Reprint from
The *Mathematica* Conference

June, 1992   Boston, MA

845.9.10.1992

Compiling *Mathematica* Procedures

Matt Cook

1 Some Simple Uses of Compile

The easiest way to introduce ourselves to `Compile` is to use it a few times.

This creates a function which squares its argument.

```
In[1]:= Compile[x, x^2]

                                2
Out[1]= CompiledFunction[{x}, x , -CompiledCode-]
```

We can use this function just like any other. Notice that a numerical answer is returned.

```
In[2]:= %[3]

Out[2]= 9.
```

This creates a function of three variables.

```
In[3]:= Compile[{x,y,z}, Sqrt[x^2 + y^2 + z^2]]

                                               2    2    2
Out[3]= CompiledFunction[{x, y, z}, Sqrt[x  + y  + z ],
  -CompiledCode-]
```

`Compile` assumes that its arguments should be compiled as machine-precision real numbers, unless it is directed to do otherwise.

This specifies that `x` is a boolean variable, and `y` and `z` are integers.

```
In[4]:= Compile[{{x, True|False}, {y, _Integer}, {z, _Integer}},
                If[x, y+z, y-z]]

Out[4]= CompiledFunction[{x, y, z}, If[x, y + z, y - z],
  -CompiledCode-]
```

Note that this time an exact integer answer is returned.

```
In[5]:= %[False, 3, 7]

Out[5]= -4
```

Compilable types are _Integer, _Real, _Complex, and True | False. Each of these types is taken to be of machine precision. Accuracy is not kept track of. Computation goes on just like it would on a calculator or in a C or Fortran program. However, the full *Mathematica* system is available for those occasions when the computation is too much for machine precision to handle.

We create a simple compiled function to multiply two integers.

```
In[6]:= Compile[{{x, _Integer}, {y, _Integer}}, x y]

Out[6]= CompiledFunction[{x, y}, x y, -CompiledCode-]
```

When the compiled function tries to execute this multiplication with machine integers, an error occurs. So the function is re-evaluated as an ordinary uncompiled *Mathematica* function, and the correct answer is obtained. `CompiledFunction`s are designed to always return the correct result, even if executing the compiled code leads to an error.

```
In[7]:= %[2^25, 2^25]

CompiledFunction::cfn:
   Numerical error encountered at instruction 4
     ; proceeding with uncompiled evaluation.

Out[7]= 1125899906842624
```

- In some graphical and numerical functions, such as `Plot` and `NIntegrate`, *Mathematica* will by default compile the function that is being repeatedly evaluated.
- This default can be overridden for these functions by changing the default value for their option `Compiled`, which can be set to either `True` or `False`.
- For these graphical and numerical functions, if the compiled function they generate fails for any reason, *Mathematica* will not generate any error messages for the `CompiledFunction`.
- If the compiled function fails twice in a row, *Mathematica* will stop trying to use it.

Internally generated `CompiledFunctions`.

Sometimes part of an expression can't be compiled, because its value could change between compilation and evaluation. For example, in the following function, the variable `anglecosine` could change after compilation. Therefore, the compiled code asks *Mathematica* at runtime for the value of `anglecosine` to use.

This creates a function to find the third side of a triangle given the first two sides. It uses a variable `anglecosine` which is not an argument to the function, and therefore must be looked up at runtime.

```
In[8]:= Compile[{x,y}, Sqrt[x^2 + y^2 - 2 x y anglecosine]]

Out[8]= CompiledFunction[{x, y},

           2    2
  Sqrt[x  + y  - 2 x y anglecosine], -CompiledCode-]
```

The compiled code asks *Mathematica* for the value of `anglecosine`, and uses it to compute the answer.

```
In[9]:= anglecosine = 0; %[3,4]

Out[9]= 5.
```

This time when the compiled code asks *Mathematica* for the value of `anglecosine`, it gets a symbolic answer. Since compiled code can only deal with machine-precision numerical quantities, it complains that `Cos[theta]` is not usable, and then it lets *Mathematica* evaluate the function in the ordinary uncompiled manner.

```
In[10]:= anglecosine = Cos[theta]; %%[3,4]

CompiledFunction::ccr:
   Expression Cos[theta] should be a machine-size real number.

CompiledFunction::cfex:
   External evaluation error at instruction 7
     ; proceeding with uncompiled evaluation.

Out[10]= Sqrt[25 - 24 Cos[theta]]
```

Similarly, if a function is used as part of the definition of a compiled function, its definition could change after compilation. So the compiled code has to evaluate that function *externally*. By this we mean that the function is evaluated in the the normal *Mathematica* manner, externally from the compiled code. For example, the following compiled function depends on the function `foo`.

The compiled code contains instructions to square its argument, evaluate `foo` externally, and then invert the answer. The definition of `foo` is not used during compilation.

```
In[11]:= Compile[x, 1/foo[x^2]]

                                   1
Out[11]= CompiledFunction[{x}, -------, -CompiledCode-]
                                    2
                               foo[x ]
```

The compiled code will always use the most current definition of `foo`.

```
In[12]:= foo[x_]:= 1/x; %[3]

Out[12]= 9.
```

2 Advanced Considerations When Using Compile

Although the compiled code does not care about the definitions of external functions, it does care about the return value. It will assume that the return value is a real number unless we specify otherwise. There is an optional third argument to `Compile` which we can use to tell it what to assume about the return values of external functions.

The third argument tells `Compile` that `myTest` returns a boolean value, and that `myRounding` returns an integer.

```
In[1]:= Compile[x, If[myTest[x], Round[x], myRounding[x]],
        {{myTest[_], True|False}, {myRounding[_], _Integer}}]

Out[1]= CompiledFunction[{x},
  If[myTest[x], Round[x], myRounding[x]], -CompiledCode-]
```

Note that an exact integer is returned. If we had not specified the return type of `myRounding` (or had specified it to be _Real), we would have gotten an inexact answer.

```
In[2]:= myTest[x_]:= x<0; myRounding[n_]:= Floor[n]; %[3.7]

Out[2]= 3
```

The third argument to `Compile` is a list of pairs {*pat*, *type*}. If something is going to have to be externally evaluated, and it matches *pat*, `Compile` will assume that the result will be of type *type*.

Furthermore, although `Compile` will in general try to compile the arguments to functions that are going to be evaluated externally, we can instruct it not to by including a third part in the type specification. If the subexpression matches the third part, `Compile` will not compile the arguments to the function being externally evaluated.

`Compile` creates code to compute the arguments to `Print`.

```
In[3]:= f = Compile[x, Print[label,Sin[Cos[x]]]]

Out[3]= CompiledFunction[{x}, Print[label, Sin[Cos[x]]],
  -CompiledCode-]
```

The compiled code finds values for the arguments to `Print`, and passes them when the time comes for the external evaluation of `Print`.

```
In[4]:= label=5; f[3]

5.-0.836022
```

Unfortunately, the value of `label` in this case is not numerical. An error occurs, and uncompiled evaluation is used to compute `f`.

```
In[5]:= label="value: "; f[3]

CompiledFunction::ccr:
   Expression value:  should be a machine-size real number.

CompiledFunction::cfex:
   External evaluation error at instruction 3
    ; proceeding with uncompiled evaluation.
value: Sin[Cos[3]]
```

This time we instruct `Compile` not to create instructions for evaluating the arguments to `Print` when it has two arguments.

```
In[6]:= f = Compile[x, Print[label,Sin[Cos[x]]],
        {{Print[___], Null, Print[_,_]}}]

Out[6]= CompiledFunction[{x}, Print[label, Sin[Cos[x]]],
  -CompiledCode-]
```

This time the value of `label` does not cause an error, and `Sin[Cos[3]]` is evaluated in an uncompiled manner, resulting in an exact answer.

```
In[7]:= label="value: "; f[3]

value: Sin[Cos[3]]
```

Now we have introduced ourselves to all the basic features of `Compile`. Of course, in the examples shown so far, the functions have been quite trivial. In practice, we need to know not only how `Compile` works, but when to use it and when not to use it.

If we have the function `f[a_,b_]:=BesselJ[a,a+b]`, there is no point in creating a compiled version of this function. Most of the time being spent to compute this function is being spent computing `BesselJ[a,a+b]`, not in iterative looping structures. *Mathematica* is already highly optimized for computing numerical values for complicated functions, so there is nothing to gain by trying to compile a function such as this.

However, `Compile` is very useful in speeding up simple time consuming iterative computations, such as the following.

Here is a function which might be used when making a picture of the Mandelbrot set.

```
In[8]:= iterationCount[c_, limit_] :=
            Module[{z=0, n=0},
        While[Abs[z]<2 && n<limit,
        z = z^2 + c;
        n++];
         n]
In[9]:= Table[iterationCount[x, 1000],
        {x, .3655 +.1 I, .366 + .1 I,
        .00001}] // Timing
Out[9]= {85.7167 Second, {290, 417, 541, 1000, 713, 607, 520,
   368, 327, 522, 271, 205, 567, 316, 160, 197, 154, 228, 142,
   147, 168, 200, 1000, 1000, 794, 686, 349, 231, 324, 182,
   480, 194, 170, 156, 418, 307, 480, 625, 680, 306, 255, 938,
   179, 240, 130, 127, 123, 119, 116, 115, 115}}
```

Here is a the same function, this time written as a compiled function.

```
In[10]:= compiledIterationCount =
        Compile[{{c, _Complex}, {limit, _Integer}},
             Module[{z=I, n=0}, z=0;
         While[Abs[z]<2 && n<limit,
         z = z^2 + c;
         n++];
          n]]
Out[10]= CompiledFunction[{c, limit},
  Module[{z = I, n = 0}, z = 0;
                                          2
    While[Abs[z] < 2 && n < limit, z = z  + c; n++]; n],
  -CompiledCode-]
In[11]:= Table[compiledIterationCount[x, 1000],
         {x, .3655 +.1 I, .366 + .1 I,
         .00001}] // Timing
Out[11]= {2.08333 Second,
  {290, 417, 541, 1000, 713, 607, 520, 368, 327, 522, 271, 205,
   567, 316, 160, 197, 154, 228, 142, 147, 168, 200, 1000,
   1000, 794, 686, 349, 231, 324, 182, 480, 194, 170, 156, 418,
   307, 480, 625, 680, 306, 255, 938, 179, 240, 130, 127, 123,
   119, 116, 115, 115}}
```

The main obstacle to achieving a speedup as good as this is when the compiled code has to externally evaluate a function. To do this, it has to create the expression to be evaluated, create all the local variables with their proper names and scopes, perform the external evaluation, and check to see if the external evaluation changed any of the local variables, updating the appropriate registers.

In light of this, our main concern when trying to create a compiled function is to have there be as few instructions for external evaluation as possible. In fact, zero is a good number to aim for, especially within any time-consuming loop, for achieving a maximal speedup.

It is important to know about some general types of functions that are not going to compile well, so we can avoid frustrating ourselves with trying to make compiled versions of them. Any function that deals primarily with matrices, arrays, lists, or strings, is not going to compile well at all, since these types do not fit nicely into registers, and there are currently no dynamic memory manipulation capabilities for compiled code.

A good rule of thumb is that if you have a repetitive numerical algorithm that does not use lists, then it is a good candidate for benefiting from compilation.

3 What's Inside a CompiledFunction

Let's look inside a `CompiledFunction`.

Curiosity killed the cat.

```
In[1]:= Compile[{{x, True|False}, {y, _Integer}, {z, _Integer}},
            If[x, y+z, y-z]] // InputForm

Out[1]//InputForm=
  CompiledFunction[{True | False, _Integer, _Integer},
   {1, 7, 0, 0}, {{1, 17}, {2, 1, 0}, {3, 2, 0}, {3, 3, 1},
    {69, 0, 4}, {32, 0, 1, 2}, {16, 2, 6}, {70, 5},
    {35, 1, 3}, {38, 3, 4}, {32, 0, 4, 5}, {16, 5, 6}, {7, 6}},
   Function[{x, y, z}, If[x, y + z, y - z]]]
```

A `CompiledFunction` has four pieces. The first piece is a list of the types of the arguments. The second piece is a list of how many "registers" of each type (boolean, integer, real, and complex) will be used. The third piece is the list of the actual compiled instructions comprising the compiled code. The fourth and final piece is the compiled function written as an ordinary *Mathematica* function, available for use in case a run time error occurs within the compiled code.

`CompiledFunction[{`arg_1`, `arg_2`, ... }, {`n_b`, `n_i`, `n_r`, `n_c`}, `*instr*`, `*func*`]`	
	compiled code taking arguments of type arg_i and and executing the instruction stream *instr* using n_k registers of type *k*

The structure of a `CompiledFunction` object.

To be able to interpret the third piece, we need to know what the compiled instructions mean. Each instruction is represented as a list of one or more numbers. The first number is an opcode, that is, an integer identifying the operation to be performed. The other numbers are either register numbers or literal constants.

Here is a list of the opcodes for the current version of the compiler.

`{0}`	do nothing (any args are ignored)

The null operation.

`{1, 17}`	specify instruction set 17 (this instruction set)

Specifying the instruction set.

`{2, `*n*`, `*boolreg*`}`	load argument number *n* into boolean register *boolreg*
`{3, `*n*`, `*intreg*`}`	load argument number *n* into integer register *intreg*
`{4, `*n*`, `*realreg*`}`	load argument number *n* into real register *realreg*
`{5, `*n*`, `*cxreg*`}`	load argument number *n* into complex register *cxreg*

Loading function arguments into registers

`{6,` *boolreg*`}`	return the contents of boolean register *boolreg*
`{7,` *intreg*`}`	return the contents of integer register *intreg*
`{8,` *realreg*`}`	return the contents of real register *realreg*
`{9,` *cxreg*`}`	return the contents of complex register *cxreg*
`{10}`	return *Null*

Returning results

`{11,` *bool*`,` *boolreg*`}`	load logical constant *bool* into boolean register *boolreg*
`{12,` *int*`,` *intreg*`}`	load integer constant *int* into integer register *intreg*
`{13,` *real*`,` *realreg*`}`	load real constant *real* into real register *realreg*
`{14,` *cx*`,` *cxreg*`}`	load complex constant *cx* into complex register *cxreg*

Loading registers with constants

`{15,` $boolreg_1$`,` $boolreg_2$`}`	copy $boolreg_1$ into $boolreg_2$
`{16,` $intreg_1$`,` $intreg_2$`}`	copy $intreg_1$ into $intreg_2$
`{17,` $realreg_1$`,` $realreg_2$`}`	copy $realreg_1$ into $realreg_2$
`{18,` $cxreg_1$`,` $cxreg_2$`}`	copy $cxreg_1$ into $cxreg_2$

Copying registers

`{19,` *boolreg*`,` *intreg*`}`	convert *boolreg* into an integer (0 or 1) and place in *intreg*
`{20,` *intreg*`,` *realreg*`}`	convert *intreg* into a real number and place in *realreg*
`{21,` $realreg_1$`,` $realreg_2$`,` *cxreg*`}`	put the complex number ($realreg_1$ `+ I` $realreg_2$) into *cxreg*

Converting from one type to another

`{22,` *expr*`,` *boolreg*`}`	evaluate *expr*`[`arg_1`, … ]` to get a boolean variable
`{23,` *expr*`,` *intreg*`}`	evaluate *expr*`[`arg_1`, … ]` to get an integer
`{24,` *expr*`,` *realreg*`}`	evaluate *expr*`[`arg_1`, … ]` to get a real number
`{25,` *expr*`,` *cxreg*`}`	evaluate *expr*`[`arg_1`, … ]` to get a complex number
`{26,` *expr*`}`	evaluate *expr*`[`arg_1`, … ]` and ignore the result

Evaluating non-compilable expressions

`{27,` *expr*`,` *type*$_1$`,` *reg*$_1$`, … ,` *boolreg*`}`	evaluate *expr*`[`*reg*$_1$`, … ]` to get a boolean variable
`{28,` *expr*`,` *type*$_1$`,` *reg*$_1$`, … ,` *intreg*`}`	evaluate *expr*`[`*reg*$_1$`, … ]` to get an integer
`{29,` *expr*`,` *type*$_1$`,` *reg*$_1$`, … ,` *realreg*`}`	evaluate *expr*`[`*reg*$_1$`, … ]` to get a real number
`{30,` *expr*`,` *type*$_1$`,` *reg*$_1$`, … ,` *cxreg*`}`	evaluate *expr*`[`*reg*$_1$`, … ]` to get a complex number
`{31,` *expr*`,` *type*$_1$`,` *reg*$_1$`, … }`	evaluate *expr*`[`*reg*$_1$`, … ]` and ignore the result

Evaluating non-compilable functions after computing their arguments

`{32,` *intreg*$_1$`,` *intreg*$_2$`, … ,` *intreg*`}`	add integers
`{33,` *realreg*$_1$`,` *realreg*$_2$`, … ,` *realreg*`}`	add real numbers
`{34,` *cxreg*$_1$`,` *cxreg*$_2$`, … ,` *cxreg*`}`	add complex numbers

Addition

`{35,` *intreg*$_1$`,` *intreg*$_2$`, … ,` *intreg*`}`	multiply integers
`{36,` *realreg*$_1$`,` *realreg*$_2$`, … ,` *realreg*`}`	multiply real numbers
`{37,` *cxreg*$_1$`,` *cxreg*$_2$`, … ,` *cxreg*`}`	multiply complex numbers

Multiplication

`{38,` *intreg*$_1$`,` *intreg*`}`	negate an integer
`{39,` *realreg*$_1$`,` *realreg*`}`	negate a real number
`{40,` *cxreg*$_1$`,` *cxreg*`}`	negate a complex number

Negation

`{41,` *realreg*$_1$`,` *realreg*`}`	reciprocal of a real number
`{42,` *cxreg*$_1$`,` *cxreg*`}`	reciprocal of a complex number

Reciprocals

{43, $realreg_1$, $intreg_1$, $realreg$}	real number to an integer power
{44, $realreg_1$, $realreg_2$, $realreg$}	real number to a real power
{45, $cxreg_1$, $cxreg_2$, $cxreg$}	complex number to a complex power
{46, $realreg_1$, $realreg$}	square root of a real number

Powers

{47, $realreg_1$, $realreg$}	Exp[x]
{48, $realreg_1$, $realreg$}	Log[x]
{49, $realreg_1$, $realreg$}	Sin[x]
{50, $realreg_1$, $realreg$}	Cos[x]
{51, $realreg_1$, $realreg$}	Tan[x]
{52, $realreg_1$, $realreg$}	ArcSin[x]
{53, $realreg_1$, $realreg$}	ArcCos[x]
{54, $realreg_1$, $realreg$}	ArcTan[x]

Transcendental functions of real variables

{55, $cxreg_1$, $cxreg$}	Exp[x]
{56, $cxreg_1$, $cxreg$}	Log[x]
{57, $cxreg_1$, $cxreg$}	Sin[x]
{58, $cxreg_1$, $cxreg$}	Cos[x]
{59, $cxreg_1$, $cxreg$}	Tan[x]
{60, $cxreg_1$, $cxreg$}	ArcSin[x]
{61, $cxreg_1$, $cxreg$}	ArcCos[x]
{62, $cxreg_1$, $cxreg$}	ArcTan[x]

Transcendental functions of complex variables

{63, $realreg_1$, $intreg$}	Floor[x]
{64, $cxreg_1$, $realreg$}	Re[z]
{65, $cxreg_1$, $realreg$}	Im[z]
{66, $cxreg_1$, $cxreg$}	Conjugate[z]
{67, $cxreg_1$, $realreg$}	Abs[z]
{68, $intreg_1$, $intreg_2$, $intreg$}	Mod[i, j]

Miscellaneous functions

{69, *logreg*, *offset*}	jump if false
{70, *offset*}	jump always
{71, *intreg*, int_1, $offset_1$, … }	switch

Branching

{72, $intreg_1$, $intreg_2$, *logreg*}	*i* == *j*
{73, $realreg_1$, $realreg_2$, *logreg*}	*x* == *y*
{74, $intreg_1$, $intreg_2$, *logreg*}	*i* < *j*
{75, $realreg_1$, $realreg_2$, *logreg*}	*x* < *y*
{76, $intreg_1$, $intreg_2$, *logreg*}	*i* <= *j*
{77, $realreg_1$, $realreg_2$, *logreg*}	*x* <= *y*

Comparisons

{78, $logreg_1$, $logreg_2$, *logreg*}	*p* && *q*
{79, $logreg_1$, $logreg_2$, *logreg*}	*p* || *q*
{80, $logreg_1$, *logreg*}	!*p*

boolean operations

{81}	force an error to occur

Forcing an error

```
opCodes =
{"NOP","VERSION","LDARG_BOOL","LDARG_INT","LDARG_REAL","LDARG_CX","RET_BOOL",
"RET_INT","RET_REAL","RET_CX","RET_NULL","LDCONST_BOOL","LDCONST_INT",
"LDCONST_REAL","LDCONST_CX","COPY_BOOL","COPY_INT","COPY_REAL","COPY_CX",
"CVT_BOOL_TO_INT","CVT_INT_TO_REAL","CVT_REAL_TO_CX","EVALARG_BOOL",
"EVALARG_INT","EVALARG_REAL","EVALARG_CX","EVALARG","EVALREG_BOOL",
"EVALREG_INT","EVALREG_REAL","EVALREG_CX","EVALREG","PLUS_INT","PLUS_REAL",
"PLUS_CX","TIMES_INT","TIMES_REAL","TIMES_CX","MINUS_INT","MINUS_REAL",
"MINUS_CX","RECIP_REAL","RECIP_CX","POWER_REAL_INT","POWER_RRR",
"POWER_CX_CX","SQRT","EXP","LOG","SIN","COS","TAN","ARCSIN","ARCCOS","ARCTAN",
"EXP_CX","LOG_CX","SIN_CX","COS_CX","TAN_CX","ARCSIN_CX","ARS_CX","ARCTAN_CX",
"FLOOR","RE","IM","CONJ","ABS_CX","MOD","BRANCH","JUMP","SWITCH","EQ_INT",
"EQ_REAL","LT_INT","LT_REAL","LE_INT","LE_REAL","AND","OR","NOT","ERR"}

dcc[fun_CompiledFunction]:=
Print[TableForm[dccNumberLines[dccOpsToStrings[fun[[3]]]],TableSpacing->{0,4}]]

dccOpsToStrings[instlist_]:= dccOpCodeToString /@ instlist

dccOpCodeToString[inst_]:=MapAt[opCodes[[1+#]]&,inst,{1}]

(* three equivalent definitions: Pick your favorite one. *)
dccNumberLines[instlist_]:=Table[Prepend[instlist[[i]],i],{i,Length[instlist]}]
dccNumberLines[instlist_]:=MapThread[Prepend,{instlist,Range[Length[instlist]]}]
dccNumberLines[instlist_]:=MapIndexed[Join[#2,#1]&,instlist]

(* If you get a CompiledFunction, you can use dcc[%] to see its code *)
```

An ugly little program for disassembling a compiled function

4 How Compile Works

`Compile` needs to transform a nested functional expression into a linear sequence of simple instructions. To do this, it views most functions as objects which produce a value based on the values of their arguments. So the compiled form of a subexpression is a sequence of instructions designed to place the "result" of that subexpression into a certain register. For example, to compile `Plus[x^2, y, 2 z]`, the three subexpressions get compiled, and then code is written to add the three results into a new register, which is then considered to be the result of this subexpression.

This creates a useful function that lets you choose between 3 and 4.

```
In[1]:= Compile[{{a, True|False}}, If[a, 3, 4]]

Out[1]= CompiledFunction[{a}, If[a, 3, 4], -CompiledCode-]
```

The compiled code stores the result of the `If` in an integer register, so an exact integer is returned.

```
In[2]:= %[True]

Out[2]= 3
```

`Compile` now has to use a real register to hold the result of the `If`.

```
In[3]:= Compile[{{a, True|False}}, If[a, 3, 4.5]]

Out[3]= CompiledFunction[{a}, If[a, 3, 4.5], -CompiledCode-]
```

The result is now returned as a real number.

```
In[4]:= %[True]

Out[4]= 3.
```

There are two exceptions to this basic approach to compiling functions.

The first is that some functions, such as `Do`, do not return any value, but instead always return Null. This is the equivalent of a function of type void in C, or a procedure in Pascal. For these functions, the compiled code does not try to put the "result" into a register. If we have an external function whose result is non-numerical and/or irrelevant, and we are getting an error because the return value is not what's expected, we may want to use `Compile`'s third argument to tell it to ignore the result, specifying the return type as `Null`.

The second is that many heads such as `For`, `While`, `Product`, `Throw`, `Goto`, `Module`, and others, are better viewed as a structural description of how the code in and around them should be put together than as a numerical function of numerical arguments, although some of these can return numerical values.

In structures such as `Module` or `Block`, local variables can be declared. The compiled code will store the value of such a variable in a register, which can then be updated whenever the variable gets a new value. This presents a problem, namely, how to guess what the *type* of the variable should be, so that `Compile` can know what type of register to use. `Compile` guesses the type of a variable based on *the first time it is set*. This is important to remember.

The first time `y` is set, it is set to zero, an integer. Based on this, `Compile` guesses that `y` is a variable of type integer, and uses an integer register to hold its value. This causes a problem when we try to give it the value `y + 2x`, since this value is a real number (because `x` is assumed to be a real number).

```
In[5]:= Compile[x, Module[{y=0}, y += 2x; y]]

Compile::cset:
   Warning: Variable y of type _Integer
     encountered in assignment of type _Real.

Out[5]= CompiledFunction[{x}, Module[{y = 0}, y += 2 x; y],
  -CompiledCode-]
```

Since the type of the first thing `y` is set to matches the type we want it to have, this works fine.

```
In[6]:= Compile[x, Module[{y=0.0}, y += 2x; y]]

Out[6]= CompiledFunction[{x}, Module[{y = 0.}, y += 2 x; y],
  -CompiledCode-]
```

At the end of the first section, in the function `compiledIterationCount`, the local variable `z` was initialized to `I` so that `Compile` would know that `z` should be considered a complex variable. Otherwise, `z` would have been considered to be an integer (due to its first assignment), and `Compile` would have complained when it got to the point where `z` gets set to `z^2 + c`, since this sum has a complex result (because `c` is declared as a complex argument to `compiledIterationCount`).

We have noted earlier that there are functions (such as `BesselJ`) for which it is faster to use ordinary *Mathematica* to compute the value than to try to execute complicated compiled code for computing the value. For this reason, `Compile` generates instructions to compute such instructions externally. Sometimes it is useful to know exactly what function `Compile` does or does not try to compile. We end this section with a list of the functions that `Compile` will compile into compiled code.

`Plus`	`Times`	`Power`
`Minus`	`Subtract`	`Divide`
`Sqrt`	`Exp`	`Log`
`Sin`	`Cos`	`Tan`
`ArcSin`	`ArcCos`	`ArcTan`
`Csc`	`Sec`	`Cot`
`ArcCsc`	`ArcSec`	`ArcCot`
`Sinh`	`Cosh`	`Tanh`
`ArcSinh`	`ArcCosh`	`ArcTanh`
`ArcCsch`	`ArcSech`	`ArcCoth`
`Round`	`Floor`	`Ceiling`
`N`	`Sign`	`Abs`
`Max`	`Min`	`Re`
`Im`	`Conjugate`	`Arg`
`Mod`	`Quotient`	

Numerical functions.

`If`	`Switch`	`Which`
`Equal`	`Unequal`	`Less`
`LessEqual`	`Greater`	`GreaterEqual`
`SameQ`	`EvenQ`	`OddQ`
`And`	`Or`	`Not`

Testing functions.

`Module`	`Block`	`With`

Structures for creating local variables.

`Set`	`AddTo`	`SubtractFrom`
`TimesBy`	`DivideBy`	`Increment`
`PreIncrement`	`Decrement`	`PreDecrement`

Setting variables.

`Sum`	`Product`	`Do`
`For`	`While`	

Looping structures.

`CompoundExpression`	`Break`	`Continue`
`Return`	`Goto`	`Label`
`Throw`	`Catch`	

Transfer of control.

If a function is not among those listed above, `Compile` will make instructions to evaluate the function externally. `Compile` will have to guess what the return type will be, and whether to create compiled code to try to evaluate the function's arguments. Here are the rules it uses to guess:

- If the function appears in a spot where its return value is known not to matter, such as in a non-final position of a `CompoundExpression` or as the body of a `While` or `Do` or as the initialization, increment, or body of a `For`, assume a return type of `Null` (which means to ignore the returned value), and don't compile the function's arguments.
- If the third argument to `Compile` contained a type specification applicable to this function, use the information given there.
- If the function has attribute `HoldFirst`, `HoldRest`, or `HoldAll`, don't create code for the function's arguments.
- If the function is `SetDelayed` or `Print`, assume a return type of `Null`.
- If the function contains any imaginary pieces, or is located anywhere inside a function that expects a complex argument (such as `Re` or `Conj`), assume a return type of `_Complex`.
- If none of the above apply, assume a return type of `_Real`.
- If none of the function's arguments are themselves functions, don't create code for them.
- If any argument of the function is uncompilable (like a string or a list), code is not created for the function's arguments.