does type checking + conversion of lvals to rvals

++e ⟷ e+=1
↑
lvalue

i += 5.0





9/8/4

x[i] = a[i]

*(x+i)

*f() += a;

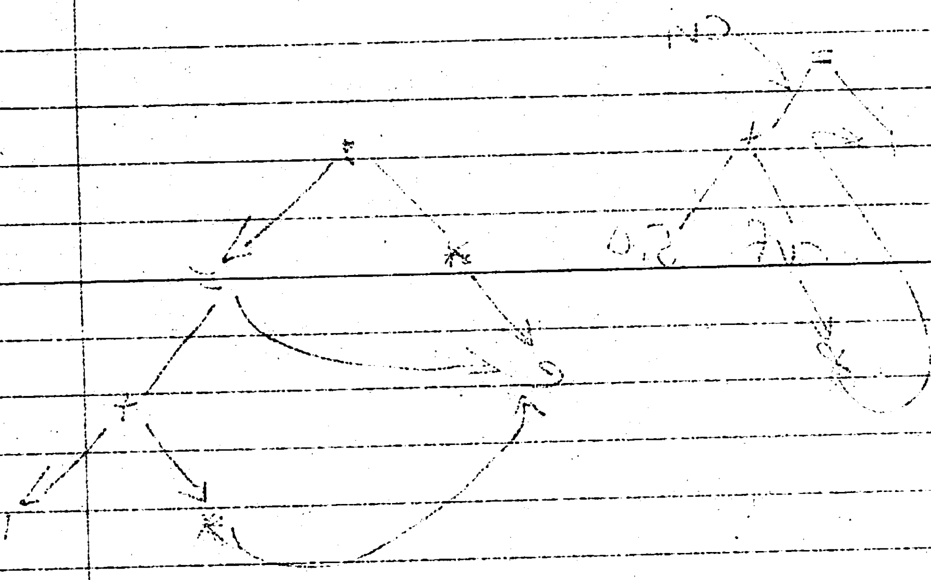
e1 op= e2

evaluation: result [illegible] when walking the tree

[illegible]

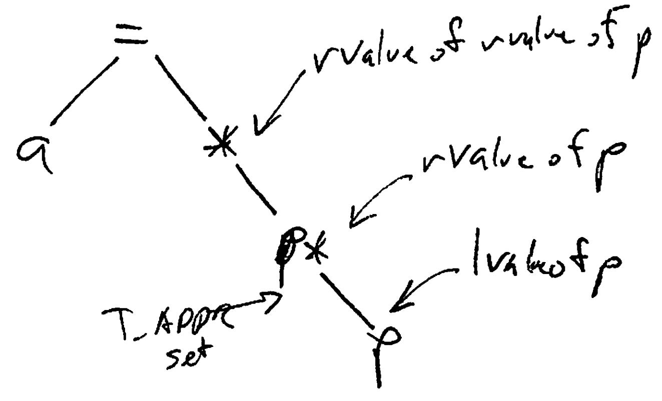
1+2 <=> 2+1

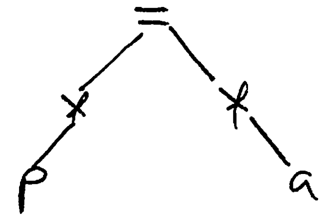
1 += 5+6



9/13T

left to right or right to left
eval often depends on
dir. of stack growth

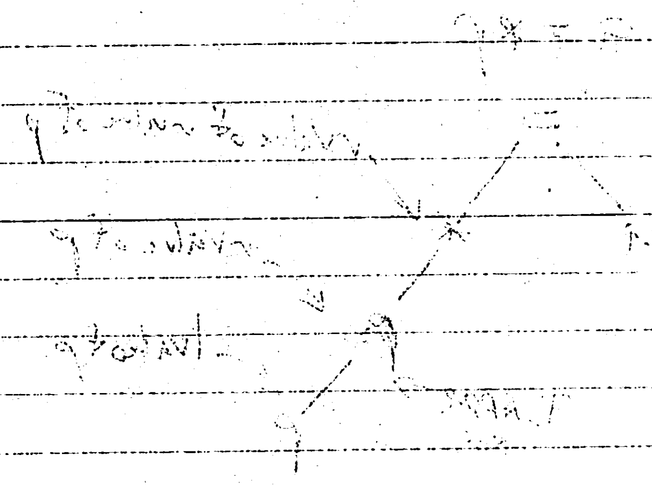


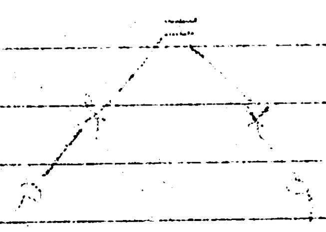


~~wants rvalue which~~ rb

See 63

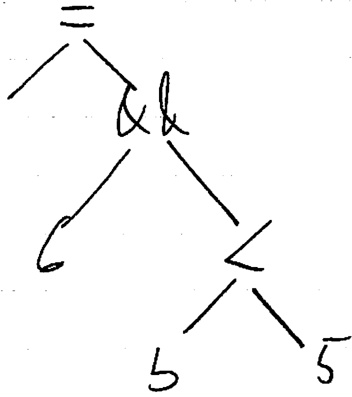
lvalue(-) - sole value is to check O_APPR



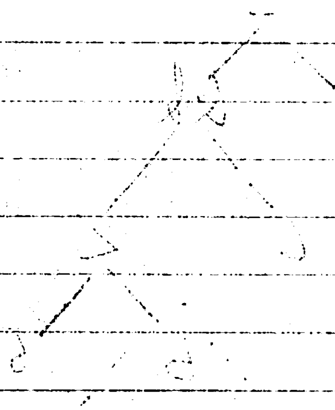


9/13T

$a = 6 \&\& b < 5$



- assignment of cond. values



x > y   2>&1   x >& y

9/15 H

50% day for 3 days

Tues at midnight

Three files named

#include "ccl.h[1.055, 0471]"

"This is probably the hardest assignment"

Start codegen on Tuesday

Augmented assignment

p = &q

cost

no runtime representation

"If handles test4 - only error handling."

Counts [illegible] on [illegible] stuff, not error handling.

If handles all programs w/o free nodes, don't need freeing of nodes

"All of the biggies will be here - drinking - that's the major occ. at these thgs"

"I don't go to talks, I just go to the bar."

FOCS

9/20 T

REF - a bit to tell if the id was refered to.
REF on + DEF off - undefined

Ref a bad address

Ref through 0 → 17₈ → Boom (SP)

a < 1; → Value expected

f(a < 1);

Not in test1-6.c - won't see it.

---

our code - 2-3 times larger than optimal

"Nothing worse than fast code that doesn't work"

op arg1, arg2 — a two address mode

add + store - three address

segment - contiguous region of code or data

a.r. - local data + return information

9/20

Ex: [see opcode.h]

```
move   2, x        ; load x into 2
add    6, 7        ; add reg 7 to reg 6
movem  13, 16(14)  ; store reg 13
addi   6, 40       ; add 40 reg 6
move   6, (7)      ; load 6 w/ word ptd at by 7
```

addi 6, 040(7) — 6 = 7+40

DEC has 50?

S_offset will hold frame offsets

9/22/4

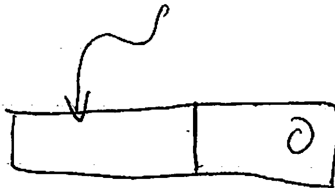
calling[,21] - calling sequences

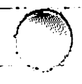
9/27T

"If that's a C.C. cookie, you damn well better give me one!"

```
#undef AND
#undef LSH
#include "opcode.h[1055,21]"
```

movsi 6, left-18



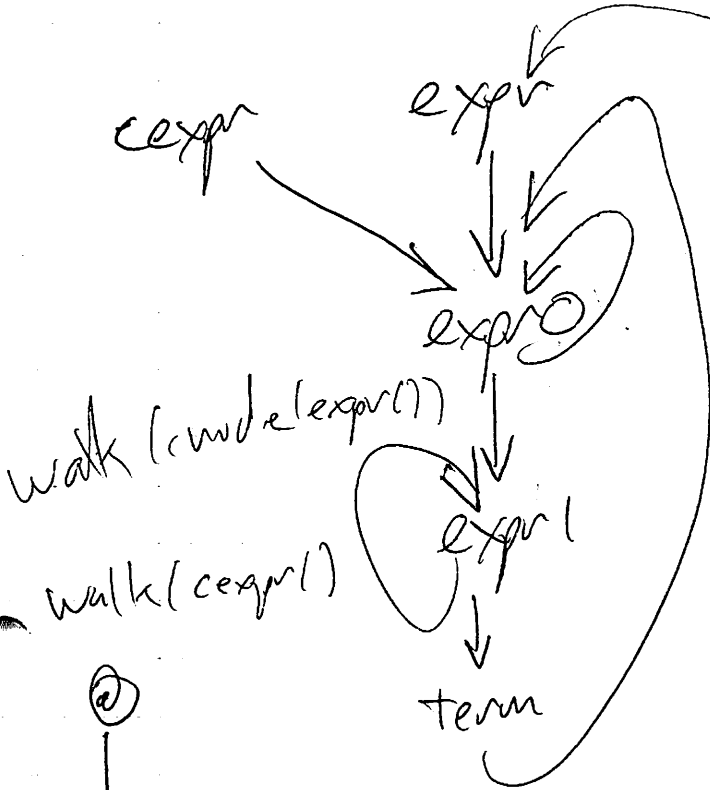


"Will look at [illegible] files in next set of dis" ↗ "I'll pass out air ordering lunges"

9/29

"I'd like to find the SOB at DEC that put this on the machine" (ADJBP)

(expr ↪

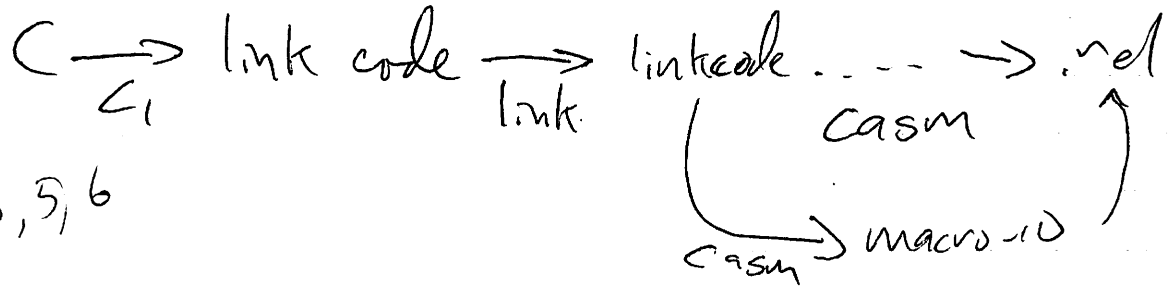


return node(@, T_int, expr(), 0)));

Exam on Tuesday
No q's about reg alloc.

node(@, 0, cnode(expr()), node(), 0

@



.c1 < test2.c | link | casm > test2.mac   Check casm output

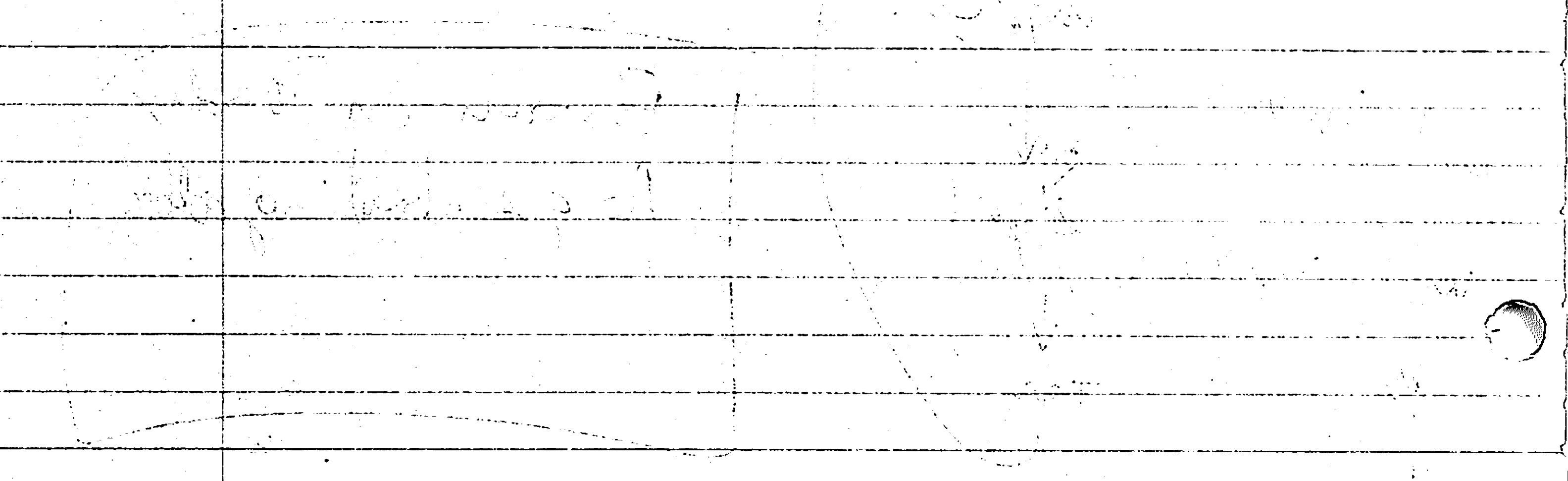
.ccom test2.mac rtlib.rel[,21]

↓ -T 1000

q.exe

```
main()
{
    puts("hello world\n");
}
```

```
puts(s)
char *s;
{
    while(*s)
        putc(*s++);
}
```



Link

10/6 Wt

* How could overlays be added?

"So the overall goal of the other
than to prolog 450 [illegible] is to..."

* "link editor" implies more than "linker"

UNIX: exec code = object code
w/ no unbound addresses

All expressions represent
one cell

move 0100 + 15 +
%x 02 -

0
0206

link assumes each input expression

loader does
9 } 18 bit
6 }

via (9<<18)+6

link replaces symbol by their values

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

3c

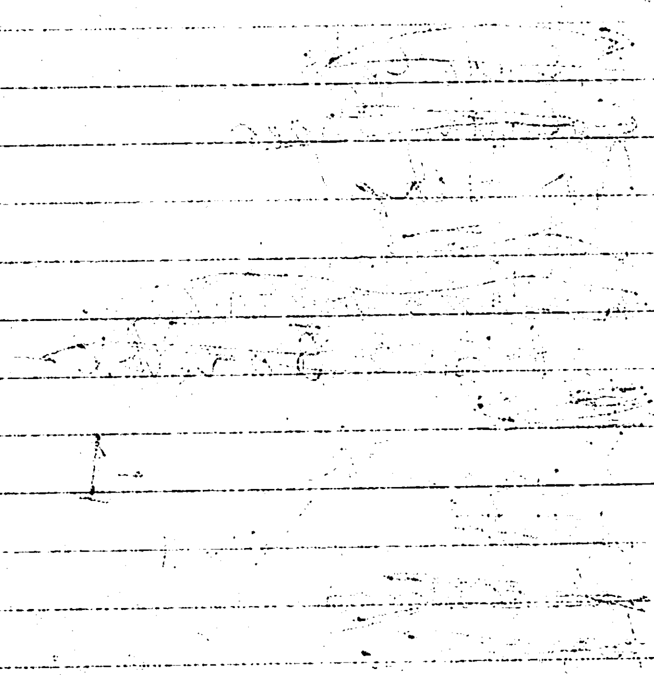
10/11T

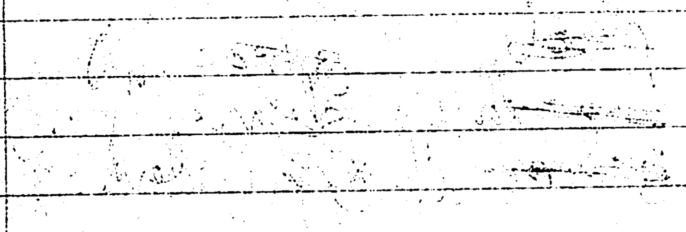
Ex prob graded on 0-75 basis

~~Simple Opts~~
~~Calling Paper~~
VR Heap
~~Dist Layout~~
~~Dbl Word VAX STACKS~~
~~Regs 0-5 on VAX~~
Y arrays TR -1
83-1

~~proc~~ proc (bp)
~~procaddr~~ proc (bp)
char *procident (bp)

c2 -2





No Class 10/20 H

10/13 H

Elevate link for compiler writers, bury link from users point of view

link [file | -u | -o file]

Issue Warning if can't find an input file

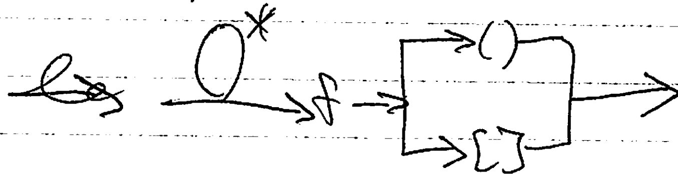
link a -u b -u c -u -o foo

take last output file

Value of segment name is base location of segment

bss - "block started by symbol"

UCF AI
ARCH
550a
550b
473
520
453
900



[illegible]

Budd

"You alright Boy?" Arch to budd 10/28

"For those of you who haven't heard about it, you're probably lucky."

"It was a L have seven line; also used to be used in some of the classes.

Everything is unresolved until pass 2.

~~Phod~~
Always give error for two define w/ same name

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

"~~Comments~~ What are comments?"

"Comments are for chumps"

10/25T

when -u is seen, output undefined symbols

l0[,2] -d b.lc a.lc

-d don't delete temp files for segment

don't worry about names > 6 chars

Array of segment ptrs

Don't worry about evaluation until end

3/2/01 [illegible]

[illegible]

[illegible]

[illegible] 5-20

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

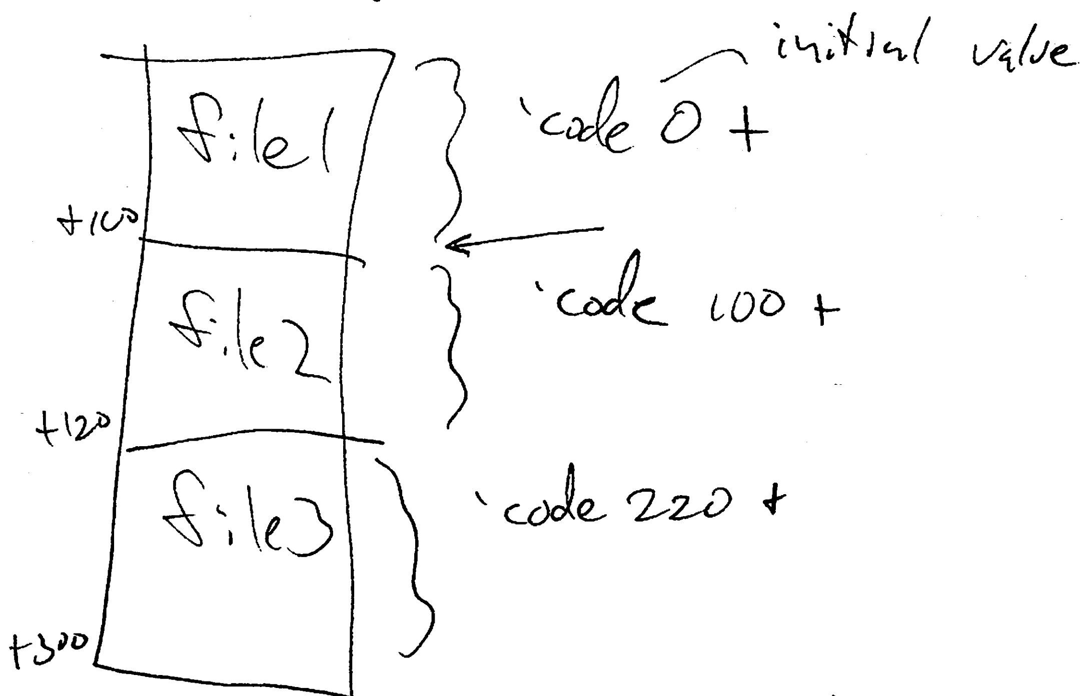
10/27 H

~~a code +~~

.def a b 2 +
.def b code 4 + → b = 'code 4 +
a code 4 + 2 +
b code + 2
'code 4 +

code can't be defined until no files are left.

Can't run flonk after value has been substituted for code



=S - dump symbol table

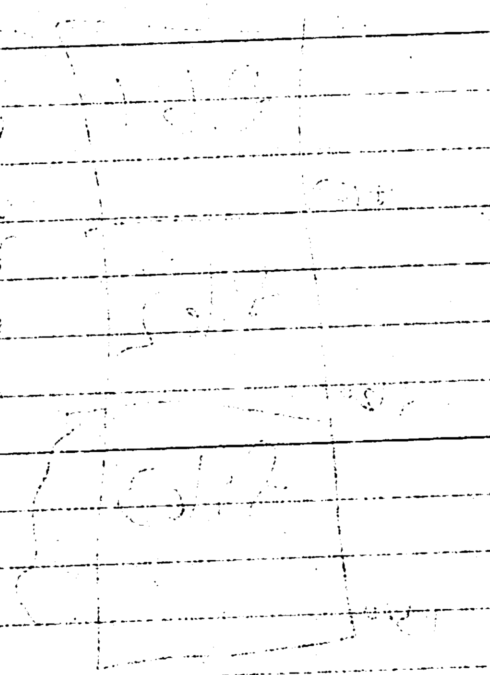
11/27/

initial value

'code 2 +

'code 100 +

'code 220 +



(only) 6/27H

DEF bits on segs for -u
no point to evaluate stuff in pass 1
since no evals in
values of code
'code 0 +
evaluate definitions
pass 1 evals definitions
a 2 + 4 +
Do only text subs, but no arith. evals.
Use octal for output

.def a code
.ses code
1
2
.len code 2

.ses code
a
code
.len code 2

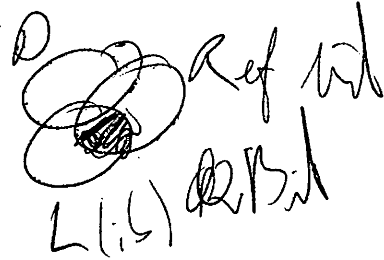
11/17

Bus:

05 2007--6

Library contains entries, each of which is an object file w/ definitions

L = set of symbols defined by all entries in library



want to load $(L \cap R) - (D \cup S)$

while $((L \cap R) - D$ is not empty)
 for (each entry q)
 if (q defines a symbol in $(L \cap R) - D$)
 pass1(q)

needs Rand access or mult passes

expunge symbols in $L - (R \cup D)$

↑ symbols only in L

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

11/17

add:
struct symbol *s_link
to symbol structure

#define LIB 020

```
struct entry {
    char *e_name       loc in lib
    long e_poss        len of entry
    int  e_size        Sum of all cont fields is # of unf symbols
    int  e_cont        # of symbols in (L∧R)-D
    struct entry? e_link
}
```

#-h- index size
symbol name

*{ Test Q: ⓆⓆ
3n: Can things other than object text be in library

VBS on OS is byte stream

"And when we're where we want to know where are..."

Test Q: What can be done to improve the index, size + pos information?

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

11/17

might want to diff indexstruct
after it's built

two phase search, 2nd phase
is only used for non
topologically sorted file

will test plospost how far
to read

Must recompute (L∧R) - D after
each update in pos1

Can't do algebra on #'s returned
by Stel

[illegible]

[illegible]
[illegible]

[illegible]
[illegible]
[illegible]

[illegible]
[illegible]

[illegible]
[illegible]

[illegible]

11/3H

fseek will be there Tuesday

.copy = dskd:[1000,102, cboot] stdio.h
clib.rel
root.rel

.as dsk csc

11-4.lc
.lib - various libraries

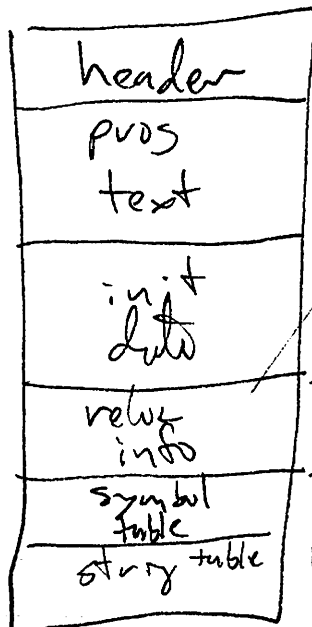
Exam - everything + anything about linking + library searching

No Code Gen. Just link. + load on test.

"The ans. to how it's done in the real world is that it's done wrong."

(print some .h files)

UNIX OBJEcts.



using segment names to cause relocs.
i.e. our seg. names were our reloc info.

11/3

- symbol table indexes into string table

string table is only for identifiers

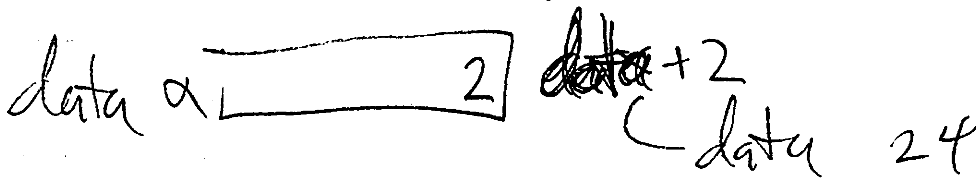
.o + bin files have the same format

If no relocation info, file can be executed

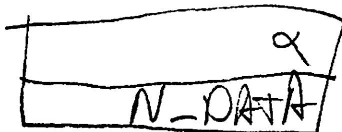
strip removes reloc, symbol, string stuff

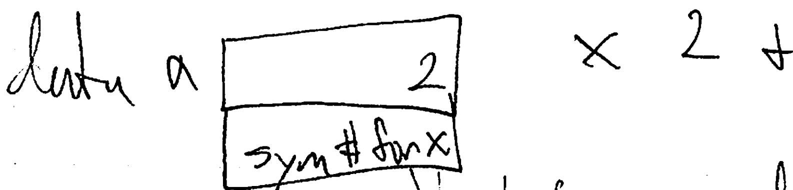
Magic #'s are 4xx because they can be identified by first byte

* Might think about how to do common in Link.



relocation





every occurance of data has a relocation word corresponding to it.

[illegible]

[illegible]
[illegible]

[illegible]

[illegible]

[illegible]
[illegible]

[illegible]

[illegible]
[illegible]

[illegible]
[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible] × [illegible]

[illegible]

[illegible]

[illegible]

11/3 H

Want to avoid having to relocate upon execution
— Something about 1k, 8k boundaries

DEC-10

UNIX format is somewhat similar to ours

File is seq of blocks:

0	type code, len, not count ended
1	18 2 bit byte hunks
2-19	18 words of type-dep data

REQUEST LIBRARY block directs link-10 to search a library

Rad50 - 50 is octal

"If I ever catch any of you doing this, I'll take your degree away."

"It is well known that LC are faster to type because they aren't as big."

11/3

~~Dyn~~ D Linking

do all lkg at exec time

usually (best?) works w/ hardware support

DL can be done on per-object file basis

11/8 T

Didn't touch his pass 1 except for making sure it didn't read too far

When symbols are defined + referenced

Do fread after using pass 1 on each entry.

-l - list of everything defined in a module

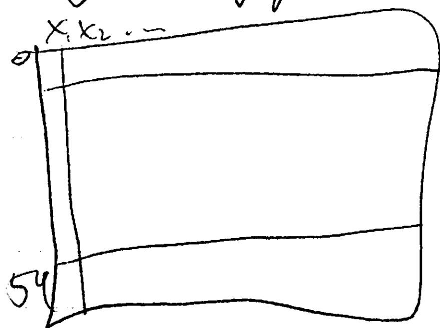
* Common question for exam"
4 questions - closed book

"Heisenbergs -- If you touch something you're probably going to break it"

* How about using debugger on the debugger?

Symbol name conflicts

* Making debuggers robust is very hard



Cyber

[illegible]

[illegible]
[illegible]

[illegible]

[illegible]
[illegible]

[illegible]

[illegible]

[illegible]

[illegible]
[illegible]

[illegible]
[illegible]

[illegible]

[illegible]

[illegible]

11/8T

400100p

p%i

400217 - printf
400100 - p

-" .. and queens are hard to place."

[illegible]

[illegible]

[illegible]

11/15 T

Won't use ~~tester~~ lib4.lib

Several undefined entries

db is a throw-away program

?- bad address
??- bad command

task is to write db1

db1 should give "tty:" in "rt"

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

11/17H

"If we needed seatbelts last time we need answers bags today".

- switch - j command + at very start

C version debugger -> debugee

Now - all Emacs fcns from Icon
fix? to call ExecuteBound

[illegible]

[illegible]
[illegible]

[illegible]

[illegible]

[illegible]
[illegible]

11/22T

set/put - set flag ~~addr~~ adrenv

dbl - init addr ~~to~~ 0

test2.rel rtlib.rel db.rel

setup struct once
compile time init

~~use atoi~~

change operand in oct/dec

all #'s?

```
aton(**cp)

~~setradix(cp)~~
~~char **cp;~~

aton(**cp)
char **cp;
{
~~int base~~
int n = 0, base
char *p = *cp, hc
~~char hc;~~
while (isspace(*p)) p++;
~~if (*p)~~
base = (*p == '0') ? 8 : 10;
hc = '0' + (base - 1);
```

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

~~ds~~cdS

11/22 T

Focus is used to guide interpretation of symbols.

* EQ → breakpoint spec?

could map line #'s to structs (at not w/ cpp)

TS411

[illegible]

[illegible]

[illegible]

[illegible]

11/29/T

4001003
b

How could vars be added at run time?

Should understand code in lieu of writing it.

RT Symbol table could stay on disk; good for large sym tables.

How to store named segments from lib input.

How can fields be used to make a linked list.

Some integration questions.

Mods to systems

12/15

code uphill by cdb is practically
(might have disguising fallacy but
that seems unlikely to me.)

"I'll take off points if you
forget to do this, unless you
find something wrong with it, in which
case, my ego will be so
bruised"

7/21

[illegible]

12/6

We have some vacant TA slots -- if you're interested you can pick up ~~the~~ ball and claim of the offer

EC Read problem 4 of first exam & answer

Exam: only db, ckg

10:30am Dec 16

Closed book, closed notes

"Obviously this class presents a challenge to both student & teacher".

How would comp/like be less painful

"This is basically a ~~two~~ pass class".

[illegible]

[illegible]
[illegible]

[illegible]
[illegible]

[illegible]
[illegible]
[illegible]
[illegible]

[illegible]
[illegible]
[illegible]

[illegible]
[illegible]

Registers
fp
dp
sp
pk — trick - last

~~New stack frm~~ ~~12/13~~ 12/13

5. ~~O_CALL~~ in

[1,2]] RTLIB.C

Init
switch

O_CALL
a PARM
ARG

~~Stmd~~
~~Stmd~~ call O_ARG f(1,2)
~~pa~~
parse (*f)()
eval
(*f)(1,2,3)

O_CALL

func args

35%

build stack frame
fix return ~~[illegible]~~ address
on return get ro

do switch \fix up

switch(func, &dbstate)

_switch(&dbstatejunk)

CALL

PAGE N

CALL

will start Game

Turn

12/13

Ø Breakpoint - interrupt

E11

Inspection – [illegible] area 10