## Building a Compiler for SnuPL/1

The course project is to implement a simple compiler for the SnuPL/1 language from scratch. Your compiler will compile SnuPL/1 source code to 32-bit Intel assembly code.
SnuPL/1 is an imperative procedural language closely related to the Oberon programming language, one of the many languages developed by Prof. Niklaus Wirth. SnuPL/1 does not support object-orientation and the only composite data type supported are arrays (not records, enumerations). Nevertheless, SnuPL/1 is complex enough to illustrate the basic concepts of writing a compiler.

Here is a program written in SnuPL/1 that computes the fibonacci numbers for given inputs:

```
module fibonacci;

var n: integer;

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

begin
 Write("Enter a number: ");
 n := ReadInt();

 // loop until the user enters a number < 0
 while (n > 0) do
  Write("Result: "); WriteInt(fib(n)); WriteLn;

  Write("Enter a number: ");
  n := ReadInt()
 end

end fibonacci.
```

Writing a compiler is difficult. We will implement the compiler in the following phases:

– lexical analysis (scanning)
– syntax analysis (parsing)
– semantic analysis (type checking)
– intermediate code generation
– code generation

Instructions for the individual phases are handed out separately.

# The SnuPL/1 Language

## EBNF Syntax Definition of SnuPL/1

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

letter           = "A”.."Z" | "a".."z" | "_".
digit            = "0".."9".
character        = printable ASCIIchar | "\n" | "\t" | "\"" | "\'" | "\\" | "\0"
char             = "'" character "'"
string           = '"' { character } '"'.

ident            = letter { letter | digit }.
number           = digit { digit }.
boolean          = "true" | "false".
type             = basetype | type "[" [ number ] "]".
basetype         = "boolean" | "char" | "integer".

qualident        = ident { "[" expression "]" }.
factOp           = "*" | "/" | "&&".
termOp           = "+" | "-" | "||".
relOp            = "=" | "#" | "<" | "<=" | ">" | ">=".

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

assignment       = qualident ":=" 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" varDeclSequence ";" ].
varDeclSequence  = varDecl { ";" varDecl }.
varDecl          = ident { "," ident } ":" type.

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

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

## Type System

### Scalar types

SnuPL/1 supports three scalar types: booleans, characters, and integers. 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 |
|---|---|---|---|
| boolean | 1 byte | 1 byte | true, false |
| char | 1 byte | 1 byte | ASCII characters (0..255) |
| integer | 4 bytes | 4 bytes | $-2^{31} .. 2^{31}-1$ |

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

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

Scalar types are not compatible with each other. No type conversion/casting is possible.

### Array types

SnuPL/1 supports multidimensional arrays of scalar types. The declaration of the array requires the dimensions to be specified as constants such as in

```
var a : integer[128];
  b : integer[16][128];
  c : integer;
```

The valid index range is from 0 to N-1. Dereferencing array variable is achieved by specifying the indices in brackets:

```
c := a[8];
c := b[1][127];
a := b[7];
```

In parameter definitions, open arrays are allowed as follows:

```
procedure WriteLn(str: char[]);
procedure foo(m: integer[][]);
```

This allows passing of arrays with matching base type and dimensions:

```
procedure bar(a: char[]);
procedure foo(b: integer[][]);

var s: char[128];
  t: char[12][12];
  m: integer[16][16][16];
  n: integer[5][5];

begin
 bar(str);         // valid
 foo(n);           // valid
 foo(m);           // invalid: dimension mismatch
 foo(m[1]);        // valid: pass m[1] as integer[][]
 foo(t);           // invalid: base type mismatch
end
```

The dimensions of open arrays can be queried using DIM(array, dimension) (refer to "Predefined Procedures and Functions" below.)

```
procedure print(matrix: integer[][]);
var i,j,N,M: integer;
begin
  N := DIM(matrix, 1);
  M := DIM(matrix, 2);

  for i := 0 to N-1 do begin
    for j := 0 to M-1 do begin
      WriteInt(matrix[i][j]); WriteChar('\t)
    end;
    WriteLn()
  end
end print;
```

Support for open arrays and at-runtime querying of array dimensions requires the implementation of arrays to carry the necessary information (i.e., number of dimensions and size per dimension). You are free to choose a memory layout that suits your needs.

Characters and Strings

The scalar char data type represents a single character. Strings are implemented as (constant) character arrays and are null-terminated.

SnuPL/1 supports the following escape sequences in the context of characters and strings.

| Escape sequence | Character | Remarks |
|---|---|---|
| \n | newline | |
| \t | tabulator | |
| \0 | NULL character | string termination character |
| \" | double quote | necessary only within double quotes |
| \' | single-quote | necessary only within single quotes |
| \\ | literal '\' | |

Immutable string constants can be used in lieu of character arrays as follows:

```
begin
  WriteLn("Hello, world!")
end
```

## Parameter Passing and Calling Convention

Scalar arguments are passed by value, array arguments by reference.

The calling convention for the various backends differ by architecture; for IA32 we follow the System V ABI for Intel386 Architectures. IA32 has eight general-purpose 32-bit registers: %eax, %ebx, %ecx, %edx, %esi, %edi, %esp, and %ebp. %ebp, %esi, and %edi are callee-, %eax, %ecx, and %edx are caller-saved. %esp and %ebp point to the current stack frame. Parameters are passed on the stack in reverse order, results returned in %eax.

## Predefined Procedures and Functions

The following procedures and functions are pre-defined (i.e., your compiler must be able to deal with them without throwing an unknown identifier error).

### Open arrays

The functions DIM/DOFS are used to deal with open arrays. The functionality can be implemented directly into the compiler or as an external library.

- function DIM(array: pointer to array; dim: integer): integer;
  returns the size of the 'dim'-th array dimension of 'array'.
- Function DOFS(array: pointer to array): integer;
  returns the number of bytes from the starting address of the array to the first data element.

  Example usage is provided above (Type System – Array Types)

### I/O

The following low-level I/O routines read/write integers, characters, and strings. An implementation is provided and can simply be linked to the compiled code.

- function ReadInt(): integer
  read and return an integer value from stdin.
- procedure WriteInt(i: integer);
  print integer value ‘i’ to stdout.
- procedure WriteChar(c: char);
  write a single character to stdout.
- procedure WriteStr(string: char[]);
  write string ‘string’ to stdout. No newline is added.
- procedure WriteLn()
  write a newline sequence to stdout.