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

Assignment 3, 2016

Released: 22 April.
Submit test data (team effort): Friday 6 May at 23:00
Submit compiler (team effort): Friday 20 May at 23:00

Objectives

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

Background and context

The task is to write a compiler for Bean. The compiler translates source programs to the assembly language of a target machine Oz[1]. These programs can then be run on a provided Oz emulator.

In an earlier stage, you wrote a parser for Bean. You may choose to start from that parser, or alternatively, start from one that has been 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 (optionally) optimization. The implementation language must be OCaml.

The source language: Bean

Bean is already known to you from Stage 1 of the project, except that we did not specify the details of its semantics. A Bean program lives in a single file and consists of a number of 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.

Bean has only two primitive types, namely *int* and *bool*. The former allows for arithmetic and comparison operators. We do not consider the Boolean values to be ordered, but Boolean values can still be compared for equality `=` and `!=`. Arithmetic operations take C semantics (so results of over- and under-flow are undefined, and division rounds towards zero). Bean also has *structures* with named *fields*. These are like structs in C or records in OCaml. Fields can hold values of any Bean type, including structures. The "write" command can print integers, floating point numbers, booleans and, additionally, strings (though not structures).

Syntax

The following are reserved words: `and`, `bool`, `do`, `else`, `end`, `false`, `fi`, `if`, `int`, `not`, `od`, `or`, `proc`, `read`, `ref`, `then`, `true`, `typedef`, `val`, `while`, `write`.

[1]Not to be confused with the multi-paradigm language Oz, see `http://en.wikipedia.org/wiki/Oz_(programming_language)`.

An identifier in Bean is a non-empty sequence of lower and/or upper case letters, underscore (_), or apostrophe ('), except it cannot start with an apostrophe. Other lexical rules can be found in the Stage 1 spec.

The arithmetic binary operators associate to the left, and unary operators have higher precedence (hence for example, '`- 5 + 6`' and '`4 - 2 - 1`' both evaluate to 1). A Boolean constant is `false` or `true`. A string literal (as can be used by "`write`") is a sequence of characters between double quotes. The sequence itself cannot contain double quotes or escape characters, except it may include "`\n`", representing a newline character. A backslash not preceding an '`n`' is just considered part of the string.

A Bean program consists of zero or more type definitions, followed by one or more procedure definitions.

Each type definition consists of

1. the keyword `typedef`,
2. a type specification,
3. an identifier

in that order.

A type specification is any one of:

- the keywords `bool` or `int`,
- a non-empty comma-separated list of field definitions surrounded by `{` and `}`,
- an identifier

where each field definition is an identifier (a field name) and a type specification, separated by a colon.

Each procedure definition consists of (in this order):

1. The keyword `proc`.
2. A procedure header.
3. A procedure body.
4. The keyword `end`.

The header has two components (in this order):

1. An identifier—the procedure's name.
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 three components:

1. A parameter passing indicator (`val` or `ref`).
2. A type specification.
3. An identifier.

The procedure body consists of zero or more local variable declarations, followed by a non-empty sequence of statements. A variable declaration consists of a type specification, followed by an identifier, terminated with a semicolon.

Atomic statements have one of the following forms:

```
<lvalue> := <rvalue> ;
read <lvalue> ;
write <expr> ;
<id> ( <expr-list> ) ;
```

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

An assignment's left-hand side <lvalue> is either a variable identifier, or an <lvalue> followed by a period, then a field name. Note that by this recursive definition, x.f.g is an <lvalue>. An assignment's right-hand side rvalue is either an expression, or a structure initialization which consists of {, then a (possibly empty) comma separated list of field initializers of the form <ident> = <rvalue>, finally followed by }. A structure initialization may omit one or more fields appearing in the structure; omitted fields are set to their default values. Fields appearing in a structure initialization may appear in any order, but may not appear more than once.

Composite statements have 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 expression has one of the following forms:

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

The list of operators is

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

not is a unary prefix operator, and the bottom "-" is a unary prefix operator (unary minus). All the 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 not associative. The six remaining binary operators are left-associative. The relational operators yield Boolean values true or false, according as the relation is true or not.

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

Composite variables and name spaces

A value of structure type (composite type) has named components that may be inspected selectively. So a variable of structure type consists of components that are themselves variables that can be inspected and updated selectively. The components are named by adding *field names* as suffixes. For example

```
typedef {a: int, b: bool} mytype;
mytype x;
```

gives rise to three variables, namely the composite `x` of type `mytype`, the scalar `x.a` (of type `int`) and the scalar `x.b` (of type `bool`).

Each type name must be unique, that is, it cannot coincide with any other type name. It is allowed to coincide with a procedure name, but not with any variable/parameter names.

Every program must contain a parameter-less procedure named "main".

All defined procedures must have distinct names, and within the scope of a given procedure, all variable names and formal parameter names must be distinct. However, the same variable or formal parameter name can be used in different procedures. Moreover, a procedure name p is allowed to be used also as a type or variable/formal parameter, both within the procedure p and in other procedures (that is, procedures are in a separate name space from variables and types).

In a given structure, each field must have a distinct name. However, the same field name can be used in different structures. Moreover, a field name may also be used as the name of a procedure, variable, or formal parameter.

Static semantics

Bean is statically typed, that is, each variable and parameter has a fixed type, chosen by the programmer.

We define a *scalar* to be an object that contains a single value, that is, the object has type `int` or `bool`. Only scalar values may be used in expressions: every valid expression has type `bool` or `int`. The type rules for expressions are as follows:

- The type of a Boolean constant is `bool`.
- The type of an integer constant is `int`.
- The type of an expression *id* is the variable *id*'s declared type.
- Arguments of the logical operators, and their results, must be of type `bool`.
- The two operands of `=` must have the same type; the same goes for `!=`. The result is Boolean.
- The two operands of any other comparison operator must be of type `int`; the result is of type `bool`.
- The two operands of a binary arithmetic operator must be of type `int`; the result type is `int`.
- The operand of unary minus is of type `int` and so is the result.

The type rules for statements are as follows:

- In assignment statements, the `<lvalue>` on the left-hand side and the `<rvalue>` on the right-hand side must have the same type. Note that a structure can be updated as a whole.
- Conditions in `if` and `while` statements must be of type `bool`.
- In procedure calls, the type of an actual parameter must be the type of the corresponding formal parameter.
- `read` takes the name of a scalar and `write` takes a well-typed expression *or* a string literal.

A type name can not be used before its definition, so type definitions must be acyclic, and must appear in a proper order, so that a type name is always defined before its first use.

Types enjoy "structural equivalence" (as opposed to "name equivalence"). So a type definition that uses the type specification `bool` or `int` merely introduces a *type alias*. For example, `typedef int height` just makes `height` an alias for `int`, so that `height h;` simply declares an integer variable `h`. This variable `h` can then legally be compared against any expression of type `int` (or other aliases of `int`), be assigned values of type `int`, and be passed to a procedure in an argument position where an `int` is expected, and vice versa.

The same principle applies to composite types. Two composite types are considered equivalent if and only if they have identical *sets* of field names, with pointwise equivalent types. For example, given

```
typedef int height;
typedef {a: int, b: bool} type_x;
typedef {b: bool, a: height} type_y;
typedef {a: bool, b: int} type_z;
type_x x; type_y y; type_z z;
```

we consider `x` and `y` to have the same type, but `z` has a different type. (Of course the components `x.a`, `y.a` and `z.b` all have the same type.) While elements of `type_x` cannot be used in comparisons, an assignment such as `x := y;` is legal, and `x` can be passed to a procedure where an element of type `type_y` is expected. Note that a structure can be passed, as a whole, to a procedure (by value or reference).

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 used. It does not need to be initialized; Bean uses default initialization: `false` for `bool`, `0` for `int`. Incomplete initialisation of structures is also permitted; unspecified fields take their default value.

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. An expression in an actual argument position where the parameter passing method is "by reference" must be an `<lvalue>`. A formal procedure parameter is treated as a local variable.

A defined procedure does not have to be called anywhere; leaving it unused is not an error. The definition of a procedure p does not have to precede the (textually) first call to p. Thus procedures can be defined by mutual recursion.

Dynamic semantics

Integer variables are automatically initialised to 0, and Boolean variables to `false`. This extends to the fields of structures. The evaluation of an expression e_1`/`e_2 results in a runtime error if e_2 evaluates to 0.

The logical operators are *strict*, that is, they evaluate all their operands fully, rather than using short-circuit evaluation. For example, `5 < 8 or 5 > 8/0` yields a runtime error rather than `true`.

`write` prints `int` and `bool` expressions to `stdout` in their standard syntactic forms, with no additional white space or newlines. If `write` is given a string literal, it prints out the characters of the string to `stdout`, with `\n` resulting in a newline character being printed. Similarly, `read` reads `int` and `bool` literals from `stdin` and assigns them to variables or structure fields. If the user input is not valid, execution terminates.[2].

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 (`val`) 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 reference (`ref`) does not involve copying. Instead the called procedure is provided with the *address* of the actual parameter (which must be a variable z), and the formal parameter v is considered a synonym for z.[3]

Some subtleties of parameter passing come about as the result of *aliasing*. Consider the program on the right. If `<method>` is `val`, the program will print '4'. If it is `ref`, the program will print '8'. If both arguments were passed by value, the result would have been '3'.

```
proc main()
int z;
    z := 3;
    p(z,z);
    write z;
end

proc p(ref int x, <method> int y)
    x := 4;
    y := y + x;
end
```

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 closely 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 type `int` or `float` (referred to as `real` in the emulator).

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

[3] 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.

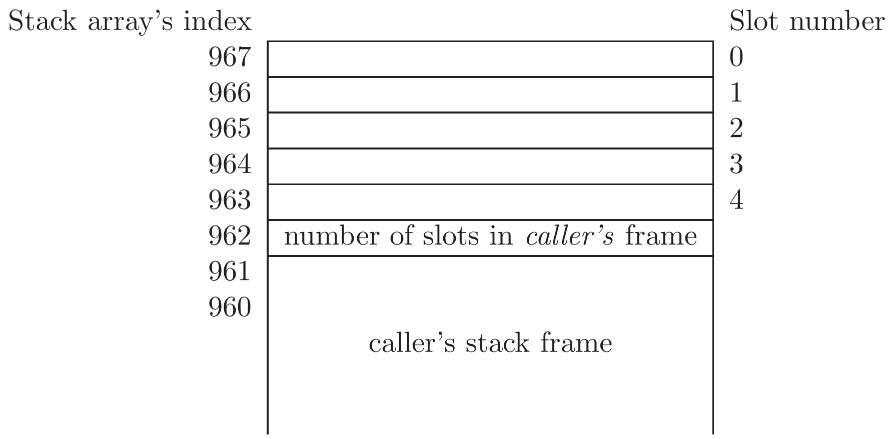

Figure 1: A stack frame on the 'stack' array

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 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 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. Identifiers have the same format in Oz as in Bean.

The following lists the 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.* (There are several other opcodes in Oz, such as for floating point operations, but you will not need those.)

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

int_const         rI, intconst
string_const      rI, stringconst

add_int           rI, rJ, rK          #  rI =  rJ + rK
add_offset        rI, rJ, rK          #  rI =  rJ + rK
```

```
sub_int          rI, rJ, rK           #  rI =  rJ - rK
sub_offset       rI, rJ, rK           #  rI =  rJ - rK
mul_int          rI, rJ, rK           #  rI =  rJ * rK
div_int          rI, rJ, rK           #  rI =  rJ / rK

cmp_eq_int       rI, rJ, rK           #  rI = rJ == rK
cmp_ne_int       rI, rJ, rK
cmp_gt_int       rI, rJ, rK           #    etc.
cmp_ge_int       rI, rJ, rK
cmp_lt_int       rI, rJ, rK
cmp_le_int       rI, rJ, rK

and              rI, rJ, rK           #  rI = rJ && rK
or               rI, rJ, rK           #  rI = rJ || rK
not              rI, rJ               #  rI = !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 `load` instruction copies a value from the stack slot with the given number to the named register. The `store` instruction copies a value from the named register to the stack slot with the given number. 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 array components 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` and `string_const` instructions load a constant of the specified type to the named register. The format of the constants is exactly the same as in Bean.

The `add_int`, `sub_int`, `mul_int`, and `div_int` 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. As with Bean, arithmetic operations take C integer semantics.

The `cmp_eq_int`, `cmp_ne_int`, `cmp_gt_int`, `cmp_ge_int`, `cmp_lt_int` and `cmp_le_int` instructions 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.

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 Figure 3). 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_bool`, `print_int`, `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) from standard input, and return it in r0. The function `read_bool` accepts the strings "true" and "false". 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 may 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.

Figure 2 shows the source program `gcd.bean`, and Figure 3 shows one possible translation. 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.

```
proc main()
    int x;
    int y;
    int temp;
    int quotient;
    int 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";
end
```

Figure 2: The Bean program gcd.bean

```
    call proc_main
    halt
proc_main:
# prologue
    push_stack_frame 5
    int_const r0, 0
    store 0, r0    # int x
    store 1, r0    # int y
    store 2, r0    # int temp
    store 3, r0    # int quotient
    store 4, r0    # int remainder
# write
    string_const r0, "Input two positive integers: "
    call_builtin print_string
# read
    call_builtin read_int
    store 0, r0
# read
    call_builtin read_int
    store 1, r0
# write
    string_const r0, "\n"
    call_builtin print_string
# if
    load r0, 0
    load r1, 1
    cmp_lt_int r0, r0, r1
    branch_on_false r0, label0
# assignment
    load r0, 0
    store 2, r0
# assignment
    load r0, 1
    store 0, r0
# assignment
    load r0, 2
    store 1, r0
label0:
# write
    string_const r0, "The gcd of "
    call_builtin print_string
# write
    load r0, 0
    call_builtin print_int
# write
    string_const r0, " and "
    call_builtin print_string
# write
    load r0, 1
    call_builtin print_int
```

Figure 3: Translated program (first part)

```
# write
    string_const r0, " is "
    call_builtin print_string
# assignment
    load r0, 0
    load r1, 1
    div_int r0, r0, r1
    store 3, r0
# assignment
    load r0, 0
    load r1, 3
    load r2, 1
    mul_int r1, r1, r2
    sub_int r0, r0, r1
    store 4, r0
# while
label1:
    load r0, 4
    int_const r1, 0
    cmp_gt_int r0, r0, r1
    branch_on_false r0, label2
# assignment
    load r0, 1
    store 0, r0
# assignment
    load r0, 4
    store 1, r0
# assignment
    load r0, 0
    load r1, 1
    div_int r0, r0, r1
    store 3, r0
# assignment
    load r0, 0
    load r1, 3
    load r2, 1
    mul_int r1, r1, r2
    sub_int r0, r0, r1
    store 4, r0
    branch_uncond label1
label2:
# write
    load r0, 1
    call_builtin print_int
# write
    string_const r0, "\n"
    call_builtin print_string
# epilogue
    pop_stack_frame 5
    return
```

Figure 4: Translated program (remaining part)

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. The executable compiler must be called `bean`.

You already have a working parser, and if not, you can use the supplied one (but 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.

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 covered as you go. It makes sense to write a module `symbol.ml` that offers the symbol table services. A module `analyze.ml` can do the semantic analysis of the AST, and it will want to store information in the symbol table. 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.ml` can be responsible for code generation. It will also want to interact with the symbol table.

A possible approach to implementing Bean incrementally is as follows:

1. Get the compiler working for the subset that consists of expressions (not including structure components) and the `write` statement. Assume that procedures do not take any arguments and cannot use recursion (no procedure calls), so that `main` works.
2. Add the `read` statement, and assignments.
3. Add compound statements (`if` and `while`). (Now you should be able to compile `gcd.bean`.)
4. Add procedure arguments and procedure calls, but initially for pass-by-value only.
5. Add reference parameters.
6. Add structures.
7. Complete static semantic checks.

If you want to extend the task, add optimizations. For example, a simple peephole analysis can be quite effective, and there is unlimited scope for other types of optimization. The emulator will provide statistics if called with an '-s' option.

The following files and folders are, or will be made, available on the LMS:

- `oz/` contains the Oz emulator. The make file will generate an executable called `oz`.
- `parser/` contains a Bean parser which is believed to be correct.
- `simple_tests/` contains a number small Bean programs for testing.
- `contributed_tests/` will contain Bean programs as submitted by teams (once they have been submitted).
- `marksheet_3.pdf` is the marking sheet we will use.

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, please let Harald know.

By 6 May, submit a single Bean 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 `teamName.bean`, where `teamName` is your team's name (use letters only), and submit a separate file `teamName.in` with the intended input to `teamName.bean`, if it requires input. For this stage, use `submit COMP90045 3a` to submit.

By 20 May, submit the code. There should be a `Makefile`, so that a `make` command generates `bean`. Do not submit the OCaml files that are generated by ocamllex and ocamlyacc; instead your `Makefile` should generate those OCaml files. 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 encourage the use of the LMS discussion forum and class time for discussions of ideas.

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 (approximately 20%). Out of the correctness marks, roughly two thirds will be directed towards code generation, with scanning, parsing, semantic analysis and symbol table handling counting for the remaining third.

Appendix: Code format rules

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

- Each file should identify the team that produced it.
- Every non-trivial OCaml 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.
- 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. Beware that some of the scaffolding code used spaces.
- Each program line should contain no more than 80 characters.

Graeme Gange and Harald Søndergaard
21 April 2016