https://folk.ntnu.no/michaeng/tdt4205_21/
michael.engel@ntnu.no

Practical exercises
Spring 2021

# Practical Exercises 3
## VSL scanning and parsing

**Please submit solutions on Blackboard by Friday, 05.03.2021 14:00h**

**Notice:** Please submit solutions on Blackboard in groups of two or three students.
The practical exercises will be graded and count as part of your final grade.

In this practical exercise, we will construct the scanner and parser of our compiler. Please note that this and the following practical exercises are more extensive than the first two, so please plan ahead in time so you can submit your code before the deadline.

## 3.1 VSL Specification

The directory in the code archive `PE3_skeleton.zip` contains code for the starting point of a compiler for a slightly modified 64-bit version of VSL ("Very Simple Language"), defined by Jeremy Bennett (Introduction to Compiling Techniques, McGraw-Hill, 1990).

Its lexical structure is defined as follows

- Whitespace consists of the characters '\t', '\n', '\r', '\v' and ' '. It is ignored after lexical analysis.
- Comments begin with the sequence '//', and last until the next '\n' character. They are ignored after lexical analysis.
- The following strings are **reserved words**:
  - **def** – function definition
  - **begin** – start of a function or block
  - **end** – end of a function or block
  - **return** – exit from a function
  - **print** – print to screen
  - **if then else** – conditions
  - **while do continue** – loop control
  - **var** – variable declaration
- Basic operators are assignment (:=), the arithmetic operators '+', '-', '*', '/' and relational operators '=', '<', '>'.
- In addition, these are the bitwise operators: '<<' (leftshift), '>>' (rightshift), '~' (NOT), '&' (AND), '^' (XOR) and '|' (OR).
- Numbers are sequences of one or more decimal digits ('0' through '9').
- Strings are sequences of arbitrary characters other than '\n', enclosed in double quote characters '"'.
- Identifiers are sequences of at least one letter followed by an arbitrary se- quence of letters and digits. Letters are the upper- and lower-case English alphabet ('A' through 'Z' and 'a' through 'z'), as well as underscore ('_'). Digits are the decimal digits, as above.

The syntactic structure is given in the context-free grammar on the last page of this document.
Building the program supplied in the archive `PE3_skeleton.zip` combines the contents of the `src/` subdirectory into a binary `src/vslc` which reads the standard input and produces a parse tree.
The structure in the `vslc` directory will be similar throughout subsequent problem sets, as the compiler takes shape.

a. **Scanner** (2 points)

   Complete the lex scanner specification in `src/scanner.l` so that it properly tokenizes VSL programs.

b. **Tree construction** (4 points)

   A `node_t` structure is defined in `include/ir.h`. Complete the auxiliary functions `node_init` and `node_finalize` so that they can initialize/free `node_t`-sized memory areas passed to them by their first argument.

   The function `destroy_subtree` should recursively remove the subtree below a given node, while `node_finalize` should only remove the memory associated with a single node.

c. **Parser** (4 points)

   Complete the yacc parser specification to include the VSL grammar, with semantic actions to construct the program's parse tree using the functions implemented above.

   The top-level production should assign the root node to the globally accessible `node_t` pointer 'root' (declared in `src/vslc.c`).

***Hint:*** To get an idea of the structure of a VSL program, you can find example programs in the `vsl_programs/` directory. This is an example program (keywords are highlighted in bold):

```
// Approximate square root by the Newton/Raphson method for f(x) = x^2 - n
// f(x) = x^2 - n = 0
// f'(x) = 2x
// xn+1 = xn - (x^2-n) / 2x

def newton ( n )
begin
    print "The square root of ", n, " is ", improve ( n, 1 )
    return 0
end

def improve ( n, estimate )
begin
    var next
    next := estimate - ( (estimate * estimate - n) / ( 2 * estimate ) )
    if next - estimate = 0 then
        // Integer precision converges at smallest int greater than the square
        return next-1
    else
        return improve ( n, next )
end
```

**VSL grammar:**

```
program          → global_list
global_list      → global | global_list global
global           → function | declaration
statement_list   → statement | statement_list statement
print_list       → print_item | print_list ',' print_item
expression_list  → expression | expression_list ',' expression
variable_list    → identifier | variable_list ',' identifier
argument_list    → expression_list | ε
parameter_list   → variable_list | ε
declaration_list → declaration | declaration_list declaration
function         → def identifier '(' parameter_list ')' statement
statement        → assignment_statement | return_statement
                  | print_statement | if_statement
                  | while_statement | null_statement | block
block            → begin declaration_list statement_list end
                  | begin statement_list end
assign_statement → identifier ':=' expression
return_statement → return expression
print_statement  → print print_list
null_statement   → continue
if_statement     → if relation then statement
if_statement     → if relation then statement else statement
whilestatement   → while relation do statement
relation         → expression '=' expression
                  | expression '<' expression
                  | expression '>' expression
expression       → expression '|' expression
                  | expression '^' expression
                  | expression '&' expression
                  | expression '<<' expression
                  | expression '>>' expression
                  | expression '+' expression
                  | expression '-' expression
                  | expression '*' expression
                  | expression '/' expression
                  | '-' expression
                  | '~' expression
                  | '(' expression ')'
                  | number | identifier | identifier '(' argument_list ')'
declaration      → var variable_list
printitem        → expression | string
identifier       → IDENTIFIER
number           → NUMBER
string           → STRING
```