# Compiler Project #1 - Scanner

2012-11269 Dongjae Lim
2013-11403 Chansu Park

## New token system

Because SnuPL/1 has more complex syntax than SnuPL/-1, we needed to build a new token system which contains more tokens. Here are the tokens in our scanner:

| | |
|---|---|
| **Data Types** | `tNumber, tIdent, tCharacter, tString, tBool` |
| **Operators** | `tPlusMinus, tMulDiv, tRelOp, tAndOr, tNot, tAssign` |
| **Special characters** | `tSemicolon, tComma, tDot, tLParen, tRParen, tLBracket, tRBracket` |
| **Keywords** | `tModule, tBegin, tEnd, tIf, tThen, tWhile, tDo, tReturn, tVar, tProcedure, tFunction, tBoolean, tChar, tInteger` |
| **Comments** | `tComment` |
| **Special tokens** | `tEOF, tIOError, tUndefined` |

## Scanning logics for new tokens

Here are summaries of scanning logics for each tokens, except operators and special characters: some of special characters are already implemented in SnuPL/-1 and other special characters also statisfies that logic, except a few of them.

- Numbers and identifiers

They are implemented just as its definitions. If the scanner meets a digit, it consumes all consecutive digits and makes them to a number token. In the same way, the scanner makes an identifier token if it meets a letter.

```
1234abc1234 → tNumber (1234), tIdent (abc1234)
```

- Keywords

There is a keyword-token map in the scanner. After a scan for an identifier token, the scanner finds the token in the map and returns a keyword token instead of an identifier token if it is found. Or, if the variable is already defined as an identifier, keyword-token map returns that variable as `tIdent` token.

```
var a ... → tVar, tIdent (a) (newly saved into keyword-token map), ...
a = 5;    → tIdent (a) (found from keyword-token map),
            tRelOp (=), tNumber (5), tSemicolon
```

- Characters and strings

If the scanner meets a quote/double quote, **it consumes all characters until it meets a closing quote/double quote**. Then it checks wheter or not it contains only valid character. Additionally, scanner also checks whether or not the character contains several - 0 or more than one - characters. If the given character/string is valid, the scanner makes an identifier of character/string. Otherwise, scanner makes an identifier as undefined.

```
'\t'          → tCharacter (\t)
"abcd\nefgh" → tString (abcd\nefgh)
```

Since CToken::escape escapes only valid escape characters, if the input contains invalid escape character, the scanner ignores the backslash and return an identifier as undefined.

```
'ax', "\x"   → tUndefined (ax), tUndefined(x)
"abcd
efgh"        → tUndefined (abcd\nefgh)
```

Especially, if the input ends before quote or double quote closes character/string, the scanner cuts the scanning loop and makes an `tUndefined` identifier.

```
"ssss        → tUndefined (ssss\n)
```

Note that when SnuPL/1 scans .mod file, it recognizes `EOF` at the first character of the next line of the last line of the file. Therefore the line feed must be occured at the end of the file. Furthermore, in this example, since the opened string is not closed by a double quote, scanner reads the line feed as an invalid character of the string because it is not escaped, and then it recognizes `EOF`.

- Comments

The comment characters("`//`") contains a division sign, so it would better scan division sign and comments both. When a division sign occurs, the scanner checks next character. If the next character is also same, the scanner consume characters until the line feed and returns a comment token. The token is unnecessary for compile at all, but we added this token just to check our scanner's handling for comments.

- Error Recovery

As we wrote several times above here, if the token is recognized as undefined, scanner returns tUndefined token with invalid token value.

## Other Fixes

- Modification in `CToken::escape`

In the previous implementation of escape function, characters following a null character('\0') are ignored. We thought this cannot provide correct result for some tests, so we had to modify it to be able to accept characters over a null character.
To do that, we changed "`while(*t != '\0'){`" … into "`for (char c : text){`" … and adapted other lines to this line, which are in `CToken::escape`. After we changed, it takes null character properly, and continue to read following characters in the string text.

```
"abcd\0efgh" → tString (abcd\0efgh)
```

We assumed that the parser we will make will recognize that `\0` is in `tokval` of that token, and it will eliminate the rest part of that `tokval`.

## Project Location

We made github repository for this whole compiler project, so any people can see our project on this URL:

https://github.com/hhosu107/Compiler.

Also we uploaded some simple .mod files to test other cases for our scanner, which is located in /Projects/1.Scanning/snuplc/test. We also uploaded the test results for our implementation.

p.s.)
We appreciate Hyeongmo Kim & Pyeongseok Oh's syntax file for .mod files.

https://github.com/cseteram/Eggompilers/tree/scanning/vim_syntax/mod.vim

To use it, deploy this source code in the folder ~/.vim/syntax, and make ~/.vim/ftdetect/mod.vim to detect .mod file as a language source file.