University of Amsterdam
Computer Systems Architecture
Dr Clemens Grelck

Compilerbouw
2013/2014
February 6, 2014

Project Milestones

CiviC Compiler

Project Mile Stone 1: Intermediate Representation (first shot)

Design an appropriate intermediate representation (abstract syntax tree) for your CiviC compiler by extending the file `ast.xml` of the compiler construction framework. Focus on the language core for now. If you need further attribute types, provide their implementation alongside. Submit the html representation of the abstract syntax tree, as automatically generated by the compiler construction framework.

Due date: February 14, 2014

Project Mile Stone 2: Intermediate Representation (final and complete)

Refine you 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. Submit the html representation of the abstract syntax tree, as automatically generated by the compiler construction framework.

Due date: February 21, 2014

Project Mile Stone 3: Lexicographic Analysis

Extend the lex-based lexicographic analyser (scanner) coming with the compiler construction framework to cover the full syntactic range of the CiviC language.

Due date: February 21, 2014

Project Mile Stone 4: Syntactic Analysis

Extend the yacc-based syntactic analyser (parser) 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 Milestone 2 in case the program is found to be well-formed. Otherwise, the parser shall produce a useful 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.

Test your scanner/parser in conjunction with the visualisation facilities developed for Milestone 5 using the example CiviC programs developed in the beginning of the course.

Due date: February 28, 2014

Project Mile Stone 5: Visualisation of Intermediate Representation

Extend the existing compiler traversal that prints the intermediate representation of a CiviC as the textual representation of a well-formed CiviC program to cover the full syntactic range of the CiviC language. Take care for suitable indentation according to the logical structure of the code.

Due date: February 28, 2014

Project Mile Stone 6: Separation of Declarations and Initialisations

CiviC like many other imperative programming languages supports variable declarations with immediate initialisation of the variable by the value of some expression, both for local and for global variables. While this syntactic sugar contributes to concise programs, it binds two compilation-wise very unrelated aspects together: declaration of a variable's type and the code that evaluates the expression and assigns the resulting value to the variable.

Implement a compiler pass that removes initialisation expressions from local variable declarations and inserts semantically equivalent but syntactically separate assignment statements into the sequence of statements in the function body. Furthermore, this desugaring compiler pass shall strip off all initialisation expressions from global variable declarations and create corresponding initialisation code in a to be created function with the name `__init`.

Last not least, remove the declaration part from `for`-loop induction variables and create corresponding local variable declarations on the level of the function definition the `for`-loop is situated in. Beware of nested `for`-loops using the same induction variable and occurrences of identically named variables outside the scope of the corresponding `for`-loop. In these cases, a systematic renaming of `for`-loop induction variables is recommended.

The array extension of CiviC requires special attention in this lowering step. A local variable declaration of the form `int[m,n] a;`, even in the absence of an initialisation expression, bundles declaration with code. Here, the declaration is that of a handle for a 2-dimensional array of integer numbers; the code is the actual allocation (and later de-allocation) of heap memory. As one can see, the size information on the array is only needed for the allocation statement, but not for the variable declaration. This observation resolves potential scoping problems when applying the above (simple) lowering transformation to array variables. Consider the following example CiviC code fragment:

```
int       m = 10;
int       n = 20;
int[m,n]  a;
```

Transforming the above code fragment naively into

```
int       m;
int       n;
int[m,n]  a;

m = 10;
n = 20;
```

would leave the variables `m` and `n` undefined at the declaration of array `a`, which would violate the scoping rules. The solution is to separate the symbolic declaration of an array's size from its actual allocation in memory and to introduce an explicit pseudo instruction for the latter part. This leads to the following intermediate code

```
int       m;
int       n;
int[m,n]  a;

m = 10;
n = 20;
a = __allocate( m, n);
```

Note the changed meaning of the array declaration in line 3. Previously, the array a came into existence when (symbolically) executing this line of code. Now, it merely represents the fact that a is an m×n array. For consistency, the same transformation should be applied to function definitions, function declarations and function applications. For example, the function definition

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

should be transformed into

```
void foo( int m, int n, int[m,n] a)
{
  bar( 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 applications. This, however, creates a challenge with respect to error reporting. A dimensionality mismatch between function definition and function application for some parameter/argument (i.e. a type system issue) may expose itself as a context inconsistency in the number of arguments. Of course, a consistency error caused by dimensionality mismatch must be distinguished from one that was present in the original source code by different error messages. This could, for instance, be achieved by tagging the added parameters and arguments accordingly.

Note further that the notations used above, e.g. the allocation pseudo statement, 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. Note further that this transformation of array variable declarations nicely fits the transformation of global variable declarations.

Due date: March 4, 2014

Project Mile Stone 7: Context Analysis

During context analysis each applied occurrence of an identifier, i.e. variables in expressions, variables in left hand side positions of assignments and function names in function applications, is associated with its corresponding definition or declaration. We suggest to eliminate the character string representation in these nodes of the abstract syntax tree and instead to add a pointer to the corresponding function definition, variable declaration or parameter declaration as a new (temporary, mandatory) attribute. A proper error message must be produced if the necessary declaration of a used identifier is missing. The compilation process shall nevertheless continue in the presence of context errors in order to report as many errors as possible in a single compiler run.

In the presence of nested function definitions in CiviC, the scoping rules must be obeyed and preserved. In addition to a reference to the point of declaration we suggest an appropriate representation of the relative difference in scoping levels between usage and declaration of an identifier.

Due date: March 7, 2014

Project Mile Stone 8: Type Checking

Type checking ensures that all operators and functions are applied to arguments of correct type. Type checking naturally falls into two separate tasks: type inference for expressions and type matching for assignments and function applications. 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 applications 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 useful error messages. The type checker should report multiple errors prior to 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 functions and operators.

Due date: March 12, 2014

Project Mile Stone 9: Compiling Conjunction and Disjunction

Add a compiler pass to your CiviC compiler that systematically replaces all occurrences of logic conjunction and disjunction operators by semantically equivalent code that makes use of conditional branching and, thus, ensures the short circuit evaluation of Boolean binary operators that semantically differs from the evaluation of all other binary operators.

Due date: March 12, 2014

Project Mile Stone 10: Array Dimension Reduction

The targeted CiviC Virtual Machine only supports vectors instead of multi-dimensional arrays (just as C or Java). Thus, 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 example illustrates this further lowering step:

```
void foo( 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 no longer need (symbolic) shape information for arrays as all relevant uses of shape information have been made explicit in the intermediate code.

Due date: March 18, 2014

Project Mile Stone 11: 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: March 21, 2014

Project Mile Stone 12: 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.

Due date: March 26, 2014

Project Mile Stone 13: Assembly Code Generation for Multi-Module Support

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

Thoroughly test your compiler with your CiviC programs testsuite and the CiviC assembler and CiviC-VM implementations provided.

Due date: March 31, 2014

Project Mile Stone 13: Optimisation

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: March 31, 2014

Project submission due date: April 1, 2014