Overview of Lexical Analyzer

Gautam Singh
CS21BTECH11018

Contents

This document provides a brief overview, implementation details and challenges in creating the given lexical analyzer (LA). The given LA lexes a program in an imaginary source language, writes the list of tokens encountered along with their type in one file, while also writing an almost equivalent C program in another file.

The outputs on four public testcases, along with the testcases themselves, are also present in the given archive.

1 Building and Usage

To build the lexical analyzer from the `lex` source code, enter the following commands at a terminal window.

```
lex lex.l # flex may also be used
gcc -O2 -lfl lex.yy.c
```

This will generate an executable file `a.out`, which is the LA. To run the LA on any select testcase, enter the following.

```
./a.out testcase.txt tokens.txt c.txt
```

where paths of the files are relative to the executable.

2 Implementation Details

The following are some important implementation details of the LA, along with the motivations behind choosing them. The lex source code has more verbose details related to implementation.

1) A regular expression for delimiters, letters, special characters and digits have been specified in the regular definitions section to make it easier to refer in the rules section of the code, and also to create other complex regular expression definitions such as whitespaces (`ws`) and numbers (`number` and `posint`).
2) To identify string and character constants, even those with escaped sequences, the `char` regular definition has been defined in the given way, considering escape and non-escape sequences.
3) The main use of the regular definition `posint` is for the program point labels (abbreviated to `pp`) in the language to be lexed.
4) The tokens encountered sequentially are also stored in a doubly linked list. Each node of the linked list represents a single token and contains the following components.
   a) Pointers to nodes to the left and right of it.
   b) Lexeme in source language, and equivalent lexeme in C language.
   c) Type of the lexeme.
5) Two reasons for using a doubly linked list are as follows.
   a) The operator _ is equivalent to `pow()` in C. Thus, we require both operands of this binary operator. This is quite simple to handle with a doubly linked list, where we can refer to the token before and after the operator.
   b) To generate C code appropriately, we require to check whether there is a `jump to` (equivalent to `goto` in C) precedes a label token. If so, then we need to add it to the program, otherwise ignore it.
6) Another issue with the label tokens in the source language is that we require to make one prior pass to find which labels should be included in the C file.
   This has been worked around by using an integer array as a direct addressing table, whose element at an index `i` is 1 if the label at line `i` has been referenced, and 0 otherwise.
7) The square brackets have two meanings based on the context they are used in the source

language. To find out which context is implied, a boolean flag variable `yycondfl` has been used.

3 Implementation Issues

Some of the implementation issues and shortcomings of the LA are described below.

1) The LA keeps reallocating the direct addressing dynamically, based on the number of lines being lexed. To prevent this, one more pass would be needed to find the number of lines, which is undesirable.
2) The LA cannot catch all the errors in the program point labels at the start of each line, lexing them as identifiers instead. Such a check would have to be left to the semantic analyzer in a compiler pipeline.
3) The LA cannot generate indented C source code. Thankfully, indentation does not affect the compilation process in C, and is only for readability.