Lecture 14
Boolean Expressions

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

Readings: 8.4

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 '4' E  
E : E '^

| E 'ten E
| E '4' E
| E 'x E
| '(' E 'x')
| - E
| id

Other expressions
E : E '^^' E
E : E 'id ' Plist 'id' // ??
E : id 'id ' Elist 'id' // ??
E : * id //
E : & id //
E : id . id //
E : id \ id //
E : ;
E : 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

**Numeric Encod. Extended to Comparisons**

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) AND (y < z) OR (y=x)
    - Label Opcode LeftOp RightOp Target
      - L0 cmp x y _
      - _ JLT L2
      - L1 LoadI 0 T1
      - _ JMP L3
      - L2 LoadI 1 _ T1
      - L3 NOP
**Example: (x < y) 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>L2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOADI</td>
<td>0</td>
<td>T1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td></td>
<td>L3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2</td>
<td>LOADI</td>
<td>1</td>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>L3</td>
<td>CMP</td>
<td>Z</td>
<td></td>
<td>_</td>
</tr>
<tr>
<td>JLT</td>
<td></td>
<td>L4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOADI</td>
<td>0</td>
<td>T2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td></td>
<td>L3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L4</td>
<td>LOADI</td>
<td>1</td>
<td>T2</td>
<td></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:

```plaintext
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.

**Example: (x < y) 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</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</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 \[\rightarrow \lambda\]

B.false: 1 \[\rightarrow\] 3 \[\rightarrow\] \lambda
**Now a Boolean in an IF-ELSE**

```
program
begin
if x < y and y < z then
    mid = b
else
    mid = c
end
```

**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

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

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

```c
void
backpatch(QuadList p, int q)
{
    while (p != NULL){
        target[p->quadnum] = (QuadListNode *) q;
        p = p -> link;
    }
}
```

**Dumplist – A Debugging Support Routine**

```c
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**

```c
%union{
    struct nlist *place;
    struct {
        QuadListNode *true;
        QuadListNode *false;
    } quadlist;
    int quad;
    int type;
    LIST next;
}
```

**Stack types – Multiple Type Attributes II**

```c
%type <place> expr
%type <list> B
%type <quad> M
%type <quad> M
%type <next> N
%type <next> L
%type <next> S
%token <next> 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 → ID RELOP ID

B: ID RELOP ID

{  
    gen($2, $1, $3, NULL, VOID);
    gen(GOTO, NULL, NULL, NULL, VOID);
    $$ .true = makelist(nextquad -2);
    $$ .false = makelist(nextquad -1);
}

Markers

Markers are typically nonterminals that derive ε 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 → ε  { M.quad = nextquad; }

So instead of

- S → if B then S else S
- We use
- S → if B then M₁ S else M₂ S

***Almost
Semantic Actions for $B \rightarrow B \text{ AND } M \text{ B}$

$$B \rightarrow B \text{ AND } M \text{ B}$$

{ 
backpatch($1\text{.true},$3);
$$\text{true} = \text{false} = \text{true} = \text{merge}($1\text{.false},$4\text{.false});$
}

Semantic Actions for $B \rightarrow B \text{ OR } M \text{ B}$

$$B \rightarrow B \text{ OR } M \text{ B}$$

{ 
backpatch($1\text{.false},$3);
$$\text{false} = \text{true} = \text{false} = \text{merge}($1\text{.true},$4\text{.true});$
}

Semantic Actions for $S \rightarrow \text{if} B \text{ then } M \text{ S else } M \text{ S }$ ***Almost

$$S: \text{IF } B \text{ THEN } M \text{ S } \text{ N ELSE } M \text{ S}$$

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

; 
● Why almost?

Semantic Actions for Assignments

$$S: \text{ID ASSIGNOP expr }$$

{ 
gen(ASSIGNOP,$<place>3$, NULL,$1$, VOID);
$$ = \text{NULL};$
}

|
Semantic Actions for Markers

M: \{ \$$ = \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 undergrads
  Code toFloat k → (float)k + f
  ```
  int k; float f;      k + f
  Code toFloat k → t1
  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
      - `-DYDEBUG` 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 everytime you copy a string value you use `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

Write up in the email soon.
**GDB - Essential Commands**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gdb program [core]</td>
<td>debug program [using coredump core]</td>
</tr>
<tr>
<td>b [file:] function</td>
<td>set breakpoint at function [in file]</td>
</tr>
<tr>
<td>run [arglist]</td>
<td>start your program [with arglist]</td>
</tr>
<tr>
<td>bt backtrace:</td>
<td>display program stack</td>
</tr>
<tr>
<td>p expr</td>
<td>display the value of an expression</td>
</tr>
<tr>
<td>c</td>
<td>continue running your program</td>
</tr>
<tr>
<td>n</td>
<td>next line, stepping over function calls</td>
</tr>
<tr>
<td>s</td>
<td>next line, stepping into function calls</td>
</tr>
</tbody>
</table>

---

**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) ← !!!