Scanner implementaion report

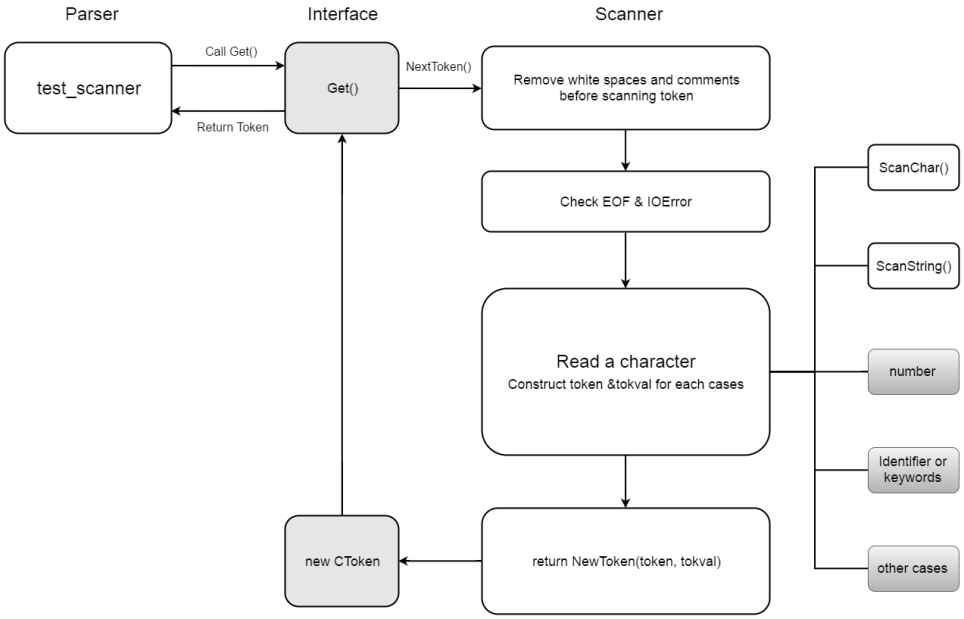
2013-11394 김형모

2013-11412 오평석

1 Development environment

- OS: Ubuntu 14.04.4 LTS
- Compile option: `g++ -std=c++0x -g -O0`
- Version control: Github

2 Structure



3 Details

3.1 Token implementation

We modified token structure —EToken— to expand SnuPL/-1. We developed `tDigit` and `tLetter` into `tNumber` and `tIdent`. We added `tChar`, `tString`, `tAndOr`, `tNot` to represent new operands/operators. `tLBrak` and `tRBrak` is devided into `tLBrak`, `tRBrak`, `tLParen`, `tRParen` to support array brakets.

We implements tokens for each keywords. Our naming convention made the character 'k' precedes the keyword name. For example, keyword 'function' is encoded into `kFunction`. We also treated 'true' and 'false' like a keyword, so they are encoded into `kTrue` and `kFalse`.

3.2 Number implenmentation

`token = tNumber, tokval = ''digits''`

We splited continuous digit chunk and converted it to number. When the scanner faces digit(and scanner does not scanning string or character or identifier at that time), it consumes input until there is no countinuous digit anymore.

```
if (IsDigit(c)) { // face digit
  token = tNumber;
  while (_in->good()) { // consume digit repeatedly
    char nc = _in->peek();
    if (!IsDigit(nc)) break; // done
    tokval += GetChar();
  }
}
```

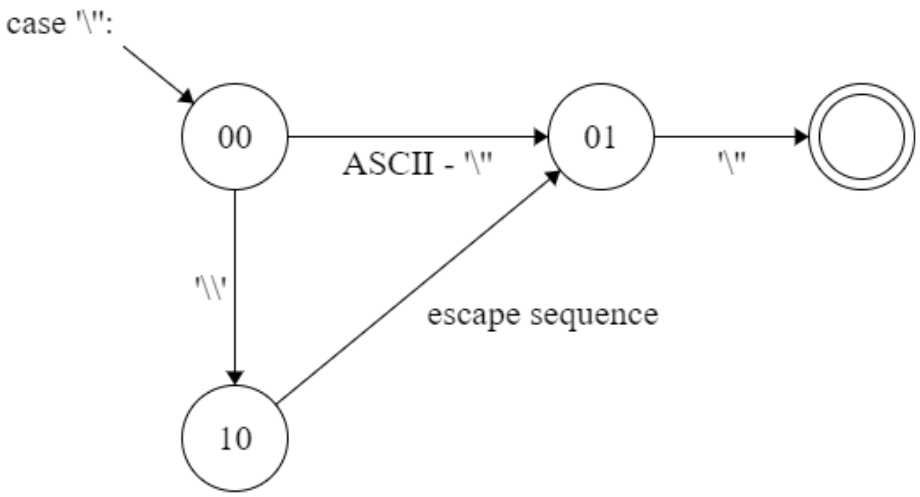
3.3 Character implementation

`token = tChar, tokval = ‘‘character’’`

We read one or two characters behind a single quotation, and then looking for another single quotation. For the precise checking case, we adopted two flag variables : `faced_escape, waiting_quot`. Initial value is false.

`faced_escape` : expect to face escape sequence.

`waiting_quot` : expect to face closing single quotation.



state encoding : [faced_escape:waiting_quot]

```
if (c == EOF) break;
```

Terminate scanning when faced EOF.

```
else if (waiting_quot) {
  if (C == ‘\’’) {
    token = tChar;
    tokval += GetChar();
```

```
  }
  break;
}
```

When the scanner encounterd single quot and it was waiting for it, assign `tChar` to token.

```
else if (!IsAsciiChar(c)) {
  tokval += GetChar();
  break;
}
```

Terminate scanning when faced undefined input character.

```
else if (c == '\\' && !faced_escape) {
  faced_escape = true;
  tokval += GetChar();
}
```

When ther scanner encounterd backslash in the case : `faced_escape` is `false`, set `faced_escape` to `true`.

```
else if (faced_escape) {
  faced_escape = false;
  waiting_quot = true;
  tokval.pop_back();

  bool valid = true;
  switch (c) {
```

```
    ...
  }
```

Construct escape sequence if `faced_escape` is `true`.

```
else if (c == '\'') {
  tokval += GetChar();
  break;
}
```

Handle the empty character exception. i.e. ‘’

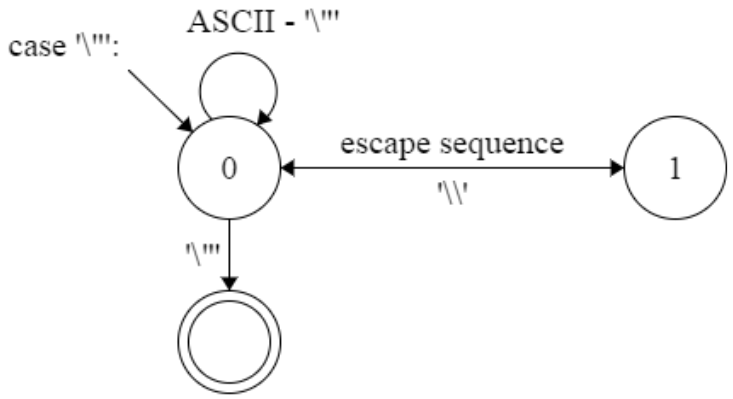
```
else {
  tokval += GetChar();
  waiting_quot = true;
}
```

Set `waiting_quot` to `true` when the scanner faced printable ASCII character, right after the scanner starts to scan character.

3.4 String implementation

`token = tString, tokval = ‘‘characters’’`

When we faced a double quotation, the scanner read characters as string (include empty string) until the closing double quotation. In similar, we adopted one flag variable : `faced_escape`, with initial value = `false`.



state encoding : [faced_escape]

```
if (c == EOF) {
  token = tUndefined;
  break;
}
else if (!IsAsciiChar(c)) {
  token = tUndefined;
  tokval += GetChar();
}
```

When we faced `EOF` or non printable ASCII char, set token to `tUndefined` and terminate scanning.

```
else if (c == '\\') {
  faced_escape ^= true;
  GetChar();

  if (faced_escape)
    tokval += c;
}
```

Toggle `faced_escape` when we encounter backslash.

```
else if (faced_escape) {
  faced_escape = false;
  tokval.pop_back();

  switch (c) {
    ...
  }
```

Construct escape sequence if `faced_escape` is true.

```
else if (c == '\"') {
  tokval += GetChar();
  break;
}
```

When we faced clsoing double quote, terminate it.

```
else
  tokval += GetChar();
```

Otherwise, this means the character is valid. Get a character and add this to `tokval`.

3.5 Keyword implementation

We managed keywords and identifiers in one symbol table. Identifiers are dynmaically added to symbol table. When scanning an identifier, check

symbol table whenever an identifier exist.

```
token = tIdent;

while (_in->good()) {
  char nc = _in->peek();
  if (!IsLetter(nc) && !IsDigit(nc)) break;  // invalid character
  tokval += GetChar();
}

auto iter = keywords.find(tokval);
if (iter != keywords.end())   // new identifier
  token = iter->second;
else                        // already taken
  keywords[tokval] = token;
```

4 Special case policy

4.1 escape sequence

SnuPL/1 supports the following escape sequences : ‘\n’, ‘\t’, ‘\0’, ‘\“’, ‘\‘’, ‘\\’. When the scanner faces ‘\’ character, it expects that the following character should generate escape sequence. Otherwise, it is considered as error.

For example, ‘\\’ is <tChar, “\\”> and ‘\a’ is <tUndefined, “\\a”>.

4.2 quotation error

Character: Stop when faced undefined situation.

1. Empty character.

The scanner converts ‘’ into <tUndefined, ‘‘\‘\‘’’>.

2. Facing EOF.

   The scanner converts ‘ into <tUndefined, ‘‘\‘’’>.

3. Several characters. i.e. ‘ab, ‘\nb, etc.

   The scanner stops to build tokval until it realized that the input is undefined. It converts ‘ab into <tUndefined, ‘‘\‘a’’> and <tIdent, ‘‘b’’>. It converts ‘\nb into <tUndefined, ‘‘\‘\n’’> and <tIdent, ‘‘b’’>.

4. Invalid escape sequence. i.e. ‘\a.

   Handlie with invalid escape sequence is prior than stop to build tokval.

   It converts ‘\a into <tUndefined, ‘‘\‘\\a’’>.

String: For unclosed string, we have a little different policy.

The scanner recognizes the string token until the closing double quote, even if it classifies the token as tUndefined. If there is no closing double quote, the scanner add the character to string until EOF.

For example,

“Hello” <tString, ‘‘Hello’’>

“Hello <tUndefined, ‘‘\‘‘Hello’’>

“Hello\n” <tString, ‘‘Hello\n’’>

“Hello\a” <tUndefined, ‘‘\‘‘Hello\\a\’’’’>

“Hello\a World” <tUndefined, ‘‘\‘‘Hello\\a World\’’’’>