Compilers Principle Project 1 Report

Basic Design

Basic part is almost the same as the document requires, I add some flavor to make error processing more comfortable.

1. Wildcard for undefined tokens

   In some cases there exist undefined character, if leave them there will stop Bison from keeping reading other tokens, for example.

   ```
   int main() {
       int @ = 1;
       float $ = 1 | 2;
   }
   // There are three undefined tokens, '@', '$' and '|' for spl.
   // If we only take care of '@', the rest two will not be found by parser.
   ```

   I add a wildcard token called `ERR`, it will match any undefined token (single length), and `ERR` can be viewed as an `ID` or a binary operator like `||`, in this way, it sort of resolve the problem of undefined token and allow the syntax analysis to be continued.

2. Error collection

   I design a global error collection mechanism, in both lex and syntax stage, the compiler can report error while keep the wheel running.

   ```
   void add_err(ERROR_TYPE type, int lineno, const char* msg, const char* token);
   void print_err(const ERROR *err);
   bool is_err();
   ```

   And this mechanism now support the following error type.

   1. Missing semicolon
   2. Extra comma
   3. Missing parenthesis
   4. Missing specifier
   5. Undefined token detection
   6. Missing expression

Bonus Design

For-Loop Support

I implemented the syntax support for `For Loop`.

```
FOR LP NullableExp SEMI NullableExp SEMI NullableExp RP CloseStmt {
        NODE* cur = new_node("Stmt", 0, NON_TER, $1->lineno);
        insert(cur, $1);
        ...
        insert(cur, $9);
    }
```

which is quite simple, with three expressions to be `[init, condition, increament]` respectively.

Empty Statement Support

In order to support simplified `For Loop` like

```
for(;;)
```

I add support for empty statement with only a single ;. This is implemented by adding a dummy token called `NullableExp`.

```
NullableExp: %empty { $$ = new_node("NullableExp", 0, NON_TER, 0);}
    | Exp;
```

It's either empty or a valid expression, which only appears inside a `ClosedStmt` or a `For Loop` rules to avoid confusion.

```
CloseStmt: NullableExp SEMI {
        NODE* cur = new_node("Stmt", 0, NON_TER, $1->lineno);
        insert(cur, $1);
        insert(cur, $2);
        $$ = cur;
    }
...
FOR LP NullableExp SEMI NullableExp SEMI NullableExp RP CloseStmt {
...}
```

File Inclusion Support

I implemented a basic preprocessor in C++ to do the file inclusion job. it now support **recursive** file inclusion. Since pre-process happens before lex and syntax phase, thus we do not need to change anything in Flex or Bison.

Due to the limit amount of time, I wasn't able to implement the macro expansion mechanism.

```
std::list<std::list<std::string>> into_lines(std::istream &fin);
std::list<std::list<std::string>> file_inclusion(std::list<std::list<std::string>>
lines);
std::string to_str(std::list<std::list<std::string>> lines);
```

1. Open the file, process it into a `token-stream`
2. Process the `token-stream`, once meet `#include`, do 1 to the new file
3. Append the `new-token-stream` to the original one
4. Until no more `#include` encountered, convert the `token-stream` back to string

Multi-line and single-line comment support

With `start condition` support, this part is fairly easy.

```
%x C_MULTILINE_COMMENT
%x C_SINGLELINE_COMMENT
%%

"//"  { BEGIN(C_SINGLELINE_COMMENT); }
<C_SINGLELINE_COMMENT>\n   { BEGIN(INITIAL); }
<C_SINGLELINE_COMMENT>[\t" "]   { }
<C_SINGLELINE_COMMENT>.    { }


"/*"            { BEGIN(C_MULTILINE_COMMENT); }
<C_MULTILINE_COMMENT>"*/" { BEGIN(INITIAL); }
<C_MULTILINE_COMMENT>[\n\t" "]   { }
<C_MULTILINE_COMMENT>.    { }
```

Once meet `//` or `/*`, it will put Flex into a special mode, where it will not process anything until a `closing symbol` is encountered.

In multi-line mode, the symbol is `*/`, in single-line mode, the symbol is `\n`.