Part 36: `break` and `continue`

A while back, when I wrote another simple compiler for a typeless language, I didn't use an abstract syntax tree. This made it awkward to add the `break` and `continue` keywords to the language.

Here, we do have an AST tree for each function. This makes it much easier to implement `break` and `continue`. I'll outline the reasons for this below.

Adding the `break` and `continue`

Unsurprisingly, we have two new tokens T_BREAK and T_CONTINUE, and the scanner code in `scan.c` recognises the `break` and `continue` keywords. As always, browse the code to see how this is done.

New AST Node Types

We also have two new AST node types in `defs.h`: A_BREAK and A_CONTINUE. When we parse a `break` keyword, we can generate an A_BREAK AST leaf; ditto an A_CONTINUE leaf for the `continue` keyword.

Then, when we walk the AST to generate the assembly code, when we encounter an A_BREAK node, we need to generate an assembly jump to the label at the end of the loop that we are currently in. And for A_CONTINUE, we jump to the label just before the loop condition is evaluated.

Now, how do we know which loop we are in?

Tracking the Most Recent Loop

Loops can be nested, and so there can be any number of loop labels in use at any point. This is what I found difficult when I wrote my previous compiler. Now that we have an AST which we traverse recursively, we can pass the details of the latest loop's labels down to our children in the AST tree.

We already do this sort of thing to get to the end of an 'if' or 'while' statement. Here's some of the code for generating the assembly for 'if' from `gen.c` :

```
// Generate the code for an IF statement
// and an optional ELSE clause
static int genIF(struct ASTnode *n) {
  int Lfalse, Lend;

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

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

The left-hand AST child is the one that evaluates the 'if' statement's condition, so it needs access to the label that we have just generated. So when we generate this child's assembly output with `genAST()` , we also pass in the label's details.

For loops, we need to pass to `genAST()` the label which is at the loop's end and also the label just before the code that evaluates the loop's condition. To this end, I've changed the interface to `genAST()` :

```
int genAST(struct ASTnode *n, int iflabel, int looptoplabel,
           int loopendlabel, int parentASTop);
```

We keep the existing `iflabel` and augment this with the two loop labels. Now we need to pass to `genAST()` the labels that are generated for each loop. So, in the code to generate the 'while' loop code:

```
static int genWHILE(struct ASTnode *n) {
  int Lstart, Lend;

  // Generate the start and end labels
  Lstart = genlabel();
  Lend = genlabel();

  // Generate the condition code followed
  // by a jump to the end label.
  genAST(n->left, Lend, Lstart, Lend, n->op);

  // Generate the compound statement for the body
  genAST(n->right, NOLABEL, Lstart, Lend, n->op);
  ...
}
```

`genAST()` is Recursive

Now, what about nested loops? Consider the code:

```
L1:
  while (x < 10) {
    if (x == 6) break;
L2:
    while (y < 10) {
      if (y == 6) break;
      y++;
    }
L3:
    x++;
  }
L4:
```

the `if (y == 6) break` should leave the inner loop and jump to the `x++` code (i.e. L3), and the `if (x == 6) break;` code should leave the outer loop and jump to label L4.

This works because `genAST()` calls `genWHILE()` for the outer loop. This calls `genAST(L1, L4)` so that the first `break` sees these loop labels. Then, when we hit the second loop, `genWHILE()` is called again. It generates new loop labels and calls `genAST(L2, L3)` to generate the inner loop code. Thus, the second `break` sees the L2 and L3 labels, not the L1 and L4 labels.

Finally, once the inner compound statement is generated, the inside `genAST()` returns, and get back to the code which sees L1 and L4 as the loop labels.

Implications of the Above

What this means, in terms of implementation, is that anywhere that something calls `genAST()` (including itself), and we could be in a loop, then the current loop labels must get propagated down to the children involved.

We have already seen the change to `genWHILE()` to pass to `genAST()` the new loop labels. Let's look at where else we need to propagate loop labels.

When I first implemented `break`, I wrote this test program

```
int main() {
  int x;
  x = 0;
  while (x < 100) {
    printf("%d\n", x);
    if (x == 14) { break; }
    x = x + 1;
  }
  printf("Done\n");
```

and generated the assembly for it. The `break` was being turned into a jump to label L0, i.e. the loop's end label wasn't getting to the code dealing with `break`. Looking at a stack trace for the compiler, I realised that:

- The `genAST()` for the function called
- `genWHILE()` for the loop which generated the labels and passed them to
- `genAST()` for the loop body, which called
- `genIF()` which passed **no** labels in to
- `genAST()` for the 'if' body. Hence, the `break` never saw the labels.

So I also had to modify the argument list for `genIF()`:

```
static int genIF(struct ASTnode *n, int looptoplabel, int loopendlabel);
```

I won't go through all the code in `gen.c`, but open up the file in an editor or text viewer and look for all the `genAST()` calls to see where the loop labels do get propagated.

Finally, we actually do need to generate the assembly code for `break` and `continue`. Here is the code to do it in `genAST()` in `gen.c`:

```
    case A_BREAK:
      cgjump(loopendlabel);
      return (NOREG);
    case A_CONTINUE:
      cgjump(looptoplabel);
      return (NOREG);
```

Parsing break and continue

This time around I covered the code generation side before the parsing, but now it's time to get to the parsing of these new keywords. Luckily the syntax is either `break ;` or `continue ;`. So it would seem that they should be easy to parse. There is, of course, a small wrinkle.

We parse individual statements in `single_statement()` in `stmt.c`, so the change is small:

```
    case T_BREAK:
      return (break_statement());
    case T_CONTINUE:
      return (continue_statement());
```

with a slight change in `compound_statement()` to ensure that the statement is followed by a semicolon:

```
compound_statement(void) {
  struct ASTnode *left = NULL;
  struct ASTnode *tree;

  ...
  while (1) {
    // Parse a single statement
    tree = single_statement();

    // Some statements must be followed by a semicolon
    if (tree != NULL && (tree->op == A_ASSIGN || tree->op == A_RETURN
                         || tree->op == A_FUNCCALL || tree->op == A_BREAK
                         || tree->op == A_CONTINUE))
      semi();
    ...
}
```

Now the wrinkle. This following program is not legal:

```
int main() {
  break;
}
```

as there is no loop to break out of. We need to track the depth of the loops we are parsing, and only allow a `break` or `continue` statement when the depth is not zero. Thus, the functions to parse these keywords look like:

```
// Parse a break statement and return its AST
static struct ASTnode *break_statement(void) {

  if (Looplevel == 0)
    fatal("no loop to break out from");
  scan(&Token);
  return (mkastleaf(A_BREAK, 0, NULL, 0));
}

// continue_statement: 'continue' ;
//
// Parse a continue statement and return its AST
static struct ASTnode *continue_statement(void) {

  if (Looplevel == 0)
    fatal("no loop to continue to");
  scan(&Token);
  return (mkastleaf(A_CONTINUE, 0, NULL, 0));
}
```

Loop Levels

We are going to need a `Looplevel` variable to track the level of the loops being parsed. This is in `data.h`:

```
extern_ int Looplevel;                  // Depth of nested loops
```

We need to set the level up as required. Each time we start a new function, the level is set to zero (in `decl.c`):

```
// Parse the declaration of function.
struct ASTnode *function_declaration(int type) {
  ...
  // Get the AST tree for the compound statement and mark
```

```
    // that we have parsed no loops yet
    Looplevel= 0;
    tree = compound_statement();
    ...
  }
```

Now, each time we parse a loop, we increment the loop level for the loop's body (in `stmt.c`):

```
  // Parse a WHILE statement and return its AST
  static struct ASTnode *while_statement(void) {
    ...
    // Get the AST for the compound statement.
    // Update the loop depth in the process
    Looplevel++;
    bodyAST = compound_statement();
    Looplevel--;
    ...
  }

  // Parse a FOR statement and return its AST
  static struct ASTnode *for_statement(void) {
    ...
    // Get the compound statement which is the body
    // Update the loop depth in the process
    Looplevel++;
    bodyAST = compound_statement();
    Looplevel--;
    ...
  }
```

And this gives us the ability to determine if we are inside a loop or not inside a loop.

The Test Code

Here is the test code, `tests/input71.c`:

```
  #include <stdio.h>

  int main() {
    int x;
    x = 0;
    while (x < 100) {
      if (x == 5) { x = x + 2; continue; }
      printf("%d\n", x);
      if (x == 14) { break; }
      x = x + 1;
    }
    printf("Done\n");
    return (0);
  }
```

As I still haven't solved the "dangling else" problem, the `break` statement has to enclosed in '{' ... '}' to make it into a compound statement. Apart from that, the code works as expected:

```
0
1
2
3
4
7
8
9
10
11
12
13
14
Done
```

Conclusion and What's Next

I knew that adding support for `break` and `continue` was going to be easier than it was for my earlier compiler, because of the AST. However, there were still some minor issues and wrinkles that we had to deal with in the process of implementing them.

Now that we have the `break` keyword in the language, I will attempt to add `switch` statements in the next part of our compiler writing journey. This is going to require the addition of switch jump tables, and I know this is going to be complicated. So get ready for an interesting next step. Next step