The University of Melbourne
School of Computing and Information Systems
COMP90045 Programming Language Implementation

Assignment 3, 2020

Released: 1 October.
Submit test data (individual): Monday 19 October at 23:00
Submit compiler (team effort): Wednesday 28 October at 23:00

Objectives

To build a better understanding of a compiler's back-end, code generation, symbol tables, run-time structures, and (optionally) optimization. To practice cooperative, staged software development.

Background and context

The task is to write a compiler for a procedural language, Roo. The compiler translates source programs to the assembly language of a target machine Oz. These programs can then be run on a provided Oz emulator.

In an earlier stage, you wrote a parser for Roo. You may choose to start from that parser, or alternatively, start from one that has been (or will be) made available. In either case, correctness of the compiler, including the parser, is your responsibility.

This final stage also involves the completion of semantic analysis, code generation, and further optional tasks. The implementation language must be Haskell.

The source language: Roo

The syntax of Roo is already known to you from Stage 1. A Roo program lives in a single file and consists of a number of type and procedure definitions. There are no global variables. One (parameter-less) procedure must be named "main". Procedure parameters can be passed by value or by reference.

There are three primitive types, namely *integer*, *boolean*, and *string*. The usual arithmetic and comparison operators are available for integer expressions. For Booleans, Roo offers `and`, `or`, and `not`, as well as comparison operators. No operations are available on strings, only string literals have string type. However, the `write` and `writeln` commands can print strings, as well as Booleans and integers.

Roo also has records and one-dimensional static arrays. A field a of a record r is accessed as $r.a$ and can only have boolean or integer type. For arrays, the index type is always `integer`, and the type of elements is either `boolean`, `integer`, or a record type. Hence there are no arrays of arrays. Arrays are homogeneous in the sense that all elements of an array must have the same type.

Array and record types can only be referred to through type aliases. Type definitions must be global, and they introduce names (aliases) for the array/record types. Types use *name equivalence* rather than structural equivalence. That is, two expressions are only considered to have the same type if they have the same type name (`boolean`, `integer`, or type alias).

We now describe the language in more detail.

Syntax

The following are reserved words: `and`, `array`, `boolean`, `call`, `do`, `else`, `false`, `fi`, `if`, `integer`, `not`, `od`, `or`, `procedure`, `read`, `record`, `then`, `true`, `val`, `while`, `write`, `writeln`. The lexical rules were given in the specification for Stage 1.

An integer literal is a non-empty sequence of digits. A Boolean constant is `false` or `true`. A string constant (as can be used by "`write`" and "`writeln`") is a sequence of characters between double quotes. The sequence itself cannot contain double quotes or newline/tab characters. It may, however, make use of C string escape characters, such as '`\"`', '`\n`', '`\t`', '`\\`', and '`\b`.

The arithmetic binary operators associate to the left, and unary operators have higher precedence It follows, for example, that '`-5+6`' and '`4-2-1`' both evaluate to 1.

A Roo program consists of zero or more record type definitions, followed by zero or more array type definitions, followed by one or more procedure definitions.

Each record type definition consists of (in the given order):

1. the keyword `record`,
2. a non-empty list of *field declarations*, separated by semicolons, the whole list enclosed in braces,
3. an identifier, and
4. a semicolon.

A field declaration is of `boolean` or `integer`, followed by an identifier (the field name).

Each array type definition consists of (in the given order):

1. the keyword `array`,
2. a (positive) integer literal enclosed in square brackets,
3. a type name which is either an identifier (a type alias) or one of `boolean` and `integer`,
4. an identifier (giving a name to the array type), and
5. a semicolon.

Each procedure definition consists of (in the given order):

1. the keyword `procedure`,
2. a procedure header, and
3. a procedure body.

The header has two components (in this order):

1. an identifier—the procedure's name, and
2. a comma-separated list of zero or more formal parameters within a pair of parentheses (so the parentheses are always present).

Each formal parameter has two components (in the given order):

1. a parameter type/mode indicator, which is one of these five: a type alias, `boolean`, `integer`, `boolean val`, or `integer val`, and
2. an identifier.

A procedure body consists of zero or more local variable declarations, followed by a non-empty sequence of statements, the statements enclosed in braces. A variable declaration consists of a type name (`boolean`, `integer`, or a type alias), followed by a non-empty comma-separated list of identifiers, the list terminated with a semicolon. There may be any number of variable declarations, in any order.

An atomic statement has one of the following forms:

```
<lvalue> <- <exp> ;
read <lvalue> ;
write <exp> ;
writeln <exp> ;
call <id> ( <exp-list> ) ;
```

where `<exp-list>` is a (possibly empty) comma-separated list of expressions.

A composite statement has one of the following forms:

```
if <expr> then <stmt-list> fi
if <expr> then <stmt-list> else <stmt-list> fi
while <expr> do <stmt-list> od
```

where `<stmt-list>` is a non-empty sequence of statements, atomic or composite. (Note that semicolons are used to *terminate* atomic statements, so that a sequence of statements—atomic or composite—does not require any punctuation to *separate* the components.)

An lvalue (`<lvalue>`) has four (and only four) possible forms:

```
<id>
<id> . <id>
<id> [ <exp> ]
<id> [ <exp> ] . <id>
```

An example lvalue is `point[0].xCoord`.

An expression (`<exp>`) has one of the following forms:

```
<lvalue>
<const>
( <exp> )
<exp> <binop> <exp>
<unop> <exp>
```

where `<const>` is the syntactic category of boolean, integer, and string literals. The list of operators (`<binop>` and `<unop>`) is:

```
or
and
not
= != < <= > >=
+ -
* /
-
```

Here `not` is a unary prefix operator, and the bottom "`-`" is a unary prefix operator (unary minus). All other operators are binary and infix. All the operators on the same line have the same precedence, and the ones on later lines have higher precedence; the highest precedence being given to unary minus. The six relational operators are non-associative (so, for example, `a = b = c` is not a well-formed expression). The six remaining binary operators are left-associative. The relational operators yield Boolean values `true` or `false`, according as the relation holds or not. The Boolean operators `or` and `and` are for disjunction and conjunction, respectively.

The language supports comments, which start at a # character and continue to the end of the line. White space is not significant.

Name spaces

There are four separate name spaces, one for type aliases, one for field names, one for procedures, and one for variables. That means all type aliases must be distinct, and all procedure names must be distinct. Within a given record, field names must be distinct. Within a given procedure, variable names, including formal parameter names, must be distinct. However, it is possible for the same identifier to be used as a type alias and/or a field name and/or a procedure name and/or a variable name. Two different record types may use overlapping field names. Two different procedures may use overlapping variable names.

Static semantics

Roo allows procedure parameters to be passed by value or by reference. With each formal parameter is associated not only a type, but also a "mode" which determines the parameter passing mechanism used for that parameter. If the formal parameter's type is followed by the keyword `val` then it has value mode. If no such keyword is present, it has reference mode.

Any expression of `boolean` or `integer` type can be passed, by value, as an argument to a procedure. *Some* expressions of `boolean` or `integer` type can also be passed by reference, namely those that are "lvalues". Such an lvalue take one of four forms. It is a variable x (of primitive type), an array element $id[e]$ (of primitive type), or a field reference $x.a$ or $id[e].a$.

A formal parameter with value mode behaves exactly as if it were a local variable. We can therefore extend the notion of a "mode" beyond formal parameters, to local variables: A "by reference" formal parameter has reference mode, while other formal parameters, and all local variables, have value mode.

A record or an array may be passed *as a whole*, but in that case it must be passed by reference. Similarly, an assignment *lval* `<-` e, where `e` has record or array type, is only allowed if *lval* and e both have reference mode.

As mentioned, all defined procedures must have distinct names. A defined procedure does not have to be called anywhere, and the definition of a procedure does not have to precede the (textually) first call to the procedure. Procedures can use (mutual) recursion. For each procedure, the number of actual parameters in a call must be equal to the number of formal parameters in the procedure's definition.

The *scope* of a declared variable (or of a formal procedure parameter) is the enclosing procedure definition. A variable must be declared (exactly once) before it is used. Similarly a list of formal parameters must all be distinct. As mentioned, the same variable/parameter name can be used in different procedures.

In a record type definition, all field names must be given types `boolean` or `integer`. In an array type definition `array [`n`] ...`, n must be a positive integer. Each reference to an array variable must include exactly one index expression. Since the array bounds are known at declaration-time, semantic analysis *could* do limited bounds checking at compile-time, but there is no requirement to do this.

The language is statically typed, that is, each variable and parameter has a fixed type, chosen by the programmer. The type rules for expressions are as follows:

- The type of a Boolean constant is `boolean`.
- The type of an integer constant is `integer`.
- The type of a string literal is `string`.
- The type of an expression *id* is the variable *id*'s declared type. If the declaration uses a type alias, the type is the one given by the type definition for that alias.

- For an expression *lval.fname*, *lval* must be of record type. The type of an expression *lval.fname* is the type associated with field name *fname*, as given in the record type associated with *lval*.
- For an expression *id*[*e*], *id* must be of array type, and *e* must have type `integer`. The type of the expression is the array element type, as given in array type associated with *id*.
- Arguments of the logical operators must be of type `boolean`. The result of applying these operators is of type `boolean`.
- The two operands of a relational operator must have the same primitive type, either `boolean` or `integer`. The result is of type `boolean`.
- The two operands of a binary arithmetic operator must have type `integer`, and the result is of type `integer`.
- The operand of unary minus must be of type `integer`, and the result type is the same.

The type rules for statements are as follows:

- In assignment statements, an lvalue on the left-hand side must have the same type t as the expression on the right-hand side. If t is a record or array type, then the types of the two sides must have been provided as identical type aliases, and both must have reference mode.
- Conditions in `if` and `while` statements must be of type `boolean`. Their bodies must be well-typed sequences of statements.
- For each procedure call, the number of arguments must agree with the number of formal parameters used in the procedure definition, and the type of each actual parameter must be the type of the corresponding formal parameter.
- The argument to `read` must be an lvalue of type `boolean` or `integer`.
- The argument to `write` must be a well-typed expression of type `boolean` or `integer`, *or* a string literal. The same goes for `writeln`.

Every procedure in a program must be well-typed, that is, its body must be a sequence of well-typed statements. Every program must contain a procedure of arity 0 named "main".

Dynamic semantics

Integer variables are automatically initialised to 0, and Boolean variables to `false`. This extends to records and arrays.

The semantics of arithmetic expressions and relations is standard. In particular, the evaluation of an expression e_1`/`e_2 results in a runtime error if e_2 evaluates to 0. The ordering on Boolean values is defined by $x \leq y$ iff x is `false` or y is `true` (or both these hold). As usual, $x < y$ iff $x \leq y \wedge x \neq y$.

The logical operators are *strict* in all arguments and their arguments are evaluated from left to right. That is, Roo does not use short-circuit evaluation of Boolean expressions. For example, '`5 < 8 or 5 > 8/0`' causes a runtime error, rather than evaluating to `true`.

If a Roo program reads or writes outside the bounds of an array, the behaviour is undefined. (There is no requirement that out-of-bounds indices are detected at runtime, though this may be implemented as an optional extension, as discussed below).

`write` prints `integer` and `boolean` expressions to `stdout` in their standard syntactic forms, with no additional whitespace or newlines. If `write` is given a string, it prints out the characters of the string to `stdout`, with `\"` resulting in a double quote being printed, `\n` in a newline character being printed, and `\t` in a tab being printed. `writeln` behaves exactly like `write`, except it prints an additional, final newline character. Similarly, `read` reads an `integer` or `boolean` literal from `stdin` and assigns it to an lvalue. If the user input is not valid, execution terminates.[1]

The procedure "main" is the entry point, that is, execution of a program comes down to execution of a call to "main".

The language allows for two ways of passing parameters. Call by value is a copying mechanism. For each parameter e passed by value, the called procedure considers the corresponding formal parameter v a local variable and initialises this local variable to e's value. Call by value is specified through the keyword `val`.

Call by reference does not involve copying. Instead the called procedure is provided with the *address* of the actual parameter (which must be a variable z, field reference $x.a$, or array element $a[e]$ and the formal parameter v is considered a synonym for the actual parameter.[2]

Some subtleties of parameter passing come about as the result of *aliasing*. Consider the program on the right. As written, passing is by reference, and the program will print 8. If `y` instead is passed by value (by specifying `integer val y`), 4 will be printed. If both `x` and `y` are passed by value, the program will print 3.

```
procedure main()
    integer z;
{
    z <- 3;
    call p(z,z);
    writeln z;
}

procedure p(integer x, integer y)
{
    x <- 4;
    y <- y + x;
}
```

The rest of the semantics should be obvious—it follows standard conventions. For example, the execution of `while` e `do` ss `od` can be described as follows. First evaluate e. If e evaluates to `false`, the statement is equivalent to a no-op (a statement that does nothing). Otherwise the statement is equivalent to 'ss `while` e `do` ss `od`'.

The target language: Oz

Oz is an artificial target machine that resembles intermediate representations used by many compilers. An emulator for the Oz machine is supplied to you as a C program. This emulator reads Oz source files (which should be the output of your compiler) and executes them without further compilation. You are not required to modify the emulator, or understand its inner workings, and may treat it as a "black box" (although you may want to study the source code for your own benefit).

Oz has 1024 registers named r0, r1, r2, ... r1023. This is effectively an unlimited set of registers, and your compiler may treat it as such; your compiler may generate register numbers without checking whether they exceed 1023. Every register may contain a value of integer type.

Oz also has an area of main memory representing the stack, which contains zero or more stack frames. Each stack frame contains a number of stack slots. Each stack slot may contain a

[1] Various instructions in the Oz assembly language can be used to take care of these rules for you.

[2] In terms of stack slots in the frame allocated for a procedure call, one slot is needed for a parameter passed by value. The parameter is simply treated as a local variable. A parameter passed by reference also needs one slot, but in this case, the *address* of the parameter is what is stored, and indirect addressing must be used to access the variable.

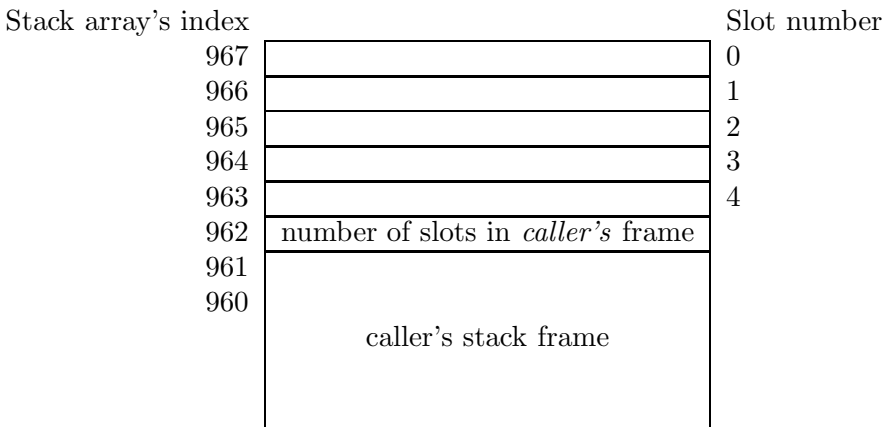

Figure 1: A stack frame on the 'stack' array

value of any type (it also holds type information about the value, for validation purposes), and you specify a stack slot by its stack slot number.

Figure 1 shows the "top" of the stack at some point. A stack pointer keeps track of the highest index used in the array, and stack slots can be accessed relative to this. Notice how the numbering of slots runs in the opposite direction of the array indices. In the example, the current stack frame (or activation record) has five slots (actually, it has one additional slot, used for remembering the size of the stack frame that will resume as current, once the active procedure returns).

An Oz program consists of a sequence of instructions, some of which may have labels. Although the emulator does not require it, good style dictates that each instruction should be written on its own line. As in most assembly languages, you can attach a label to an instruction by preceding it with an identifier (the name of the label) and a colon. The label and the instruction may be on the same line or different lines. Again, good style suggests that labels are somehow made to stand out in the generated Oz code. Identifiers have the same format in Oz as in Roo.

The following lists some relevant opcodes of Oz, and for each opcode, shows how many operands it has, and what they are. *The destination operand is always the leftmost operand.* The opcodes that have `real` as part of their name operate on floating point numbers and will not be of use to you—but it is good to know that they are there. There are also some further opcodes that you will not need.

```
push_stack_frame  framesize
pop_stack_frame   framesize
                                      #  C analogues:
store             slotnum, rI         #       x =  rI
load              rI, slotnum         #      rI =   x
load_address      rI, slotnum         #      rI =  &x
load_indirect     rI, rJ              #      rI = *rJ
store_indirect    rI, rJ              #     *rI =  rJ

int_const         rI, intconst
real_const        rI, realconst
string_const      rI, stringconst
```

```
add_int           rI, rJ, rK          #  rI =  rJ + rK
add_real          rI, rJ, rK          #  rI =  rJ + rK
add_offset        rI, rJ, rK          #  rI =  rJ + rK
sub_int           rI, rJ, rK          #  rI =  rJ - rK
sub_real          rI, rJ, rK          #  rI =  rJ - rK
sub_offset        rI, rJ, rK          #  rI =  rJ - rK
mul_int           rI, rJ, rK          #  rI =  rJ * rK
mul_real          rI, rJ, rK          #  rI =  rJ * rK
div_int           rI, rJ, rK          #  rI =  rJ / rK
div_real          rI, rJ, rK          #  rI =  rJ / rK
neg_int           rI, rJ              #  rI = -rJ
neg_real          rI, rJ              #  rI = -rJ

cmp_eq_int        rI, rJ, rK          #  rI = rJ == rK
cmp_ne_int        rI, rJ, rK          #  rI = rJ != rK
cmp_gt_int        rI, rJ, rK          #  rI = rJ >  rK
cmp_ge_int        rI, rJ, rK          #  rI = rJ >= rK
cmp_lt_int        rI, rJ, rK          #  rI = rJ <  rK
cmp_le_int        rI, rJ, rK          #  rI = rJ <= rK
cmp_eq_real       rI, rJ, rK          #  rI = rJ == rK
cmp_ne_real       rI, rJ, rK          #  rI = rJ != rK
cmp_gt_real       rI, rJ, rK          #  rI = rJ >  rK
cmp_ge_real       rI, rJ, rK          #  rI = rJ >= rK
cmp_lt_real       rI, rJ, rK          #  rI = rJ <  rK
cmp_le_real       rI, rJ, rK          #  rI = rJ <= rK

and               rI, rJ, rK          #  rI = rJ && rK
or                rI, rJ, rK          #  rI = rJ || rK
not               rI, rJ              #  rI = !rJ

int_to_real       rI, rJ              #  rI = (float) rJ
move              rI, rJ              #  rI = rJ

branch_on_true    rI, label           #  if (rI) goto label
branch_on_false   rI, label           #  if (!rI) goto label
branch_uncond     label               #  goto label

call              label
call_builtin      builtin_function_name
return
halt

debug_reg         rI
debug_slot        slotnum
debug_stack
```

The `push_stack_frame` instruction creates a new stack frame. Its argument is an integer specifying how many slots the stack frame has; for example the instruction `stack_frame 5` creates a stack frame with five slots numbered 0 through 4. (In the emulator, it also reserves an extra slot, slot 5, to hold the size of the previous stack frame, for error detection purposes.)

The `pop_stack_frame` instruction deletes the current stack frame. Its argument is an integer specifying how many slots that stack frame has; it must match the argument of the `push_stack_frame` instruction that created the stack frame being popped.

The `store` instruction copies a value from the named register to the stack slot with the given number. The `load` instruction copies a value from the stack slot with the given number to the named register. When a slot is first created, Oz marks it as uninitialized. An attempt to load an uninitialized value results in a runtime error.

The `load_address` instruction can be used by a caller to facilitate call by reference. The called procedure, having stored the address in the current stack frame, can then access and change the content of that address, by moving the address to a register and using `load_indirect` and `store_indirect`.

The `add_offset` and `sub_offset` instructions calculate addresses based on the offset from a given stack slot. They are useful when record fields or array elements need to be accessed or updated. The instruction '`add_offset rI rJ rK`' assumes that `rJ` holds an address, and `rK` holds an integer offset to be added to that address, the result being placed in `rI` (and similarly for `sub_offset`). Note that the Oz emulator is designed so that slot numbers grow in the opposite direction to how addresses grow, so `sub_offset` is appropriate when you want to *add* offsets to slot numbers, see Figure 1.

The `int_const`, `real_const`, and `string_const` instructions all load a constant of the specified type to the named register. The format of the constants is exactly the same as in Roo.

The `add_int`, `add_real`, `sub_int`, `sub_real`, `mul_int`, `mul_real`, `div_int`, `div_real`, `neg_int`, and `neg_real` instructions perform arithmetic. The first part of the instruction name specifies the arithmetic operation, while the second part specifies the shared type of all the operands.

The `cmp_eq_int`, `cmp_ne_int`, `cmp_gt_int`, `cmp_ge_int`, `cmp_lt_int` and `cmp_le_int` instructions, and their equivalents for floats, perform comparisons, generating integer results. The middle part of the instruction name specifies the comparison operation, while the last part specifies the shared type of both input operands.

The `and`, `or` and `not` instructions each perform the "logical" operation of the same name. Oz represents boolean values as integers, taking 0 to represent 'false' and a non-zero integer to represent 'true'.

The `int_to_real` instruction converts the integer in the source register to a floating point number in the destination register.

The `move` instruction copies the value in the source register (which may be of any type) to the destination register.

The `branch_on_true` instruction transfers control to the specified label if the named register contains a non-zero integer value. The `branch_on_false` instruction transfers control to the specified label if the named register contains 0. The `branch_uncond` instruction always transfers control to the specified label.

The `call` instruction calls the procedure whose code starts with the label whose name is the operand of the instruction, while the `call_builtin` instruction calls the built-in function whose name is the operand of the instruction. Procedures and functions take their first argument from register r0, their second from r1, and so on. During the call, the procedure may destroy the values in all the registers, so they contain nothing meaningful when the procedure returns. The exception is that the built-in functions that return a value, such as the read functions, put their return value in r0 (see the example in Appendix B). When the called procedure executes the `return` instruction, execution continues with the instruction following the call instruction.

The following are all built-in functions: `read_int`, `read_real`, `read_bool`, `print_int`, `print_real`, `print_bool`, and `print_string`. (There are a few other built-ins that you will not need.) The read functions take no argument. They read a value of the indicated type (using C's scanf function) from standard input, and return it in r0. The function `read_bool` accepts the strings "true" and "false" and returns a 1 and a 0, respectively. The print functions take a single argument of the named type in r0, and print it to standard output; they return nothing.

Each `call` instruction pushes the return address (the address of the instruction following it) onto the stack. The `return` instruction transfers control to the address it pops off the stack.

The `halt` instruction stops the program.

Oz also supports comments, which start at a # character and continue until the end of the line. It can be useful to have the code generator insert comments, as in the example below.

The `debug_reg`, `debug_slot` and `debug_stack` instructions are Oz's equivalent of debugging printfs in C programs: they print the value in the named register or stack slot or the entire stack. They are intended for debugging only; your submitted compiler should not generate them. If your code generator generates Oz code that does the wrong thing and you cannot sort out why, you can manually insert these instructions to better see what goes wrong. Calling the emulator with an '-i' option gives a trace of execution.

Appendix A shows a Roo source program, and Appendix B shows its possible translation.[3]

The Oz emulator starts execution with the first instruction in the program and stops when it executes the halt instruction. Note that the generated code therefore starts with a fixed two-instruction sequence that represents the Oz runtime system: the first instruction calls main, while the second (executed when main returns) is a halt instruction.

Summary of Tasks, Suggestions

The compiler should take the name of a source file on the command line. It should write the corresponding target program to standard output, or report errors. If an error has been discovered in any phase of compilation, no (partial) code should be written. The executable compiler must be called `Roo`. On success, it should use an exit code of 0. Upon failure, it should use some non-0 exit code.

You already have a working parser, and if not, you can use the supplied one (but note that, in any case, correctness of the parser is your responsibility). There is no requirement to submit the pretty-printer, so it does not matter if you have to make changes to the AST that invalidate your pretty-printer. If you want to emit helpful comments in the Oz code, as shown in Appendix B, parts of your pretty-printer may come in handy.

The semantic analysis phase consists of a lot of checking that well-formedness conditions are met. The code generation phase consists of generating correct Oz code from the AST. Of these, well-formedness checking is arguably the part that has the lowest learning-outcome benefit for the time invested. The marking scheme encourages you to concentrate on code generation, and then deal with the correct handling of ill-formed programs as time allows.

You are encouraged to work stepwise and increase the part of the language you can handle as you go. It makes sense to write a module `SymTable` that offers the symbol table services. The `Data.Map` and `Data.Map.Strict` libraries offer a dictionary type that may be useful. A module `Analyze` could do the semantic analysis of the AST, and it might want to store information in the symbol table.[4]

[3] As discussed in a tutorial, different (indeed simpler) translations are possible for this example—the translation scheme used here makes heavy use of a `load_address` r n; `store_indirect` r r' pattern which could be replaced by `store` n r' (as peephole optimisation may well do).

[4] One common problem that you avoid in this assignment is that, in the context of both integer and floating point values, compilers usually end up having to deal with type conversion.

Work on getting the AST ready for code generation quickly—you can always add the well-formedness checks later, as time permits. A module `Codegen` can be responsible for code generation. It will also want to interact with the symbol table.

A possible approach to implementing `Roo` incrementally is as follows (note that some static analysis is needed from the outset—types need to be determined for code generation):

1. Get the compiler working for the subset that consists of expressions (not including records and arrays) and the `write` and `writeln` statements. At first assume that procedures do not take any arguments and cannot use recursion (no procedure calls), so that `main` works. Now you should be able to compile `hello` program from the Joey examples given in Stage 1.
2. Add the `read` statement, and assignments.
3. Add compound statements (`if` and `while`). Now you should be able to compile `gcd.roo` from Appendix A.
4. Add procedure arguments and procedure calls, but initially for pass-by-value only.
5. Add reference parameters.
6. Add records. At this point you will probably have had to introduce an amount of static semantic analysis and decide how you want to keep track of type aliases—one possibility is to have the static semantic analyzer incorporate all information from global type definitions into variable tables used locally when analysing procedures.
7. Add arrays. Finally you can implement sorting algorithms.
8. Complete static semantic analysis.

If you want to extend the task, here are some ideas:

- (Not much of an extension, really.) Add run-time discovery of array bounds violations.
- Add optimisations based on peephole analysis (even a simple version can be quite effective). The Oz emulator will provide statistics if called with an '-s' option.
- On top of the previous extension, for a much more substantial challenge, add a static analysis to (1) remove unnecessary array bounds checks and (2) find definite bounds violations already at compile-time. The simplest solution would be to perform constant propagation. A more sophisticated approach would be an interval analysis that tracks which values integer expressions can take, over-approximating a set of values as an interval.
- Add C-style functions to the source language.

The following directories are, or will be made, available on the project module on Canvas:

- `oz/` contains the Oz emulator. The make file will generate an executable called `oz`.
- `parser/` contains a `Roo` parser which is believed to be correct.
- `simple_tests/` contains a number of small Roo programs for testing.
- `contributed_tests/` will later contain Roo programs submitted by you. (Be prepared for the possibility that some of these may not actually be valid Roo, or they may not behave the way their authors assumed.)

Procedure and assessment

The project may be solved in the teams, continuing from Stage 1. Each team should only submit once (under one of the members' name). If your team has changed since Stage 1, or if you are interested in taking on an extra team member, please let Harald know.

By 19 October, submit a single Roo program, which will be entered into a collection of test cases that will be made available to all. The program should be (syntactically, type, etc.) correct, but its runtime behaviour does not matter (whether it terminates, asks for input, divides by zero or whatever). Call your program `user.roo`, where `user` is your unimelb userid, and submit a separate file `user.in` with the intended input to `user.roo`, if it requires input. For this test submission stage, use `submit COMP90045 3a` to submit.

By 28 October, submit the code. There should be a `Makefile`, so that a `make` command generates `Roo`. Do not submit any files that are generated by a program generator such as alex or happy. Instead your `Makefile` should generate those files from alex or happy specifications that you submit. Also, do not submit `oz.c` or other files related to Oz. For this last stage, use `submit COMP90045 3b` to submit. It is possible to submit late, using `submit COMP90045 3b.late`, but late submissions will attract a penalty of 2 marks per calendar day late.

This project counts for 14 of the 30 marks allocated to project work in this unit. Members of a group will receive the same mark, unless the group collectively sign a letter, specifying how the workload was distributed. We very much encourage the use of Piazza and class time for discussions of ideas. On the other hand, soliciting of help from outside of our small community will be considered cheating and will lead to disciplinary action.

The marking sheet for Stage 3 will be made available on the LMS. Marks will be awarded on the basis of correctness (some 80%) and programming structure, style, readability, commenting and layout (some 20%). Of the correctness marks, most will be directed towards code generation, with scanning, parsing, semantic analysis and symbol table handling counting for less.

Code format rules

Your Haskell programs should adhere with the following simple formatting rules:

- Each file should identify the team that produced it.
- Every non-trivial Haskell function should contain a comment at the beginning explaining its purpose and behaviour.
- Variable and function names must be meaningful.
- Significant blocks of code must be commented. However, not every statement in a program needs to be commented. Just as you can write too few comments, it is possible to write too many comments.
- Code should be laid out not only to satisfy Haskell's layout rules, but also to look neat. Program blocks appearing in if-expressions, let clauses, etc., must be indented consistently. They can be indented using tabs or spaces, and can be indented 2, 4, or 8 spaces, as long as it is done consistently. (Tabs can sometimes be rendered differently by different printers/viewers; spaces are more likely to preserve the intended layout.)
- Each program line should contain no more than 80 characters.

Harald Søndergaard
27 September 2020

Appendix A: A Roo program

```
procedure main()
    integer x, y, temp;
    integer quotient;
    integer remainder;

{   write "Input two positive integers: ";

    read x;
    read y;

    write "\n";

    if x < y then
        temp <- x;
        x <- y;
        y <- temp;
    fi

    write "The gcd of ";
    write x;
    write " and ";
    write y;
    write " is ";

    quotient <- x / y;
    remainder <- x - quotient * y;

    while remainder > 0 do
        x <- y;
        y <- remainder;
        quotient <- x / y;
        remainder <- x - quotient * y;
    od

    write y;
    write "\n";
}
```

Appendix B: A generated Oz program

```
    call proc_main
    halt
proc_main:
# prologue
    push_stack_frame 5
# initialise int val quotient
    int_const r0, 0
    store 0, r0
# initialise int val remainder
    int_const r0, 0
    store 1, r0
# initialise int val temp
    int_const r0, 0
    store 2, r0
# initialise int val x
    int_const r0, 0
    store 3, r0
# initialise int val y
    int_const r0, 0
    store 4, r0
# write "Input two positive integers: ";
    string_const r0, "Input two positive integers: "
    call_builtin print_string
# read x;
    call_builtin read_int
    load_address r1, 3
    store_indirect r1, r0
# read y;
    call_builtin read_int
    load_address r1, 4
    store_indirect r1, r0
# write "\n";
    string_const r0, "\n"
    call_builtin print_string
# if x < y
    load r0, 3
    load r1, 4
    cmp_lt_int r0, r0, r1
    branch_on_false r0, label_1
label_0:
# then
```

```
# temp <- x;
    load r0, 3
    load_address r1, 2
    store_indirect r1, r0
# x <- y;
    load r0, 4
    load_address r1, 3
    store_indirect r1, r0
# y <- temp;
    load r0, 2
    load_address r1, 4
    store_indirect r1, r0
# fi
label_1:
# write "The gcd of ";
    string_const r0, "The gcd of "
    call_builtin print_string
# write x;
    load r0, 3
    call_builtin print_int
# write " and ";
    string_const r0, " and "
    call_builtin print_string
# write y;
    load r0, 4
    call_builtin print_int
# write " is ";
    string_const r0, " is "
    call_builtin print_string
# quotient <- x / y;
    load r0, 3
    load r1, 4
    div_int r0, r0, r1
    load_address r1, 0
    store_indirect r1, r0
# remainder <- x - quotient * y;
    load r0, 3
    load r1, 0
    load r2, 4
    mul_int r1, r1, r2
    sub_int r0, r0, r1
    load_address r1, 1
    store_indirect r1, r0
# while remainder > 0
label_2:
    load r0, 1
    int_const r1, 0
    cmp_gt_int r0, r0, r1
    branch_on_false r0, label_3
# do
# x <- y;
    load r0, 4
    load_address r1, 3
    store_indirect r1, r0
# y <- remainder;
    load r0, 1
    load_address r1, 4
    store_indirect r1, r0
# quotient <- x / y;
    load r0, 3
    load r1, 4
    div_int r0, r0, r1
    load_address r1, 0
    store_indirect r1, r0
# remainder <- x - quotient * y;
    load r0, 3
    load r1, 0
    load r2, 4
    mul_int r1, r1, r2
    sub_int r0, r0, r1
    load_address r1, 1
    store_indirect r1, r0
    branch_uncond label_2
# od
label_3:
# write y;
    load r0, 4
    call_builtin print_int
# write "\n";
    string_const r0, "\n"
    call_builtin print_string
# epilogue
    pop_stack_frame 5
    return
```