0. Abstract

The compiler for SnuPL/1 consists of five phases: scanning, parsing, semantic analysis, intermediate code generation, and code generation. Currently, only the first phase is implemented. The implemented phase is to be explained below.

1. Scanner

The scanner converts a character stream to a stream of tokens. A token indicates the detected lexeme of SnuPL/1 in a specific position of the character stream. A lexeme represents the basic meaningful unit in SnuPL/1. Custom tokens were defined according to the SnuPL/1 EBNF and were used within the implementation of the scanner. The scanner consists of two C++ classes: CToken, and CScanner. A CToken instance represents a token with all the necessary information such as the token's actual string value. A CScanner instance represents the scanner itself. It repeatedly gets characters from an input stream and turns them into a stream of tokens.

1.1 Tokens

Tokens are enumerated as *ETokens* in *scanner.h*. To match the SnuPL/1, the enumeration was modified as follows.

```
//----------------------------------------------------------------------------
///
/// @brief SnuPL/1 token type
///
/// Each member of this enumeration represents a token in SnuPL/1.
///
enum EToken {
  tDigit=0,                     ///< a digit
  tRelOp,                       ///< relational operator
  tAssign,                      ///< assignment operator
  tColon,
  tSemicolon,                   ///< a semicolon
  tDot,                         ///< a dot
  tComma,                       ///< a comma
  tLParen,                      ///< a left parenthesis '('
  tRParen,                      ///< a right parenthesis ')'
  tLBrak,                       ///< a left bracket '['
  tRBrak,                       ///< a right bracket ']'

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

  tModule,                      ///< keyword "module"
  tProcedure,                   ///< keyword "procedure"
  tFunction,                    ///< keyword "function"
  tVar,                         ///< keyword "var"
  tConst,                       ///< keyword "const"
```

```
  tInteger,                     ///< keyword "integer"
  tBoolean,                     ///< keyword "boolean"
  tChar,                        ///< keyword "char"
  tBegin,                       ///< keyword "begin"
  tEnd,                         ///< keyword "end"
  tIf,                          ///< keyword "if"
  tThen,                        ///< keyword "then"
  tElse,                        ///< keyword "else"
  tWhile,                       ///< keyword "while"
  tDo,                          ///< keyword "do"
  tReturn,                      ///< keyword "return"
  tTrue,                        ///< keyword "true"
  tFalse,                       ///< keyword "false"

  tIdent,                       ///< an identifier

  tFactOp,                      ///< a factor operation '*', '/', or '&&'
  tTermOp,                      ///< a term operation '+', '-', or '||'
  tNot,                         ///< boolean negation '!'

  tCharConst,                   ///< a character constant
  tStringConst,                 ///< a string constant
};
```

Of these, the new additions can be grouped as:

1. *tLParen, tRParen*
2. *tModule, tProcedure, tFunction, tVar, tConst, tInteger, tBoolean, tChar, tBegin, tEnd, tIf, tThen, tElse, tWhile, tDo, tReturn, tTrue, tFalse*
3. *tIdent*
4. *tFactOp, tTermOp,*
5. *tNot*

The first group represents parentheses, or the character ‘(‘ and ‘)’. As the name suggests, *tLParen* represents ‘(‘ and *tRParen* represents ‘)’. The second group consists of the keywords of SnuPL/1. Each keyword token’s name corresponds to which keyword it represents. For example, *tModule* would represent the keyword “module”. The third group consists of only one member, *tIdent*. *tIdent* represents identifiers that follow the SnuPL/1’s rule: letter { letter | digit }. How identifiers are differentiated with keywords will be explained in section 1.2

1.2 Keywords vs. Identifiers

Keywords are predefined set of strings that are reserved. Each keyword has a fixed, specific meaning. All identifiers should not be equivalent to any element in the keyword set. Therefore, it is crucial that the scanner matches keywords well. Matching keywords well means that the scanner should never mistakenly take a keyword as an identifier. In order to implement this safely, Predefined pairs of a keyword and a corresponding token are stored as a global variable. When the CScanner instance is initialized, the predefined pairs are stored in the instance’s map type member *keywords*, and when a sequence of alphanumeric characters and underscores

that is not a part of a string constant is met, the sequence is scanned until it meets a non-alphanumeric and non-underscore character. Then, the scanner looks the sequence in the map *keywords*. If there is a match, the corresponding token is returned. If there isn't a match, then the sequence is treated as an identifier and *tIdent* with the token value of the sequence is returned.

1.3 Comments

SnuPL/1 compiler must ignore whitespaces and comments. Ignoring whitespaces was pre-implemented. And for comments, The scanner uses a lookahead of one character when it meets a slash ('/'), to see if there is another slash that comes after. When this is the case, the scanner flushes the stream until a newline character is met, which would be the end of the single line comment.

1.4 Escape Sequences

Escape sequences should be well detected in character constants and string constants. Upon meeting a backslash ('\') while scanning for character constants and string constants, the scanner tries to match the next character with 'n', 't', '\', ''', '"', '0', or 'x'. For each character, a newline character, a tab, a backslash, a single quote, a double quote, and a null character is set as or appended to the token value, respectively. For 'x', the scanner tries to match the next two characters with hexdigits, and calculates the value denoted by the characters and appends it to the token value.