Scanner Implementation

Specification

Compiler Design Project 1

: implement a C-Minus scanner in two methods

1. Reads an input source code string
2. Tokenizes the string
3. Returns or prints the identified tokens

Basic Knowledge

Lexical Analysis (Scanner) divides programs into tokens as language is recognizing words from sentences

- Specification: specify lexical patterns (RE)
  - if multiple token matches, choose one with the highest priority
  - Type of Tokens in C-Minus
    - Keywords: int void if else while return
    - Symbols: `+ - * / < <= > >= == != = ; , ( ) [ ] { }`
    - Identifiers
      - ID = letter (letter | digit)*
      - NUM = digit digit*
    - Whitespaces: Spaces, newlines, tabs
      - ignore at the beginning/end of line
      - use these to distinguish tokens
    - Comments: /* */
- Recognition: recognize the specified patterns (DFA)
- Automation: RE→DFA (Lex)
  - Lex Specification
    - Definition section: define variables, enumeration, sub-rules
    - Rules section: lexcial patterns, actions
    - User Func section
  - Thompson's construction(RE→NFA), Subset construction(NFA→DFA)

Compilation Environment

Windows 11

Ubuntu 22.04.3 LTS

GNU/Linux 5.15.153.1-microsoft-standard-WSL2 ×86_64

Common Implementation

FILE `globals.h` (about global variables)

- change `MAXRESERVED` to `6` because there are 6 kinds of reserved words

- remove Tiny's tokens and add C-Minus tokens to `TokenType`

FILE `main.c` (about executing function)

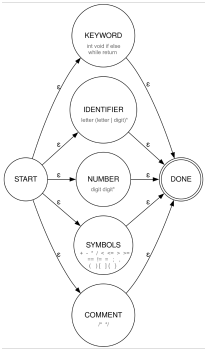
- set `NO_PARSE`, `TraceScan` to `TRUE` to get a scanner-only compiler and can also set `EchoSource` to `TRUE` to print every line of the test code before scanning for debugging

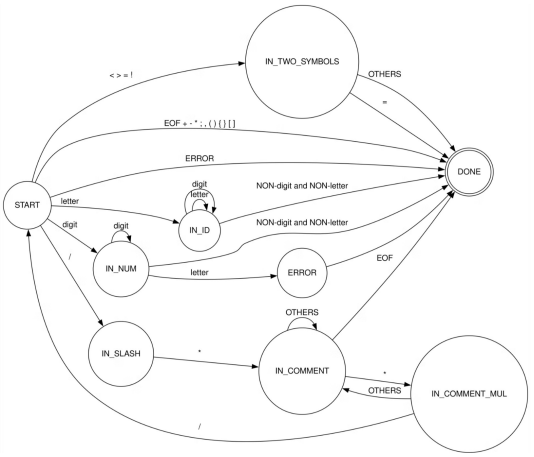
FILE `util.c` (about printing function)

- modify `printToken()` for C-Minus tokens following the specifications

#1 Custom C Code

Implementation





DFA Optimization

- <=, >=, ==, != These 4 symbols are containing symbol '=' at the last part, so we can merge 4 states(`IN_LESS`, `IN_GREAT`, `IN_BANG`, `IN_EQ`) into 1 state (`IN_TWO_SYMBOLS`).

FILE `scan.c`

- define all my custom states of DFA that are needed for recognition of specified patterns in enum `StateType`
- add all keywords for C-Minus in the struct `reservedWords`. Whether the token is a keyword or not will be checked in the last part of the function `getToken()` if the token was recognized as an identifier at that time.
- `getToken()` is the main part of the scanner. It should be modified for C-Minus tokens. In this function, `state` means the current state in DFA, and `currentToken` means recognized token. I use `getNextChar()` to read every char, and `ungetNextChar()` to unread the last char. If I set the variable `save` to `TRUE`, that means it will be printed out.

  I mostly just followed my DFA diagram.

  At the `START` state, the scanner will recognize if it is potential for numbers or identifiers, will be forwarded to next state if it has a potential to be symbols that have more than one char(`==`, `<=`, `>=`, `!=`, `/*`). Comments should not be printed, so for `/`, `save` is set as `FALSE`. The scanner also recognize `EOF`, all whitespaces, and all one-letter symbols(`+`, `-`, `*`, `;`, `,`, `(`, `)`, `{`, `}`, `[`, `]`).

  If the last char was `/` (`INSLASH` state), then we should determine if the next char is `*` or not because it can be either the starting point of comments or just a `DIV` symbol. If it satisfies `/*`, then the state will be forwarded to `INCOMMENT`. Otherwise, it's forwarded to `DONE` state as a `DIV` token.

At the `INCOMMENT` state, we check `EOF` and `*` to find when comments end. However, we need further checking for `*/`, so we forward the state to `INCOMMENTMUL` if we get a char `*`.

In `INCOMMENTMUL` state, we check if we can get a `/` symbol right after and then directly set the state as `START` because we don't need further calculation for comments. If not, then the last `*` symbol was just a letter in comments, so set the state as `INCOMMENT` again.

For `INTWOSYMBOLS`, it means that we already have a letter among `=`, `<`, `>`, `!`. We know that all symbols with two letters have `=` as the last letter. Therefore, we combine all states as `INTWOSYMBOLS` and can just check if the next symbol is `=` or not. Set the `currentToken` as it says, and if not, unread the char and set the `currentToken` as before.

For the identifiers, we accept the mixture of letters and digits as long as it starts with a letter. So I changed the condition `!isalpha(c)` to `!isalpha(c) && !isdigit(c)`.

Trouble Shooting

1. If an error occurs in front of `NUM`, the number will be printed as an `ERROR:` also. Since I modified the code to print the error when we got the mixture of numbers and digits. So, I added the code which is setting `currentToekn` as a `NUM` at the `START` state.

Execution

```
make cminus_cimpl // compile
./cminus_cimpl test.cm // execute
```

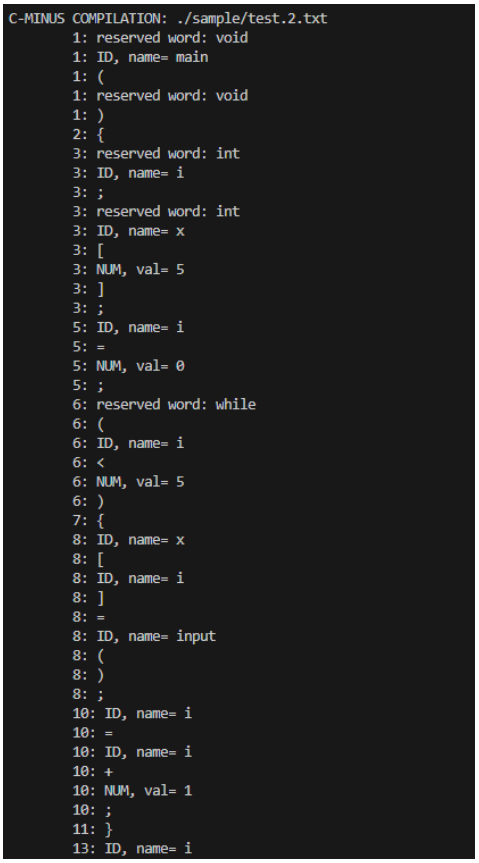
Result

```
C-MINUS COMPILATION: ./sample/test.1.txt
        4: reserved word: int
        4: ID, name= gcd
        4: (
        4: reserved word: int
        4: ID, name= u
        4: ,
        4: reserved word: int
        4: ID, name= v
        4: )
        5: {
        6: reserved word: if
        6: (
        6: ID, name= v
        6: ==
        6: NUM, val= 0
        6: )
        6: reserved word: return
        6: ID, name= u
        6: ;
        7: reserved word: else
        7: reserved word: return
        7: ID, name= gcd
        7: (
        7: ID, name= v
        7: ,
        7: ID, name= u
        7: -
        7: ID, name= u
        7: /
        7: ID, name= v
        7: *
        7: ID, name= v
        7: )
```

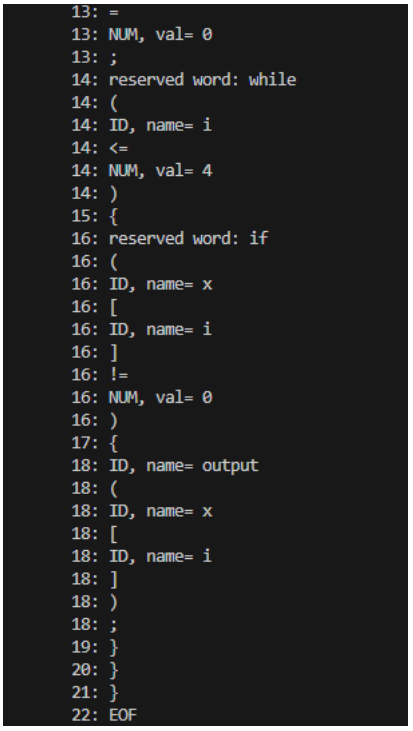
test.1.txt (1)

```
        7: ;
        9: }
       11: reserved word: void
       11: ID, name= main
       11: (
       11: reserved word: void
       11: )
       12: {
       13: reserved word: int
       13: ID, name= x
       13: ;
       13: reserved word: int
       13: ID, name= y
       13: ;
       14: ID, name= x
       14: =
       14: ID, name= input
       14: (
       14: )
       14: ;
       14: ID, name= y
       14: =
       14: ID, name= input
       14: (
       14: )
       14: ;
       15: ID, name= output
       15: (
       15: ID, name= gcd
       15: (
       15: ID, name= x
       15: ,
       15: ID, name= y
       15: )
       15: )
       15: ;
       16: }
       17: EOF
```

test.1.txt (2)



test.2.txt (1)



test.2.txt (2)

#2 Lex Implementation

Implementation

FILE `cminus.l`

- copy and paste the code from `lex/tiny.l` and define my lexical rules based on C-Minus.
- For the definition section, declare some sub rules in regular expressions. I modified `identifier` regular expression, because in C-Minus it allows a mixture of alphabets and numbers. Thus, we should not allow identifiers starting with numbers, so I defined an error for that.
- About the rule section, I modified token rules according to C-Minus rules. Especially for comments, I add a variable `last_c` to check these two symbols: `*/`.
- didn't modify the subroutine section.

Execution

```
apt-get install flex // install -> lex.yy.c will be created
flex cminus.l // lex.yy.c is created
make cminus_lex // compile
./cminus_lex test.cm // execute
```

Result

same as above