A tiny GCC front end – Part 8

Jan 30, 2016 • Roger Ferrer Ibáñez • compilers, GCC

Now that we have the basic language set implemented we can consider adding new features to it. Today we will add arrays.

Array type and array values

An important element of programming languages is their type system. Type systems are crucial in the semantics of programming languages and are an actively researched topic nowadays. tiny, so far, has a very simple type system: there are only four types (int, float, boolean and string). We can express lots of things already with those types but it may fall short in some contexts.

A *type system* is a set of *types* along with the rules that govern them. An element of the type system, i.e. a type, will be denoted by τ. As we said, tiny has four types.

```
τ → int
  | float
  | bool
  | string
```

A type is a set of *values*: **int** values are the 32 bit signed integers, **float** values are the reals encoded by IEEE 754 Binary32, **bool** has only two values true or false and values of **string** type are (possibly empty) finite sequences of characters.

Now we want to add an *array type*. An array type has a *size* and an *element type*. The size is an integer expression of the language, that we will denote as ε that evaluates to a positive (nonzero) integer.

```
τ → array ε τ
```

This means that our typesystem has a type array constructed using an integer expression ε (the size) and a type τ (the element type).

After this addition, our typesystem looks like this.

τ → **int**
| **float**
| **bool**
| **string**
| **array** ε τ

What are, thus, the values of a type **array** ε τ? A value of array type is a set of values of type τ called the *elements* of the array. There is an integer associated to each element, called the *index*. The set of indexes of the elements is such that they form an ascending sequence, where each index is the previous one plus one. The first index is called the lower bound (say it L) and the last one is the upper bound (say it U). This way it holds that U - L + 1 = ε.

I know that at this point this seems unnecessarily theoretic but let's make a simple example. Consider *array 3 float*. A possible array value could be the following one, where L = 0 and U = 2.

⟨0 → 1.2, 1 → 2.3, 2 → 2.3⟩

for another example where L = 4 and U = 6

⟨4 → 1.2, 5 → 2.3, 6 → 2.3⟩

The indexes form a growing sequence wherer each index equals the previous one plus one. The following would not be the value of an array.

⟨12 → 1.2, 25 → 2.3, 42 → 2.3⟩

Syntax

We will extend the rule of types of tiny to let us define a variable of array type.

⟨type⟩ → **int** | **float** | ⟨type⟩ **[** ⟨expression⟩ **]** | ⟨type⟩ **(** ⟨expression⟩ **:** ⟨expression⟩ **)**

We will also need to extend expressions so we can designate one of the elements of the array.

⟨primary⟩ → **(** expression **)**
| ⟨identifier⟩
| ⟨integer-literal⟩
| ⟨float-literal⟩
| ⟨string-literal⟩

| ⟨array-element⟩

⟨array-element⟩ → ⟨primary⟩ **[** ⟨expression⟩ **]**

Semantics

A ⟨type⟩ of the form

⟨type⟩ **[** ⟨expression⟩ **]**

designates an array type. If ⟨type⟩ is not an array then the designated type is just **array** ⟨type⟩ ⟨expression⟩. The set of indexes range from 0 to ⟨expression⟩ minus one.

```
var a : int[10];        # array 10 int
```

Things are a bit more complicated if ⟨type⟩ is an array because now there are two possible interpretations. In the comments below, parentheses are used only to express grouping

```
var b : int[10][20];  # array 10 (array 20 int)
                      #    or
                      # array 20 (array 10 int) ?
```

We will chose the first interpretation. Some programming languages, like Fortran, choose the second one.

For the case when ⟨type⟩ is an array, let's assume it is of the form **array** ε_0 τ_0. Then the designated type will be **array** ε_0 (**array** τ_0 ⟨expression⟩)

The other syntax is similar.

⟨type⟩ → ⟨type⟩ **(** ⟨expression$_0$⟩ **:** ⟨expression$_1$⟩ **)**

Now ε is ⟨expression$_1$⟩ - ⟨expression$_0$⟩ + 1 and the indexes of the array range from ⟨expression$_0$⟩ to ⟨expression$_1$⟩ (both ends included). ⟨expression$_1$⟩ must be larger or equal than ⟨expression$_0$⟩, otherwise this is an error.

```
var a1 : int(0:9);        # array 10 int
var b1 : int(0:9)(1:20); # array 10 (array 20 int)
var c1 : int(5:5);        # array 1 int
var d1 : int(-5:-3)       # array 3 int
```

A ⟨primary⟩ of the form

⟨array-element⟩ → ⟨primary⟩ **[** ⟨expression⟩ **]**

designates a single element of ⟨primary⟩. The type of ⟨primary⟩ must be array, otherwise this is an error. The ⟨expression⟩ must be an expression of integer type the value of which must be contained in the range of indexes of the array type, otherwise this is an error. The type of an array element is the same as the element type of the array.

Given the declarations of `a1`, `b1`, `c1`, `d1` above, valid array elements are.

```
a1[0]
a1[9]
b1[0][1]
b1[3][4]
b1[9][20]
c1[5]
d1[-5]
d1[-4]
d1[-3]
```

Primaries of the form ⟨identifier⟩ and ⟨array-element⟩ can be used in the left hand side of an assignment and in the read statement. We will call this subset of expressions as *variables*. Some programming languages, like C and C++, name these expressions *lvalues* (or L-values) for historical reasons: an lvalue can appear in the *left* hand side of an assignment.

⟨assignment⟩ → ⟨variable⟩ **:=** ⟨expression⟩ **;**
⟨read⟩ → **read** ⟨variable⟩ **;**

⟨variable⟩ → ⟨identifier⟩
| ⟨array-element⟩

```
a1[1] := 3;
read a1[2];
```

This opens up many possibilities. For instance now we can write a tiny program (`bubble.tiny`) that sorts a given set of numbers.

```
# bubble.tiny
var n : int;
write "Enter the number of integers:";
read n;

write "Enter the integers:";

var i : int;
var a : int[n];
for i := 0 to n - 1
do
   read a[i];
end

# Very inefficient bubble sort used
```

```
# only as an example

var swaps : int;
swaps := 1;
while swaps > 0
do
  swaps := 0;
  for i := 1 to n - 1
  do
     if a[i - 1] > a[i]
     then
       var t : int;
       t := a[i-1];
       a[i-1] := a[i];
       a[i] := t;
       swaps := swaps + 1;
     end
  end
end

write "Sorted numbers:";

for i := 0 to n - 1
do
   write a[i];
end
```

Implementation

Adding support for arrays to our front end is not too hard.

Minor issue first

Before we proceed we need to fix an issue that may cause us problems when we play with arrays: We want all the declarations have a `DECL_CONTEXT`. Current code only sets it for `LABEL_DECL` but all declarations (except those that are global) should have some `DECL_CONTEXT`. In our case `VAR_DECL`s and the `RESULT_DECL` of main are missing the `DECL_CONTEXT`. We have to set it to the `FUNCTION_DECL` of the main function (this effectively makes them local variables of the main function).

```
diff --git a/gcc/tiny/tiny-parser.cc b/gcc/tiny/tiny-parser.cc
index 709b517..0ce295d 100644
@@ -242,6 +242,7 @@ Parser::parse_program ()
   // Append "return 0;"
   tree resdecl
     = build_decl (UNKNOWN_LOCATION, RESULT_DECL, NULL_TREE, integer_type_node);
+  DECL_CONTEXT (resdecl) = main_fndecl;
   DECL_RESULT (main_fndecl) = resdecl;
   tree set_result
     = build2 (INIT_EXPR, void_type_node, DECL_RESULT (main_fndecl),
@@ -455,6 +456,7 @@ Parser::parse_variable_declaration ()
   Tree decl = build_decl (identifier->get_locus (), VAR_DECL,
```

```
                              get_identifier (sym->get_name ().c_str ()),
                              type_tree.get_tree ());
+  DECL_CONTEXT (decl.get_tree()) = main_fndecl;

   gcc_assert (!stack_var_decl_chain.empty ());
   stack_var_decl_chain.back ().append (decl);
```

Lexer

For the lexer we only have to add three tokens **[** and **]**. The remaining punctuation required for arrays **(**, **)** and **:** were already in tiny.

```
diff --git a/gcc/tiny/tiny-token.h b/gcc/tiny/tiny-token.h
index d469980..2d81386 100644
@@ -40,6 +40,8 @@ namespace Tiny
   TINY_TOKEN (INTEGER_LITERAL, "integer literal")                              \
   TINY_TOKEN (REAL_LITERAL, "real literal")                                    \
   TINY_TOKEN (STRING_LITERAL, "string literal")                                \
+  TINY_TOKEN (LEFT_SQUARE, "[")                                                \
+  TINY_TOKEN (RIGHT_SQUARE, "]")                                               \
                                                                                \
   TINY_TOKEN_KEYWORD (AND, "and")                                              \
   TINY_TOKEN_KEYWORD (DO, "do")                                                \
```

```
diff --git a/gcc/tiny/tiny-lexer.cc b/gcc/tiny/tiny-lexer.cc
index 1b9c8be..b67470d 100644
@@ -223,6 +223,12 @@ Lexer::build_token ()
             }
           continue;
           break;
+       case '[':
+         current_column++;
+         return Token::make (LEFT_SQUARE, loc);
+       case ']':
+         current_column++;
+         return Token::make (RIGHT_SQUARE, loc);
         }

       // ***************************
```

Parser

Array type

First let's see how to parse a type that designates an array. In member function Parser::parse_type we cannot just return the parsed type. Instead we will keep it.

```
@@ -517,24 +534,91 @@ Parser::parse_type ()
 {
```

```
    const_TokenPtr t = lexer.peek_token ();

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

Now we will start parsing the indexes ranges. We will have a list of pairs of expressions, each pair denoting the lower and the upper indexes of the array type. For arrays of the form `[e]` we will set the lower bound to zero and the upper bound to the `e - 1`. For arrays of the form `(e0:e1)`, the lower and the upper will be `e0` and `e1` respectively.

```
+
+   typedef std::vector<std::pair<Tree, Tree> > Dimensions;
+   Dimensions dimensions;
+
+   t = lexer.peek_token ();
+   while (t->get_id () == Tiny::LEFT_PAREN || t->get_id () == Tiny::LEFT_SQUARE)
+     {
+       lexer.skip_token ();
+
+       Tree lower_bound, upper_bound;
+       if (t->get_id () == Tiny::LEFT_SQUARE)
+        {
+          Tree size = parse_integer_expression ();
+          skip_token (Tiny::RIGHT_SQUARE);
+
+          lower_bound = Tree (build_int_cst_type (integer_type_node, 0),
+                              size.get_locus ());
+          upper_bound
+            = build_tree (MINUS_EXPR, size.get_locus (), integer_type_node,
+                          size, build_int_cst (integer_type_node, 1));
+
+        }
+       else if (t->get_id () == Tiny::LEFT_PAREN)
+        {
+          lower_bound = parse_integer_expression ();
+          skip_token (Tiny::COLON);
```

```
+
+         upper_bound = parse_integer_expression ();
+         skip_token (Tiny::RIGHT_PAREN);
+       }
+      else
+       {
+         gcc_unreachable ();
+       }
+
+      dimensions.push_back (std::make_pair (lower_bound, upper_bound));
+      t = lexer.peek_token ();
+    }
```

Now we can start building the array type.

```
 1 +  for (Dimensions::reverse_iterator it = dimensions.rbegin ();
 2 +       it != dimensions.rend (); it++)
 3 +    {
 4 +      it->first = Tree (fold (it->first.get_tree ()), it->first.get_locus ());
 5 +      it->second
 6 +       = Tree (fold (it->second.get_tree ()), it->second.get_locus ());
 7 +
 8 +      Tree range_type
 9 +       = build_range_type (integer_type_node, it->first.get_tree (),
10 +                           it->second.get_tree ());
11 +      type = build_array_type (type.get_tree (), range_type.get_tree ());
12 +    }
13 +
14 +  return type;
```

Due to the semantics of the array types described above, we have to traverse the list in reverse order. We get the lower and upper expressions and we *fold* it (lines 4 to 5). This GCC function will attempt to simplify the expression if possible. For instance 1+2*3 will become 7. Now we build a GCC range type. A range type is a type the values of which are integers in the specified range. In this case we use the lower and the upper to create the range type (lines 8 to 10). A range type is represented as a GENERIC tree with tree code `RANGE_TYPE`. Once we have this range type, we take the current type (which may be at this point an integer type, a float type or another array type) and the range type to build an array type (line 11). An array type is represented as a GENERIC tree with three code `ARRAY_TYPE`.

> Note that we currently do not check that the ε of the array type is actually a positive, nonzero, integer value. If the bounds of the array are constant, such error can be detected at compile time (the earlier an error is detected the better). If the bounds are non-constant then the semantics of the language should specify what to do during the execution of the program. Tiny semantics simply say that it is an error. Since we have not clarified what “to be an error” is, we will not do anything special yet.

Array element

Now we have to add support for array elements in expressions. Recall that we use a Pratt parser to recognize them. We can recognize an array element by just acting as if **[** were a binary operation with very high priority.

```
diff --git a/gcc/tiny/tiny-parser.cc b/gcc/tiny/tiny-parser.cc
index 0ce295d..37c6397 100644
@@ -1157,6 +1220,8 @@ enum binding_powers
   // Highest priority
   LBP_HIGHEST = 100,

+  LBP_ARRAY_REF = 80,
+
   LBP_UNARY_PLUS = 50,  // Used only when the null denotation is +
   LBP_UNARY_MINUS = LBP_UNARY_PLUS, // Used only when the null denotation is -

@@ -1189,6 +1254,8 @@ Parser::left_binding_power (const_TokenPtr token)
 {
   switch (token->get_id ())
     {
+    case Tiny::LEFT_SQUARE:
+      return LBP_ARRAY_REF;
```

This will require a binary handler, like other infix operators.

```
@@ -116,7 +117,9 @@ private:
   BINARY_HANDLER (greater_equal, GREATER_OR_EQUAL)                              \
                                                                                 \
   BINARY_HANDLER (logical_and, AND)                                             \
-  BINARY_HANDLER (logical_or, OR)
+  BINARY_HANDLER (logical_or, OR)                                               \
+                                                                                \
+  BINARY_HANDLER (array_ref, LEFT_SQUARE)

 #define BINARY_HANDLER(name, _)                                                 \
   Tree binary_##name (const_TokenPtr tok, Tree left);
```

The binary handler is actually rather straightforward.

```
 1 Tree Parser::binary_array_ref(const const_TokenPtr tok, Tree left) {
 2   Tree right = parse_integer_expression();
 3   if (right.is_error())
 4     return Tree::error();
 5
 6   if (!skip_token(Tiny::RIGHT_SQUARE))
 7     return Tree::error();
 8
 9   if (!is_array_type(left.get_type())) {
10     error_at(left.get_locus(), "does not have array type");
11     return Tree::error();
12   }
13
14   Tree element_type = TREE_TYPE(left.get_type().get_tree());
15
```

```
16   return build_tree(ARRAY_REF, tok->get_locus(), element_type, left, right,
17                     Tree(), Tree());
18 }
```

Recall that a binary handler has the lexer positioned right after the infix operator. This means that we have already consumed **[**. So we have to parse the integer expression enclosed by the square brackets (line 4). Recall that any token unknown to the Pratt parser has the lowest possible *binding power*, this means that parsing the integer expression will stop when it encounters the **]**. This behaviour is actually the one we want. We still have to consume the **]** (line 8). Now we verify if the left operand has array type (line 9). If it does not, this is an error. If it does, we compute the type of the array element. To do this we have to use the accessor `TREE_TYPE` from GCC which given an `ARRAY_TYPE` will return its element type (line 14). Finally we build the GENERIC tree `ARRAY_REF` that repreents an access the array element (line 16).

Checking if a tree in GENERIC represents an array type is done using this auxiliar function.

```
bool
is_array_type (Tree type)
{
  gcc_assert (TYPE_P (type.get_tree ()));
  return type.get_tree_code () == ARRAY_TYPE;
}
```

> Likewise with ε, we are not verifying that the expression of the array element evaluates to an integer contained in the range of indexes of the declared array. Recall that the semantics of tiny are not complete enough regarding errors.

Final touches

As we said above we allow variables and array elements in the expression of a read statement and in the left hand side of an assignment. Let's first create a couple of functions that expression r that check this for us.

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

  if (expr.get_tree_code () != VAR_DECL && expr.get_tree_code () != ARRAY_REF)
    {
      error_at (expr.get_locus (),
                "does not designate a variable or array element");
      return Tree::error ();
    }
  return expr;
}
```

```
Tree
Parser::parse_lhs_assignment_expression ()
{
  return parse_expression_naming_variable();
}
```

Since we allow the same thing in both cases, `parse_lhs_assignment_expression` just forwards to `parse_expression_naming_variable`. Now we can update parse_assignment.

```
@@ -572,24 +656,11 @@
 Tree
 Parser::parse_assignment_statement ()
 {
-  const_TokenPtr identifier = expect_token (Tiny::IDENTIFIER);
-  if (identifier == NULL)
-    {
-      skip_after_semicolon ();
-      return Tree::error ();
-    }
-
-  SymbolPtr sym
-    = query_variable (identifier->get_str (), identifier->get_locus ());
-  if (sym == NULL)
-    {
-      skip_after_semicolon ();
-      return Tree::error ();
-    }
+  Tree variable = parse_lhs_assignment_expression ();

-  gcc_assert (!sym->get_tree_decl ().is_null ());
-  Tree var_decl = sym->get_tree_decl ();
+  if (variable.is_error ())
+    return Tree::error ();

   const_TokenPtr assig_tok = expect_token (Tiny::ASSIG);
   if (assig_tok == NULL)
@@ -606,18 +677,17 @@ Parser::parse_assignment_statement ()

   skip_token (Tiny::SEMICOLON);

-  if (var_decl.get_type () != expr.get_type ())
+  if (variable.get_type () != expr.get_type ())
     {
       error_at (first_of_expr->get_locus (),
-                "cannot assign value of type %s to variable '%s' of type %s",
-                print_type (expr.get_type ()), sym->get_name ().c_str (),
-                print_type (var_decl.get_type ()));
+                "cannot assign value of type %s to a variable of type %s",
+                print_type (expr.get_type ()),
+                print_type (variable.get_type ()));
       return Tree::error ();
     }
```

```
   Tree assig_expr = build_tree (MODIFY_EXPR, assig_tok->get_locus (),
-                                void_type_node, var_decl, expr);
+                                void_type_node, variable, expr);

   return assig_expr;
 }
```

Language hook

If we want to use arrays with non-constant size, GCC will invoke a language hook when internally computing the size of the array. This is for those cases where the language supports variable-sized types in a global scope. In this case the hook must return true, false otherwise.

Since in tiny where everything is conceptually inside an implicit main function, the binding must return false.

Our hook, currently crashes the compiler, so we need to adjust it first. Recall that this hook is in `tiny1.cc`.

```
diff --git a/gcc/tiny/tiny1.cc b/gcc/tiny/tiny1.cc
index dcd6f45..3a92eaa 100644
@@ -159,8 +159,7 @@ tiny_langhook_builtin_function (tree decl)
 static bool
 tiny_langhook_global_bindings_p (void)
 {
-  gcc_unreachable ();
-  return true;
+  return false;
 }
```

Trying it

```
# array.tiny
var a : int[10];

a[1] := 11;
a[2] := 22;

write a[1];
write a[2];

var b : int(2:4);

b[2] := 55;
b[3] := 66;
b[4] := 77;

write b[2];
write b[3];
write b[4];
```

```
$ gcctiny -o array array.tiny
$ ./array
11
22
55
66
77

# matrix.tiny
var a : int[10][20];

a[1][2] := 11;
a[2][3] := 22;

write a[1][2];
write a[2][3];

$ gcctiny -o matrix matrix.tiny
$ ./matrix
11
22
```

Let's try the `bubble.tiny` program shown earlier.

```
$ gcctiny -o bubble bubble.tiny
$ ./bubble
Enter the number of integers:
4
Enter the integers:
1 3 2 4
Sorted numbers:
1
2
3
4
```

Yay!

That's all for today.

« A tiny GCC front end – Part 7 A tiny GCC front end – Part 9 »