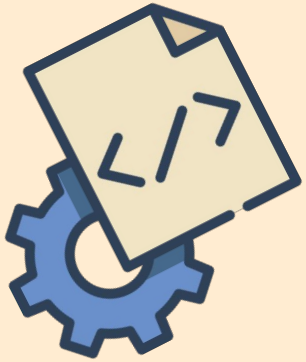
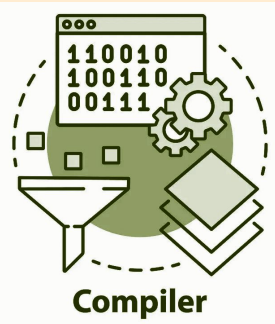


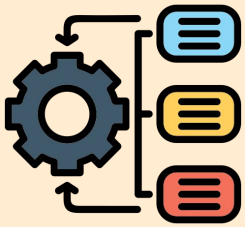
MINI C COMPILER

(LLVM codegen back-end)

CMPE 152
Thuong Mai - Ivan Gomez
SAN JOSE STATE UNIVERSITY
Fall 2025



BACKGROUND



System Architecture (Two-pass Core)

1. Lexical Analyzer
2. Syntax Analyzer
3. Semantic Analyzer
4. IR Code Generator
5. Code Generator

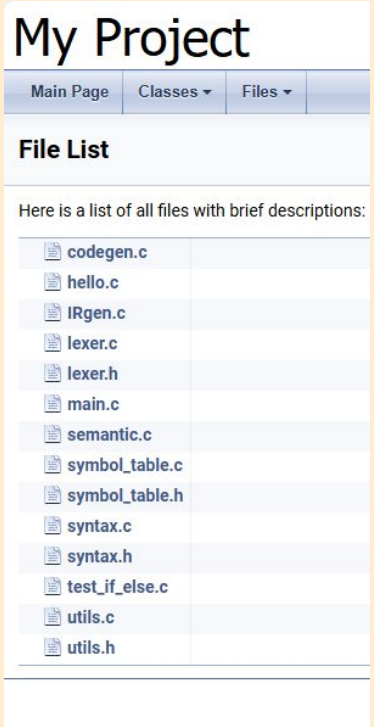
Key Features:

- Type Interference: deduce variable types and reduce explicit type annotations.
- Modular Architecture: facilitate easy extension and maintenance of compiler components.
- Error Handling: provide comprehensive error messages and recovery for easy debugging process.

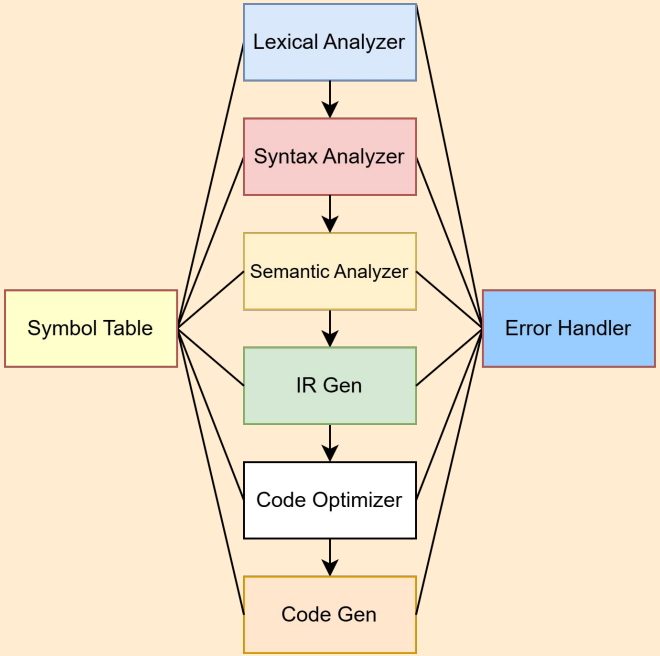
Tools and Environments:

- Programming language: C
- Environments: LLVM toolchain 12.0+, Git
- Dependencies: LLVM docs (code generator)
- Doxygen (file management)
- **Build and Run in Linux with clang– / llvm**

File List



Phases of Compiler



What does this MINI C COMPILER do?

TABLE OF CONTENTS

(1) Source Code	Blocks of Codes
(2) Main Operations	2a) token generation 2b) token classification 2c) parsing and lexical analyzers 2d) RL and RegEx 2e) NFA ⇔ RegEx 2f) CFG and CFL 2g) Applications of CNF and GNF 2h) assembly code 2i) dataflow analysis 2j) memory management.
(3) Analysis	Error Handlers, Symbol Tables, Data and Memory Analysis of Two-Pass Compiler
(4) Test Case	Three Test Cases
(5) Limitations	MINI Compilers and References

Our compiler project supports a broad set of operators, control-flow keywords, and expression forms (C-like).
Two-pass compiler: first pass collects info (symbols, types, forward declarations), second pass uses that info to generate code or interpret. More flexible and realistic, richer features.

An overview of the major language features

Category	Short Meaning	Examples
Assignment	set / update	`a=b`, `a+=b`, `a-=b`, `a*=b`, `a/=b`
Increment / Decrement	add 1, sub 1	`++a`, `a++`, `--a`, `a-- (pre & post)`
Arithmetic	math ops	`a+b`, `a-b`, `a*b`, `a/b`
Bitwise	bit ops	`~a`, `a&b`, `a
Logical	true/false ops	`!a`, `a&&b`, `a
Comparison	compare values	`==`, `!=`, `<`, `>`, `<=`, `>=`
Other	misc ops	`a(...)` (call), `a,b` (comma), `(type)a` (cast)

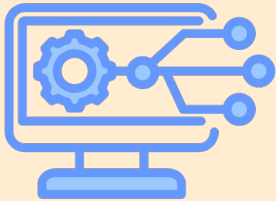
Data Flow Architecture

Lexical Analyzer
(output: token stream)

- Generate streams of Tokens, convert raw texts into tokens,
- Token Classifications: scans text and slices it into lexemes and assigns each lexemes a token kind (keyword, Identifier, Number, Operator, Punctuation)

Syntax Analyzer
(output: AST tree)

- Parse trees: known as syntax tree that shows how a grammar produce a string of tokens. CFG and CFL.
- Left/ Right Derivation trees.

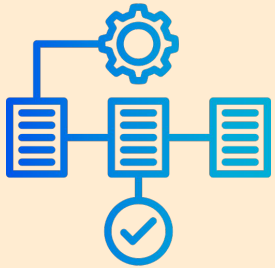


Semantic Analyzer
(output: annotated AST)

- The semantic analysis of a regular expression refers to its meaning such as the language (set of strings) it denotes. It is the mapping from the expression (syntax) to the set of strings (meaning).

LLVM IR Generator
(output: LLVM IR)

- The stages in a compiler that takes the AST and performs semantic checks and produce LLVM IR (Intermediate Representations)



Code Generator
(output: executable code)

- Code generation is the final phase of a compiler. It takes the intermediate representation (IR) and translates it into target code — typically machine code, assembly, or bytecode (for virtual machines).
- => converting the compiler's internal representation of a program into executable instructions for a target architecture.

(1) Lexer Block Code

TokenType

```
typedef enum {
    TOK_INT,     // integer literal
    TOK_ID,      // id for single var
    TOK_EQ , // equal ==
    TOK_PLUS,    // '+'
    TOK_COMPOUND_PLUS, //+=
    TOK_INCREMENT,    // '++'
    TOK_DECREMENT,    // '--'
    TOK_MINUS,   // '-'
    TOK_COMPOUND_MINUS, //-=
    TOK_MUL,   // '*'
    TOK_COMPOUND_MUL, //*=
    TOK_DIV,  // '/'
    TOK_COMPOUND_DIV, // /=
    TOK_LPAREN,  // '('
    TOK_RPAREN,  // ')'
    TOK_EOF,      // end of file
    TOK_GREATER, // >  NEWLY ADDED
    TOK_GREATER_EQ, // >=  NEWLY ADDED
    TOK_LESS,// <
    TOK_LESS_EQ,// <=
    TOK_SHIFT_LEFT, // <<
    TOK_SHIFT_RIGHT,// >>
    TOK_BITWISE_AND, //  bitwise binary &
    TOK_AND, // LOGICAL AND &&
    TOK_BITWISE_XOR, //^
    TOK_BITWISE_OR, // |
    TOK_OR, // LOGICAL OR ||
    TOK_NOT, // LOGICAL NOT !
    TOK_NOT_EQ, // NOT EQ !=
    TOK_BITWISE_NOT, //~ 1'S COMPLEMENT
    TOK_IF, // "if" // newly add here
    TOK_ELSE, // else
    TOK_BREAK, //break
    TOK_DO, // do
    TOK_WHILE, // "while"
    TOK_RETURN, // "return"
    TOK_INT_VAR, // "int"
    TOK_LCURLY, // "{"
    TOK_RCURLY, // "}"
    TOK_SEMI, // ";"
    TOK_COMMA, //,
    TOK_VOID, // "void"
    TOK_FLOAT,
    TOK_DBL,
    TOK_ELIF, // "else if"
    TOK_FOR, // "for"
    //TOK_QUESTION, //?
    //TOK_COLON //  :
} TokenType;
```

Token struct

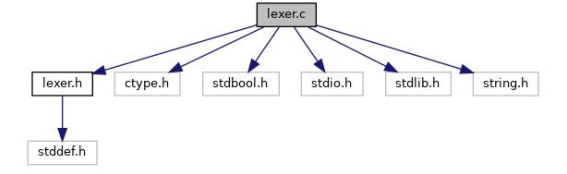
```
typedef struct {
    TokenType type;
    int value;       // onl
    float float_val;
    char lexeme[128];
    size_t pos;      // ind
    size_t id_no;
} Token;
```

Lexer struct (for parser)

```
typedef struct {
    const char *input;
    size_t pos;
    char current;
    size_t id_cnt;
} Lexer;
```

Lexer dependencies

```
#include "lexer.h"
#include <ctype.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```




```
while (lex->current) {
    // checking if white space
    if (isspace((unsigned char)lex->current)) {skip_whitespace(lex);continue;}
    // checking if number
    if (isdigit((unsigned char)lex->current)) {return number(lex);}
    //checking if identifiers.
    if (isalpha((unsigned char)lex->current) || lex->current == '_') {return identifier(lex); }
```

```
size_t p = lex->pos;
char ch = lex->current;
advance(lex);
// lord, this is for single char operations, please do double char later. -Thuong
switch (ch) {
    case '+': if (lex->current == '+') {advance(lex); return token_gen(TOK_INCREMENT, 0, "++", p);}
              else if (lex->current == '=') {advance(lex); return token_gen(TOK_COMPOUND_PLUS, 0, "+=", p);}
              else { return token_gen(TOK_PLUS, 0, "+" , p); }
    case '-': if (lex->current == '-') { advance(lex); return token_gen(TOK_DECREMENT, 0, "--", p); }
              else if (lex->current == '=') { advance(lex); return token_gen(TOK_COMPOUND_MINUS, 0, "-=", p); }
              else {return token_gen(TOK_MINUS, 0,"-", p);}
    case '*': if (lex->current == '=') { advance(lex); return token_gen(TOK_COMPOUND_MUL, 0, "*=", p); }
              return token_gen(TOK_MUL, 0, "*",p);
    case '/': if (lex->current == '=') { advance(lex); return token_gen(TOK_COMPOUND_DIV, 0, "/=", p); }
              return token_gen(TOK_DIV, 0, "/",p);
    case '(': return token_gen(TOK_LPAREN, 0,"(", p);
    case ')': return token_gen(TOK_RPAREN, 0, ")",p);
    case '{': return token_gen(TOK_LCURLY, 0,"{", p);
    case '}': return token_gen(TOK_RCURLY, 0, "}",p);
    case '=': if (lex->current == '=') { advance(lex); return token_gen(TOK_EQ, 0,"==" ,p); }
              else {return token_gen(TOK_ASSIGN, 0,"=" ,p);}
    case '>': if (lex->current == '>') { advance(lex); return token_gen(TOK_SHIFT_RIGHT, 0,">>" ,p); }
              else if (lex->current == '=') {advance(lex); return token_gen(TOK_GREATER_EQ, 0, ">=", p);}
              else {return token_gen(TOK_GREATER, 0,">" ,p);}
    case '<': if (lex->current == '<') { advance(lex); return token_gen(TOK_SHIFT_LEFT, 0,"<<" ,p); }
              else if (lex->current == '=') {advance(lex); return token_gen(TOK_LESS_EQ, 0, "<=", p);}
              else {return token_gen(TOK_LESS, 0,"<" ,p);}
    case '&': if (lex->current == '&') { advance(lex); return token_gen(TOK_AND, 0,"&&" ,p); }
              else {return token_gen(TOK_BITWISE_AND, 0,"&" ,p);}
    case '^': return token_gen(TOK_BITWISE_XOR, 0,"^" ,p);
    case '~': return token_gen(TOK_BITWISE_NOT, 0,"~" ,p);
    case '?': return token_gen(TOK_QUESTION, 0,"?" ,p);
    case ':': return token_gen(TOK_COLON, 0,":" ,p);
    case '!': if (lex->current == '=') { advance(lex); return token_gen(TOK_NOT_EQ, 0,"!=" ,p); }
              else {return token_gen(TOK_NOT, 0,"!" ,p);}
    case '|': if (lex->current == '|') { advance(lex); return token_gen(TOK_OR, 0,"||" ,p); }
              else {return token_gen(TOK_BITWISE_OR, 0,"|" ,p);}
    case ';': return token_gen(TOK_SEMI, 0, ";", p);
    default:  // unknown char: consume until end; caller can treat as END

        return token_gen(TOK_EOF, 0, "EOF",p);
}
```

(1) PARSER (main.c) SOURCE CODE

main.c

```c
int main(void)
{
    char *examples[] = {"int choice(int a, int b, int choice)
        {
        for (int i = 0; i < 10; i++)
        { a + b; }\n
        if (a + b)
        { return a;} else {return b;}"};
    for (int i = 0; examples[i]; ++i)
    {
        Lexer lx;
        print_sep();
        puts("\n============INPUT STREAM (TEXT)==============================");
        printf("%s\n", examples[i]);
        lexer_init(&lx, examples[i]);
        puts("\n============TOKEN STREAM (TOKENIZE)==========================");
        print_tokens(&lx);
        lexer_init(&lx, examples[i]);
        puts("\n============TOKEN CLASSIFICATIONS (TokenType)================");
        print_token_classification(&lx);
        print_sep();
        lexer_init(&lx, examples[i]);
        Parser ps;
        parser_init(&ps, &lx);
        while (ps.current.type != TOK_EOF)
        {
            AST *tree = NULL;
            if (is_function_definition(&ps))
            {
                tree = parse_fn(&ps);
                if (tree && tree->type == AST_FUNC)
                {
                    puts("\n================PARSE TREE (AST)============================");
                    print_tree_better(tree);
                    printf("\n============Executing function==========================\n");
                    double result = eval_function_call(tree, NULL, 0);
                    printf("Function returned: %.2f\n", result);
                    codegen_run(tree);
                    print_sep();
                }
            }
            else
            {
                tree = parse_statement(&ps);
                if (tree)
                {
                    double result = eval_ast_assignment(tree);
                    printf("result = %.2f\n", result);
                    puts("\n============Parse tree=======================================");
                    print_tree_better(tree);
                }
            }
            print_sep();
            if (tree)
                free_ast(tree);
        }
    }
```

syntax libraries

Classes

struct ASTWrapper

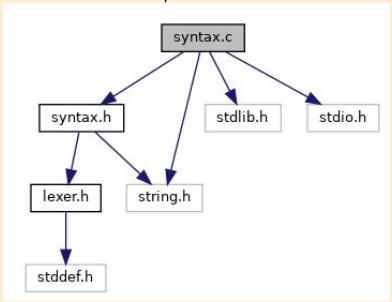
Macros

#define FN_STMTS_CAP 128
#define FN_PARAM_CAP 16
#define MAX_HEIGHT 1000
#define INFINITY (1<<20)

Typedefs

typedef struct ASTWrapper ASTWrapper

```c
#include "syntax.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define FN_STMTS_CAP 128
#define FN_PARAM_CAP 16
static int error = 0;
```




```c
// Check if a function definition
if (is_function_definition(&ps))
{tree = parse_fn(&ps);}
else {tree =
parse_statement(&ps);}
```

```c
static AST* parse_definition(Parser*);
  static AST* parse_term(Parser *ps);
  static AST* parse_block(Parser*);
  static AST* parse_return(Parser*);
  static AST* parse_id(Parser*);
```

```c
typedef struct
{
    Lexer *lexer;
    Token current;
    Token next;
} Parser;
```

Data Structure →

```c
typedef struct AST
{
    ASTType type;
    double value;
    Token op;
    char name[128];
    struct AST *left;
    struct AST *right;
    struct AST *cond;
    struct AST *stmnts[128];
    struct AST *params[16];
    struct AST *init; // init

    int used;
    int stmnt_cnt; // what is
    int param_cnt; // what is
} AST;
```

PARSER IF

```
 // Handle IF LOGIC
if(tok.type == TOK_IF){
AST* if_AST = (AST*)calloc(1,
sizeof(AST));
if_AST->type = AST_IF;
eat(ps, TOK_IF);
eat(ps, TOK_LPAREN);
if_AST->cond = parse_expr(ps);
eat(ps, TOK_RPAREN);
eat(ps, TOK_LCURLY);
if_AST->left = parse_statement(ps);
if_AST->right = NULL;
eat(ps, TOK_RCURLY);
if (ps->current.type == TOK_ELSE||
ps->current.type == TOK_ELIF
){if_AST->right = parse_term (ps);}
return if_AST;
}
```

PARSER WHILE

```
 // Handle WHILE LOGIC
if (tok.type == TOK_WHILE){
AST* while_AST = (AST*)calloc(1,
sizeof(AST));
while_AST->type = AST_WHILE;
eat(ps, TOK_WHILE);
eat(ps, TOK_LPAREN);
while_AST->cond = parse_expr(ps);
eat(ps, TOK_RPAREN);
eat(ps, TOK_LCURLY);
while_AST->left =
parse_statement(ps);
while_AST->right = NULL;
eat(ps, TOK_RCURLY);

return while_AST;
}
```

PARSER FOR

```
if (tok.type == TOK_FOR){
AST* for_AST = (AST*)calloc(1,
sizeof(AST));
for_AST->type = AST_FOR;
eat(ps, TOK_FOR);
eat(ps, TOK_LPAREN);
for_AST->init =
parse_definition(ps);
eat_SEMI (ps);
for_AST->cond = parse_expr(ps); //
greater or condition
eat_SEMI (ps);
for_AST->left = parse_expr(ps);
eat(ps, TOK_RPAREN);
eat(ps, TOK_LCURLY);
for_AST->right=parse_statement(ps);
//bodu
eat(ps, TOK_RCURLY);
return for_AST;
}
```

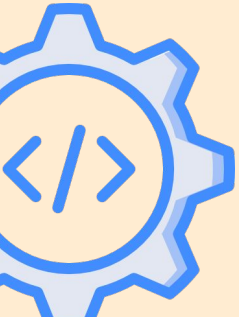
(2a) Understanding Lex

Token Generation : the lexer scans the sequence of raw code and turn it into tokens for parsers to understand. We implemented a type struct for token to store TokenType, integer and float values, lexeme, and its position.

```
typedef struct {
    TokenType type;
    int value;      //
    float float_val;
    char lexeme[128];
    size_t pos;     //
    size_t id_no;
} Token;
```

In our design, we built the Token struct to have:

- **`TokenType`** tells what kind of token it is.
- **`int value`** hold the numeric values of token (TOK_INT)
- **`float float_val`** contains the floating point value (TOK_FLOAT)
- **`lexeme`** carries the exact sequence of characters from the source code that formed the token (even numeric values), original text is kept here.



Matching Patterns using Regular Expressions

- Identifier → [A-Za-z_][A-Za-z0-9_]*
- Integer → 0|[1-9][0-9]*
- Float → [0-9]+"."[0-9]+
- Operators → ==, <=, !=, <, >, +, -, ;, etc.

=> Lexer uses these patterns to decide what kinds of token the current characters form.

TokenType

```
typedef enum {
    TOK_INT,     // integer literal
    TOK_ID,      // id for single var
    TOK_EQ , // equal ==
    TOK_PLUS,    // '+'
    TOK_COMPOUND_PLUS, //+=
    TOK_INCREMENT,    // '++'
    TOK_DECREMENT,    // '--'
    TOK_MINUS,   // '-'
    TOK_COMPOUND_MINUS, //-=
    TOK_MUL,   // '*'
    TOK_COMPOUND_MUL, //*=
    TOK_DIV,  // '/'
    TOK_COMPOUND_DIV, // /=
    TOK_LPAREN,  // '('
    TOK_RPAREN,  // ')'
    TOK_EOF,      // end of file
    TOK_GREATER, // >  NEWLY ADDED
    TOK_GREATER_EQ, // >=  NEWLY ADDED
    TOK_LESS,// <
    TOK_LESS_EQ,// <=
    TOK_SHIFT_LEFT, // <<
    TOK_SHIFT_RIGHT,// >>
    TOK_BITWISE_AND, //  bitwise binary &
    TOK_AND, // LOGICAL AND &&
    TOK_BITWISE_XOR, //^
    TOK_BITWISE_OR, // |
    TOK_OR, // LOGICAL OR ||
    TOK_NOT, // LOGICAL NOT !
    TOK_NOT_EQ, // NOT EQ !=
    TOK_BITWISE_NOT, //~ 1'S COMPLEMENT
    TOK_IF, // "if" // newly add here
    TOK_ELSE, // else
    TOK_BREAK, //break
    TOK_DO, // do
    TOK_WHILE, // "while"
    TOK_RETURN, // "return"
    TOK_INT_VAR, // "int"
    TOK_LCURLY, // "{"
    TOK_RCURLY, // "}"
    TOK_SEMI, // ";"
    TOK_COMMA, //,
    TOK_VOID, // "void"
    TOK_FLOAT,
    TOK_DBL,
    TOK_ELIF, // "else if"
    TOK_FOR, // "for"
    //TOK_QUESTION, //?
    //TOK_COLON //  :
} TokenType;
```

(2b) Classification of tokens

(i) Keywords

TOK_IF
TOK_ELSE
TOK_BREAK
TOK_DO
TOK_WHILE
TOK_FOR
TOK_RETURN
TOK_INT_VAR
TOK_FLOAT_VAR
TOK VOID
TOK_ELIF

(ii) Identifiers

TOK_ID
TOK_INT
TOK_FLOAT
TOK_DBL

(iii) Punctuation

TOK_LPAREN
TOK_RPAREN
TOK_LCURLY
TOK_RCURLY
TOK_SEMI
TOK_COMMA
TOK_EOF

48 Total Token Types

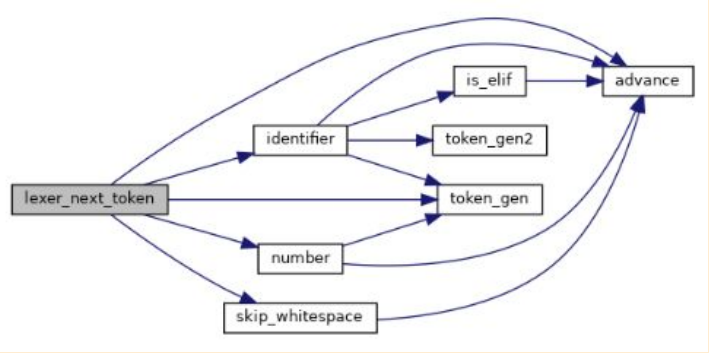
(iv) Operators

TOK_ASSIGN
TOK_EQ
TOK_PLUS
TOK_COMPOUND_PLUS
TOK_INCREMENT
TOK_DECREMENT
TOK_MINUS
TOK_COMPOUND_MINUS

TOK_MUL
TOK_COMPOUND_MUL
TOK_DIV
TOK_COMPOUND_DIV
TOK_GREATER
TOK_GREATER_EQ
TOK_LESS
TOK_LESS_EQ

TOK_SHIFT_LEFT
TOK_SHIFT_RIGHT
TOK_BITWISE_AND
TOK_AND
TOK_BITWISE_XOR
TOK_OR
TOK_NOT
TOK_NOT_EQ
TOK_BITWISE_NOT

(2c)Lexical Analyzer



The role of the Lexer:

A lexer (short for lexical analyzer) is the first stage of a compiler or interpreter. Its main role is to convert raw source code (a stream of characters) into a sequence of tokens, which are meaningful units for the next stage — the parser.

Macros and Functions

Macros

`#define PUSH(ch) do { if (bi + 1 < sizeof(buf)) buf[bi++] = (char)(ch); } while (0)`

Functions

static void	advance (Lexer *lex)
static void	skip_whitespace (Lexer *lex)
static Token	token_gen (TokenType t, int v, const char *lex, size_t p)
static Token	token_gen2 (TokenType t, int v, const char *lex, size_t p, size_t id_no)
static Token	number (Lexer *lex)
int	is_key (char *str)
int	is_elif (char *buf, size_t *buf_idx, Lexer *lex)
static Token	identifier (Lexer *lex)
void	lexer_init (Lexer *lex, char *input)
Token	lexer_next_token (Lexer *lex)

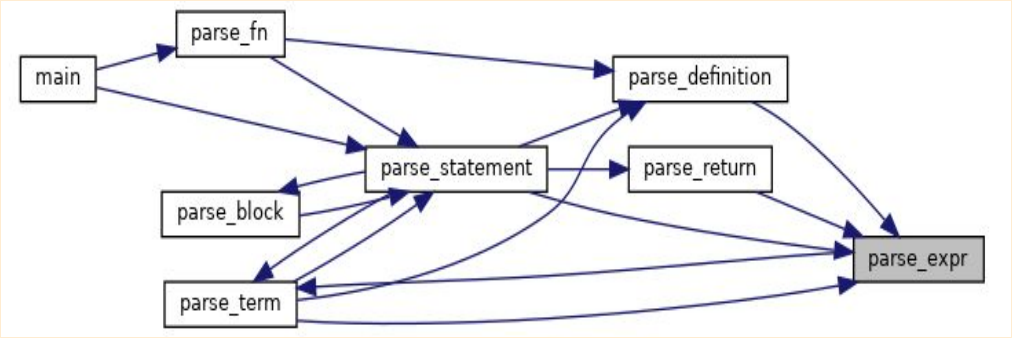
Lexical Analysis

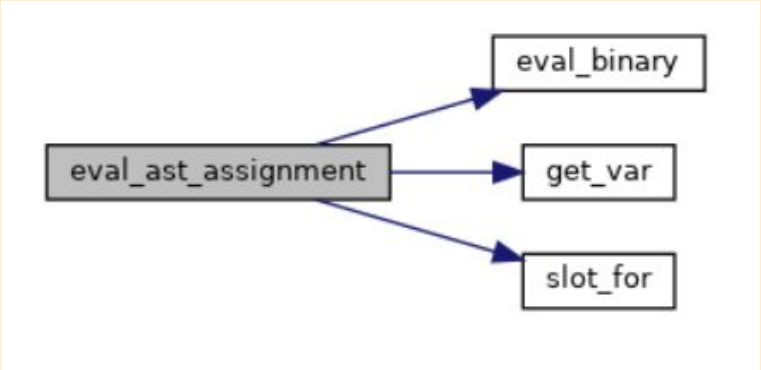
Step 1: Read the input stream (text source code) `void lexer_init`
Step 2: The lexer scans source text and slices it into lexemes(substrings). Group characters into tokens.
Step 3: Discard whitespace, comments and advance to next char. `void skip_whitespace(Lexer *lex)`, `void advance(Lexer *lex)`
Step 4: Then assigns each lexeme a token kind (Keyword, Identifier, Number, Operator, Punctuation).
Step 5: Return EOF if end of file detected, save the token streams.
Step 6: Provide Tokens to the Parser.

(2c) Syntax Analyzer

The role of the Parser:

A parser is the second stage of a compiler. Its main role is to check that the stream of tokens generated by the lexer follows the grammar defined by us, alerts us of any errors, and generates a parse tree.





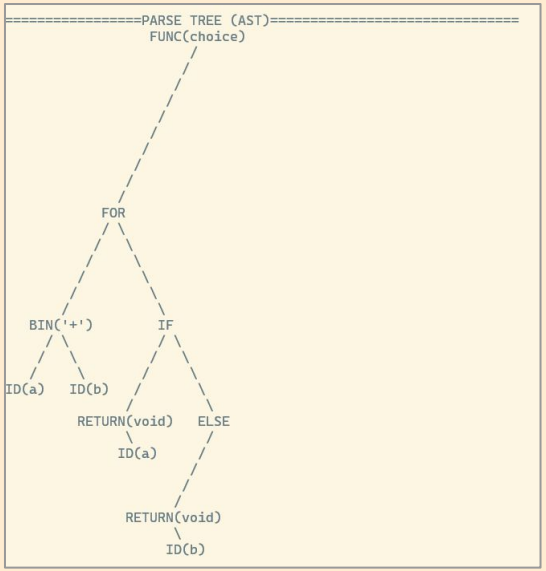
Syntax Analysis

In our code, the syntax analyzer is defined by two big parts:

(1) The parse functions consisting of parse_statement(), parse_expression(), and parse_term(). Together these make up the programming equivalent of following a set of productions

(2) The Abstract Syntax Tree builder consisting of eval_ast_assignment() and eval_binary(). As the name suggests, it builds up the abstract syntax tree based on the AST nodes generated by the parser

(2c) (i) Parse Trees




```
typedef struct AST
{
    ASTType type;
    double value;
    Token op;
    char name[128];
    struct AST *left;
    struct AST *right;
    struct AST *stmnts[128];
    struct AST *params[16];
    struct AST *init; // init var

    int used;
    int stmnt_cnt;
    int param_cnt;
} AST;
```

Hierarchical Structure

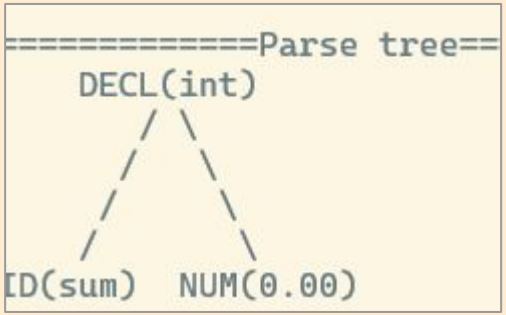
function->statements->expression -> term

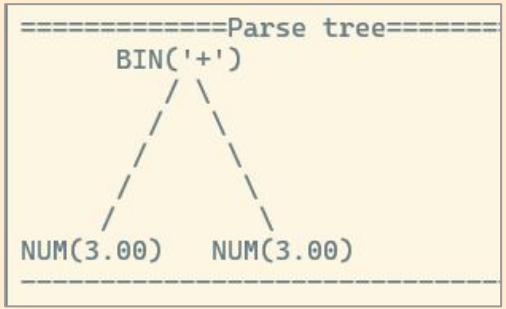
- Functions - multiple statements
- Statements
- Expressions
- Terms are atomic (ID, Literals, etc)

- AST nodes are powerful
- Generic left, right are given meaning through the ASTType

Helper methods for expression:
parse_block()
parse_return()
parse_id()
parse_definition()

(2c) (i) Parse Trees (cont)






```
}
static AST* parse_definition(Parser* ps){
    Token type_tok = ps->current;  // Save the type token

    if (ps->current.type != TOK_INT_VAR &&
        ps->current.type != TOK_FLOAT_VAR &&
        ps->current.type != TOK_VOID) {
        error_at(ps, &ps->current, "expected type specifie
        return NULL;
    }
    eat(ps, ps->current.type);

    char name[64];
    strncpy(name, ps->current.lexeme, sizeof(name)-1);
    name[63] = '\0';
    eat(ps, TOK_ID);

    // ALWAYS create a DECLARATION node
    AST* node = calloc(1, sizeof(*node));
    node->type = AST_DECLARATION;  // ← ALWAYS DECLARATION
    node->op = type_tok;            // Store the type (TOK
    node->left = make_id(name);

    // Check for initializer
    if (ps->current.type == TOK_ASSIGN) {
        eat(ps, TOK_ASSIGN);
        node->right = parse_expr(ps); // Store initializer
    } else {
        node->right = NULL; // No initializer
    }

    return node;
}
```

Example (Variable Definition):

- Left node is an AST_ID type
- Right node is an AST_INT type

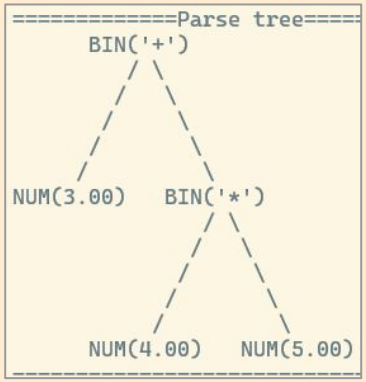
Example (Binary Operation):

- Left Node is an AST_INT
- Right Node is an AST_INT

Enough information and structure for the semantic analyzer to make sense of the tree

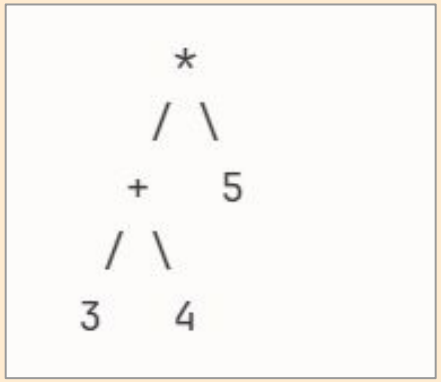
(2c) (ii) Parse Trees (Ambiguity)

Resolution of Ambiguity



3 + 4 * 5

vs



Our Implementation

Alternative Tree

- Arithmetic expressions can be produced by ambiguous grammars
- In normal cases, operator precedence is used to disambiguate
- Precedence used in our implementation for only (), {}
- Simple straight line parsing

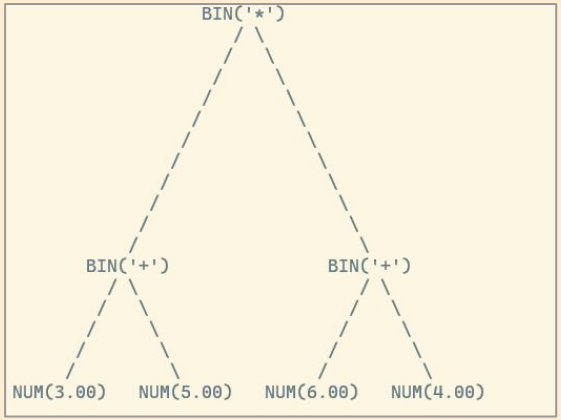
(2c) (iii) Left/Right Derivation Trees

- Custom written recursive descent top-down parser looks 1 token ahead
- This makes it an LL(1) grammar
- LL(1) grammars use leftmost derivations creating Left Derivation Trees
- Compare against Yacc/Bison which are LALR(1) producing rightmost derivations (Right Derivation Trees)

Example Code, Tree, and Grammar Production

```
}
// PARSE EXPRESSION ##################
AST* parse_expr(Parser *ps) {
    AST *node = parse_term(ps); // eat
    if (ps->current.type == TOK_INCREM
        Token post = ps->current;
        eat(ps, ps->current.type);
        AST *postfix = calloc(1, sized
        postfix->type = AST_UNARY;
        postfix->op = post;
        postfix->left = node;
        node = postfix;
    }
    while (is_binOp(ps->current.type))
        Token op = ps->current;
        eat(ps, op.type);
        AST *expr = parse_expr(ps);
        AST *binOp = (AST*)calloc(1, s
        binOp->type = AST_BINOP;
        binOp->op = op;
        binOp->left = node;
        binOp->right = expr;
        node = binOp;
    }
    if (is_assignment(ps)) {
        Token op = ps->current;
        eat(ps, ps->current.type);
        AST* expr = parse_expr(ps);
        AST* assign = (AST*)calloc(1,
        assign->op = op;
        assign->type = AST_ASSIGN;
        assign->left = node;
        assign->right = expr;
        node = assign;
    }
```

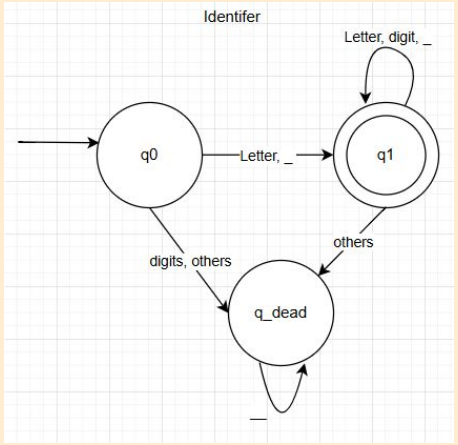
```
> (3 + 5) * (6 + 4);
```

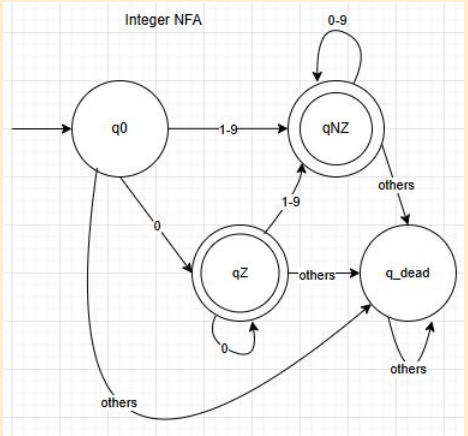


expression ::= term ('++' | '--')? (binaryOp expression)* (assignmentOp expression)?
term ::= for | if | else| while | unaryOp | '(' expression ')' | INT | ID

(2e) COMPREHENSIVE NFA ⇔ REGEX

Diagrams





RegEx

NFA 1: Identifiers/Keywords
⇔ **RegEx:[A-Za-z_][A-Za-z0-9_]***

Formal Description:
M =(Q, Σ,δ, q0,F)
Q= q0, q1, q_dead
Σ = {L, d, _, others)
q0 = q0
F= q1

NFA 2: Integer literal
⇔ **RegEx: 0|[1-9][0-9]***

Formal Description:
M =(Q, Σ, δ, q0, F)
Q= q0, qZ, qNZ, q_dead
Σ = {[0-9], others)
q0 = q0
F= q1

Transition Tables

Transition Table NFA 1 - Identifiers

State	L	d	_	others
q0	q1*	q_dead	q1	q_dead
q1	q1*	q1*	q1*	q_dead
q_dead	q_dead	q_dead	q_dead	q_dead

Transition Table NFA 2 - Int Literals

State	0	1-9	others
q0	qZ	qNZ*	q_dead
qZ*	qZ*	qNZ*	q_dead
qNZ*	qNZ*	qNZ*	q_dead
q_dead	q_dead	q_dead	q_dead

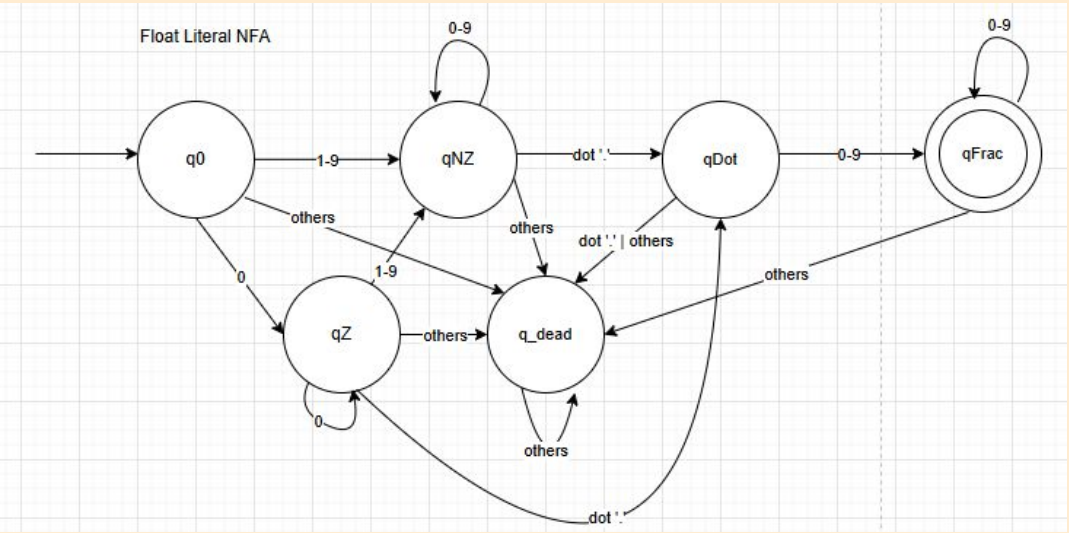
(2e) COMPREHENSIVE NFA ⇔ REGEX (cont)

NFA3: Float literals ⇔ **RegEx: [0-9]+ "." [0-9]+**

Formal Description:

M =(Q, Σ,δ, q0,F)
Q= q0, qZ, qNZ, qFrac, q_dead
Σ = {[0-9], '.', others)
q0 = q0
F= qFrac

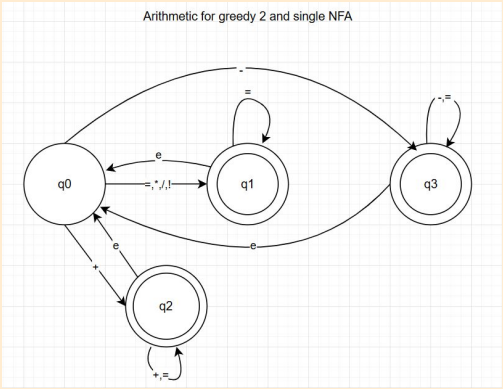
Diagrams

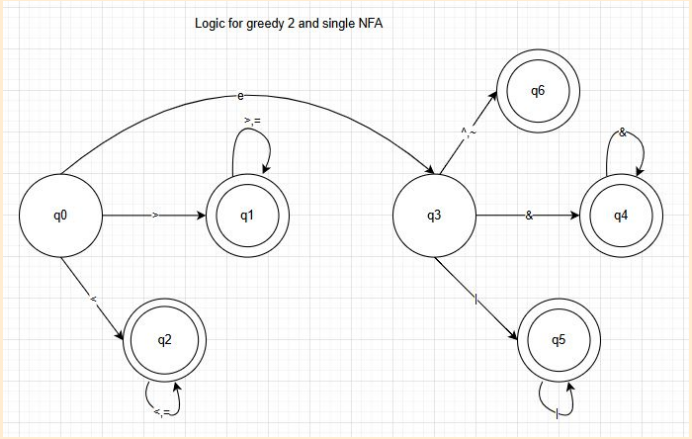


Transition Table NFA 3 - Float Literals				
State	**0**	**1-9**	**dot '.'**	**others**
q0	qZ	qNZ	-	q_dead
qZ	qZ	qNZ	qDot	q_dead
qNZ	qNZ	qNZ	qDot	q_dead
qDot	qFrac*	qFrac*	q_dead	q_dead
q_dead	q_dead	q_dead	q_dead	q_dead

(2e) COMPREHENSIVE NFA ⇔ REGEX (cont)

NFA 4: Operators (greedy 2-char > 1-char) ⇔ **RegEx: Targets:** ==|!=|<=|>=|<<|>>|&&|||| **and singletons** = ||<|>|&|!|+|-|*|/|~|(|)|{|};

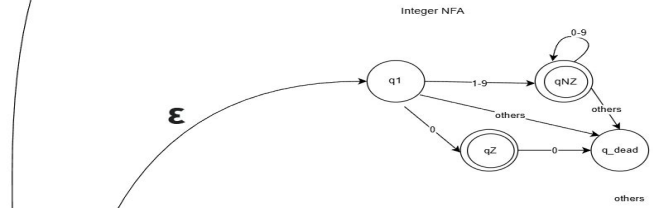


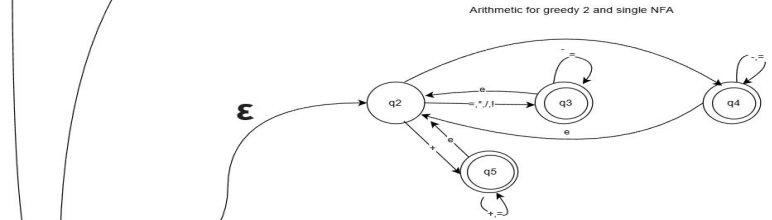


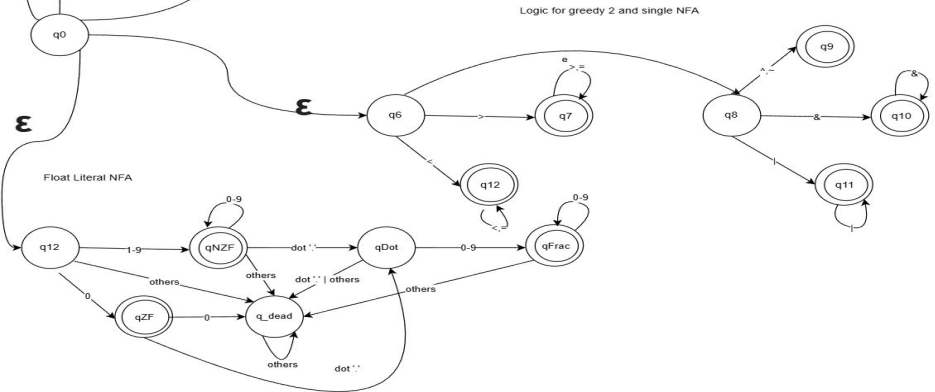
How it it implemented in code? => Switches

```
case '+':       if (lex->current == '+') {advance(lex); return token_gen(TOK_INCREMENT, 0, "++", p);}
                else if (lex->current == '=') {advance(lex); return token_gen(TOK_COMPOUND_PLUS, 0, "+=",
p);}
                else { return token_gen(TOK_PLUS, 0, "+" , p); }
case '-':       if (lex->current == '-') { advance(lex); return token_gen(TOK_DECREMENT, 0, "--", p); }
                else if (lex->current == '=') { advance(lex); return token_gen(TOK_COMPOUND_MINUS, 0, "-=",
p);}
                else {return token_gen(TOK_MINUS, 0,"-", p);}
case '*':       if (lex->current == '=') { advance(lex); return token_gen(TOK_COMPOUND_MUL, 0, "*=", p); }
                return token_gen(TOK_MUL, 0, "*",p);
case '/':       if (lex->current == '=') { advance(lex); return token_gen(TOK_COMPOUND_DIV, 0, "/=", p); }
                return token_gen(TOK_DIV, 0, "/",p);
case '=':       if (lex->current == '=') { advance(lex); return token_gen(TOK_EQ, 0,"==" ,p); }
                else {return token_gen(TOK_ASSIGN, 0,"=" ,p);}
case '>':       if (lex->current == '>') { advance(lex); return token_gen(TOK_SHIFT_RIGHT, 0,">>" ,p); }
                else if (lex->current == '=') {advance(lex); return token_gen(TOK_GREATER_EQ, 0, ">=", p);}
                else {return token_gen(TOK_GREATER, 0,">" ,p);}
case '<':       if (lex->current == '<') { advance(lex); return token_gen(TOK_SHIFT_LEFT, 0,"<<" ,p); }
                else if (lex->current == '=') {advance(lex); return token_gen(TOK_LESS_EQ, 0, "<=", p);}
                else {return token_gen(TOK_LESS, 0,"<" ,p);}
case '&':       if (lex->current == '&') { advance(lex); return token_gen(TOK_AND, 0,"&&" ,p); }
                else {return token_gen(TOK_BITWISE_AND, 0,"&" ,p);}
case '!':       if (lex->current == '=') { advance(lex); return token_gen(TOK_NOT_EQ, 0,"!=" ,p); }
                else {return token_gen(TOK_NOT, 0,"!" ,p);}
case '|':       if (lex->current == '|') { advance(lex); return token_gen(TOK_OR, 0,"||" ,p); }
                else {return token_gen(TOK_BITWISE_OR, 0,"|" ,p);}
case '^':       return token_gen(TOK_BITWISE_XOR, 0,"^" ,p);
```

(2e) NFA⇔ RegEx (cont)







Transition Table Total NFA						
State	a-zA-Z_	0	1-9	'.'	others	ε
q0	-	-	-	-	-	{q1, q2, q6, q12, q13}
q1	-	qZ	qNZ		-	-
qZ	-	-	-	-	-	-
qNZ	-	qNZ	qNZ	-	-	-
q12	-	qZF	qNZF	-	-	-
qZF	-	-	-	qDot	-	-
qNZF	-	qNZF	qNZF	qDot	-	-
qDot	-	qFrac	qFrac	-	-	-
qFrac	-	qFrac	qFrac	-	-	-
q13	q14	-	-	-	-	-
q14	q14	q14	q14	-	-	-

Transition Table Total NFA								
State	+	*	/	e	-	=	!	ε
q2	q5	q3	q3	-	q4	q3	q3	
q3	-	-	-	q2	-	q3	-	-
q4	-	-	-	-	q4	q4	-	-
q5	q5	-	-	-	-	q5	-	-

Transition Table Total NFA									
State	<	>	=	e	^	~	&	|	ε
q6	q12	q7	-	q8	-	-	-	-	-
q7	-	q7	q7	-	-	-	-	-	-
q8	-	-	-	-	q9	q9	q10	q11	-
q9	-	-	-	-	-	-	-	-	-
q10	-	-	-	-	-	-	q10	-	-
q11	-	-	-	-	-	-	-	q11	-
q12	q12	-	q12	-	-	-	-	-	-

(2f) BNF and EBNF - SYNTAX ANALYZER

BNF

function ::= is_returnType "(" param_list ")" "{" statement "}" ;
param_list ::= definition _stmt ;
definition_stmt ::= definition | expr_stmt ;
definition ::= "int" | "void" | "float" ;

statement ::=definition _stmt | term | expr_stmt | return_stmt | block | empty ;
term ::= for_stmt | if_stmt | while_stmt | () | INT | FLOAT | is_unOP;
for_stmt ::= "for" "(" definition";" expression ";" expression ")" "{" statement "}" ;
if_stmt ::= "if" "(" expression ")" "{" statement "}" "else" "{" term "}" ;
while_stmt ::= "while" "(" expression")" "{" statement "}" ;
expr_stmt ::= expression ";" ;
expression ::= is_binOP | is_assigment;
return_stmt ::= "return" expression ";" ;
block ::= "{" statement "}" ;
empty ::= ";" ;

is_unOP ::= "—" | "++" | "-" | "!" | "~" ;
is_binOP ::="+" | "-" | "&" | "|" | "<" | ">" | "||" | "&&" | "<=" | ">=" | "<<" | ">>" | "==" | "!=" | "*" | "/" ;
is_assignment ::= "=" | "-=" | "+=" | " *= " | " /=" ;
ID ::= letter { letter | digit | "_" } ;
INT ::= digit { digit } ;
FLOAT ::= digit { digit } "." digit { digit } ;

EBNF

function ::= returnType '(' {returnType ID} {',} ')' '{' {statement} '}' ;

statement ::= (definition | return | expression)?';' | if | for | while | block
definition ::= returnType ID '=' expression
return ::= 'return' expression ';'
expression ::= term ('++' | '--')? (binaryOp expression)* (assignmentOp expression)?
term ::= for | if | else | while | unaryOp | '(' expression ')' | INT | ID

for ::= 'for' '(' definition ';' expression ';' expression ';' ')' '{' statement '}'

if ::= 'if' '(' expression ')' '{' statement '}' ('else' term)?

else ::= 'else' '{' statement '}'
while-statement ::= 'while' '(' expression ')' '{' statement '}'

returnType ::= 'int' | 'double' | 'float' | 'void'
binaryOp ::= '+' | '-' | '*' | '/' | '<' | '>' | '<=' | '>=' | '==' | '!=' | '&' | '|' | '<<' | '>>' | '&&' | '||'
unaryOp ::= '~' | '!' | '-' | '++' | '--'
assignmentOp ::= '=' | '+=' | '-=' | '*=' | '/='

(2g) Application of Chomsky's & GNF

Example:

We want
expr -> expr + expr |int | id
Variables: {expr}
T: {+, int, id}

Problem! Left Recursion

Solution: Convert to CNF -> GNF to eliminate left recursion

By applying these forms to each of our grammars, the compiler is able to parse essential expressions

(i) Chomsky's Normal Form:

expr -> EC_1 | int | id
C_1 -> +E

(ii) Greibach Normal Form:

E -> int | int Z_E | id | id Z_e
Z_E-> C_1 | C_1Z_E
C_1 -> +E

(2g) Application of Chomsky's & GNF (cont)

```
}
// PARSE EXPRESSION ##################################################################
AST* parse_expr(Parser *ps) {
    AST *node = parse_term(ps); // eats the term token
    if (ps->current.type == TOK_INCREMENT || ps->current.type == TOK_DECREMENT) {
        Token post = ps->current;
        eat(ps, ps->current.type);
        AST *postfix = calloc(1, sizeof(AST));
        postfix->type = AST_UNARY;    // new node type
        postfix->op = post;
        postfix->left = node;         // store operand on the left
        node = postfix;
    }
    while (is_binOp(ps->current.type)) {
        Token op = ps->current;
        eat(ps, op.type);
        AST *expr = parse_expr(ps);
        AST *binOp = (AST*)calloc(1, sizeof(AST));
        binOp->type = AST_BINOP;
        binOp->op = op;
        binOp->left = node;
        binOp->right = expr;
        node = binOp;
    }
    if (is_assignment(ps)) {
        Token op = ps->current;
        eat(ps, ps->current.type);
        AST* expr = parse_expr(ps);
        AST* assign = (AST*)calloc(1, sizeof(AST));
        assign->op = op;
        assign->type = AST_ASSIGN;
        assign->left = node;
        assign->right = expr;
        node = assign;
    }
    return node;
}
```

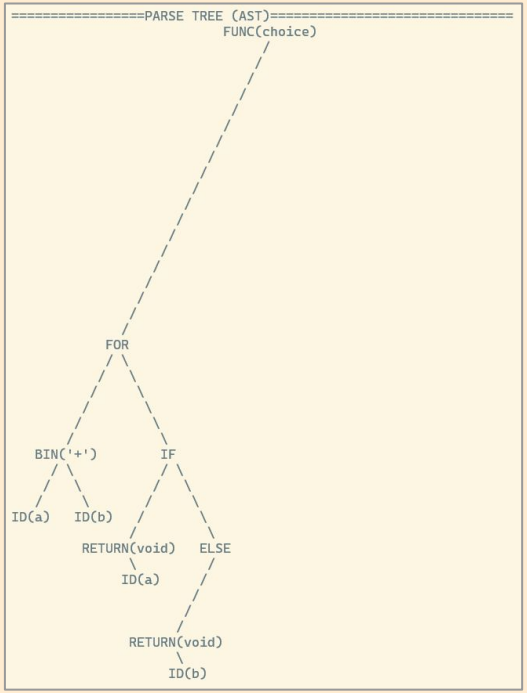
Note that the first line in the parse_expr() function is parse_term. A direct application of Greibach Normal Form.

(2h) Assembly Code Generation

```
Registered Targets:
  aarch64    - AArch64 (little endian)
  aarch64_32 - AArch64 (little endian ILP32)
  aarch64_be - AArch64 (big endian)
  amdgcn     - AMD GCN GPUs
  arm        - ARM
  arm64      - ARM64 (little endian)
  arm64_32   - ARM64 (little endian ILP32)
  armeb      - ARM (big endian)
  avr        - Atmel AVR Microcontroller
  bpf        - BPF (host endian)
  bpfeb      - BPF (big endian)
  bpfel      - BPF (little endian)
  hexagon    - Hexagon
  lanai      - Lanai
  m68k       - Motorola 68000 family
  mips       - MIPS (32-bit big endian)
  mips64     - MIPS (64-bit big endian)
  mips64el   - MIPS (64-bit little endian)
  mipsel     - MIPS (32-bit little endian)
  msp430     - MSP430 [experimental]
  nvptx      - NVIDIA PTX 32-bit
  nvptx64    - NVIDIA PTX 64-bit
  ppc32      - PowerPC 32
  ppc32le    - PowerPC 32 LE
  ppc64      - PowerPC 64
  ppc64le    - PowerPC 64 LE
  r600       - AMD GPUs HD2XXX-HD6XXX
  riscv32    - 32-bit RISC-V
  riscv64    - 64-bit RISC-V
  sparc      - Sparc
  sparcel    - Sparc LE
  sparcv9    - Sparc V9
  systemz    - SystemZ
  thumb      - Thumb
  thumbeb    - Thumb (big endian)
  ve         - VE
  wasm32     - WebAssembly 32-bit
  wasm64     - WebAssembly 64-bit
  x86        - 32-bit X86: Pentium-Pro and above
  x86-64     - 64-bit X86: EM64T and AMD64
  xcore      - XCore
```

Fig. 1 Available Assembly Languages

- LLVM serves as a powerful Intermediate Representation
- We can take our custom-defined syntax tree and convert it into LLVM




```
Function name: choice
params[0] = a
params[1] = b
params[2] = choice
i32 %aPlacing a into the hashmap
i32 %bPlacing b into the hashmap
i32 %choicePlacing choice into the hashmap
Node Type: 12
  %i = phi i32 [ 0, %entry ]Node Type: 3
  %addtmp = add i32 %a, %b
i32 1
Node Type: 5
  %iftmp = phi i32 [ %a, %then ], [ %b, %else ]

=== Generated IR ===
; ModuleID = 'JIT'
source_filename = "JIT"
target datalayout = "E-m:m-p:32:32-i8:8:32-i16:16:32-i64:64-n32-S64"

define i32 @choice(i32 %a, i32 %b, i32 %choice) {
entry:
  br label %loop

loop:                                             ; preds = %loop, %entry
  %lsr.iv = phi i32 [ %lsr.iv.next, %loop ], [ -1, %entry ]
  %addtmp = add i32 %a, %b
  %lsr.iv.next = add nsw i32 %lsr.iv, 1
  %lttmp = icmp slt i32 %lsr.iv.next, 10
  br i1 %lttmp, label %loop, label %afterloop

afterloop:                                        ; preds = %loop
  %addtmp1 = add i32 %a, %b
  %ifcond = icmp ne i32 %addtmp1, 0
  br i1 %ifcond, label %ifcont, label %else

else:                                             ; preds = %afterloop
  br label %ifcont

ifcont:                                           ; preds = %afterloop, %else
  %iftmp = phi i32 [ %b, %else ], [ %a, %afterloop ]
  ret i32 %iftmp
}
```

(2h) Assembly Code Generation (cont)

```
=============INPUT STREAM (TEXT)===============================
int choice(int a, int b, int choice) { for (int i = 0; i < 10; i++) { a + b; }
 if (a + b) { return a;} else {return b;}}
```

```
~/Complete_Compiler [main]
$ cat mips_out.s
        .text
        .abicalls
        .option pic0
        .section        .mdebug.abi32,"",@progbits
        .nan    legacy
        .text
        .file   "JIT"
        .globl  choice
        .p2align        2
        .type   choice,@function
        .set    nomicromips
        .set    nomips16
        .ent    choice
choice:
        .cfi_startproc
        .frame  $sp,0,$ra
        .mask   0x00000000,0
        .fmask  0x00000000,0
        .set    noreorder
        .set    nomacro
        .set    noat
        addiu   $2, $zero, -1
$BB0_1:
        addiu   $2, $2, 1
        slti    $1, $2, 10
        bnez    $1, $BB0_1
        nop
        addu    $1, $4, $5
        beqz    $1, $BB0_4
        nop
        jr      $ra
        move    $2, $4
$BB0_4:
        move    $4, $5
        jr      $ra
        move    $2, $4
        .set    at
        .set    macro
        .set    reorder
        .end    choice
$func_end0:
        .size   choice, ($func_end0)-choice
        .cfi_endproc

        .section        ".note.GNU-stack","",@progbits
        .text
```

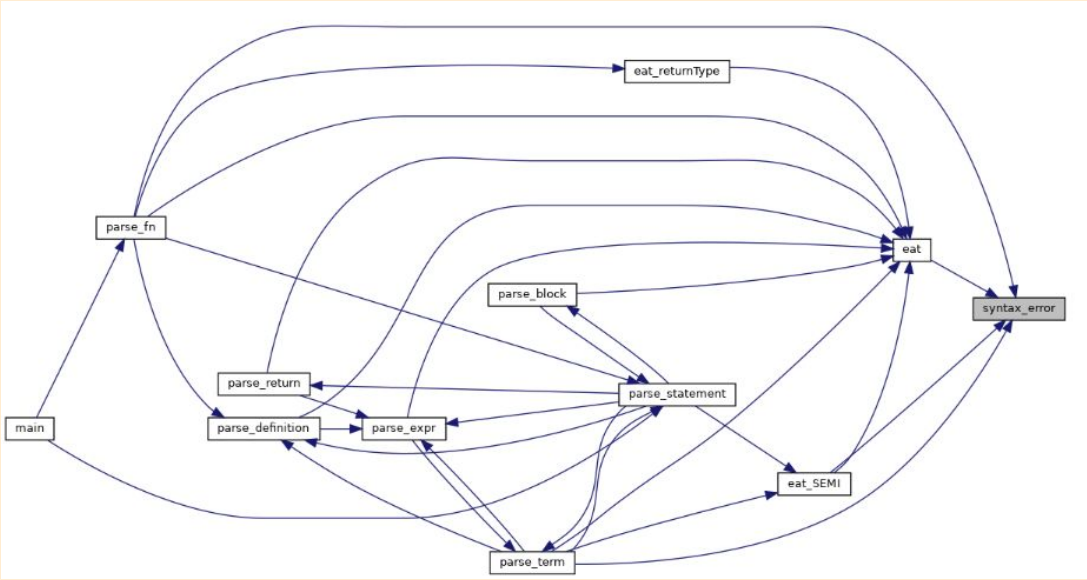
Fig. 1 MIPS Assembly Output

```
~/Complete_Compiler [main]
$ cat native_out.s
        .text
        .file   "JIT"
        .globl  choice
        .p2align        4, 0x90
        .type   choice,@function
choice:
        .cfi_startproc
        movl    %edi, %eax
        movl    $-1, %ecx
        .p2align        4, 0x90
.LBB0_1:
        addl    $1, %ecx
        cmpl    $10, %ecx
        jl      .LBB0_1
        movl    %eax, %ecx
        addl    %esi, %ecx
        je      .LBB0_3
        retq
.LBB0_3:
        movl    %esi, %eax
        retq
.Lfunc_end0:
        .size   choice, .Lfunc_end0-choice
        .cfi_endproc

        .section        ".note.GNU-stack","",@progbits
```

Fig. 2 x86 Assembly Output

(3)Analysis: Error Handlers

The error handler finds mistakes, explains where and what they are, and lets the compiler continue instead of stopping immediately.



Case 1: Undefined Variables

```
> h;
 <ID,0>  <;>

 End of File at pos=2, "EOF"
 EAT: expecting 1, got 42 (;)
EAT: expecting 42, got 16 (EOF)
>> Echo (get_var) : Undefined variable 'h'
>> Echo (get_var) : Currently registered variables:
  [0] name='a', type=38
  [1] name='b', type=38
  [2] name='choice', type=38
  [3] name='i', type=38
```

Case 2: Wrong function call

```
> int main ( { int a=10;}
 <int>  <ID,0>  <(>  <{>  <int>  <ID,1>  <=>  int(10)  <;>  <}>

 End of File at pos=23, "EOF"
 EAT: expecting 38, got 1 (main)
EAT: expecting 1, got 14 (()
EAT: expecting 14, got 40 ({)
>> Echo (parse_fn): Parsing parameter, current token: {
Syntax error: expected type specifier ('int' or 'float')
  at token '{'
```

```
>> Echo (parse_fn): Adding statement, type: 13
>> Echo (parse_for): Parsing statement, current token: cal
EAT: expecting 1, got 2 (=)
EAT: expecting 2, got 1 (cal)
EAT: expecting 1, got 10 (*)
EAT: expecting 10, got 1 (k)
EAT: expecting 1, got 42 (;)
EAT: expecting 42, got 16 (EOF)
>> Echo (parse_fn): Adding statement, type: 2
>> Echo (parse_for): Total nodes (params + statements): 2
Syntax error at 48: Syntax Error >> parse_for : expected '}' before end of file (got token 16)
```

(3)Analysis: Symbol Tables

A symbol table is a data structure used by a compiler to keep track of information about identifiers in the source program. It stores variables names, function names, types, scopes, memory locations, attributes (param, return types, etc).

Symbol table is used differently in each phase:
-> Parser recognizes the declarations and insert it into the symbol table.
-> Semantic analyzer will do type checking, scopes, and meanings.
-> LLVM IR code gen will use the information in the symbol table to produce correct instructions (to map identifiers to memory locations)

Example Snippet of Code

```
int choice(int a, int b, int choice)
{
for (int i = 0; i < 10; i++)
        { a + b; }
if (a + b)
        { return a;}
else {return b;}
}
```

```
========== CODEGEN START ==========
Function name: choice
params[0] = a
params[1] = b
params[2] = choice
i32 %aPlacing a into the hashmap
i32 %bPlacing b into the hashmap
i32 %choicePlacing choice into the hashmap
Node Type: 12
  %i = phi i32 [ 0, %entry ]Node Type: 3
  %addtmp = add i32 %a, %b
i32 1
Node Type: 5
  %iftmp = phi i32 [ %a, %then ], [ %b, %else ]
```

(3)Analysis: LLVM Symbol Tables

How is LLVM helpful?

- LLVM has llvm-nm, is a tool that shows the symbols inside a compiled file and reads the metadata stored in the binary file.
- Using this tool to check if the compiler is generating the right functions or to debug the linker errors or duplicate and missing undefined definitions.
- We can see from the LLVM Symbol Tables => **Memory address/offset - Symbol Types - Symbol names**

```
0000000000012010 D _edata
000000000001b7e0 B _end
00000000000d8dc T _fini
0000000000004000 T _init
0000000000005910 T _start
00000000000094c0 T advance
0000000000008750 t build_wrapper_tree
                 U calloc@GLIBC_2.2.5
000000000000c660 T codegen
000000000000cc10 T codegen_binary_expr
000000000000cea0 T codegen_call
000000000000c740 T codegen_for
000000000000d030 T codegen_function
000000000000ca00 T codegen_if
000000000000c590 T codegen_number
000000000000c3f0 T codegen_prototype
000000000000d1d0 T codegen_run
000000000000c390 T codegen_var
0000000000012058 b completed.0
00000000000081b0 t compute_edge_lengths
00000000000007fc0 t compute_lprofile
00000000000080b0 t compute_rprofile
000000000000d6e0 T create_map
0000000000012000 W data_start
0000000000005940 t deregister_tm_clones
0000000000008be0 t eat
0000000000009620 T eat_SEMI
0000000000008bc0 t eat_SEMI.part.0
00000000000097a0 T error_at
```

```
000000000000b200 T eval_ast_assignment
000000000000b780 T eval_ast_decl
000000000000af90 T eval_binary
0000000000005f60 t eval_function_call.constprop.0
                 U exit@GLIBC_2.2.5
                 U fgets@GLIBC_2.2.5
00000000000059f0 t frame_dummy
                 U free@GLIBC_2.2.5
000000000000b960 T free_ast
0000000000008d70 t free_wrapper_tree
000000000000d690 T hash
0000000000009420 T is_binOp
0000000000006690 T is_elif
0000000000006120 T is_function_definition
0000000000005cd0 t is_function_definition.part.0
00000000000065b0 T is_key
0000000000009440 T is_unOp
00000000000067e0 T lexer_init
0000000000006810 T lexer_next_token
0000000000013020 b lprofile
00000000000052e0 T main
0000000000009460 T make_fn
                 U malloc@GLIBC_2.2.5
000000000000d870 T map_clear
000000000000d7e0 T map_get
000000000000d700 T map_put
00000000000062b0 t number
0000000000009c00 t parse_definition
0000000000009800 T parse_expr
000000000000a820 T parse_fn
0000000000009e10 T parse_statement
0000000000009fe0 t parse_term
```

```
000000000000ae30 T parser_init
0000000000008560 t print_level
0000000000012060 b print_next
0000000000005c90 t print_sep
0000000000006160 T print_tokens
000000000000c1c0 T print_tree_better
                 U putc@GLIBC_2.2.5
                 U puts@GLIBC_2.2.5
0000000000005970 t register_tm_clones
0000000000012080 b rprofile
0000000000008a80 t slot_for
0000000000012040 B stderr@GLIBC_2.2.5
0000000000012050 B stdin@GLIBC_2.2.5
0000000000012020 B stdout@GLIBC_2.2.5
                 U strcmp@GLIBC_2.2.5
                 U strcspn@GLIBC_2.2.5
                 U strdup@GLIBC_2.2.5
                 U strlen@GLIBC_2.2.5
                 U strncpy@GLIBC_2.2.5
                 U strtod@GLIBC_2.2.5
0000000000008b80 t syntax_error.isra.0
0000000000005a00 t token_type_name
0000000000013fc0 b vars
```

(3)Analysis (cont)

Data analysis verifies the correctness of variables, types, and expressions using the symbol table and semantic rules.
Memory management ensures that AST nodes, symbol tables, variables, and generated code use memory safely and efficiently, with proper allocation and deallocation.

- **malloc (size)** allocates a block of memory of the given size, but the memory is not initialized=> creating map entries for the symbol tables, structures for lexer and parser.
- **calloc(n, size)** is utilized to allocate memory for n elements, and initializes all bytes to zero => memory safety for default values.
- **free_ast() / map_clear()** to later free the memory to prevent memory leakage

Together, these components allow the compiler to execute or compile programs correctly without leaks, undefined behavior, or type errors.

(4) Limitations of Mini Compilers

- **Arithmetic operations** are parsed and executed through a functional syntax tree, but **only a subset** currently supports LLVM assembly code generation.
- **Function definitions** are implemented, but **function calls are not yet supported**.
- **Operator precedence is missing**, causing expressions to be evaluated in the wrong order.
- **Error handling is unstable** and only partially implemented, leading to inconsistent recovery and occasional crashes.

Test Case 1 (if- elif - else)

Code Snippet:

```
int x=5;
if (x>23) { int y=1;}
else if (x>2) { int z=2;}
else { z=1;}
```

=>OUTPUT z=2

Token Generation

```
==========INPUT STREAM (TEXT)==========
int x = 5;
if (x > 23) {int y = 1;}
 else if(x>2) {int z = 2;}
 else {z=1;}


==========TOKEN STREAM (TOKENIZE)==========
 <int>  <ID,0>  <=>  int(5)  <;>  <if>  <(>  <ID,1>  <>>  int(23)  <)>  <{>  <int>  <ID,2>  <=>  int(1)  <;>
<}>  <else if>  <(>  <ID,3>  <>>  int(2)  <)>  <{>  <int>  <ID,4>  <=>  int(2)  <;>  <}>  <else>  <{>  <ID,5>
 <=>  int(1)  <;>  <}>

 End of File at pos=77, "EOF"
```

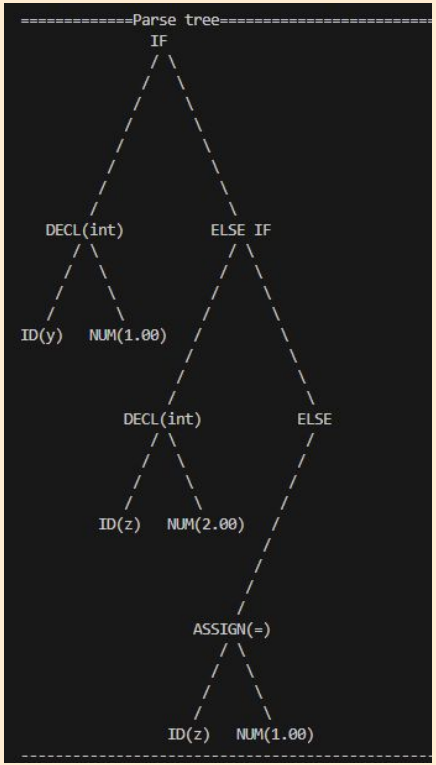
Output

```
> z;
 <ID,0>  <;>

 End of File at pos=2, "EOF"
 EAT: expecting 1, got 42 (;)
EAT: expecting 42, got 16 (EOF)
>> Echo (get_var) : name: 'z', value:'2.000000'
 = 2.00
== Parse tree down eher ==
ID(z)
```

Token Classification

```
==========TOKEN CLASSIFICATIONS (TokenType)==========
Keyword: 'int' at pos -> 0
Identifier: x pos 4
Operator: '=' at pos -> 6
Number: '5' at pos -> 8
Punctuation: ';' at pos -> 9
Keyword: 'if' at pos -> 11
Punctuation: '(' at pos -> 14
Identifier: x pos 15
Operator: '>' at pos -> 17
Number: '23' at pos -> 19
Punctuation: ')' at pos -> 21
Punctuation: '{' at pos -> 23
Keyword: 'int' at pos -> 24
Identifier: y pos 28
Operator: '=' at pos -> 30
Number: '1' at pos -> 32
Punctuation: ';' at pos -> 33
Punctuation: '}' at pos -> 34
Keyword: 'else if' at pos -> 37
Punctuation: '(' at pos -> 44
Identifier: x pos 45
Operator: '>' at pos -> 46
Number: '2' at pos -> 47
Punctuation: ')' at pos -> 48
Punctuation: '{' at pos -> 50
Keyword: 'int' at pos -> 51
Identifier: z pos 55
Operator: '=' at pos -> 57
Number: '2' at pos -> 59
Punctuation: ';' at pos -> 60
Punctuation: '}' at pos -> 61
Keyword: 'else' at pos -> 65
Punctuation: '{' at pos -> 70
Identifier: z pos 71
Operator: '=' at pos -> 72
Number: '1' at pos -> 73
Punctuation: ';' at pos -> 74
Punctuation: '}' at pos -> 75
```

AST Tree



Test Case 2 (maths)

```
=============INPUT STREAM (TEXT)================================
(1 + 2) * 3 - 4 / 2;

============TOKEN STREAM (TOKENIZE)============================
 <(> int(1) <+> int(2) <)> <*> int(3) <-> int(4) </> int(2) <;>

 End of File at pos=20, "EOF"
```

```
============TOKEN CLASSIFICATIONS (TokenType)=======
Punctuation: '(' at pos -> 0
Number: '1' at pos -> 1
Operator: '+' at pos -> 3
Number: '2' at pos -> 5
Punctuation: ')' at pos -> 6
Operator: '*' at pos -> 8
Number: '3' at pos -> 10
Operator: '-' at pos -> 12
Number: '4' at pos -> 14
Operator: '/' at pos -> 16
Number: '2' at pos -> 18
Punctuation: ';' at pos -> 19
--------------------------------------------------
EAT: expecting 14, got 0 (1)
EAT: expecting 0, got 4 (+)
EAT: expecting 4, got 0 (2)
EAT: expecting 0, got 15 ())
EAT: expecting 15, got 10 (*)
EAT: expecting 10, got 0 (3)
EAT: expecting 0, got 8 (-)
EAT: expecting 8, got 0 (4)
EAT: expecting 0, got 12 (/)
EAT: expecting 12, got 0 (2)
EAT: expecting 0, got 42 (;)
EAT: expecting 42, got 16 (EOF)
result = 3.00
```

```
=============Parse tree================================
                 BIN('*')
                /        \
               /          \
          BIN('+')       BIN('-')
           /   \          /    \
    NUM(1.00) NUM(2.00) NUM(3.00) BIN('/')
                                   /    \
                           NUM(4.00)  NUM(2.00)
------------------------------------------------------
```

Code Snippet:

(1+2)*3 -4/2;

OUTPUT => 3

Output

```
result = 3.00
```

Code Gen

```
========== CODEGEN START ==========

=== Generated IR ===
; ModuleID = 'JIT'
source_filename = "JIT"
target datalayout = "E-m:m-p:32:32-i8:8:32-i16:16:32-i64:64-n32-S64"

define i32 @calculate(i32 %0, i32 %1) {
entry:
  %addtmp = add i32 %0, %1
  ret i32 %addtmp
}
========== CODEGEN END ==========
```

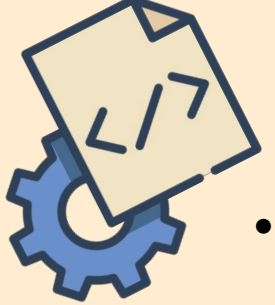
Test Case 3 (Functions)

```
============INPUT STREAM (TEXT)==============================
int sum(int a, int b, int c, float d){
        a + b + c;
        3+3;
        return a + b;
}
```

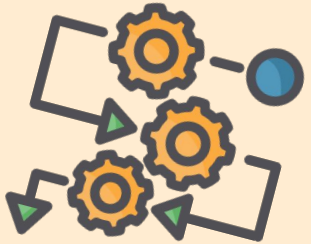
```
============TOKEN CLASSIFICATIONS (TokenType)=================
Keyword: 'int' at pos -> 0
Identifier: sum pos 4
Punctuation: '(' at pos -> 7
Keyword: 'int' at pos -> 8
Identifier: a pos 12
Punctuation: ',' at pos -> 13
Keyword: 'int' at pos -> 15
Identifier: b pos 19
Punctuation: ',' at pos -> 20
Keyword: 'int' at pos -> 22
Identifier: c pos 26
Punctuation: ',' at pos -> 27
Keyword: 'float' at pos -> 29
Identifier: d pos 35
Punctuation: ')' at pos -> 36
Punctuation: '{' at pos -> 37
Identifier: a pos 40
Operator: '+' at pos -> 42
Identifier: b pos 44
Operator: '+' at pos -> 46
Identifier: c pos 48
Punctuation: ';' at pos -> 49
Number: '3' at pos -> 52
Operator: '+' at pos -> 53
Number: '3' at pos -> 54
Punctuation: ';' at pos -> 55
Keyword: 'return' at pos -> 58
Identifier: a pos 65
Operator: '+' at pos -> 67
Identifier: b pos 69
Punctuation: ';' at pos -> 70
Punctuation: '}' at pos -> 72
```

```
================PARSE TREE (AST)========
                    FUNC(sum)
                   /
                  /
                 /
                /
               /
              /
             /
            /
           /
          /
         /
        /
       /
      /
     /
    /
   /
  /
 /
/
    BIN('+')
    / \
   /   \
  /     \
ID(a)   BIN('+')
        / \
       /   \
      /     \
     /       \
    /         \
   /           \
NUM(3.00)   RETURN(void)
                \
                 \
                  \
                BIN('+')
                / \
               /   \
              /     \
             /       \
            /         \
           /           \
        ID(a)   ID(b)
```

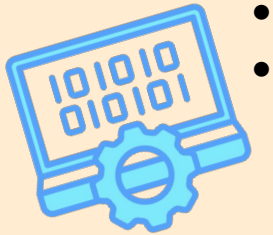
```
========== CODEGEN START ==========
Function name: sum
params[0] = a
params[1] = b
params[2] = c
params[3] = d
i32 %aPlacing a into the hashmap
i32 %bPlacing b into the hashmap
i32 %cPlacing c into the hashmap
float %dPlacing d into the hashmap
Node Type: 3
  %addtmp1 = add i32 %a, %addtmp
Node Type: 3
i32 6
Node Type: 8
  %addtmp2 = add i32 %a, %b

=== Generated IR ===
; ModuleID = 'JIT'
source_filename = "JIT"
target datalayout = "E-m:m-p:32:32-i8:8:32-i16:16:32-i64:64-n32-S64"

define i32 @sum(i32 %a, i32 %b, i32 %c, float %d) {
entry:
  %addtmp = add i32 %b, %c
  %addtmp1 = add i32 %a, %addtmp
  %addtmp2 = add i32 %a, %b
  ret i32 %addtmp2
}
========== CODEGEN END ==========
```



Conclusion



- A **custom scanner and parser** supporting up to **48 token types**, capable of recognizing functions, (int and float) variables, and types.
- A **semantic analyzer** that validates types, operations, and tokens to ensure correct program meaning.
- A full **two-pass architecture** (front-end + LLVM back-end) supporting functions, assignments, arithmetic, and logical operations.
- A robust design that separates **definition-gathering** (Pass 1) from **usage-checking** (Pass 2), improving maintainability, accuracy, and resilience to errors.
- Clear benefits of the two-pass approach, including simpler analyzer design, resolution of forward-reference issues, enhanced error handling, more precise semantic checks, and a solid foundation for code generation.
- Successful integration with **LLVM**, enabling real backend code generation.
- Using git, linux OS, clang, and doxyfile for code management and developments.



References

[1]

A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, & tools*, Second edition. Boston: Pearson/Addison Wesley, 2007.

[2]

“LLVM Tutorial: Table of Contents — LLVM 22.0.0git documentation.” Accessed: Dec. 01, 2025. [Online]. Available: https://llvm.org/docs/tutorial/

[3]

A. W. Appel, *Modern compiler implementation in C*, Digital print. Cambridge: Cambridge Univ. Press, 2010.