Building a Compiler for SnuPL/1

The term 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. Records or enumerations are not supported. Nevertheless, SnuPL/1 is complex enough to illustrate the basic concepts of 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
  WriteStr("Enter a number: ");
  n := ReadInt();

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

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

end fibonacci.
```

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

SnuPL/1 Language Specification

EBNF Syntax Definition of SnuPL/1

The SnuPL/1 EBNF defines the syntax of the language. Semantical restrictions are provided on the following pages.

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

letter            = "A".."Z" | "a".."z" | "_".
digit             = "0".."9".
hexdigit          = digit | "A".."F" | "a".."f".
character         = LATIN1 char | "\n" | "\t" | "\"" | "\'" | "\\" | hexencoded.
hexedcoded        = "\x" hexdigit hexdigit.
char              = "'" character  | "\0" "'".
string            = '"' { character } '"'.

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

qualident         = ident { "[" simpleexpr "]" }.
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 } ].
```

```
constDeclaration   = [ "const" constDeclSequence ].
constDeclSequence  = constDecl ";" { constDecl ";" }
constDecl          = varDecl "=" expression.

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     = constDeclaration varDeclaration
                     "begin" statSequence "end".

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

Type System

Scalar types

SnuPL/1 supports three scalar types: boolean, character, and integer types. 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	ISO 8859-1 characters
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
		Arithmetic operations	
+	*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
		Logical operations	
&&	<bool> ⟵ <bool> ∧ <bool>	*n/a*	*n/a*
||	<bool> ⟵ <bool> ∨ <bool>	*n/a*	*n/a*
!	<bool> ⟵ ¬ <bool>	*n/a*	*n/a*
		Equality and relational operations	
=	<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. Type conversion/casting is not supported.

Array types

SnuPL/1 supports multidimensional arrays of scalar types. The declaration of the array requires the dimensions to be specified as *simpleexpr* that can be evaluated to an integer value at compile-time:

```
const N : integer = 1024;
var a : integer[128];
    b : integer[N][12*12];
```

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

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

In parameter definitions, open arrays are allowed as follows:

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

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);

  i := 0;
  while (i < N) do
    j := 0;
    while (j < M) do
      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.

One possible implementation is provided with the reference compiler; however, that implementation stores the number of dimensions and the size of each dimension at the beginning of the array. As a consequence, it is impossible to pass sub-arrays as full arrays because the necessary array meta-data is missing. Consider:

```
procedure foo(b: integer[][]);

var n: integer[5][5];
    m: integer[16][16][16];

begin
  foo(n);        // valid
  foo(m);        // invalid: dimension mismatch
  foo(m[1])      // not supported by reference implementation even though technically correct
end
```

Characters and Strings

The 8-bit `char` data type represents a single character. Strings are implemented as (constant) character arrays and are null-terminated. SnuPL/1 characters and strings support the ISO-8859-1 standard (latin1). Non-printable characters (0 – 31 and 0x7f) must be escaped. Internally, all unprintable and characters ≥ 0x80 are escaped. The following escape sequences are defined:

Escape sequence	Character	Remarks
`\n`	newline	
`\t`	tabulator	
`\0`	NULL	not allowed within strings
`\"`	double quote	necessary only within double quotes
`\'`	single-quote	necessary only within single quotes
`\\`	literal '\'	
`\xHH`	ASCII character HH	specified in hexadecimal notation

The NULL character (\0) is only allowed in character constants but not within strings. Special rules apply for the printable characters backslash (\), double quotes ("), and single quotes ('):

- Backslash always has to be escaped
  Example: `"This is a backslash: \\"`
- Double quotes must be escaped in a string and can be escaped in a character constant
  Example: `"She said, \"That's what he said.\""`
  Example: `c = '"'; c = '\"';`
- Single quotes must be escaped in a character constant and can be escaped in a string
  Example: `c = '\'';`
  Example: `"To which he replied, \'That's what she said!'"`

The enclosing quotes are not part of the string/character constant itself.

Computations with variables of datatype `char` are not allowed , i.e., unlike C, the `char` datatype is not a numerical character type. Relational operators are allowed, however; the order of the characters follows the ISO-8859-1 standard (for a free version, refer to this page).

Hint: to save files in the latin1 character encoding from within VIM, use

```
:w ++enc=latin1 <filename>
```

Assignments to Compound Types

The reference implementation of SnuPL/1 does not support assignments to arrays. You are free to lift this limitation in your implementation.

Example:

```
var c: char[32];
c := "Hello, world!";
```

results in

```
parse error: assignments to compound types are not supported.
```

Symbolic constants

SnuPL/1 allows the specification of symbolic constants for notational convenience.

```
module constants;

const
  K: integer = 1024;
  M: integer = 1024 * K;
  Message: char[] = "Size converter:";

procedure PrintSize(size: integer);
const a,b,c: integer = 1;
      K    : integer = 1000;
var i,j,k: integer;
begin
  WriteStr(Message); WriteLn();
  WriteStr("Size: ");   WriteInt(size);
  WriteStr("bytes = "); WriteInt(size / K);
  WriteStr("KB = ");    WriteInt(size / M);
  WriteStr("MB"); WriteLn()
end PrintSize;

begin
  PrintSize(1111)
end constants.
```

The following rules apply when defining symbolic constants:

- Symbolic constants are typed and defined by a constant expression.
- The constant expression must only contain constant values and previously defined symbolic constants.
- The type of the constant and the expression must match.
- Variables and constants share the same name space, i.e., constant and variable names must be unique within the same scope.
- Array constants are not supported with the exception of constant character arrays (i.e., strings). The size of the constant array is computed from the length of the constant string.

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)

Input/Output

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.

Examples are provided throughout the text in this document.

Parameter Passing and Backend ABIs

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

IA32 Backend

The IA32 backend follows the System V ABI for Intel386 Architectures. Parameters are passed on the stack in reverse order, results returned in %eax.

Register	Function	Save	Register	Function	Save
`eax`	g-p, return value	caller	`esi`	general-purpose	callee
`ecx`	general-purpose	caller	`edi`	general-purpose	callee
`edx`	general-purpose	caller	`esp`	stack pointer	callee
`ebx`	general-purpose	callee	`ebp`	frame pointer	callee