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

Assignment 3, 2018

Released: 27 April.
Submit test data (team effort): Tuesday 8 May at 23:00
Submit compiler (team effort): Tuesday 22 May at 23:00

Objectives

To gain understanding of a compiler's middle- and 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 a procedural language, Paz. The compiler translates source programs to the assembly language of a target machine Taz. These programs can then be run on a provided Taz emulator.

In an earlier stage, you wrote a parser for Paz. 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 (optionally) optimization. The implementation language must be Haskell.

The source language: Paz

The Stage 1 specification introduced the syntax of the language Paz. We now define Paz properly, including its static and dynamic semantics. Although its syntax was inspired by Pascal, Paz semantics differs from Pascal's in a number of ways.

A Paz program lives in a single file and consists of a number of procedure definitions and the body of the program. Procedure parameters can be passed by value or by reference.

The language has three primitive types, namely *integer*, *real*, and *boolean*. The first two are *numeric* types and allow for arithmetic and comparison operators. Paz also has static one-dimensional arrays. The built-in "write" procedure can print integers, floating point numbers, booleans and, additionally, strings.

Syntax

The lexical and syntax rules were provided with the specification for Stage 1. Here we just recall the arithmetic operators:

```
+  -
*  /  div
-
```

and the fact that the first five (the infix binary operators) associate to the left. All the operators on the same line have the same precedence, and the ones below them have higher precedence; the highest precedence being given to unary minus. (For example, '`- 5 + 6`' and '`4 - 2 - 1`' both evaluate to 1). The six relational operators:

```
=  <>  <  <=  >  >=
```

are non-associative and have lower precedence than the arithmetic operators.

Boolean-typed expressions can be combined with the infix binary operators `or` and `and` (which also associate to the left) and the unary prefix `not`. Here `not` has higher precedence than `and`, which has higher precedence than `or`.

A real constant is a sequence of one or more digits, followed by a decimal point, and another sequence of one or more digits. A Boolean constant is `false` or `true`. A string constant (as can be used by the built-in procedure "`write`" and only by that) is a sequence of characters between single quotes (apostrophes). If an apostrophe (`'`) is needed in a string constant, it can be produced by using two apostrophes in succession (`''`). No other characters are special within string constants. That is, no escape sequences are processed, and a string constant can contain braces, which are otherwise used for comments.

The language supports comments; these are strings enclosed in braces ({ and }). Paz does not support nested comments. White space is significant only in strings and comments.

Static semantics

Static semantics refers to language aspects (beyond the narrowly syntactic) that can be checked at compile-time.

Naturally, a variable *id* can be inspected and updated, as can an indexed variable *id*[*e*]. Each holds a single value. Paz does not allow an array *as a whole* to be assigned or passed as a parameter. Only *selective updating* is possible. In other words, if *id* has been declared as an array then *id* is illegal as an expression on its own—every valid expression has type `boolean`, `integer`, or `real` (except the argument to `write` can be a string).

A variable must be declared before it is used. The also applies to control variables used in `for` loops. Within a given scope, all variables and formal parameters must have distinct names.

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. 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 the program variables (that is, any variable declared before the first "`begin`" in a program) is the program body. These variables are not visible to procedures.

A formal procedure parameter is treated as a local variable. 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. Similarly a list of formal parameters must all be distinct. However, the same variable/parameter name can be used in different procedures. An expression in an actual argument position where the parameter passing method is "by reference" must be a variable or indexed variable.

For the integer interval $m..n$ of an array declaration, $m \leq n$ must hold. Array identifiers obey the same scope rules as other variables. 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 real constant is `real`.
- The type of an expression *id* is the variable *id*'s declared type, which is one of the three primitive types.
- In an indexed variable expression $id[e]$, the expression e must have type `integer`. The type of $id[e]$ is the (primitive) type given in the declaration of *id*.
- Arguments of the logical operators must be of type `boolean`. The result is of type `boolean`. (Note that the logical operators are not bitwise operators; C programmers should be aware that they correspond to `||`, `&&`, and `!`, rather than `|`, `&`, and `~`.)
- The two operands of a comparison operator must either have the same numeric type, or one must be `integer` and the other `real` (in which case the compiler must ensure the integer gets converted to a real before the comparison). The result is of Boolean type.
- The two operands of `div` (which is for integer division) must be of type `integer`, and the result is of type `integer`.
- The two operands of `/` must either have the same numeric type, or one must be `integer` and the other `real`. The compiler should ensure that arguments of `integer` type are converted to `real`. In all cases the type of the result is `real`.
- For any other binary arithmetic operator, the two operands must either have the same numeric type, or one must be `integer` and the other `real` (in which case the compiler must ensure the integer gets converted to a real before the comparison). The result type is `real`, unless both operands have type `integer`, in which case the result type is `integer`.
- The operand of unary minus is of type `integer` or `real`, and the result type is the same as the operand's type.

The type rules for statements are as follows:

- In assignment statements, the variable (or indexed variable) on the left-hand side and the expression on the right-hand side must have the same type, with one exception: an `integer` expression may be assigned to a `real` (indexed) variable (the compiler must ensure the value to be assigned is converted from `integer` to `real`). Note that arrays can only be updated selectively; an array as a whole cannot be assigned to.
- Conditions in `if` and `while` statements must be of type `boolean`.
- In a statement '`for` i `:=` e_1 `(down)to` e_2 `do` s', i, e_1, and e_2 must all have type `integer`.
- For procedure calls, if a formal parameter is declared `var` (see explanation of call-by-reference in the dynamic semantics section), then the corresponding actual parameter must be a variable or an indexed variable, of the exact same primitive type as the formal parameter.
- For procedure calls, if a formal parameter is not declared `var` (see explanation of call-by-value in the dynamic semantics section), then the corresponding actual parameter can be any expression, as long as the two have the same type, or, alternatively, the actual parameter is of type `integer` and the formal parameter is of type `real` (in which case the compiler must ensure the integer is converted to a real).

Finally, the built-in `writeln` takes no argument, whereas `read` and `write` each take a single argument. The argument to `read` is either an indexed variable or a variable of primitive type. The argument to `write` is a well-typed expression *or* a string literal.

Dynamic semantics

Local variables to a procedure are not automatically initialized, so it is up to the Paz programmer to initialize them before use. Accessing an uninitialized variable causes a runtime error.

The evaluation of an expression e_1`/`e_2 results in a runtime error if e_2 evaluates to 0. The same goes for e_1 `div` e_2.

The relational operators yield `boolean` values *true* or *false*, according to whether the relation holds or not. 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`.

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

`write` prints `integer`, `real` and `boolean` expressions to `stdout` in their standard syntactic forms, with no additional whitespace or newlines. If `write` is given a string literal, it prints out the characters of the string to `stdout`, with each occurrence of two successive single quotes (apostrophes) replaced by a single apostrophe. `writeln` writes a single newline character to `stdout` (and this is the only way to print a newline).

`read` reads `integer`, `real` and `boolean` literals from `stdin` and assigns them to variables or indexed variables. If the user input is not valid, execution terminates.[1]

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 initializes this local variable to e's value.

Call by reference (indicated by the keyword `var` in front of the formal parameter) does not involve copying. Instead the called procedure is provided with the *address* of the actual parameter, which must be a (possibly indexed) variable z, and the formal parameter v is considered a synonym for z.[2]

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

```
program p;

var z: integer;

procedure p(var x: integer;
            <?> y: integer);
begin
    x := 4;
    y := y + x
end;

begin
    z := 3;
    p(z,z);
    write(z)
end.
```

The empty statement has no effect. A statement of the form '`for` i `:=` e_1 `to` e_2 `do` s' is equivalent to

i `:=` e_1`; while` i `<=` e_2 `do begin` s`;` i `:=` i `+ 1 end`.

As for all variables, the control variable i must have been declared. A statement of the form '`for` i `:=` e_1 `downto` e_2 `do` s' is equivalent to

i `:=` e_1`; while` i `>=` e_2 `do begin` s`;` i `:=` i `- 1 end`.

The semantics of conditionals and while loops should be clear. Code generation for these can be done as described in a lecture. Loosely, the execution of `while` e `do` s can be described as follows. First evaluate e. If e evaluates to `false`, the statement is equivalent to the empty statement. Otherwise the statement is equivalent to 's`; while` e `do` s'.

[1] Various instructions in the Taz assembly language can be used to take care of correct read/write behaviour.

[2] In terms of stack slots allocated, a parameter passed by reference also needs a single slot, but in this case the slot holds the parameter's *address*, and indirect addressing must be used to access the variable.

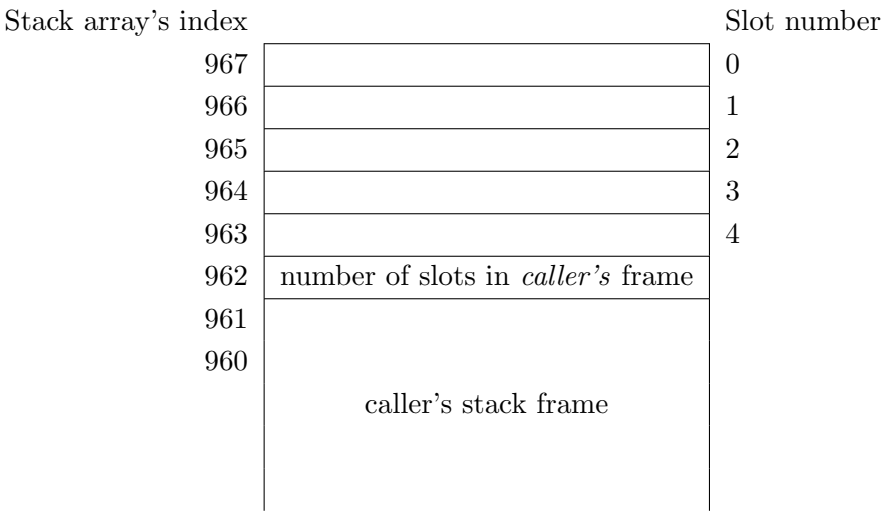

Figure 1: A stack frame on the 'stack' array

The target language: Taz

Taz is an artificial target machine that closely resembles intermediate representations used by many compilers. An emulator for the Taz machine is supplied to you as a C program. This emulator reads Taz 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).

Taz 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 `real` (referred to as `real` in the emulator).

Taz 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).

A Taz 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 Taz as in Paz.

The following lists the relevant opcodes of Taz (there are a few more), and for each opcode, shows how many operands it has, and what they are. *The destination operand is always the leftmost operand.*

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

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 Taz 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 the same as in Paz.

The `add_int`, `add_real`, `sub_int`, `sub_real`, `mul_int`, `mul_real`, `div_int` and `div_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 reals, 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 `int_to_real` instruction converts the integer in the source register to a real 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 Figure 3). When the called procedure executes the `return` instruction, execution continues with the instruction following the call instruction.

The built-in functions include `read_int`, `read_real`, `read_bool`, `print_int`, `print_real`, `print_bool`, `print_string`, and `print_newline` (there are a few others that you will not need). The read functions take no argument. They read a value of the indicated type (using scanf) from standard input, and return it in r0. The function `read_bool` accepts the strings "true" and "false", and returns 1 or 0 in r0 respectively. A print function takes a single argument of the named type in r0 and prints it to standard output; it returns 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.

Taz 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 Taz'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 Taz 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.paz`, and Figure 3/4 gives one possible translation. The Taz 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 Taz runtime system: the first instruction executes the program's body, treating it as a top-level procedure, while the second (executed when that top-level procedure 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. The executable compiler must be called `Paz`. On success, it must return 0 from main, and non-0 on failure.

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, as well as decorating of the AST with attributes that support code generation. The code generation phase consists of generating correct Taz 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.

```
program gcd;

{ Greatest common divisor of two positive integers
  using a variant of Euclid's method }

var
    x, y, temp, quotient, remainder: integer;

begin
    write('Input a positive integer: ');
    read(x);
    write('And another positive integer: ');
    read(y);

    writeln;

    if x < y then
    begin
        temp := x;
        x := y;
        y := temp
    end;

    write('The gcd of ');
    write(x);
    write(' and ');
    write(y);
    write(' is ');

    quotient := x div y;
    remainder := x - quotient * y;

    while remainder > 0 do
    begin
        x := y;
        y := remainder;
        quotient := x div y;
        remainder := x - quotient * y
    end;

    write(y);
    writeln
end.
```

Figure 2: The Paz program gcd.paz

```
    call main
    halt
main:
# prologue
    push_stack_frame 5
# write
    string_const r0, 'Input a positive integer: '
    call_builtin print_string
# read
    call_builtin read_int
    store 0, r0
# write
    string_const r0, 'And another positive integer: '
    call_builtin print_string
# read
    call_builtin read_int
    store 1, r0
# writeln
    call_builtin print_newline
# 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
# write
    string_const r0, ' is '
    call_builtin print_string
```

Figure 3: Translated program (first part)

```
# 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
# writeln
    call_builtin print_newline
# epilogue
    pop_stack_frame 5
    return
```

Figure 4: Translated program (remaining part)

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.hs` that offers the symbol table services. A module `Analyze.hs` 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.hs` can be responsible for code generation. It will also want to interact with the symbol table.

A possible approach to implementing `Paz` incrementally is as follows (some static analysis may be 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 arrays) and `write`. To treat the program's body as a procedure, for now assume that procedures do not take any arguments and cannot use recursion (no procedure calls).
2. Add variables, `read`, assignments, and compound statements.
3. Add `if` and `while`. (Now you should be able to compile `gcd.paz`.)
4. Add procedure arguments and procedure calls, but initially for pass-by-value only.
5. Add reference parameters.
6. Add arrays.
7. Add `for` and any other missing bits of source language.
8. Complete the static semantic analysis.

If you want to extend the task, various improvements are possible (and if one or more are done well, a bonus mark will apply):

- Add run-time checking for array bounds violations.
- Add optimisations. A simple peephole analysis could be quite effective. The Taz emulator will provide statistics if called with an '-s' option.
- Allow an assignment to copy an array as a whole, and similarly, allow an array to be passed as an argument to a procedure, by value and/or by reference.

The following are, or will soon be, available on the LMS, under "Programming Project":

- `taz_emulator.zip` contains the Taz emulator. The make file will generate an executable called `taz`.
- `parser.zip` contains a `Paz` parser which is believed to be correct.
- `milestones.zip` contains small `Paz` programs intended to support the milestones suggested above.
- `vis.zip` contains other small `Paz` programs for testing (currently intended for when we do the submit system's "visible" testing).
- `contrib.zip` will contain `Paz` programs as submitted by teams (once they have been submitted).

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 8 May, submit a single Paz 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.paz`, where `teamName` is your team's name, and submit a separate file `teamName.in` with the intended input to `teamName.paz`, if it requires input. For this stage, use `submit COMP90045 3a` to submit.

By 22 May, submit the code. There should be a `Makefile`, so that a `make` command generates `Paz`. If you use `alex` and/or `happy`, do not submit the Haskell files they generate (only the `alex`/`happy` specifications; your `Makefile` should generate the corresponding Haskell files. Also, do not submit `taz.c` or other files related to Taz. 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 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 70%) and programming structure, style, readability, commenting and layout (some 30%). Out of the correctness marks, 60% will be directed towards code generation, with scanning, parsing, semantic analysis and symbol table handling counting for the remaining 40%.

Appendix: Code format rules

Your Haskell programs should follow some 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 should 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, where clauses, etc., must be indented consistently. They can be indented using tabs or spaces, and can be indented 2, 3, 4, or 8 spaces, as long as it is done consistently. Beware that some of the scaffolding code may use spaces.
- Each program line should contain no more than 80 characters.

Nick Downing and Harald Søndergaard
27 April 2018