Part 49: The Ternary Operator

In this part of our compiler writing journey, I've implemented the ternary operator. This is one of the really natty operators in the C language which can reduce lines of code in your source file. The basic syntax is:

```
ternary_expression:
        logical_expression '?' true_expression ':' false_expression
        ;
```

We evaluate the logical expression. If this is true, we then evaluate only the true expression. Otherwise, we only evaluate the false expression. The result of either the true or the false expression becomes the result of the whole expression.

One subtlety here is that, for example, in:

```
   x= y != 5 ? y++ : ++y;
```

If `y != 5` then `x= y++`, otherwise `x= ++y`. Either way, `y` is only incremented once.

We can rewrite the above code as an IF statement:

```
if (y != 5)
  x= y++;
```

```
else
  x= ++y;
```

However, the ternary operator is an expression, so we can also do:

```
  x= 23 * (y != 5 ? y++ : ++y) - 18;
```

This can't be easily converted into an IF statement now. However, we can borrow some of the mechanics from the IF code generator to use for the ternary operator.

Tokens, Operators and Operator Precedence

We already have ':' in our grammar as a token; now we need to add the '?' token. This is going to be treated as an operator, so we'd better set its precedence.

According to this list of C operators, the '?' operator has precedence just above the assignment operators.

The way we have designed our precedence, our operator tokens must be in precedence order and the AST operators must correspond to the tokens.

Thus, in `defs.h`, we now have:

```
// Token types
enum {
  T_EOF,

  // Binary operators
  T_ASSIGN, T_ASPLUS, T_ASMINUS,
  T_ASSTAR, T_ASSLASH,
  T_QUESTION,                     // The '?' token
  ...
enum {
  A_ASSIGN = 1, A_ASPLUS, A_ASMINUS, A_ASSTAR, A_ASSLASH,
  A_TERNARY,                      // The ternary AST operator
  ...
```

And in `expr.c`, we now have:

```
static int OpPrec[] = {
  0, 10, 10,                      // T_EOF, T_ASSIGN, T_ASPLUS,
  10, 10, 10,                     // T_ASMINUS, T_ASSTAR, T_ASSLASH,
    15,                           // T_QUESTION
    ...
```

As always, I will leave you to browse the changes in `scan.c` for the new T_QUESTION token.

Parsing the Ternary Operator

Even though the ternary operator isn't a binary operator, because it has precedence, we need to implement it in `binexpr()` with the binary operators. Here's the code:

```
struct ASTnode *binexpr(int ptp) {
  struct ASTnode *left, *right;
  struct ASTnode *ltemp, ...

    switch (ASTop) {
    case A_TERNARY:
      // Ensure we have a ':' token, scan in the expression after it
      match(T_COLON, ":");
      ltemp= binexpr(0);

      // Build and return the AST for this statement. Use the middle
      // expression's type as the return type. XXX We should also
      // consider the third expression's type.
      return (mkastnode(A_TERNARY, right->type, left, right, ltemp, NULL, 0));
      ...
    }
    ...
}
```

When we hit the A_TERNARY case, we have the AST tree of the logical expression stored in `left`, the true expression in `right` and we have parsed the '?' token. Now we need to parse the ':' token and the false expression.

With all three tokens parsed, we can now build an AST node to hold all three. One problem is how to determine the type of this node. As you can see, it's easy to choose the type of the middle token. To do it properly, I should see, of the true and false expressions, which one is wider and choose that one. I'll leave it for now and revisit it.

Generating The Assembly Code: Issues

Generating the assembly code for the ternary operator is very similar to that of the IF statement: we evaluate a logical expression. If true, we evaluate one expression; if false, we evaluate the other. We are going to need some labels, and we are going to have to insert jumps to these labels as required.

I did actually try to modify the `genIF()` code in `gen.c` to do both IF and the ternary operator, but it was easier just to write another function.

There is one wrinkle to the generation of the assembly code. Consider:

```
x= (y > 4) ? 2 * y - 18 : y * z - 3 * a;
```

We have three expressions, and we need to allocate registers to evaluate each one. After the logical expression is evaluated and we have jumped to the correct next section of code, we can free all the registers used in the evaluation. For the true and false expressions, we can free all the registers *except one*: the register holding the expression's rvalue.

We also can't predict which register this will be, because each expression has different operands and operators; thus, the number of registers used will differ, and the (last) register allocated to hold the result may be different.

But we need to know which register does hold the result of both the true and false expressions, so when we jump to the code that will use this result, it knows which register to access.

Thus, we need to do three things:

- allocate a register to hold the result *before* we run either the true of false expression,
- copy the true and false expression result into this register, and
- free all registers *except* the register with the result.

Freeing Registers

We already have a function to free all registers, `freeall_registers()`, which takes no arguments. Our registers are numbered zero upwards. I've modified this function to take, as an argument, the register we want to *keep*. And, in order to free *all* registers, we pass it NOREG which is defined to be the number `-1`:

```
// Set all registers as available.
// But if reg is positive, don't free that one.
void freeall_registers(int keepreg) {
  int i;
  for (i = 0; i < NUMFREEREGS; i++)
    if (i != keepreg)
      freereg[i] = 1;
}
```

Throughout the compiler, you will now see `freeall_registers(-1)` to replace what used to be `freeall_registers()`.

Generating The Assembly Code

We now have a function in `gen.c` to deal with the ternary operator. It gets called from the top of `genAST()`:

```
  // We have some specific AST node handling at the top
  // so that we don't evaluate the child sub-trees immediately
  switch (n->op) {
  ...
  case A_TERNARY:
    return (gen_ternary(n));
```

Let's have a look at the function in stages.

```
// Generate code for a ternary expression
static int gen_ternary(struct ASTnode *n) {
  int Lfalse, Lend;
  int reg, expreg;

  // Generate two labels: one for the
  // false expression, and one for the
  // end of the overall expression
  Lfalse = genlabel();
  Lend = genlabel();

  // Generate the condition code followed
  // by a jump to the false label.
  genAST(n->left, Lfalse, NOLABEL, NOLABEL, n->op);
  genfreeregs(-1);
```

This is pretty much exactly the same as the IF generating code. We pass the logical expression sub-tree, the false label and the A_TERNARY operator into `genAST()`. When `genAST()` sees this, it knows to generate a jump if false to this label.

```
    // Get a register to hold the result of the two expressions
    reg = alloc_register();

    // Generate the true expression and the false label.
    // Move the expression result into the known register.
    expreg = genAST(n->mid, NOLABEL, NOLABEL, NOLABEL, n->op);
    cgmove(expreg, reg);
    // Don't free the register holding the result, though!
    genfreeregs(reg);
    cgjump(Lend);
    cglabel(Lfalse);
```

With the logical expression done, we can now allocate the register to hold both the true and false expression results. We call `genAST()` to generate the true expression code, and we get back the register with the result. We now have to move this register's value into the known register. With this done, we can free all registers except the known register. If we did the true expression, we now jump to the end of the ternary assembly code.

```
    // Generate the false expression and the end label.
    // Move the expression result into the known register.
    expreg = genAST(n->right, NOLABEL, NOLABEL, NOLABEL, n->op);
    cgmove(expreg, reg);
    // Don't free the register holding the result, though!
    genfreeregs(reg);
    cglabel(Lend);
    return (reg);
  }
```

And the code to evaluate the false expression is very similar. Either way, execution will get to the end label and, once we get here, we know that the ternary result is in the known register.

Testing the New Code

I was worried about nested ternary operators, which I've used quite a bit in other code. The ternary operator is *right associative*, which means we bind the '?' to the right more tightly than to the left.

Fortunately, as we greedily seek out the ':' token and the false expression once we have parsed the '?' token, our parser is already treating the ternary operator as right associative.

`tests/input121.c` is an example of a nested ternary operator:

```
#include <stdio.h>

int x;
int y= 3;

int main() {
  for (y= 0; y < 10; y++) {
    x= (y < 4) ? y + 2 :
       (y > 7) ? 1000 : y + 9;
    printf("%d\n", x);
  }
  return(0);
}
```

If `y<4`, then `x` becomes `y+2`. If not, we evaluate the second ternary operator. If `y>7`, `x` becomes 1000, otherwise it becomes `y+9`.

The effect is to do `y+2` for `y` values 0 to 3, `y+9` for `y` values 4 to 7, and 1000 for higher y values:

```
2
3
4
5
13
14
15
16
1000
1000
```

Conclusion and What's Next

Like a few of the steps so far, I was apprehensive to tackle the ternary operator because I thought it would be very difficult. I did have problems putting it into the IF generating code, so I stepped back a bit. Actually, I went out to see a movie with my wife and this gave me a chance to mull things over. I realised that I had to free all but one registers, and I should write a separate function. After that, writing the code was straight forward. It's always good to step away from the keyboard now and then.

In the next part of our compiler writing journey, I will feed the compiler to itself, look at the parse errors I get and choose one or more of them to fix.

> P.S. We've reached 5,000 lines of code and 90,000 words in the Readme files. We must be nearly there! Next step