University of Amsterdam

Dr. Clemens Grelck
Informatics Institute
Systems and Networking Lab
Parallel Computing Systems Group

CiviC
Compiler Project Milestones

Document Version 1.0

Clemens Grelck

Introduction

This document is supposed to help you structure your work towards a fully-fledged CiviC compiler. For this we define 15 milestones. Some milestones only take a few words to describe, e.g. implement a scanner / a parser for CiviC. Other milestones provide detailed suggestions for compiler passes that systematically contribute to compiling source level CiviC code into assembly for the CiviC-VM.

Milestones have due dates, but milestones are not assignments. The due dates rather form a reference time scale for completing the compiler project in time. Should you be ahead of schedule, that's perfect. Don't forget to do the extensions. Should you find yourself behind time, seek assistance during the labs and discuss with us what leads to the delay and what could be done to catch up.

Milestone 1: Lexicographic Analysis

Extend the lex-based scanner coming with the compiler framework towards a fully-fledged lexicographic analysis for the full range of the CiviC language.

Due date: Week 2

Milestone 2: Intermediate Representation

Refine your intermediate representation (abstract syntax tree) for your CiviC compiler reflecting upon the discussions during the labs. Extend it to cover the entire language including extensions. If you need further attribute types, provide their implementation alongside (types.h).

Note that a good intermediate representation does not necessarily follow the grammar that defines the source language one-to-one. A context-free grammar is not a unique description of a language, and thus many different context-free grammars would define the same language. The style in which a grammar is defined could be motivated by reasons that do not apply to intermediate representations of compilers. For example, the grammar used to define the syntax of CiviC is fairly verbose and uses various levels of non-terminal symbols merely for the purpose of documenting the various language features.

A good intermediate representation liberates itself from the concrete syntax definition of a language and aims at being both concise and handy throughout compilation.

Due date: Week 3

Milestone 3: Visualisation of Intermediate Representation

Extend the existing compiler traversal that prints the intermediate representation of a CiviC program to cover the full syntactic range of the CiviC language.

Ensure suitable indentation and formatting to properly illustrate the logical structure of the code.

Due date: Week 3

Milestone 4: Syntactic Analysis

Extend the yacc-based parser (syntactic analysis) coming with the compiler construction framework to cover the full syntactic range of the CiviC language. The parser implementation shall create an abstract syntax tree according to your specification of mile stone 2 in case the program is found to be well-formed. Otherwise, the parser shall produce a meaningful error message that allows the programmer to relate the error back to the original source code.

Make sure that your parser is free of shift/reduce and reduce/reduce conflicts.

The production/derivation rules in the parser do not necessarily have to follow the specification of the context-free grammar as used to define the source language proper. The same reasons as brought forward under mile stone 2 for the definition of an intermediate representation apply. Notwithstanding, the parser grammar must implement the exact same language as described by the specification.

Test your scanner/parser in conjunction with the visualisation facilities developed for mile stone 3 using the example CiviC programs developed in the beginning of the course as well as those provided as a test suite.

Due date: Week 4

Milestone 5: Context Analysis

During context analysis applied occurrences of identifiers, i.e. variables in expressions, left hand side variables in assignment statements and function names in function calls, are associated with their corresponding definition or declaration according to the scoping rules of the CiviC language. A proper error message must be produced if the necessary declaration of a used identifier is missing (no matching declaration/definition) or is ambiguous (multiple matching declarations/definitions). Likewise argument numbers in function calls must match parameter numbers of called function.

Note that matching types is not done here but left for a separate type checking pass.

The compilation process shall as far as possible continue in the presence of context errors in order to report multiple such errors in a single compiler run. However, stop compilation after context analysis in case errors have occurred.

In addition to ruling out context errors an important aspect of context checking is to enhance the intermediate representation such that future compiler passes can easily extract context information, e.g. types, from variable representations. The aim is to store the findings of context analysis in the intermediate representation instead of rerunning context analysis itself any time the corresponding information is needed in subsequent compiler passes.

There are many ways of technically accomplishing context analysis in a compiler intermediate representation; in the following we describe possible ways.

We suggest to extend each context, i.e. nested function definitions and the global context, by a symbol table for variables. Despite the name, this symbol table could simply be a list of symbol table entries, set up of corresponding to be defined AST nodes. Each symbol table entry would feature the name of the variable as a character string, its type as well as its nesting level (starting with zero for the global context). Symbol table entries can and should be extended by all information about the variable gathered during the compilation process.

If defined as above, the symbol table looks very similar to the internal representation of the list of parameters of a function or the list of local variable declarations. So, why having another list? From a syntactic point of view a variable in CiviC can be defined in three different ways: as a local variable, as a function parameter or as a global variable. All three ways typically have different representations in the compiler's IR while at the same time they hold similar though not identical information, among others the name of the variable and its declared type. Consequently, a possible reference from the node representing some variable in an expression to its point of declaration could potentially target three different AST node types. This is of course possible, but from experience very unhandy during the further compilation process.

Overcoming this problem, a symbol table entry is in essence nothing but a unique representation of the common features of all three forms of variable declarations. At this stage of the compilation the original representations of variables in the abstract syntax tree could be (and should be) replaced by other representations that instead of holding partial information directly, i.e. mainly the variable name, have a reference ("link") to the corresponding symbol table entry.

Make sure that your compiler appropriately prints the symbol table, for instance as a structured comment in the beginning of the function body or preceding the entire function definition.

In principle, we could apply the same techniques for functions as for variables. This would in particular be recommended if your abstract syntax tree foresees different representations for function definitions and for (external) function declarations. In this case we would be confronted with the exact same situation as with variables, namely that a function call could refer to either a function definition or a function declaration, both having different AST representations.

However, in the (fairly simple) context of CiviC we suggest (but in no way prescribe) a slightly simpler solution. Externally declared functions could be represented in the very same way as locally defined functions are, simply by making the function body optional in the intermediate representation. In this way a single type of AST node could easily be used to represent either function declarations or function definitions, and function calls could uniformly be equipped with a reference to the corresponding node representing the called function.

These two different solutions among others demonstrate the range of design choices that is characteristic for compiler construction in general.

Context analysis disambiguates equally named symbols according to the scoping rules of the language, both variables and functions. For documentation as well as debugging purposes this disambiguation should also be visualised when displaying the abstract syntax tree after context analysis. This could, for example, be achieved by consistent renaming of identifiers incorporating a suitable representation of the scope level into the variable name. Alternatively, you could also simply print the scope information together with the variable name when visualising the abstract syntax tree.

Last but not least, remove the declaration part from `for`-loop induction variables and create corresponding local variable declarations on the level of the (innermost) function definition. Beware of nested `for`-loops using the same induction variable and occurrences of identically named variables outside the scope of the corresponding `for`-loop. Like explained above, context disambiguation and possibly a systematic renaming of `for`-loop induction variables is needed.

Due date: Week 5

Milestone 6: Turning Variable Initialisations into Regular Assignments

At least in the case of local and global variables the symbol table could entirely substitute the original declarations, but as of now either declarations may still feature initialisation expressions. It is now time to transform them into regular assignments to the corresponding variables.

In the case of local variables the new assignment statements can simply prefix the original sequence of statements, but for global variables it is a-priori unclear where such initialisation assignments could be stored. Therefore, the compiler shall create one top-level function named `__init` in every compilation unit. In this function the compiler collects all initialisation assignments for global variables in the order of their syntactic appearance in the original code. This `__init` function will later also be used by the CiviC-VM.

Another area that requires special attention is the initialisation of arrays. Recall that CiviC supports two forms of array initialisation: heterogeneous initialisation through possibly nested sequences of values enclosed in square brackets as well as homogeneous initialisation by a single scalar value. Replace the former by a sequence of scalar array assignments, and substitute the latter by an appropriate nesting of `for`-loops.

Beware that in CiviC the (scalar) initialisation expression may contain a function call and that function calls may be side-effecting, e.g. by doing any form of input/output or by manipulating the values of global variables. Make sure that the execution of function calls is not duplicated by your above transformations and that the original meaning of the code is preserved even under these circumstances. As an illustration of the problem consider the following CiviC program:

```
int twos = 0;

int two()
{
  twos = twos + 1;
  return 2;
}

void foo()
{
  int[2, two()] a = two();
  printInt( twos);
  printInt( a[1, 1]);
}
```

The correct behaviour in this marginally tricky CiviC example program is to evaluate the function two() exactly twice. Since it always yields the value 2, the array a is a 2×2-matrix, where each element is set to 2. Therefore, the two subsequent print statements each print the value 2.

A possible solution for these problems is to transform the above code in a systematic way such that complex expressions (in particular function calls) in index or arrays initialisation expressions are removed and substituted by fresh variables introduced by the compiler. The above example, for instance, could be transformed into the following intermediate representation:

```
int twos = 0;

int two()
{
  twos = twos + 1;
  return 2;
}

void foo()
{
  int tmp_1 = 2;
  int tmp_2 = two();
  int tmp_3 = two();
  int[tmp_1, tmp_2] a = tmp_3;
  printInt( twos);
  printInt( a[1, 1]);
}
```

From here the function foo could further be transformed into:

```
void foo()
{
  int tmp_1;
  int tmp_2;
  int tmp_3;
  int[tmp_1, tmp_2] a;
  int tmp_4;
  int tmp_5;

  tmp_1 = 2;
  tmp_2 = two();
  tmp_3 = two();
  a = __allocate( tmp_1, tmp2);

  for (tmp_4 = 0, tmp_1) {
    for (tmp_5 = 0, tmp_2) {
      a[tmp_4, tmp_5] = tmp_3;
    }
  }

  printInt( twos);
  printInt( a[1, 1]);
}
```

Note the changed meaning of the array declaration in line 6. Previously, the array a came into existence when (symbolically) executing this line of code. Now, it merely represents the fact that a is a tmp_1×tmp_2 array. Note in particular the introduction of the pseudo function __allocate in the function body that explicitly allocates heap memory for the array before initialising it with the specified value.

It is noteworthy that the notations used above, e.g. the allocation pseudo function, are just examples of a potential textual visualisation of an appropriate intermediate representation. As they are independent of the source language, you are free to choose any intermediate representation and any pseudo-syntax for pretty printing that you deem appropriate.

Due date: Week 5

Milestone 7: Type Checking

Type checking ensures that operators are applied to arguments of supported types, functions are called with arguments of correct type as specified by the function definition or declaration and values of correct type are assigned to variables as defined by the corresponding variable declaration.

Type checking naturally falls into two separate tasks: type inference for expressions and type matching for assignments and function calls. Type inference infers the type of an expression based on the declared types of identifiers, the natural types of constants, the typing rules of built-in operators and type declarations of defined functions. Type matching checks whether inferred types match declared types, e.g. in assignments to variables, in argument/return positions of function calls or in predicate positions of control flow constructs. For arrays, type checking involves additional dimensionality checks. For example, reading from an array and writing to an array requires exactly as many indices as the array has dimensions.

Non-matching types shall lead to meaningful error messages. The type checker should report multiple errors before termination.

Type checking takes advantage of the preceding context analysis as declared types of variables and functions are easily accessible from every occurrence following the reference to the identifier's declaration. Think about a suitable compiler-internal representation of the type signatures of built-in operators.

Due date: Week 6

Milestone 8: Parameter Passing for Arrays

Arrays can be passed as arguments into functions, as illustrated in the following code example:

```
void foo( int[m,n] a)
{
  bar( a);
  baz( a, a);
}
```

Here, the special type parameters `m` and `n` allow us to access the extent of argument array a along each of the two dimensions within the function bodies of `foo()` and `bar()`. At runtime these properties of arrays must also be passed into functions along with the array itself. A relatively simple compiler transformation is supposed to turn the above code example into

```
void foo( int m, int n, int[m,n] a)
{
  bar( m, n, a);
  baz( m, n, a, m, n, a);
}
```

where the implicit index arguments of multi-dimensional array parameters become explicit regular function parameters as well as explicit additional arguments in function calls.

Note that in the above example it may be tempting to reduce the number of arguments given to function `baz`, but of course this is not possible (at least in the general case) because function `baz` may also be called elsewhere in the code with different argument arrays.

Due date: Week 6

Milestone 9: Compiling Boolean Disjunction and Conjunction

The evaluation of standard Boolean conjunction and disjunction in CiviC (and almost all other programming languages) significantly differs from that of all other binary operators. Arithmetic and relational operators first evaluate their left operand expression followed by their right operand expression and only then perform the operation itself. In contrast, Boolean disjunction and conjunction operators only evaluate their right operand if following the evaluation of the left operand the right operand determines the result of the operation.

This behaviour, called *short circuit Boolean evaluation*, cannot be mimicked by special VM instructions alone. The standard compiler approach is to systematically replace disjunction and conjuction operators by semantically equivalent if-then-else constructs. However, experience tells us that this can be challenging, in particular in the presence of deeply nested Boolean expressions.

Instead, we recommend to extend the intermediate representation of the compiler by *conditional expressions* as exemplified by the C ternary operator *`pred?then:else`* and to systematically transform all Boolean operations in question into semantically equivalent conditional expressions. Conditional expressions may be left until code generation, which turns out to be much easier.

Note that CiviC also features *eager* disjunction (Boolean addition) and conjunction (Boolean multiplication) with standard left-to-right operand evaluation semantics. They are not concerned by this milestone and can be compiled into corresponding VM instructions during code generation.

Due date: Week 6

Milestone 10: Compiling Boolean Cast Expressions

The CiviC language features cast expressions to convert values between all three basic types, possibly with loss of precision. The CiviC-VM, however, only has conversion instructions between integer numbers and floating point numbers and vice versa.

Conversions between Boolean values and numerical values or vice versa must be represented differently in the intermediate representation, e.g. again by making use of conditional expressions as already recommended for milestone 9. This milestone 10 is about implementing a compiler pass for the systematic transformation of cast expressions with Boolean argument or result value into semantically equivalent non-cast expressions.

Due date: Week 6

Milestone 11: Array Dimension Reduction

The targeted CiviC-VM only supports flat vectors rather than multi-dimensional arrays as the CiviC language. This is a common restriction in many programming languages, and only few languages have full support for truly multi-dimensional arrays.

As a consequence, multi-dimensional CiviC arrays must explicitly be lowered to single-dimensional arrays. This involves array selection both on the left hand side of assignments as well as in expression positions, the allocation of arrays and the parameter passing of array arguments. The following code example (implementing matrix transposition) illustrates this step.

```
void transpose( int m, int n, int[m,n] a)
{
  int i;
  int j;
  int[n,m] b;

  b = __allocate( n, m)

  for (i = 0,n) {
    for (j = 0,m) {
      b[i,j] = a[j,i];
    }
  }

  bar( n, m, b);
}
```

should be transformed into

```
void foo( int m, int n, int[] a)
{
  int i;
  int j;
  int[] b;

  b = __allocate( n * m)

  for (i = 0,n) {
    for (j = 0,m) {
      b[i*m+j] = a[j*n+i];
    }
  }

  bar( n, m, b);
}
```

where `int[]` represents an integer array type that is stripped off all structural information. With this final lowering step we do not need (symbolic) shape information for arrays any more as all relevant uses of shape information have meanwhile been made explicit in the intermediate code.

Think of an efficient index calculation scheme for arrays with more than two dimensions. Have a look into *Horner* schemes, for instance.

Due date: Week 6

Milestone 12: Assembly Code Generation for Expressions and Statements

Implement a code generator that transforms your internal representation into a flat sequence of CiviC-VM assembly instructions, pseudo instructions and labels. For this milestone leave out the function call interface and restrict yourself to the body of the main function.

Due date: Week 7

Milestone 13: Assembly Code Generation for the Function Call Protocol

Extend your code generator to support the full function call protocol of the CiviC-VM, but leave out support for multiple modules.

Intensively test your code generator with the provided CiviC assembler and virtual machine.

Due date: Week 7

Milestone 14: Assembly Code Generation for Multi-Module Support

Extend your code generator to support separate compilation of CiviC modules.

Thoroughly test your compiler with your own CiviC programs as well as the test suite provided.

Due date: Week 8

Milestone 15: Optimised Assembly Code Generation

Extend your code generator such that it takes advantage of specialised instructions to reduce the size of the corresponding byte code and to improve program execution times.

Due date: Week 8