Overview of Syntax Analyzer

Gautam Singh
CS21BTECH11018

Contents

This document provides a brief overview, implementation details and challenges in creating the given syntax analyzer, which lexes a program in a C-like source language, writes the list of tokens encountered along with their type in one file, while also parsing statement by statement and reporting the types of statements parsed.

1 Building and Usage

To build the syntax analyzer from the source code, enter the following commands at a terminal window in the directory where the source files are present.

```
flex lex.l
bison -d -by parser.y
gcc -O2 -lfl -ly lex.yy.c y.tab.c
```

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

```
./a.out src.clike tokens.txt statements.
   parsed
```

where paths of the files are relative to the executable.

2 Implementation Details

The following are some important implementation details of the syntax analyzer. The comments in the source code provide more details.

1) The lexical analyzer analyzes the source file token-by-token and passes the information on to the syntax analyzer for statement-level parsing. This is achieved using the `%token` directive in `bison`.
2) Both the lexical analyzer and syntax analyzer share the following variables, with the `extern` modifier.
   a) `FILE *tfile`: The filestream to write the tokens.
   b) `FILE *pfile`: The filestream to write the parsed statements along with their types.
   c) `int yyerrfl`: A flag variable to indicate error in parsing.
   d) `int yytype`: A variable indicating the type of statement parsed, based on the last character of that statement. This is to handle reporting of incorrect syntax at the end of the parsed statement.
3) It is much more convenient to handle necessary parts of the file output in the lexical analysis stage (such as error diagnostics) as it considers the source file token-by-token, rather than in the syntax analyzer, where source code is considered phrase-by-phrase.
4) The syntax analyzer also handles errors that are not specific to one statement. As an example, incorrectly matched braces can be reported only by the syntax analyzer.

3 Implementation Issues

Some of the implementation issues and challenges faced in creating the syntax analyzer are listed below.

1) The syntax analyzer did not handle missing braces and count return statements, as shown in public test case 1. This was fixed by adding error productions to the grammar for the start variable `program_body`. The global variable `yyretcnt` maintained a count of return statements with corresponding error productions in the grammar.

2) Public test case 2 tests if the grammar parses nested expressions and predicates, along with their unary operators properly. Especially for predicates, the grammar had to be modified to introduce right associativity so that shift-reduce and reduce-reduce conflicts were avoided. Another subtle detail exposed by this public test case is the handling of multiple statements in the same line, as in C. Redefining `yytype` enabled better error diagnostics for the *correct* statement in the line, and not report the error at the end of the line.
3) Public test case 3 checks the for loop construct, whether empty scopes are allowed, and whether statements are allowed outside the scope of a block. An extra production for `block_scope` enabled the handling of empty scopes.