Lecture 14
Boolean Expressions

Topics
- Numeric Implementation of Booleans
- Positional Encoding of Booleans
- Short Circuit Evaluation
- If Then else (incorrect version)

Readings: 8.4

March 1, 2006
Overview

Last Time
- LALR(1) Parse Table construction
- Handling Ambiguous Programming Language Constructs
- An Expression Interpreter
- Generating Postfix code

Today's Lecture
- Evaluations of Expressions
- Numeric Implementation of Booleans
- Positional Encoding of Booleans
- Short Circuit Evaluation
- If-then-else semantic actions almost

References: Sections 8.4

Homework:
Intermediate Code

Forms of Intermediate Representations

- Code-like: quadruples (like postfix.y)
- Parse-tree like
## Arithmetic Expressions

### E : E ' + ' E

<table>
<thead>
<tr>
<th></th>
<th>E ' * ' E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E ' - ' E</td>
</tr>
<tr>
<td></td>
<td>E ' / ' E</td>
</tr>
<tr>
<td></td>
<td>( ' E ' )'</td>
</tr>
<tr>
<td></td>
<td>- E</td>
</tr>
<tr>
<td></td>
<td>id</td>
</tr>
</tbody>
</table>

### Other expressions

### E : E ' ^ ' E

<table>
<thead>
<tr>
<th></th>
<th>id '(' Plist ')' // ??</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id '[' Elist ']' // ??</td>
</tr>
<tr>
<td></td>
<td>* id                  //</td>
</tr>
<tr>
<td></td>
<td>&amp; id                  //</td>
</tr>
<tr>
<td></td>
<td>id . id               //</td>
</tr>
<tr>
<td></td>
<td>id \rightarrow id     //</td>
</tr>
</tbody>
</table>

### Others ? - Bitwise and or xor, shifts,
Arithmetic Expressions: Attributes

Possibilities

- **E.place** – pointer to the symbol table
  - Type information
  - Offset information for structures
  - Scope information global/local/nested contexts

- **E.code** – pointer to code that will evaluate the expression

- **E.type**

- **We will generate** “postfix” code and **assume** E.place and install every temporary (as in Examples/PostfixCode)
Boolean Expression Grammar

BoolExpr → not OrTerm | OrTerm
OrTerm → OrTerm OR AndTerm | AndTerm
AndTerm → AndTerm AND Bool | Bool
Bool → RelExpr | true | false
RelExpr → E RelOp E
E → E + E | E * E | ( E ) | ID | NUM | …
Numeric Encoding

True = non-zero
False = zero

Example
   B OR C AND NOT D

Quadruples
NOT  rD  _   T1
AND  rC  T1  T2
OR   rB  T2  T3
Comparison operations in Hardware

- IA32 -
  - CC register set as result of arithmetic operations
    » Add, Subtract, ... CMP = subtract without saving result
  - Jumps then test certain bits in the CC register
    » JLT, JLE, JEQ, JGE, JGT, JNEQ
  - So encoding includes a lot of Jumps
  - \((x < y) \text{ AND } (y < z) \text{ OR } (y=x)\)

<table>
<thead>
<tr>
<th>Label</th>
<th>Opcode</th>
<th>LeftOp</th>
<th>RightOp</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>cmp</td>
<td>x</td>
<td>y</td>
<td>_</td>
</tr>
<tr>
<td>_</td>
<td>JLT</td>
<td>L2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1</td>
<td>LoadI</td>
<td>0</td>
<td></td>
<td>T1</td>
</tr>
<tr>
<td>_</td>
<td>JMP</td>
<td>L3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2</td>
<td>LoadI</td>
<td>1</td>
<td></td>
<td>T1</td>
</tr>
<tr>
<td>L3</td>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: \((x < y) \text{ AND } (y < z)\)

<table>
<thead>
<tr>
<th>Label</th>
<th>Op</th>
<th>A1</th>
<th>A2</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP</td>
<td>X</td>
<td>Y</td>
<td>_</td>
<td></td>
</tr>
<tr>
<td>JLT</td>
<td></td>
<td></td>
<td>L2</td>
<td></td>
</tr>
<tr>
<td>LOADI</td>
<td>0</td>
<td></td>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td></td>
<td></td>
<td>L3</td>
<td></td>
</tr>
<tr>
<td>L2</td>
<td>LOADI</td>
<td>1</td>
<td></td>
<td>T1</td>
</tr>
<tr>
<td>L3</td>
<td>CMP</td>
<td>Y</td>
<td>Z</td>
<td>_</td>
</tr>
<tr>
<td>JLT</td>
<td></td>
<td></td>
<td>L4</td>
<td></td>
</tr>
<tr>
<td>LOADI</td>
<td>0</td>
<td></td>
<td>T2</td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td></td>
<td></td>
<td>L3</td>
<td></td>
</tr>
<tr>
<td>L4</td>
<td>LOADI</td>
<td>1</td>
<td></td>
<td>T2</td>
</tr>
<tr>
<td>L5</td>
<td>AND</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
</tr>
</tbody>
</table>
Positional Encoding

In Positional Encoding we represent the value of an expression by where (the position) you end up in the code.

Example

```java
while(k < 20) {
    sum = sum + k*k;
    k = k + 1;
}
```

Note in the code on the right the value of the boolean expr `k<20` is never explicitly represented other than in the Condition Code Register (CC).

The value is represented by whether you end up at quad 7 or quad 3.
Attributes for Booleans

Consider the example on the next slide.

As we generate the code we don’t know what the target branches should be.

We need to build lists of quads whose target fields need to be filled in later.

B.True – list of quadruples that need to be filled in later (backpatched) to the location that we should branch to if this Boolean is true.

B.False - …
### Example: \((x < y) \text{ AND } (y < z)\)

<table>
<thead>
<tr>
<th>QuadNum</th>
<th>Op</th>
<th>A1</th>
<th>A2</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>IF &lt;</td>
<td>X</td>
<td>Y</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>GOTO</td>
<td></td>
<td></td>
<td>?</td>
</tr>
<tr>
<td>2</td>
<td>IF &lt;</td>
<td>Y</td>
<td>Z</td>
<td>?</td>
</tr>
<tr>
<td>3</td>
<td>GOTO</td>
<td></td>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>

B.true → 2 → \(\lambda\)

B.false → 1 → 3 → \(\lambda\)
### Now a Boolean in an IF-ELSE

<table>
<thead>
<tr>
<th>Program</th>
<th>0</th>
<th>if &lt;</th>
<th>x</th>
<th>y</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin</td>
<td>1</td>
<td>GOTO</td>
<td>_</td>
<td>_</td>
<td>6</td>
</tr>
<tr>
<td>If x &lt; y and y &lt; z</td>
<td>2</td>
<td>if &lt;</td>
<td>y</td>
<td>z</td>
<td>4</td>
</tr>
<tr>
<td>Then</td>
<td>3</td>
<td>GOTO</td>
<td>_</td>
<td>_</td>
<td>6</td>
</tr>
<tr>
<td>Mid = b</td>
<td>4</td>
<td>ASSIGN</td>
<td>b</td>
<td>_</td>
<td>mid</td>
</tr>
<tr>
<td>Else</td>
<td>5</td>
<td>GOTO</td>
<td>_</td>
<td>_</td>
<td>VOID</td>
</tr>
<tr>
<td>Mid = c</td>
<td>6</td>
<td>ASSIGN</td>
<td>c</td>
<td>_</td>
<td>mid</td>
</tr>
<tr>
<td>End</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Functions for Boolean Attributes

- **int nextquad variable** – a variable which maintain the quad number of the next quad that will be generated
- **Makelist (quad)** – build a list with a single quad # on it
- **Merge(list1, list2)** – merge two lists of quads
- **Backpatch(list, target)** – for every quad on “list” fill in the branch target to “target”
Quadlists

A quadlist is just a list of ints

typedef struct node{
    int quadnum;
    struct node *link;
} *QuadList, QuadListNode;
Quadlists

QuadListNode *
makelist(int q)
{
    QuadListNode *tmp;
    tmp = (QuadListNode *) malloc(sizeof (QuadListNode));
    tmp -> quadnum = q;
    return(tmp);
}

void backpatch(QuadList p, int q)
{
    while (p != NULL)
    {
        target[p->quadnum] = (QuadListNode *) q;
        p = p -> link;
    }
}
Dumplist – A Debugging Support Routine

void
dumplist(char *label, LIST p)
{
    printf("DUMPLIST %s", label);
    while (p != NULL){
        printf(" %d ",p->quadnum);
        p = p -> link;
    }
}
Stack types – Multiple Type Attributes

%union{
    struct nlist *place;
    struct {
        QuadListNode  *true;
        QuadListNode  *false;
    } quadlist;
    int quad;
    int type;
    LIST next;
}
Stack types – Multiple Type Attributes II

%type <place> expr
%type <list> B
%type <quad> M
%type <next> N
%type <next> L
%type <next> S
%token <place> ID
%token <type> RELOP
## Intermediate Code Generation

### Quadruples
- **OPCODE** | **LeftOperand** | **RightOperand** | **Result**
- ADD | “x” | “y” | “z”
- GOTO | quadnumber
- What are the type of these?

### Quadruples – implemented as 5 parallel arrays
- int opcode[CODESIZE];
- struct nlist *left[CODESIZE];
- struct nlist *right[CODESIZE];
- struct nlist *result[CODESIZE];
- int branchTarget[CODESIZE];

### We could use a union and merge result and branchTarget, but this just over complicates issues.
void

gen(int op, struct nlist *p1, struct nlist *p2, struct nlist *r, int t)
{
    opcode[nextquad] = op;
    op1[nextquad] = p1;
    op2[nextquad] = p2;
    result[nextquad] = r;
    branchTarget[nextquad] = t;
    nextquad = nextquad + 1;
}
Semantic Actions for $B \rightarrow \text{ID RELOP ID}$

\[
B: \quad \text{ID RELOP ID} \quad \{
\text{gen}($2, \$1, \$3, \text{NULL, VOID}$);
\text{gen}(\text{GOTO, NULL, NULL, NULL, NULL, VOID}$);
\$$.\text{true} = \text{makelist}(\text{nextquad} - 2)$;
\$$.\text{false} = \text{makelist}(\text{nextquad} - 1)$;
\}
\]
Markers

Markers are typically nonterminals that derive $\epsilon$ that are inserted to insure an action is performed at a given time.

A common thing need is to remember the quad number where something starts, so the attribute of a marker is just the next quad number.

$M \rightarrow \epsilon \quad \{ \text{M.quad} = \text{nextquad}; \}$

So instead of

- $S \rightarrow \text{if } B \text{ then } S \text{ else } S$

We use

- $S \rightarrow \text{if } B \text{ then } M_1 S \text{ else } M_2 S \quad ***\text{Almost}$
Semantic Actions for $B \rightarrow B$ AND $M$ B

$B \rightarrow B$ AND $M$ B

{ 
backpatch($1$.true,$3$);

$\$.true = $4$.true;

$\$.false = merge($1$.false,$4$.false);
}


Semantic Actions for \texttt{B} \rightarrow \texttt{B OR M B}

\texttt{B} \rightarrow \texttt{B OR M B} \quad \{ \\
\texttt{backpatch($1.\texttt{false},$3);} \\
\texttt{$$.\texttt{false} = $4.\texttt{false};} \\
\texttt{$$.\texttt{true} = \texttt{merge($1.\texttt{true},$4.\texttt{true});}} \\
\}
Semantic Actions for
S → if B then M S else M S ***Almost

S: IF B THEN M S N ELSE M S {

    backpatch($2.true, $4);
    backpatch($2.false, $8);
    tmplist = merge($5, $6);
    $$ = merge(tmplist, $9);
}

;

● Why almost?
Semantic Actions for Assignments

S: ID ASSIGNOP expr { 
    gen(ASSIGNOP, $<place>3, NULL, $1, VOID);
    $$ = NULL;
}
Semantic Actions for Markers

\[ M: \quad \{ \text{nextquad}; \} \]
Project 3 – Generating Postfix Code for Expressions

- Expressions
- Booleans
- If B then assign else assign
- Undeclared variables print error message including line number
- Mixed mode expressions (Graduate students only) Extra credit for ugrads
  - \( \text{int } k; \text{ float } f; \quad k + f \rightarrow \text{(float)}k + f \)
  - Code \( \text{toFloat } k \_ t1 \)
  - \( \text{addf } t1 f t2 \)
- Write up in the email soon
Debugging Parsers written with Yacc

1. **Debug the grammar**
   1. Rewrite grammar to eliminate reduce/reduce and as many shift/reduce as you can.
   2. Tracing parses using
      - `-t` option to bison or yacc
      - `-DYYDEBUG` compile option
      - `int yydebug=1;` in bison specification (C definitions section `%{ ..%}`
      - `extern int yydebug;` in lex specification

2. **Debug the semantic actions**
   - Compile with `-g` option; set `CFLAGS=-g` in Makefile and use `gcc $(CFLAGS) ...` as the compile (or rely on the builtin rules)
   - Use `gdb` (Gnu debugger) to debug the program
Common Mistakes

- **Segmentation fault** - This means you have referenced a memory location that is outside of the memory segment of your program.
  - You have a pointer that has a bad value!
  - First make sure every time you copy a string value you use `strdup`. Several people have had errors with `strcat(s,t)` where they did not allocate space for the string “s”.
  - Use `gdb` and `bt` (backtrace) to trace down the pointer with the bad value.
GDB - Essential Commands

gdb program [core] - debug program [using coredump core]

b [file:] function  set breakpoint at function [in file]
run [arglist]      start your program [with arglist]
bt backtrace:       display program stack
p expr              display the value of an expression

continue running your program
next line, stepping over function calls
next line, stepping into function calls
Example using gdb

deneb> make
bison -d decaf.y
decaf.y contains 51 shift/reduce conflicts.
gcc -c -g decaf.tab.c
flex decaf.l
gcc -DYYDEBUG -g -c lex.yy.c
gcc -c -g tree.c
gcc -DYYDEBUG -g decaf.tab.o lex.yy.o tree.o -ly -o decaf
deneb> ./decaf < t1

Keyword int

Segmentation Fault (core dumped) ⬅️ !!!
Example using gdb

deneb> make
bison -d decaf.y

decaf.y contains 51 shift/reduce conflicts.
gcc -c -g decaf.tab.c
flex decaf.l
gcc -DYYDEBUG -g -c lex.yy.c
gcc -c -g tree.c
gcc -DYYDEBUG -g decaf.tab.o lex.yy.o tree.o -ly -o decaf
deneb> ./decaf < t1

Keyword int

Segmentation Fault (core dumped)

Note the use of the –g option (CFLAGS=-g in Makefile)