A tiny GCC front end – Part 1

Jan 5, 2016 • Roger Ferrer Ibáñez • compilers, GCC • compilers, frontend, gcc, tiny

In this series we will see the process of adding a new front end for a very simple language in GCC. If you, like me, marvel at the magic of compilers then these posts may be for you.

Tiny Imperative Language

We are going to implement a front end for a really simple language called *Tiny Imperative Language* (TIL) or just *tiny*. This language has not been standardized or defined elsewhere but we will not start from scratch. Our tiny implementation will be based on the description available in the wiki of Software Transformation Systems.

Programming languages have three facets that we have to consider:

- Syntax, that deals with the *form*
- Semantics, that deals with the *meaning*
- *Pragmatics*, that deals with the *implementation*

These three facets are not independent and affect each other. In this series we will deal mostly about the pragmatics but we still need a minimal definition of the syntax and semantics of tiny before we start implementing anything. This is important as the syntax and the semantic obviously have an impact in the implementation. In this post we will define to some detail (although incompletely) the syntax and the semantics of our tiny language. The rest of the series will be all about the pragmatics.

Syntax

A tiny *program* is composed by a, possibly empty, sequence of *statements*. This means that an empty program is a valid tiny program. In this syntax description ⟨name⟩ means a part of the language and * means the preceding element zero or more times.

⟨program⟩ → ⟨statement⟩ *

In tiny there are 7 kinds of statements. In this syntax description a vertical bar | is used to separate alternatives

⟨statement⟩ → ⟨declaration⟩ | ⟨assignment⟩ | ⟨if⟩ | ⟨while⟩ | ⟨for⟩ | ⟨read⟩ | ⟨write⟩

A declaration is used to introduce the name of a variable and its type. In this syntax description a bold monospaced font face like **`this`** is used to denote *keywords* or verbatim lexical elements.

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

Our language will support, for the moment, only two types for variables.

⟨type⟩ → **`int`** | **`float`**

An identifier is a letter (or underscore) followed zero or more letters, digits and underscores. In this syntax description { and } act as parentheses so * can be applied to the resulting group.

⟨identifier⟩ → { ⟨letter⟩ | ⟨underscore⟩ }{ ⟨letter⟩ | ⟨digit⟩ | ⟨underscore⟩ }*
⟨letter⟩ → **`a`** | **`b`** | **`c`** | ... | **`x`** | **`y`** | **`z`** | **`A`** | **`B`** | **`C`** | ... | **`X`** | **`Y`** | **`Z`**
⟨digit⟩ → **`0`** | **`1`** | **`2`** | **`3`** | **`4`** | **`5`** | **`6`** | **`7`** | **`8`** | **`9`**
⟨underscore⟩ → **`_`**

Examples of identifiers are `foo`, `foo123`, `foo_123`, `hello_world`, `_foo`, `foo12a`. If an identifier would match a keyword (like `var`) then it is always a keyword, never an identifier.

Except where necessary for the proper recognition of lexical elements of the language, whitespace is not relevant. This means that the three lines below are syntactically equivalent:

```
var a : int;
var       a    :  int   ;
var a:int;
```

The following two are not (in fact they are syntactically invalid).

```
vara : int;
var a : i nt;
```

This is the form of an assignment statement.

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

This is the form of an if statement.

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

This is the form of a while statement.

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

This is the form of a for statement.

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

This is the form of a read statement.

⟨read⟩ → **read** ⟨identifier⟩ **;**

This is the form of a write statement.

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

An expression is either a primary, a prefix unary operator and its operand or a binary infix operator with a left hand side operand and a right hand side operand.

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

A primary can be a parenthesized expression, an identifier, an integer literal, a float literal or a string literal. In this syntax description + means the preceding element one or more times.

⟨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⟩ ***"**

Unary operators have the following forms.

⟨unary-op⟩ → **+** | **-** | **not**

Binary operators have the following forms.

⟨binary-op⟩ → **+** | **-** | ***** | **/** | **%** | **==** | **!=** | **<** | **<=** | **>** | **>=** | **and** | **or**

All binary operators associate from left to right so `x ⊕ y ⊕ z` is equivalent to `(x ⊕ y) ⊕ z`. Likewise for binary operators with the same priority.

The following table summarizes priorities between operators. Operators in the same row have the same priority.

Operators	**Priority**
(unary)`+` (unary)`-`	Highest priority
`* / %`	
(binary)`+` (binary)`-`	
`== != < <= > >=`	
`not`, `and`, `or`	Lowest priority

This means that `x + y * z` is equivalent to `x + (y * z)` and `x > y and z < w` is equivalent to `(x > y) and (z < w)`. Parentheses can be used if needed to change the priority like in `(x + y) * z`.

A symbol #, except when inside a string literal, introduces a comment. A comment spans until a newline character. It is not part of the program, it is just a lexical element that is discarded.

A tiny example program follows

```
1 var i : int;
2 for i := 0 to 10 do     # this is a comment
3    write i;
4 end
```

Semantics

Since a tiny program is a sequence of statements, executing a tiny program is equivalent to execute, in order, each statement of the sequence.

A tiny program, like any imperative programming language, can be understood as a program with some state. This state is essentially a mapping of identifiers to values. In tiny, there is a stack of those mappings, that we collectivelly will call the *scope*. A tiny program starts with a scope consisting of just a single empty mapping.

A declaration introduces a new entry in the top mapping of the current scope. This entry maps an identifier (called the *variable name*) to an undefined value of the ⟨type⟩ of the declaration. This value is called the *value of the variable*. There can be up to one entry that maps an identifier to a value, so declaring twice the same identifier in the same scope is an error.

> This is obviously a *design decision*: another language might choose to define a sensible initial mapping. For example, to a zero value of the type (in our case it would be `0` for `int` and `0.0` for `float`). Since the initial mapping is to an undefined value, this means that the variable does not have to be initialized with any particular value.

In tiny the set of values of the `int` type are those of the 32-bit integers in two's complement (i.e. -2^{31} to $2^{31} - 1$). The set of values of the `float` type is the same as the values of the of the Binary32 IEEE 754 representation, excluding (for simplicity) NaN and Infinity. The value of a variable may be undefined or an element of the set of values of the type of its declaration.

The set of values of the boolean type is just the elements “true” and “false”. Values of string type are sequences of characters of 1 byte each.

An assignment, defines a new state where all the existing mappings are left untouched except for the entry of the identifier which is updated to the value denoted by the expression. The old state is discarded and the new state becomes the current state. If there is not an entry for the identifier in any of the mappings of the scope, this is an error. The expression must denote an `int` or `float` type, otherwise this is an error. The identifier must have been declared with the same type as the type of the expression, otherwise this is an error.

> Note that we do not allow assigning a float value to an `int` variable nor an `int` value to a `float` variable. I may lift this restriction in the future.

For instance, the following tiny program is annotated with the changes in its state. Here ⊥ means an undefined value.

```
# [ ]
var x : int;
# [ x → ⊥ ]
x := 42;
# [ x → 42 ]
x := x + 1;
# [ x → 43 ]
var y : float;
# [ x → 43, y → ⊥ ]
y = 1.0;
# [ x → 43, y → 1.0 ]
y = y + x;
# [ x → 43, y → 44.0 ]
```

The bodies of if, while and for statements (i.e. their 〈statement〉 * parts) introduce a new mapping on top of the current scope. The span of this new mapping is restricted to the body. Since the mapping is new, it is valid to declare a variable whose identifier has already been used before. This is commonly called *hiding*.

```
1 # [ ]
2 var x : int;
3 # [ x → ⊥ ]
4 var y : int;
5 # [ x → ⊥, y → ⊥ ]
6 x := 3;
7 # [ x → 3, y → ⊥ ]
8 if (x > 1) then
9     # [ x → 3, y → ⊥ ], [ ]
```

```
10    var x : int;
11    # [ x → 3, y → ⊥ ], [ x → ⊥ ]
12    x := 4;
13    # [ x → 3, y → ⊥ ], [ x → 4 ]
14    y := 5
15    # [ x → 3, y → 5 ], [ x → 4 ]
16    var z : int
17    # [ x → 3, y → 5 ], [ x → 4, z → ⊥ ]
18    z := 8
19    # [ x → 3, y → 5 ], [ x → 4, z → 8 ]
20 end
21 # [ x → 3, y → 5 ]
22 z := 8 # ← ERROR HERE, z is not in the scope!!
```

The meaning of an identifier used in an assignment expression always refers to the entry in the latest mapping introduced. This is why in the example above, inside the if statement, `x` does not refer to the outermost one (because the declaration in line 9 hides it) but `y` does.

> This kind of scoping mechanism is called static or lexical scoping.

An if statement can have two forms, but the first form is equivalent to **`if`** ⟨expression⟩ **`then`** ⟨statement⟩ * **`else end`**, so we only have to define the semantics of the second form. The execution of an if statement starts by evaluating its ⟨expression⟩ part, called the *condition*. The condition expression must have a boolean type, otherwise this is an error. If the value of the condition is true then the first ⟨statement⟩ * is evaluated. If the value of the condition is false, then the second ⟨statement⟩ * is evaluated.

The execution of a while statement starts by evaluating its ⟨expression⟩ part, called the *condition*. The condition expression must have a boolean type, otherwise this is an error. If the value of the condition is false, nothing is executed. If the value of the condition is true, then the ⟨statement⟩ * is executed and then the while statement is executed again.

A for statement of the form

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

is semantically equivalent to

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

Execution of a read statement causes a tiny program to read from the standard input a textual representation of a value of the type of the identifier. Then, the identifier is updated as if by an

assignment statement, with the represented value. If the textual representation read is not valid for the type of the identifier, then this is an error.

Execution of a write statement causes a tiny program to write onto the standard output a textual representation of the value of the expression.

For simplicity, the textual representation used by read and write is the same as the syntax of the literals of the corresponding types.

Semantics of expressions

We say that an expression has a specific type when the evaluation of the expression yields a value of that type. Evaluating an expression is computing such value.

An integer literal denotes a value of `int` type, i.e. a subset of the integers. Given an integer literal of the form $d_n d_{n-1} ... d_0$, the denoted integer value is $d_n \times 10^n + d_{n-1} \times 10^{n-1} + ... + d_0$. In other words, an integer literal denotes the integer value of that number in base 10.

A float literal denotes a value of `float` type. A float of the form $d_n d_{n-1} ... d_0 . d_{-1} d_{-2} ... d_{-m}$ denotes the closest IEEE 754 Binary32 float value to the value $d_n \times 10^n + d_{n-1} \times 10^{n-1} + ... + d_0 + d_{-1} 10^{-1} + d_{-2} 10^{-2} + ... + d_{-m} 10^{-m}$

A string literal denotes a value of string type, the value of which is the sequence of bytes denoted by the characters in the input, not including the delimiting double quotes.

An expression of the form **(**`e`**)** denotes the same value and type of the expression `e`.

An identifier in an expression denotes the entry in the latest mapping introduced in the scope (likewise the identifier in the assignment statement, see above). If there is not such mapping or maps to the undefined value, then this is an error.

An expression of the form **+**`e` or **-**`e` denotes a value of the same type as the expression `e`. Expression `e` must have `int` or `float` type. The value of code>**+**e</code> is the same as `e`. Value of **-**`e` is the negated value of `e`.

The operands of (binary) operators **+**, **-** *****, **/**, **<**, **<=**, **>**, **>=**, **==** and **!=** must have `int` or `float` type, otherwise this is an error. If only one of the operands is `float`, the `int` value of the other one is coerced to the corresponding value of `float`. The operands of `%` must have `int` type. The operands of `not`, `and`, `or` must have boolean type.

> We've seen above that assignment seems overly restrictive by not allowing assignments between `int` and `float`. Conversely, binary operators are more relaxed by allowing coercions of `int` operands to `float` operands. I know at this point it is a bit arbitrary, but it illustrates some points in programming language design that we usually take for granted but may not be obvious.

Operators **+**, **-** and *****, compute, respectively, the arithmetic addition, subtraction and multiplication of its (possibly coerced) operands (for the subtraction the second operand is subtracted from the first operand, as usually). The expression denotes a `float` type if any operand is `float`, `int` otherwise.

Operator **/** when both operands are `int` computes the integer division of the first operand by the second operand rounded towards zero, the resulting value has type `int`. When any of the operands is a `float`, an arithmetic division between the (possibly coerced) operands is computed. The resulting value has type `float`.

Operator **%** computes the remainder of the integer division of the first operand (where the remainder has the same sign as the first operand). The resulting value has type `int`.

> This is deliberately the same modulus that the C language computes.

Operators **<**, **<=**, **>**, **>=**, **==** and **!=** compare the (possibly coerced) first operand with the (possibly coerced) second operand. The comparison checks if the first operand is, respectively, less than, less or equal than, greater than, greater or equal than, different (not equal) or equal than the second operand. The resulting value has boolean type.

Operators **not**, **and**, **or** perform the operations $\neg$, $\wedge$, $\vee$ of the boolean algebra. The resulting value has boolean type.

> Probably you have already figured it now, but it is possible to create expressions with types that cannot be used for variables. There are no variables of string or boolean type. For string types we can create a value using a string literal but we cannot operate it in any way. Only the write statement allows it. For boolean values, we can operate them using **and**, **or** and **not** but there are no boolean literals or boolean variables (yet).

Wrap-up

Ok, that was long but we will refer to this document when implementing the language. Note that the languages, as it is, is underspecified. For instance, we have not specified what happens when an addition overflows. We will revisit some of these questions in coming posts.

That's all for today.

« Toying with GCC JIT – Part 3 A tiny GCC front end – Part 2 »