A tiny GCC front end – Part 11

Sep 5, 2016 • Roger Ferrer Ibáñez • compilers, GCC

Our tiny language features a few types: `int`, `float`, `bool`, `string` and arrays of those types. We can even declare new type names based on other types but it still missing a record type. Today we will address this.

Record types

Before we added arrays to tiny, the value of our variables was simple, atomic, non-structured at all. When we introduced arrays we let a single variable denote several values, but all the values have to be of the same type. We are missing the possibility of representing several values of different types: we need a record type (or tuple type).

Record types are constructed using an ordered set of types. We will need to refer the elements of such set in some way. We could access the field using some index derived from its order (e.g. the first element could be indexed using a zero, the second element using one). Usually, though, the elements of the record are given names, so the name can be used to refer to the element. While both approaches are feasible using names is probably easier for the programmer. In tiny we will use names.

Syntax

Similar to arrays, we will need special syntax to define a record type. A record type is made of a sequence of pairs of names and types that we will call *fields*. First let's see how to declare a field.

⟨field-declaration⟩ → ⟨identifier⟩ **:** ⟨type⟩ **;**

A field declaration has the same syntax as a variable/type declaration but without an initial `var`/`type`. A keyword is not needed because a field declaration will always appear inside a record type.

Recall that * means the previous element of the language repeated zero or more times

⟨type⟩ → **record** ⟨field-declaration⟩ * **end**

For instance we can declare a variable with a record type made of two floats x and y, like this.

```
var one_point : record x : float; y : float; end;
```

In chapter 10 we introduced type declarations, so we can declare a `point` type of record type.

```
type point : record x : float; y : float; end;
```

This way we can now declare several points without having to repeat the record type. There is a reason for this that we will discuss below.

```
var p1 : point;
var p2 : point;
```

We need a way to access the field of a record type inside an expression or in the left hand side of an assignment. We will add a new ⟨primary⟩ expression.

⟨primary⟩ → **(** expression **)**
| ⟨identifier⟩
| ⟨integer-literal⟩
| ⟨float-literal⟩
| ⟨string-literal⟩
| ⟨array-element⟩
| ⟨field-access⟩
⟨field-access⟩ → ⟨primary⟩ **.** ⟨identifier⟩

For instance, fields of `p1` and `p2` can be accessed like this.

```
p1.x := 1.2;
if p2.y < 3.4 then
  p1.x := p2.y + 3.4;
end
```

We still need to clarify the priority between an ⟨array-element⟩ and a ⟨field-access⟩. The following expressions (assuming they are valid given appropiate types for the variable `a` and the field `b`)

```
a.b[1]
a[1].b
a[1].b[2]
```

are to be interpreted like

```
(a).b[1]
(a[1]).b
((a[1]).b)[2]
```

Semantics

A record type is a type the values of which is the cartesian product of the values that can be represented by the fields of a record type. A record type with n fields named ϕ_0, ϕ_1, …, ϕ_{n-1} where

each field ϕ_i has an associated type τ_i will be able to represent a value $(\varepsilon_0, \varepsilon_1, \ldots, \varepsilon_{n-1})$ where each ε_i is a value of the type τ_i. Given a value of record type, we can select a single value of it, in our case using its name.

Two values of record type are the same only if they come from the same declaration. This means that `p1` and `p2` below have different types because their record types come from different declarations, even if their sequences of fields are the same.

```
var p1 : record x : float; y : float; end;
var p2 : record x : float; y : float; end;
```

Conversely, below `p1` and `p2` are the same, because their record type comes from the same declaration.

```
type point : record x : float; y : float; end;
var p1 : point;
var p2 : point;
```

This kind of type equivalence is called `equivalence by name` in contrast to `equivalence by structure`. Both have advantages and drawbacks but most programming languages choose the former.

As we discussed in chapter 10, in a type declaration, we cannot use the type being declared in the type part. So this will be invalid.

```
type invalid : record a : invalid; end;
```

The name of each field must be unique inside a record. It is possible to have a field with record type.

An expression of the form ⟨primary⟩ **.** ⟨identifier⟩ is a field access. The primary expression must have record type and ⟨identifier⟩ must be the name of a field of that record type. The type of a field access is the ⟨type⟩ as the type of the corresponding field declaration. A field access can be used as the left hand side operator of an assignment and can be used as the operand of a `read` statement.

Implementation

Now that we have a specification of this extension, we can start implementing it.

Lexer

We need to recognize two new tokens: a new keyword **`record`** and the dot operator. So we add bot to our set of tokens.

```
diff --git a/gcc/tiny/tiny-token.h b/gcc/tiny/tiny-token.h
index b1008a6..ed6961c 100644
@@ -42,6 +42,7 @@ namespace Tiny
```

```
   TINY_TOKEN (STRING_LITERAL, "string literal")                          \
   TINY_TOKEN (LEFT_SQUARE, "[")                                          \
   TINY_TOKEN (RIGHT_SQUARE, "]")                                         \
+  TINY_TOKEN (DOT, ".")                                                  \
                                                                          \
   TINY_TOKEN_KEYWORD (AND, "and")                                        \
   TINY_TOKEN_KEYWORD (BOOL, "bool")                                      \
@@ -56,6 +57,7 @@ namespace Tiny
   TINY_TOKEN_KEYWORD (NOT, "not")                                        \
   TINY_TOKEN_KEYWORD (OR, "or")                                          \
   TINY_TOKEN_KEYWORD (READ, "read")                                      \
+  TINY_TOKEN_KEYWORD (RECORD, "record")                                  \
   TINY_TOKEN_KEYWORD (THEN, "then")                                      \
   TINY_TOKEN_KEYWORD (TO, "to")                                          \
   TINY_TOKEN_KEYWORD (TRUE_LITERAL, "true")                              \
```

Our existing machinery will handle **`record`**, so only the dot must be tokenized. Given the current specification, the dot is relatively simple as long as it is not followed by a number, a `.` in the code will be the token `DOT`. This restriction makes sense as we want `.1` to be a `FLOAT_LITERAL` not a `DOT` followed by an `INTEGER_LITERAL`.

```
diff --git a/gcc/tiny/tiny-lexer.cc b/gcc/tiny/tiny-lexer.cc
index b67470d..a4268c2 100644
@@ -229,6 +229,13 @@ Lexer::build_token ()
        case ']':
          current_column++;
          return Token::make (RIGHT_SQUARE, loc);
+       case '.':
+         if (!ISDIGIT(peek_input ()))
+           {
+             // Only if followed by a non number
+             current_column++;
+             return Token::make (DOT, loc);
+           }
        }

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

Parse a record type

To parse a record type we first have to be able to parse a field declaration. It is pretty straightforward. GENERIC represents field declarations using a `FIELD_DECL` tree which simply has the name of the field and its type. We also have to make sure to mark the field addressable otherwise the **`read`** statement will not work on fields. Note also that we pass a vector of field names so we can diagnose repeated field names (I'm using a vector because the number of fields is usually small and it does not pay to use a more sophisticated data structure).

```
+Tree
+Parser::parse_field_declaration (std::vector<std::string> &field_names)
+{
+  // identifier ':' type ';'
```

```
+  const_TokenPtr identifier = expect_token (Tiny::IDENTIFIER);
+  if (identifier == NULL)
+    {
+      skip_after_semicolon ();
+      return Tree::error ();
+    }
+
+  skip_token (Tiny::COLON);
+
+  Tree type = parse_type();
+
+  skip_token (Tiny::SEMICOLON);
+
+  if (type.is_error ())
+    return Tree::error ();
+
+  if (std::find (field_names.begin (), field_names.end (),
+                 identifier->get_str ())
+      != field_names.end ())
+    {
+      error_at (identifier->get_locus (), "repeated field name");
+      return Tree::error ();
+    }
+  field_names.push_back (identifier->get_str ());
+
+  Tree field_decl
+    = build_decl (identifier->get_locus (), FIELD_DECL,
+                  get_identifier (identifier->get_str ().c_str ()),
+                  type.get_tree());
+  TREE_ADDRESSABLE (field_decl.get_tree ()) = 1;
+
+  return field_decl;
+}
```

Now that we can parse a field declaration, let's parse a record type. First lets extend `parse_type` so it forwards to `parse_record` when it finds the token `RECORD`.

```
@@ -630,6 +720,9 @@ Parser::parse_type ()
           type = TREE_TYPE (s->get_tree_decl ().get_tree ());
       }
       break;
+    case Tiny::RECORD:
+      type = parse_record ();
+      break;
     default:
       unexpected_token (t);
       return Tree::error ();
```

Parsing a record type is not particularly complex. Once we have skipped the **`record`** keyword we keep parsing field declarations until we find an **`end`** keyword. A record type in GENERIC is represented using a `RECORD_TYPE` tree, so we will have to create first an empty `RECORD_TYPE` tree. Field declarations must have their `DECL_CONTEXT` set to this `RECORD_TYPE` (so they know of which

record type they are fields). The set of fields in a `RECORD_TYPE` is chained using `TREE_CHAIN`. The code simply remembers the first field and the last one so it can chain each field with the previous one. Finally the first field is used to set the `TYPE_FIELDS` attribute of the `RECORD_TYPE`. At this point we also need to request to GCC to lay out this type. The reason is that a `RECORD_TYPE` will have to be represented in memory in a way that can hold all the field values, the function `layout_type` makes sure each field gets the appropiate location in the record type.

```
+Tree
+Parser::parse_record ()
+{
+  // "record" field-decl* "end"
+  const_TokenPtr record_tok = expect_token (Tiny::RECORD);
+  if (record_tok == NULL)
+    {
+      skip_after_semicolon ();
+      return Tree::error ();
+    }
+
+  Tree record_type = make_node(RECORD_TYPE);
+  Tree field_list, field_last;
+  std::vector<std::string> field_names;
+
+  const_TokenPtr next = lexer.peek_token ();
+  while (next->get_id () != Tiny::END)
+    {
+      Tree field_decl = parse_field_declaration (field_names);
+
+      if (!field_decl.is_error ())
+       {
+         DECL_CONTEXT (field_decl.get_tree ()) = record_type.get_tree();
+         if (field_list.is_null ())
+           field_list = field_decl;
+         if (!field_last.is_null ())
+           TREE_CHAIN (field_last.get_tree ()) = field_decl.get_tree ();
+         field_last = field_decl;
+       }
+      next = lexer.peek_token ();
+    }
+
+  skip_token (Tiny::END);
+
+  TYPE_FIELDS (record_type.get_tree ()) = field_list.get_tree();
+  layout_type (record_type.get_tree ());
+
+  return record_type;
+}
```

Parse a field access

Parsing a field access is done by handling the dot as a binary operator with very high priority. So we assign it a high left binding power.

```
@@ -1324,6 +1417,8 @@ enum binding_powers
   // Highest priority
   LBP_HIGHEST = 100,

+  LBP_DOT = 90,
+
   LBP_ARRAY_REF = 80,

   LBP_UNARY_PLUS = 50,  // Used only when the null denotation is +
@@ -1358,6 +1453,9 @@ Parser::left_binding_power (const_TokenPtr token)
 {
   switch (token->get_id ())
     {
+    case Tiny::DOT:
+      return LBP_DOT;
+    //
     case Tiny::LEFT_SQUARE:
       return LBP_ARRAY_REF;
     //
```

We will use a convenience function `is_record_type` with the obvious meaning.

```
+bool
+is_record_type (Tree type)
+{
+  gcc_assert (TYPE_P (type.get_tree ()));
+  return type.get_tree_code () == RECORD_TYPE;
+}
```

In GENERIC a field access is represented with a tree of kind `COMPONENT_REF`, where the first tree is an tree expression of record type and the second tree is a `FIELD_DECL`. Parsing a field access involves just checking that the left expression has a record type and the dot is followed by an identifier that is the name of a field of that record type. Recall that the list of fields of a `RECORD_TYPE` is available in the `TYPE_FIELDS` attribute. We traverse each `FIELD_DECL` chaining through `TREE_CHAIN`. Like all other declarations in GENERIC, a `FIELD_DECL` has a `DECL_NAME` which contains an attribute `IDENTIFIER_POINTER` where we will find the name of the field. If we do not find a field with the given name, then this is an error, otherwise we create a tree COMPONENT_REF using the left tree (that we checked it is of record type) and the appropiate `FIELD_DECL`.

```
+Tree
+Parser::binary_field_ref (const const_TokenPtr tok, Tree left)
+{
+  const_TokenPtr identifier = expect_token (Tiny::IDENTIFIER);
+  if (identifier == NULL)
+    {
+      return Tree::error ();
+    }
+
+  if (!is_record_type (left.get_type ()))
+    {
+      error_at (left.get_locus (), "does not have record type");
+      return Tree::error ();
```

```
+    }
+
+  Tree field_decl = TYPE_FIELDS (left.get_type ().get_tree ());
+  while (!field_decl.is_null ())
+    {
+      Tree decl_name = DECL_NAME (field_decl.get_tree ());
+      const char *field_name = IDENTIFIER_POINTER (decl_name.get_tree ());
+
+      if (field_name == identifier->get_str ())
+       break;
+
+      field_decl = TREE_CHAIN (field_decl.get_tree ());
+    }
+
+  if (field_decl.is_null ())
+    {
+      error_at (left.get_locus (),
+                "record type does not have a field named '%s'",
+                identifier->get_str ().c_str ());
+      return Tree::error ();
+    }
+
+  return build_tree (COMPONENT_REF, tok->get_locus (),
+                     TREE_TYPE (field_decl.get_tree ()), left, field_decl,
+                     Tree ());
+}
```

Finally we must update `parse_expression_naming_variable` because a COMPONENT_REF tree also names a variable. This way we can put it in the left hand side of an assignment or as the operand of a **`read`** statement.

```
@@ -1884,10 +2022,11 @@ Parser::parse_expression_naming_variable ()
   if (expr.is_error ())
     return expr;

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

Smoke test

And we are done. Let's try a simple program.

```
# struct.tiny
type my_tuple : record
```

```
  a : int;
  b : float;
end;

var x : my_tuple;

write "Enter an integer:";
read x.a;
write "Enter a float:";
read x.b;

x.a := x.a + 1;
x.b := x.b + 3.4;

write "Tuple:";
write "  x.a=";
write x.a;
write "  x.b=";
write x.b;
```

```
$ ./gcctiny -o test struct.tiny
$ ./test
Enter an integer:
1
Enter a float:
1.23
Tuple:
  x.a=
2
  x.b=
4.630000
```

Yay!

That's all for today

« A tiny GCC front end – Part 10 Exploring AArch64 assembler – Chapter 1 »