Notes on Generating Control Structures for Paxi

David Nordstrom
CS 440
George Mason University

Welcome to Program 4! These notes cover only the control structures part of the assignment. This involves the boolean expressions, conditionals, and loops.

Boolean stuff

Generating code to leave a boolean on the stack happens in production 42:

```
     42. <boolean_atom> ->( <arithmetic_expression>
                            <relational_operator>
                            <arithmetic_expression> )
                          | ( <boolean_expression> )
```

Generated code from the first RHS should look like (* and ** are branching targets):

```
     POP       // move values from stack to scratch area
     POP
     SUB       // since branch instructions can only compare to 0
     branch to *    // conditional branch
     PUSHI 0
     B **           // unconditional branch
*    PUSHI 1
**
```

This code will leave the difference of the two values (from <arithmetic_expression>) being compared in a scratch area and then push either 0 or 1 on the stack depending on what the relational operator is. The relational operator is passed up the parse tree from production 43 (<relational_operator>). It remains to determine what the (code store) addresses for * and ** are.

When generating code you keep a (global) variable with the index of the next available location in the code store. (This variable is used in your "emit()" function.) Let's call this variable int current_code. The conditional branch must be to a location three instructions ahead of current_code, hence to current_code + 9. The unconditional branch must be to a location two instructions ahead of

current_code: current_code + 6.

The other productions for boolean operations (productions 39 through 41) are handled in a manner similar to the way we handled arithmetic expressions (but easier).

No code need be generated from the second RHS of production 42 since the desired boolean value is already on the stack.

If and if ... else

Now we are looking at productions 19 and 20:

```
     19. <conditional> -> if <boolean_expression> <statement_list>
                          <else_clause> endif

     20. <else_clause> -> else <statement_list> | EMPTY
```

Code generated from production 19 should look like (the &s represent places in your code, not target addresses):

```
     POPD          // boolean to scratch location
&                  // save address for backpatching
     BEQ *         // branch if boolean was false
     // code generated from <statement_list> appears here
&&                 // save address for backpatching
     B **          // unconditional branch over <else_clause>
&&&                // time to backpatch BEQ
*
     // code generated from <else_clause> appears here
&&&&               // time to backpatch B
**
```

We first notice that we need actions embedded in several places in the RHS: before <statement_list>, between <statement_list> and <else_clause> and after <else_clause>. Remember that actions occupy a position in tne $n numbering!

The generated code will test the boolean popped from the stack for false with the BEQ (jumps over the "true" case) then after the statement list does an unconditional branch (B) over the else clause.

There are two problems here. The first is determining the target addresses (* and **) for the branching instructions. At the time we would like to generate the BEQ we can't know the target address since it depends on the code from <statement_list> which hasn't been generated yet. Similarly we don't know the target for B since it

depends on the code generated from <else_clause>.

We could resolve this problem if we had a computer with a clock speed faster than the speed of light so that we could see into the future. Failing this we must *backpatch*.

Backpatching

The idea is that we *don't* generate the branching instructions from the actions suggested by the pseudo-code shown above. Instead we generate a dummy instruction to hold a place open and fill in the instruction from a later action where we know the target address. (Hint: for a dummy address generate something you will recognize in your code like 999, 999, 999 -- this may help in your debugging.) You will know the target address for the BEQ at &&& and you will know the target address for B at &&&&. At these points in your program you will overwrite the dummy instructions with the branching instructions with their proper target addresses.

The locations where you left the dummy instructions must be saved somewhere so that when you backpatch you know the location to write to. We do this at & and &&.

This presents the second problem. Our first thought is to use global variables in the parser file to store the locations of the dummy instructions. This won't work because the conditionals can be nested. These locations must be stored on a stack. Fortunately there is a stack provided for us: we can piggyback on the parse stack!

Recall that the symbols from the CFG are stored on a parse stack in LR parsing. In bison pointers to actions are also stored on the parse stack (hence occupying positions in the $n numbering) and the yylval values attached to the symbols are interleaved with them on the stack. Even if there was nothing assigned to a grammar symbol the space for "$n" is still allocated and sitting unused on the parse stack. We can use this space as we like -- in this case to store backpatching addresses. To store the address for (for example) the BEQ you must store the value of current_counter *just before* generating the dummy instruction. Store the value in $<intval>n where n indicates a symbol (or action) which we are not using and which precedes the action we are in. Slick, huh?

You *must* (for readability of your program) write a backpatch(int, int, int, int) function which puts an instruction at a location in the code store. Call this function when backpatching -- do *not* just write integers for assignments into your actions.

Very important: Before you generate code for the conditionals make sure that you thoroughly understand this discussion and know what you are doing. Don't just formalize the pseudo-code above.

Else

There is no need to generate code from production 20 (<else_clause>).

Loops

You will handle the loops (productions 22 and 23) in a similar manner. You can figure these out.