# Lisa Pascal Compiler Documentation

## Overview

The Lisa Pascal Compiler is designed to parse and perform semantic analysis on Pascal language programs using Lex (Flex) and Yacc (Bison). It includes lexical analysis, syntax analysis, and semantic checks using a symbol table (hashmap) to store and validate identifiers.

## Dependencies

Flex: A fast lexical analyzer generator.

Bison: A general-purpose parser generator.

## File Structure

The project consists of two main files:

1. lisa.l: Contains the lexical analyzer rules.

2. lisa.y: Contains the grammar rules and semantic actions for parsing.

## Lexer (Flex) - lisa.l

### Header and Definitions

%{

#include "lisa.tab.h"

#include <string.h>

int line\_number = 1;

char id[100];

%}

%option case-insensitive

Header: Includes necessary header files and function declarations.

Options: Sets case-insensitive matching.

Line Number Tracking: Tracks the current line number for error reporting.

### Token Definitions and Rules

"(\*"([^\*]|[\r\n]|"\*"[^)])\*"\*)" { /\* Multi-line Comment \*/ }

"{""([^}]|[\r\n])\*"}" { /\* Multi-line Curly Comment \*/ }

\n { line\_number++; }

[ \t\r]+ { /\* Ignore whitespace \*/ }

Comment Handling: Ignores multi-line comments in Pascal style.

Line Number Handling: Increments line number on every newline character.

Whitespace Handling: Ignores spaces, tabs, and carriage returns.

### Keyword Tokens

"and" { return TOK\_AND; }

"end" { return TOK\_END; }

etc ...

"dbf\_scale\_factor" {printf("found dsf\n"); return dbg\_scale\_factor;}

Defines tokens for Pascal keywords and a custom token `dbg\_scale\_factor`.

### Operator and Identifier Patterns

"<>" { return TOK\_NE; }

[a-zA-Z][a-zA-Z0-9\_]\* { strncpy(id, yytext, yyleng); id[yyleng] = '\0'; strcpy(yylval.id, id); return IDENTIFIER; }

[0-9]+ { return DIGIT\_SEQUENCE; }

[-+\_\*/=<>\[\].,():;^@] { return \*yytext;} Operators

Operators: Defines pattern for different operators.

Identifiers: Captures variable names and stores them in `id`.

Numbers: Captures digit sequences.

### Error Handling

. { fprintf(stderr, "Unrecognized character at line %d: %d ('%c')\n", line\_number, (unsigned char)\*yytext, \*yytext); }

Reports unrecognized characters.

### File End Handling

int yywrap(void) {

return 1;

}

Handles end-of-file for input.

## Parser (Bison) - lisa.y

### Header and Definitions

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

extern int line\_number;

extern int yylex(void);

extern FILE \*yyin;

extern char id[100];

void yyerror(char const \*s) {

fprintf(stderr, "SYNTAX ERROR AT LINE %d: %s\n", line\_number, s);

exit(0);

}

#define HASH\_SIZE 128

typedef struct Item { ... } Item;

typedef struct IdentifierTable { ... } IdentifierTable;

...

char ambito[100] = "global";

IdentifierTable myhashmap = { NULL };

int hash(const char \*id) { ... }

int insert(IdentifierTable \*table, const Item \*item) { ... }

...

%}

Header: Includes necessary header files and function declarations.

Symbol Table Structures & Functions: Implements hash-based symbol table for semantic analysis.

### Token Definitions

%token TOK\_AND TOK\_END ... TOK\_EXTERNAL

%token DIGIT\_SEQUENCE HEX\_DIGIT\_SEQUENCE E STRING

%token dbg\_scale\_factor

%left '+' '-'

%right TOK\_NOT

Defines tokens and their precedence/associativity rules.

### Grammar Rules and Actions

%%

script: program { printf("valid program\n"); };

program: program\_heading ';' block opt\_dot

| program\_heading ';' uses\_clause ';' block opt\_dot

;

opt\_dot: | '.';

program\_heading: TOK\_PROGRAM IDENTIFIER { ... }

| TOK\_PROGRAM IDENTIFIER '(' program\_parameters ')' { ... }

;

uses\_clause: TOK\_USES identifier\_list;

block: opt\_label\_declaration\_part

...

statement\_part;

Defines the grammar rules for recognizing Pascal programs, program headers, uses clauses, and blocks. Actions are provided to handle symbol table insertions, scope management, and error handling.

### Semantic Actions

program\_heading:

TOK\_PROGRAM IDENTIFIER {

Item tmpItem;

strcpy(tmpItem.identificador, $2);

...

insert(&myhashmap, &tmpItem);

append(ambito, $2);

};

variable\_declaration: identifier\_list ':' type ';' {

for (int i = 0; i < idindex; i++) {

Item item;

strcpy(item.identificador, idlist[i]);

...

insert(&myhashmap, &item);

}

idindex = 0;

};

Introduces semantic actions, where entries are made in the symbol table and scope management functions are used.

### Function Definitions for Symbol Table

int insert(IdentifierTable \*table, const Item \*item) { ... }

Item \*get(const IdentifierTable \*table, const char \*id) { ... }

void append(char \*ambito, const char \*new\_segment) { ... }

int pop(char \*ambito) { ... }

Defines supporting functions to handle symbol table operations such as insertion, lookup, and scope management.

### Main Function

int main(int argc, char \*\*argv) {

...

char globalIdentifiers[3][100] = { "string", "real", "integer" };

for (int i = 0; i < 3; i++) {

Item tmpItem;

...

insert(&myhashmap, &tmpItem);

}

++argv, --argc;

if (argc > 0)

yyin = fopen(argv[0], "r");

else

yyin = stdin;

yyparse();

show(&myhashmap);

}

Initialization: Initializes predefined types.

File Handling: Opens and reads the Pascal source file.

Parsing: Invokes the parser and displays the symbol table once parsing is complete.

## Build and Execution

### Compiling

1. Generate the lexical analyzer and parser:

$ flex lisa.l

$ bison -d lisa.y

2. Compile the generated C files:

$ gcc -o lisa\_compiler lex.yy.c lisa.tab.c -lfl

### Running

Execute the compiler with a Pascal source file as input:

$ ./lisa\_compiler source.pas

## Error Handling

Error handling within the compiler includes:

Lexer Errors: Unrecognized characters are reported with their location.

Parser Errors: Syntax errors trigger an error message with the line number.

Semantic Errors: Duplicate entries and undeclared variables result in error messages and terminate the compiler.

## Symbol Table

The symbol table is implemented as a hashmap to efficiently manage identifiers, their scope, and their types. Built-in functions manage insertions, lookups, and scope changes, ensuring semantic accuracy during parsing.