A tiny GCC front end – Part 5

Jan 16, 2016 • Roger Ferrer Ibáñez • compilers, GCC • gcc, tiny

In the last installment of this series we saw how to verify that the sequence of tokens of the input is syntactically valid. Today we will see what we need to give it meaning.

Semantics concerns to the meaning of the program. This means that our sequence of tokens, once they follow some syntax, have meaning in the context of the programming language. In part 1 we gave a more or less abstract semantics of tiny. Now, as compiler writers, it is up to us to materialize such semantics in an implementation that fulfills it.

GENERIC

If you recall part 2, the final goal of our front end is creating a GENERIC tree and handing it to the rest of the compiler. Let's talk about bit more about GENERIC trees.

GENERIC trees are represented using the type `tree`. A tree can be a `NULL_TREE` or point to an actual tree. Each tree has a *tree code* that is specified at the moment of creation. Given a tree we can use the macro `TREE_CODE` to get the tree code. Most trees, but not all, have a location that we can obtain using the macro `EXPR_LOC`, if it does not have location it will return `UNKNOWN_LOCATION`.

Trees are created using macros `build0`, `build1`, `build2`, ..., `build5`. The first parameter of each `buildN` macro is the tree code and the remaining `N` arguments are trees, called the *operands*. As an alternative `build0_loc`, `build1_loc`, `build2_loc`, ..., `build5_loc` can be used instead to create a tree along with a location. The location goes in the first argument and the remaining arguments are the same as in `buildN`.

Despite their name, GENERIC trees do not collectively form a tree but a graph. This happens because it is not an error that a tree appears as the operand of two or more trees.

Each tree of a specific tree code may have associated several attributes. These attributes are accessed using macros. Most of these macros expand in a way that can be used to set the attribute to the tree. So given a tree `t`, an attribute can be queried doing `SOME_TREE_PROPERTY(t)` and can be set doing `SOME_TREE_PROPERTY(t) = property`. These attributes are of different nature, sometimes are other trees, sometimes are boolean values (zero or nonzero), etc.

GENERIC trees are used to represent many aspects of a program but there are three important classes of trees: declarations, expressions and types.

Declarations are used to tell the compiler about the existence of something. Variables go into a tree with code `VAR_DECL`. Labels of the program (used for `goto`s) go into a `LABEL_DECL`. tiny does not have functions explicitly but if we declare a function, it goes into a `FUNCTION_DECL` and each of its parameters would be represented using `PARM_DECL`.

Expressions represent trees that can be evaluated. There are a lot of tree codes related to expressions that we will see later. One distinguished node, `error_mark_node`, will be used as a marker for erroneous trees that may appear during semantic analysis. Given a tree `t`, the macro `error_operand_p(t)` returns true if `t` is `error_mark_node`.

Finally, types represent data types. They are represented as trees because most type systems have a recursive structure that fits well in a graph-like structure like GENERIC. Type trees are heavily reused in GENERIC. In tiny we will need tree types for int, float, boolean and strings. Expressions and declarations have type and it can be accessed using `TREE_TYPE`.

GENERIC is an intermediate representation that is heavily biased towards a C model of execution (like a relatively high-level assembler). The reason is that GCC was originally a C compiler that later on was extended to support other programming languages. Imperative programming languages, like tiny, fit relatively well in GENERIC. Other programming languages, like functional ones, do not fit so well in GENERIC and a front end for such languages likely uses its own representation that ends being lowered to GENERIC.

Almost GENERIC

Tiny is so simple that we can use GENERIC trees almost directly. Almost, because not all GENERIC trees may have locations so we will pair a tree and a location, to make sure we have a location. Getting the GENERIC tree is, then, as simple as requesting the tree member of the pair. We want to have location in all trees for diagnostic purposes.

In order to ease using GENERIC trees, we will use a `Tree` class (mind the uppercase) that will be a very thin wrapper to `tree`.

```
#include "tree.h"

namespace Tiny
{
struct Tree
{
public:
  Tree () : t (NULL_TREE), loc (UNKNOWN_LOCATION) {}
  Tree (tree t_) : t (t_), loc (EXPR_LOCATION (t)) {}
  Tree (tree t_, location_t loc_) : t (t_), loc (loc_) {}
  Tree (Tree t_, location_t loc_) : t (t_.get_tree ()), loc (loc_) {}

  location_t
  get_locus () const
  {
    return loc;
  }

  void
  set_locus (location_t loc_)
  {
    loc = loc_;
  }

  tree
  get_tree () const
  {
    return t;
  }

  tree_code
  get_tree_code () const
  {
    return TREE_CODE (t);
  }

  void
  set_tree (tree t_)
  {
    t = t_;
  }

  bool
  is_error () const
  {
    return error_operand_p (t);
  }

  bool
  is_null ()
  {
    return t == NULL_TREE;
  }

  static Tree
  error ()
  {
    return Tree (error_mark_node);
  }

  Tree
  get_type () const
  {
    return TREE_TYPE (t);
  }

private:
  tree t;
  location_t loc;
};
```

A GENERIC tree is actually a pointer, so comparison by identity is possible. For simplicity, let's teach `Tree` to do identity comparisons as well.

```
inline bool operator==(Tree t1, Tree t2) { return t1.get_tree () == t2.get_tree (); }
inline bool operator!=(Tree t1, Tree t2) { return !(t1 == t2); }
```

For convenience we will also wrap the creation of `Tree`s into a set of `build_tree` overloaded functions.

```
inline Tree
build_tree (tree_code tc, location_t loc, Tree type, Tree t1)
{
  return build1_loc (loc, tc, type.get_tree (), t1.get_tree ());
}

inline Tree
build_tree (tree_code tc, location_t loc, Tree type, Tree t1, Tree t2)
{
  return build2_loc (loc, tc, type.get_tree (), t1.get_tree (), t2.get_tree ());
}

inline Tree
build_tree (tree_code tc, location_t loc, Tree type, Tree t1, Tree t2, Tree t3)
{
  return build3_loc (loc, tc, type.get_tree (), t1.get_tree (), t2.get_tree (),
                     t3.get_tree ());
}

inline Tree
build_tree (tree_code tc, location_t loc, Tree type, Tree t1, Tree t2, Tree t3,
            Tree t4)
{
  return build4_loc (loc, tc, type.get_tree (), t1.get_tree (), t2.get_tree (),
                     t3.get_tree (), t4.get_tree ());
}

inline Tree
build_tree (tree_code tc, location_t loc, Tree type, Tree t1, Tree t2, Tree t3,
            Tree t4, Tree t5)
{
  return build5_loc (loc, tc, type.get_tree (), t1.get_tree (), t2.get_tree (),
                     t3.get_tree (), t4.get_tree (), t5.get_tree ());
}
```

Scope

In the definition of tiny we also talked about a stack of mappings from identifiers to values that we collectively called the *scope*. Note that the mappings in the scope, as defined in the tiny definition, are a dynamic entity so the exact value of the mapping will likely not be known at compile time.

That said, the mapping itself must exist. We will represent this mapping in a class called `SymbolMapping`. It will map identifiers (i.e. strings) to `SymbolPtrs` (later on we will see what is a `SymbolPtr`).

```
struct SymbolMapping
{
public:

  void insert (SymbolPtr s);
  SymbolPtr get (const std::string &str) const;

private:

  typedef std::map<std::string, SymbolPtr> Map;
  Map map;
};
```

As you can see it is a very thin wrapper to a map of strings to Symbol (for this reason sometimes a structure like this is called a *symbol table*).

`SymbolMapping::insert` adds a new `Symbol` into the map using its name as the key. It also checks that the name is not being added twice: this is not possible in tiny.

```
void
SymbolMapping::insert (SymbolPtr s)
{
  gcc_assert (s != NULL);
  std::pair<Map::iterator, bool> p
    = map.insert (std::make_pair (s->get_name (), s));

  gcc_assert (p.second);
}
```

`SymbolMapping::get` returns the mapped `Symbol` for the given string. Since it may happen that there is no such mapping this function may return a nul `Symbol`.

```
SymbolPtr
SymbolMapping::get (const std::string &str) const
{
  Map::const_iterator it = map.find (str);
  if (it != map.end ())
    {
      return it->second;
    }
  return SymbolPtr();
}
```

Class `Scope` is, as we said, a stack of `SymbolMapping`.

```
struct Scope
{
public:
  SymbolMapping &
  get_current_mapping ()
  {
    gcc_assert (!map_stack.empty ());
    return map_stack.back ();
  }

  void push_scope ();
  void pop_scope ();

  Scope ();

  SymbolPtr lookup (const std::string &str);

private:
  typedef std::vector<SymbolMapping> MapStack;
  MapStack map_stack;
};
```

We can manage the current symbol mapping using `Scope::push_scope()` and `Scope::pop_scope()`. The former will be used when we need a fresh mapping (as it will happen when handling `if`, `while` and `for` statements). `Scope::get_current_mapping` returns the current mapping (i.e. the one that was created in the last push_scope that has not been *popped* yet).

Function `Scope::lookup` is used to get the last mapping for a given string (or null if there is no such mapping).

```
SymbolPtr
Scope::lookup (const std::string &str)
{
  for (MapStack::reverse_iterator map = map_stack.rbegin ();
       map != map_stack.rend (); map++)
    {
      if (SymbolPtr sym = map->get (str))
        {
          return sym;
        }
    }
  return SymbolPtr();
}
```

We have to traverse the stack from the top (end of the `MapStack`) to the bottom (beginning of the `MapStack`), so we use a `reverse_iterator` for this.

`Scope::push_scope` and `Scope::pop_scope` have obvious implementations.

```
void
Scope::push_scope ()
{
  map_stack.push_back (SymbolMapping());
}

void
Scope::pop_scope ()
```

```
{
  gcc_assert (!map_stack.empty());
  map_stack.pop_back ();
}
```

Symbol

We will use the class `Symbol` to represent a named entity of a tiny program. So far the only named entities we have in tiny are variables. Other languages may have types, constants and functions in their set of entities with names. `Symbol` class would be used as well for such entities.

There will be a single Symbol object for each named instance, so this class is mostly used by reference. Similar to what we did with tokens in part 3, we will define `SymbolPtr` and `const_SymbolPtr` as smart pointers. We have already used `SymbolPtr` in classes `Scope` and `SymbolMapping` above.

```
typedef std::tr1::shared_ptr<Symbol> SymbolPtr;
typedef std::tr1::shared_ptr<const Symbol> const_SymbolPtr;
```

Tiny is so simple that we only need to keep the name of a symbol (something slightly redundant since GENERIC will have the name somewhere as well) and the associated `VAR_DECL` tree. In a language with other kind of symbols we would probably want to keep the kind of the symbol and we would probably store other kind of `_DECL` trees.

```
struct Symbol
{
public:
  Symbol (const std::string &name_) : name (name_), decl (error_mark_node)
  {
    gcc_assert (name.size () > 0);
  }

  std::string
  get_name () const
  {
    return name;
  }

  void
  set_tree_decl (Tree decl_)
  {
    gcc_assert (decl_.get_tree_code() == VAR_DECL);
    decl = decl_;
  }

  Tree
  get_tree_decl () const
  {
    return decl;
  }
private:
  std::string name;
  Tree decl;
};
```

Current layout

Our `gcc-src/gcc/tiny` directory now looks like this.

```
gcc-src/gcc/tiny
├── config-lang.in
├── lang-specs.h
├── Make-lang.in
├── tiny1.cc
├── tiny-buffered-queue.h
├── tiny-lexer.cc
├── tiny-lexer.h
├── tiny-parser.cc
├── tiny-parser.h
├── tiny-scope.cc
├── tiny-scope.h
├── tinyspec.cc
├── tiny-symbol.cc
├── tiny-symbol.h
├── tiny-symbol-mapping.cc
├── tiny-symbol-mapping.h
├── tiny-token.cc
├── tiny-token.h
└── tiny-tree.h
```

Today we will stop here. We have seen the objects that will be required for the semantic analysis itself. In the next part we will change the parser to generate GENERIC trees that will represent the semantics of our program.

That's all for today.

« A tiny GCC front end – Part 4 A tiny GCC front end – Part 6 »