University of Amsterdam
System and Network Engineering Lab
Dr Clemens Grelck

Compilerbouw
2014/2015
February 6, 2015

Course Project
CiviC Language Manual
Document Version 2.1

1 Introduction

The course project is the step-wise engineering of a fully-fledged compiler for the model programming language CiviC (Civilised C). In this context the term *model programming language* emphasises that CiviC is not meant to be the next big hype in programming language research, but rather it is a programming language that exhibits characteristic features of structured imperative programming languages in general while at the same time its design avoids repetition of features and thus is sufficiently restricted in size and complexity to actually succeed in implementing it in an 8-week course.

This document describes CiviC, the source language for the compiler project. The target language is assembly code for the CiviC virtual machine, CiviC-VM, which could best be described as a simplified variant of the Java virtual machine. The virtual machine is described in a separate document made available as the course proceeds.

Whenever this document (deliberately or not) leaves certain aspects of the CiviC language unspecified or, more precisely, under-specified, the corresponding rule of the C language shall take effect.

2 CiviC Programs

A CiviC program is a non-empty sequence of function declarations, function definitions, global variable declarations and global variable definitions. Fig. 1 shows the corresponding syntax used.

CiviC supports separate compilation in a similar way as C does. The scope of symbols, i.e. functions and global variables, can either be confined to the compilation unit (i.e. the file) they are defined in or their scope can extend to the entire program typically consisting of a number of such compilation units. Unlike C, CiviC is conservative with respect to visibility between compilation units. This means that by default the scope of a symbol is limited to the current compilation unit. Only if the definition of the function or global variable is preceded by the key word `export`, the symbol is visible outside the current compilation unit.

If some symbol defined outside the current compilation unit is to be used, a local function or global variable declaration is required. Like in C, we use the key word `extern` for this purpose. The C preprocessor is run on each source file by the CiviC compiler. This allows for the use of source and header files just as in C.

A CiviC program, consisting of potentially multiple compilation units, must contain exactly one exported function named `main`, which serves as the starting point of program execution. This function is supposed to take no parameters (no support for command line interpretation for now) and yields an integer value that will be returned to the calling environment, typically a Unix command shell.

Program	⇒	[*Declaration*]+
Declaration	⇒	*FunDec* | *FunDef* | *GlobalDec* | *GlobalDef*
FunDec	⇒	**extern** *FunHeader* ;
FunDef	⇒	[**export**] *FunHeader* { *FunBody* }
FunHeader	⇒	*RetType* *Id* ([*Param* [, *Param*]*])
RetType	⇒	**void** | *BasicType*
GlobalDec	⇒	**extern** *Type* *Id* ;
GlobalDef	⇒	[**export**] *Type* *Id* [= *Expr*] ;
Type	⇒	*BasicType*
BasicType	⇒	**bool** | **int** | **float**
Param	⇒	*Type* *Id*

Figure 1: Top-level syntax of CiviC programs

Identifiers may consist of letters, both small and capital, digits and the underscore character. However, identifiers must start with a letter. This restriction facilitates the introduction of fresh identifiers that are guaranteed not to conflict with any user supplied identifiers.

3 Statement Language

Fig. 2 shows the statement syntax of CiviC. The body of a CiviC function consists of a potentially empty sequence of declarations of local variables followed by an again potentially empty sequence of statements and an optional return statement. Like in C variables can be initialised at the point of declaration. Unlike in C, only one variable can be declared at a time. Again unlike C, the return statement must be the last statement within a function's body and, consequently, no more than one return statement per function is allowed. CiviC only supports the declaration of local variables in the beginning of function bodies and not in further (nested) blocks.

A statement is either an assignment, a procedure call or a control flow construct. Conditionals and uncounted loops (`if`-, `while`- and `do`-statements) are equivalent to their C counterparts with the exception that any predicate expression must evaluate to a Boolean value. More precisely, CiviC explicitly distinguishes between integer and Boolean values and variables.

Unlike C, CiviC features a proper counted loop with a dedicated induction variable. The three expressions that must be of type `int` denote the (inclusive) start value, the (exclusive) stop value and (optionally) the increment/decrement step value. The default step value is 1. With a positive step value the start value becomes the inclusive lower bound of the iteration space and the stop value the exclusive upper bound. A step value of zero is not permitted and may lead to undefined behaviour. With a negative step value the start value becomes the inclusive upper bound and the stop value the exclusive lower bound of the iteration space. All three expressions are evaluated exactly once, before the execution of the loop. Assignments to the induction variable inside the loop body are illegal. Consequently, the trip count of a CiviC `for`-loop is always known in

```
FunBody    ⇒  [ VarDec ]* [ Statement ]* [ Return ]

VarDec     ⇒  Type Id [ = Expr ] ;

Statement  ⇒  Id = Expr ;
           |   Id ( [ Expr [ , Expr ]* ] ) ;
           |   if ( Expr ) Block [ else Block ]
           |   while ( Expr ) Block
           |   do Block while ( Expr ) ;
           |   for ( int Id = Expr , Expr [ , Expr ] ) Block

Block      ⇒  { [ Statement ]* }
           |   Statement

Return     ⇒  return Expr ;
```

Figure 2: Syntax of CiviC statement language

advance. The scope of the induction variable is confined to the loop body. The special nature of the induction variable is emphasised by preceding its definition by the type key word `int`, which resembles the syntax of a variable declaration. Technically, of course, the induction variable must be of type `int` anyways, hence no additional information is provided.

Like in C, any function regardless of its return type can be referred to in a procedure call. In case that function turns out not to be a (`void`-function), but in fact does yield a value, that value is simply discarded. Last but not least, CiviC supports single-line as well as multi-line comments in the same style as C and C++.

4 Expression Language

The syntax of the CiviC expression language is shown in Fig. 3. It features the usual arithmetic, relational and logical operators as known from C. All operator associativities and precedences are defined as in C proper.

All binary operators are only defined on operands of exactly the same type, i.e. there is no implicit conversion in CiviC. Arithmetic operators are defined on integer numbers, where they again yield an integer number, and on floating point numbers, where they again yield a floating point number. The modulo operator is an exception: it is only defined on integer numbers and yields an integer number. The arithmetic operators for addition and multiplication are also defined on Boolean operands where they implement strict logic disjunction and conjunction, respectively.

Regardless of their operand types, relational operators yield a Boolean value. The relational operators for equality and inequality are defined on all basic types. On Boolean operands they complement strict logic disjunction and conjunction in supporting all potential relationships between two Boolean values. The remaining four relational operators are only defined for integer and floating point numbers as operand values.

The logic operators are only defined for Boolean operand values. They differ from arithmetic and relational operators in one more aspect: if the value of the left operand determines the operation's result, the right operand expression is not evaluated. This operational behaviour is called *short circuit Boolean evaluation*. The operators are also called to be *non-strict* or *lazy* in their right operand expression. This is more than an (optional) optimisation as the following predicate illustrates: `(x!=0 && y/x > 42)` .

$$
\begin{array}{lcl}
\mathit{Expr} & \Rightarrow & \texttt{(}\ \mathit{Expr}\ \texttt{)} \\
 & | & \mathit{Expr}\ \mathit{BinOp}\ \mathit{Expr} \\
 & | & \mathit{MonOp}\ \mathit{Expr} \\
 & | & \texttt{(}\ \mathit{BasicType}\ \texttt{)}\ \mathit{Expr} \\
 & | & \mathit{Id}\ \texttt{(}\ [\mathit{Expr}\ [\ \texttt{,}\ \mathit{Expr}\ ]^*\ ]\ \texttt{)} \\
 & | & \mathit{Id} \\
 & | & \mathit{Const} \\
 & & \\
\mathit{BinOp} & \Rightarrow & \mathit{ArithOp}\ |\ \mathit{RelOp}\ |\ \mathit{LogicOp} \\
 & & \\
\mathit{ArithOp} & \Rightarrow & \texttt{+}\ |\ \texttt{-}\ |\ \texttt{*}\ |\ \texttt{/}\ |\ \texttt{\%} \\
 & & \\
\mathit{RelOp} & \Rightarrow & \texttt{==}\ |\ \texttt{!=}\ |\ \texttt{<}\ |\ \texttt{<=}\ |\ \texttt{>}\ |\ \texttt{>=} \\
 & & \\
\mathit{LogicOp} & \Rightarrow & \texttt{\&\&}\ |\ \texttt{||} \\
 & & \\
\mathit{MonOp} & \Rightarrow & \texttt{-}\ |\ \texttt{!} \\
 & & \\
\mathit{Const} & \Rightarrow & \mathit{BoolConst} \\
 & | & \mathit{IntConst} \\
 & | & \mathit{FloatConst} \\
 & & \\
\mathit{BoolConst} & \Rightarrow & \textbf{true} \\
 & | & \textbf{false}
\end{array}
$$

Figure 3: Syntax of CiviC expression language

Unlike C, CiviC does not support implicit conversion between values of the different basic types. Consequently, any type mismatch between operand expressions and operator expectations, as detailed above, is a type error. Furthermore, types of argument expressions in function applications must match the corresponding declared parameter types. Likewise, the type of a function's return expression must match the function's return type.

Cast expressions can be used to explicitly convert values between the three basic types. Conversions between integer and floating point numbers are defined as in C. The conversion of the Boolean value `false` into an integer or a floating point number yields `0` or `0.0`, respectively. Likewise, the conversion of the Boolean value `true` into an integer or a floating point number yields `1` or `1.0`, respectively. The conversion of both integer and floating point numbers into Boolean values is equivalent to applying the inequality operator to that number and the numerical constant `0` or `0.0` respectively.

The unary minus operator can be applied to both integer and floating point numbers, whereas the unary negation operator can only be applied to Boolean values. Last not least, numerical constants are defined as in C proper, just as function calls with and without return value assignment.

5 Scoping Rules

The scoping rules in C are fairly odd when it comes to comparing assignment statements and variable declarations with initialisation. Take for instance the assignment statement

```
  a = a + 1;
```

that increments the value of integer variable `a`. In this example, quite obviously, the `a` on the right hand side of the assignment refers to the old value of `a`, which is then overwritten by the incremented value. In other words the new value of a is only visible from the subsequent statement onwards.

In a variable declaration with initialisation expression, however, almost the same code example has a very different meaning. In a C variable declaration

```
int a = a + 1;
```

the scope of the newly declared variable a does extend to the initialisation expression. In the above example, the a in the initialisation expression actually refers to the just declared variable a, not a possibly earlier declared variable a. As a consequence, the value of a on the right hand side is undefined no matter what, and hence the attempt to actually initialise it fails as the value of the initialisation expression is again undefined. This is the case regardless of whether or not an outer declaration of a variable named a exists (e.g. as a global variable) and whether or not that has a defined value. Unfortunately, this is a rather subtle bug in practice as C does not rule out undefined values, they may just differ between program runs.

In the spirit of the name Civilised C we decided for a different definition of scopes that treats both examples above in the same way. More precisely, the initialisation expression of a variable declaration is *not* in the scope of the declared variable. Technically, a compiler would always first traverse the initialisation expression in the existing scope before creating a new variable a and updating the current scope accordingly.

Scoping rules for functions are somewhat different from those for variables. Firstly, CiviC syntactically distinguishes between functions and variables and their scopes are clearly separated. In other words, a function definition never shadows a variable declaration and vice versa.

Unlike in C, all function definitions are visible and thus callable from all other function definitions (on the same level, see extension 1 below). The textual order in which functions are defined does not matter. As a consequence, so-called forward declarations as in C and many other imperative languages are obsolete, and function definitions can be mutually recursive without restrictions.

Why are does the textual order matter for variable declarations but not for function definitions?

The main motivation for this choice is that mutual recursion between functions is very useful, see the following text book example:

```
bool odd( int x)
{
  int odd;
  if (x == 0) odd = false;
  else odd = even( x-1);
  return odd;
}

bool even( int x)
{
  int even;
  if (x == 0) even = true;
  else even = odd( x-1);
  return even;
}
```

In contrast mutually recursive definition of variables makes little sense as the following example clearly illustrates:

```
int a = b+1;
int b = a+1;
```

That is why we use different scoping rules for functions and variables and emphasise their differences.

6 CiviC Standard Library

In analogy to C, CiviC does not provide any built-in operations to communicate with the execution environment other than the integer value returned by the main function. CiviC support for separate compilation opens an avenue to provide support for input and output through a standard library of functions and global variables. For the time being, assume the availability of the following 6 functions:

```
extern void printInt( int val);
extern void printFloat( float val);

extern int scanInt( );
extern float scanFloat( );

extern void printSpaces( int num);
extern void printNewlines( int num);
```

that write integer and floating point numbers to the standard output stream, read them from the standard input stream or write a given number of space or newline characters to the standard output, respectively. In conjunction they allow you to do simple numerical input/output and provide basic formatting capabilities. The above function definitions can be made available to a CiviC program by including the header file `civic.h`. Remember that the C preprocessor is used by the CiviC compiler.

7 Extension 1: Nested Functions

As an extension to the core language CiviC features nested function definitions according to the extended syntax shown in Fig. 4. With this extension the body of a function may again contain function definitions, located between the local variable declarations and the statement sequence. The scope of these local functions is confined to the body of the outer function, hence the name.

FunBody ⇒ [*VarDec*]* [*LocalFunDef*]* [*Statement*]* [*Return*]

LocalFunDef ⇒ *FunHeader* { *FunBody* }

Figure 4: Extended syntax for nested function definitions

Local functions can be called from within the subsequent statement sequence of the parent function. Local functions defined on the same nesting level can call each other regardless of the textual order of definitions. This is the same as for the flat sequence of function definitions, as defined initially. Local functions may call functions defined in any (directly or indirectly) surrounding scope. In addition to their own parameters and local variables, local functions can access the parameters as well as the local variables of any parent function. Of course, local functions can access global variables as well. The two examples in Fig. 5 illustrate the scoping rules for nested function definitions.

They can be called from within the subsequent statement sequence. The local functions defined on the same nesting level may call each other regardless of the textual order of definitions. Local functions may call any function defined in a surrounding scope. In addition to their own parameters and local variables, local functions can access the parameters as well as the local variables of any surrounding function. Of course, local functions can also access global variables. The two examples in Fig. 5 illustrate the scoping rules of nested function definitions.

```
void foo()
{
   void bar()
   {
      baz(); // OK, no forward declaration needed
   }

   void baz()
   {
      bar();
   }
}
```

```
------------------------------------------------------

void foo()
{
  void bar()
  {
    baz(); // OK, baz within scope
  }
}

void baz()
{
  bar(); // error: bar not visible outside of foo
}
```

Figure 5: Example code illustrating the scopes of function definitions

8 Extension 2: Multi-dimensional Arrays

The scalar core of CiviC shall be extended by multi-dimensional arrays. Fig. 6 shows the syntactic extensions. Arrays can be passed as arguments to functions. Parameter passing for arrays is always call-by-reference. Like in C arrays are indexed from 0 to n-1. Arrays can be multidimensional and always carry their extents along each dimension with them. For example, supposed a function has a parameter `int[m,n] x`, then this denotes an integer $m \times n$-matrix, and the extents `m` and `n`, which (implicitly) are of type `int`, can be used in the function body as ordinary parameters. CiviC arrays always have one of the basic types as element types, i.e. there is no support for arrays of arrays.

CiviC supports multidimensional indexing into arrays both in expression positions as well as in assignment statements. The number of integers used for indexing must exactly conform with the number of dimensions in the declaration of the array. Index expressions must be of type `int`.

The square bracket notation is convenient to define array-valued expressions and to initialise reasonably small arrays with little syntactic overhead. If the array defined using this notation has fewer elements than the declaration of the array it is assigned to suggests, some elements of that array remain uninitialised. If the number of expressions within square brackets exceeds the declared size of an array, this is a program error.

Multidimensional arrays must be initialised by a corresponding nesting of square bracket expressions, i.e. an array declared to be 2-dimensional can only be initialised by a 2-dimensional array value. Alternatively, an array of any number of dimensions can be assigned a scalar value. In this case, all elements of the array are uniformly set to that value. The obvious extension that supports the initialisation of a multi-dimensional array by an array value of lower dimensionality through replication along the outer dimensions is not required, but could be taken as a challenge.

Note that array values are only permitted in the initialisation expressions of variable declarations.

```
GlobalDec   ⇒   extern Type [ [ Id [ , Id ]* ] ] Id ;

GlobalDef   ⇒   [ export ] Type Id [ = Expr ] ;
            |   [ export ] Type [ Expr [ , Expr ]* ] Id ;

Param       ⇒   Type [ [ Id [ , Id ]* ] ] Id

VarDec      ⇒   Type [ [ Expr [ , Expr ]* ] ] Id [ = ArrExpr ] ;

Statement   ⇒   ...
            |   Id [ Expr [ , Expr ]* ] = Expr ;

ArrExpr     ⇒   [ ArrExpr [ , ArrExpr ]* ]
            |   Expr

Expr        ⇒   ...
            |   Id [ Expr [ , Expr ]* ]
```

Figure 6: Extended syntax for CiviC arrays

Arrays declared in the scope of a function may have dynamic extents computed at runtime. Memory management for arrays is automatic. They are created (in the heap) as program execution reaches their declaration and their life time is determined by the scoping rules. Arrays cannot be returned by functions.

Note the following difference in the syntax of array types depending on whether they are used in global variable declarations and function parameters on the one hand side or global variable definitions and local variable declarations on the other side. While in the former case array indices are restricted to identifiers, fully-fledged expressions can be used in the latter case. The motivation for this distinction is as follows. Both global variable definitions and local variable declarations define an array object. Hence, we need to specify the shape of that array properly. Fully-fledged expressions in index position support even computations based on the shapes of existing arrays and other values. Variables occurring in these index expressions are accessed reading. In contrast, both global variable declarations and function parameters refer to an object that has been declared before. Hence, the index variables provide access to the shape properties of the array within the current context.These index variables are written to.

9 The Role of Extensions

Extensions are optional for passing the course, but they do affect your grade.

We strongly suggest to only start working on one or both extensions once you have successfully accomplished the same compilation step for the language core. If you struggle, do not waste time on extensions.

As an illustration: building a complete compiler for the core language that generates correct code for the CiviC virtual machine will let you pass the course. In contrast, an incomplete compiler that accepts programs including all extensions, but fails to generate correct VM code even for the core language, will not let you succeed.