Introduction

This is a high level overview of MOY implementation of JOY in small pieces. When taken together, these pieces will give a picture of why and how MOY differs from the Joy1 implementation of JOY. MOY and Joy1 are implementations; the language is called JOY; 42minjoy is a minimal implementation of JOY.

Execution model of JOY

This is how a JOY program gets executed:

1) First, the program is read from file into memory.
2) Then, the program is compiled to internal format.
3) After that, the internal format is executed.
4) Execution proceeds by stepping through the internal format, factor after factor, executing one factor at a time.

This description concerns itself mostly with JOY as an interpreted language. Now, when compiling JOY, only step 3) remains during the execution phase. The result, of course, will be the same as when JOY is interpreted.

JOY compared to other Concatenative languages

JOY has conditions that, after reading the condition code, restore the stack to what it was before the condition started. All other concatenative languages discard this behavior, because of performance considerations.

Valid JOY programs

2 +

is a program that expects a number on top of the stack, adds 2 to that number and replaces the number with the result of the addition. It is a valid program.

If this program is executed on an empty stack, it will issue a runtime error. That error does not make the program invalid. It means that one of the assumptions about the start condition is violated.

What this means is that all sequences of factors are valid programs. It also means that runtime errors can occur.

More abstract: if P is a valid program and Q is a valid program, then so is P Q. In the example, both 2 and + are valid programs and so is 2 +.

Interface to C, implementation agnostic

One way to interface between a program written in C and a procedure written in JOY is to have the C program call the JOY procedure with *argc* and *argv* initialized with input parameters and one output parameter: a file-pointer received from tmpfile(). This file-pointer is then used in the JOY procedure to write data to. For that purpose a new built-in was introduced: *casting*, because *argv* is a table of strings, not file-pointers and the runtime type checker rejects the use of a string as file-pointer (or should reject such use).

Interface to C, implementation aware

Another way to interface with C is to have the C functions #include “runtime.h”. The C functions are now aware of the way that the stack is implemented in JOY and can PUSH input parameters on the stack before calling the JOY procedure and also read output from the stack after the JOY procedure returns.

In addition to that, it is possible to have definitions like A == A … that cause the compiler to generate a declaration void do_A(void); and no definition of do_A. This definition has to be supplied by the programmer during the link phase. This is an example of JOY calling a C function.

Additions in MOY

Some built-ins have been added in addition to *casting*. The tutorial from 42minjoy can be replayed with the addition of *nothing*, *index*, and *sametype*. For reasons of symmetry *fget* and *getch* have been added, next to the existing *fput* and *putch*. An arithmetic operator was added, *round*, because there is also *floor* and *ceil*. Lastly, *filetime* was added, because it is needed in an implementation of the *make* program. It is not part of standard C, so that is a good reason why it is not part of Joy1.

Compiling in MOY

MOY allows JOY sources to be compiled. The output is C source (to *stdout*) that can be compiled and linked with a JOY library of built-ins. As a consequence of the split between compile time behavior and runtime behavior two directives have been borrowed from 42minjoy. This is %PUT that prints a diagnostic message at compile time to *stderr* and %INCLUDE that includes source files at compile time.

Changes in MOY

Not all of the built-ins of JOY are supported in MOY. Echo of input to output is not supported. It is used in JOY to build listings of input alongside the result of interpreting that input. And error messages can be as short as “syntax error”. To mention one example: when trying to redefine a built-in, MOY will react with “syntax error” without even mentioning in what line the error should be located. Because of these changes, MOY is not a full replacement of Joy1.

Compiler internals

The compiler writes out the code that otherwise the interpreter would execute, in case the program is interpreted instead of compiled. This summarizes the task of the compiler. In practice a little more is going on. The compiler can also evaluate the program before writing it out. Writing out becomes necessary when seeing built-ins that can only be executed at runtime or when a function is called recursively or when there are not enough parameters on the compile-time stack available to allow executing a built-in. The MOY implementation has two compile options: –c and –o. The latter more thoroughly compiles the code and is used when benchmarking.

Benchmarking

When compiling, the user expects that the resulting program will run faster than the interpreted program. That is indeed the case, when using the –o compile option, but the main slowdown is caused by the way that the stack is organized. Any improvement there will benefit compiled programs as well as the interpreter.

Customizing

In Joy1 there is only one location where the program can be configured, in globals.h. In MOY the configuration is scattered over several .h and .c files. An important decision is whether 32 bits or 64 bits will be used as the width of the value field. That decision can be made in joy.h. The size of the static part of the stack can be set in node.h. The length of symbol names and the size of the symbol table can be set in symbol.h; the line length of input files can also be set in symbol.h; there are limits in history.c, listing.c, outfile.c, lexer.l, help.c, node.c, scan.c, and symbol.c

Type systems

MOY uses the same type system as JOY. JOY only does runtime typing. MOY does the same typing at compile time and if the types are ok and the built-in to be executed is not recursive, the built-in is executed at compile time. It is a bit more complicated than that, because some built-ins are generic w.r.t. type and thus need to be compiled more than once. The types are kept save in history.c and restored from there in between the multiple executions that are needed on behalf of the multiple types. In addition to that, there is also the arity count in arity.c that is used to decide whether conditions need to save and restore the stack with CONDITION and RELEASE statements.

Roadmap

There is no roadmap for future developments. The language is best used in a real application and missing parts can be added if and when the need arises. Maybe a multi-tasking version? Maybe multi-precision numbers? It all depends on the application whether these additions are needed.