# Part 8: If Statements

Now that we can compare values, it's time to add IF statements to our language. So, firstly, let's look at the general syntax of IF statements and how they get converted into assembly language.

## The IF Syntax

The IF statement syntax is:

```
    if (condition is true)
      perform this first block of code
    else
      perform this other block of code
```

Now, how is this normally converted into assembly language? It turns out that we do the opposite comparison and jump/branch if the opposite comparison is true:

```
        perform the opposite comparison
        jump to L1 if true
        perform the first block of code
        jump to L2
L1:
        perform the other block of code
```

```
L2:
```

where `L1` and `L2` are assembly language labels.

## Generating The Assembly in Our Compiler

Right now, we output code to set a register based on a comparison, e.g.

```
int x; x= 7 < 9;         From input04
```

becomes

```
        movq    $7, %r8
        movq    $9, %r9
        cmpq    %r9, %r8
        setl    %r9b        Set if less than
        andq    $255,%r9
```

But for an IF statement, we need to jump on the opposite comparison:

```
if (7 < 9)
```

should become:

```
        movq    $7, %r8
        movq    $9, %r9
        cmpq    %r9, %r8
        jge     L1          Jump if greater then or equal to
        ....
L1:
```

So, I've implemented IF statements in this part of our journey. As this is a working project, I did have to undo a few things and refactor them as part of the journey. I'll try to cover the changes as well as the additions along the way.

## New Tokens and the Dangling Else

We are going to need a bunch of new tokens in our language. I also (for now) want to avoid the dangling else problem. To that end, I've changed the grammar so that all groups of statements are wrapped around '{' ... '}' curly brackets; I called such a grouping a "compound statement". We also need '(' ... ')' parentheses to hold the IF expression, plus keywords 'if' and 'else'. Thus, the new tokens are (in `defs.h`):

```
T_LBRACE, T_RBRACE, T_LPAREN, T_RPAREN,
// Keywords
..., T_IF, T_ELSE
```

## Scanning the Tokens

The single-character tokens should be obvious and I won't give the code to scan them. The keywords should also be pretty obvious, but I'll give the scanning code from `keyword()` in `scan.c`:

```
switch (*s) {
  case 'e':
    if (!strcmp(s, "else"))
      return (T_ELSE);
    break;
  case 'i':
    if (!strcmp(s, "if"))
      return (T_IF);
    if (!strcmp(s, "int"))
      return (T_INT);
    break;
  case 'p':
    if (!strcmp(s, "print"))
      return (T_PRINT);
    break;
}
```

## The New BNF Grammar

Our grammar is starting to get big, so I've rewritten it somewhat:

```
compound_statement: '{' '}'          // empty, i.e. no statement
     |      '{' statement '}'
```

```
      |      '{' statement statements '}'
      ;

statement: print_statement
      |     declaration
      |     assignment_statement
      |     if_statement
      ;

print_statement: 'print' expression ';'  ;

declaration: 'int' identifier ';'  ;

assignment_statement: identifier '=' expression ';'   ;

if_statement: if_head
      |        if_head 'else' compound_statement
      ;

if_head: 'if' '(' true_false_expression ')' compound_statement  ;

identifier: T_IDENT ;
```

I've left out the definition of `true_false_expression`, but at some point when we've added a few more operators I'll add it in.

Note the grammar for the IF statement: it's either an `if_head` (with no 'else' clause), or an `if_head` followed by a 'else' and a `compound_statement`.

I've separated out all the different statement types to have their own non-terminal name. Also, the previous `statements` non-terminal is now the `compound_statement` non-terminal, and this requires '{' ... '}' around the statements.

This means that the `compound_statement` in the head is surrounded by '{' ... '}' and so is any `compound_statement` after the 'else' keyword. So if we have nested IF statements, they have to look like:

```
if (condition1 is true) {
  if (condition2 is true) {
    statements;
  } else {
    statements;
  }
} else {
  statements;
}
```

and there is no ambiguity about which 'if' each 'else' belongs to. This solves the dangling else problem. Later on, I'll make the '{' ... '}' optional.

## Parsing Compound Statements

The old `void statements()` function is now `compound_statement()` and looks like this:

```
// Parse a compound statement
// and return its AST
struct ASTnode *compound_statement(void) {
  struct ASTnode *left = NULL;
  struct ASTnode *tree;

  // Require a left curly bracket
  lbrace();

  while (1) {
    switch (Token.token) {
      case T_PRINT:
        tree = print_statement();
        break;
      case T_INT:
        var_declaration();
        tree = NULL;            // No AST generated here
        break;
      case T_IDENT:
        tree = assignment_statement();
        break;
      case T_IF:
        tree = if_statement();
        break;
    case T_RBRACE:
        // When we hit a right curly bracket,
        // skip past it and return the AST
        rbrace();
        return (left);
      default:
        fatald("Syntax error, token", Token.token);
    }

    // For each new tree, either save it in left
    // if left is empty, or glue the left and the
    // new tree together
    if (tree) {
      if (left == NULL)
        left = tree;
      else
```

```
      left = mkastnode(A_GLUE, left, NULL, tree, 0);
    }
  }
```

Firstly, note that the code forces the parser to match the '{' at the start of the compound statement with `lbrace()`, and we can only exit when we've matched the ending '}' with `rbrace()`.

Secondly, note that `print_statement()`, `assignment_statement()` and `if_statement()` all return an AST tree, as does `compound_statement()`. In our old code, `print_statement()` itself called `genAST()` to evaluate the expression, followed by a call to `genprintint()`. Similarly, `assignment_statement()` also called `genAST()` to do the assignment.

Well, this means that we have AST trees over here, and others over there. It makes some sense to generate just a single AST tree, and call `genAST()` once to generate the assembly code for it.

This isn't mandatory. For example, SubC only generates ASTs for expressions. For the structural parts of the language, like statements, SubC makes specific calls to the code generator as I was doing in the previous versions of the compiler.

I've decided to, for now, generate a single AST tree for the whole input with the parser. Once the input has been parsed, the assembly output can be generated from the one AST tree.

Later on, I'll probably generate an AST tree for each function. Later.

## Parsing the IF Grammar

Because we are a recursive descent parser, parsing the IF statement is not too bad:

```
// Parse an IF statement including
// any optional ELSE clause
// and return its AST
struct ASTnode *if_statement(void) {
  struct ASTnode *condAST, *trueAST, *falseAST = NULL;

  // Ensure we have 'if' '('
  match(T_IF, "if");
  lparen();

  // Parse the following expression
  // and the ')' following. Ensure
  // the tree's operation is a comparison.
```

```
  condAST = binexpr(0);

  if (condAST->op < A_EQ || condAST->op > A_GE)
    fatal("Bad comparison operator");
  rparen();

  // Get the AST for the compound statement
  trueAST = compound_statement();

  // If we have an 'else', skip it
  // and get the AST for the compound statement
  if (Token.token == T_ELSE) {
    scan(&Token);
    falseAST = compound_statement();
  }
  // Build and return the AST for this statement
  return (mkastnode(A_IF, condAST, trueAST, falseAST, 0));
}
```

Right now, I don't want to deal with input like `if (x-2)`, so I've limited the binary expression from `binexpr()` to have a root which is one of the six comparison operators A_EQ, A_NE, A_LT, A_GT, A_LE or A_GE.

## The Third Child

I nearly smuggled something past you without properly explaining it. In the last line of `if_statement()` I build an AST node with:

```
mkastnode(A_IF, condAST, trueAST, falseAST, 0);
```

That's *three* AST sub-trees! What's going on here? As you can see, the IF statement will have three children:

- the sub-tree that evaluates the condition
- the compound statement immediately following
- the optional compound statement after the 'else' keyword

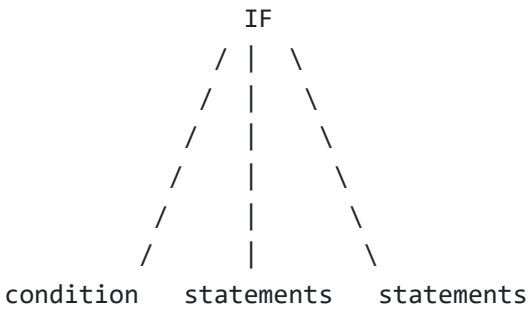
So we now need an AST node structure with three children (in `defs.h`):

```
// AST node types.
enum {
  ...
  A_GLUE, A_IF
```

```
};

// Abstract Syntax Tree structure
struct ASTnode {
  int op;                       // "Operation" to be performed on this tree
  struct ASTnode *left;         // Left, middle and right child trees
  struct ASTnode *mid;
  struct ASTnode *right;
  union {
    int intvalue;               // For A_INTLIT, the integer value
    int id;                     // For A_IDENT, the symbol slot number
  } v;
};
```

Thus, an A_IF tree looks like this:




## Glue AST Nodes

There is also a new A_GLUE AST node type. What is this used for? We now build a single AST tree with lots of statements, so we need a way to glue them together.

Review the end of the `compound_statement()` loop code:

```
    if (left != NULL)
      left = mkastnode(A_GLUE, left, NULL, tree, 0);
```

Each time we get a new sub-tree, we glue it on to the existing tree. So, for this sequence of statements:

```
  stmt1;
  stmt2;
  stmt3;
  stmt4;
```

we end up with:

```
          A_GLUE
           /  \
       A_GLUE stmt4
         /  \
     A_GLUE stmt3
       /  \
  stmt1  stmt2
```

And, as we traverse the tree depth-first left to right, this still generates the assembly code in the correct order.

## The Generic Code Generator

Now that our AST nodes have multiple children, our generic code generator is going to become a bit more complicated. Also, for the comparison operators, we need to know if we are doing the compare as part of an IF statement (jump on the opposite comparison) or a normal expression (set register to 1 or 0 on the normal comparison).

To this end, I've modified `getAST()` so that we can pass in the parent AST nodes operation:

```
// Given an AST, the register (if any) that holds
// the previous rvalue, and the AST op of the parent,
// generate assembly code recursively.
// Return the register id with the tree's final value
int genAST(struct ASTnode *n, int reg, int parentASTop) {
   ...
}
```

### Dealing with Specific AST Nodes

The code in `genAST()` now has to deal with specific AST nodes:

```
  // We now have specific AST node handling at the top
  switch (n->op) {
    case A_IF:
      return (genIFAST(n));
    case A_GLUE:
      // Do each child statement, and free the
      // registers after each child
      genAST(n->left, NOREG, n->op);
```

```
    genfreeregs();
    genAST(n->right, NOREG, n->op);
    genfreeregs();
    return (NOREG);
  }
```

If we don't return, we carry on to do the normal binary operator AST nodes, with one exception: the comparison nodes:

```
    case A_EQ:
    case A_NE:
    case A_LT:
    case A_GT:
    case A_LE:
    case A_GE:
      // If the parent AST node is an A_IF, generate a compare
      // followed by a jump. Otherwise, compare registers and
      // set one to 1 or 0 based on the comparison.
      if (parentASTop == A_IF)
        return (cgcompare_and_jump(n->op, leftreg, rightreg, reg));
      else
        return (cgcompare_and_set(n->op, leftreg, rightreg));
```

I'll cover the new functions `cgcompare_and_jump()` and `cgcompare_and_set()` below.

## Generating the IF Assembly Code

We deal with the A_IF AST node with a specific function, along with a function to generate new label numbers:

```
// Generate and return a new label number
static int label(void) {
  static int id = 1;
  return (id++);
}

// Generate the code for an IF statement
// and an optional ELSE clause
static int genIFAST(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.
  // When there is no ELSE clause, Lfalse _is_
```

```
  // the ending label!
  Lfalse = label();
  if (n->right)
    Lend = label();

  // Generate the condition code followed
  // by a zero jump to the false label.
  // We cheat by sending the Lfalse label as a register.
  genAST(n->left, Lfalse, n->op);
  genfreeregs();

  // Generate the true compound statement
  genAST(n->mid, NOREG, n->op);
  genfreeregs();

  // If there is an optional ELSE clause,
  // generate the jump to skip to the end
  if (n->right)
    cgjump(Lend);

  // Now the false label
  cglabel(Lfalse);

  // Optional ELSE clause: generate the
  // false compound statement and the
  // end label
  if (n->right) {
    genAST(n->right, NOREG, n->op);
    genfreeregs();
    cglabel(Lend);
  }

  return (NOREG);
}
```

Effectively, the code is doing:

```
  genAST(n->left, Lfalse, n->op);       // Condition and jump to Lfalse
  genAST(n->mid, NOREG, n->op);         // Statements after 'if'
  cgjump(Lend);                         // Jump to Lend
  cglabel(Lfalse);                      // Lfalse: label
  genAST(n->right, NOREG, n->op);       // Statements after 'else'
  cglabel(Lend);                        // Lend: label
```

## The x86-64 Code Generation Functions

So we now have a few new x86-64 code generation functions. Some of these replace the six `cgXXX()` comparison functions we created in the last part of the journey.

For the normal comparison functions, we now pass in the AST operation to choose the relevant x86-64 `set` instruction:

```
// List of comparison instructions,
// in AST order: A_EQ, A_NE, A_LT, A_GT, A_LE, A_GE
static char *cmplist[] =
  { "sete", "setne", "setl", "setg", "setle", "setge" };

// Compare two registers and set if true.
int cgcompare_and_set(int ASTop, int r1, int r2) {

  // Check the range of the AST operation
  if (ASTop < A_EQ || ASTop > A_GE)
    fatal("Bad ASTop in cgcompare_and_set()");

  fprintf(Outfile, "\tcmpq\t%s, %s\n", reglist[r2], reglist[r1]);
  fprintf(Outfile, "\t%s\t%s\n", cmplist[ASTop - A_EQ], breglist[r2]);
  fprintf(Outfile, "\tmovzbq\t%s, %s\n", breglist[r2], reglist[r2]);
  free_register(r1);
  return (r2);
}
```

I've also found an x86-64 instruction `movzbq` that moves the lowest byte from one register and extends it to fit into a 64-bit register. I'm using that now instead of the `and $255` in the old code.

We need a functions to generate a label and to jump to it:

```
// Generate a label
void cglabel(int l) {
  fprintf(Outfile, "L%d:\n", l);
}

// Generate a jump to a label
void cgjump(int l) {
  fprintf(Outfile, "\tjmp\tL%d\n", l);
}
```

Finally, we need a function to do a comparison and to jump based on the opposite comparison. So, using the AST comparison node type, we do the opposite comparison:

```
// List of inverted jump instructions,
// in AST order: A_EQ, A_NE, A_LT, A_GT, A_LE, A_GE
static char *invcmplist[] = { "jne", "je", "jge", "jle", "jg", "jl" };

// Compare two registers and jump if false.
int cgcompare_and_jump(int ASTop, int r1, int r2, int label) {

  // Check the range of the AST operation
  if (ASTop < A_EQ || ASTop > A_GE)
    fatal("Bad ASTop in cgcompare_and_set()");

  fprintf(Outfile, "\tcmpq\t%s, %s\n", reglist[r2], reglist[r1]);
  fprintf(Outfile, "\t%s\tL%d\n", invcmplist[ASTop - A_EQ], label);
  freeall_registers();
  return (NOREG);
}
```

## Testing the IF Statements

Do a `make test` which compiles the `input05` file:

```
{
  int i; int j;
  i=6; j=12;
  if (i < j) {
    print i;
  } else {
    print j;
  }
}
```

Here's the resulting assembly output:

```
        movq    $6, %r8
        movq    %r8, i(%rip)    # i=6;
        movq    $12, %r8
        movq    %r8, j(%rip)    # j=12;
        movq    i(%rip), %r8
        movq    j(%rip), %r9
        cmpq    %r9, %r8        # Compare %r8-%r9, i.e. i-j
        jge     L1              # Jump to L1 if i >= j
```

```
        movq    i(%rip), %r8
        movq    %r8, %rdi       # print i;
        call    printint
        jmp     L2              # Skip the else code
L1:
        movq    j(%rip), %r8
        movq    %r8, %rdi       # print j;
        call    printint
L2:
```

And, of course, `make test` shows:

```
cc -o comp1 -g cg.c decl.c expr.c gen.c main.c misc.c
      scan.c stmt.c sym.c tree.c
./comp1 input05
cc -o out out.s
./out
6                   # As 6 is less than 12
```

## Conclusion and What's Next

We've added our first control structure to our language with the IF statement. I had to rewrite a few existing things along the way and, given I don't have a complete architectural plan in my head, I'll likely have to rewrite more things in the future.

The wrinkle for this part of the journey was that we had to perform the opposite comparison for the IF decision than what we would do for the normal comparison operators. My solution was to inform each AST node of the node type of their parent; the comparison nodes can now see if the parent is an A_IF node or not.

I know that Nils Holm chose a different approach when he was implementing SubC, so you should look at his code just to see this different solution to the same problem.

In the next part of our compiler writing journey, we will add another control structure: the WHILE loop. Next step