fns: Lexical Analysis Phase

Using flex

Lexical Analysis

- the first phase of building a compiler for a language
- it takes the source code of our program as input and produces a token stream as output
- we can either make a hand-written lexical analyser or use a lexical analyzer generating tool like flex and ANTLR
- it is implemented as a deterministic finite automata
- the output is sent to the parser for the next phase of the compiler

Flex: Fast Lexical Analyzer Generator

- flex is a free and open-source tool that's part of the POSIX standard
- produces a lexical analyzer upon receiving a .l file
- a .l file is written by the programmer and it contains a set of rules for pattern-matching
- pattern-matching is done to regular expressions
- the lexical analyzer produced is the lex.yy.c file
- lex.yy.c is a C program that contains the DFA equivalent to the regular expressions in the .l file
- it implements the DFA in a switch-case
- Flex is inspired from lex but with more flexibility, speed and checks in place

Flex: Fast Lexical Analyzer Generator

We use flex as a tool to do our lexical analysis due to the following reasons:

- Flex has a familiar feel because it includes segments of a C program
- The lex-specification of the entire ANSI-C grammar is publicly available and is a helpful reference
- Flex takes up lot less memory and is faster compared to other tools like ANTLR
- Our language doesn't have a lot of new fancy features that can't be handled by flex or would require more powerful modern lexers
- for the same reason handwritten parsers are cumbersome and redundant

Flex Structure

The input to flex is a .l file containing 3 sections:

1. Declaration Section: It includes headers that might be used and declares variables and functions. It's copied to lex.yy.c as it is.
2. Rules Section: It contains patterns as regular expressions and corresponding rules to follow when a pattern is matched.
3. Program Section: It defines subroutines and special functions like yywrap and a main function.

Design Overview

1. We first match keywords, followed by operators and punctuators.
2. We then match comments, real and integer literals, identifiers and string literals.
3. Lastly, we deal with whitespaces and new lines. Any other pattern is reported as error.

For each pattern that is matched, we print the token, it's value, length and line number.

Later, depending upon the input required by the parser, we may change the actions to be performed on the identified pattern.

As of now, the .l file can be compiled using:

```
flex lexer.l
```

Then, the token stream can be generated by running:

```
g++ -o a.out lex.yy.c -ll
./a.out < input.fns > tokenstream.txt
```

Member Contributions