The SnuPL/1 Language

SnuPL/1 is an imperative procedural language developed to teach Compiler Construction and Advanced Compiler Construction by the Computer Systems and Platforms Laboratory at the Dept. of Computer Science and Engineering, Seoul National University. SnuPL/1 is 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.

The following lists 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.
```

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" | "\"" | "\'" | "\\"
char             = "'" character  | "\0" "'"
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> unary: <int> ← <int>
-	*n/a*	*n/a*	binary: <int> ← <int> - <int> unary: <int> ← -<int>
*	*n/a*	*n/a*	<int> ← <int> * <int>
/	*n/a*	*n/a*	<int> ← <int> / <int> 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 an 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) (see "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; we may provide one possible implementation if needed.

Characters and Strings

The `char` data type represents a single character. Strings are implemented as (constant) character arrays and are null-terminated. Computations are not allowed, i.e., unlike C, the `char` datatype is not a numerical character type. Relational operators are allowed on characters. The order of the characters follows the ASCII standard (https://en.wikipedia.org/wiki/ASCII). You are free to decide whether you limit yourself to the 7-bit ASCII charset or allow 8-bit ASCII characters.

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	only allowed in character constants
\"	double quote	necessary only within double quotes
\'	single-quote	necessary only within single quotes
\\	literal '\'	

Printable ASCII characters are ASCII characters between 0x32 (“ ”, space) and 0x7f (7-bit ASCII) or 0xff (8-bit ASCII) *except* 0x7f (delete character). While backslash (\), double quotes ("), and single quotes (') are printable characters, special rules apply:

- backslash always has to be escaped
- double quotes must be escaped in a string and can be escaped in a character constant
- single quotes must be escaped in a character constant and can be escaped in a string

The double/single quotes are not part of the string/character constant, but part of the syntax.

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. %ebx, %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.

For x86_64 (AMD64) we follow the [System V ABI for AMD64 Architectures](). AMD64 has sixteen general-purpose 64-bit registers: %rax, %rbx, %rcx, %rdx, %rsi, %rdi, %rsp, %rbp, and %r8-%r15. %rbx, %rbp, and %r12-%r15 are callee-, %rax, %rcx, %rdx, %rdi, %rsi, and %r8-%r11 are caller-saved. %rsp points to the current stack frame. The first six parameters are passed in %rdi, %rsi, %rdx, %rcx, %r8, and %r9, from the 7^{th} parameter on the stack is used; results returned in %rax. Refer to the ABI chapter 3.2.3 for details.

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. Implementation for IA32 and x86_64 are provided in `rte/[IA32,AMD64]` and can 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.