Lexical Analyzer Documentation

Design and Implementation of Lexical Analyzer

Tyler LeBlanc
Introduction to Compiling
Professor Fatima
Wilfrid Laurier University
02/02/2025

Introduction

The lexical analyzer is the first phase of the compiler and is responsible for converting characters into tokens. The purpose of this phase is to read a file, identify the tokens and their types, and handle any lexical errors that arise using panic mode recovery.

Overview of Lexical Analysis

Lexical analysis is the tokenization of characters based on predetermined patterns called regular expressions. Tokens represent keywords, identifiers, and operators that are essential for later stages of the compiler. The analyzer utilizes a double buffer to read the file more efficiently and to handle any large tokens. Additionally, a transition table is used to manage the state transitions.

Design of the Lexical Analyzer

Double Buffer

The analyzer uses a double buffer technique to read the files more efficiently and handle any tokens that may be cut off when added to the buffer. The buffer alternates between two halves, one being read from while the other is filled.

State Transition

The transition table maps ASCII characters to states, with specific states being acceptance states. Each token type has a different acceptance state, allowing for classification of tokens through regular expressions. The panic mode error recovery is implemented to allow the analyzer to continue reading despite any lexical errors.

Implementation Details

Programming Language

The C programming language was chosen due to its low-level access to memory and ability to create new structures.

Major Components

Main.c: The main program file to handle inputs, tokenization, and error handling

LexicalAnalysis.h: Header file containing key structures and function declarations

Key Structures

DOUBLE_BUFFER: Used to manage file inputs using a double buffer

TOKEN: Used to store individual tokens and their token type

TOKEN_ARRAY: A dynamic array used to store all tokens

Token Specification

Keywords: and, def, do double, else, fed fi, int, if, not, od, or, print, return, then, while

Identifiers: Variable and function names

Operators: split into comparison, mathematical, and assignment operators

Delimiters: comma, semicolon, parentheses, spaces, etc.

Algorithm Explanation

1. Initialize the double buffer and token array
2. Read the characters using getNextChar and store them in a buffer to read the token
3. Determine the next state using the transition table
4. Once a delimiting character is read the current state is set to 0
5. Backtrack in order to remove the delimiting character and read the token
6. Determine the tokens type and store it in the token array
7. A lexical error is found the the state becomes -1
8. Handle invalid characters by logging errors and resetting the state
9. Continue until the entire file is read

Handling Lexical Errors

Panic Mode: The lexical analyzer skips over the invalid character and continues processing the rest of the file

Error Log: The analyzer will indicate if any invalid characters are found and which line they are found on. The analyzer will also indicate if any unknown token types are found.

Challenges and Limitations

Challenges

Some challenges when implementing the lexical analyzer were properly handling the state transitions, keeping track of the current token that is being read for later storage, and handling lexical errors.

Limitations

The lexical analyzer is limited to predefined keywords and tokens patterns. In addition, the performance of the analyzer will likely degrade as the input file increases.

Conclusion

The lexical analyzer successfully identifies tokens from the input file and handles lexical errors using panic mode. Future improvements may include additional token types, and optimization for larger input files.