```
==> include.h <==
struct row_entry {
  struct row_entry* first_child;
  struct row_entry* next_sibling;
  char* name;
};
==> lang.y <==
%token<row> NOT
%token<row> PLUS
%token<row> MINUS
%token<row> MULT
%token<row> DIV
%token<row> MOD
%token<row> EQUAL
%token<row> NOT_EQUAL
%token<row> LT
%token<row> LE
%token<row> GT
%token<row> GE
%token<row> ASSIGNMENT
%token<row> AND
%token<row> OR
%token<row> L_BRACKET
%token<row> R_BRACKET
%token<row> L_SQUARE_BRACKET
%token<row> R_SQUARE_BRACKET
%token<row> L_PARANTHESIS
%token<row> R_PARANTHESIS
%token<row> SEMICOLON
%token<row> COMMA
%token<row> APOSTROPHE
%token<row> COLON
%token<row> PERIOD
%token<row> LET
%token<row> IF
%token<row> ELSE
%token<row> WHILE
%token<row> PRINT
%token<row> READ_I32
%token<row> READ_U32
%token<row> READ_STR
%token<row> READ_BOOL
%token<row> READ_F32
%token<row> I32
%token<row> U32
%token<row> STR
```

```
%token<row> BOOL
%token<row> F32
%token<row> ARRAY
%token<row> TRUE
%token<row> FALSE
%token<row> IDENTIFIER
%token<row> CONSTANT

%{
  #include "include.h"
  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>

  static struct row_entry* cons(const char* name, struct row_entry* first_child) {
    struct row_entry* answer = malloc(sizeof(struct row_entry));
    answer->name = strdup(name);
    answer->first_child = first_child;
    answer->next_sibling = NULL;
    return answer;
  }

  static void display(struct row_entry* node, int count_tabs) {
    if(node == NULL) return;
    for(int i = 0; i < count_tabs; i++) {
      printf("\t");
    }
    printf("%s\n", node->name);
    display(node->first_child, count_tabs + 1);
    display(node->next_sibling, count_tabs);
  }
  static void free_row_entry(struct row_entry* node) {
    if(node == NULL) return;
    free_row_entry(node->first_child);
    free_row_entry(node->next_sibling);
    free(node->name);
    free(node);
  }
  extern int yylex();
  extern int yylex_destroy();
%}

%union {
  struct row_entry* row;
}
```

```
%type<row> program
%type<row> statement
%type<row> type
%type<row> decl_statement
%type<row> assignment_statement
%type<row> input
%type<row> output
%type<row> output_expression
%type<row> loop
%type<row> conditional
%type<row> expression
%type<row> term
%type<row> operator

%%
accept: program                                    { display($1, 0); free_row_entry($1); }
program:                                           { $$ = cons("program", NULL); }
       | program statement                         { $$ = cons("program", $1); $1->next_sibli
       ;

statement: SEMICOLON                               { $$ = cons("statement", $1); }
         | decl_statement SEMICOLON                { $$ = cons("statement", $1); $1->next_sib
         | assignment_statement SEMICOLON          { $$ = cons("statement", $1); $1->next_sib
         | input SEMICOLON                         { $$ = cons("statement", $1); $1->next_sib
         | output SEMICOLON                        { $$ = cons("statement", $1); $1->next_sib
         | conditional                             { $$ = cons("statement", $1); }
         | loop                                    { $$ = cons("statement", $1); }
         ;

type: I32                                          { $$ = cons("type", $1); }
    | U32                                          { $$ = cons("type", $1); }
    | STR                                          { $$ = cons("type", $1); }
    | BOOL                                         { $$ = cons("type", $1); }
    | F32                                          { $$ = cons("type", $1); }
    | ARRAY L_SQUARE_BRACKET type SEMICOLON expression R_SQUARE_BRACKET { $$ = cons("type",
    ;


decl_statement: LET IDENTIFIER COLON type              { $$ = cons("decl_statement", $1); $
              ;

assignment_statement: IDENTIFIER ASSIGNMENT expression  { $$ = cons("assignment_statement",
                    ;
input: IDENTIFIER ASSIGNMENT READ_I32 L_PARANTHESIS R_PARANTHESIS            { $$ = cons("i
     | IDENTIFIER ASSIGNMENT READ_U32 L_PARANTHESIS R_PARANTHESIS            { $$ = cons("i
     | IDENTIFIER ASSIGNMENT READ_STR L_PARANTHESIS R_PARANTHESIS            { $$ = cons("i
```

```
     | IDENTIFIER ASSIGNMENT READ_BOOL L_PARANTHESIS R_PARANTHESIS           { $$ = cons("i
     | IDENTIFIER ASSIGNMENT READ_F32 L_PARANTHESIS R_PARANTHESIS           { $$ = cons("i
     ;

output: PRINT L_PARANTHESIS output_expression R_PARANTHESIS   { $$ = cons("output", $1); $1-
      ;

output_expression: expression                                 { $$ = cons("output_expressio
                 | output_expression COMMA expression         { $$ = cons("output_expressio
                 ;

conditional: IF expression L_BRACKET program R_BRACKET        { $$ = cons("conditional", $1)
           | IF expression L_BRACKET program R_BRACKET ELSE L_BRACKET program R_BRACKET { $$
           ;

loop: WHILE expression L_BRACKET program R_BRACKET  { $$ = cons("loop", $1); $1->next_siblin
    ;

operator: NOT               { $$ = cons("operator", $1); }
        | PLUS              { $$ = cons("operator", $1); }
        | MINUS             { $$ = cons("operator", $1); }
        | MULT              { $$ = cons("operator", $1); }
        | DIV               { $$ = cons("operator", $1); }
        | MOD               { $$ = cons("operator", $1); }
        | EQUAL             { $$ = cons("operator", $1); }
        | NOT_EQUAL         { $$ = cons("operator", $1); }
        | LT                { $$ = cons("operator", $1); }
        | LE                { $$ = cons("operator", $1); }
        | GT                { $$ = cons("operator", $1); }
        | GE                { $$ = cons("operator", $1); }
        | AND               { $$ = cons("operator", $1); }
        | OR                { $$ = cons("operator", $1); }
        ;

expression: NOT expression                { $$ = cons("expression", $1); $1->next_sibling =
          | term operator expression      { $$ = cons("expression", $1); $1->next_sibling =
          | term                          { $$ = cons("expression", $1); }
          | MINUS expression              { $$ = cons("expression", $1); $1->next_sibling =
          ;

term: IDENTIFIER                                  { $$ = cons("term", $1); }
    | CONSTANT                                    { $$ = cons("term", $1); }
    | TRUE                                        { $$ = cons("term", $1); }
    | FALSE                                       { $$ = cons("term", $1); }
    | L_PARANTHESIS expression R_PARANTHESIS      { $$ = cons("term", $1); $1->next_siblin
    ;
```

```
%%

int yyerror(char *s) {
  fprintf(stderr, "Error: %s\n", s);
  return 0;
}

int main() {
  yyparse();
  yylex_destroy();
  return 0;
}
==> lang.lxi <==
%{
#include "lang.tab.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "include.h"

static struct row_entry* cons(const char* c) {
  struct row_entry* answer = malloc(sizeof(struct row_entry));
  answer->first_child = NULL;
  answer->next_sibling = NULL;
  answer->name = strdup(c);
  return answer;
}

%}

LETTER                [A-Za-z]
DIGIT                 [0-9]
NONZERODIGIT          [1-9]
INTCONSTANT           [+-]?({NONZERODIGIT}{DIGIT}*|0)
STRINGCONSTANT        \"([^"]|\\\")*\"
BOOLCONSTANT          true|false
FLOATCONSTANT         {INTCONSTANT}(\.{DIGIT}+)?
WHITESPACE            [\n\t\r ]
COMMENT               \/\/.*$
CONSTANT              {INTCONSTANT}|{FLOATCONSTANT}|{BOOLCONSTANT}|{STRINGCONSTANT}
IDENTIFIER            (_|{LETTER})({LETTER}|{DIGIT}|_)*
%%
{WHITESPACE} {}
{COMMENT} {}
"!" { yylval.row = cons(yytext); return NOT; }
"+" { yylval.row = cons(yytext); return PLUS; }
```

```
"-" { yylval.row = cons(yytext); return MINUS; }
"*" { yylval.row = cons(yytext); return MULT; }
"/" { yylval.row = cons(yytext); return DIV; }
"%" { yylval.row = cons(yytext); return MOD; }
"==" { yylval.row = cons(yytext); return EQUAL;  }
"!=" { yylval.row = cons(yytext); return NOT_EQUAL; }
"<" { yylval.row = cons(yytext); return LT; }
"<=" { yylval.row = cons(yytext); return LE; }
">" { yylval.row = cons(yytext); return GT; }
">=" { yylval.row = cons(yytext); return GE; }
"=" { yylval.row = cons(yytext); return ASSIGNMENT; }
"&&" { yylval.row = cons(yytext); return AND; }
"||" { yylval.row = cons(yytext); return OR; }
"{" { yylval.row = cons(yytext); return L_BRACKET; }
"}" { yylval.row = cons(yytext); return R_BRACKET; }
"(" { yylval.row = cons(yytext); return L_PARANTHESIS; }
")" { yylval.row = cons(yytext); return R_PARANTHESIS; }
";" { yylval.row = cons(yytext); return SEMICOLON; }
"," { yylval.row = cons(yytext); return COMMA; }
"'" { yylval.row = cons(yytext); return APOSTROPHE; }
":" { yylval.row = cons(yytext); return COLON; }
"." { yylval.row = cons(yytext); return PERIOD; }
"let" { yylval.row = cons(yytext); return LET; }
"if" { yylval.row = cons(yytext); return IF; }
"else" { yylval.row = cons(yytext); return ELSE; }
"while" { yylval.row = cons(yytext); return WHILE; }
"print" { yylval.row = cons(yytext); return PRINT; }
"readI32" { yylval.row = cons(yytext); return READ_I32; }
"readU32" { yylval.row = cons(yytext); return READ_U32; }
"readStr" { yylval.row = cons(yytext); return READ_STR; }
"readBool" { yylval.row = cons(yytext); return READ_BOOL; }
"readF32" { yylval.row = cons(yytext); return READ_F32; }
"i32" { yylval.row = cons(yytext); return I32; }
"u32" { yylval.row = cons(yytext); return U32; }
"str" { yylval.row = cons(yytext); return STR; }
"bool" { yylval.row = cons(yytext); return BOOL; }
"f32" { yylval.row = cons(yytext); return F32; }
"array" { yylval.row = cons(yytext); return ARRAY; }
"true" { yylval.row = cons(yytext); return TRUE; }
"false" { yylval.row = cons(yytext); return FALSE; }
"[" { yylval.row = cons(yytext); return L_SQUARE_BRACKET; }
"]" { yylval.row = cons(yytext); return R_SQUARE_BRACKET; }
{IDENTIFIER} { yylval.row = cons(yytext); return IDENTIFIER; }
{CONSTANT} { yylval.row = cons(yytext); return CONSTANT; }
. { printf("UNKNOWN "); ECHO; printf("\n");exit(1); }
%%
```

```
==> ../lab1a/p1.crs <==
let x: i32;
let y: i32;

x = readI32();
y = readI32();

while y != 0 {
  let z: i32;
  z = x % y;
  x = y;
  y = z;
}

// This is a comment
print("Gcd is ", x);

==> output <==

program
    program
        program
            program
                program
                    program
                        program
                            program
                            statement
                                decl_statement
                                    let
                                    x
                                    :
                                    type
                                        i32
                                ;
                        statement
                            decl_statement
                                let
                                y
                                :
                                type
                                    i32
                            ;
                    statement
                        input
                            x
                            =
```

```
                readI32
                (
                )
            ;
        statement
            input
                y
                =
                readI32
                (
                )
            ;
    statement
        loop
            while
            expression
                term
                    y
                operator
                    !=
                expression
                    term
                        0
            {
            program
                program
                    program
                        program
                            program
                                program
                                statement
                                    decl_statement
                                        let
                                        z
                                        :
                                        type
                                            i32
                                    ;
                            statement
                                assignment_statement
                                    z
                                    =
                                    expression
                                        term
                                            x
                                        operator
                                            %
```

```
                                    expression
                                        term
                                            y
                            ;
                    statement
                        assignment_statement
                            x
                            =
                            expression
                                term
                                    y
                            ;
                statement
                    assignment_statement
                        y
                        =
                        expression
                            term
                                z
                    ;
            }
statement
    output
        print
        (
        output_expression
            output_expression
                expression
                    term
                        "Gcd is "
            ,
            expression
                term
                    x
        )
    ;
```