Building a Compiler for SnuPL/0

In this project we implement a simple compiler for the SnuPL/0 language from scratch. Your compiler will compile SnuPL/0 source code to ARM assembly code.
SnuPL/0 is an imperative procedural language closely related to the Oberon programming language. It provides no support for object-orientation or composite types (arrays, records, enumerations), but is complex enough to illustrate the basic concepts of writing a compiler.

Here's a program in SnuPL/0 that computes the fibonacci number for a given input:

```
//
// fibonacci
//
module fibonacci;

var n: integer;

// fib(n): integer
// compute the fibonacci number of n. n >= 0
function fib(n): integer;
begin
  if (n <= 1) then
    return n
  else
    return fib(n-1) + fib(n-2)
  end
end fib;

begin
  n := Input();

  // loop until the user enters a number < 0
  while (n > 0) do
    Output(fib(n));
    n := Input()
  end
end fibonacci.
```

The project is split into six phases:

- scanning
- parsing
- type checking
- instruction selection
- data flow analysis
- register allocation

Instructions for the individual phases are handed out separately.

The SnuPL/0 Language

Syntax Definition of SnuPL/0

```
module            =  "module" ident ";" varDeclaration { subroutineDecl }
                     "begin" statSequence "end" ident ".".

letter            =  "A"..."Z" | "a"..."z" | "_".
digit             =  "0"..."9".

ident             =  letter { letter | digit }.
number            =  digit { digit }.
boolean           =  "true" | "false".
type              =  "integer" | "boolean".

factOp            =  "*" | "/" | "&&".
termOp            =  "+" | "-" | "||".
relOp             =  "=" | "#" | "<" | "<=" | ">" | ">=".

factor            =  ident | number | boolean |
                     "(" expression ")" | subroutineCall | "!" factor.
term              =  factor { factOp factor }.
simpleexpr        =  ["+"|"-"] term { termOp term }.
expression        =  simpleexpr [ relOp simplexpr ].

assignment        =  ident ":=" expression.
subroutineCall    =  ident "(" [ expression {"," expression} ] ")".
ifStatement       =  "if" "(" expression ")" "then" statSequence
                     [ "else" statSequence ] "end".
whileStatement    =  "while" "(" expression ")" "do" statSequence "end".
returnStatement   =  "return" [ expression ].

statement         =  assignment | subroutineCall | ifStatement |
                     whileStatement | returnStatement.
statSequence      =  [ statement { ";" statement } ].
varDeclaration    =  [ "var" { ident { "," ident } ":" type ";" } ].

subroutineDecl    =  (procedureDecl | functionDecl)
                     subroutineBody ident ";".
procedureDecl     =  "procedure" ident [ formalParam ] ";".
functionDecl      =  "function" ident [ formalParam ] ":" type ";".
formalParam       =  "(" [ident { "," ident } ] ")".
subroutineBody    =  varDeclaration "begin" statSequence "end".

comment           =  "//" {[^\n]} \n
whitespace        =  { " " | \t | \n }
```

Type System

SnuPL/0 supports two scalar types, integers and booleans. The types are not compatible, and there is no type casting.

The storage size, the alignment requirements and the value range are given in the table below:

Type	Storage Size	Alignment	Value Range
integer	4 bytes	4 bytes	-2^{31} .. $2^{31}-1$
boolean	1 byte	1 byte	true, false

The semantics of the different operations for the two types are as follows:

Operator	integer	boolean
+	binary: <integer> ← <integer> + <integer> unary: <integer> ← <integer>	*n/a*
–	binary: <integer> ← <integer> - <integer> unary: <integer> ← -<integer>	*n/a*
*	<integer> ← <integer> * <integer>	*n/a*
/	<integer> ← <integer> / <integer> rounded towards zero	*n/a*
&&	*n/a*	<boolean> ← <boolean> ∧ <boolean>
||	*n/a*	<boolean> ← <boolean> ∨ <boolean>
!	*n/a*	<boolean> ← ¬ <boolean>
=	<boolean> ← <integer> = <integer>	<boolean> ← <boolean> = <boolean>
#	<boolean> ← <integer> # <integer>	<boolean> ← <boolean> # <boolean>
<	<boolean> ← <integer> < <integer>	*n/a*
<=	<boolean> ← <integer> <= <integer>	*n/a*
=>	<boolean> ← <integer> => <integer>	*n/a*
>	<boolean> ← <integer> > <integer>	*n/a*

Parameter Passing and Calling Convention on ARM

All procedure/function arguments are passed by value. SnuPL/0 follows the Procedure Call Standard for the ARM Architecture as defined by ARM calling convention: three registers hold special values: r15 contains the program counter (PC), r14 the link register (LR), r13 is the stack pointer (SP). r0 to r3 are used to pass the first four parameters, parameters >= 5 are passed on the stack in ascending order. Functions return the function value in the r0 register. Registers r4 – r11, SP, and LR are callee-saved. The stack pointer must be aligned on a 4-byte boundary.

I/O

SnuPL/0 provides input/output through two implicitly defined functions/procedures:

- function Input(): integer
  reads and returns an integer value from stdin
- procedure Output(x);
  prints integer value ‘x’ to stdout

The compiler must assume that these two functions are defined. An implementation is provided and can simply be linked to the compiled code.