Lecture 15
Control Flow

Topics
- Review Positional Encoding of Booleans
- Short Circuit Evaluation
- Control Flow Statements

Readings: 8.4, 8.6

March 13, 2006
Overview

Last Time

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

Today’s Lecture

- Review
  - Booleans, quadruples
- Control Flow using numeric implementation of Booleans
- Control Flow using

References: Sections 8.4, 8.5

Homework:
Rest of the Semester Plan

Project 3 due Sunday March 19

Project 4 – Booleans and Control Flow due Sunday March 26

Test 2 – April 5

Project 5 – Function Calls

Project 6 – Nested Scopes

Exam April 27
Recall Numeric Rep. of Booleans

E → id₁ relop id₂

{ E.place = newtemp();
  emit(if id₁.place relop.op id₂.place goto nextquad+3)
  emit(E.place := ‘0’);
  emit(goto nextquad +2);
  emit(E.place := ‘1’);
}
AND, OR in numeric implementation

E \rightarrow E_1 \text{ AND } E_2 \quad \{ \\
\quad \text{E.place} = \text{newtemp}(); \\
\quad \text{emit(E.place ‘:=’ E}_1\text{.place ‘and’ E}_2\text{.place);} \\
\}

E \rightarrow E_1 \text{ OR } E_2 \quad \{ \\
\quad \text{E.place} = \text{newtemp}(); \\
\quad \text{emit(E.place ‘:=’ E}_1\text{.place ‘or’ E}_2\text{.place);} \\
\}
Booleans Numeric Impl. – Finish Up

\[ E \rightarrow \text{true} \]
\[ \{ E\.place = \text{newtemp}(); \]
\[ \hspace{1cm} \text{emit}(E\.place \text{ \texttt{:=} } '1'); \} \]

\[ E \rightarrow \text{false} \]
\[ \{ E\.place = \text{newtemp}(); \]
\[ \hspace{1cm} \text{emit}(E\.place \text{ \texttt{:=} } '0'); \} \]

\[ E \rightarrow ( E_1 ) \]
\[ \{ E\.place = E_1\.place; \} \]
Positional Implementation of Booleans

- **Attributes**
  - `E.true`
  - `E.false`

- **Functions**
  - `makelist( quadnum )`
  - `merge( list1, list2 )`
  - `backpatch( list, quadnum )`
void

backpatch(QuadList p, int q)
{
    while (p != NULL){
        target[p->quadnum] = (QuadListNode *) q;
        p = p -> link;
    }
}
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 \{ \\
\quad \text{gen}($2, $1, $3, \text{NULL, VOID}); \\
\quad \text{gen}(\text{GOTO, NULL, NULL, NULL, VOID}); \\
\quad \text{$$}.true = \text{makelist(nextquad -2)}; \\
\quad \text{$$}.false = \text{makelist(nextquad - 1)}; \\
\}
\]

;
Markers

Markers are typically nonterminals that derive $\varepsilon$ 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.

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

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 → B AND M B

B → B AND M B { 
  backpatch($1.true,$3);
  $$\cdot true = $4.true;
  $$\cdot false = merge($1.false,$4.false);
}

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

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

\{

backpatch($1$.false,$3$);

$\$$.false = $4$.false;

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

\}


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

- $N \rightarrow \epsilon$  

  { $N$.next = nextquad; 
    gen(goto, _, _, _, void); }
Semantic Actions for Assignments

S: ID ASSIGNOP expr
   { 
     gen(ASSIGNOP, $3$, NULL, $1$, VOID);
     $$ = NULL;
   }

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
c continue running your program
n next line, stepping over function calls
s 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)
Run GDB
Attributes for Booleans and Control Flow*

*Assuming positional encoding.

**B** (for boolean expression)
- B.true – A list of …… that need to …
- B.false

**Functions for Boolean Attributes**
- int nextquad variable –
- Makelist (quad)
- Merge(l1, l2)
- Backpatch(List, target)
Markers

Markers – Grammar symbol that only derive $\varepsilon$ which are used to perform some semantic action and usually to save some attribute value.

Example $S \rightarrow A\ B\ M\ C\ D$

- In this case we want to perform some action when the production $M \rightarrow \varepsilon$ is reduced, like saving the starting place of the code the evaluates $C$.

Markers for handling Booleans using the positional encoding:

- $M \rightarrow \varepsilon$ (M.quad – marks the start of a piece of code)
- $N \rightarrow \varepsilon$ (N.next – a list consisting of a single quad that needs to have its target field filled in later.)
Semantic Actions Control Flow

\[
S \rightarrow \text{LHS ASSIGNOP E ';'} \quad \text{L} \rightarrow \text{L } S
\]

| if-statement | S |
| if-else-statement | ; |
| while-statement | |
| for-statement | |
| BEGIN L END | |
| function-calls ??? | |

/* this is not really what we mean by control flow we will do this later */

Now what attributes do we need for these nonterminals? (S, L)
As an example consider

```
while (B-exp){
  S1
  S2
  S3
}
```

Now what attributes do we need for these nonterminals? (S, L)
Semantic Actions for
S → if B then M S N else M S

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);
}

;  
N → ε  
{ N.next = nextquad;
gen(goto, _, _, _, void); }
Semantic Actions for $S \rightarrow \text{ if } B \text{ then } M \ S$

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

  backpatch($2.\text{true}, \ 4);$

  $$ = \text{ merge}(2.\text{false}, \ 5);$

  \}

;
While Statement
For statement
Project 4 – Generating Postfix Code for Expressions and Control Flow

- Booleans
- If B then assign else assign
- Undeclared variables print error message including line number
- Write up in the email soon
Procedure Calls