Lexical Analysis

Sungbin Jo (2021-13630)

Implementation notes

New tokens

Nothing to see here, just boring token additions.

```
enum EToken {
  tNumber = 0,  ///< a number
  tIdent,       ///< an identifier
  tPlusMinus,   ///< '+' or '-'
  tMulDiv,      ///< '*' or '/'
  tAnd,         ///< '&&'
  tOr,          ///< '||'
  tNot,         ///< '!'
  tRelOp,       ///< relational operator
  tAssign,      ///< assignment operator
  tColon,       ///< a colon
  tSemicolon,   ///< a semicolon
  tDot,         ///< a dot
  tComma,       ///< a comma
  tLParen,      ///< a left parenthesis
  tRParen,      ///< a right parenthesis
  tLBrack,      ///< a left bracket
  tRBrack,      ///< a right bracket

  tBoolean,  ///< 'boolean'
  tChar,     ///< 'char'
  tInteger,  ///< 'integer'
  tLongInt,  ///< 'longint'

  tConst,      ///< 'const'
  tVar,        ///< 'var'
  tExtern,     ///< 'extern'
  tProcedure,  ///< 'procedure'
  tFunction,   ///< 'function'
  tModule,     ///< 'module'
  tBegin,      ///< 'begin'
```

```
  tEnd,        ///< 'end'

  tIf,       ///< 'if'
  tThen,     ///< 'then'
  tElse,     ///< 'else'
  tWhile,    ///< 'while'
  tDo,       ///< 'do'
  tReturn,   ///< 'return'

  tBoolConst,    ///< boolean constant
  tCharConst,    ///< character constant
  tStringConst,  ///< string constant

  tEOF,             ///< end of file
  tIOError,         ///< I/O error
  tInvCharConst,    ///< invalid char constant
  tInvStringConst,  ///< invalid string constant
  tUndefined,       ///< undefined
}
```

Handling comments

The initial implementation was to call and return `Scan()` in the big switch-case statement, but it felt brittle that `Scan()` now has to be re-entrant and to be careful not to introduce any cleanup for `Scan()`. The current implementation now skips comments before the switch while reading the first character.

Handling invalid char and string literals

To avoid cascading errors in the case of invalid string literals, the scanner tries to find a known valid recovery point. The language spec doesn't allow multiline string literals, so the scanner can just skip characters until it finds a closing quote or a newline.

Avoiding cascading errors in multichar char literals are harder because the scanner can't determine if it's in a middle of a multichar char literal or if it's just missing a closing quote and the next character is a valid token. The current implementation just gives up on the first error, tries to consume a closing quote and assume that it's now normal state again. This unfortunately causes spurious errors when the code contains a multichar char literal.

Notable differences from the provided code

Addition of `TryChar()`

The simple helper function `TryChar()` was introduced for cases when the return value of `GetChar()` is not needed after testing `PeekChar()` for a specific character.

This is usually only the case when handling token delimiters that don't get included in the value.

Return values of `GetChar()` and `PeekChar()`

The functions `GetChar()` and `PeekChar()` are used throughout the scanner without checking whether `_in->good()` or not. Because the standard doesn't provide any guarantees on the value of `eof` except that it is negative (though most platforms set them to -1), theoretically comparing the returned `unsigned char` value with any character can succeed even when the stream is returning an `eof` value.

To avoid confusion between real characters and `eof`, `GetChar()` and `PeekChar()` are modified to return an `int`. The return value should only be assigned to `unsigned char` if `_in->good()` is guaranteed.

Rename of `t*Brak` tokens

The provided code handles parens with `t*Brak` tokens, while the current implementation handles them with `t*Paren` tokens. To avoid bugs while merging provided code in the future, the bracket tokens were deliberately named differently to cause compile errors.

Random notes to future self

Avoiding infinite loops on `eof` or I/O errors

If the while loop condition is negative (e.g. consume until `PeekChar()` is not one of these characters), always check `_in->good()` in the condition as well to avoid infinite loops.

Testing regressions

Committing the output of test cases makes detecting whether some code change resulted in a regression or not easier. This should probably go into a top-level Makefile and a git commit hook.

```
for tf in test/scanner/test*.mod; do
    snuplc/test_scanner "$tf" > currrent/scanner/"$(basename "${tf%.mod}.stdout")";
done
```