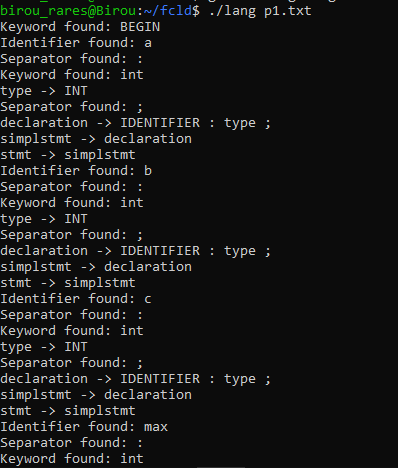
Documentation   
  
  
Commands used for running (after installing the packages):

* Generate the parser (it shouldn’t print anything if successful)
  + processes the lang.y file
  + **lang.tab.c**: The C source file implementing the parser based on your grammar.
  + **lang.tab.h**: A header file containing token definitions. Used in other parts like lexer (for ex).
  + **-d** generate the header file lang.tab.h
  + **-o lang.tab.c**: Specifies the output filename for the generated C source code (lang.tab.c)
  + 
* Generate the lexer (it shouldn’t print anything if successful)
  + processes the lang.lxi file (lexical analyzer rules)
  + **lex.yy.c**: A C source file implementing the lexer. This file contains a yylex function that reads input, matches tokens, and passes them to the parser.
  + **-o lex.yy.c**: Specifies the output filename for the generated lexer (lex.yy.c)
  + 
* Compile and link (it shouldn’t print anything if successful)
  + compiles and links the parser (lang.tab.c) and lexer (lex.yy.c) into an executable named lang
  + **lang.tab.c** and **lex.yy.c**: Parser source file generated by Bison and lexer source file generated by Flex.
  + **-lfl**: Links the Flex library, which provides required functions like yywrap used by the lexer
  + **-o lang**: the name of the output executable
  + 
* Run for your chosen file (Outputs debug information, syntax analysis, or error messages)
  + The lexer (lex.yy.c) tokenizes the input and passes tokens to the parser.
  + The parser (lang.tab.c) processes the tokens based on your grammar rules.
  + Outputs debug information, syntax analysis, or error messages.
  + runs the compiled program (lang) with an input file (p1.txt)
  + 

lang.lxi

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "lang.tab.h"

int line\_num = 1;

%}

%option noyywrap

%option caseless

DIGIT [0-9]

LETTER [a-zA-Z]

SYMBOLS [!#%^\*+-/<=>\_.,:;]

IDENTIFIER ({LETTER}|\_)+({LETTER}|{DIGIT})\*

INVALID\_IDENTIFIER ({DIGIT}+{LETTER}+|{DIGIT}+\_)

INTEGER\_CONSTANT [+-]?{DIGIT}+

STRING\_CONSTANT \"({LETTER}|{DIGIT}|{SYMBOLS})\*\"

%%

"BEGIN" { printf("Keyword found: %s\n", yytext); return BEGIN\_BLOCK; }

"END" { printf("Keyword found: %s\n", yytext); return END\_BLOCK; }

"if" { printf("Keyword found: %s\n", yytext); return IF; }

"else" { printf("Keyword found: %s\n", yytext); return ELSE; }

"while" { printf("Keyword found: %s\n", yytext); return WHILE; }

"read" { printf("Keyword found: %s\n", yytext); return READ; }

"write" { printf("Keyword found: %s\n", yytext); return WRITE; }

"int" { printf("Keyword found: %s\n", yytext); return INT; }

"string" { printf("Keyword found: %s\n", yytext); return STRING; }

"char" { printf("Keyword found: %s\n", yytext); return CHAR; }

"boolean" { printf("Keyword found: %s\n", yytext); return BOOLEAN; }

"real" { printf("Keyword found: %s\n", yytext); return REAL; }

"+" { printf("Operator found: %s\n", yytext); return PLUS; }

"-" { printf("Operator found: %s\n", yytext); return MINUS; }

"\*" { printf("Operator found: %s\n", yytext); return TIMES; }

"/" { printf("Operator found: %s\n", yytext); return DIVIDE; }

"%" { printf("Operator found: %s\n", yytext); return MODULO; }

"=" { printf("Operator found: %s\n", yytext); return ASSIGN; }

"==" { printf("Operator found: %s\n", yytext); return EQ; }

"!=" { printf("Operator found: %s\n", yytext); return NEQ; }

"<" { printf("Operator found: %s\n", yytext); return LESS; }

"<=" { printf("Operator found: %s\n", yytext); return LESSEQ; }

">" { printf("Operator found: %s\n", yytext); return GREATER; }

">=" { printf("Operator found: %s\n", yytext); return GREATEREQ; }

"{" { printf("Separator found: %s\n", yytext); return BRACEOPEN; }

"}" { printf("Separator found: %s\n", yytext); return BRACECLOSE; }

"(" { printf("Separator found: %s\n", yytext); return PARENOPEN; }

")" { printf("Separator found: %s\n", yytext); return PARENCLOSE; }

"[" { printf("Separator found: %s\n", yytext); return SQBRACKETOPEN; }

"]" { printf("Separator found: %s\n", yytext); return SQBRACKETCLOSE; }

":" { printf("Separator found: %s\n", yytext); return COLON; }

";" { printf("Separator found: %s\n", yytext); return SEMICOLON; }

{IDENTIFIER} { printf("Identifier found: %s\n", yytext); return IDENTIFIER; }

{INTEGER\_CONSTANT} { printf("Integer constant found: %s\n", yytext); return INTCONSTANT; }

{STRING\_CONSTANT} { printf("String constant found: %s\n", yytext); return STRINGCONSTANT; }

{INVALID\_IDENTIFIER} { printf("Invalid identifier: %s at line %d\n", yytext, line\_num); return INVALID; }

[ \t]+ { /\* Skip whitespace \*/ }

"//".\* { /\* Skip comments \*/ }

\n { ++line\_num; }

. { printf("Unrecognized token: %s at line %d\n", yytext, line\_num); exit(1); }

%%

lang.y  
  
%{

#include "lexer.h"

#include <stdio.h>

#include <stdlib.h>

#define YYDEBUG 1

int yyerror(const char \*s);

%}

%token BEGIN\_BLOCK END\_BLOCK IF ELSE WHILE READ WRITE INT STRING CHAR BOOLEAN REAL

%token PLUS MINUS TIMES DIVIDE MODULO ASSIGN EQ NEQ LESS LESSEQ GREATER GREATEREQ

%token BRACEOPEN BRACECLOSE PARENOPEN PARENCLOSE SQBRACKETOPEN SQBRACKETCLOSE

%token COLON SEMICOLON IDENTIFIER INTCONSTANT STRINGCONSTANT

%token INVALID

%start program

%%

program : BEGIN\_BLOCK stmtlist END\_BLOCK {

printf("program -> BEGIN\_BLOCK stmtlist END\_BLOCK\n");

}

;

stmtlist : stmt {

printf("stmtlist -> stmt\n");

}

| stmt stmtlist {

printf("stmtlist -> stmt stmtlist\n");

}

;

stmt : simplstmt {

printf("stmt -> simplstmt\n");

}

| structstmt {

printf("stmt -> structstmt\n");

}

;

simplstmt : declaration {

printf("simplstmt -> declaration\n");

}

| assignstmt {

printf("simplstmt -> assignstmt\n");

}

| iostmt {

printf("simplstmt -> iostmt\n");

}

;

declaration : IDENTIFIER COLON type SEMICOLON {

printf("declaration -> IDENTIFIER : type ;\n");

}

;

type : INT {

printf("type -> INT\n");

}

| STRING {

printf("type -> STRING\n");

}

| CHAR {

printf("type -> CHAR\n");

}

| BOOLEAN {

printf("type -> BOOLEAN\n");

}

| REAL {

printf("type -> REAL\n");

}

;

assignstmt : IDENTIFIER ASSIGN expression SEMICOLON {

printf("assignstmt -> IDENTIFIER = expression ;\n");

}

;

expression : term {

printf("expression -> term\n");

}

| term PLUS expression {

printf("expression -> term + expression\n");

}

| term MINUS expression {

printf("expression -> term - expression\n");

}

;

term : IDENTIFIER {

printf("term -> IDENTIFIER\n");

}

| INTCONSTANT {

printf("term -> INTCONSTANT\n");

}

| STRINGCONSTANT {

printf("term -> STRINGCONSTANT\n");

}

;

iostmt : READ PARENOPEN IDENTIFIER PARENCLOSE SEMICOLON {

printf("iostmt -> READ ( IDENTIFIER ) ;\n");

}

| WRITE PARENOPEN expression PARENCLOSE SEMICOLON {

printf("iostmt -> WRITE ( expression ) ;\n");

}

;

structstmt : ifstmt {

printf("structstmt -> ifstmt\n");

}

| whilestmt {

printf("structstmt -> whilestmt\n");

}

;

ifstmt : IF condition BRACEOPEN stmtlist BRACECLOSE {

printf("ifstmt -> IF condition { stmtlist }\n");

}

| IF condition BRACEOPEN stmtlist BRACECLOSE ELSE BRACEOPEN stmtlist BRACECLOSE {

printf("ifstmt -> IF condition { stmtlist } ELSE { stmtlist }\n");

}

;

whilestmt : WHILE condition BRACEOPEN stmtlist BRACECLOSE {

printf("whilestmt -> WHILE condition { stmtlist }\n");

}

;

condition : expression LESS expression {

printf("condition -> expression < expression\n");

}

| expression LESSEQ expression {

printf("condition -> expression <= expression\n");

}

| expression GREATER expression {

printf("condition -> expression > expression\n");

}

| expression GREATEREQ expression {

printf("condition -> expression >= expression\n");

}

| expression EQ expression {

printf("condition -> expression == expression\n");

}

| expression NEQ expression {

printf("condition -> expression != expression\n");

}

;

%%

int yyerror(const char \*s) {

printf("Error: %s\n", s);

return 0;

}

extern FILE \*yyin;

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

if (argc > 1) {

yyin = fopen(argv[1], "r");

if (!yyin) {

perror("Failed to open file");

return 1;

}

}

if (!yyparse()) {

printf("Parsing completed successfully.\n");

} else {

printf("Parsing failed.\n");

}

return 0;

}