A Minimalist's Retargetable C Compiler

Christopher W. Fraser, AT&T Bell Laboratories
David R. Hanson, Princeton University

`http://www.cs.princeton.edu/software/lcc`

Optimize *Our* Time

- economize source code
  - fits in a book
  - retargetable
  - cross compiles
- compile fast
- emit code *we* can use — requires compiling the full language
- good local code; e.g., automatics in registers, but no global optimizations
- `lcc` is a 1.2 MB *literate program*
  - a single source generates the executable and the camera-ready book
- Ramsey, Literate programming simplified, *IEEE Software* 11, 9/94

Statistics

- 12 K lines of target-independent code
- 700 lines for each of three targets (MIPS R3000, SPARC, Intel x86)
- 1K lines of code-generator generator (or 500 in Icon)
- 400 KB text segment includes ***all*** code generators
- compiles itself in half the time that `gcc` does
- speed of emitted code usually is within 20% of `cc` and `gcc` (without `-O`)

Storage Management

- **generality of `malloc`/`free` is often unnecessary in compilers**
- **allocations occur at many *times*, most deallocations occur at *one time***

```
int a;
void f(int b) {
    int c;
    a = b + c;
}
```

  allocation algorithms based on *lifetimes* are better

  stack allocation $\leftrightarrow$ garbage collection

- **arena-based allocation: allocate from large arenas, deallocate entire arenas**

  allocations are nearly as efficient as stack allocation

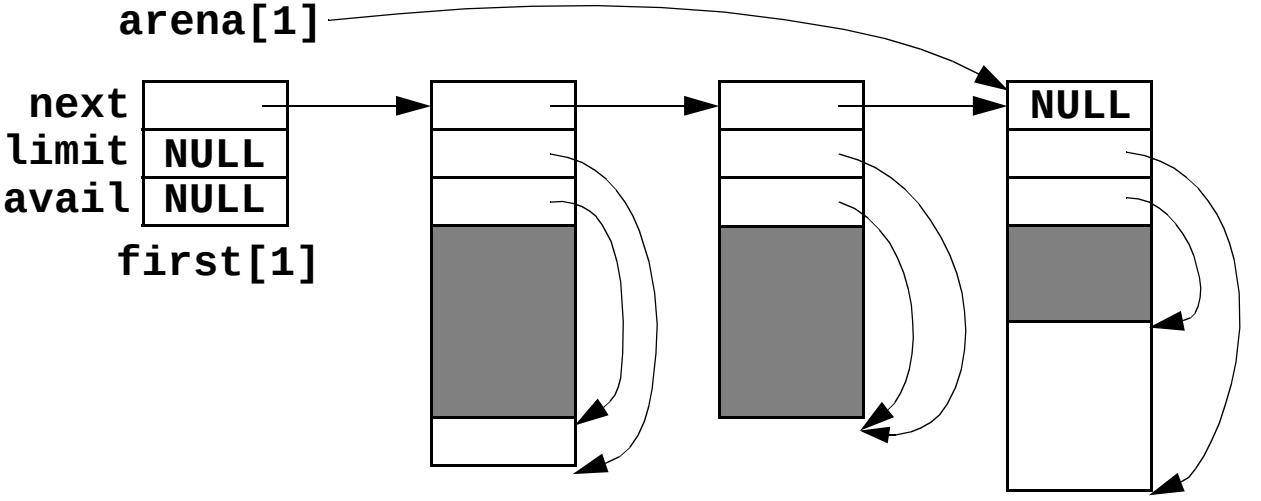
  deallocations are essentially free

  simplifies code: encourages applicative algorithms

- **Hanson, Fast allocation and deallocation of memory based on object lifetimes, *SPE* 20, 1/90**

Arena-Based Allocation

- **maintain *N* arenas, each of which is a linked list of large blocks**



- **to allocate n bytes in arena a**

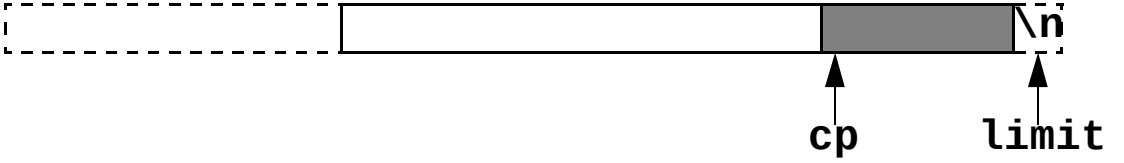
```
while (arena[a]->avail + n > arena[a]->limit)
    ⟨get a new block⟩
arena[a]->avail += n;
return arena[a]->avail - n;
```

- **to deallocate everything in arena a**

```
arena[a]->next = freeblocks;
freeblocks = first[a].next;
first[a].next[a] = NULL;
arena[a] = &first[a];
```

Lexical Analysis

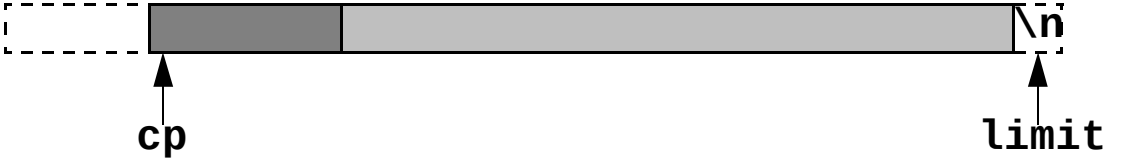
- **minimize character ‘touches’**
- **read input in large chunks, scan directly out of input buffer, don’t move much**



newlines are sentinels; only they require function calls

```
register unsigned char *rcp = cp;
...
while (map[*rcp]&BLANK)
    rcp++;
...
cp = rcp;
```

move only the tail end of the buffer when it’s refilled



exceptions may cause premature refill: string contants, identifiers, numbers

- **Waite, The cost of lexical analysis, *SPE* 16, 5/86**

Recognizing Keywords

- **avoid tables — use inline code**

```
case 'i':
    if (rcp[0] == 'f'
    && !(map[rcp[1]]&(DIGIT|LETTER))) {
        cp = rcp + 1;
        return IF;
    }
    if (rcp[0] == 'n' && rcp[1] == 't'
    && !(map[rcp[2]]&(DIGIT|LETTER))) {
        cp = rcp + 2;
        return INT;
    }
    ...
```

  faster than perfect hashing

- **newer scanner generators (`re2c`, ELI) that generate hard code might do as well**

Code Generation Interface

- shared data structures: symbols, types, nodes, metrics
- 36 generic intermediate representation (IR) operators

```
ADDRF   CVF     BAND    RSH     LT
ADDRG   CVI     BOR     SUB     NE
ADDRL   CVP     BXOR    ASGN    ARG
CNST    CVS     DIV     EQ      CALL
BCOM    INDIR   LSH     GE      RET
CVC     NEG     MOD     GT      JUMP
CVD     ADD     MUL     LE      LABEL
```

- 9 type extensions: `F D C S I U P V B`
- but only 108 type-specific operators

```
ADD+F   ADD+D           ADD+I   ADD+U   ADD+P
RET+F   RET+D           RET+I
CALL+F  CALL+D          CALL+I                  CALL+V  CALL+B
```

- 18 functions

  initialize/finalize the back end (`progbeg progend`)
  define/initialize symbols (`address defsymbol global local import export`)
  initialize/finalize scopes (`blockbeg blockend`)
  generate and emit code (`function gen emit`)
  generate initialized data (`defconst defaddress defstring space segment`)

Interface Records

- **interface records encapsulate target-specific interface data**

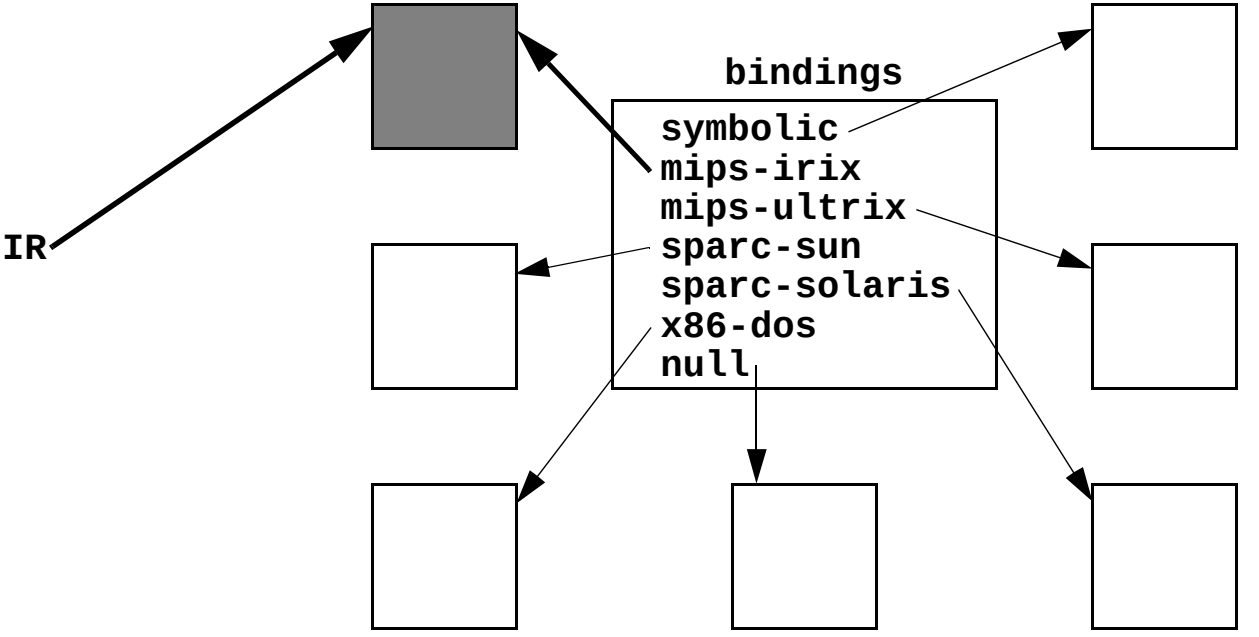
```
typedef struct metrics { unsigned char size, align, inline; } Metrics;

typedef struct interface {
    Metrics charmetric, shortmetric, ..., structmetric;
    unsigned little_endian:1, left_to_right:1, ..., jump_on_return:1;
    void (*address)  (Symbol, Symbol, int);
    void (*blockbeg) (Env *);
    void (*blockend) (Env *);
    void (*defaddress)(Symbol);
    void (*defconst) (int, Value);
    void (*defstring)(int, char *);
    void (*defsymbol)(Symbol);
    void (*emit)     (Node);
    void (*export)   (Symbol);
    void (*function) (Symbol, Symbol [], Symbol [], int);
    Node (*gen)      (Node);
    void (*global)   (Symbol);
    void (*import)   (Symbol);
    void (*local)    (Symbol);
    void (*progbeg)  (int, char **);
    void (*progend)  (void);
    void (*segment)  (int);
    void (*space)    (int);
    Xinterface x;
}  Interface;
```

Cross Compilation

- **`lcc` is built with the code generators for *all* targets**
- **command-line options *bind* the front end to the desired interface record**

```
lcc -Wf-target=mips-irix -S foo.c
```



front end uses indirect calls, ala object-oriented languages

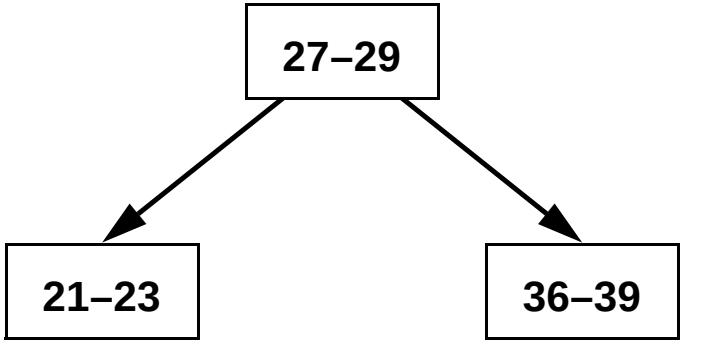
```
(*IR->defsymbol)(p);
```

Switch Statements

- **`lcc` generates a binary search of *dense* branch tables**

  e.g., case labels 21 22 23 27 28 29 36 37 39

```
    if (t1  < 27) goto L1
    if (t1  > 29) goto L2
    t1 = t1 - 27
    goto (L27, L28, L29), t1
L1: if (t1  < 21) goto L3
    if (t1  > 23) goto L3
    t1 = t1 - 21
    goto (L21, L22, L23), t1
L2: if (t1  < 36) goto L3
    if (t1  > 39) goto L3
    t1 = t1 - 36
    goto (L36, L37, L38, L39), t1
```



  for M tables, each with range R_k: $\log M + \text{constant}$ time, $\log M + \sum_{k=1}^{M} R_k$ space

- **the density is the fraction of a table occupied by nondefault destination labels**

  density = 0.50 $\Rightarrow$ (21–23, 27–29, 36–39)
  = 0.66 $\Rightarrow$ (21–23, 27–29) (36–39)
  = 0.75 $\Rightarrow$ (21–23) (27–29) (36–39)
  > 1.00 $\Rightarrow$ binary search of one-element tables

- **command-line option specifies density: `lcc -d0.1 foo.c`; default is 0.5**

Code Quality

- **good enough — hundreds of users, few complaints**
- **lay out loops to avoid unnecessary branches (Baskett, *SIGPLAN Notices* 13, 4/78)**

```
for (e1; e2; e3) S
```

```
        e1
        goto L3
L:      S
L1:     e3
L3:     if (e2 != 0) goto L
L2:
```

- **eliminate common subexpressions in 'extended' basic blocks**

```
if (a[i] && a[i]+b[i] > 0 && a[i]+b[i] < 10) ...
```

`0`, `4*i`, `a[i]`, `b[i]`, and `a[i]+b[i]` are each computed once

- **estimate frequency of use for scalars; allocate registers to those used frequently**

```
{   register int i;
    for (i = 0; i < 100; i++)
        if (a[i] && a[i]+b[i] > 0 && a[i]+b[i] < 10)
            ...   /* i used 10/2=5 times */
}
```

Code Quality, cont'd

- **generated MIPS code**

```
     move $30,$0          i = 0
L.2: sll $25,$30,2        4*i
     lw $24,a($25)        a[i]
     beq $24,$0,L.6       if (a[i]==0) goto L.6
     lw $25,b($25)        b[i]
     addu $25,$24,$25     a[i]+b[i]
     ble $25,$0,L.6       if (a[i]+b[i] <= 0) goto L.6
     bge $25,10,L.6       if (a[i]+b[i] >= 0) goto L.6
     ...
L.6: la $30,1($30)        i++
     blt $30,100,L.2      if (i < 100) goto L.2
```

- **`lcc`'s code generators optimal local code for trees**

Specifying Code Generators

- **tree grammars**

  match trees of intermediate code

  emit assembly code

- **rule syntax**

  nonterminal: pattern template [***cost***]

- **sample instruction rule**

```
reg:    ADDI(reg,reg)    "addu $%c,%0,%1\n"
```

- **sample operand rules**

```
con:    CNSTI            "%a"

addr:   con              "%0"
addr:   ADDI(reg,con)    "%1($%0)"
```

- **optional cost identifies cheapest match per nonterminal**

```
reg:    ADDI(reg,reg)    "addu $%c,%0,%1\n" 1
reg:    con              "la $%c,%0\n"      1

addr:   ADDI(reg,con)    "%1($%0)"          0
```

A Hard-Coded Code Generator

- **traverses each tree bottom-up**

```
static void label(Node p) {
    switch (p->op) { ... }
}
```

- **one case per operator**

```
case CNSTI:
    p->cost[CON] = 0;
    p->rule.con = 2;      /* con: CNSTI */
    closure_con(a, 0);    /* reg: con */
```

- **a typical case (out of 108+9=117)**

```
case ADDI:
    label(p->left);
    label(p->right);      /* reg: ADDI(reg,con) */
    c = p->left->cost[REG] + p->right->cost[CON] + 0;
    if (c < p->cost[ADDR]) ...
```

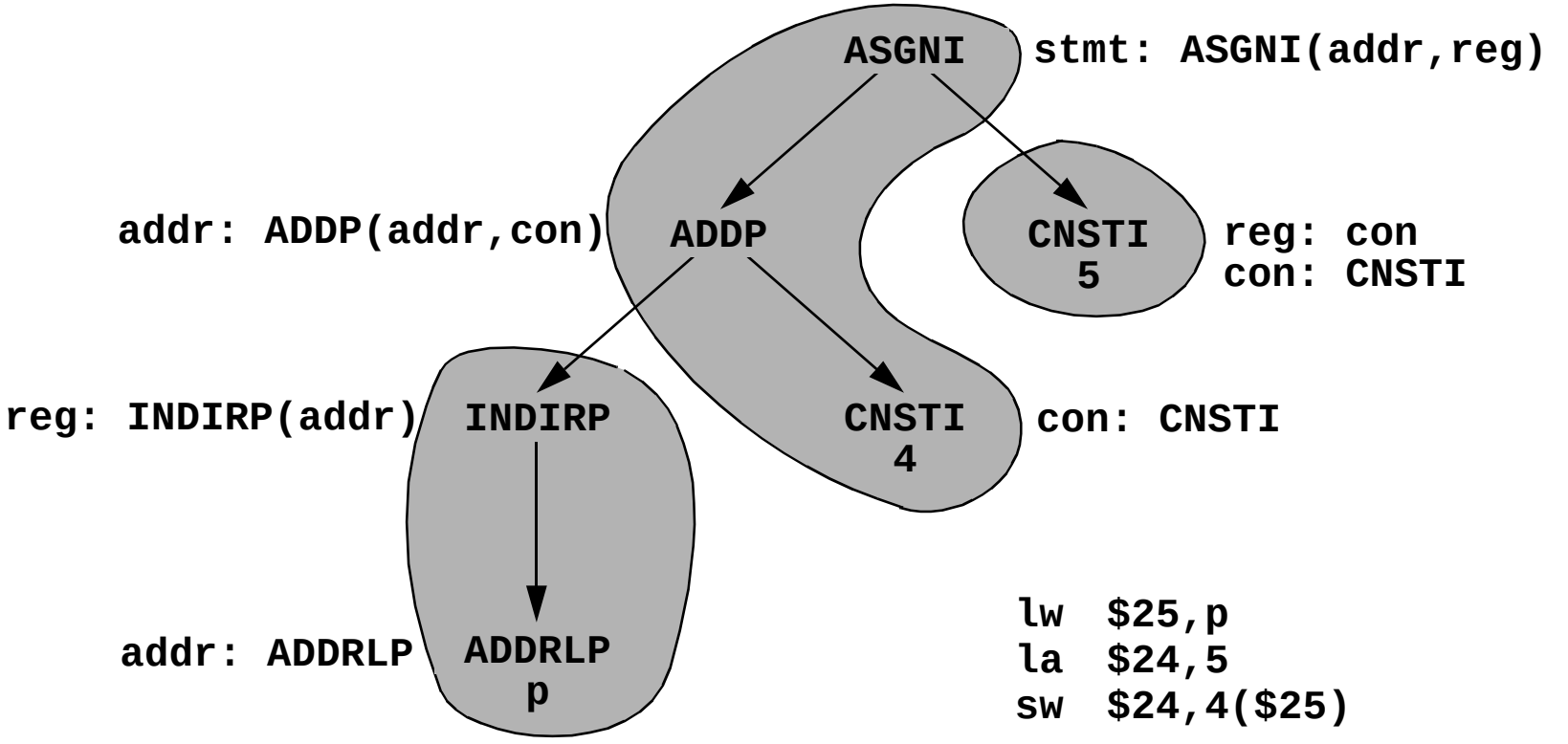
- **`lburg` generates an instance of `label` for each target**

156 rules	**1712 lines**	**MIPS**
185	**2107**	**SPARC**
210	**2422**	**X86**

- **based on `iburg`: Fraser, Hanson, and Proebsting, *LOPLAS* 1, 9/92**

Generating Code

- label tree, e.g., for `int *p; p[1] = 5;`
- pick cheapest rule matching start symbol
- recursively pick rules for the frontier
- fill in and emit corresponding templates



Target Independent Profiling

- **`-b` generates code that counts expression executions, appends data to `prof.out`**
- **bprint reads `prof.out`, prints annotated listing of source program**

```
% lcc -b 8q.c
% a.out
1 5 8 6 3 7 2 4
...
8 4 1 3 6 2 7 5
% bprint
...
queens(c)
<1965>{
    int r;

    for (<1965>r = 0; <15720>r < 8; <15720>r++)
        if (<15720>rows[r] && <5508>up[r-c+7] && <3420>down[r+c]) {
            <2056>rows[r] = up[r-c+7] = down[r+c] = 0;
            <2056>x[c] = r;
            if (<2056>c == 7)
                <92>print();
            else
                <1964>queens(c + 1);
            <2056>rows[r] = up[r-c+7] = down[r+c] = 1;
        }
<1965>}
...
```

More Target Independent Features

- **`-Wf-a` reads `prof.out`, uses measured frequencies instead of estimates**
- **`-t` generates code to trace calls and returns**

```
% lcc -t struct.c
% a.out
main#1() called
makepoint#1(x=-10,y=-10) called
makepoint#1 returned (point){x=-10,y=-10}
addpoint#1(p1=(point){x=320,y=320},p2=(point){x=-10,y=-10}) called
addpoint#1 returned (point){x=310,y=310}
...
```

- **`-n` generates code to catch dereferencing null pointers**

```
% cat bug.c
main() { int *p = 0; *p = 1; }
% lcc -n bug.c
% a.out
null pointer dereferenced @bug.c:1
```

- **`-P` prints ANSI declarations for top-level variables; edit these to convert to ANSI**

```
% lcc -P wf1.c
int main(int, char **);
...
struct node *lookup(char *, struct node **);
int tprint(struct node *);
```

Retrospective

- **nothing's perfect ...**
  - **we should build ASTs?**
  - **we should have built flow graphs**
  - **the *interface* should have distinguished `int` from `long` from `void*`**
  - **we should have provided an interface pickle**
  - **we need to schedule instructions (e.g., on the SPARC)**
  - **we should use a graph coloring register allocator?**
  - **about half of the changes introduce a new error — we need a heftier test suite?**
- **but ...**
  - **it's portable and simple**
  - **it compiles fast**
  - **it's ideal infrastructure for compiler & programming environment research**
  - **we miss global optimization a lot less than we'd miss fast compiles**
  - **it compiles the full language**
  - **it's validated and, worse, kept that way**