# A tiny GCC front end – Part 6

Jan 17, 2016 • Roger Ferrer Ibáñez • compilers, GCC • gcc, tiny

In part 5 we described the objects that we will need to semantically analyze a tiny program. In current part we will extend the parser of part 4 to do the semantic analysis and create the GENERIC trees.

## Semantic values

If you recheck part 4 you will see that several `parse_xxx` functions returned a false boolean value when there was a syntax error, true otherwise. If we are just checking if the input is syntactically valid this will do. But we want to compute something more interesting thus we need something a bit more useful. What if we were able to compute a value representing what the part of the language *does*. This set of values that are computed by a syntax rule are commonly called *semantic values*.

Tiny is a simple language that will require only a single semantic value: a `Tree` (recall that it is just a wrapper to GENERIC trees). If the parsing succeeds, the tree will express what the input does. If parsing fails it will simply return `error_mark_node`.

## Variable declaration

Let's recall the syntax of a variable declaration.

⟨declaration⟩ → **var** ⟨identifier⟩ **:** ⟨type⟩ **;**

Recall that a variable declaration statement adds a new mapping for the ⟨identifier⟩ in the topmost mapping of the scope. Let's see how we have to change `parse_variable_declaration` to do this.

```
1 Tree
2 Parser::parse_variable_declaration ()
3 {
4   if (!skip_token (Tiny::VAR))
5     {
6       skip_after_semicolon ();
7       return Tree::error ();
8     }
```

```
 9
10   const_TokenPtr identifier = expect_token (Tiny::IDENTIFIER);
11   if (identifier == NULL)
12     {
13       skip_after_semicolon ();
14       return Tree::error ();
15     }
16
17   if (!skip_token (Tiny::COLON))
18     {
19       skip_after_semicolon ();
20       return Tree::error ();
21     }
22
23   Tree type_tree = parse_type ();
24
25   if (type_tree.is_error ())
26     {
27       skip_after_semicolon();
28       return Tree::error ();
29     }
30
31   skip_token (Tiny::SEMICOLON);
32
33   if (scope.get_current_mapping ().get (identifier->get_str ()))
34     {
35       error_at (identifier->get_locus (),
36                 "variable '%s' already declared in this scope",
37                 identifier->get_str ().c_str ());
38       return Tree::error ();
39     }
40   SymbolPtr sym (new Symbol (identifier->get_str ()));
41   scope.get_current_mapping ().insert (sym);
42
43   Tree decl = build_decl (identifier->get_locus (), VAR_DECL,
44                           get_identifier (sym->get_name ().c_str ()),
45                           type_tree.get_tree ());
46
47   gcc_assert (!stack_var_decl_chain.empty ());
48   stack_var_decl_chain.back ().append (decl);
49
50   sym->set_tree_decl (decl);
51
52   Tree stmt
53     = build_tree (DECL_EXPR, identifier->get_locus (), void_type_node, decl);
54
55   return stmt;
56 }
```

We first parse the syntactic elements of a variable declaration. We skip the initial var in lines 4 to 8. In line 10 we keep the identifier token because it will be used later. We skip the colon in lines 17 to

21. In line 23 we parse the type (by calling `parse_type`, more on this later) and finally in line 31 we skip the semicolon.

Now the semantic checks of a variable declaration can start. In line 33, we check if the current mapping of the scope already contains a mapping for the identifier. If there is such a mapping, this is an error and we give up, otherwise we create a new symbol (line 39) using the given identifier and we insert it into the current mapping (line 40).

Now we need to create some GENERIC for this new variable declaration (line 43). It will have a tree code of `VAR_DECL`. The first operand of that tree is an `IDENTIFIER_NODE` for the identifier itself. These trees are shared in GENERIC: two identical identifiers will use the same tree. For this reason we need to request an `IDENTIFIER_NODE` rather than creating it manually. We do that calling the (GCC-provided) function `get_identifier` (line 44). The second operand that we will need is the type of the declaration. This was obtained in an earlier call to `parse_type`. Note that we are calling the (GCC-provided) function `build_decl`. This is so because there is an extra step (setting some internal type and operation mode of the declaration) that has to be performed for a `VAR_DECL`. Function `build_decl` takes care of that for us and it is in practice like calling `build2_loc`.

In line 50 we associate the new `Symbol` with the `VAR_DECL` we have created. We do this because every time we need to refer to an existing variable in GENERIC we will need to use a `VAR_DECL`. But it cannot be a new `VAR_DECL` every time since this would mean a new variable with the same name. So we just keep a single `VAR_DECL` in a `Symbol` so we can reuse it as many times as needed.

The `VAR_DECL` is also kept in the top list of the stack `stack_var_decl_chain`. We will need this later when we talk about blocks.

# Types

A variable declaration has a type.

⟨type⟩ → **`int`** | **`float`**

In part 5 we classified nodes in three kinds: declarations, expressions and types. In GENERIC, types are represented obviously as trees. Some basic types have dedicated trees, other may have to be constructed. For tiny we will use `integer_type_node`, `float_type_node`, `boolean_type_node` and `void_type_node`. The last one will be used to designate that the computed value of an expression is of no interest (i.e. the expression is computed only for its side-effects).

Our `parse_type` will return either `integer_type_node` or `float_type_node` as we do not allow other types in a variable declaration.

```
Tree
Parser::parse_type ()
{
  const_TokenPtr t = lexer.peek_token ();
```

```
  switch (t->get_id ())
    {
    case Tiny::INT:
      lexer.skip_token ();
      return integer_type_node;
      break;
    case Tiny::FLOAT:
      lexer.skip_token ();
      return float_type_node;
      break;
    default:
      unexpected_token (t);
      return Tree::error ();
      break;
    }
}
```

An additional type will be used for string-literals but let's postpone discussing it until then.

# Variable assignment

Ok, now we can declare variables. Let's assign them some value.

⟨assignment⟩ → ⟨identifier⟩ := ⟨expression⟩ ;

```
 1 Tree
 2 Parser::parse_assignment_statement ()
 3 {
 4   const_TokenPtr identifier = expect_token (Tiny::IDENTIFIER);
 5   if (identifier == NULL)
 6     {
 7       skip_after_semicolon ();
 8       return Tree::error ();
 9     }
10
11   SymbolPtr sym
12     = query_variable (identifier->get_str (), identifier->get_locus ());
13   if (sym == NULL)
14     {
15       skip_after_semicolon ();
16       return Tree::error ();
17     }
18
19   gcc_assert (!sym->get_tree_decl ().is_null ());
20   Tree var_decl = sym->get_tree_decl ();
21
22   const_TokenPtr assig_tok = expect_token (Tiny::ASSIG);
23   if (assig_tok == NULL)
24     {
25       skip_after_semicolon ();
```

```
26        return Tree::error ();
27      }
28
29    const_TokenPtr first_of_expr = lexer.peek_token ();
30
31    Tree expr = parse_expression ();
32    if (expr.is_error ())
33      return Tree::error ();
34
35    skip_token (Tiny::SEMICOLON);
36
37    if (var_decl.get_type () != expr.get_type ())
38      {
39        error_at (first_of_expr->get_locus (),
40                  "cannot assign value of type %s to variable '%s' of type %s",
41                  print_type (expr.get_type ()), sym->get_name ().c_str (),
42                  print_type (var_decl.get_type ()));
43        return Tree::error ();
44      }
45
46    Tree assig_expr = build_tree (MODIFY_EXPR, assig_tok->get_locus (),
47                                  void_type_node, var_decl, expr);
48
49    return assig_expr;
50  }
```

In lines 4 to 9 we gather the identifier at the left hand side of the assignment token `:=`. Next we will query in the current scope the Symbol associated to this identifier, lines 11 to 17. We skip the assignment token and then we parse the expression.

In line 37 we enforce the tiny rule that the right hand side of the assignment has to have the same type as the type of the variable in the left hand side. For the diagnostic we will need a function `print_type` that we will see below.

The GENERIC tree that is used to express the update of a variable is `MODIFY_EXPR` and has two operands: the variable being updated and the new value for it. And that's it.

# Expressions

In part 4 we used a Pratt parser to parse expressions. Now it is time to extend it so it creates GENERIC trees that represent the expressions of the program.

⟨expression⟩ → ⟨primary⟩ | ⟨unary-op⟩ ⟨expression⟩ | ⟨expression⟩ ⟨binary-op⟩ ⟨expression⟩

## Null denotations

Recall that a Pratt parser works by decomposing the expression into a null denotation and then a left denotation. The null denotation receives as a parameter the current token. In the expression grammar of tiny, null denotations handle primaries and unary operands.

```
1 Tree
2 Parser::null_denotation (const_TokenPtr tok)
3 {
4   switch (tok->get_id ())
5     {
```

⟨primary⟩ → **(** expression **)** | ⟨identifier⟩ | ⟨integer-literal⟩ | ⟨float-literal⟩ | ⟨string-literal⟩
⟨integer-literal⟩ → ⟨digit⟩ +
⟨float-literal⟩ → ⟨digit⟩ +**.** ⟨digit⟩ * | **.** ⟨digit⟩ +
⟨string-literal⟩ → **"** ⟨any-character-except-newline-or-double-quote⟩ ***"**

When we encounter an identifier, we have to look it up in the scope (this was defined in part 5). The expression is just its `VAR_DECL` that we stored in the `Symbol` when it was declared.

```
6     case Tiny::IDENTIFIER:
7       {
8         SymbolPtr s = scope.lookup (tok->get_str ());
9         if (s == NULL)
10          {
11            error_at (tok->get_locus (),
12                      "variable '%s' not declared in the current scope",
13                      tok->get_str ().c_str ());
14            return Tree::error ();
15          }
16        return Tree (s->get_tree_decl (), tok->get_locus ());
17      }
```

Note that using `Tree` rather than the GENERIC `tree` is essential for primaries. In the code above `s->get_tree_decl()` returns a tree with the location of the variable declaration. We could use this tree but for diagnostics purposes we want the location where the variable is being referenced.

For literals, the literal itself encodes the value. So the text of the token will have to be interpreted as the appropiate value. For integers we can just use `atoi`.

```
18    case Tiny::INTEGER_LITERAL:
19      // We should check the range. See note below
20      return Tree (build_int_cst_type (integer_type_node,
21                                       atoi (tok->get_str ().c_str ())),
22                   tok->get_locus ());
23      break;
```

Note: we still have to check that the value represented by the token lies in the valid range of the integer type. Let's ignore this for now.

Real literals are similar.

```
24      case Tiny::REAL_LITERAL:
25        {
26          REAL_VALUE_TYPE real_value;
27          real_from_string3 (&real_value, tok->get_str ().c_str (),
28                             TYPE_MODE (float_type_node));
29
30          return Tree (build_real (float_type_node, real_value),
31                       tok->get_locus ());
32        }
```

For a real literal we have to invoke the (GCC-provided) function `real_from_string3` (line 27) to get a real value representation from a string. This function expects the *machine* (i.e. architecture dependent) mode of the type, that we can obtain using `TYPE_MODE`. It returns its value in a `REAL_VALUE_TYPE` that then can be used to build a real constant tree using the (GCC-provided) function `build_real`.

Likewise with string literals.

```
33      case Tiny::STRING_LITERAL:
34        {
35          std::string str = tok->get_str ();
36          const char *c_str = str.c_str ();
37          return Tree (build_string_literal (strlen (c_str) + 1, c_str),
38                       tok->get_locus ());
39        }
```

To create a string literal we use the (GCC-provided) function `build_string_literal`. For practical reasons our string literal will contain the NULL terminator, otherwise the string literal itself will not be useable in C functions (more on this later).

While the type GENERIC trees created for integer and real literals was obviously `integer_type_node` and `float_type_node`, it is not so clear for string literals. The tree created by build_string_literal has type pointer to a character type. Pointer types have a tree code of `POINTER_TYPE` and the *pointee* type is found in `TREE_TYPE`. Sometimes we will need to check if an expression has the type of a string literal, so we will use the following auxiliar function.

```
bool
is_string_type (Tree type)
{
  gcc_assert (TYPE_P (type.get_tree ()));
  return type.get_tree_code () == POINTER_TYPE
         && TYPE_MAIN_VARIANT (TREE_TYPE (type.get_tree ())) == char_type_node;
}
```

In the function above, `TYPE_MAIN_VARIANT` returns the main variant of the pointee of the given pointer type and checks if it is `char_type_node`. In C parlance, this function checks if `type` represents the type «`char *`».

Back to the nullary denotation: a parenthesized expression like **(** e **)** just parses the expression e and returns its tree.

```
40      case Tiny::LEFT_PAREN:
41        {
42          Tree expr = parse_expression ();
43          tok = lexer.peek_token ();
44          if (tok->get_id () != Tiny::RIGHT_PAREN)
45            error_at (tok->get_locus (), "expecting ')' but %s found\n",
46                      tok->get_token_description ());
47          else
48            lexer.skip_token ();
49          return Tree (expr, tok->get_locus ());
50        }
```

Unary plus operator actually does nothing in tiny but it can only be applied to integer and float expressions.

```
51      case Tiny::PLUS:
52        {
53          Tree expr = parse_expression (LBP_UNARY_PLUS);
54          if (expr.is_error ())
55            return Tree::error ();
56          if (expr.get_type () != integer_type_node
57              || expr.get_type () != float_type_node)
58            {
59              error_at (tok->get_locus (),
60                        "operand of unary plus must be int or float but it is %s",
61                        print_type (expr.get_type ()));
62              return Tree::error ();
63            }
64          return Tree (expr, tok->get_locus ());
65        }
```

Now we can define the `print_type` function that we use to print human readable names for the types.

```
const char *
Parser::print_type (Tree type)
{
  gcc_assert (TYPE_P (type.get_tree ()));

  if (type == void_type_node)
      return "void";
  else if (type == integer_type_node)
      return "int";
  else if (type == float_type_node)
      return "float";
  else if (is_string_type (type))
      return "string";
  else if (type == boolean_type_node)
      return "boolean";
```

```
  else
    return "<<unknown-type>>";
}
```

Note that `print_type` uses the `is_string_type` function we defined above.

Unary minus operator is similar to the plus operator but it negates its operand.

```
    case Tiny::MINUS:
      {
        Tree expr = parse_expression (LBP_UNARY_MINUS);
        if (expr.is_error ())
          return Tree::error ();

        if (expr.get_type () != integer_type_node
            || expr.get_type () != float_type_node)
          {
            error_at (
              tok->get_locus (),
              "operand of unary minus must be int or float but it is %s",
              print_type (expr.get_type ()));
            return Tree::error ();
          }

        expr
          = build_tree (NEGATE_EXPR, tok->get_locus (), expr.get_type (), expr);
        return expr;
      }
```

A GENERIC tree with tree code `NEGATE_EXPR` computes the negation of its operand.

Unary not operator computes the logical negation of its boolean argument.

```
66      case Tiny::NOT:
67        {
68          Tree expr = parse_expression (LBP_LOGICAL_NOT);
69          if (expr.is_error ())
70            return Tree::error ();
71
72          if (expr.get_type () != boolean_type_node)
73            {
74              error_at (tok->get_locus (),
75                        "operand of logical not must be a boolean but it is %s",
76                        print_type (expr.get_type ()));
77              return Tree::error ();
78            }
79
80          expr = build_tree (TRUTH_NOT_EXPR, tok->get_locus (), boolean_type_node,
81                             expr);
82          return expr;
83        }
```

The GENERIC tree code for a logical negation is `TRUTH_NOT_EXPR`.

Finally, any other token is a syntax error, so diagnose them as usual. This completes the handling of null denotations.

```
84      default:
85        unexpected_token (tok);
86        return Tree::error ();
87      }
88  }
```

# Left denotations

Left denotations are used for infix operators.

```
Tree
Parser::left_denotation (const_TokenPtr tok, Tree left)
{
  BinaryHandler binary_handler = get_binary_handler (tok->get_id ());
  if (binary_handler == NULL)
    {
      unexpected_token (tok);
      return Tree::error ();
    }

  return (this->*binary_handler) (tok, left);
}
```

If you recall from part 4, we used the function `get_binary_handler` to get a handler of our binary expression and then dispatch to it the handling of the current token. In contrast to the version of `left_denotation` in part 4, in addition to the token we will have to pass the left tree (computed by a call to `null_denotation` or `left_denotation`, possibly in a recursive way).

Now come a bunch of expression handlers for binary operators. We will focus on the most interesting ones. You can find the remaining ones in the tiny parser. Let's start with the addition.

```
Tree
Parser::binary_plus (const_TokenPtr tok, Tree left)
{
  Tree right = parse_expression (LBP_PLUS);
  if (right.is_error ())
    return Tree::error ();

  Tree tree_type = coerce_binary_arithmetic (tok, &left, &right);
  if (tree_type.is_error ())
    return Tree::error ();

  return build_tree (PLUS_EXPR, tok->get_locus (), tree_type, left, right);
}
```

We parse the right hand side (recall that the token `tok` has already been consumed in `parse_expression`). Now using the left hand side and the right hand side we have to compute the resulting type of this binary operator. We call `coerce_binary_arithmetic` that returns the type of

the binary operation and may modify its input trees, more on this below. Finally we construct a GENERIC tree with code `PLUS_EXPR` that is used to represent binary addition.

Function `coerce_binary_arithmetic` simply applies the rules of tiny regarding arithmetic operations: operating two integers or two floats returns integer and float respectively. Mixing a float and an integer returns a float value. The integer operand, thus, must be first converted to a float. The tree code `FLOAT_EXPR` is used to convert from integer to float.

```
Tree
Parser::coerce_binary_arithmetic (const_TokenPtr tok, Tree *left, Tree *right)
{
  Tree left_type = left->get_type ();
  Tree right_type = right->get_type ();

  if (left_type.is_error () || right_type.is_error ())
    return Tree::error ();

  if (left_type == right_type)
    {
      if (left_type == integer_type_node || left_type == float_type_node)
        {
          return left_type;
        }
    }
  else if ((left_type == integer_type_node && right_type == float_type_node)
           || (left_type == float_type_node && right_type == integer_type_node))
    {
      if (left_type == integer_type_node)
        {
          *left = build_tree (FLOAT_EXPR, left->get_locus (), float_type_node,
                              left->get_tree ());
        }
      else
        {
          *right = build_tree (FLOAT_EXPR, right->get_locus (),
                               float_type_node, right->get_tree ());
        }
      return float_type_node;
    }

  // i.e. int + boolean
  error_at (tok->get_locus (),
            "invalid operands of type %s and %s for operator %s",
            print_type (left_type), print_type (right_type),
            tok->get_token_description ());
  return Tree::error ();
}
```

Subtraction and multiplication are exactly the same code but the GENERIC tree is `MINUS_EXPR` and `MULT_EXPR` respectively.

Binary division is a bit more interesting. When both operands are integer, we will do an integer division, otherwise a real division. Each operation is represented using different tree codes.

```
Tree
Parser::binary_div (const_TokenPtr tok, Tree left)
{
  Tree right = parse_expression (LBP_DIV);
  if (right.is_error ())
    return Tree::error ();

  if (left.get_type () == integer_type_node
      && right.get_type () == integer_type_node)
    {
      // Integer division (truncating, like in C)
      return build_tree (TRUNC_DIV_EXPR, tok->get_locus (), integer_type_node,
                         left, right);
    }
  else
    {
      // Real division
      Tree tree_type = coerce_binary_arithmetic (tok, &left, &right);
      if (tree_type.is_error ())
        return Tree::error ();

      gcc_assert (tree_type == float_type_node);

      return build_tree (RDIV_EXPR, tok->get_locus (), tree_type, left, right);
    }
}
```

Modulus is similar to division but there is no real modulus operation, so this case diagnoses an error. The tree code for the integer modulus is `TRUNC_MOD_EXPR`.

All handlers for relational operators **==**, **!=**, **<**, **>**, **<=** and **>=** have the same code. Only their tree codes change.

```
Tree
Parser::binary_equal (const_TokenPtr tok, Tree left)
{
  Tree right = parse_expression (LBP_EQUAL);
  if (right.is_error ())
    return Tree::error ();

  Tree tree_type = coerce_binary_arithmetic (tok, &left, &right);
  if (tree_type.is_error ())
    return Tree::error ();

  return build_tree (EQ_EXPR, tok->get_locus (), boolean_type_node, left,
                     right);
}
```

Tree codes for **!=**, **<**, **>**, **<=** and **>=** are (respectively) `NE_EXPR`, `LT_EXPR`, `GT_EXPR`, `LE_EXPR` and `GE_EXPR`.

Likewise, binary logical operators **and** and **or** only differ in their tree codes.

```
Tree
Parser::binary_logical_and (const_TokenPtr tok, Tree left)
{
  Tree right = parse_expression (LBP_EQUAL);
  if (right.is_error ())
    return Tree::error ();

  if (!check_logical_operands (tok, left, right))
    return Tree::error ();

  return build_tree (TRUTH_ANDIF_EXPR, tok->get_locus (), boolean_type_node,
                     left, right);
}
```

The tree code for logical or is `TRUTH_ORIF_EXPR`. Function `check_logical_operands` simply verifies that both operands are logical.

```
bool
Parser::check_logical_operands (const_TokenPtr tok, Tree left, Tree right)
{
  if (left.get_type () != boolean_type_node
      || right.get_type () != boolean_type_node)
    {
      error_at (
        tok->get_locus (),
        "operands of operator %s must be boolean but they are %s and %s\n",
        tok->get_token_description (), print_type (left.get_type ()),
        print_type (right.get_type ()));
      return false;
    }

  return true;
}
```

And we are done with the expressions!

# Write statement

⟨write⟩ → **write** ⟨expression⟩ **;**

A write statement is not particularly complicated at first.

```
1 Tree
2 Parser::parse_write_statement ()
3 {
4   // write_statement -> "write" expression ";"
5
6   if (!skip_token (Tiny::WRITE))
7     {
```

```
 8       skip_after_semicolon ();
 9       return Tree::error ();
10     }
11
12   const_TokenPtr first_of_expr = lexer.peek_token ();
13   Tree expr = parse_expression ();
14
15   skip_token (Tiny::SEMICOLON);
16
17   if (expr.is_error ())
18     return Tree::error ();
```

Now we have to print the value of the expression. To do this we will emit a call to `printf` with the appropiate format conversion: `%d` for integers, and `%f` for floats. For strings, we will simply call `puts` (although we could have called printf with a format conversion `%s`).

Let's see the case for integers.

```
20   if (expr.get_type () == integer_type_node)
21     {
22       // printf("%d\n", expr)
23       const char *format_integer = "%d\n";
24       tree args[]
25         = {build_string_literal (strlen (format_integer) + 1, format_integer),
26            expr.get_tree ()};
27
28       Tree printf_fn = get_printf_addr ();
29
30       tree stmt
31         = build_call_array_loc (first_of_expr->get_locus (), integer_type_node,
32                                 printf_fn.get_tree (), 2, args);
33
34       return stmt;
35     }
```

In line 31 we build a call to the print function (represented in `printf_fn`). In this call we will pass two arguments, that we have in the array `args`. The first argument is the format string, so we build a string literal "%d\n" (line 25, mind the NULL terminator) and the second is our expression of type integer (line 26). Function `build_call_array_loc` is provided by GCC.

To be able to call `printf` we need first to obtain its declaration, i.e. a `FUNCTION_DECL`. But for some reason, though, GENERIC trees do not allow calling a `FUNCTION_DECL` directly, it has to be done through an address to the function declaration. Function `get_printf_addr` thus, returns an address to a function declaration of `printf`.

```
1 Tree
2 Parser::get_printf_addr ()
3 {
4   if (printf_fn.is_null ())
5     {
6       tree fndecl_type_param[] = {
```

```
 7         build_pointer_type (
 8           build_qualified_type (char_type_node,
 9                                 TYPE_QUAL_CONST)) /* const char* */
10       };
11       tree fndecl_type
12         = build_varargs_function_type_array (integer_type_node, 1,
13                                              fndecl_type_param);
14
15       tree printf_fn_decl = build_fn_decl ("printf", fndecl_type);
16       DECL_EXTERNAL (printf_fn_decl) = 1;
17
18       printf_fn
19         = build1 (ADDR_EXPR, build_pointer_type (fndecl_type), printf_fn_decl);
20     }
21
22   return printf_fn;
23 }
```

To avoid repeatedly creating function declarations to the same `printf` function, our `Parser` class will keep a `printf_fn` tree with the address to `printf`. The first time we request the address of `printf` it will be a `NULL_TREE` so we will have to compute it.

Functions, like variables, have type. We need to create a function with a variable number of arguments that returns integer and has one fixed argument of type `const char*`. This is because the definition in C of printf is `int printf(const char*, ...)`. The type `const char*` is created by constructing a pointer type to a const qualified version of the `char_type_node` (line 7). Then we build the function type itself (line 12).

Once we have the function type, we can build the declaration as a variable argument function (line 15). This function will not be defined by tiny, but it will come elsewhere, so we set that property in the declaration itself by marking it as `DECL_EXTERNAL` (line 16). Finally we build an ADDR_EXPR which simply returns a pointer to the type of the function type. This tree represents the address to the function. This is what the function will return.

Back to the implementation of the write statement, the case for float is similar to that of the integer but requires us to convert the float value into a double value, because this is how it works in C.

```
else if (expr.get_type () == float_type_node)
  {
    // printf("%f\n", (double)expr)
    const char *format_float = "%f\n";
    tree args[]
      = {build_string_literal (strlen (format_float) + 1, format_float),
         convert (double_type_node, expr.get_tree ())};

    Tree printf_fn = get_printf_addr ();

    tree stmt
      = build_call_array_loc (first_of_expr->get_locus (), integer_type_node,
                              printf_fn.get_tree (), 2, args);
```

```
      return stmt;
    }
```

To convert the float into a double we invoke the GCC `convert` function that will require an extra file with some generic boilerplate. That file is not interesting enough to put it here. Alternatively a `CONVERT_EXPR` tree could be used instead.

Finally to print a string, we just call `puts`.

```
  else if (is_string_type (expr.get_type ()))
    {
      // Alternatively we could use printf('%s\n', expr) instead of puts(expr)
      tree args[] = {expr.get_tree ()};

      Tree puts_fn = get_puts_addr ();

      tree stmt
        = build_call_array_loc (first_of_expr->get_locus (), integer_type_node,
                                puts_fn.get_tree (), 1, args);
      return stmt;
    }
```

In contrast to `printf`, `puts` is not a variable argument function, so its type is constructed slightly different. Everything else is the same.

```
Tree
Parser::get_puts_addr ()
{
  if (puts_fn.is_null ())
    {
      tree fndecl_type_param[] = {
        build_pointer_type (
          build_qualified_type (char_type_node,
                                TYPE_QUAL_CONST)) /* const char* */
      };
      tree fndecl_type
        = build_function_type_array (integer_type_node, 1, fndecl_type_param);

      tree puts_fn_decl = build_fn_decl ("puts", fndecl_type);
      DECL_EXTERNAL (puts_fn_decl) = 1;

      puts_fn
        = build1 (ADDR_EXPR, build_pointer_type (fndecl_type), puts_fn_decl);
    }

  return puts_fn;
}
```

Having handled all valid types, this completes our write statement.

```
  else
    {
      error_at (first_of_expr->get_locus (),
```

```
                "value of type %s is not a valid write operand",
                print_type (expr.get_type ()));
      return Tree::error ();
    }

  gcc_unreachable ();
}
```

# Blocks

Both a tiny program and statements if, while and for statements have in their syntax 〈statement〉*. In addition, if, while and for statements introduce a new symbol mapping in each of its 〈statement〉*. The top level is actually not that different if we understand that the program has a top level symbol mapping.

This suggests that, whenever we are going to parse a 〈statement〉*, the same process will happen: a) we will push a new symbol mapping b) parse the statements c) pop the scope.

```
enter_scope ();
parse_statement_seq (done);
leave_scope ();
```

Functions enter_scope and leave_scope will make sure a new symbol mapping is pushed/popped.

GENERIC trees represent mappings using a `BIND_EXPR`. A `BIND_EXPR` will contain a list of statements and also a list of `VAR_DECL`s related to the variable declarations in the current symbol mapping. Recall that when we declared a variable, one of the things we did is adding the variable into a stack of lists called `stack_var_decl_chain`. This is the list we will use to gather all the `VAR_DECL`s in a mapping.

Unfortunately at this point, GENERIC makes things a bit complicated because of another kind of tree called block. Let's explain this seeing the code of `enter_scope`.

```
 1 void
 2 Parser::enter_scope ()
 3 {
 4   scope.push_scope ();
 5
 6   TreeStmtList stmt_list;
 7   stack_stmt_list.push_back (stmt_list);
 8
 9   stack_var_decl_chain.push_back (TreeChain ());
10   stack_block_chain.push_back (BlockChain ());
11 }
```

We first push a new symbol mapping (line 4). And then we have three stacks: a first stack of lists of statements, a second stack of lists var declarations and a third stack of chains of blocks (lines 7 to 10).

`TreeStmtList` is just a tiny wrapper around `STATEMENT_LIST`. This is a tree used to represent lists of statements.

```
struct TreeStmtList
{
public:
  TreeStmtList () : list (alloc_stmt_list ()) {}
  TreeStmtList (Tree t) : list (t.get_tree ()) {}

  void
  append (Tree t)
  {
    append_to_statement_list (t.get_tree (), &list);
  }

  tree
  get_tree () const
  {
    return list;
  }

private:
  tree list;
};
```

Functions `alloc_stmt_list` and `append_to_statement_list` are GCC-provided and do the obvious things.

`TreeChain` and `BlockChain` are conceptually singly-linked lists implemented using GENERIC trees. In fact they work exactly the same, but unfortunately a different accessor has to be used for each. To reduce the differences both have been wrapped in two classes that inherit from a generic one.

```
template <typename Append> struct TreeChainBase
{
  Tree first;
  Tree last;

  TreeChainBase () : first (), last () {}

  void
  append (Tree t)
  {
    gcc_assert (!t.is_null());
    if (first.is_null())
      {
        first = last = t;
      }
    else
      {
        Append () (last, t);
        last = t;
      }
  }
```

```
};

struct tree_chain_append
{
  void operator() (Tree t, Tree a) { TREE_CHAIN (t.get_tree()) = a.get_tree(); }
};

struct TreeChain : TreeChainBase<tree_chain_append>
{
};

struct block_chain_append
{
  void operator() (Tree t, Tree a) { BLOCK_CHAIN (t.get_tree()) = a.get_tree(); }
};

struct BlockChain : TreeChainBase<block_chain_append>
{
};
```

We keep the first tree and the last one in order to handle this list. The `Append` process has been abstracted away since a `TreeChain` must use `TREE_CHAIN` and `BlockChain` must use `BLOCK_CHAIN`.

So, what are these two stacks of `TreeChain` and `BlockChain`? `TreeChain` will be used for the `VAR_DECL`s. So the append you saw in line 48 of `parse_variable_declaration` above is actually appending to the top `TreeChain` in `stack_var_decl_chain`.

`BlockChain` is used for a chain of blocks that GENERIC requires us to maintain. Each block, a GENERIC tree of tree code `BLOCK`, has a list of `VAR_DECL`s. This list is the same as the `BIND_EXPR` representing the symbol mapping. Blocks also have a list of subblocks and a parent context. This parent context is another block, except for the topmost one that will be a function declaration, more on this below.

The complexity arises when we have several sibling blocks. We have to gather them in a way that when we leave their containing block: a) we set the containing block as the parent of the blocks b) that containing block has a list of subblocks. And this is where the `BlockChain` is used.

All this complex process happens in `leave_scope`.

```
 1 Parser::TreeSymbolMapping
 2 Parser::leave_scope ()
 3 {
 4   TreeStmtList current_stmt_list = get_current_stmt_list ();
 5   stack_stmt_list.pop_back ();
 6
 7   TreeChain var_decl_chain = stack_var_decl_chain.back ();
 8   stack_var_decl_chain.pop_back ();
 9
10   BlockChain subblocks = stack_block_chain.back ();
11   stack_block_chain.pop_back ();
12
```

```
13   tree new_block
14     = build_block (var_decl_chain.first.get_tree (),
15                    subblocks.first.get_tree (),
16                    /* supercontext */ NULL_TREE, /* chain */ NULL_TREE);
17
18   // Add the new block to the current chain of blocks (if any)
19   if (!stack_block_chain.empty ())
20     {
21       stack_block_chain.back ().append (new_block);
22     }
23
24   // Set the subblocks to have the new block as their parent
25   for (tree it = subblocks.first.get_tree (); it != NULL_TREE;
26        it = BLOCK_CHAIN (it))
27     BLOCK_SUPERCONTEXT (it) = new_block;
28
29   tree bind_expr
30     = build3 (BIND_EXPR, void_type_node, var_decl_chain.first.get_tree (),
31               current_stmt_list.get_tree (), new_block);
32
33   TreeSymbolMapping tree_scope;
34   tree_scope.bind_expr = bind_expr;
35   tree_scope.block = new_block;
36
37   scope.pop_scope();
38
39   return tree_scope;
40 }
```

We get the current list of statements and we pop them from the stack of statement lists (lines 4 and 5). Likewise for the list of VAR_DECLs (lines 7 to 8). And again for the current chain of blocks (lines 10 and 11). Now we have to build a new block using the (GCC-provided) function `build_block`. Its first operand will be the list of VAR_DECLs, its second is the list of sub blocks that we may have gathered. We cannot set yet the parent (called the *supercontext*) so we leave it is a NULL_TREE. This block does not have any chain, yet, either, so the fourth operand is also NULL_TREE.

Then, if this block is not the topmost one, it has to be added to the current block chain (lines 19 to 21). The topmost block will not be enclosed anywhere, this is why the stack might be empty (see line 11).

When we create the blocks, we leave their parent empty (line 16). Now it is the right moment to take all the subblocks and set their parent (i.e. their supercontext) to the newly created block (lines 25 to 27). This is required because BLOCKs are somehow doubly-linked: the current block knows its subblocks and each subblock knows its parent.

Now we can create a BIND_EXPR (line 29). Its first operand is the list of VAR_DECLs. The same as the block we have just created. The second operand is the list of statements and the third operand is the block we have just created.

This function does not return a single `Tree` but a `TreeMapping` that is just a tuple containing both the just created `bind_expr` and the just created `block` (lines 33 to 35). The block is only required in the top level statement. Any other statement will use `bind_expr`.

Finally we pop the symbol mapping and return (line 37).

# Statement sequences

Ok, `enter_scope` and `leave_scope` are to be called after and before 〈statement〉 * but we still have to parse the statements themselves.

```
void
Parser::parse_statement_seq (bool (Parser::*done) ())
{
  // Parse statements until done and append to the current stmt list;
  while (!(this->*done) ())
    {
      Tree stmt = parse_statement ();
      get_current_stmt_list ().append (stmt);
    }
}
```

In contrast to most `parse_xxx` functions, this one does not return a `Tree`. What it does instead is appending each parsed statement to the current statement list. Recall that `enter_scope` and `leave_scope` update the stack of statement list, thus the current one is always the one in the top of the stack.

```
TreeStmtList &
Parser::get_current_stmt_list ()
{
  return stack_stmt_list.back ();
}
```

# Program

Since this post is already too long, let's end with what we have to do for a tiny program.

```
 1 void
 2 Parser::parse_program ()
 3 {
 4   // Built type of main "int (int, char**)"
 5   tree main_fndecl_type_param[] = {
 6     integer_type_node,                                      /* int */
 7     build_pointer_type (build_pointer_type (char_type_node)) /* char** */
 8   };
 9   tree main_fndecl_type
10     = build_function_type_array (integer_type_node, 2, main_fndecl_type_param);
11   // Create function declaration "int main(int, char**)"
12   main_fndecl = build_fn_decl ("main", main_fndecl_type);
13
```

```
14   // Enter top level scope
15   enter_scope ();
16   // program -> statement*
17   parse_statement_seq (&Parser::done_end_of_file);
18   // Append "return 0;"
19   tree resdecl
20     = build_decl (UNKNOWN_LOCATION, RESULT_DECL, NULL_TREE, integer_type_node);
21   DECL_RESULT (main_fndecl) = resdecl;
22   tree set_result
23     = build2 (INIT_EXPR, void_type_node, DECL_RESULT (main_fndecl),
24               build_int_cst_type (integer_type_node, 0));
25   tree return_stmt = build1 (RETURN_EXPR, void_type_node, set_result);
26
27   get_current_stmt_list ().append (return_stmt);
28
29   // Leave top level scope, get its binding expression and its main block
30   TreeSymbolMapping main_tree_scope = leave_scope ();
31   Tree main_block = main_tree_scope.block;
32
33   // Finish main function
34   BLOCK_SUPERCONTEXT (main_block.get_tree ()) = main_fndecl;
35   DECL_INITIAL (main_fndecl) = main_block.get_tree ();
36   DECL_SAVED_TREE (main_fndecl) = main_tree_scope.bind_expr.get_tree ();
37
38   DECL_EXTERNAL (main_fndecl) = 0;
39   DECL_PRESERVE_P (main_fndecl) = 1;
40
41   // Convert from GENERIC to GIMPLE
42   gimplify_function_tree (main_fndecl);
43
44   // Insert it into the graph
45   cgraph_node::finalize_function (main_fndecl, true);
46
47   main_fndecl = NULL_TREE;
48 }
```

In order for our program to be able to start, we need a `main` function (like in C). We do this in lines 4 to 12. This is similar to what we did for the `printf` and `puts` functions.

Then we enter the top level scope. We parse the sequence of statements (line 17). Before leaving the current scope, we want to return 0, to signal that the program ends correctly. We will append a return expression to the current statement list. Before we can return anything, though, in GENERIC we first need to create a `RESULT_DECL` declaration (lines 19 to 21) and initialize it with some value using a `INIT_EXPR` (lines 22 to 24). Now we can create a return expression that, aside from returning, initializes the return *variable* (line 25). Finally we append it to the current statement list (line 27).

Now we leave the scope, this returns a pair of trees `block` and `bind_expression`. We have to set the parent (i.e. the supercontext) of the `block` to the main function, since it is the topmost block (line 34). Then we set the main block of the function (line 35) and the code proper of the function to be

`bind_expr` (line 36). Then we make sure the main function is not set as extern because it is being defined by ourselves (line 38) and we tell the compiler to preserve it, otherwise it would be removed since nobody is explicitly using it (line 39).

Now, we have to convert this function from GENERIC to GIMPLE (line 42) by calling the (GCC-provided) `gimplify_function_tree`. GIMPLE is a subset of GENERIC that is used by the middle end. Once converted the function can be queued for compilation in later passes of the compiler (line 45).

# Smoke test

At this point our tiny front end is starting to be useful. A very basic smoketest that should work is the following one.

```
# test.tiny
var a : int;
a := 42;
write a;
```

```
$ gcctiny -o test test.tiny
$ ./test
42
```

Yay!

# Wrap-up

Ok, that post was again a long one. I have skipped some statements (read, if, while and for) that we will see in the next chapter but at least now we can play with assignment and the write statement.

That's all for today.

« A tiny GCC front end – Part 5 A tiny GCC front end – Part 7 »