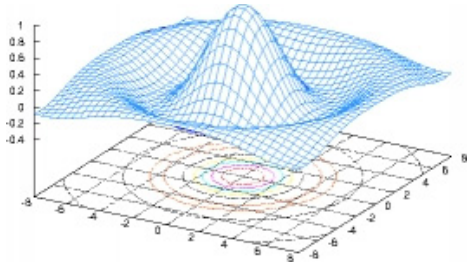


Department of System and Control Theory
Faculty of Computer Science and Automation
Technical University Ilmenau
www.tu-ilmenau.de

GNU Octave Project
Department of Chemical Engineering
University of Wisconsin
www.octave.org

Student research project

The Octave interpreter and possibilities of using it to implement an Octave to C++ compiler

Date:	26. May 2004
Author:	Jens Rücknagel born on 22.12.1978 in Pößneck
Study Course:	Computer science
Student ID:	29031
Matriculation Year:	99
Supervisor:	Univ. Prof. Dr. H. Salzwedel

Please send comments to ruena@web.de

Version 1.0

Contents

0.Introduction

Comparably small efforts are needed to implement flexible reusable artifacts of the information technology. While the performance of hardware components doubles every two years. This led to the building and wide use of complex computerized systems of which the correct behavior is satisfyingly proven on component level, but can not be guaranteed for the whole system.

Todays development processes fail because the gap between problems and tools grew to big to proof automatically or manually that a system behaves as desired. Right now there are no development processes sufficiently addressing the problem.

One approach is putting the whole development process on a more abstract level by increasing the abstraction level of its tools. Many causes of complexity are avoided by stepping from an all purpose programming language to a domain language to implement functionality with.

Octave is a free (GPL) domain language for numerical problems. One reason why many projects still use C++ is its lack of performance. One small aid in moving to a more abstract level is developing a compiler for the free numerical language Octave.

0.1.About this document

This document is the final paper of my three month student research project. I put much effort in making it usable beyond my academic carrier.

My original aim was to implement an Octave to C++ code generator reusing parts of the interpreter. It was quiet difficult for me to understand the Octave internals. Thats why I used half of the paper to write a documentation on the Octave interpreter. I hereby encourage the Octave community to replicate or modify this document in parts or complete.

There are many examples in the text. All copies of Octave source code and of source code of the Octave to C++ compiler are simplified to focus on the issues of the text.

Please do not mind the bad picture quality. Umbrello UML modeleer – an excellent free UML tool still has trouble with eps output.

1.The Octave language

The first chapter describes the Octave language. In fact the hardest work in building a compiler was to get familiar with the implementation of the Octave interpreter.

1.1.Acknowledgments

Octave is free software (GPL). It was developed by John W. Eaton and many others. In this paper I will call them the Octave community.

In my personal opinion the Octave community did a great job developing Octave, though this document points out the limits of the Octave interpreter. Building an Octave compiler reusing the interpreter was not a goal in developing Octave. The fact that it is possible, only shows the high quality of its design.

I also want to thank the Octave community for answering all my questions.

1.2.Language features - overview

Octave is a high-level interpreter language, primarily intended for numerical and matrix computations. It is mostly compatible with Matlab. The strength of Matlab and Octave is implementing a short syntax for matrix operations. It is in fact almost as easy readable and writable as the mathematical notation.

Octave is a procedural[1] language, therefore it is extensible via user-defined functions written in the language itself. In addition Octave functions may be implemented in C++ (§1.6).

Octave supports the imperative structured programming paradigm: statements are executed in the order they appear in source code, except a loop or a conditional statement is encountered. Statements may be grouped to functions.

Variables can be defined, read and written anywhere in the source code. Variables are typed dynamically, this means a variable holding a float may be assigned a string without redefinition.

For an example on how the use Octave see §3.5.

1 Sometimes Octave is called functional language, but in the field of computer languages a functional language is something totally different, but very interesting see [APPEL98].

1.3.The Octave distribution

The Octave distribution can be downloaded from www.octave.org. This is where all information about Octave is collected. The site is operated mainly by John W. Eaton.

Octave is developed to be as platform independent as possible. Anyway Octave is usually used on a x86-Linux. So on this platform seldom incompatibilities are encountered. The Octave community also supports Windows (Cygwin), Mac OS X and Solaris. On many other platforms Octave will probably compile. The make process needs the GNU tools.

The Octave interpreter is implemented in C++. Octave is usually compiled using GCC. The interpreter uses the own LibOctave (which is well documented and may be used without Octave). LibOctave encapsulates diverse numerical libraries such as BLAS. Most are included in the Octave distribution. Big parts of the numerical library are implemented in Fortran.

This paper focuses on building an Octave compiler, using technologies of the interpreter. So numerical issues are not discussed here.

1.4.Dynamic typing – octave_value

This chapter describes the implementation of the data type octave_value. All variables the user defines are represented by an octave_value. Octave_value and its descendants implement the dynamic typing system of Octave.

Octave_value is the class, that interpreter uses to wrap all values of a program in. It is not needed for refactoring .m files as C++ functions. In strong typed languages like C++ the type of an object (variable or function) is set at the definition and stays till the object is destroyed. In Octave the type may alter during the lifetime of an object.

```
octave:1> a = char(a=42)
```

It may even depend on user inputs.

```
i=input( "Number: " );
a = [1,2];
if i == 1
        a = "string";
else
        a = 10;
endif
```

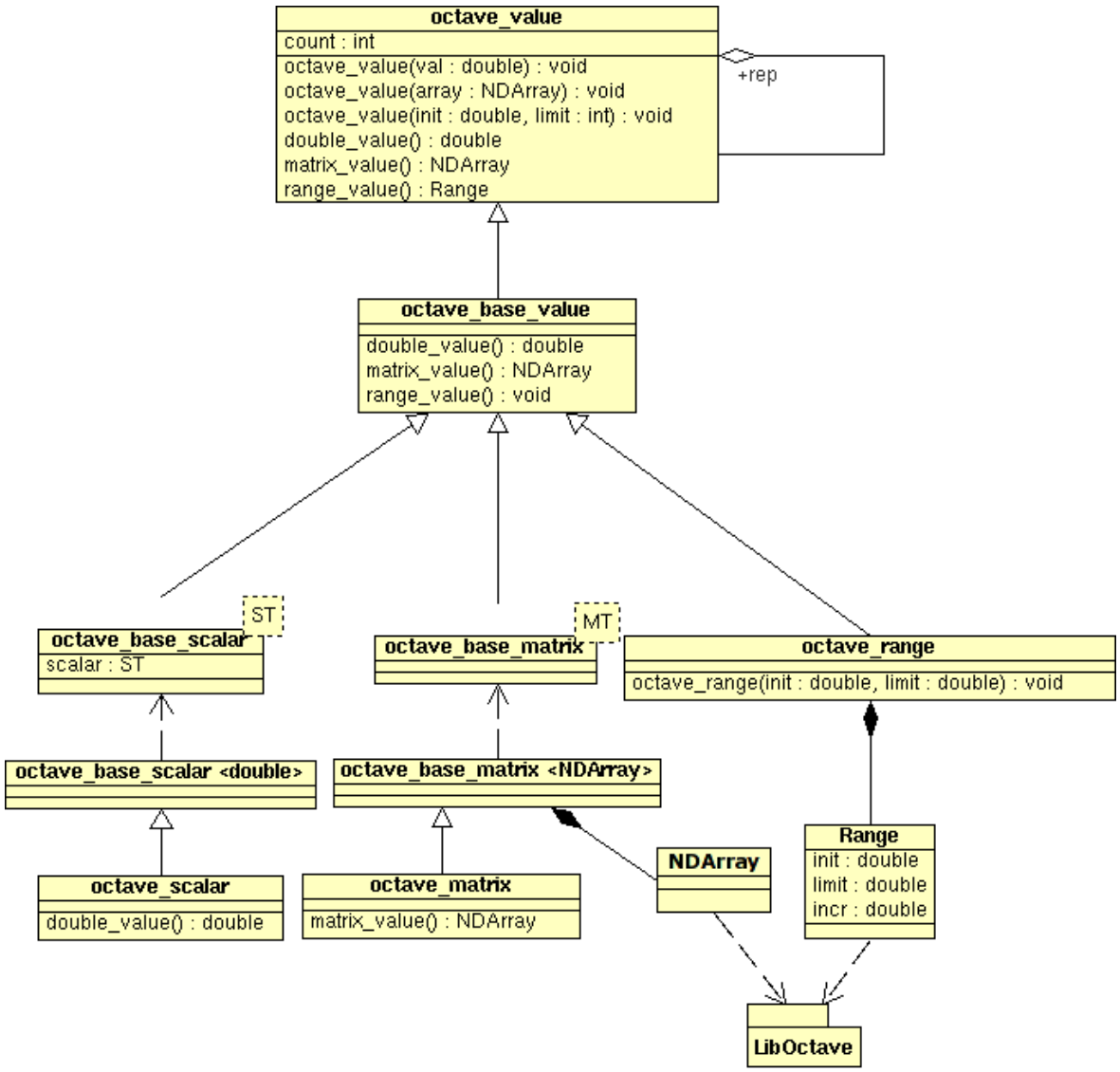
Functions may even have different return types depending on input values or even random.

```
octave:1> typeinfo( sqrt( 1 ) )
ans = scalar
octave:2> typeinfo( sqrt( -1 ) )
ans = complex scalar
```

One might say: the type depends on the value.

So an Octave interpreter or a runtime environment of a compiler needs to deal with variables on a more abstract level.

Here is an excerpt of the octave_value class hierarchy. Please do not mind the complex diagram – I will explain all parts.

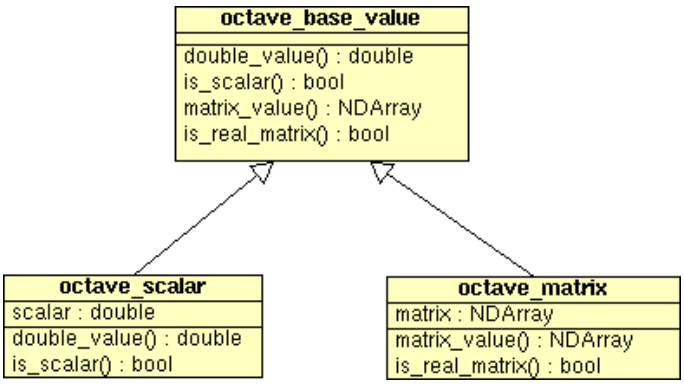


1.4.1.Wrapper for all types

Some dynamically typed languages save values in the most flexible data type they support. For example Tcl saves every value as a string. So numerical operations in Tcl have to convert their operands from string to float. Of course this is not always possible and is always slow. It would be inefficient for a numerical language.

That is why Octave uses a different approach. It stores every value as its actual type. A wrapper class is defined for every type Octave supports. So a scalar of a real number is wrapped in octave_scalar, matrices of real values are wrapped in octave_matrix, strings in octave_char_matrix_str.

To make it possible for the interpreter to handle all these different types the same way, they all use the same interface – they inherit from the same class: octave_base_value. In the simplified diagram octave_base_value and two of its descendants are shown:



There are distinct access functions for each type. The functions `bool is_...()` are used to test the type at runtime.

Octave_base_value implements all functions for all types. It provides the "Wrong Type" error messages:

```
virtual double
octave_base_value::double_value () {
        err_out << "double_value()"
                << ": Wrong Type ";
        return NAN_VALUE;
}
```

or just answer `false` to all `bool is_...()` questions:

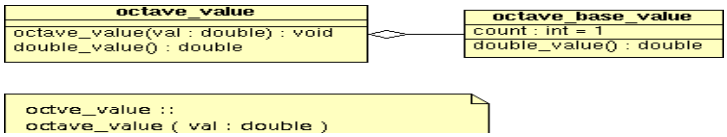
```
virtual bool
octave_base_value::is_scalar () {
        return false;
}
```

The descendants of octave_base_value only reimplement the access functions concerning themselves, as shown in the diagram above. Mind that in UML member functions use dynamic binding (`virtual`) by default.

```
virtual double
octave_scalar::double_value () {
        return val;
}
```

1.4.2.Reference counting

Octave_value and its descendants implement reference counting, as described in [STR00]. Therefore octave_value has two attributes `octave_value* rep` and `int count`. One may think of it as this:



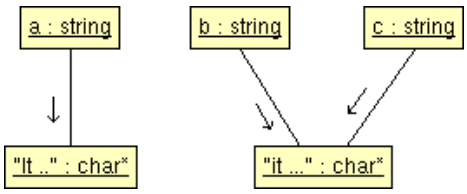
Octave_value resembles the reference and octave_base_value holds the actual value. So the access function `octave_value::double_value()` simply delegates the work to `octave_base_value`:

```
virtual double
octave_value :: double_value () {
        return rep -> double_value ();
}
```

Reference counting is used to prevent copying of large objects as often as possible. Think of a long character string `a` assigned to `b` and `c`. Copying this string is only necessary if `a`, `b` or `c` is altered.

```
string a = "it takes time to copy this many chars";
string b = a;
string c = a;
a[0] = "I";
```

A naive implementation of `string` would have made at least two copies of `"it..."`. Using reference counting only the last line forces one copy, because it changes `a`. So at the end this little example looks in memory like this:



1.4.2.1.Implementation of reference counting

Octave_value holds a pointer `ref` to octave_base_value. `Octave_base_value::count` counts the number of references to this one Octave_base_value. So the constructor of octave_value initializes `rep` with a new octave_base_value (or here with the descendant octave_scalar) and then sets count of the new created octave_base_value to `1`:

```
octave_value :: octave_value ( val : double )
        :rep( new octave_scalar( val ) ) {
        rep -> count = 1;
}
```

It follows that octave_value needs to know all of its descendants, so it is able to call their constructors. This way all these constructors make octave_value its own factory.

If an octave_value is supposed to copy itself, it just creates a new reference to the same value. The copy constructor of octave_value is responsible for this (the copy assignment operator is similar):

```
octave_value :: octave_value (const octave_value& a) {
        rep = a.rep;
        rep->count++;
}
```

If a value is changed, it needs to be copied. Every function which changes an octave_value first calls make_unique.

```
void octave_value :: make_unique (void) {
        if (rep->count > 1) {
                --rep->count;
                rep = rep->clone ();
                rep->count = 1;
        }
}
```

And finally the destructor of octave_value just releases one reference. If there is no reference left the octave_base_value is destroyed:

```
octave_value::~octave_value (void) {
        if (rep && --rep->count == 0) {
                delete rep;
                rep = 0;
        }
}
```

1.4.3.More than reference counting – proxy

Reference counting is often used in smart pointers. Smart pointer allow access to the value they point at. Octave_value does not allow any direct access at all. Octave_base_value is totally hidden. The user of the class hierarchy does not notice octave_base_value. Octave_value already implements all access functions needed to deal with all types of stored values.

Octave_value is a proxy class [GAMA95] for all descendants of octave_base_value. Lets take a look at the interface of octave_value:

```
//Constructors for octave_scalar
octave_value (int i);
octave_value (double d);

//Constructors for octave_matrix
octave_value (const NDArray &nda);
octave_value (const ComplexNDArray &cnda);
```

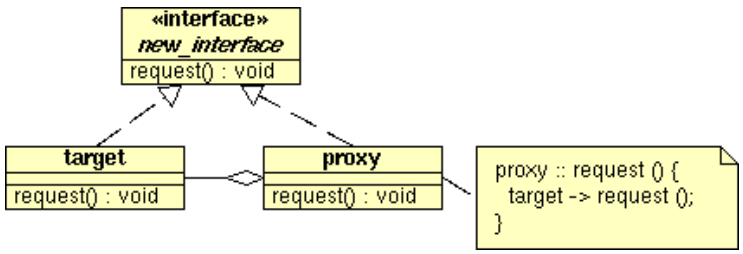
```
//Constructors for octave_range
octave_value (double base, double limit);
octave_value (const Range &r);

//Run time type information
virtual bool is_real_scalar ( ) const;
virtual bool is_real_matrix ( ) const;
virtual bool is_range ( ) const;

//Access functions for diverse types
virtual int int_value ( ) const;
virtual double double_value ( ) const;
virtual NDArray array_value ( ) const;
virtual ComplexMatrix complex_matrix_value ( ) const;
virtual Range range_value ( ) const;

//One of many helper functions
virtual octave_value convert_to_str ( ) const
```

All these functions do is to delegate the work along their `rep` pointers:

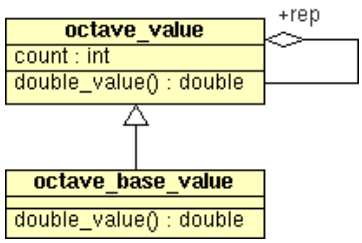
```
virtual double double_value( ) const {
        return rep->double_value ( );
}
```

1.4.3.1.Placing the proxy into the class hierarchy

In [GAMA95] proxy and target are descendants of the same interface class:



In contrast octave_value is both, base class and proxy, as you can see here:



This spares the definition of a separate interface. But every descendant of octave_value has the unused attributes `octave_value* ref` and `int count`.

As seen in the class diagram above, the reference counting mechanism is flatten. Octave_value counts how often it is referenced by octave_values. The constructors of octave_value need to make sure that octave_value references a descendant of octave_base_value. For the arising problems see §1.4.4.1.

1.4.4.Using the octave_value class hierarchy

Here I explain details on how to use the octave_value class hierarchy. The most important operations on octave_value are creating one and destroying one without causing segmentation faults.

1.4.4.1.Creating octave_values

Create an octave_value with:

```
octave_value a(40.7);
```

Actually this creates an octave_scalar and an octave_value referencing it (as described above). To create octave_values of other types, call another constructor.

For example `octave_value (const Matrix& m)`:

```
Matrix m(2,2);                    //create a 2x2 Matrix
m(0,0)=1;  m(0,1)=2;
m(1,0)=3;  m(1,1)=4;
octave_value om (m);
```

But **do not** do anyone of these:

```
octave_value f (octave_scalar (1) );    // ERROR

// ERROR, equivalent to above
octave_value g = octave_scalar (1);     // ERROR

octave_value h (1);                     // FINE
h = octave_scalar (1);                  // ERROR
```

The C++ compiler does not announce an error but Octave **crashes**. Because the copy constructor (or the assignment operator) of octave_value is called with an octave_scalar.

```
octave_value :: octave_value (const octave_value& a) {
        rep = a.rep;
        rep->count++;
}
```

In an octave_value `rep` and `count` are put in a union. `a.rep` on the octave_scalar `a` interprets the bit pattern of `count` as pointer, which is not valid. Dereferencing the pointer in the next line causes the crash. Fortunately the memory management detects it in most cases at runtime.

Summing up: the compile time type checking cannot deduce this error because of the flattened proxy pattern (§1.4.3.1). The runtime system cannot check class mismatch because of the union. John W. Eaton told me, that he does not know the reason any more, it is probably a legacy. Maybe this will be changed soon.

Simply the programmer should never call a function of a descendant of octave_value!

1.4.4.2. Type of octave_values

Use the `is...()` and `type_name()` functions to determine the type of the octave_value. Octave_scalar tells the `type_name()` "scalar". Table1 (§5.1) shows all classes which implement `type_name()` or are octave_values. Most of them actually are octave_values.

1.4.4.3. Getting values out of an octave_value

To get values out of an octave_value use one of the `..._value()` members. For example:

```
virtual double octave_value :: double_value ();
```

to get a double. Of course this only works if there is a double stored in it.

```
octave_value a(1);
std::cout << a.type_name();
double b = a.double_value();
```

1.4.4.4.Operations supported by octave_value

Octave_value implements many further operations as member functions:

```
const octave_value&
octave_value::assign(assign_op,const octave_value &rhs)
...
```

or as friend functions:

```
octave_value do_unary_op (unary_op op,
                          const octave_value &a)

octave_value do_binary_op (binary_op op,
                          const octave_value &a,
                          const octave_value &b)
...
```

While `assign_op`, `unary_op` and `binary_op` are enums defined in ov.h. The descendants of octave_value implement the usual C++ operators (+, ++, -, --, etc) which fit to the class. Take a look in [OCTSRC] to find many examples.

1.4.4.5.Getting help on octave_value

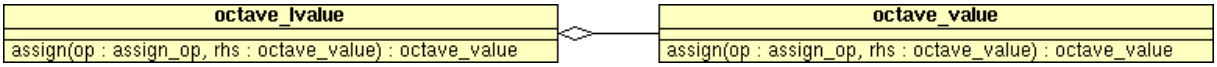
Look into the source ov.h, ov.cc and all files ov-*.h and ov-*.cc [OCTSRC]. To make reading source code easier you may use doxygen to generate a source code documentation. This generated documentation is a hyper linked list with interface definitions of all classes. It is not much but it is very useful.

Also read the diverse information on octave.org, the wiki and the mailing list archive.

1.5.Octave_lvalue

The left side of an assignment is called l-value. L-values identify a space in memory an object can be assigned to. For a full discussion on l-values see [AHO86] or any other book on compiler construction.

The Octave interpreter needs a facility to construct l-values. L-values are references to octave_values. An l-value in Octave may be just a variable in the .m source, or may be a compound expression such as: `M(1,3)`. But it always evaluates to a reference to an octave_value.



1.6.Dynamically linked functions vs. stand alone applications

Once you have developed some C++ functions for Octave, you can decide whether you want to build them to dynamically linked functions or to stand alone applications.

It is not recommended, to use octave_value in C++ code for numerical calculations. The abstraction mechanisms slow down execution compared to using the actual types. In this sense the examples below are bad programming style. Anyway for building an Octave compiler it is necessary to deal with octave_value a lot.

1.6.1.Dynamically linked function - .oct-file

This example is implemented as a small dynamically linked (dl) function. If you develop for Octave I highly recommend developing your code first as dl-function.[2] Because you only have to compile your module, not Octave.

Every linkable object (which is almost every object) within Octave is accessible from a dl-function.

```
#include <octave/oct.h>
#include <octave/oct-lvalue.h>
```

2 Dynamically linked functions are not supported on all platforms. Octave must be compiled with special options depending on your platform. See Installing Octave at [OCTDOC].

```
DEFUN_DLD(ov_example, args, , "ov_example") {
        octave_value a(40.7);
        octave_value b(1.3);
        octave_lvalue lhs( &b );
        lhs.assign( octave_value::op_add_eq, a);
        b.print(octave_stdout);
        return octave_value(b);
}
```

Compile with:

```
$ mkoctfile -v ov_example.cc -o ov_example.oct
```

Run from the Octave prompt by entering `ov_example`. It should yield 42. For further information read Dynamically Linked Functions in [OCTDOC].

1.6.2.Stand alone application

In stand alone applications octave_value is usable too. That way it is possible to build applications reusing big parts of the interpreter. Building a stand alone application from a dl-function is not directly possible. But it can easily be transformed by rewriting it to a main function and by adding some initializations:

```
#include <octave/oct.h>
#include <octave/oct-lvalue.h>
#include <octave/ops.h>

int main() {
        install_types ();
        // Important: types are installed before ops
        install_ops ();
```

```
        octave_value a(40.7);
        octave_value b(1.3);
        octave_lvalue lhs( &b );
        lhs.assign( octave_value::op_add_eq, a);
        b.print(octave_stdout);
        return 0;
}
```

Compile the file using the `-link-stand-alone` parameter.

```
$ mkoctfile -v --link-stand-alone ov_example.cc -o
ov_example
```

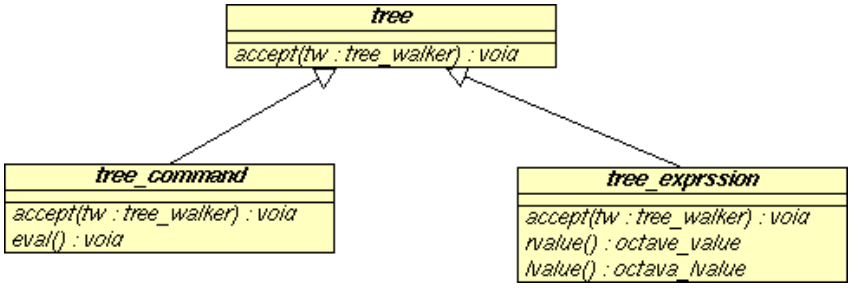
1.7.Parse tree and evaluator

Octave uses a Lex/Yacc parser to parse .m-files and command line input into a parse tree. Each type of parse tree node is represented by an own class. The parse tree classes offer member functions for evaluation.

User defined functions are parsed when entered at the command line and evaluated when called. So they only have to be parsed once. .m-functions are parsed when they are called the first time. On further calls the time stamp of the .m-file is checked to avoid useless reloads.

1.7.1.The parse tree classes

Nodes of the parse tree are descendants of `tree` in pt.h. Their names start with "tree-" and are declared in the files pt-*.h. `Tree` has two derived classes: `tree_command` and `tree_expression`. They offer the evaluation functions `eval()` (respectively `rvalue()` and `lvalue()` for expressions).



Most node classes have references to other node classes to represent the parse tree. This im-

plements a composite pattern as in [GAMA95].

1.7.2.Tree walker

All tree classes accept a tree_walker. The tree_walker implements a visitor pattern as explained in [GAMA95]. That way the user can specify an own tree_walker class to traverse the parse tree. Every tree node class implements a member function `accept( tree_walker& )` which calls the corresponding tree walker function.

```
class tree_simple_assignment : public tree_expression {
public:
        void accept (tree_walker& tw) {
                tw.visit_simple_assignment (*this);
        }
...
}
```

`tree_simple_assignment` has two subtrees `lhs` (left hand side) and `rhs`. So an implementation of `visit_simple_assignment(tree_simple_assignment&)` will probably call `lhs->accept(*this)` and `rhs->accept(*this)`.

Octave uses tree_walker to implement brake points and to print the parse tree as Octave code . The compiler described below is implemented as tree_walker.

1.7.3.Evaluator

Due to dynamic typing and the incremental development of Octave the evaluator functions are rather complex and in some cases scattered between the parse tree and octave_value. To evaluate something simple as a matrix with a parameter, Octave needs three pases through the matrix.

```
b="a";
a=[b,1;2,3];
```

Because depending on the type of the parameter and the types of the constants Octave has to construct a different type of matrix.

2.Compiling Octave: Analysis

My aim was to build an Octave to C++ compiler prototype within one month from scratch. To do my job I had to decide which techniques I want to use.

2.1.Structure of a compiler

The purpose of this chapter is mainly to introduce the vocabulary of the next chapter. For a full discussion I personal recommend the classic [AHO86].

Compilers are structured in the following phases:

- Lexical analysis
- Syntax analysis
- Semantic analysis
- Generation of intermediate representation
- Optimization
- Code generation or evaluation

All phases need access to a symbol table.

2.1.1.Symbol table

The symbol table is the main data structure of a compiler. All phases of a compiler usually need access to the symbol table. In it all identifiers of variables or functions are saved. Usually attributes are saved together with the identifier. The most important attributes are:

- Attributes for compilation
  - name
  - type
  - address in memory
- Attributes for error handling
  - position in source code
  - source code

Octave has its own symbol table to aid interpretation. Variables and function names are saved there. It is build up while evaluating the parse tree. So it is of no use for a compiler. The compiler has to translate code without evaluating it.

2.1.2.Lexical analysis and syntax analysis

The lexical analysis takes the input character stream and groups them to tokens using regular expressions. During syntax analysis the resulting token stream is tested against grammatical rules. While testing most syntax analyzers construct a parse tree (concrete syntax tree [AHO86]) just by the way.

The lexical analysis and the syntax analysis are done by a flex and bison parser. This parser can be reused for a compiler, though the syntax tree is optimized for interpretation.

2.1.3.Semantic analysis

Semantic analysis collects typing informations and checks them: it tests whether the operands of an operation are valid according to the specification of the language.

Semantic analysis in Octave is done while evaluating the source code. The type checking facilities are placed in the dynamic typing class octave_value (see §1.4). Operations on octave_value simply return an error if an operation is not specified. Octave does not have an explicit specification, nor an explicit type checking system.

2.1.4.Generation of intermediate representation

Usually a compiler or interpreter creates an intermediate representation of the source program. This internal representation is totally equivalent to the source code. Common internal representations are 3-address-machine and parse tree (more accurate: decorated parse tree [AHO86]).

Octave uses a decorated parse tree as intermediate representation. This parse tree is constructed while syntax analysis (syntax driven translation [AHO86]). The interpreter evaluates this parse tree. The compiler explained below takes this parse tree as input.

2.1.5.Optimization

An optimizer tries to transform the intermediate representation so that compiled code executes faster, uses less memory or is smaller. Often these aims are in conflict. Optimization works by replacing one piece of intermediate code by another which implements the same functionality in a more efficient way.

A simple approach would be to find duplicate subtrees. Octave does no optimization in the in-

terpreter.

2.1.6.Code generation or evaluation

In the last phase the intermediate representation is translated to target code or interpreted.

Each node of the Octave parse tree has its evaluation function. Evaluation is invoked by calling the evaluation function of the first node. The nodes are responsible for invoking evaluation of child nodes by themselves.

2.2.Planning the compiler

As said above: due to the time limit on student research projects my aim was to implement a first compiler as quick as possible. Therefore I reused as much as possible of the interpreter.

2.2.1.Dynamically linked functions

The Octave to C++ compiler needs access to the whole Octave source code. Nevertheless it should not interfere with the Octave project. So I did not change Octave at all.

I decided to put the code generator into a dynamically linked function. It can be invoked from the Octave command line and can access anything within Octave. A nice side effect: I only had to recompile my project on changes, which only takes some seconds. For more informations on dynamically linked functions see §1.6.

2.2.2.Compilation entities

Octave is a procedural language, so a compiler prototype should show how to compile Octave functions. The compiler needs a parameter: the name of the function it is supposed to compile. Next it needs to gain access to the parse tree of that function, build a C++ interface from the Octave interface and than run the compiler on the body.

2.2.2.1.Function interface

An Octave function can be translated into a C++ function with the same name. Octave functions may have multiple inputs and multiple output parameters. An input parameter can be translated to a value parameter. Output parameters and parameters used for both in- and output can be translated to reference parameters.

2.2.3.Reuse of the parser and the tree walker

I decided to reuse the parser and do code generation on basis of the parse tree. The parse tree accepts a tree walker. So a tree walker class which generates C++ code is to be build.

The tree_print_code class form pt-pr-code.h and pt-pr-code.cc [OCTSRC] implements a pretty printer. This class is a good example for a tree walker which generates code. A copy of it is used as base of the development of an own tree walker. All nodes that do not build C++ code yet, at least build Octave code. Keeping them till they are rewritten accelerates identifying the functionality of parse tree nodes.

The tree_walker functions do not have a return value nor parameters. If values have to be exchanged with sub nodes, other mechanisms must be used. The symbol table transports information between nodes. It will be implemented as attribute of the tree walker class.

2.2.4.No reuse of the evaluator

The evaluator is implemented as a set of member functions of the parse tree. To reuse these functions the whole node must be rebuild. Afterwards the compiled code would be slower than the interpreter.

Parts of the interpreter are build into the dynamic typing class octave_value. By reusing octave_value these functions can be reused (§2.2.6.1).

2.2.5.Control flow

Control flow governs the order in which statements are executed. Octave's control flow concepts are a subset of the C++ concepts. Octave offers: if, loops, brake, continue and rudimentary exception handling (see [OCTDOC]). Octave has no data driven control flow concept like dynamic binding. So control flow can be translated into C++ control flow.

2.2.6.Typing – Memory management

One of the most important questions is how the compiled program handles variables: How it maps Octave variables to C++ variables. Octave variables occur in the .m-source code and may occur as intermediate values while evaluating the source code. The examples in §1.4 state that the type of a variable is certain only after evaluating its assignment. And it may change later.

Closely connected to this issue are the questions how the compiler deals with variables during compilation (name handling, symbol table), and whether and how semantic errors are reported (type errors).

2.2.6.1.Using dynamic typing

The approach closest to the interpreter is to use the dynamic typing facilities of the interpreter. For every variable in the .m-source the compiler creates an octave_value in the C++ target. The symbol table is confined to keep track of defined names. If one is not yet defined, a definition is included in the target code.

Another advantage comes from the fact that the interpreters arithmetic functions all work on octave_values or are implemented as methods of octave_value.

And finally variable names in Octave can be translated to the same names in C++.

2.2.6.2.Using type inference

As stated above the type of a variable is known after evaluating the assignment, but not at compile time. But in many cases the type of intermediate variables or even of named variables can be determined without evaluating.

Many operations take only operands of certain types and evaluate only to certain types. So every operation has a list of input types for each operand and one list of output types. This specification is implicitly defined by the implementation of the interpreter.

The possible types of a variable can be deduced button up or top down. Top down means that the possible types depend on the types accepted by an operation with that variable. Button up means that possible types of a variable depend on the types of its defining expression.

Mind the operator overloading concept used in other languages is necessary for operators like +. But it is not sufficient for sqrt(), of which the return type depends on the input **value**. Overloading depends on the input type not value.

Compiler theory offers a tool for typing values implicitly. This is type inference as used in polymorphic static typed functional languages[3] like Haskell and ML. In those languages types

3 Octave is a procedural language, in the field of computer languages a functional language is something totally different, but very interesting see [APPEL98].

are grouped to classes (not object oriented classes). For example int, float and complex can be grouped to the class scalar. Classes can be grouped to new classes. This way a type hierarchy is formed. Classes are defined by the operations on the elements of the class. For example on all scalar types the operations +,-, *, /, sqrt are defined.

For further discussions on type inference see [APPEL98]. For further readings on functional programing see [BIRD98]. It is subject to further research if and how type inference for Octave can be implemented. I did not implement a type inference system due to its complexity.

2.2.7.Helper functions

Translating a node of the parse tree to C++ results in many C++ statements. The compiler gets more complicated if it has to generate many lines of code for one node. It would be better if the actual C++ statements are separated from the tree walker.

The C++ translation of one node can be grouped to an inline function. These helper functions are put into a header file which is included into the compiled code. The compiler just generates function calls to these helper functions.

2.2.8.Compiling Expressions

2.2.8.1.Temporary variables

Expressions with more than one operator can be translated into a list of C++ statements of the form:

```
t = op ( p1, p2, ..., pn );
```

Where `t`, p_1 ... p_n are parameters of the original expression or are temporary variables introduced by the compiler. And where `op` is a function with `n` parameters.

The Octave expression `a=(a+1)*b` may be translated to:

```
octave_value t1 = scalar( 1 );
octave_value t2 = add( a, t1);
octave_value t3 = t2 * b;
octave_lvalue t4 = a;
octave_value t5 = assign( t4 , t3 );
```

Or with reuse of temporary variables:

```
octave_value t1 = scalar( 1 );
t1 = add( a, t1);
t1 = t1 * b;
octave_lvalue t2 = a;
octave_value t1 = assign( t2 , t1 );
```

To implement this a name management for temporary variables is needed. And nodes in the parse tree need to communicate the temporary variables while compilation. If the compiler is implemented as tree walker, the functions of this tree walker have no return value. So they cannot easily communicate the temporary variables. Therefore I did not use temporary variables.

2.2.8.2.Expressions as C++ expressions

Octave expressions are similar to C++ expressions. It is tempting to translate Octave expressions to C++ expressions. The only difference is that in Octave variables can be defined within expressions. This problem can be solved by putting the declaration of the variable in front of the expression. Helper functions (inline) can be used for Octave operations.

The Octave expression `a=(a+1)*b` can be translated to:

```
assign( lvalue(a) , mul( add( a, scalar( 1 )) , b ))
```

This looks nice but becomes unreadable for big expressions. It also is a bit inflexible. The compiler is only allowed to generate function calls. If it wants to include a statement, it has to use the C++ "`,`" operator.

This approach seems to be a short version of using temporary variables (§2.2.8.1). It seems to be faster because it does not create all these temporaries. But that is a misconception. The C++ compiler has to build these temporaries anyway while compiling the expressions. For every temporary it has to call the constructor and the destructor. So the approach with the reused temporaries may be faster because there is no need to call so many constructors and destructors.

By the time I had to decide whether to use this approach I knew not much about Octave. So I

considered the restrictions combined with the uncertainty about Octave a design risk. I did not use this approach.

2.2.8.3. Stack machine

To remove the restriction on functions of the last example the call stack is handled by the program itself.

Expressions are evaluated in postfix order using a runtime stack. Octave operands and operations are replaced by helper functions which operate on a stack. The Octave expression `a=(a+1)*b` may be translated to:

```
push_lvalue ( octave_lvalue( & a ) );
push_rvalue ( a );
push_rvalue ( 1 );
add ();
push_rvalue ( b );
mul ();
assign ();
```

I chose this approach because it promised the fastest results.

2.3. Verification, validation and testing

Verification is the process of comparing two levels of system specification for proper correspondence (http://sun.soci.niu.edu/~rslade/secgloss.htm#verification).

Because there is no specification of the Octave language it is not possible to do proper verification of the compiler. The Octave documentation, Matlab and the Octave interpreter can be analyzed and a real specification of Octave can be created. The compiler can be checked against this specification.

Validation is the process of increasing the confidence that the outputs of a model conform to reality in the required range. In some cases, the model's output can be compared to data from historical sources or from an experiment conducted for validation. A model can never be completely validated (http://pespmc1.vub.ac.be/ASC/INdexASC.html).

Comparing the compiler to the interpreter can be achieved by running test cases through the

compiler and the interpreter and comparing the results.

Octave offers a test suite for its components. A compiler that evolves beyond a prove of concept needs to be checked automatically. Therefore test cases need to be designed and integrated into the Octave test suite.

While developing the compiler, the developer will create test cases anyway. Keeping them and managing them will assure, that once correct implemented cases will not become incorrect during further development. Additional test cases need to come also from testers, from bug reports and from the language specification.

Due to the fact, that this paper questions the compiler design itself, I did not invest time to find bugs in the compiler.

3.Implementation of a code generator

This chapter gives a description of the Octave to C++ compiler implemented according to the decisions made in the planning phase.

3.1.Structure description

The compiler is implemented in the files compile.cc, pt-emit-cplusplus.h, pt-emit-cplusplus.cc, o2c_helper.h and o2c_helper.cc . Additionally a Makefile is provided.

3.2.compile.cc

Compile.cc consists of one function. It is the main entrance point to the compiler. It is implemented as dynamically linked function `compile(function_name)` according to §2.2.1.

`Function_name` must be the name of a user defined function or an .m-file. Compile uses the usual Octave method to search for a function with the name `function_name`. If Octave has found the function, it accesses to the parse tree of that function.

Compile initializes the tree walker class pt-emit-cplusplus and sets the debug flag of the tree walker. It initializes the output stream for compiled code to octave_stdout. It also prints the needed header definitions.

Then it translates the Octave function header into a C++ function header according §2.2.2. An input parameter is translated to a value parameter. Output parameters and parameters used for both in- and output are translated to reference parameters. All parameters are entered into the symbol table as defined identifiers.

Most of the code in compile.cc is by J.D. Cole from the Octave community. After asking for hints, how to get and handle the parse tree, he sent me compile.cc, pt-emit-cplusplus.cc, and pt-emit-cplusplus.h. Compile.cc was implemented with some bugs which I fixed. Pt-emit-cplusplus.* was just a renamed copy of pt-pr-code.* from [OCTSRC]. It did not generate any C++ code at all.

In this example I define the empty function cover and show how function headers are translated:

```
octave:1> function c=cover(n) endfunction
octave:2> compile("cover")
#include <octave/oct.h>
#include <o2c_helper.h>

extern void
cover (octave_value n, octave_value& c)
{
}
```

In the following examples I will only show the compiled statements. I will remove function and header definitions from Octave and C++ code.

*3.3.pt-emit-cplusplus.**

In pt-emit-cplusplus.h the tree-walker class pt-emit-cplusplus is declared. Its member functions are defined in pt-emit-cplusplus.cc.

The pt-emit-cplusplus class is a renamed copy of pt-pr-code. The tree walker has a function for each node in the parse tree. Pt-pr-code just prints the Octave source of that node. I reimplemented one function after another. Functions not yet reimplemented still print Octave code.

Pt-emit-cplusplus holds the reference `os` to an output stream. All compiler output is put on `os`.

3.3.1.Debug flag

One of the first things added to pt-emit-cplusplus was the debug flag `bool debug_characters`. If the debug flag is set, each parse tree function prints its unique number and parenthesizes around its output. So the output can be associated with a node and a function.

3.3.2.Symbol table

Pt-emit-cplusplus holds a symbol table. The symbol table is accessible through the member function:

```
bool symbol_used (const std::string name,
                  const bool add = false);
```

If `add` is true this function checks the entry `name`, and, if not found, adds it.

The only aim of the symbol table is to find out, whether an identifier is already defined or whether a C++ definition is needed before the identifier can be used.

3.3.3.L-values

Some expressions (like identifiers) have to be translated different for l-values. When the tree walker translates a node of an assignment it has to tell to the left hand side to build an l-value on the stack. It does so by setting the attribute `pt-emit-cplusplus::lvalue` to true. The function `pt-emit-cplusplus::visit_identifier()` checks the l-value attribute.

```
void tree_emit_cplusplus::
visit_identifier (tree_identifier& id) {
        std::string nm = id.name ();
        if (lvalue) {

                if ( !symbol_used (id.name(),true) )
                        os << "octave_value "
                                << id.name() << ";/n";
                os << "o2c :: push_lvalue(& "
                        << id.name() << " );/n";

        } else  //not lvalue
                os << "o2c :: push_rvalue( "
                        << id.name() << " );/n";
}
```

3.4.o2c_helper.h

As already seen in the example above the compiler generates calls to functions in the namespace `o2c`. To enable the compiler to inline these functions, all are defined in o2c_helper.h. Only void end_build_matrix() is defined in o2c_helper.cc, because it is so big.

These functions make up the stack machine proposed in §2.2.8.3. Usually they pop some values off the stack, do some calculations and push results.

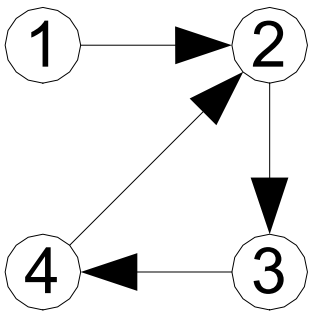
The stack is implemented as C++ STL stack. The class o2c_value enables pushing octave_lvalues and octave_values together on one stack. It is a replacement for:

```
union o2c_value {
        octave_value ov;                //ERROR
        octave_lvalue ol;               //ERROR
}
```

Because classes with self defined constructors are not allowed in unions [STR00].

3.5.Working example

Lets go through a somewhat realistic working example. Think of the directed graph shown below with its adjacency matrix A. A(i,j) ≠ 0 means there is a connection from i to j.



$$A = \begin{bmatrix} 1100 \\ 0110 \\ 0011 \\ 0101 \end{bmatrix}$$

The question: For all nodes k: Which nodes can be reached from k in n steps? To solve this just compute A^n. A very simple implementation is shown below:

```
function c = cover(n)
        c = 1;
        a = [1 1 0 0; 0 1 1 0; 0 0 1 1; 0 1 0 1]
        i = 1;
        while ( i++ <= n )
                c = c * a;
        endwhile
endfunction
```

Running it with `cover(4)` yields: $ans = \begin{bmatrix} 1564 \\ 0556 \\ 0655 \\ 0565 \end{bmatrix}$ A(i,j)≠0 means there is a path from i to j.

To compile this function enter compile("cover") at the Octave prompt. Here is the compiled code as produced by the compiler. I added line numbers to reference them later.

```
01  #include <octave/oct.h>
02  #include <o2c_helper.h>
03
04  extern void
05  cover (octave_value n, octave_value& c) {
06  o2c :: push_lvalue( octave_lvalue( & c ) );
07  o2c :: push_rvalue ( 1 );
08  o2c :: assign ( octave_value::op_asn_eq );
09  o2c :: empty_stack();
10  octave_value a;
11  o2c :: push_lvalue( octave_lvalue( & a ) );
12  o2c :: push_rvalue ( o2c :: matrix_constant( "# rows: 4\n#
columns: 4\n 1 1 0 0\n 0 1 1 0\n 0 0 1 1\n 1 0 0 1\n" ) );
13  o2c :: assign ( octave_value::op_asn_eq );
14  o2c :: print_result( "a =" );
15  o2c :: empty_stack();
16  octave_value i;
17  o2c :: push_lvalue( octave_lvalue( & i ) );
18  o2c :: push_rvalue ( 1 );
19  o2c :: assign ( octave_value::op_asn_eq );
20  o2c :: empty_stack();
21  o2c :: push_lvalue( octave_lvalue( & i ) );
22  o2c :: postfix_op ( octave_value::op_incr );
23  o2c :: push_rvalue( n );
24  o2c :: bin_op ( octave_value::op_le );
25  while ( o2c :: top_true() ) {
26    o2c :: push_lvalue( octave_lvalue( & c ) );
27    o2c :: push_rvalue( c );
28    o2c :: push_rvalue( a );
29    o2c :: bin_op ( octave_value::op_mul );
30    o2c :: assign ( octave_value::op_asn_eq );
31    o2c :: empty_stack();
32    o2c :: push_lvalue( octave_lvalue( & i ) );
33    o2c :: postfix_op ( octave_value::op_incr );
34    o2c :: push_rvalue( n );
35    o2c :: bin_op ( octave_value::op_le );
36  }
37  }
```

This code can be used in any program. A wrapper to make it a dynamically linked function may look like this.

```
DEFUN_DLD(cover, args, , "cover") {
    if(args.length() > 0 && args(0).is_real_scalar()) {
        octave_value a(2);
        octave_value n=args(0).double_value();
        cover (n,a);
        return octave_value(a);
    }
}
```

3.5.1.Function body

Lines 1 to 5 and line 37 are by compile.cc as shown in §3.2. The function body is a tree_statement_list and is therefore translated by visit_statement_list. Visit_statement_list walks its list of statements and calls visit_statement on each one of them.

From now on I assume that it is known that a node `tree_xxx` is translated by the tree walker function `visit_xxx(tree_xxx& node)`. I will use them both as synonyms.

3.5.2.Statements

Now the tree walker looks at the tree of the statement `c = 1;` (translation in lines 06-09). Visit_statement decides whether the statement is a command or an expression. To distinguish between the two, tree_statement has two pointers: `tree_command *cmd` and `tree_expression *expr`. The one not zero is valid.

3.5.3.Expressions

The example `c = 1` of course is an expression. The result of the expression will be on top of the stack after evaluating the expression at runtime. It has to be removed. So visit_statement puts an `o2c :: empty_stack();` behind the expression.

To keep track, I will print the current state of compilation of the example. Instead of compiling the not yet walked parse tree, it is printed as Octave code (*italic*) where its compilation belongs. The usual compiler output is printed **bold**. The current output of `c = 1;` would be:

```
    c = 1
09  o2c::empty_stack();
```

3.5.4.Simple_assignment

The `c = 1` is a simple assignment. Tree_simple_assignment is a subclass of tree_expression. Visit_simple_assignment first sets `pt_emit_cplusplus::lvalue` to true. Then invokes the building up of the l-value and sets `pt_emit_cplusplus::lvalue` to false again. After that the r-value is build up and finally the assignment call is put.

Octave has different assignment operators (=, +=,...). All are handled by tree_simple_assignment. The assignment operator is specified as parameter to `o2c::assign()`.

```
    c
    1
08  o2c :: assign( octave_value::op_asn_eq );
09  o2c :: empty_stack();
```

3.5.5.Identifiers

The left hand side `c` is a tree_identifier. Visit_identifier puts `push_rvalue` or `push_lvalue` depending on the variable `pt_emit_cplusplus::lvalue`. The explicit type conversion `octave_lvalue` is added to. `push_lvalue.` If an l-value name is not yet defined, a definition is put in front (as in line 10). But `c` is already defined in the function header.

```
06  o2c :: push_lvalue( octave_lvalue( & c ) );
    1
08  o2c :: assign( octave_value::op_asn_eq );
09  o2c :: empty_stack();
```

3.5.6.Constants

`1` is translated by visit_constant. Visit constant prints `o2c :: push_rvalue( octave_value v )`, where `v` is the C++ representation of the constant. Constants (literals) are embedded as octave_values in the parse tree. Tree_constant simply holds the octave_value. So for every type of octave_value a way to construct it from C++ literals is needed. Therefore every type of constant is treaded a bit different.

I implemented real and complex scalar, string, range and matrix. Other types of constants are not yet implemented.

3.5.6.1.Real scalar

The easiest is real scalar. A real scalar is put as integer or double literal in the C++ code. Octave_values constructor octave_value(double) is implicitly used by C++ to convert the literal to octave_value. So the literal `42.42` translates to:

```
o2c :: push_rvalue ( 42.42 );
```

3.5.6.2.String

Strings are treaded equally, except that in a string literal some characters have to be escaped with "\". The string `Hi there "\` translates to:

```
o2c :: push_rvalue (  "hi there \"\\"  );
```

3.5.6.3.Range

For ranges the appropriate constructor of octave_value is inserted. So `1:10:100` translates to

```
o2c :: push_rvalue ( octave_value ( 1 , 100 , 10 ) );
```

3.5.6.4.Complex Scalar

For complex scalars a temporary Complex value is created, so C++ can insert the appropriate constructor of octave_value.

```
o2c :: push_rvalue ( Complex (2,3) );
```

3.5.6.5.Matrix

Matrices are more complex. There is no constructor of octave_value or of a temporary, that can be called. The helper function `o2c :: matrix_constant( string )` builds a matrix out of a string. So the matrix `[1 1 0 0; 0 1 1 0; 0 0 1 1; 0 1 0 1]` translates to:

```
o2c :: push_rvalue ( o2c :: matrix_constant( "# rows:
4\n#  columns: 4\n 1 1 0 0\n 0 1 1 0\n 0 0 1 1\n 1 0 0
1\n" ) );
```

So finally `c = 1;` compiles to:

```
06  o2c :: push_lvalue( octave_lvalue( & c ) );
07  o2c :: push_rvalue ( 1 );
08  o2c :: assign ( octave_value::op_asn_eq );
09  o2c :: empty_stack();
```

Lets now continue the example “cover”.

3.5.7.Print result

After the line

```
a = [1 1 0 0; 0 1 1 0; 0 0 1 1; 0 1 0 1]
```

the trailing “`;`” is missing. This tells Octave to print the result of the last expression. So the compiler inserts the line:

```
14  o2c :: print_result( "a =" );
```

With the explanations given up to this point, it is no problem to understand the cover example up to line 20. I will now shorten my explanations.

3.5.8.While

In line 21 the evaluation of the while condition begins. An Octave while loop has this structure:

```
while ( condition )
        body
endwhile
```

Because *`condition`* is an expression, it may include variable declarations, and so its com-

piled version may not fit into a C++ expression. Thats why *condition* is duplicated and put in front of `while` and behind *body*. The *condition* is checked with `o2c::top_true ()`. The C++ code is this:

```
condition
while ( o2c :: top_true() ) {
        body
        condition
}
```

Because the `condition` might include an Octave variable declarations, it must not just be copied to the back, but must be translated twice.

Applied to the example “cover”:

```
    i++ <= n;
25  while ( o2c :: top_true() ) {
            c = c * a;
            i++ <= n;
36  }
```

3.5.9.Unary and binary operations

To do unary or binary operations on a stack machine is simple. The operands have to be pushed on the stack and than the operation has to be performed. The responsible tree walker functions are visit_prefix_expression, visit_postfix_expression and visit_binary_expression. The appropriate helper functions are bin_op, postfix_op, prefix_op. The operator is given as parameter to these functions.

The binary expression `c * a` translates to:

```
    c
    a
29  o2c :: bin_op ( octave_value::op_mul );
```

The complete “cover” example can now be understood. To explain the rest of the implemented features would go beyond the scope of this document. If needed they can be found in the source of the compiler.

3.6.Implemented features

My major aim was to implement loops and some matrix operations in the prototype as soon as possible. To get started I implemented assignments, identifiers and scalar constants first, later I added unary and binary operators. Table2 (§5.2) is a list of all implemented features with corresponding tree walker functions and helper functions.

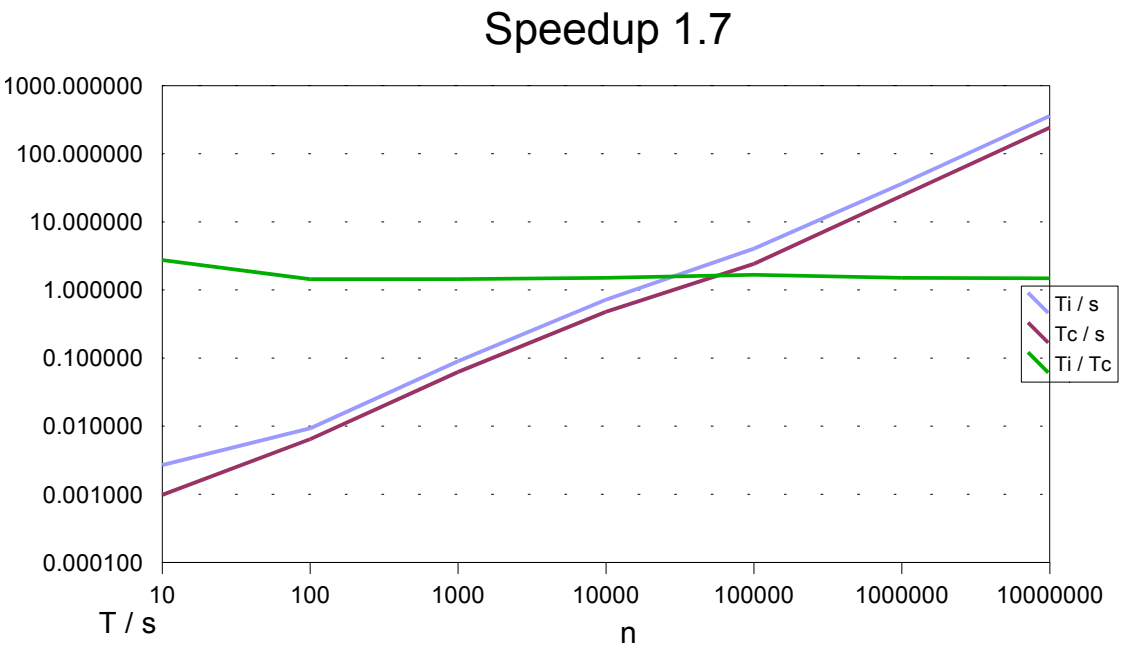
Table3 (§5.3) shows a list of not yet implemented features with the corresponding tree walker functions.

4.Benchmarks of generated code and ways to improve

The compiler cuts down execution time because it makes control flow more efficient. Control flow in the interpreted code is defined by the parse tree. The parse tree consists of pointers to classes with dynamically binded (virtual) functions. In the compiled code control flow is defined by lined up inline C++ functions.

The compiled code is slowed down by the extra stack and the dynamic typing facilities.

Now the speedup of the compiled code against interpreted code is tested. The speedup is the ratio between Ti and Tc. Where Ti is the time the interpreter needs to compute a function. And Tc the time the compiled code needs. To test the speedup on a somewhat realistic case, I ran the cover example from above with n from 10 to 10 000 000. I expected the speedup to be not higher than 10.

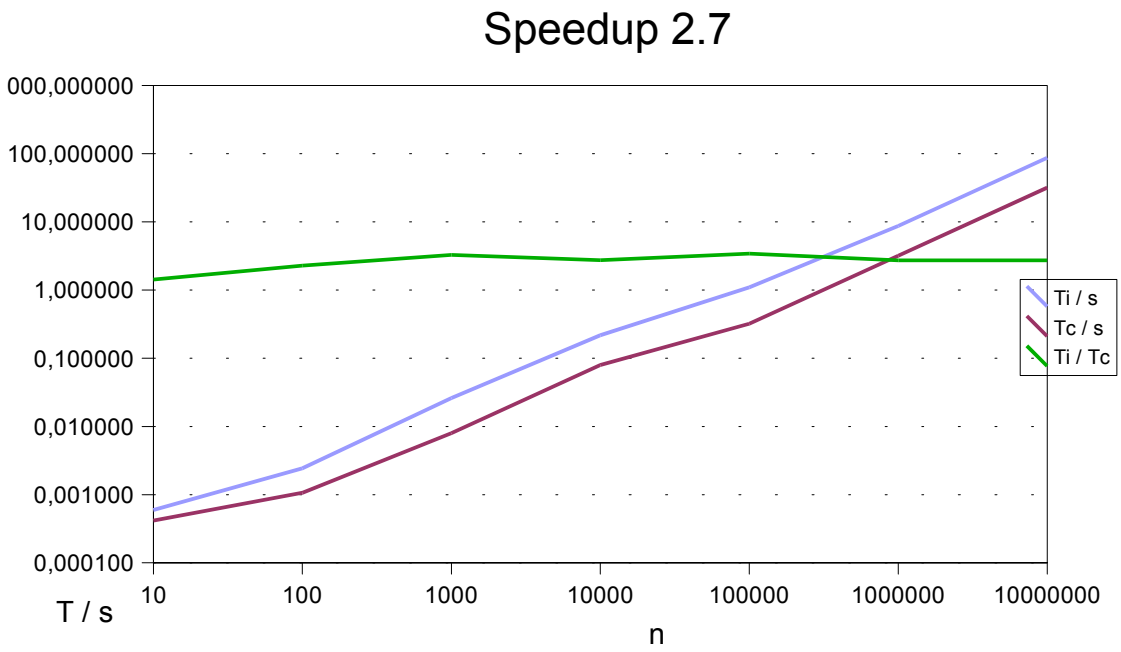


The average speedup is 1.7. For a real compiler this is not satisfying at all.

In the second benchmark the matrix multiplication is removed, to reduce its influence.

```
function c = cover(n)
        a = [1 1 0 0; 0 1 1 0; 0 0 1 1; 0 1 0 1]
        i = 1;
        c = 1;
        while ( i++ <= n )
#removed          c = c * a;
        endwhile
endfunction
```

This time the compiled code performs a bit better. Though still not satisfying. The average speedup is up to 2.7. But the empty loop is a rather unrealistic example.



4.1.Static typing

A facility to deduce as much type information as possible is needed to remove dynamic typing as far as possible. Obviously it cannot totally be removed (§1.4).

It may be needed to introduce different C++ identifiers for one variable in Octave, if the variable changes its type.

The thoughts of §2.2.6.2 have to be completed in a new research project beyond the scope of this paper.

4.2.Removing the o2c::o2c_value class

The stack machine needs to save octave_values and octave_lvalues on the stack (§3.4). Therefore it uses the class o2c::o2c_value as a union. Octave_lvalues are basically references to octave_values. So it may be possible to push only octave_values on the stack and spare the o2c::o2c_value class.

4.3.Removing the runtime stack

The runtime stack seems to be one big slow down. It is not necessary in the compiled code. Different approaches have been discussed in §2.2.8.

4.4.Replacing helper functions with macros

The helper functions can be replaced by macros to remove the overhead of calling them. These functions are inline. If the code is compiled with the compiler options -O3 -Winline -finline-limit=6000, all helper functions are inlined (and GCC would have issued a warning if not). I think there is no overhead in calling these inline functions. Though it may be worth to check the compiled assembly code.

4.5.Communication between parse tree nodes

Complex communication between the nodes of the parse tree is needed to implement type inference. The Octave' s tree_walker class does not support communication between the nodes of the parse tree. So it must be implemented manually if needed.

The Octave to C++ compiler dispense communication by building a stack machine. Anyway it needs to communicate to tell subexpressions whether they are building an l-value or an r-value. The compiler communicates this using the class wide variable lvalue. Manually rebuilding function parameters this way or in any other way is inflexible, inconvenient and error prone.

A tree walker which uses function parameters will not be accepted by the Octave parse tree. The Octave parse tree can be expanded by a decorator [GAMA95] to accept a more sophisticated tree walker. Or the parse tree can be translated into another parse tree designed for the compiler. Such a new parse tree can also include facilities to support type inference, optimization and code generation.

4.6.Optimization

The compiler often generates more complicated code than necessary. For example look at the

assignment `a = expression`, where a simple variable is target of the assignment and the = assignment operator is used. There is no need to build up the l-value on the stack and call octave_value::assign. Building the r-value on the stack and calling `a = o2c::st.top()` is much more efficient.

Including an optimizer which finds such special cases in the parse tree and replaces them with special nodes can help here.

5.Appendix

5.1.Table1 - Type of octave_values

This table shows all classes which implement type_name() or are octave_values. Use the is...() and type_name() functions to determine the type of the octave_value. E. g. octave_scalar tells the type_name() “scalar”.

Filename	Class name	Base Class	type_name()
ov.h	octave_value		
ov-base.h	octave_base_value	octave_value	<unknown type>
ov-base-mat.h	<> octave_base_matrix[4]	octave_base_value	
ov-bool-mat.h	octave_bool_matrix	octave_base_matrix <boolNDArray>	bool matrix
ov-cell.h	octave_cell	octave_base_matrix <Cell>	cell
ov-ch-mat.h	octave_char_matrix	octave_base_matrix <charNDArray>	char matrix
ov-str-mat.h	octave_char_matrix_str	octave_char_matrix	string
ov-cx-mat.h	octave_complex_matrix	octave_base_matrix <ComplexNDArray>	complex matrix
ov-fcn-handle.h	octave_fcn_handle	<fcn_handle_array>	function handle
ov-re-mat.h	octave_matrix	<NDArray>	matrix
ov-streamoff.h	octave_streamoff	<streamoff_array>	streamoff
ov-base-scalar.h	<> octave_base_scalar[5]	octave_base_value	
ov-bool.h	octave_bool	octave_base_scalar <bool>	bool
ov-complex.h	octave_complex	octave_base_scalar <Complex>	complex scalar
ov-scalar.h	octave_scalar	octave_base_scalar <double>	scalar
ov-colon.h	octave_magic_colon	octave_base_value	magic-colon

4 Class is a template

5 Class is a template

Filename	Class name	Base Class	type_name()
ov-cs-list.h	octave_cs_list	octave_base_value	cs-list
ov-fcn.h	octave_function	octave_base_value	
ov-builtin.h	octave_builtin	octave_function	built-in function
ov-dld-fcn.h	octave_dld_function	octave_function	dynamically-linked function
ov-mapper.h	octave_mapper	octave_function	built-in mapper function
ov-usr-fcn.h	octave_user_function	octave_function	user-defined function
ov-file.h	octave_file	octave_base_value	file
ov-list.h	octave_list	octave_base_value	list
ov-range.h	octave_range	octave_base_value	range
ov-struct.h	octave_struct	octave_base_value	struct
ov-va-args.h	octave_all_va_args	octave_base_value	va-arg
ov-fcn-handle.cc	fcn_handle_elt		function handle

5.2. Table2 - Implemented features

This table shows a list of features, yet implemented in the compiler, with the corresponding tree walker functions and runtime helper functions.

Feature	**Tree walker function**	**o2c_helper function**
parameter matrix	visit_argument_list	build_matrix_add_element
	visit_matrix	begin_build_matrix, build_matrix_begin_row, build_matrix_finish_row, end_build_matrix
unary and binary operations	visit_binary_expression	bin_op
	visit_postfix_expression	postfix_op
	visit_prefix_expression	prefix_op
range	visit_colon_expression	build_range, build_range_inc1
identifier	visit_identifier	push_lvalue, push_rvalue
if else	visit_if_clause	top_true
	visit_if_command	
	visit_if_command_list	
Constant (literal)	visit_constant	push_rvalue, matrix_constant
assignment	visit_simple_assignment	assign
statements	visit_statement	empty_stack
	visit_statement_list	
comments	print_comment_elt	
	print_comment_list	
	print_indented_comment	
while	visit_while_command	top_true

5.3.Table3 - Not yet implemented features

This table shows a list of not yet implemented features with the corresponding tree walker functions.

Feature	**Tree walker function**
loops	visit_break_command, visit_continue_command, visit_simple_for_command, visit_complex_for_command, visit_do_until_command
declarations	visit_decl_command, visit_decl_elt, visit_decl_init_list
function handling	visit_octave_user_function, visit_octave_user_function_header, visit_octave_user_function_trailer, visit_fcn_handle, visit_parameter_list, visit_return_command, visit_return_list
index	visit_index_expression
cell	visit_cell
multi assignment	visit_multi_assignment
no op	visit_no_op_command
plots	visit_plot_command, visit_plot_limits, visit_plot_range, visit_subplot, visit_subplot_axes, visit_subplot_list, visit_subplot_style, visit_subplot_using
switch	visit_switch_case, visit_switch_case_list, visit_switch_command
exceptions	visit_try_catch_command, visit_unwind_protect_command

5.4.References

AHO86 Alfred V. Aho, Ravi Sethi, Jeffry D. Ullman, Compilers Principles, Techniques and Tools, 1986

APPEL98 Andrew W. Appel, Modern Compiler Design in C, 1998

BIRD98 Richard Bird, Introduction to Functional Programming Using Haskell (Second Edition), 1998

GAMA95 Erich Gama, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns - Elements of Reusable Object-Oriented Software, 1995

OCTDOC John W. Eaton, Octave Documentation, 2004,www.octave.org

OCTSRC John W. Eaton, Octave Source Code, 1992,www.octave.org

STR00 Bjarne Stroustrup, The C++ Programming Language Special Edition, 2000