We propose to write a program that performs source-to-source translation (i.e. compilation) from the Tiger programming language into SPIM assembly code.

The Tiger language was introduced as an example in Andrew Appel’s Modern Compiler Implementation book. There is a partial definition presented as an appendix of his book, and since it has been eliminated from future editions, a complete definition has since been posted online.

SPIM is a simulated assembly language written for the RISC-based MIPS microprocessor architecture. There are simulators available for Windows and Linux systems.

Source-to-source translation

The source language we choose is Tiger, an example of which is provided below. It is, as any programming language, more or less human readable.

```
let
  function print_conditional (string : s, int : i) =
    if i then
      print (s)
    else
      print ("error")
  in
    print_string ("foo", 1)
end
```

The target language is SPIM assembly code. The language is well-documented, and there is a public-domain assembler/interpreter that we can use to test the validity of our project’s structured output. An example of SPIM follows.

```
.data
beq_: .asciiz "Testing BEQ and BEQL\n"
.text
  li $v0 4   # syscall 4 (print_str)
  la $a0 beq_
  syscall

  addiu $2 $0 0
  beq $0 $0 1020
  addiu $2 $0 1  # Delayed instruction
1020: addiu $3 $0 1
  bne $2 $3 fail
  nop
```
Name analysis

We will need to perform name analysis for variables, types and function names. These fall into two name spaces, one for type names and the other for variable names and function names. All three must be explicitly declared.

It is always legal to declare a new name in a nested scope that hides an existing one in the same name space, including hiding the predefined types. Tiger has only two predefined types, integer and string, and it also allows for definition of more named types in code.

Tiger uses regular static scoping similar to C, except for some special rules about recursive declarations. Tiger allows for recursive or mutually recursive declarations for types and functions, so long as the mutually recursive declarations are consecutive. The scope of each type or function identifier in a consecutive sequence of declarations starts at the beginning of the sequence, meaning that type is available for use in its own declaration or any other in the consecutive sequence.

The following example is mutually recursive, and the second line is also recursive to itself.

```plaintext
type tree = {key: int, children: treelist}
type treelist = {hd: tree, tl: treelist}
```

Structured Output

Now that we can build and decorate the source language AST, we need to generate the structured output: the SPIM assembly code.

A SPIM encoder is provided, and documentation is provided for it, hence one approach is to attribute our AST and generate the abstract target tree that can be fed into the encoder to generate the SPIM code for us.

Another option is to write our own PTG functions and do computations on the tree to output the SPIM code.

Given the time we have, we certainly prefer the first approach, which on the other hand is an even better representation of the domain specific concept.