A tiny GCC front end – Part 7

Jan 19, 2016 • Roger Ferrer Ibáñez • compilers, GCC • gcc, tiny

In this part we will complete the missing statements from part 6 and finish our front end.

Read statement

A read statement is a bit like the dual of a write statement. We will implement it using a call to `scanf`.

⟨read⟩ → **read** ⟨identifier⟩ **;**

```
1 Tree
2 Parser::parse_read_statement ()
3 {
4   if (!skip_token (Tiny::READ))
5     {
6       skip_after_semicolon ();
7       return Tree::error ();
8     }
9
10  const_TokenPtr first_of_expr = lexer.peek_token ();
11  Tree expr = parse_expression ();
12
13  skip_token (Tiny::SEMICOLON);
14
15  if (expr.is_error ())
16    return Tree::error ();
17
18  if (expr.get_tree_code () != VAR_DECL)
19    {
20      error_at (first_of_expr->get_locus (),
21                "invalid expression in read statement");
22      return Tree::error ();
23    }
24
25  // Now this variable must be addressable
26  TREE_ADDRESSABLE (expr.get_tree ()) = 1;
27
28  const char *format = NULL;
29  if (expr.get_type () == integer_type_node)
```

```
30      {
31        format = "%d";
32      }
33    else if (expr.get_type () == float_type_node)
34      {
35        format = "%f";
36      }
37    else
38      {
39        error_at (first_of_expr->get_locus (),
40                  "variable of type %s is not a valid read operand",
41                  print_type (expr.get_type ()));
42        return Tree::error ();
43      }
44
45    tree args[]
46      = {build_string_literal (strlen (format) + 1, format),
47         build_tree (ADDR_EXPR, first_of_expr->get_locus (),
48                     build_pointer_type (expr.get_type ().get_tree ()), expr)
49           .get_tree ()};
50
51    Tree scanf_fn = get_scanf_addr ();
52
53    tree stmt
54      = build_call_array_loc (first_of_expr->get_locus (), integer_type_node,
55                              scanf_fn.get_tree (), 2, args);
56
57    return stmt;
58  }
```

Here we depart a bit from the specification in part 1 because it says that a read statement expects an identifier. We abuse a bit `parse_expression` (line 11) and we force it to be a variable name (lines 18 to 23). Of course we could have manually looked up the identifier token. But `parse_expression` does this for us anyway, why not use it? Note that this strategy could be applied to the left part of the assignment statement.

Now comes an interesting aspect of GENERIC: `VAR_DECL`s do not have to be in memory. We want `scanf` to update our variable and the only way to do this is by passing to `scanf` the address of the variable. So we have to state that this variable will have its address computed (line 26). Failing to do this, GCC would create a temporary from our variable and would use that one instead: our variable would stay untouched.

We then prepare the call to `scanf`, first we set the appropiate format string depending on the type of the variable (lines 28 to 43). Then we build the arguments to `scanf`. The first one is the format string as a string literal (line 46) and the second one (line 47) is an `ADDR_EXPR`. This tree means getting the address of its operand. The type of this expression should be a pointer type to our variable. Similar to what we did with `puts` and `printf` in the write statement, we get the address of `scanf` (line 51). Finally everything is set to make the call to scanf (line 55).

If statement

⟨if⟩ → **if** ⟨expression⟩ **then** ⟨statement⟩ * **end**
| **if** ⟨expression⟩ **then** ⟨statement⟩ * **else** ⟨statement⟩ * **end**

Control statements are a bit more complicated than other statements so we will split the parsing proper and the GENERIC tree construction. You will also see that the tree synthesized for these control statements is often a TreeStmtList: the implementation of these statements require several GENERIC trees. Let's see first how to parse an if statement.

```
1 Tree
2 Parser::parse_if_statement ()
3 {
4   if (!skip_token (Tiny::IF))
5     {
6       skip_after_end ();
7       return Tree::error ();
8     }
9
10   Tree expr = parse_boolean_expression ();
11
12   skip_token (Tiny::THEN);
13
14   enter_scope ();
15   parse_statement_seq (&Parser::done_end_or_else);
16
17   TreeSymbolMapping then_tree_scope = leave_scope ();
18   Tree then_stmt = then_tree_scope.bind_expr;
19
20   Tree else_stmt;
21   const_TokenPtr tok = lexer.peek_token ();
22   if (tok->get_id () == Tiny::ELSE)
23     {
24       // Consume 'else'
25       skip_token (Tiny::ELSE);
26
27       enter_scope ();
28       parse_statement_seq (&Parser::done_end);
29       TreeSymbolMapping else_tree_scope = leave_scope ();
30       else_stmt = else_tree_scope.bind_expr;
31
32       // Consume 'end'
33       skip_token (Tiny::END);
34     }
35   else if (tok->get_id () == Tiny::END)
36     {
37       // Consume 'end'
38       skip_token (Tiny::END);
39     }
```

```
40   else
41     {
42       unexpected_token (tok);
43       return Tree::error ();
44     }
45
46   return build_if_statement (expr, then_stmt, else_stmt);
47 }
```

It is not uncommon in control structures to find expressions that are slightly more restricted than the general expressions. It makes sense, thus, to parse the condition expression using a specialized function `parse_boolean_expression` (line 10) that verifies that the expression has boolean type.

```
Tree
Parser::parse_boolean_expression ()
{
  Tree expr = parse_expression ();
  if (expr.is_error ())
    return expr;

  if (expr.get_type () != boolean_type_node)
    {
      error_at (expr.get_locus (),
                "expected expression of boolean type but its type is %s",
                print_type (expr.get_type ()));
      return Tree::error ();
    }
  return expr;
}
```

Both the *then* part and the *else* part of an if statement are 〈statement〉*. According to the tiny definition, there is a new symbol mapping for them. So we simply enter the scope, parse the statement sequence and then leave the scope to get the `BIND_EXPR` of the block (lines 14 to 18). We do the same if there is an else part (lines 27 to 30).

Now we call the function `build_if_statement` that will be the responsible for building the GENERIC tree of this if statement (line 46).

```
 1 Tree
 2 Parser::build_if_statement (Tree bool_expr, Tree then_part, Tree else_part)
 3 {
 4   if (bool_expr.is_error ())
 5     return bool_expr;
 6
 7   Tree then_label_decl = build_label_decl ("then", then_part.get_locus ());
 8
 9   Tree else_label_decl;
10   if (!else_part.is_null ())
11     else_label_decl = build_label_decl ("else", else_part.get_locus ());
12
13   Tree endif_label_decl = build_label_decl ("end_if", then_part.get_locus ());
14
```

```
15   Tree goto_then = build_tree (GOTO_EXPR, bool_expr.get_locus (),
16                                void_type_node, then_label_decl);
17   Tree goto_endif = build_tree (GOTO_EXPR, bool_expr.get_locus (),
18                                 void_type_node, endif_label_decl);
19
20   Tree goto_else_or_endif;
21   if (!else_part.is_null ())
22     goto_else_or_endif = build_tree (GOTO_EXPR, bool_expr.get_locus (),
23                                      void_type_node, else_label_decl);
24   else
25     goto_else_or_endif = goto_endif;
26
27   TreeStmtList stmt_list;
28
29   Tree cond_expr
30     = build_tree (COND_EXPR, bool_expr.get_locus (), void_type_node, bool_expr,
31                   goto_then, goto_else_or_endif);
32   stmt_list.append (cond_expr);
33
34   Tree then_label_expr = build_tree (LABEL_EXPR, then_part.get_locus (),
35                                      void_type_node, then_label_decl);
36   stmt_list.append (then_label_expr);
37
38   stmt_list.append (then_part);
39
40   if (!else_part.is_null ())
41     {
42       // Make sure after then part has been executed we go to the end if
43       stmt_list.append (goto_endif);
44
45       Tree else_label_expr = build_tree (LABEL_EXPR, else_part.get_locus (),
46                                          void_type_node, else_label_decl);
47       stmt_list.append (else_label_expr);
48
49       stmt_list.append (else_part);
50     }
51
52   Tree endif_label_expr = build_tree (LABEL_EXPR, UNKNOWN_LOCATION,
53                                       void_type_node, endif_label_decl);
54   stmt_list.append (endif_label_expr);
55
56   return stmt_list.get_tree ();
57 }
```

When GENERIC trees were introduced in part 5 we said that some of them can be classified as declarations. We have mostly used `VAR_DECL`s and some function declarations (albeit indirectly for calls and the main function). Now we will need `LABEL_DECL`s. These trees represent the mere existence of a label. Since each label must be linked to its function, that in tiny it will be the main, we will use an auxiliar function to create them.

```
Tree
Parser::build_label_decl (const char *name, location_t loc)
```

```
{
  tree t = build_decl (loc, LABEL_DECL, get_identifier (name), void_type_node);

  gcc_assert (main_fndecl != NULL_TREE);
  DECL_CONTEXT (t) = main_fndecl;

  return t;
}
```

Labels represent locations of our program (in contrast to variables that represent data). The location represented by a label is defined by a `LABEL_EXPR` tree. Once a label has been defined, then we can use it to change the program execution to that label. Lists of statements implicitly execute in sequence unless a `GOTO_EXPR` changes the control flow.

Back to the implementation of the if statement, we start by creating 2 or 3 labels: one for the then part, another for the else part (if any) and another one for the end if (lines 7 to 13).

An if statement will first evaluate its condition, that we have represented in the parameter bool_expr. If this expression is true the program will branch to the then part, otherwise if there is else the program will branch to the else part. If there is no else part and the condition does not evaluate to true we will branch directly to the end of the if. When a then part ends it will also have to branch to the end of the if. The else part does not have to branch to end if, as implicit sequencing will achieve the same.

Branching is achieved using `GOTO_EXPR` trees. So the first thing we do is creating several `GOTO_EXPR`s (lines 15 to 25). Now we need to perform the conditional branching. This is done using a tree `COND_EXPR`, its three operands are the boolean expression, the true expression and the false expression. We will branch to the then part in the true expression and to the else part or the end of the if for the false expression (line 30). We will create a statement list for the if statement (line 27) where we will append all the statements required to implement an if statement. Obviously the `COND_EXPR` tree goes first (line 32).

Now we define the location related to the then part. We do that by creating a `LABEL_EXPR` tree for the label declaration of the then part (line 34) and we append it to the statement list (line 36). Now we append the tree `then_part` that we got as a parameter and that contains the then part parsed above (line 38).

If there is else part we append a goto endif, so the then part branches to the end of the if when completed (line 43). Similarly to the then part, we define the location of the else label (line 45), we append it (line 47) and then we append the else part tree that we got in the parameter `else_part` (line 49). As we said above, there is no need to jump to end if in the else part.

Finally we define the label for the end if (lines 52 and 53), append it to the statement list (line 54) before we just return it (line 56).

While statement

We will use the same strategy for the while statement: first parse its syntactic elements and then build a statement list to implement it.

⟨while⟩ → **while** ⟨expression⟩ **do** ⟨statement⟩ * **end**

```
Tree
Parser::parse_while_statement ()
{
  if (!skip_token (Tiny::WHILE))
    {
      skip_after_end ();
      return Tree::error ();
    }

  Tree expr = parse_boolean_expression ();
  if (!skip_token (Tiny::DO))
    {
      skip_after_end ();
      return Tree::error ();
    }

  enter_scope ();
  parse_statement_seq (&Parser::done_end);
  TreeSymbolMapping while_body_tree_scope = leave_scope ();

  Tree while_body_stmt = while_body_tree_scope.bind_expr;

  skip_token (Tiny::END);

  return build_while_statement (expr, while_body_stmt);
}
```

Parsing a while statement is relatively easy: a condition expression of boolean type and then a body. We then call `build_while_statement` with these two parts.

```
 1 Tree
 2 Parser::build_while_statement (Tree bool_expr, Tree while_body)
 3 {
 4   if (bool_expr.is_error ())
 5     return Tree::error ();
 6
 7   TreeStmtList stmt_list;
 8
 9   Tree while_check_label_decl
10     = build_label_decl ("while_check", bool_expr.get_locus ());
11
12   Tree while_check_label_expr
13     = build_tree (LABEL_EXPR, bool_expr.get_locus (), void_type_node,
14                   while_check_label_decl);
15   stmt_list.append (while_check_label_expr);
16
```

```
17   Tree while_body_label_decl
18     = build_label_decl ("while_body", while_body.get_locus ());
19   Tree end_of_while_label_decl
20     = build_label_decl ("end_of_while", UNKNOWN_LOCATION);
21
22   Tree cond_expr
23     = build_tree (COND_EXPR, bool_expr.get_locus (), void_type_node, bool_expr,
24                   build_tree (GOTO_EXPR, bool_expr.get_locus (), void_type_node,
25                               while_body_label_decl),
26                   build_tree (GOTO_EXPR, bool_expr.get_locus (), void_type_node,
27                               end_of_while_label_decl));
28   stmt_list.append (cond_expr);
29
30   Tree while_body_label_expr
31     = build_tree (LABEL_EXPR, while_body.get_locus (), void_type_node,
32                   while_body_label_decl);
33   stmt_list.append (while_body_label_expr);
34
35   stmt_list.append (while_body);
36
37   Tree goto_check = build_tree (GOTO_EXPR, UNKNOWN_LOCATION, void_type_node,
38                                 while_check_label_decl);
39   stmt_list.append (goto_check);
40
41   Tree end_of_while_label_expr
42     = build_tree (LABEL_EXPR, UNKNOWN_LOCATION, void_type_node,
43                   end_of_while_label_decl);
44   stmt_list.append (end_of_while_label_expr);
45
46   return stmt_list.get_tree ();
47 }
```

We start by creating a label for the condition check (line 10) and defining its location that we will append to the statement list (lines 12 to 15). Then we define two other labels one for the body of the loop and one to end the loop (lines 17 to 20). Now we add a COND_EXPR tree that evaluates the condition expression. It will branch to the body of the loop when the condition is true, to the end of the while otherwise (lines 22 to 28). Then we define the location of the label for the body of the loop (lines 30 to 33) and append the while body (line 35). Then we have to branch back (this is why it is a loop) to the condition check (lines 37 to 39). Then we just define the location of the label for the end of the while (lines 41 to 44). Our while statement is done, so let's return it (line 46).

For-statement

⟨for⟩ → **for** ⟨identifier⟩ **:=** ⟨expression⟩ **to** ⟨expression⟩ **do** ⟨statement⟩ * **end**

If you recall part 1, we defined a for statement like the following

```
for id := L to U do
  S
```

```
end
```

to be semantically equivalent to

```
id := L;
while (id <= U) do
 S
 id := id + 1;
end
```

Now we will appreciate that it has paid off to create a `build_while_statement` function. But first we parse the for statement.

```
Parser::parse_for_statement ()
{
  if (!skip_token (Tiny::FOR))
    {
      skip_after_end ();
      return Tree::error ();
    }

  const_TokenPtr identifier = expect_token (Tiny::IDENTIFIER);
  if (identifier == NULL)
    {
      skip_after_end ();
      return Tree::error ();
    }

  if (!skip_token (Tiny::ASSIG))
    {
      skip_after_end ();
      return Tree::error ();
    }

  Tree lower_bound = parse_integer_expression ();

  if (!skip_token (Tiny::TO))
    {
      skip_after_end ();
      return Tree::error ();
    }

  Tree upper_bound = parse_integer_expression ();

  if (!skip_token (Tiny::DO))
    {
      skip_after_end ();
      return Tree::error ();
    }

  enter_scope ();
  parse_statement_seq (&Parser::done_end);

  TreeSymbolMapping for_body_tree_scope = leave_scope ();
```

```
  Tree for_body_stmt = for_body_tree_scope.bind_expr;

  skip_token (Tiny::END);

  // Induction var
  SymbolPtr ind_var
    = query_integer_variable (identifier->get_str (), identifier->get_locus ());

  return build_for_statement (ind_var, lower_bound, upper_bound, for_body_stmt);
}
```

Now `build_for_statement` just creates the statements shown above. The variable of the for statement is commonly known as the *induction variable*.

```
 1 Tree
 2 Parser::build_for_statement (SymbolPtr ind_var, Tree lower_bound,
 3                              Tree upper_bound, Tree for_body_stmt_list)
 4 {
 5   if (ind_var == NULL)
 6     return Tree::error ();
 7   Tree ind_var_decl = ind_var->get_tree_decl ();
 8
 9   // Lower
10   if (lower_bound.is_error ())
11     return Tree::error ();
12
13   // Upper
14   if (upper_bound.is_error ())
15     return Tree::error ();
16
17   // ind_var := lower;
18   TreeStmtList stmt_list;
19
20   Tree init_ind_var = build_tree (MODIFY_EXPR, UNKNOWN_LOCATION,
21                                   void_type_node, ind_var_decl, lower_bound);
22   stmt_list.append (init_ind_var);
23
24   // ind_var <= upper
25   Tree while_condition
26     = build_tree (LE_EXPR, upper_bound.get_locus (), boolean_type_node,
27                   ind_var_decl, upper_bound);
28
29   // for-body
30   // ind_var := ind_var + 1
31   Tree incr_ind_var
32     = build_tree (MODIFY_EXPR, UNKNOWN_LOCATION, void_type_node,
33                   ind_var_decl,
34                   build_tree (PLUS_EXPR, UNKNOWN_LOCATION, integer_type_node,
35                               ind_var_decl,
36                               build_int_cst_type (integer_type_node, 1)));
37
38   // Wrap as a stmt list
```

```
39   TreeStmtList for_stmt_list = for_body_stmt_list;
40   for_stmt_list.append (incr_ind_var);
41
42   // construct the associated while statement
43   Tree while_stmt
44     = build_while_statement (while_condition, for_stmt_list.get_tree ());
45   stmt_list.append (while_stmt);
46
47   return stmt_list.get_tree ();
48 }
```

First we need to initialize the induction variable with the value of the lower bound. We do this by using a `MODIFY_EXPR` tree, the same we used for an assignment statement (lines 20 to 22). We append this initialization to the list of statements that will be the whole for statement tree.

Then we define the condition that we will use for the while. In this case we simply compute `i <= upper` (lines 25 to 27).

Now we synthesize the increment of the induction variable, again we use a `MODIFY_EXPR` and a `PLUS_EXPR` that represents `ind_var := ind_var + 1` (lines 31 to 36). We append this increment to the body of the for statement (lines 39 and 40).

Next is a call to `build_while_statement` with the while condition built above (lines 25 to 27) and the body of the for statement plus the increment of the induction variable (line 44). This will return a tree with the while statement that we append to the initialization of the induction variable (line 45). Finally we return the whole list.

Completion

Ok, so far our front end is more or less complete since it implements all the statements and expressions we defined in part 1. Let's try it with some not-totally trivial examples.

The sum 1 + 2 + ... + 10

```
# for.tiny
var i : int;
var s : int;
s := 0;
for i := 1 to 10 do
  s := s + i;
end
write s;
```

```
$ gcctiny -o for for.tiny
$ ./for
55
```

The square root computed using 100 steps of the Newton method.

```
# sqrt.tiny
var s : float;
s := 2.0;

var i : int;

var x : float;
x := 1.0;
for i := 1 to 100 do
  x := 0.5 * (x + s / x);
end

write x;
```

```
$ gcctiny  -o sqrt sqrt.tiny
$ ./sqrt
1.414214
```

Github

I have uploaded all the code in my github. The code is in `gcc/tiny`.

What next

While this post marks the end of this series there are still a few things possible to do for tiny.

- Define a coercion (similar to that of binary operators) from the right hand side of the assignment to the left hand side, so we can write `x := i;` where `x` is a float and `i` is an int.
- Add the possibility of defining boolean variables (`var b : bool`) along with the two boolean literals `true` and `false`.
- Add array types (e.g `var a : int[10];`) and expressions to reference array elements `a[i]`, array literals like `[1, 2, 3, 4]`. Coercions between non-arrays and arrays, etc.
- Add pointer types (e.g. `var p : ->int`) along with two statements to reserve and free the memory (e.g `new p;` and `delete p;`). Assignment between pointers of the same type. Dereference of pointers (e.g. `->p := 3;`), etc.
- and many, many more

That's all for today.

« A tiny GCC front end – Part 6 A tiny GCC front end – Part 8 »