Part 12: Types, part 1

I've just begun the process to add types to our compiler. Now, I should warn you that this is new to me, as in my previous compiler I only had `int` s. I've resisted the urge to look at the SubC source code for ideas. Thus, I'm striking out on my own and it's likely that I will have to redo some of the code as I deal with the greater issues involving types.

What Types for Now?

I'll start with `char` and `int` for our global variables. We've already added the `void` keyword for functions. In the next step I will add function return values. So, for now, `void` exists but I'm not fully dealing with it.

Obviously, `char` has a much more limited range of values that `int` . Like SubC, I'm going to use the range 0 .. 255 for `char` s and a range of signed values for `int` s.

This means that we can widen `char` values to become `int` s, but we must warn the developer if they try to narrow `int` values down to a `char` range.

New Keywords and Tokens

There is only the new 'char' keyword and the T_CHAR token. Nothing exciting here.

Expression Types

From now on, every expression has a type. This includes:

- integer literals, e.g 56 is an `int`
- maths expressions, e.g. 45 - 12 is an `int`
- variables, e.g. if we declared `x` as a `char`, then it's *rvalue* is a `char`

We are going to have to track the type of each expression as we evaluate it, to ensure we can widen it as required or refuse to narrow it if necessary.

In the SubC compiler, Nils created a single *lvalue* structure. A pointer to this single stucture was passed around in the recursive parser to track the type of any expression at a point in its parsing.

I've taken a different tack. I've modified our Abstract Syntax Tree node to have a `type` field which holds the type of the tree at that point. In `defs.h`, here are the types I've created so far:

```
// Primitive types
enum {
  P_NONE, P_VOID, P_CHAR, P_INT
};
```

I've called them *primitive* types, as Nils did in SubC, because I can't think of a better name for them. Data types, perhaps? The P_NONE value indicates that the AST node *doesn't* represent an expression and has no type. An example is the A_GLUE node type which glues statements together: once the left-hand statement is generated, there is no type to speak of.

If you look in `tree.c`, you will see that the functions to build AST nodes have been modified to also assign to the `type` field in the new AST node structure (in `defs.h`):

```
struct ASTnode {
  int op;                       // "Operation" to be performed on this tree
  int type;                     // Type of any expression this tree generates
  ...
};
```

Variable Declarations and Their Types

We now have at least two ways to declare global variables:

```
int x; char y;
```

We'll need to parse this, yes. But first, how do we record the type for each variable? We need to modify the `symtable` structure. I've also added the details of the "structural type" of the symbol which I'll use in the future (in `defs.h`):

```
// Structural types
enum {
  S_VARIABLE, S_FUNCTION
};

// Symbol table structure
struct symtable {
  char *name;                   // Name of a symbol
  int type;                     // Primitive type for the symbol
  int stype;                    // Structural type for the symbol
};
```

There's new code in `newglob()` in `sym.c` to initialise these new fields:

```
int addglob(char *name, int type, int stype) {
  ...
  Gsym[y].type = type;
  Gsym[y].stype = stype;
  return (y);
}
```

Parsing Variable Declarations

It's time to separate out the parsing of the type from the parsing of the variable itself. So, in `decl.c` we now have:

```
// Parse the current token and
// return a primitive type enum value
int parse_type(int t) {
  if (t == T_CHAR) return (P_CHAR);
  if (t == T_INT)  return (P_INT);
```

```
  if (t == T_VOID) return (P_VOID);
  fatald("Illegal type, token", t);
}

// Parse the declaration of a variable
void var_declaration(void) {
  int id, type;

  // Get the type of the variable, then the identifier
  type = parse_type(Token.token);
  scan(&Token);
  ident();
  id = addglob(Text, type, S_VARIABLE);
  genglobsym(id);
  semi();
}
```

Dealing with Expression Types

All of the above is the easy part done! We now have:

- a set of three types: `char`, `int` and `void`,
- parsing of variable declarations to find their type,
- capture of each variable's type in the symbol table, and
- storage of the type of an expression in each AST node

Now we need to actually fill in the type in the AST nodes that we build. Then we have to decide when to widen types and/or reject type clashes. Let's get on with the job!

Parsing Primary Terminals

We'll start with the parsing of integer literal values and variable identifiers. One wrinkle is that we want to be able to do:

```
  char j; j= 2;
```

But if we mark the `2` as a P_INT, then we won't be able to narrow the value when we try to store it in the P_CHAR `j` variable. For now, I've added some semantic code to keep small integer literal values as P_CHARs:

```
// Parse a primary factor and return an
// AST node representing it.
```

```c
static struct ASTnode *primary(void) {
  struct ASTnode *n;
  int id;

  switch (Token.token) {
    case T_INTLIT:
      // For an INTLIT token, make a leaf AST node for it.
      // Make it a P_CHAR if it's within the P_CHAR range
      if ((Token.intvalue) >= 0 && (Token.intvalue < 256))
        n = mkastleaf(A_INTLIT, P_CHAR, Token.intvalue);
      else
        n = mkastleaf(A_INTLIT, P_INT, Token.intvalue);
      break;

    case T_IDENT:
      // Check that this identifier exists
      id = findglob(Text);
      if (id == -1)
        fatals("Unknown variable", Text);

      // Make a leaf AST node for it
      n = mkastleaf(A_IDENT, Gsym[id].type, id);
      break;

    default:
      fatald("Syntax error, token", Token.token);
  }

  // Scan in the next token and return the leaf node
  scan(&Token);
  return (n);
}
```

Also note that, for identifiers, we can easily get their type details from the global symbol table.

Building Binary Expressions: Comparing Types

As we build maths expressions with our binary maths operators, we will have a type from the left-hand child and a type from the right-hand child. Here is where we are going to have to either widen, do nothing, or reject the expression if the two types are incompatible.

For now, I have a new file `types.c` with a function that compares the types on either side. Here's the code:

```c
// Given two primitive types, return true if they are compatible,
// false otherwise. Also return either zero or an A_WIDEN
// operation if one has to be widened to match the other.
// If onlyright is true, only widen left to right.
int type_compatible(int *left, int *right, int onlyright) {

  // Voids not compatible with anything
  if ((*left == P_VOID) || (*right == P_VOID)) return (0);

  // Same types, they are compatible
  if (*left == *right) { *left = *right = 0; return (1);
  }

  // Widen P_CHARs to P_INTs as required
  if ((*left == P_CHAR) && (*right == P_INT)) {
    *left = A_WIDEN; *right = 0; return (1);
  }
  if ((*left == P_INT) && (*right == P_CHAR)) {
    if (onlyright) return (0);
    *left = 0; *right = A_WIDEN; return (1);
  }
  // Anything remaining is compatible
  *left = *right = 0;
  return (1);
}
```

There's a fair bit going on here. Firstly, if both types are the same we can simply return True. Anything with a P_VOID cannot be mixed with another type.

If one side is a P_CHAR and the other is a P_INT, we can widen the result to a P_INT. The way I do this is to modify the type information that comes in and I replace it either with zero (do nothing), or a new AST node type A_WIDEN. This means: widen the more narrow child's value to be as wide as the wider child's value. We'll see this in operation soon.

There is one extra argument `onlyright`. I use this when we get to A_ASSIGN AST nodes where we are assigning the left-child's expression to the variable *lvalue* on the right. If this is set, don't let a P_INT expression be transferred to a P_CHAR variable

Finally, for now, let any other type pairs through.

I think I can guarantee that this will need to be changed once we bring in arrays and pointers. I also hope I can find a way to make the code simpler and more elegant. But it will do for now.

Using `type_compatible()` in Expressions

I've used `type_compatible()` in three different places in this version of the compiler. We'll start with merging expressions with binary operators. I've modified the code in `binexpr()` in `expr.c` to do this:

```
    // Ensure the two types are compatible.
    lefttype = left->type;
    righttype = right->type;
    if (!type_compatible(&lefttype, &righttype, 0))
      fatal("Incompatible types");

    // Widen either side if required. type vars are A_WIDEN now
    if (lefttype)
      left = mkastunary(lefttype, right->type, left, 0);
    if (righttype)
      right = mkastunary(righttype, left->type, right, 0);

    // Join that sub-tree with ours. Convert the token
    // into an AST operation at the same time.
    left = mkastnode(arithop(tokentype), left->type, left, NULL, right, 0);
```

We reject incompatible types. But, if `type_compatible()` returned non-zero `lefttype` or `righttype` values, these are actually the A_WIDEN value. We can use this to build a unary AST node with the narrow child as the child. When we get to the code generator, it will now know that this child's value has to be widened.

Now, where else do we need to widen expression values?

Using `type_compatible()` to Print Expressions

When we use the `print` keyword, we need to have an `int` expression for it to print. So we need to change `print_statement()` in `stmt.c`:

```
static struct ASTnode *print_statement(void) {
  struct ASTnode *tree;
  int lefttype, righttype;
  int reg;

  ...
  // Parse the following expression
  tree = binexpr(0);

  // Ensure the two types are compatible.
```

```
  lefttype = P_INT; righttype = tree->type;
  if (!type_compatible(&lefttype, &righttype, 0))
    fatal("Incompatible types");

  // Widen the tree if required.
  if (righttype) tree = mkastunary(righttype, P_INT, tree, 0);
```

Using `type_compatible()` to Assign to a Variable

This is the last place where we need to check types. When we assign to a variable, we need to ensure that we can widen the right-hand side expression. We've got to reject any attempt to store a wide type into a narrow variable. Here is the new code in `assignment_statement()` in `stmt.c`:

```
static struct ASTnode *assignment_statement(void) {
  struct ASTnode *left, *right, *tree;
  int lefttype, righttype;
  int id;

  ...
  // Make an lvalue node for the variable
  right = mkastleaf(A_LVIDENT, Gsym[id].type, id);

  // Parse the following expression
  left = binexpr(0);

  // Ensure the two types are compatible.
  lefttype = left->type;
  righttype = right->type;
  if (!type_compatible(&lefttype, &righttype, 1))  // Note the 1
    fatal("Incompatible types");

  // Widen the left if required.
  if (lefttype)
    left = mkastunary(lefttype, right->type, left, 0);
```

Note the 1 at the the end to this call to `type_compatible()`. This enforces the semantics that we cannot save a wide value to a narrow variable.

Given all of the above, we now can parse a few types and enforce some sensible language semantics: widen values where possible, prevent type narrowing and prevent unsuitable type clashes. Now we move to the code generation side of things.

The Changes to x86-64 Code Generation

Our assembly output is register based and essentially they are fixed in size. What we can influence is:

- the size of the memory locations to store variables, and
- how much of a register is used hold data, e.g. one byte for characters, eight bytes for a 64-bit integer.

I'll start with the x86-64 specific code in `cg.c`, and then I'll show how this is used in the generic code generator in `gen.c`.

Let's start with generating the storage for variables.

```
// Generate a global symbol
void cgglobsym(int id) {
  // Choose P_INT or P_CHAR
  if (Gsym[id].type == P_INT)
    fprintf(Outfile, "\t.comm\t%s,8,8\n", Gsym[id].name);
  else
    fprintf(Outfile, "\t.comm\t%s,1,1\n", Gsym[id].name);
}
```

We extract the type from the variable slot in the symbol table and choose to allocate 1 or 8 bytes for it depending on this type. Now we need to load the value into a register:

```
// Load a value from a variable into a register.
// Return the number of the register
int cgloadglob(int id) {
  // Get a new register
  int r = alloc_register();

  // Print out the code to initialise it: P_CHAR or P_INT
  if (Gsym[id].type == P_INT)
    fprintf(Outfile, "\tmovq\t%s(\%%rip), %s\n", Gsym[id].name, reglist[r]);
  else
    fprintf(Outfile, "\tmovzbq\t%s(\%%rip), %s\n", Gsym[id].name, reglist[r]);
  return (r);
```

The `movq` instruction moves eight bytes into the 8-byte register. The `movzbq` instruction zeroes the 8-byte register and then moves a single byte into it. This also implicitly widens the one byte value to eight bytes. Our storage function is similar:

```c
// Store a register's value into a variable
int cgstorglob(int r, int id) {
  // Choose P_INT or P_CHAR
  if (Gsym[id].type == P_INT)
    fprintf(Outfile, "\tmovq\t%s, %s(\%%rip)\n", reglist[r], Gsym[id].name);
  else
    fprintf(Outfile, "\tmovb\t%s, %s(\%%rip)\n", breglist[r], Gsym[id].name);
  return (r);
}
```

This time we have to use the "byte" name of the register and the `movb` instruction to move a single byte.

Luckily, the `cgloadglob()` function has already done the widening of P_CHAR variables. So this is the code for our new `cgwiden()` function:

```c
// Widen the value in the register from the old
// to the new type, and return a register with
// this new value
int cgwiden(int r, int oldtype, int newtype) {
  // Nothing to do
  return (r);
}
```

The Changes to The Generic Code Generation

With the above in place, there are only a few changes to the generic code generator in `gen.c`:

- The calls to `cgloadglob()` and `cgstorglob()` now take the symbol's slot number and not the symbol's name.
- Similarly, `genglobsym()` now receives the symbol's slot number and passes it on to `cgglobsym()`

The only major change is the code to deal with the new A_WIDEN AST node type. We don't need this node (as `cgwiden()` does nothing), but it's here for other hardware platforms:

```c
    case A_WIDEN:
      // Widen the child's type to the parent's type
      return (cgwiden(leftreg, n->left->type, n->type));
```

Testing the New Type Changes

Here is my test input file, `tests/input10`:

```
void main()
{
  int i; char j;

  j= 20; print j;
  i= 10; print i;

  for (i= 1;   i <= 5; i= i + 1) { print i; }
  for (j= 253; j != 2; j= j + 1) { print j; }
}
```

I check that we can assign to and print from `char` and `int` types. I also verify that, for `char` variables, we will overflow in the value sequence: 253, 254, 255, 0, 1, 2 etc.

```
$ make test
cc -o comp1 -g cg.c decl.c expr.c gen.c main.c misc.c scan.c
   stmt.c sym.c tree.c types.c
./comp1 tests/input10
cc -o out out.s
./out
20
10
1
2
3
4
5
253
254
255
0
1
```

Let's look at some of the assembly that was generated:

```
        .comm   i,8,8                   # Eight byte i storage
        .comm   j,1,1                   # One   byte j storage
        ...
        movq    $20, %r8
        movb    %r8b, j(%rip)           # j= 20
        movzbq  j(%rip), %r8
```

```
        movq    %r8, %rdi               # print j
        call    printint

        movq    $253, %r8
        movb    %r8b, j(%rip)           # j= 253
L3:
        movzbq  j(%rip), %r8
        movq    $2, %r9
        cmpq    %r9, %r8                # while j != 2
        je      L4
        movzbq  j(%rip), %r8
        movq    %r8, %rdi               # print j
        call    printint
        movzbq  j(%rip), %r8
        movq    $1, %r9                 # j= j + 1
        addq    %r8, %r9
        movb    %r9b, j(%rip)
        jmp     L3
```

Still not the most elegant assembly code, but it does work. Also, `$ make test` confirms that all the previous code examples still work.

Conclusion and What's Next

In the next part of our compiler writing journey, we will add function calls with one argument, and returning a value from a function. [Next step]