Zhejiang Unversity Experiment Report

- **Course Name: Compile Principle**
- **Student Name: Feng Xiang, Wang Junzhe, Xu Zhen**
- **Student ID: 3180103426, 3180103011, 3180105504**
- **Department: Computer Science & Technology**
- **Major: Computer Science & Technology**
- **Instructor: Feng Yan**

Nano C Compiler

Nano C Compiler

The C programming language compiler with extremely limited functionality😄

Visit the tree visualizer to see what abstract tree the developer is lately developing.

Usage:

```
# This will: read the source code, get tokens, generate parse tree, generate AST, send AST to server, emit IR, store IR to file, compile IR to Assembly, compile Assembly to Executable
python src/nanoirgen.py -i <input_file_path> -o <output_IR_path> -g
# Executable/Assemble file names/path are derived from <output_IR_path>
```

Example:

```
# under folder: compiler
python src/nanoirgen.py -i samples/quicksort.c -o results/quicksort.ll -g -e .exe
# executable saved at: ./results/

# run the executable
./results/quicksort.exe

# check output (in return values)
echo $lastExitCode

# if you see 1, the code works (see ./samples/quicksort.c)
# if you see 0, the code failed to quicksort
```

More usage about **nanoirgen.py** and **nanoyacc.py**

```
# python src/nanoirgen.py -h
usage: nanoirgen.py [-h] [-input INPUT] [-output OUTPUT] [-target TARGET] [-url URL] [-tree TREE] [-generate] [-ext EXT]

optional arguments:
  -h, --help      show this help message and exit
  -input INPUT
  -output OUTPUT
  -target TARGET
  -url URL
  -tree TREE
  -generate       Whether to generate the target machine code
  -ext EXT        Executable file extension
```

```
# python src/nanoyacc.py -h
usage: nanoyacc.py [-h] [-input INPUT] [-tree TREE] [-url URL]

optional arguments:
  -h, --help    show this help message and exit
  -input INPUT
  -tree TREE
  -url URL
```

If You're Viewing the PDF Format of This File

IMPORTANT: To better grasp the ability of our Abstract Syntax Tree visualizer, go to the GitHub of this repo to see for yourself.

IMPORTANT: The visualizer is hosted at: my server

H2

Chapter 1 - Lexical Analysis

§1.1 Token Specification

H3

- Types: `int`, `long`, `double`, `float`, `char`, `void`, `[]`, **pointer**
- Control Flow: `if`, `else`, `while`, `for`, `continue`, `break`, `do-while`
- Function: `return`, **typed functions**, **scope**(`{}` `;`)
- Operators (with precedence):
  1. `()` `[]` Function Call, Array Subscription
  2. `-` `+` `++` `--` `!` `&` `*` `~` `(type)` Negation, Positive Number, Minus Minus, Plus Plus, Logical Not, Bitwise And, Get Element Of Pointer, Bitwise Not, Type Casting
  3. `*` `/` `%` Times, Divide, Modulus
  4. `+` `-` Plus, Minus
  5. `<<` `>>` Shift Left, Shift Right
  6. `<` `≤` `>` `≤` Less Than, Less Than Or Equal To, Greater Than, Greater Than Or Equal To
  7. `==` `≠` Equality, Inequality
  8. `&` Get Element Pointer
  9. `^` Bitwise XOR
  10. `|` Bitwise Or

11. `&&` Logical And
12. `||` Logical Or
13. `?:` Conditional Expression
14. `=` `<<=` `>>=` `&=` `|=` `^=` `+=` `-=` `/=` `*=` `%=` Assignment Operation

- Comment:
  1. `//` (one-line comment)
  2. `/* */` (multi-line comment)

§1.2 Token Definition

Token List:

H3

```
keywords = (
    'INT', 'LONG', 'FLOAT', 'DOUBLE', 'CHAR',
    'UNSIGNED', 'CONST', 'VOID', "STATIC"
    'ENUM', 'STRUCT', 'UNION', 'IF', 'ELSE',
    'DO', 'WHILE', 'FOR', 'CONTINUE', 'BREAK', 'RETURN'
)
# All the tokens recognized by the lexer
tokens = keywords + (
    # Identifiers
    'ID',

    # constants
    'INT_CONST_DEC',
    'FLOAT_CONST',
    'CHAR_CONST',

    # String literals
    'STRING_LITERAL',

    # Operators
    'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'MOD',
    'OR', 'AND', 'NOT', 'XOR', 'LSHIFT', 'RSHIFT',
    'LOR', 'LAND', 'LNOT',
    'LT', 'LE', 'GT', 'GE', 'EQ', 'NE',

    # Assignment
    'EQUALS', 'TIMESEQUAL', 'DIVEQUAL', 'MODEQUAL',
    'PLUSEQUAL', 'MINUSEQUAL',
    'LSHIFTEQUAL', 'RSHIFTEQUAL', 'ANDEQUAL', 'XOREQUAL',
    'OREQUAL',

    # Increment/decrement
    'PLUSPLUS', 'MINUSMINUS',

```

```
    # Structure dereference (->)
    'ARROW',

    # Conditional operator (?)
    'CONDOP',

    # Delimeters
    'LPAREN', 'RPAREN',         # ( )
    'LBRACKET', 'RBRACKET',     # [ ]
    'LBRACE', 'RBRACE',         # { }
    'COMMA', 'PERIOD',          # . ,
    'SEMI', 'COLON',            # ; :
)
```

Important Regular Expression Definitions

```
identifier = r'[a-zA-Z_$][0-9a-zA-Z_$]*'

decimal_constant = '(0+)|[1-9][0-9]*'

cconst_char = r"""[^'\\\n]"""
char_const = "'"+cconst_char+"'"
multicharacter_constant = "'"+cconst_char+"{2,4}'"
unmatched_quote = "('"+cconst_char+"*\\n)|('"+cconst_char+"*$)"

# string literals (K&R2: A.2.6)
string_char = r"""[^"\\\n]"""
string_literal = '"'+string_char+'*"'

# floating constants (K&R2: A.2.5.3)
exponent_part = r"""([eE][-+]?[0-9]+)"""
fractional_constant = r"""([0-9]*\.[0-9]+)|([0-9]+\.)"""
floating_constant = '((((('+fractional_constant+')'+exponent_part+'?)|([0-9]+'+exponent_part+'))[FfLl]?)'

# empty space
t_ignore = ' \t'

# Comment
t_ignore_SING_COMMENT = r'//.*?\n'
t_ignore_MULT_COMMENT = r'/\*(\*(?!\/)|[^*])*\*\/'
```

We used the **PLY** (Python Lex Yacc) to help better recognize regular expressions and BNF grammars to make our life easier.

We only need to import the "lexer" from the PLY module using `from ply import lex`

And build it with `lex.build`, passing specific module into the build function.

The lexer utilizes Python's object reflection (introspection) so it needs the current building context to extract regular expressions and map them to corresponding tokens.

It mainly recognize variables defined as `t_TOKEN_NAME`, the content of the variable corresponds to the regular expression, and the `TOKEN_NAME` part would be the token this RE recognizes for.

§1.3 Specific Optimizations

Token Removal

By using `t_ignore` pattern or not returning the recognized token, we discard unwanted information provided by the program source code for
H3 readability of us **humans** (but not for the **parser**)

H4

- White Space
- New Line Characters
- Comments
  - **Single-lined comment**
  - **Multi-lined comment**

```
1   # empty space
2   t_ignore = ' \t'
```

Line Number Memory

To help the user pinpoint what's gone wrong the tokenization process, `PLY` "remembers" every token's location (in terms of line number and token column), which will even be used in the later syntax analysis process.

H4 Specifically, we used `r'\n+'` to indicate newline(s)

- **It's important to notice that multi-line comment/single-line comment might also consume the newline character**
- **Also note that `r'\n+'` regular expression might contain multiple newline characters**

So we should use patterns like `t.lexer.lineno += t.value.count("\n")` to update the new line count accordingly.

```python
# Define a rule so we can track line numbers
def update_lineno(self, t):
    t.lexer.lineno += t.value.count("\n")

# Comment
def t_SINGLE_LINE_COMMENT(self, t):
    r'//.*?\n'
    self.update_lineno(t)

def t_MULTI_LINE_COMMENT(self, t):
    r'/\*(\*(?!\/)|[^*])*\*\/'
    self.update_lineno(t)

def t_NEWLINE(self, t):
    r'\n+'
    self.update_lineno(t)
```

Order of Regular Expression

We carefully optimized the **order** in which each regular expression is provided to the lexer, to make sure some overlapping definitions don't get mixed up, for example:

H4

- `//` for line comment comes before `/` Operator
- `/*` for multi-line comment comes before `/` Operator
- `ID` for identifier comes after other constants that may include characters and numbers (char literal, string literal, int/float constant values)

H2

Chapter 2 - Syntax Analysis

§2.1 Grammar Syntax for Nano C language

Firstly, let's take a comprehensive look at our grammar:

1. We *don't* implement preprocessing like **macros** and **includes**

H3

   - You cannot `#define` or `#include`

2. We *don't* implement multi-file compilation, as a direct cause of the first rule
   - You cannot `python nanoirgen.py a.c b.c c.c -o a.out`
3. Thus we want a program to define the whole program (a single C source file)
4. This program should contain some **global variable definitions**
5. This program should also contain some **function definitions**

6. We *don't* support **external linkage** variable definitions due to rule number one
7. We *don't* support function/global variable **declaration**s since it won't be of much use in this setup.
8. Every valid statement should be
    a. A **block of statements**

        Wrapped within two paired curly brackets, namely `{`, `}`

        This block can **recursively contain other statements**

        You can happily and inconsequently do `{{{{{;}}}}}`

    b. Some **control flows**
        a. `for (int i = 0; i < 100 ; i++ ) {;}` for loops
        b. `while (1) {;}` while loops
        c. `do {;} while(1);` do-while loops
        d. `if(1) {;} else if(2) {;} else {;}` if-else statements
        e. Note that we forbid empty block `{}`
        f. But **one single (non-blocked) statement will be valid**, for example: `if(1) print(1); else print(2);`

            Actually, the nested `if` `else if` `else` block is implemented by viewing the second `if-else` block as a single statement

            `if (1) {;} else { if (2) {;} else {;} }`

            You can do crazy things as long as you remember you're writing out one single statement

            `while (1) while(0) while(controller) do p = "inside while loop"; while ( condition == "OK" );`
        g. Every control flow has its own block whether it's wrapped within the brackets, or just a **valid single statement** mentioned above

            Since **blocks** are used for scope resolution

    c. Or **ends with** `;`
        a. `return;` statement (you can return nothing)
        b. declaration statement to be talked about below
        c. `;` is also a valid statement
9. The variable definition takes traditional C form, with corresponding scope resolution

a. `int a;` will define the variable `a` as uninitialized memory space

b. `int a = 1;` will define the variable `a`, and initialize it to `1`

c. `int a = 1, b = 2` will define both `a` and `b` and individually initialize them as specified by the user

   On a grammar level, this is expanded to be a bunch of variable definition and initialization to avoid a deep traversal into the actual AST

   This optimization would be later illustrated in better detail in the next section

d. Note that type node should only be declared once in one declaration statement or declaration list, meaning `int * a, * b` is illegal, while `int ********** a, b, c=1, d` is OK

10. Every **block** of statements indicates a new name scope, whose resolution will be later talked about in the Code Generation section

    This optimization would be later illustrated in better detail in the next section

11. An expression falls in the following group:

    a. **Binary Operations**: left hand side and right hand size operated by the operator

    b. **Unary Operations**: a single operator acted upon some other expression

       We use **assignment operation** to simplify the use of `++`, `--` unary operators

       These're simply reconstructed to `a = a + 1` (with assignment operation returning the assigned values)

       This optimization would be later illustrated in better detail in the next section

    c. **Ternary Operation**(s): currently only supporting `?:` as ternary operators

    d. **Assignment Expression**: the assignment of some `ID` or a dereferenced valid pointer `*(a+3)`, typically referred to as *left values*

       Specific operators and their corresponding operations/precedence are defined in Lexical Analysis sections

       Note that we define the grammar from a **low to high** precedence order to account for their ambiguous order and associativity if not carefully specified.

- Note that **compound assignment** operations can be easily comprehended as a corresponding expression with a regular assignment operation: `<<=` `+=` `-=`, etc.

  This optimization would be later illustrated in better detail in the next section

e. **Function Calls** in the form of `ID(expression list)`

   You can also specify **no parameter**

f. **Array Subscription** in the form of `ID[expression]`

   In array definition (not an expression), you can also specify a set of empty bracket pairs, indicating a so-called "multidimensional array"

   Although they're expected to be allocated as a continuous blob in the runtime memory

12. Expressions can be grouped by `(` and `)` to indicate their correspondence

    As long as the grammar is unambiguous in this section, the programmer should be able to define arbitrarily complex expressions

13. Expressions should also able to be downgraded to some specific stuff:

    a. `ID` **for identifiers, this can be variables or functions names**
    b. **integer constant for some literal integers**
    c. **float constant for some floating points**
    d. **character constant wrapped with** `''`
    e. **string constant wrapped with** `""`

14. We restrict that only **Unary Operation** can be used at the left side of an assignment. Though this restriction is far from achieving a true **valid left value** check, it would surely make the process of type checking less painful

    This optimization would be later illustrated in better detail in the next section

§2.2 BNF Definition for the Nano C Language

According to the above grammar specification, we define the following BNF grammars to recognize the specific token patterns

H3

1. **We used the careful ordering of grammar production to support complex precedence**
2. **We inherited the line number and column identification from tokens to give some comprehensive error message**
3. **BNF are written in a nest-able form, allowing for easy recognition of recursively nested grammar**

```
"""
Grammars written here is for you (the human reading this) to have a comprehensive understanding of the NanoC language. They're written in BNF for better copy-pasting compability for the parser, and it should be just the same as the ones used in the parser to build the AST.

Productions used in the parser:

        program             : program function
                            | program declaration
                            |
        function            : type id LPAREN param_list RPAREN curl_block
        param_list          : type id comma_paramters
                            | VOID
                            |
        comma_params        : comma_params COMMA type id
                            |
        block               : block curl_block
                            | block statement
                            |
        id                  : ID
        type                : INT
                            | VOID
                            | LONG
                            | FLOAT
                            | DOUBLE
                            | CHAR
                            | type TIMES
        statement           : expression SEMI
                            | declaration
                            | IF LPAREN expression RPAREN ctrl_block ELSE ctrl_block
                            | IF LPAREN expression RPAREN ctrl_block
                            | FOR LPAREN for_init RPAREN ctrl_block
                            | WHILE LPAREN expression RPAREN ctrl_block
                            | DO ctrl_block WHILE LPAREN expression RPAREN SEMI
                            | RETURN empty_or_exp SEMI
                            | BREAK SEMI
                            | CONTINUE SEMI
                            | SEMI
        for_init            : empty_or_exp SEMI empty_or_exp SEMI empty_or_exp
                            | declaration empty_or_exp SEMI empty_or_exp
        exp_list            : expression comma_exps
                            |
        comma_exps          : comma_exps COMMA expression
                            |
```

```
43          empty_or_exp        : expression
44                              |
45          ctrl_block          : curl_block
46                              | statement
47          curl_block          : LBRACE block RBRACE
48          declaration         : type dec_list SEMI
49          dec_list            : dec_list COMMA id array_list typeinit
50                              | id array_list typeinit
51          typeinit            : EQUALS expression
52                              |
53          array_list          : array_list LBRACKET INT_CONST_DEC RBRACKET
54                              |
55          expression          : assignment
56          assignment          : conditional
57                              | unary EQUALS expression
58          conditional         : logical_or
59                              | logical_or CONDOP expression COLON
    conditional
60          logical_or          : logical_and
61                              | logical_or LOR logical_and
62          logical_and         : bitwise_or
63                              | logical_and LAND bitwise_or
64          bitwise_or          : bitwise_xor
65                              | bitwise_or OR bitwise_xor
66          bitwise_xor         : bitwise_and
67                              | bitwise_xor XOR bitwise_and
68          bitwise_and         : equality
69                              | bitwise_and AND equality
70          equality            : relational
71                              | equality (EQ|NE) relational
72          relational          : shiftable
73                              | relational (LT|GT|LE|GE) shiftable
74          shiftable           : additive
75                              | shiftable (LSHIFT|RSHIFT) additive
76          additive            : multiplicative
77                              | additive (PLUS|MINUS) multiplicative
78          multiplicative      : unary
79                              | multiplicative (TIMES|DEVIDE|MOD) unary
80          unary               : postfix
81                              |
    (PLUS|MINUS|NOT|LNOT|TIMES|AND|PLUSPLUS|MINUSMINUS) unary
82                              | LPAREN type RPAREN unary
83          postfix             : primary
84                              | id LPAREN exp_list RPAREN
85                              | postfix LBRACKET expression RBRACKET
86                              | id PLUSPLUS
87                              | id MINUSMINUS
88          primary             : INT_CONST_DEC
89                              | FLOAT_CONST
```

```
                              | CHAR_CONST
                              | STRING_LITERAL
                              | id
                              | LPAREN expression RPAREN
    """
```

As mentioned above, we used the `PLY` package for token/syntax recognition

With a well-defined grammar, the next step is to parse corresponding production into a well-organized **Abstract Syntax Tree**, which will be illustrated in more detail in the following section

Being an **abstract** syntax tree, a large portion of the parsing is simply to define which production results in which kind of tree node, and how those nodes are to be organized correctly

```
##############################################################
#                  Arithmetic/Logical Operations             #
##############################################################

def p_binary_operators(self, p):
    '''
    logical_or          : logical_or LOR logical_and
    logical_and         : logical_and LAND bitwise_or
    bitwise_or          : bitwise_or OR bitwise_xor
    bitwise_xor         : bitwise_xor XOR bitwise_and
    bitwise_and         : bitwise_and AND equality
    equality            : equality EQ relational
                        | equality NE relational
    relational          : relational LT shiftable
                        | relational GT shiftable
                        | relational GE shiftable
                        | relational LE shiftable
    shiftable           : shiftable LSHIFT additive
                        | shiftable RSHIFT additive
    additive            : additive PLUS multiplicative
                        | additive MINUS multiplicative
    multiplicative      : multiplicative TIMES unary
                        | multiplicative DIVIDE unary
                        | multiplicative MOD unary
    '''
    p[0] = BinaryNode(p[2], p[1], p[3])

```

Take this binary operation parsing as an example:

- The precedence are defined within the grammar itself

- The actual operator takes similar form (a string of one/two special character indicating a specific operation)
- Nested parenthesis pairs and the actual positioning (where these operations should appear) of those operations are well defined into the **tree-structure** (*the parse tree*)
- Boiling down to the AST building, all we have to do is construct the AST node and store it for later (upper in the parse tree)

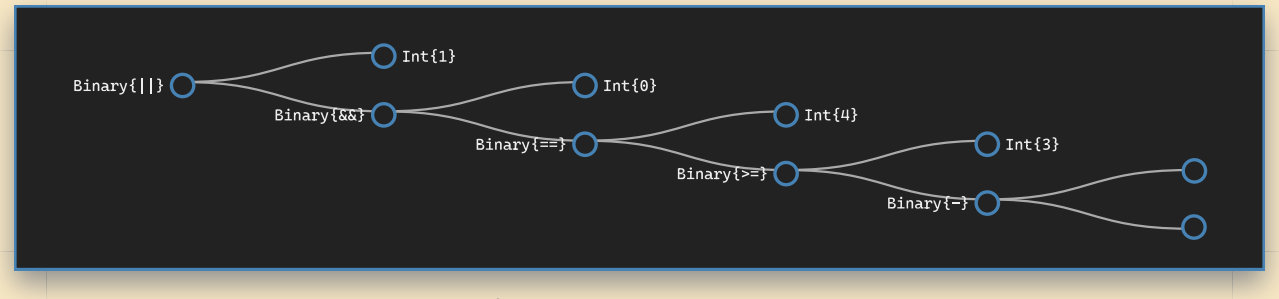
```
1    p[0] = BinaryNode(p[2], p[1], p[3])
```

Thanks to the parse tree, `p[1]`, `p[2]`, `p[3]` are already valid elements of the AST (constructed in the parsing process before this one and saved to the **parsed object** `p[0]`)

For example, some consecutive parsing of binary operations (`LPAREN` and `RPAREN` taken care of, omitted here) might be flattened like this:

```
1    p[0] = BinaryNode(BinaryNode(BinaryNode(p[2], p[1], BinaryNode(p[2], p[1],
     p[3])), p[1], p[3]), p[1], BinaryNode(BinaryNode(p[2], p[1], p[3]), p[1],
     BinaryNode(p[2], p[1], p[3])))
```

A nice visualization of this might be like:



§2.3 Actual Implementation

H3

```
1   ##################################################################
2   #                    Syntax Error Rules                          #
3   ##################################################################
4
5   def p_error(self, p):
6       # with a syntax error, the token should contain corresponding location
7       print(colored("Error: ", "red")+"Syntax error when parsing "+str(p))
8
9   ##################################################################
10  #                    External Functions                          #
11  ##################################################################
```

```
12
13  def parse(self, input, **kwargs) -> Node:
14      try:
15          return self.parser.parse(input, **kwargs)
16      except Exception as e:
17          traceback.print_exc()
18          print(colored("Error: ", "red")+f"{e}")
19
20  def build(self, **kwargs):
21      self.parser = yacc.yacc(module=self, **kwargs)
22
23  tokens = NanoLexer.tokens
```

Similar to the lexer, `yacc` needs to be specifically built for using.

We pass in `module=self` to make YACC track production definition in current object scope using object reflection.

```
1   tokens = NanoLexer.tokens
```

is needed to define valid tokens to recognize for (usually marked UPPER CASE CHARACTER).

To make the parsing process more visible and viable, we defined the main function of the `nanoyacc.py` program as this:

```
1   if __name__ == '__main__':
2       import argparse
3       parser = argparse.ArgumentParser()
4       parser.add_argument("-input", default="samples/fx.c", type=str)
5       parser.add_argument("-tree", type=str)
6       parser.add_argument("-url", default="http://neon-
    cubes.xyz:8000/src/tree.json", type=str)
7       args = parser.parse_args()
8
9       with open(args.input, 'r', encoding='utf-8') as f:
10          content = f.read()
11          lexer = NanoLexer()
12          lexer.build()
13          parser = NanoParser()
14          parser.build()
15          root = parser.parse(content, debug=0)
16          print(colored(f"Abstract Syntax Tree:", "yellow", attrs=["bold"]))
17          print(root)
18
19          tree = traverse(root)
```

```
20          print(colored("Structrued Tree: ", 'yellow', attrs=['bold']))
21          print(tree)
22          addinfo(tree, args.input)
23          payload = json.dumps(tree)
24
25          if args.tree:
26              with open(args.tree, 'w') as f:
27                  f.write(payload)
28              print(colored(f"Saved Structrued Tree to {args.tree}",
    'yellow', attrs=['bold']))
29
30          r = requests.post(url=args.url, data=payload)
31          print(colored(f"POST response: {r}", "yellow", attrs=["bold"]))
```

1. `argparse` exists so we have a friendly command-line interface for the nanoparser
2. `lexer` is built and automatically stored in a safe position (a global variable `ply.lex.lexer`)
3. `parser` is also built and automatically stored in a safe position `ply.yacc.parser`
4. `root` is returned by the parser as the root of the AST, to be used by the intermediate representation generator
5. `tree` is the simplified version (ready to be displayed with good visual look)
6. We set up a server to receive the AST JSON object for mobility and compatibility for the visualization of the AST nodes
7. `colored` from `termcolor` is applied extensively to give a visually pleasant command-line output
8. `tree.json` will also be saved as a file for inspection

§2.4 Specific Optimizations

Flattening

We flatten declaration operations to make the actual IR generation a little bit less painful.

H3

H4

- Given a list of declarations, initialized or not, we flatten all declarations into individual statements, so the compiler backend can uniformly take care of them

  `int a, b, c;` would initially produce a list of declaration as `[DecNode, DecNode, DecNode]`

  When the outer block is encountered, those expressions are flattened out as

  `DecNode; DecNode; DecNode;`

- This is implemented with the help of `BlockNode`

```
def p_block_stmt(self, p):
    '''
    block               : block curl_block
                        | block statement
    '''
    if p[1] is None:
        p[1] = BlockNode()
    if isinstance(p[2], list):
        for dec in p[2]:
            p[1].append(dec)
    else:
        p[1].append(p[2])
    p[0] = p[1]
```

The above implementation also illustrates other implementation of flattening listed grammar

```
'''
    block               : block curl_block
                        | block statement
                        |
'''
```

is **left recursive**, thus the first block might just be `None`, and later all other blocks should be appended accordingly to whether we've already constructed a list from the first element.

Similar optimization occurs when we're parsing **expression list** of function call/**parameter list** of function definition

- **Expression List** in function call:

```
def p_exp_list(self, p):
    '''
    exp_list            : expression comma_exps
    '''
    if p[2] is None:
        p[2] = []
    p[2] = [p[1]] + p[2]
    p[0] = p[2]

def p_comma_exp_list(self, p):
    '''
```

```
    comma_exps          : comma_exps COMMA expression
    '''
    if p[1] is None:
        p[1] = []
    p[1].append(p[3])
    p[0] = p[1]
```

- **Parameter List** in function definition

```
################################################################
#                     Function Definition                      #
################################################################

def p_func_def(self, p):
    '''
    function            : type id LPAREN param_list RPAREN curl_block
    '''
    p[0] = FuncNode(p[1], p[2], p[4], p[6])

def p_params(self, p):
    '''
    param_list          : type id comma_params
    '''
    param = ParamNode(p[1], p[2])
    if p[3] is None:
        p[3] = []
    p[3] = [param] + p[3]
    p[0] = p[3]

def p_comma_params(self, p):
    '''
    comma_params        : comma_params COMMA type id
    '''
    param = ParamNode(p[3], p[4])
    if p[1] is None:
        p[1] = []
    p[1].append(param)
    p[0] = p[1]
```

Preparations for Scope Resolution

Every **block** of statements indicates a new name scope, whose resolution will be later talked about in the Code Generation section

Note that `BlockNode` is ***ABSTRACT***, meaning with the total removal of it, the compiler should still be able to work properly. The purpose of the aggregated node is to indicate nested scope creation

H4

- Thus, when parsing curly brackets pairs, we explicitly generate a `BlockNode` to mark the creation of a new scope
- We also took care of `if-else-stmt` and `for-do-while-loop` statements, whose statement body is valid even when they've only got one individual statement

  Those individual statement are also wrapped with an abstract `BlockNode`

Abstraction of Complex Syntax

For the compiler backend, `a += 1` is the same as `a = a + 1` and in our implementation, `a++` or `++a`

H4 So, when implementing similar relatively complex grammar, the parser automatically translates it to the format that the compiler recognizes

Thus a simple `AssNode` (meaning, `Assignment Node`) can take care of all of these and free the IR from recognizing complex assignment and syntax sugar

This is implemented as:

```
def p_assignment(self, p):
    '''
    assignment          : unary EQUALS expression
                        | unary TIMESEQUAL expression
                        | unary DIVEQUAL expression
                        | unary MODEQUAL expression
                        | unary PLUSEQUAL expression
                        | unary MINUSEQUAL expression
                        | unary LSHIFTEQUAL expression
                        | unary RSHIFTEQUAL expression
                        | unary ANDEQUAL expression
                        | unary XOREQUAL expression
                        | unary OREQUAL expression
    unary               : PLUSPLUS unary
                        | MINUSMINUS unary
    postfix             : postfix PLUSPLUS
                        | postfix MINUSMINUS

    '''
    if len(p) == 4:
        if len(p[2]) == 1:  # true assignment
            p[0] = AssNode(p[1], p[3])
        else:
            if len(p[2]) == 3:
                op = p[2][:2]
```

```
26              else:
27                  op = p[2][:1]
28              p[0] = AssNode(p[1], BinaryNode(op, p[1], p[3]))
29
30      else:
31          if p[1] == "++":
32              p[0] = AssNode(p[2], BinaryNode('+', p[2], IntNode(1)))
33          elif p[1] == "--":
34              p[0] = AssNode(p[2], BinaryNode('-', p[2], IntNode(1)))
35          elif p[2] == "++":
36              p[0] = AssNode(p[1], BinaryNode('+', p[1], IntNode(1)))
37          elif p[2] == "--":
38              p[0] = AssNode(p[1], BinaryNode('-', p[1], IntNode(1)))
39          p[0].exp.update_pos(p.lineno(1),
    self._find_tok_column(p.lexpos(1)))
40      p[0].update_pos(p.lineno(1), self._find_tok_column(p.lexpos(1)))
```

Trick for Solving the Dangling Else Problem

The dangling else problem in grammar syntax parsing appears when no restriction is applied on the end of if statement

with a grammar definition like:

H4

```
1     '''
2        statment                  : IF LPAREN expression RPAREN ctrl_block
    ELSE ctrl_block
3                                  | IF LPAREN expression RPAREN ctrl_block
4     '''
```

and a program like this:

```
1    if (1) if (2) ; else ;
```

The else can be associated with the first `if` like with brackets:

```
1    if (1) { if (2) ; } else ;
```

Or the second `if`:

```
1    if (1) { if (2) ; else ; }
```

without violating the grammar definition!

This is commonly known as *the dangling else* problem:

- **Whether and else statement should be paired with the outer most if statement or the inner most unpaired one?**

There's an ugly solution to this:

- Define the grammar so that **only unmatched statement** can be associated with the if statement

```
'''
statement               : matched-stmt | unmatched-stmt
matched-stmt            : if  ( exp ) matched-stmt else matched-stmt
                        | other
unmatched-stmt          : if  ( exp ) statement
                        | if  ( exp ) matched-stmt else unmatched-stmt
exp                     : 0 | 1
'''
```

> **This works by permitting** only a matched-statement to come before an else in an if-statement**, thus forcing all else-parts to be matched as soon as possible.**

  But this solution would require some painful modification of the parser code

- And another solution would be marking the end of the if-else block with some special token like `end-if`

```
if (1) ; else ; endif
```

  Effectively acting as a pair (pairs) of curly brackets

  solving the ambiguity by make the programmer do more

- The **most popular** solution is to actually leave the grammar be

  Leave everything be and let the **LALR** parser **prefer shift over reduce** when there's a shift-reduce conflict

  And this is also the solution that we used

```
...
...
state 196
```

```
4
5       (17) statement -> IF LPAREN expression RPAREN ctrl_block .
6       (18) statement -> IF LPAREN expression RPAREN ctrl_block . ELSE
    ctrl_block
7
8     ! shift/reduce conflict for ELSE resolved as shift
9       ELSE            shift and go to state 202
10
11    ! ELSE            [ reduce using rule 17 (statement -> IF LPAREN
    expression RPAREN ctrl_block .) ]
12    ...
13    ...
14  WARNING:
15  WARNING: Conflicts:
16  WARNING:
17  WARNING: shift/reduce conflict for ELSE in state 196 resolved as shift
```

> If the parser is produced by an SLR, LR(1) or LALR LR parser generator, the programmer will often rely on the generated parser feature of preferring shift over reduce whenever there is a conflict.[2] Alternatively, the grammar can be rewritten to remove the conflict, at the expense of an increase in grammar size

H2 # Chapter 3 - Abstract Syntax Tree

§3.1 Node Design

We adopted the OOP design pattern to make life easier for `pylance`, the type checking utility and auto-complete functionality of the developer's IDE

H3 `Node` is defined to be the base class of every node and this type can be used to distinguish an actual `NanoAST` node from some string/number literals and original python literals like `list`s or `dict`s.

Specifically, We have

- A base class `Node` for all AST nodes
- A base class `EmptyStmtNode` sub-classing `Node`, acting as base class of all primitive statements, including
  - `IfStmtNode`
  - `LoopNode` for all looping including
    - `FOR` loop

        - `WHILE` loop
        - `DO-WHILE` loop
    - `DecNode`
    - `RetNode`
    - `BlockNode` for aggregating all other statements
    - `BreakNode`
    - `ContinueNode`

    Note that an `Expression` followed by a `SEMI` (semicolon) is also a valid statement, but for **abstraction**, we extract that to be able to be directly embedded in `BlockNode`

- A base class `EmptyExpNode` sub-classing `Node`, acting as base class of all primitive expressions, including
    - `CallNode` for function call
    - `UnaryNode` for unary operations
    - `BinaryNode` for binary operations
    - `TernaryNode` for ternary operations
    - `AssNode` for assignment expressions
    - `ArrSubNode` for subscription of an array/pointer

§3.2 Tree Visualization and Interaction

`IMPORTANT: To better grasp the ability of our Abstract Syntax Tree visualizer, go to the GitHub of this repo to see for yourself.`

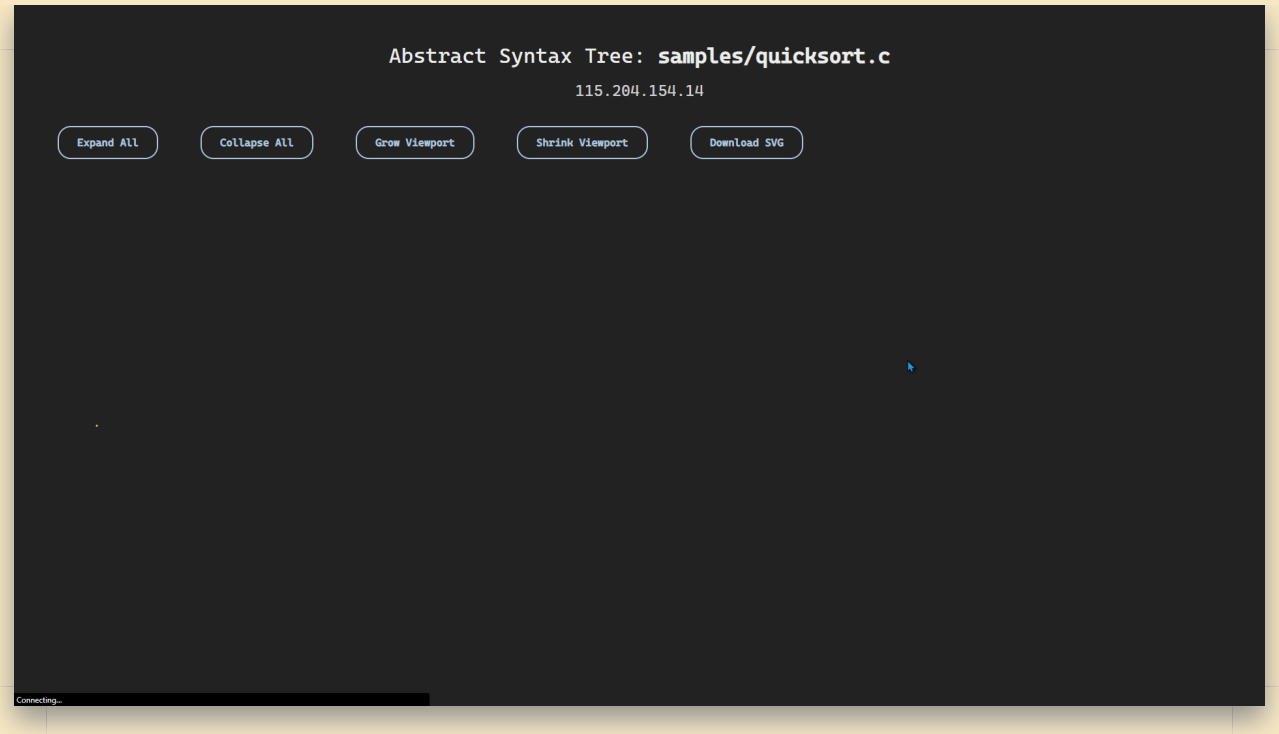
`IMPORTANT: The visualizer is hosted at: my server`

H3

On the hosted server mentioned above, we have a nice little html (including some `javascript`) to display the abstract syntax tree in an understandable and **interactive** manner.

If you're able to see the moving GIF, I don't think I need to explain more.

If you're seeing a static image, please read the `IMPORTANT` message a few lines above.



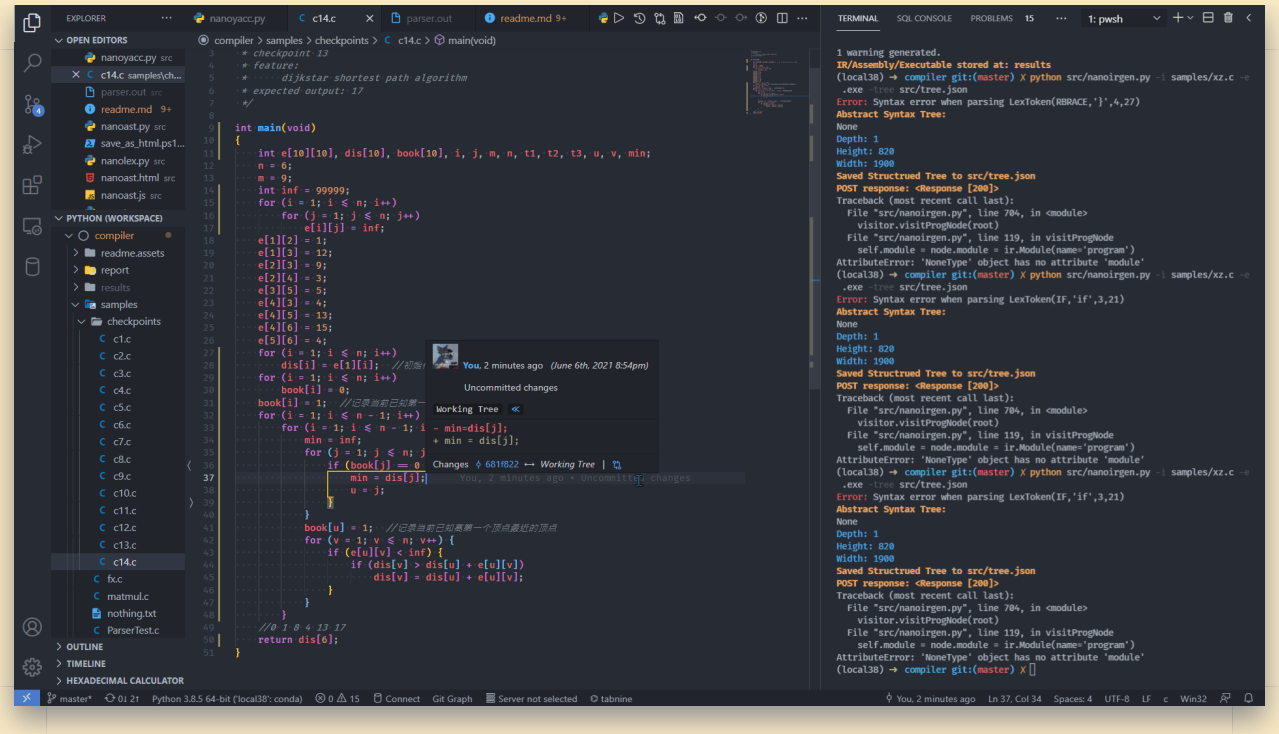
The server is also designed to accept incoming compilation, so you can basically update what to display to everyone by uploading the `tree.json` traversed to the server.

Implementation:

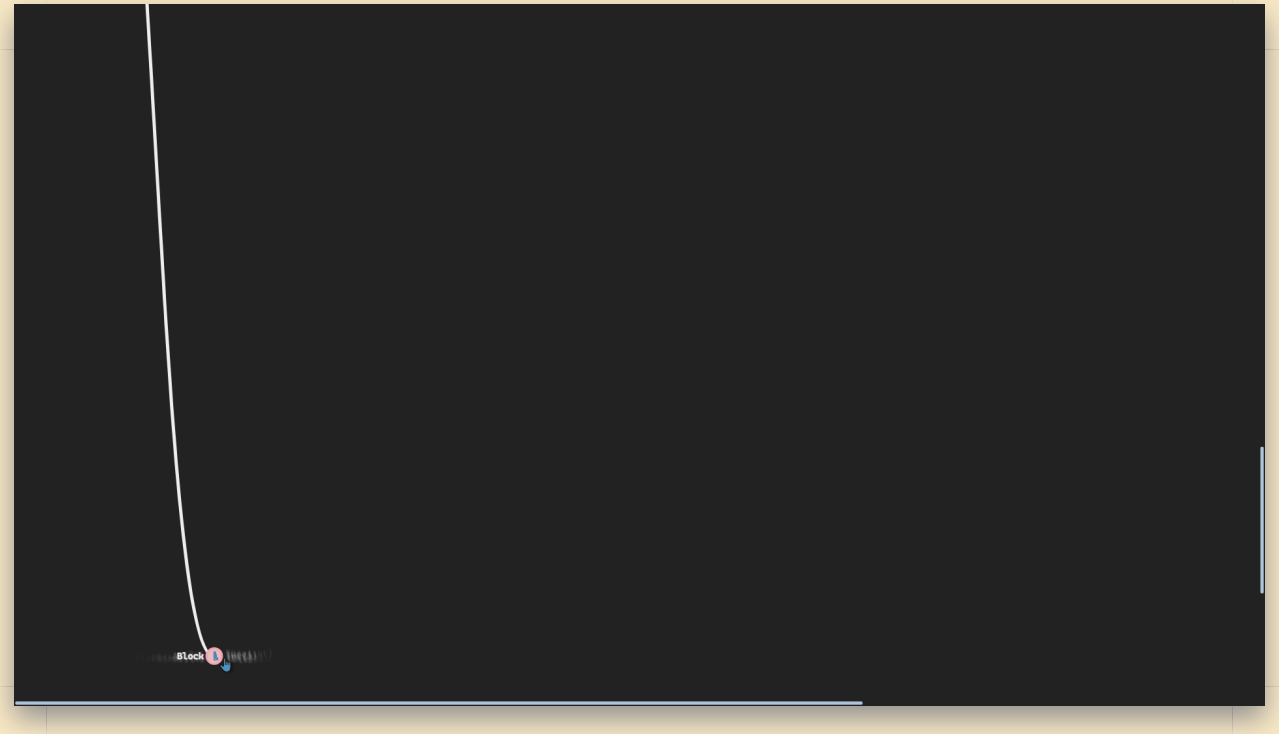
```
tree = traverse(root)
addinfo(tree, args.input)
# Print Struct Tree (data sent to server)
print(colored("Structrued Tree: ", 'yellow', attrs=['bold']))
print(tree)
payload = json.dumps(tree)

if args.tree:
    with open(args.tree, 'w') as f:
        f.write(payload)
        print(colored(f"Saved Structrued Tree to {args.tree}", 'yellow', attrs=['bold']))

r = requests.post(url=args.url, data=payload)
print(colored(f"POST response: {r}", "yellow", attrs=["bold"]))
```



You're also able to download a fully viewable tree from the server directly (or rather, from the `nanoast.html` you're visiting)



You'll see similar **SVG** embedded in our report later in the Code Generation section.

§3.3 Optimization Considerations

Better Debugging Interface

When designing the base class of all nodes, we considered the need to pretty print all things from command-line, thus a `indentLevel` is added and all `__str__` methods of nodes are designed to recursively do a depth first search on the abstract syntax tree to produce some human readable

H3

H4

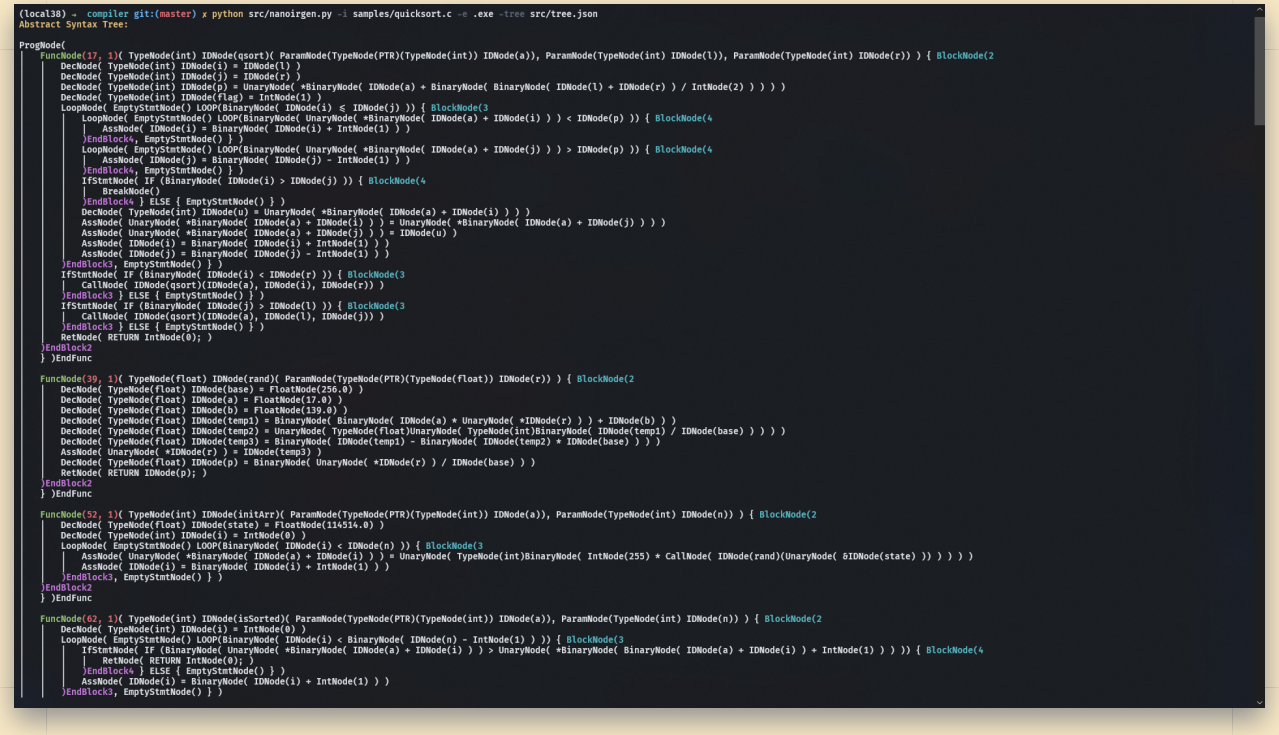
parsing results (with formats!)

```python
class Node(object):
    # A simple Abstract Syntax Tree node
    TABSTR = '|   '

    def __init__(self):
        self._indentLevel = 0
        self._lineno = self._colno = 0

    def update_pos(self, line: int, col: int):
        self._lineno = line
        self._colno = col

    def accept(self, visitor: NanoVisitor):
        pass
```



Also, with error message in mind, we printed detailed line numbering and column to help the compiler user location where things might have gone wrong as early as possible.

```
int main()
{
    { ret
    }
}
```

```
(local38) ➜  compiler git:(master) ✗ python src/nanoirgen.py -i samples/xz.c -e .exe -tree src/tree.json
Error: Syntax error when parsing LexToken(RBRACE,'}',4,27)
Abstract Syntax Tree:
None
Depth: 1
Height: 820
Width: 1900
Saved Structrued Tree to src/tree.json
```

```
1    int main()
2    {
3        if (if)
4    }
```

```
(local38) ➜  compiler git:(master) ✗ python src/nanoirgen.py -i samples/xz.c -e .exe -tree src/tree.json
Error: Syntax error when parsing LexToken(IF,'if',3,21)
Abstract Syntax Tree:
None
Depth: 1
Height: 820
Width: 1900
Saved Structrued Tree to src/tree.json
```

Correct warning location during later phases:

This is the source for a simple quicksort algorithm

```
1   /**
2    * feature:
3    *      integer type & float type & void type
4    *      pointer type & array type
5    *      & and * operator
6    *      type casting:
7    *          xd array -> pointer
8    *          int <-> float
9    *      calculations:
10   *          *(pointer + integer)
11   *      quicksort
12   *      random integer generation
13   * expected output: 1
14   */
15
16  int qsort(int *a, int l, int r)
17  {
18      int i = l;
19      int j = r;
20      int p = *(a + ((l + r) / 2));
21      int flag = 1;
22      while (i ≤ j) {
23          while (*(a + i) < p) i++;
24          while (*(a + j) > p) j--;
25          if (i > j) break;
26          int u = *(a + i);
27          *(a + i) = *(a + j);
```

```
28          *(a + j) = u;
29          i++;
30          j--;
31      }
32      if (i < r) qsort(a, i, r);
33      if (j > l) qsort(a, l, j);
34      return 0;
35  }
36
37  // random floating point number distributed uniformly in [0,1]
38  float rand(float *r)
39  {
40      float base = 256.0;
41      float a = 17.0;
42      float b = 139.0;
43      float temp1 = a * (*r) + b;
44      float temp2 = (float)(int)(temp1 / base);
45      float temp3 = temp1 - temp2 * base;
46      *r = temp3;
47      float p = *r / base;
48      return p;
49  }
50
51  int initArr(int *a, int n)
52  {
53      float state = 114514.0;
54      int i = 0;
55      while (i < n) {
56          *(a + i) = (int)(255 * rand(&state));
57          i += 1;
58      }
59  }
60
61  int isSorted(int *a, int n)
62  {
63      int i = 0;
64      while (i < n - 1) {
65          if ((*(a + i)) > (*(a + i + 1)))
66              return 0;
67          i += 1;
68      }
69      return 1;
70  }
71
72  int main()
73  {
74      int n = 100;
75      int arr[100];
76      int *a = (int *)arr;
```

```
77      initArr(a, n);
78      qsort(a, 0, n - 1);
79      return isSorted(a, n);
80  }
```

With the help of carefully designed line number/column memory and the help of a tracking parser, it's easy for the user to locate the erroneous code.

```
ImplicitTypeCastingWarning: implicit type casting from i32 to i32* at position (line 21, col 15)
ImplicitTypeCastingWarning: implicit type casting from i32 to i1 at position (line 23, col 12)
ImplicitTypeCastingWarning: implicit type casting from i32 to i1 at position (line 23, col 12)
ImplicitTypeCastingWarning: implicit type casting from i32 to i32* at position (line 24, col 18)
ImplicitTypeCastingWarning: implicit type casting from i32 to i1 at position (line 24, col 16)
ImplicitTypeCastingWarning: implicit type casting from i32 to i1 at position (line 24, col 16)
ImplicitTypeCastingWarning: implicit type casting from i32 to i32* at position (line 25, col 18)
ImplicitTypeCastingWarning: implicit type casting from i32 to i1 at position (line 25, col 16)
ImplicitTypeCastingWarning: implicit type casting from i32 to i1 at position (line 25, col 16)
ImplicitTypeCastingWarning: implicit type casting from i32 to i1 at position (line 26, col 13)
ImplicitTypeCastingWarning: implicit type casting from i32 to i1 at position (line 26, col 13)
ImplicitTypeCastingWarning: implicit type casting from i32 to i32* at position (line 27, col 19)
ImplicitTypeCastingWarning: implicit type casting from i32 to i32* at position (line 28, col 11)
ImplicitTypeCastingWarning: implicit type casting from i32 to i32* at position (line 28, col 22)
ImplicitTypeCastingWarning: implicit type casting from i32 to i32* at position (line 29, col 11)
ImplicitTypeCastingWarning: implicit type casting from i32 to i1 at position (line 33, col 9)
ImplicitTypeCastingWarning: implicit type casting from i32 to i1 at position (line 33, col 9)
ImplicitTypeCastingWarning: implicit type casting from i32 to i1 at position (line 34, col 9)
ImplicitTypeCastingWarning: implicit type casting from i32 to i1 at position (line 34, col 9)
ImplicitTypeCastingWarning: implicit type casting from i32 to i1 at position (line 56, col 12)
ImplicitTypeCastingWarning: implicit type casting from i32 to i1 at position (line 56, col 12)
ImplicitTypeCastingWarning: implicit type casting from i32 to i32* at position (line 57, col 11)
ImplicitTypeCastingWarning: implicit type casting from i32 to float at position (line 57, col 26)
IFuncExitWarning: function initArr does not has explicit exit at position (line 52, col 1)
ImplicitTypeCastingWarning: implicit type casting from i32 to i1 at position (line 65, col 12)
ImplicitTypeCastingWarning: implicit type casting from i32 to i1 at position (line 65, col 12)
ImplicitTypeCastingWarning: implicit type casting from i32 to i32* at position (line 66, col 16)
ImplicitTypeCastingWarning: implicit type casting from i32 to i32* at position (line 66, col 29)
ImplicitTypeCastingWarning: implicit type casting from i32 to i32* at position (line 66, col 31)
ImplicitTypeCastingWarning: implicit type casting from i32 to i1 at position (line 66, col 14)
ImplicitTypeCastingWarning: implicit type casting from i32 to i1 at position (line 66, col 14)
0 error(s), 31 warning(s)
```

Better Coding

We adopted the OOP design pattern to make life easier for `pylance`, the type checking utility and auto-complete functionality of the developer's IDE

H4 Before, you might need to check whether a node is valid by comparing some raw string:

```
1    if node.name == "StmtNode": pass
```

This design makes the compiler writer get trapped in the pitfall of **typos**.

Now you only need to do

```
1    if isinstance(node, StmtNode): pass
```

or

```
    if type(node) == StmtNode: pass
```

Writing things out explicitly makes the checker's life, and your life much easier by providing richer error messages.

I believe you've all had that afternoon spent digging into your code trying to find which tiny typo crashed your delicate, complex, strong program.

`Node` is defined to be the base class of every node and this type can be used to distinguish an actual `NanoAST` node from some string/number literals and original python literals like `list`s or `dict`s.

H2 # Chapter 4 - Semantic Analysis

§4.1 Name Resolution

H3 During compilation, our compiler associates identifiers such as the name of a variable with an address (memory location), datatype, or actual value. This process is called *binding*. The association lasts through all subsequent executions until a recompilation occurs, which might cause a rebinding. Before binding the names, our compiler must resolve all references to them in the compilation unit. This process is called *name resolution* Cour compiler considers all names to be in the same namespace. So, one declaration or definition in an inner scope can hide another in an outer scope.

Scope Checking

H4
```
        def _get_identifier(self, name):
            for d in self.scope_stack[::-1]:  # reversing the scope_block
                if name in d:
                    return d[name]['ref']
            return None
```

We use a scope stack to store the different symbol tables. We will recursively checking the scope from the last created scope to the oldest scope which is known as most recent variable matching rule.

Binding Reference With Name

H4

```
    def _add_identifier(self, name, reference, type):
            if self.scope_stack == []:
                return None
            if name in self.scope_stack[-1]:
                return None
            self.scope_stack[-1][name] = {'ref': reference, 'typ': type}
            return self.scope_stack[-1][name]
```

As we already have a scope stack, we can set the binding between name and reference by a python `dict` object and add the key-value to the latest scope.

§4.2 Type Checking (L value Checking)

Type checking happens at many places and operations. For binary operation, we need to check that the left type and the right type is compatible. We use a survey to illustrate compatible rules and return type:

H3

```
binCompatDict = {
    #left       right      op       ret_type
    ('i32',     'i32',     '+' ):   'i32',
    ('i32',     'i32',     '-' ):   'i32',
    ('i32',     'i32',     '*' ):   'i32',
    ('i32',     'i32',     '/' ):   'i32',
    ('i32',     'i32',     '%' ):   'i32',
    ('i32',     'i32',     '<<'):   'i32',
    ('i32',     'i32',     '>>'):   'i32',
    ('i32',     'i32',     '≠' ):   'i1',
    ('i32',     'i32',     '==' ):   'i1',
    ('i32',     'i32',     '<' ):   'i1',
    ('i32',     'i32',     '>' ):   'i1',
    ('i32',     'i32',     '≤' ):   'i1',
    ('i32',     'i32',     '≥' ):   'i1',
    ('i32',     'i32',     '||'):   'i1',
    ('i32',     'i32',     '&&'):   'i1',
    ('float',   'float',   '+' ):   'float',
    ('float',   'float',   '-' ):   'float',
    ('float',   'float',   '*' ):   'float',
    ('float',   'float',   '/' ):   'float',
    ('float',   'float',   '%' ):   'float',
    ('float',   'float',   '≠' ):   'i1',
    ('float',   'float',   '==' ):   'i1',
    ('float',   'float',   '<' ):   'i1',
    ('float',   'float',   '>' ):   'i1',
    ('float',   'float',   '≤' ):   'i1',
    ('float',   'float',   '≥' ):   'i1',
    ('float',   'float',   '||' ):   'i1',
```

```
    ('float',   'float',   '&&'):   'i1',
    ('i1',      'i1',      '+' ):   'i32',
    ('i1',      'i1',      '-' ):   'i32',
    ('i1',      'i1',      '*' ):   'i32',
    ('i1',      'i1',      '/' ):   'i32',
    ('i1',      'i1',      '%' ):   'i32',
    ('i1',      'i1',      '≠'):   'i1',
    ('i1',      'i1',      '=='):   'i1',
    ('i1',      'i1',      '<' ):   'i1',
    ('i1',      'i1',      '>' ):   'i1',
    ('i1',      'i1',      '≤'):   'i1',
    ('i1',      'i1',      '≥'):   'i1',
    ('i1',      'i1',      '||'):   'i1',
    ('i1',      'i1',      '&&'):   'i1',
    ('i32',     'i1',      '+' ):   'i32',
    ('i32',     'i1',      '-' ):   'i32',
    ('i32',     'i1',      '*' ):   'i32',
    ('i32',     'i1',      '/' ):   'i32',
    ('i32',     'i1',      '%' ):   'i32',
    ('i32',     'i1',      '≠'):   'i1',
    ('i32',     'i1',      '=='):   'i1',
    ('i32',     'i1',      '<' ):   'i1',
    ('i32',     'i1',      '>' ):   'i1',
    ('i32',     'i1',      '≤'):   'i1',
    ('i32',     'i1',      '≥'):   'i1',
    ('i32',     'i1',      '||'):   'i1',
    ('i32',     'i1',      '&&'):   'i1',
    ('i1',      'i32',     '+' ):   'i32',
    ('i1',      'i32',     '-' ):   'i32',
    ('i1',      'i32',     '*' ):   'i32',
    ('i1',      'i32',     '/' ):   'i32',
    ('i1',      'i32',     '%' ):   'i32',
    ('i1',      'i32',     '≠'):   'i1',
    ('i1',      'i32',     '=='):   'i1',
    ('i1',      'i32',     '<' ):   'i1',
    ('i1',      'i32',     '>' ):   'i1',
    ('i1',      'i32',     '≤'):   'i1',
    ('i1',      'i32',     '≥'):   'i1',
    ('i1',      'i32',     '||'):   'i1',
    ('i1',      'i32',     '&&'):   'i1',
    ('float',   'i32',     '+' ):   'float',
    ('float',   'i32',     '-' ):   'float',
    ('float',   'i32',     '*' ):   'float',
    ('float',   'i32',     '/' ):   'float',
    ('float',   'i32',     '%' ):   'float',
    ('float',   'i32',     '≠'):   'i1',
    ('float',   'i32',     '=='):   'i1',
    ('float',   'i32',     '<' ):   'i1',
    ('float',   'i32',     '>' ):   'i1',
```

```
    ('float',   'i32',     '≤'):   'i1',
    ('float',   'i32',     '≥'):   'i1',
    ('float',   'i32',     '||'):   'i1',
    ('float',   'i32',     '&&'):   'i1',
    ('i32',     'float',   '+' ):   'float',
    ('i32',     'float',   '-' ):   'float',
    ('i32',     'float',   '*' ):   'float',
    ('i32',     'float',   '/' ):   'float',
    ('i32',     'float',   '%' ):   'float',
    ('i32',     'float',   '≠'):   'i1',
    ('i32',     'float',   '=='):   'i1',
    ('i32',     'float',   '<' ):   'i1',
    ('i32',     'float',   '>' ):   'i1',
    ('i32',     'float',   '≤'):   'i1',
    ('i32',     'float',   '≥'):   'i1',
    ('i32',     'float',   '||'):   'i1',
    ('i32',     'float',   '&&'):   'i1',
    ('i1',      'float',   '+' ):   'float',
    ('i1',      'float',   '-' ):   'float',
    ('i1',      'float',   '*' ):   'float',
    ('i1',      'float',   '/' ):   'float',
    ('i1',      'float',   '%' ):   'float',
    ('i1',      'float',   '≠'):   'i1',
    ('i1',      'float',   '=='):   'i1',
    ('i1',      'float',   '<' ):   'i1',
    ('i1',      'float',   '>' ):   'i1',
    ('i1',      'float',   '≤'):   'i1',
    ('i1',      'float',   '≥'):   'i1',
    ('i1',      'float',   '||'):   'i1',
    ('i1',      'float',   '&&'):   'i1',
    ('float',   'i1',      '+' ):   'float',
    ('float',   'i1',      '-' ):   'float',
    ('float',   'i1',      '*' ):   'float',
    ('float',   'i1',      '/' ):   'float',
    ('float',   'i1',      '%' ):   'float',
    ('float',   'i1',      '≠'):   'i1',
    ('float',   'i1',      '=='):   'i1',
    ('float',   'i1',      '<' ):   'i1',
    ('float',   'i1',      '>' ):   'i1',
    ('float',   'i1',      '≤'):   'i1',
    ('float',   'i1',      '≥'):   'i1',
    ('float',   'i1',      '||'):   'i1',
    ('float',   'i1',      '&&'):   'i1',
    ('i32*',    'i32',     '+' ):   'i32*',
    ('i32',     'i32*',    '+' ):   'i32*',
    ('float*',  'i32',     '+' ):   'float*',
    ('i32',     'float*',  '+' ):   'float*',
}
```

Notice that some rules will involving implicit type casting. For example `int1` and `int32` by arithemetic operations, we need to implicitly cast `int1` to `int32` which we defined yielding a warning:

```
        elif exp_type(left) == 'i1' and exp_type(right) == 'i32':
            left.value = tp_visitor._get_builder().zext(val(left), int32)
        elif exp_type(left) == 'i32' and exp_type(right) == 'i1':
            right.value = tp_visitor._get_builder().zext(val(right), int32)
```

```
     if ret_type != exp_type(left):
            tp_visitor.n_warnings += 1
            print(str(TImplicitCastWarning(exp_type(left), ret_type)) + f" at
    position (line {left._lineno}, col {left._colno})")
```

Also we can force the expression to do type casting of which the ruls are also defined in a survey:

```
allowed_casting = [
    ('i1'    , 'i32'  ),
    ('i32'   , 'float'),
    ('i1'    , 'float'),
    ('float' , 'i1'   ),
    ('float' , 'i32'  ),
    ('[@ x i32]*'  , 'i32*'  ),
    ('[@ x float]*', 'float*'),
    ('[@ x [@ x i32]]*'  , 'i32*'  ),
    ('[@ x [@ x float]]*', 'float*'),
    ('[@ x [@ x [@ x i32]]]*'  , 'i32*'  ),
    ('[@ x [@ x [@ x float]]]*', 'float*'),
]
```

For the conversion between int and float:

```
        elif (src_type, tgt_type) == ('i32', 'float'):
            return tp_visitor._get_builder().sitofp(value, flpt)
        elif (src_type, tgt_type) == ('float', 'i32'):
            return tp_visitor._get_builder().fptosi(value, int32)
```

We support cast 1d/2d/3d-arry to pointers:

```
    elif (src_type, tgt_type) == ('[@ x i32]*', 'i32*'):
        val = tp_visitor._get_builder().gep(ref, [ir.Constant(int32, 0),
                                                  ir.Constant(int32, 0),])
        return val
    elif (src_type, tgt_type) == ('[@ x [@ x i32]]*', 'i32*'):
        val = tp_visitor._get_builder().gep(ref, [ir.Constant(int32, 0),
                                                  ir.Constant(int32, 0),])
        return tp_visitor._get_builder().bitcast(val, make_ptr(int32))
    elif (src_type, tgt_type) == ('[@ x [@ x [@ x i32]]]*', 'i32*'):
        val = tp_visitor._get_builder().gep(ref, [ir.Constant(int32, 0),
                                                  ir.Constant(int32, 0),
                                                  ir.Constant(int32, 0),
                                                  ir.Constant(int32, 0),])
```

H2

Chapter 5 - Code Generation

In the previous part, we have already obtained an abstract syntax tree. Then, we need to visit this tree to generate the target code. Of course we can directly generate target code like assembly language from this AST. However, on the consideration of scalability, we will first generate intermediate representation(IR) of this abstract syntax tree and then synthesize target code from this intermediate representation.

As for the generation of intermediate representation, we use a python package named `llvmlite`. `LLVMlite` is a small subset of the LLVM IR that we will be using throughout the course as the intermediate representation in our compiler. Conceptually, it is either an abstract assembly-like language or a even lower-level C-like language that is convenient to manipulate programmatically.

§5.1 LLVM Intermediate Representation

To give you a sense of structure of `LLVMlite` programs and the most basic features, the following is our running example, the simple recursive factorial function written in the concrete syntax of the `LLVMlite` IR.

H3

```
define i64 @fac(i64 %n) {              ; (1)
  %1 = icmp sle i64 %n, 0              ; (2)
  br i1 %1, label %ret, label %rec     ; (3)
ret:                                   ; (4)
  ret i64 1
rec:                                   ; (5)
  %2 = sub i64 %n, 1                   ; (6)
  %3 = call i64 @fac(i64 %2)           ; (7)
  %4 = mul i64 %n, %3
  ret i64 %4                           ; (8)
```

```
11      }
12
13      define i64 @main() {                    ; (9)
14        %1 = call i64 @fac(i64 6)
15        ret i64 %1
16      }
```

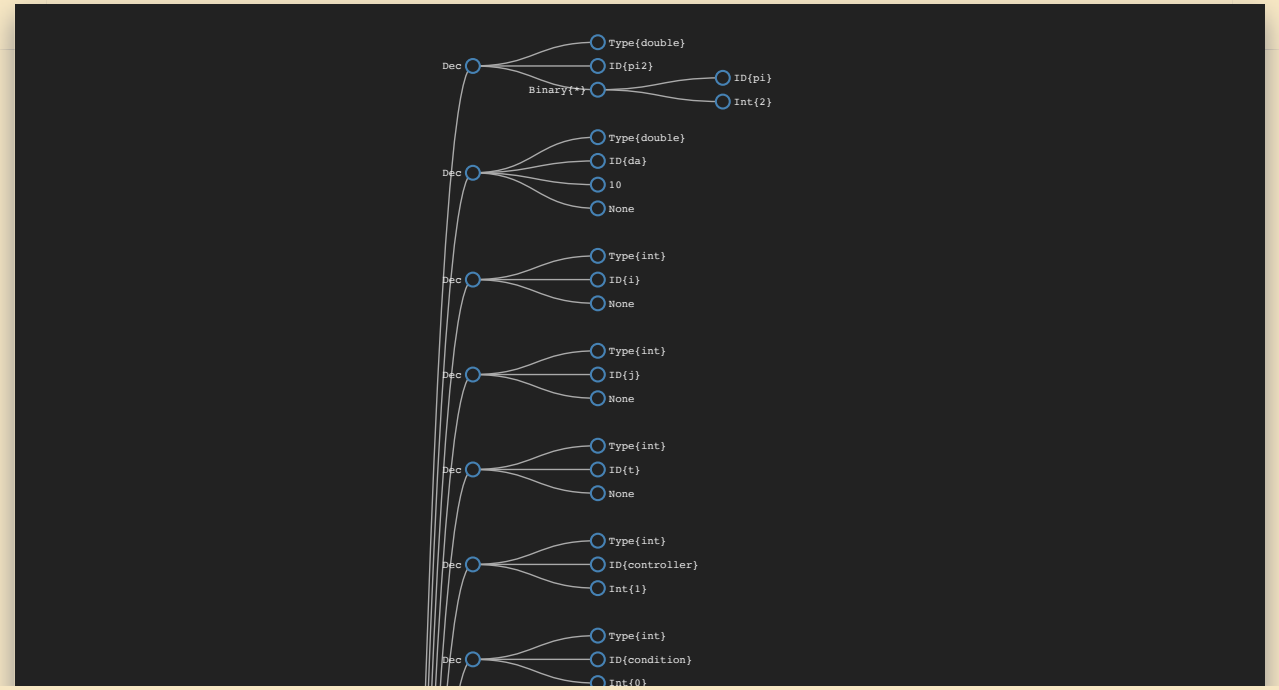
First, notice the function definition at (1). The i64 annotations declare the return type and the type of the argument n. The argument is prefixed with "%" to indicate that it's an identifier local to the function, while `fac` is prefixed with "@" to indicate that it is in scope in the entire compilation unit.

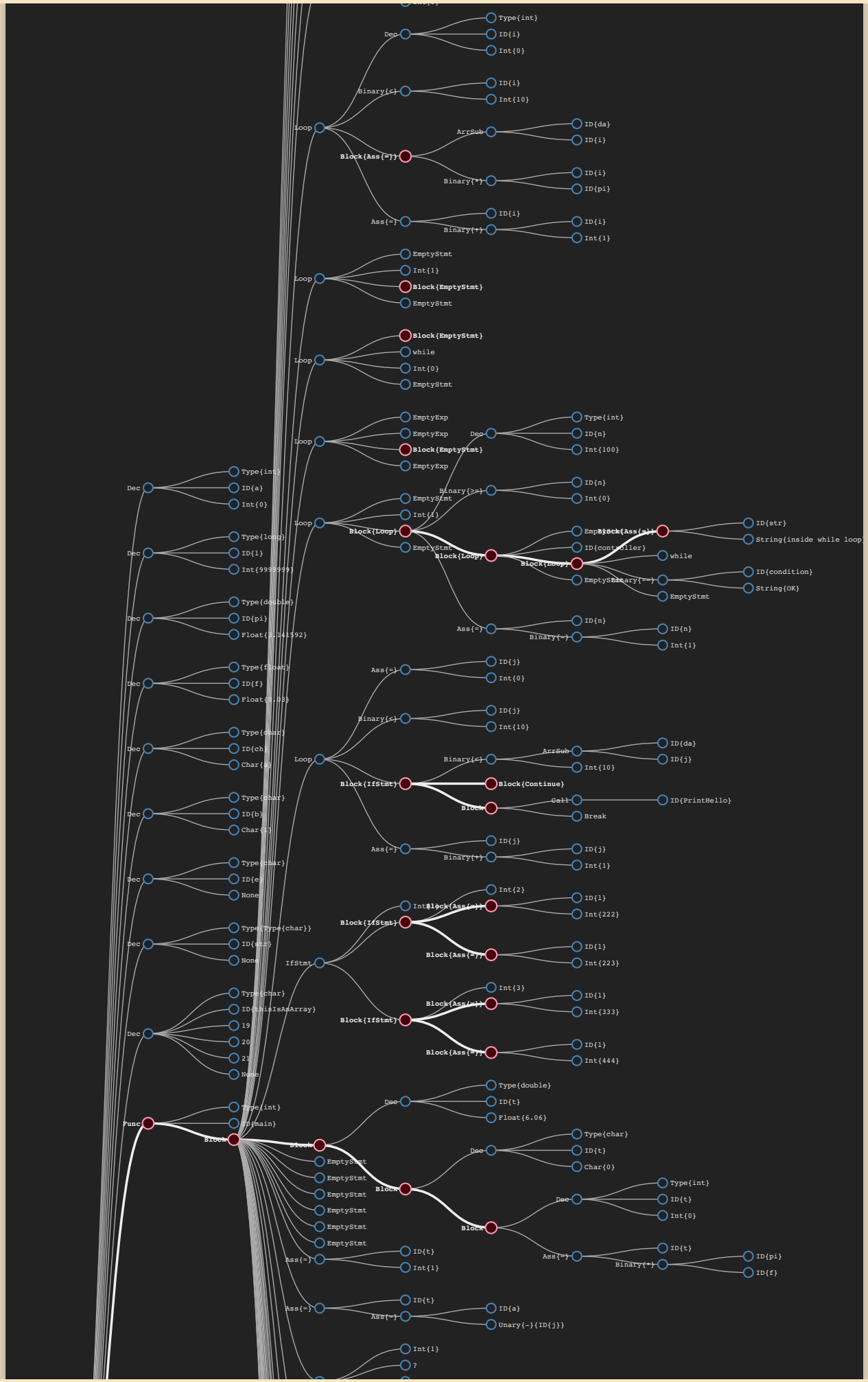
Next, at (2) we have the first instruction of the body of `fac`, which performs a signed comparison of the argument %n and 0 and assigns the result to the temporary %1. The instruction at (3) is a "terminator", and marks the end of the current block. It will transfer control to either ret at (4) or rec at (5). The labels at (4) and (5) each indicate the beginning of a new block of instructions. Notice that the entry block starting at (2) is not labeled: in LLVM it is illegal to jump back to the entry block of a function body. Moving on, (6) performs a subtraction and names the result %2. The i64 annotation indicates that both operands are 64-bit integers. The function `fac` is called at (7), and the result named %3. Again, the i64 annotations indicate that the single argument and the returned value are 64-bit integers.

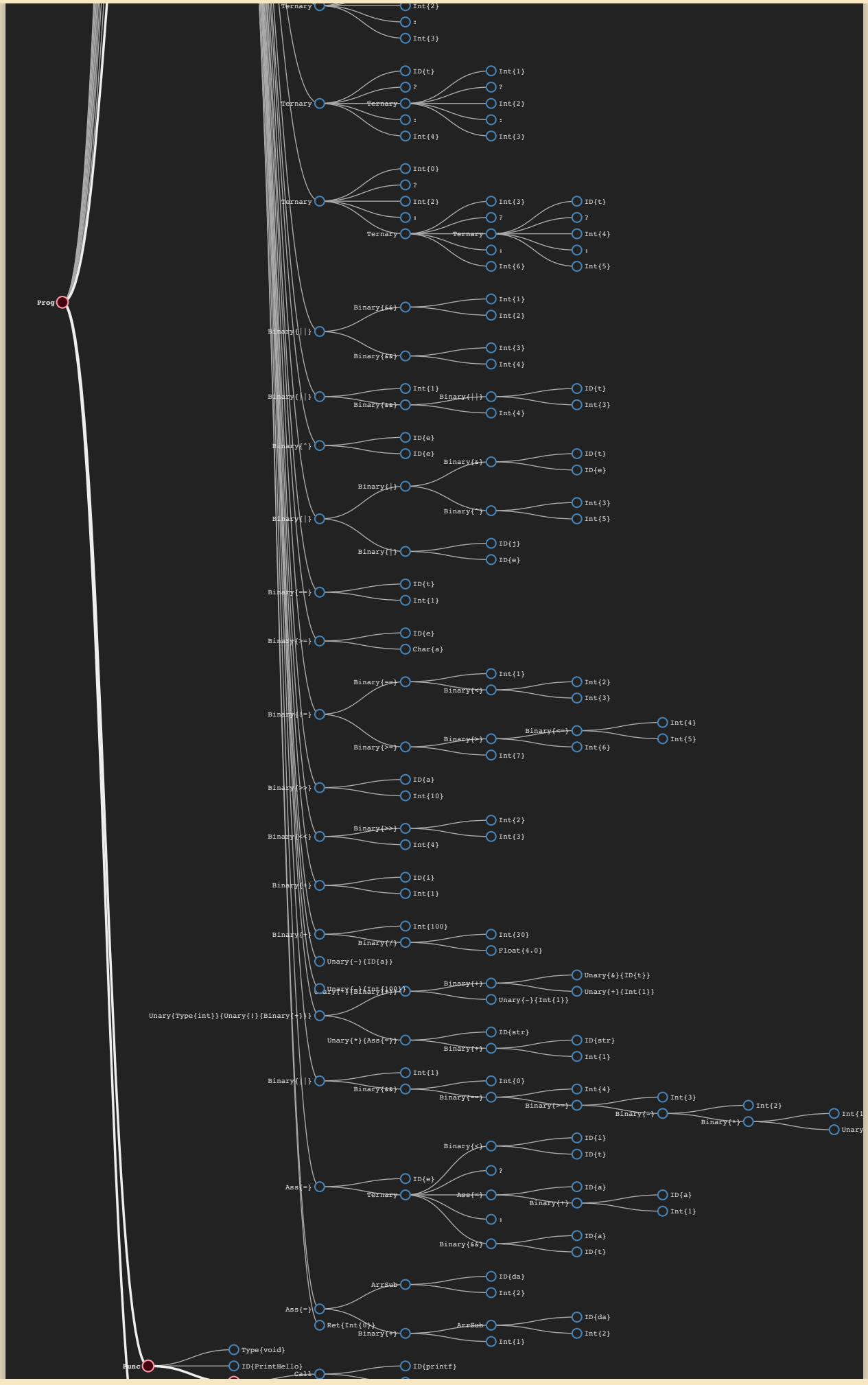
Finally, (8) returns from the function with the result named by %4, and (9) defines the main function of the program, which simply calls fac with a literal i64 argument.

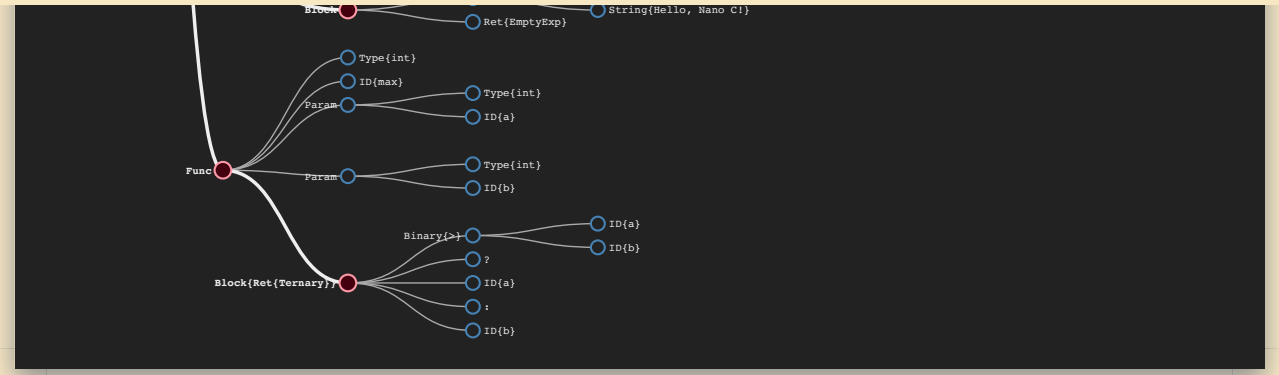
§5.2 Introduction of Visitor

H3




Dec
Type{int}
ID{i}
Int{0}
Binary{<}
ID{i}
Int{10}
Loop
ArrSub
ID{da}
ID{i}
Block{Ass{=}}
Binary{*}
ID{i}
ID{pi}
Ass{=}
ID{i}
Binary{+}
ID{i}
Int{1}
Loop
EmptyStmt
Int{1}
Block{EmptyStmt}
EmptyStmt
Loop
Block{EmptyStmt}
while
Int{0}
EmptyStmt
Loop
EmptyExp
EmptyExp
Block{EmptyStmt}
EmptyExp
Dec
Type{int}
ID{n}
Int{100}
Dec
Type{int}
ID{a}
Int{0}
Loop
EmptyStmt
Int{1}
Block{Loop}
EmptyStmt
Binary{>=}
ID{n}
Int{0}
Block{Loop}
Block{Ass{=}}
ID{str}
String{inside while loop}
ID{controller}
while
Binary{==}
ID{condition}
String{OK}
EmptyStmt
Ass{=}
ID{n}
Binary{+}
ID{n}
Int{1}
Dec
Type{long}
ID{l}
Int{9999999}
Dec
Type{double}
ID{pi}
Float{3.141592}
Dec
Type{float}
ID{f}
Float{1.03}
Dec
Type{char}
ID{ch}
Char{a}
Dec
Type{char}
ID{b}
Char{c}
Dec
Type{char}
ID{e}
None
Dec
Type{Type{char}}
ID{str}
None
Dec
Type{char}
ID{thisIsAnArray}
19
20
21
None
Ass{=}
ID{j}
Int{0}
Loop
Binary{<}
ID{j}
Int{10}
Block{IfStmt}
Binary{<}
ArrSub
ID{da}
ID{j}
Int{10}
Block{Continue}
Block
Call
ID{PrintHello}
Break
Ass{=}
ID{j}
Binary{+}
ID{j}
Int{1}
IfStmt
Block{IfStmt}
Int{2}
Block{Ass{=}}
ID{l}
Int{222}
Block{Ass{=}}
ID{l}
Int{223}
Block{IfStmt}
Int{3}
Block{Ass{=}}
ID{l}
Int{333}
Block{Ass{=}}
ID{l}
Int{444}
Func
Type{int}
ID{main}
Block
Block
Dec
Type{double}
ID{t}
Float{6.06}
EmptyStmt
EmptyStmt
EmptyStmt
EmptyStmt
EmptyStmt
EmptyStmt
Block
Dec
Type{char}
ID{t}
Char{0}
Block
Dec
Type{int}
ID{t}
Int{0}
Ass{=}
ID{t}
Binary{+}
ID{pi}
ID{f}
Ass{=}
ID{t}
Int{1}
Ass{=}
ID{t}
Ass{=}
ID{a}
Unary{-}{ID{j}}
Int{1}
?


Prog
Ternary
Int{2}
:
Int{3}
ID{t}
?
Ternary
Int{1}
?
Int{2}
:
Int{3}
Int{4}
Int{0}
?
Int{2}
:
Ternary
Int{3}
?
Ternary
ID{t}
?
Int{4}
:
Int{5}
:
Int{6}
Binary{&&}
Int{1}
Int{2}
Binary{||}
Binary{&&}
Int{3}
Int{4}
Binary{|}
Int{1}
Binary{&&}
Binary{|||}
ID{t}
Int{3}
Int{4}
Binary{^}
ID{e}
ID{e}
Binary{&}
ID{t}
ID{e}
Binary{|}
Binary{^}
Int{3}
Int{5}
Binary{|}
ID{j}
ID{e}
Binary{==}
ID{t}
Int{1}
Binary{>=}
ID{e}
Char{a}
Binary{==}
Int{1}
Binary{<}
Int{2}
Int{3}
Binary{<}
Binary{>=}
Binary{>}
Binary{<=}
Int{4}
Int{5}
Int{6}
Int{7}
Binary{>>}
ID{a}
Int{10}
Binary{<<}
Binary{>>}
Int{2}
Int{3}
Int{4}
Binary{+}
ID{i}
Int{1}
Binary{+}
Int{100}
Binary{/}
Int{30}
Float{4.0}
Unary{-}{ID{a}}
Unary{!}{Binary{+}}
Binary{+}
Unary{&}{ID{t}}
Unary{+}{Int{1}}
Unary{-}{Int{1}}
Unary{Type{int}}{Unary{!}{Binary{+}}}
Unary{*}{Ass{=}}
ID{str}
Binary{+}
ID{str}
Int{1}
Binary{||}
Int{1}
Binary{&&}
Int{0}
Binary{==}
Int{4}
Binary{>=}
Int{3}
Binary{-}
Int{2}
Binary{*}
Int{1}
Unary
Ass{=}
ID{e}
Ternary
Binary{<}
ID{i}
ID{t}
?
Ass{=}
ID{a}
Binary{+}
ID{a}
Int{1}
:
Binary{&&}
ID{a}
ID{t}
Ass{=}
ArrSub
ID{da}
Int{2}
Ret{Int{0}}
Binary{*}
ArrSub
ID{da}
Int{2}
Int{1}
Func
Type{void}
ID{PrintHello}
Call
ID{printf}



To generate the IR mentioned above, we use a visitor class named `NanoVisitor`. Given an abstract syntax tree like the above figure, the visitor will first visit the `ProgNode` which is marked as root. Then the visitor will well grounded travel the children of root.

Design Pattern: Reflection

One problem in traveling step is that visitor does not know the type of node and thus can not do specific actions depending on the type of node. To solve this problem, we use the design pattern of reflection. That is, the visitor will call the `accept` method of class node. The derived classes of `Node` will override the `accept` method to their own visiting procedure. So, the calling flow is just like `Visitor->Node->Visitor` which can be interpreted as reflection. A piece of sample code is shown as below:

H4

```
class NanoVisitor(Visitor):
    # ...
    def visitProgNode(self, node: ProgNode):
        # ...
        for func in node.funcs:
            func.accept(self)
        # ...
```

```
class ProgNode(Node):
    # ...
    def accept(self, visitor: NanoVisitor):
        return visitor.visitProgNode(self)
class FuncNode(Node):
    # ...
    def accept(self, visitor: NanoVisitor):
        return visitor.visitFuncNode(self)
# ...
```

§5.4 Attributes

During the IR generation, or in the other words, travelly visiting the AST, we need to set some attributes which may be referenced afterwards. Several key attributes in the visitor are shown as below:

H3

- cur_module: a `llvmlite.ir.Module` object. This obj provides the IR generation environment, we can do nothing without it.
- cur_func_name: a `str` object. Mark which function is currently processed, initialized with empty string.
- builder_stack: a `llvmlite.ir.IRBuilder` object. This obj is used to insert IR instructions after the blocks.
- scope_stack: a `list` object. Each element in the stack is of type `dict`. This obj denotes the symbol table and scopes. The procedure of getting an identifier in the current scope just work like below:

```
1       def _get_identifier(self, name):
2           for d in self.scope_stack[::-1]:  # reversing the scope_block
3               if name in d:
4                   return d[name]['ref']
5           return None
```

- loop_exit_stack & loop_entr_stack: a `list` object. Since we support nested loop statements, continue and break statements, we need to remember the last entrance/exit point(usually interpreted as the beginning of a block). The next time we analysis break or continue, we can pop the exit stack or entrance stack and branch there.
- in_global: a `bool` object. Mark whether the visitor is currently at the global scope. If it is, the declared variable will be allocated in the global space.

```
class NanoVisitor(Visitor):
    def __init__(self):
        # we do not need list since we only support single filre compiling
        self.cur_module = None
        # we do not need list since we do not support sub-procedure
        self.cur_func_name = ''
        self.builder_stack = []
        self.block_stack = []
        self.scope_stack = []
        self.loop_exit_stack = []
        self.loop_entr_stack = []
        self.in_global = True
        #  ...
```

§5.3 Implementation of Visitor

We select several visiting examples to illustrate how visiting procedure generates IR.

VisitBlockNode

H3

The special point of the BlockNode is that, we need to create a new scope every time we enter the block.

H4

```
    def visitBlockNode(self, node: BlockNode, scope=None):
        self._push_scope(scope)
        for stmt in node.stmts:
            stmt.accept(self)
        return self._pop_scope()
```

for every statements in this block, it will call accept method and generates their own part IR.

VisitIfStmtNode

For the IfStmtNode, we will first evaluate the condition:

H4

```
    def visitIfStmtNode(self, node: IfStmtNode):
        node.cond.accept(self)
        pred = self._get_builder().icmp_signed('≠',
                                               val(node.cond),
                                               ir.Constant(int32, 0))
        if type(node.ifbody) == EmptyStmtNode:
            return
        #...
```

And then generate IR in `then` and `otherwise` part:

```
        if type(node.elsebody) == EmptyStmtNode:
            with self._get_builder().if_then(pred) as then:
                self._push_scope()
                node.ifbody.accept(self)
                self._pop_scope()
        else:
            with self._get_builder().if_else(pred) as (then, othrewise):
                with then:
                    # ifbody
                    self._push_scope()
                    node.ifbody.accept(self)
                    self._pop_scope()
                with othrewise:
                    # elsebody
                    self._push_scope()
                    node.elsebody.accept(self)
                    self._pop_scope()
```

Notice that we use the APIs of llvmlite which are `IRBuilder.if_then` and `IRBuilder.if_else`. The former will yield a basic block, and we can generate `then` part IR in this block. The latter will yield two context managers, we can use keyword `with` to set at the begining of that block and generate `then/otherwise` part IR in them. The generating IR work is recursively done in the accept calling.

VisitLoopNode

We support three loop statements: `for`, `while` and `do-while`. Their procedure of generating IR are somewhat different. This is because the difference of their loop structures which are illustrated as below:

H4

```
for:
    pre: EmptyStmtNode / DecNode
    cond: EmptyExpNode / subclass of EmptyExpNode / subclass of EmptyLiteralNode / IDNode
    body: BlockNode
    post: EmptyExpNode / subclass of EmptyExpNode / subclass of EmptyLiteralNode / IDNode
while:
    pre: EmptyStmtNode
    cond: EmptyExpNode / subclass of EmptyExpNode / subclass of EmptyLiteralNode / IDNode
    body: BlockNode
    post: EmptyStmtNode
```

```
do-while:
    pre: EmptyStmtNode()
    cond: EmptyExpNode / subclass of EmptyExpNode / subclass of EmptyLiteralNode / IDNode
    body: BlockNode()
    post: EmptyStmtNode()
```

Based on above analysis, we divide the loop statements into two cases:

- **`for`** and **`while`**:

```
""" do-while
     ...
    branch body_block
body_block:
    body ...
    branch cond_block
cond_block [continue target]:
    cond ...
    branch(cond, body_block, tail_block)
tail_block [break target]:
     ...
"""
```

- **`do-while`**:

```
""" for & while
     ...
    pre ...
    branch cond_block
cond_block:
    cond ...
    branch(cond, body_block, tail_block)
body_block:
    body ...
    branch post_block
post_block [continue target]:
    post ...
    branch cond_block
tail_block [break target]:
     ...
"""
```

In addition, to support the declaration in the for statements like `for(int i=0;i<n;i++)`, we adopt the scope creating/deleting procedure like this:

```
                    ... scope_original ...
           /=======================scope_new_0===/
           /  pre -> cond <----|                 /
           /  /=========|======|==scope_new_1===/
           /  /         body    |                /
           /  /=========|======|================/
           /            post----|                /
           /====================================/
```

We show the for/while IR generation below, for more details pleased refer to our source code.

```
            cond_block = self._get_builder().append_basic_block()
            body_block = self._get_builder().append_basic_block()
            post_block = self._get_builder().append_basic_block()
            tail_block = self._get_builder().append_basic_block()
            self.loop_entr_stack.append(post_block)
            self.loop_exit_stack.append(tail_block)

            # scope_new_0
            self._push_scope()
            if type(node.pre) != EmptyStmtNode:
                node.pre.accept(self)
            self._get_builder().branch(cond_block)

            # entrance
            self._get_builder().position_at_start(cond_block)
            if type(node.cond) != EmptyExpNode:
                node.cond.accept(self)
                cond = self._get_builder().icmp_signed('≠',
                                                       val(node.cond),
                                                       ir.Constant(int32,
0))
            else:
                cond = int1(1)
            self._get_builder().cbranch(cond, body_block, tail_block)

            # scope_new_1 auto created when visitBlockNode
            self._get_builder().position_at_start(body_block)
            node.body.accept(self)
            self._get_builder().branch(post_block)

            # return to scope 0
```

```
31              self._get_builder().position_at_start(post_block)
32              if type(node.post) != EmptyStmtNode:
33                  node.post.accept(self)
34              self._get_builder().branch(cond_block)
35              self._pop_scope()
36
37              # after
38              self._get_builder().position_at_start(tail_block)
39
40              self.loop_entr_stack.pop()
41              self.loop_exit_stack.pop()
```

VisitBinopNode

The generation of binary operation IR is very simple, it only calls some APIs of llvmlite. The used APIs are: `llvm.ir.IRBuilder.add`, `llvm.ir.IRBuilder.sub`, `llvm.ir.IRBuilder.mul`, `llvm.ir.IRBuilder.sdiv`, `llvm.ir.IRBuilder.srem`, `llvm.ir.IRBuilder.shl`, `llvm.ir.IRBuilder.ashr`, `llvm.ir.IRBuilder.icmp_signed`, `llvm.ir.IRBuilder.fadd`, `llvm.ir.IRBuilder.fsub`, `llvm.ir.IRBuilder.fmul`, `llvm.ir.IRBuilder.fdiv`, `llvm.ir.IRBuilder.frem`, `llvm.ir.IRBuilder.fcmp_ordered`, etc.

For example, the addition IR of two floating point number is generated by:

```
1               if node.op == '+':
2                   node.value = self._get_builder().fadd(
3                       val(node.left),
4                       val(node.right))
```

where the fadd method will automatically write an IR correspoing to floating point number addition into the whole IR module.

VisitUnaryNode

The unary operations are quite like binary operations. However, the differences are that some of unary operations like `*` and `&` are related to addressed and pointers which make them more complicated. For example, if we do `UnaryNode(*, IDNode(a))`, we need to first guarantee identifier is previously decalared and is a pointer type; Then we get the reference of a which is `ref(a):=&a` and then load the value which is `val(a):=load(ref(a))` and finally do the start operation `ref(*a):=val(a)`. The next time we need to load the value, we just `val(*a)=load(ref(*a))` and in the other case, store the value into `*a`, we use `store(value, ref(*a))`.

Some code is shown as below:

```
            # ...
            elif node.op == '*':
                node.ref = val(node.node)
            elif node.op == '&':
                node.value = ref(node.node)
            # ...
```

VisitArrSubNode

Since we support array type, we need to support getting value by indices like `arr[i][j][k]`. In LLVM IR, reference of `arr` is stored as pointer to its memory, and the type is `[i x [j x [k x int32]]]*`. In our AST, this
H4 expression is interpreted as `ArrSubNode(ArrSubNode(ArrSubNode(IDNode(arr), IDNode(i)), IDNode(j)), IDNode(k))`. So we can iteratively get the pointer use `llvmlite.ir.IRBuilder.gep`:

```
        def visitArrSubNode(self, node: ArrSubNode):
            node.subee.accept(self)
            node.suber.accept(self)
            node.ref = self._get_builder().gep(ref(node.subee),
                                               [ir.Constant(int32, 0),
                                                val(node.suber), ])
            node.exp_type = str(node.ref.type.pointee)
```

Notice that every time we do indexing, we need to pass one extral 0 value. This is because when the gep method is returned, the return value will be wrapped by as pointer type. We need a 0 index to get rid of this wrapping.

H2 Chapter 6 - Compilation

```
(local38) →  compiler git:(master) x cd results
(local38) →  results git:(master) x clang -S irgen.ll -o irgen.s
warning: overriding the module target triple with x86_64-pc-windows-msvc19.28.29334 [-Woverride-module]
1 warning generated.
(local38) →  results git:(master) x clang irgen.s -o irgen.exe
(local38) →  results git:(master) x ./irgen.exe
(local38) →  results git:(master) x echo $lastExitCode
1
(local38) →  results git:(master) x
```

This is an example of manually compiling the intermediate representation generated by our `nanoirgen.py`. You can always just use `python src/nanoirgen.py -g -i samples/quicksort.c -o results/irgen.ll -e .exe` to generate the output directly

§6.1 IR to Assembly

This process can be executed by a the LLVM static compiler `llc` or just `clang`

H3 If using clang, we should use `clang irgen.ll -S -o irgen.s` to produce the assembly for viewing

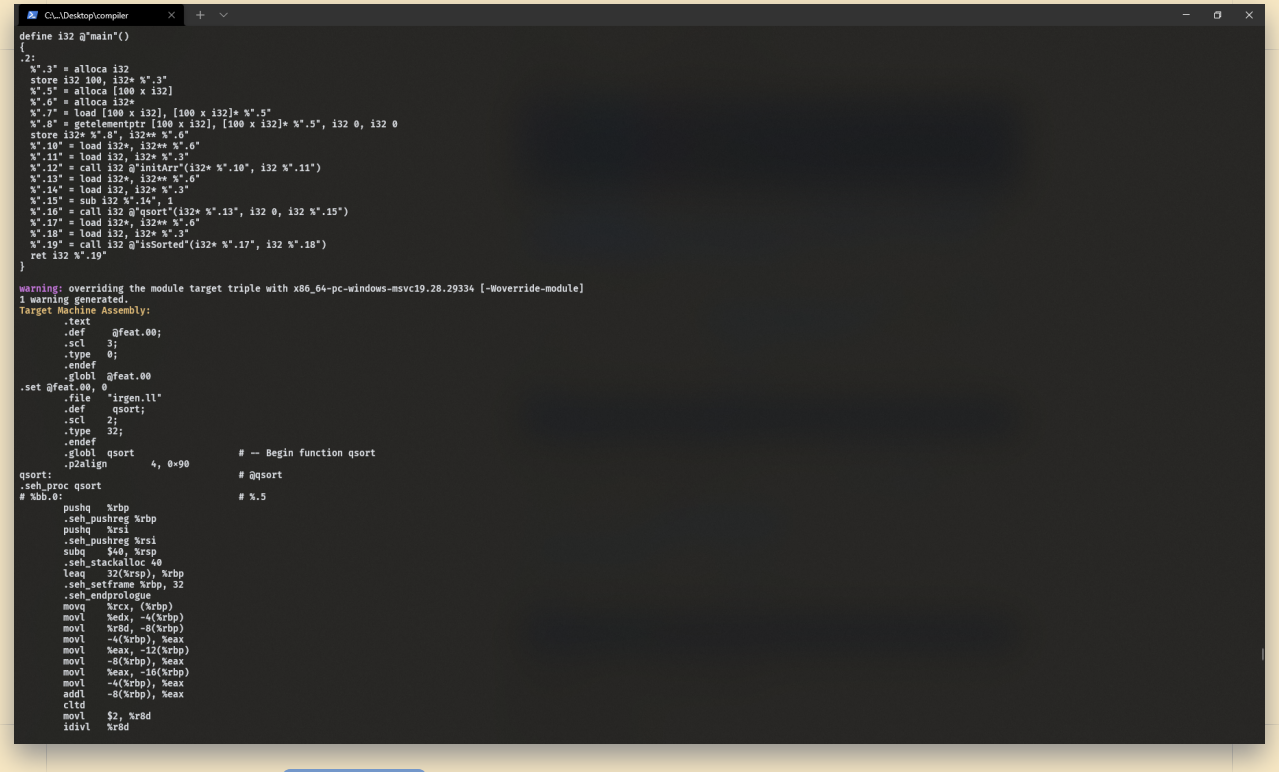
```
clang <input_llvm_ir_file> -S -o <output_asm_file>
```

In our implementation, we used the `os` package to call a system program and generate the result for us

shell:

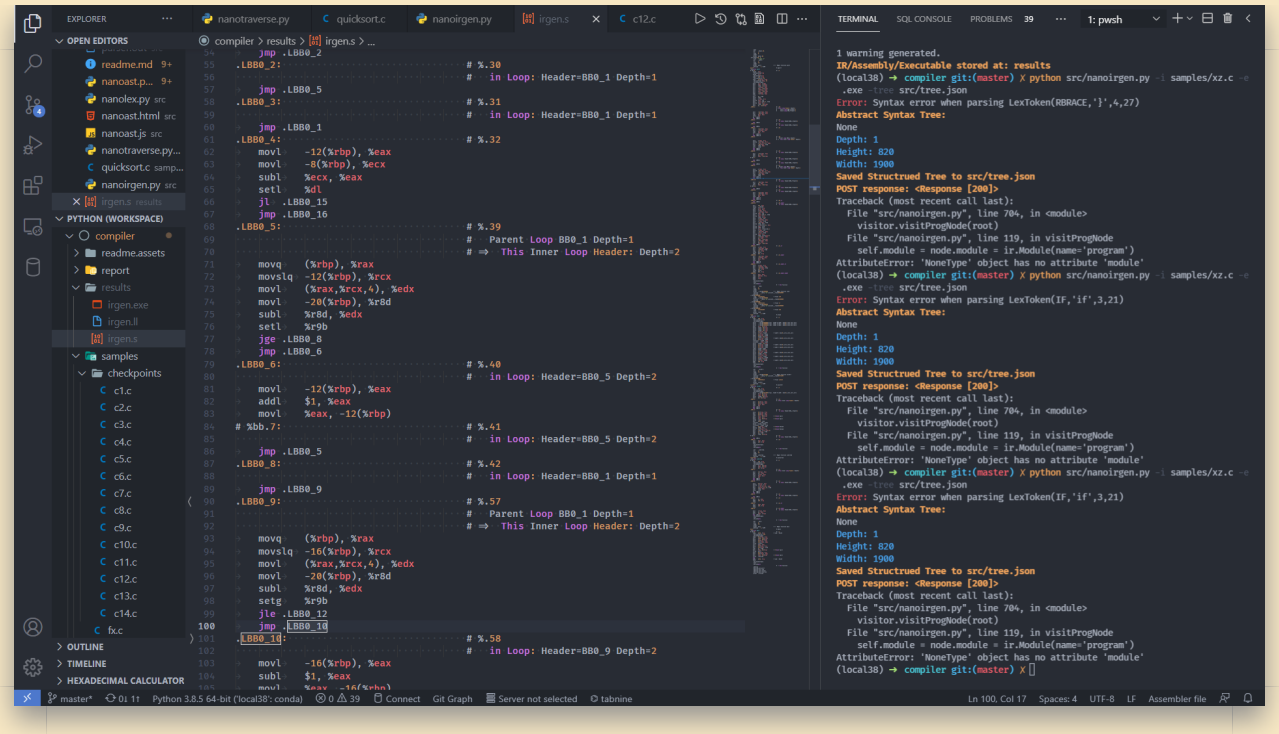
python:

```
os.system(' '.join(["clang", args.output, "-S", "-o", ass]))
```



File content of `irgen.s`

(Target Assembly Code)



§6.2 Assembling the Executable

With the assembly, we can simple use a compiler like `gcc` or `clang` to produce the final executable machine code

H3

```
clang <input_asm_file> -o <output_exec_file>
```

This can be done with `clang irgen.s -o irgen`

or `gcc irgen.s -o irgen`

In our implementation, we used the `os` package to call a system program and generate the result for us
shell:

python:

```
os.system(' '.join(["clang", ass, "-o", exe]))
```

H2 Chapter 7 - Test Cases

§7.1 Lexer

As the token list is defined in the Token Definition section, we wrote a file including all possible tokens (which is listed below) to test our lexer (`src/nanolex.py`). To run the lexer, use the command:

H3

```
$ python3 src/nanolex.py samples/LexTest.txt
```

- **Input: The file to be tokenized.**
- **Output: Tokens (in the format of `LexToken(<type>,<value>,<lineno>,<lexpos>)`).**

Test cases:

1. **Keywords**

   Input:

   ```
   int float char void
   if else
   else if
   do while
   for
   continue break
   return
   ```

   Output:

   ```
   asudy@Asudys-MacBook compiler % python3 src/nanolex.py samples/LexTest.txt
   LexToken(INT,'int',2,12)
   LexToken(FLOAT,'float',2,16)
   LexToken(CHAR,'char',2,22)
   LexToken(VOID,'void',2,27)
   LexToken(IF,'if',3,32)
   LexToken(ELSE,'else',3,35)
   LexToken(ELSE,'else',4,40)
   LexToken(IF,'if',4,45)
   LexToken(DO,'do',5,48)
   LexToken(WHILE,'while',5,51)
   LexToken(FOR,'for',6,57)
   LexToken(CONTINUE,'continue',7,61)
   LexToken(BREAK,'break',7,70)
   LexToken(RETURN,'return',8,76)
   ```

2. **Identifiers**

   Input:

   ```
   thisIsAnIdentifier
   these are four Identifiers
   ```

   Output:

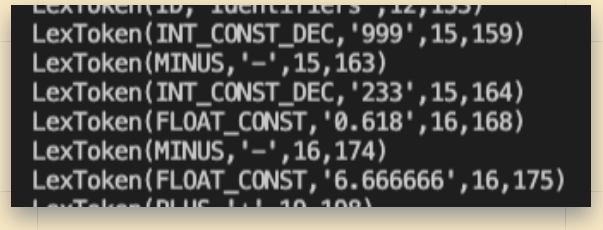
   ```
   LexToken(ID,'thisIsAnIdentifier',11,99)
   LexToken(ID,'these',12,118)
   LexToken(ID,'are',12,124)
   LexToken(ID,'four',12,128)
   LexToken(ID,'Identifiers',12,133)
   LexToken(INT_CONST_DEC,'999',15,159)
   ```

3. **Constants**

Input:

```
999 -233
0.618 -6.666666
```

Output:

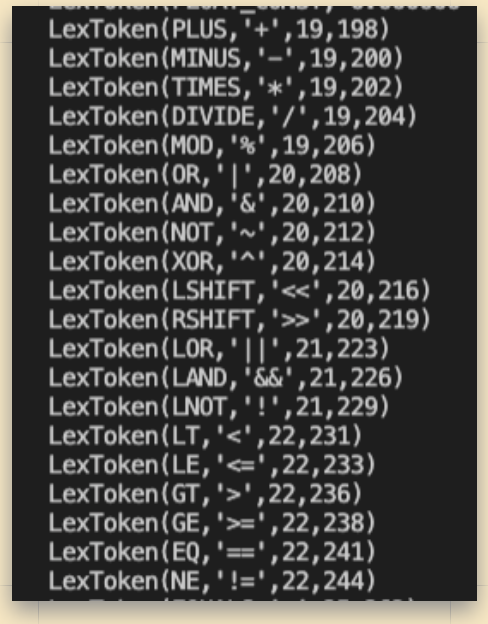


4. **Operators**

Input:

```
+ - * / %
| & ~ ^ << >>
|| && !
< ≤ > ≥ == ≠
```

Output:

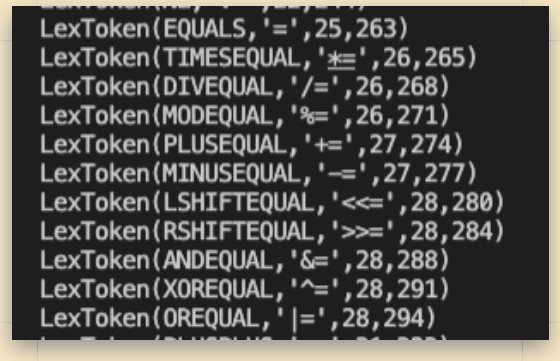


5. **Assignments**

Input:

```
=
*= /= %=
+= -=
<<= >>= &= ^= |=
```

Output:

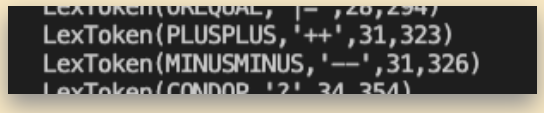


6. **Increment & Decrement**

Input:

```
++ --
```

Output:



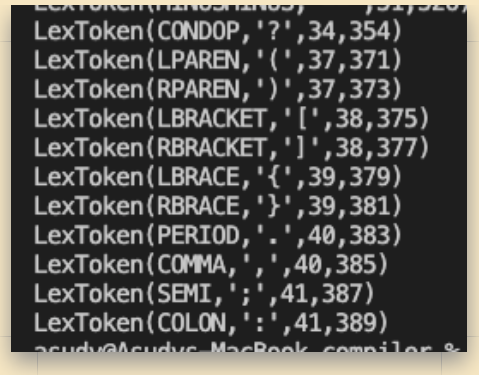
7. **Conditional Operator & Delimeters**

Input:

```
// Conditional Operator
?

// Delimeters
( )
[ ]
{ }
. ,
; :
```

Output:



In conclusion, the program `nanolex.py` tokenized the input files as desired in all the test cases. Test passed.

§7.2 Yacc

The grammar productions used in Nano C is listed in BNF Definition for the Nano C Language. File `samples/ParserTest.c` contains **all grammar elements** which Nano C supports and is designed to test the parser implemented by yacc. To test the parser, simply use the command below:

H3

```
$ python3 src/nanoyacc.py -i samples/ParserTest.c
```

The invocation of `nanoyacc.py` supports *command line options*. To learn how to use the options, you may do:

```
$ python3 src/nanoyacc.py -h
```

- **Input:** The file to be parsed.
- **Output:**
  1. **Abstract syntax tree (AST)**
  2. **Structed AST**
  3. **Visualized AST**

The testing file (`samples/ParserTest.c`) is as the following. And the purposes of each statement / expression is commented right above them.

```
/* Multi-line Comment */
/**
 * @file    ParserTest.c
 * @note    This file gives test cases to all language elements
 *          (statements, expressions, etc.) in our parser. Each test
 *          case is marked by a one-line comment.
 */


```

```c
/* Program */
/* Global Declarations */
int a = 0;
long l = 9999999;
double pi = 3.141592;
float f = 0.03;
// declaration list
char ch = 'a', b = '1', e;
// pointer
char * str;
// array
char thisIsAnArray[19][20][21];


// Function 1
int main() {
    double pi2 = pi * 2;
    double da[10];
    int i, j, t, controller = 1, condition = 0;


    /**
     *  @note   Statements, Control Flows, Scopes
     **/

    /* For Loop */
    for ( int i = 0 ; i < 10 ; i++ )
        /* Block of single statementm && Array Substitution */
        da[i] = i * pi;

    /* Empty Loops */
    while (1);
    do { ; } while (0);
    for ( ; ; ) { ; }

    /* Nested Loop */
    while ( 1 )
        for ( int n = 100; n ≥ 0; n-- )
            while ( controller )
                do str = "inside while loop";
                while ( condition == "OK" );


    for ( j = 0; j < 10; j++ ) {
        /* If-else Statement */
        if ( da[j] < 10 )
            // continue control
            continue;
        else {
```

```
            /* Function Call */
            PrintHello();
            // break control
            break;
        }
    }

    /* Dangling If */
    if ( 1 )
        if ( 2 ) l = 222;
        else l = 223;
    else if ( 3 ) l = 333;
    else l = 444;

    /* Nested Scopes */
    {   double t = 6.06;
        {   char t = '0';
            {   int t = 0;
                t = pi * f;
            }
        }
    }

    /* Empty Statements */
    ;;;;;;


    /**
     *  @note   Expressions
     **/

    /* Operators & Nested */
    // assignment
    t = 1;
    t = a = -j;
    // t <<= a;
    // t &= 0;
    // t ^= b;
    // t += ch;
    // t *= 20;
    // t %= j;
    // conditional (ternary op)
    1 ? 2 : 3;
    t ? 1 ? 2 : 3 : 4;
    0 ? 2 : 3 ? t ? 4 : 5 : 6;
    // logical
    1 && 2 || 3 && 4;
    1 || (t || 3) && 4;
    // bitwise
```

```
108     e ^ e;
109     t & e | 3 ^ 5 | (j | e);
110     // comparison
111     t == 1;
112     e ≥ 'a';
113     1 == 2 < 3 ≠ 4 ≤ 5 > 6 ≥ 7;
114     // shift
115     a >> 10;
116     2 >> 3 << 4;
117     // arithmetic
118     i + 1;
119     100 + 30 / 4.0;
120     // unary
121     ~a;
122     -100;
123     (int)!( *(&t + +1 + -1) + *++str );      // p++ is treated the same as
    ++p
124
125     /* Operator Precedence */
126     1 || 0 && 4 == 3 ≥ 2 - 1 * -1;
127     e = i < t ? ++a : a && t;                // e = ( ((a < t) ? (a++) : a)
    = t )
128     // a ? t : e = ch;                       // ERROR, assignment must have
    a unary lvalue.
129     ++da[2];                                 // (da[2])++
130
131
132     return 0;
133 }
134
135
136 /* Function 2, void Return Value, void Parameter */
137 void PrintHello(void) {
138     /* Function Call */
139     printf("Hello, Nano C!");
140     /* Empty Return */
141     return;
142 }
143
144
145 /* Function 3, Typed Return Value */
146 int max(int a, int b) {
147     // return exp
148     return a > b ? a : b;
149 }
```

Output after parsing the testing file:

1. CLI-printed **Abstract Syntax Tree (AST)** (partial)

```
asudy@Asudys-MacBook compiler % python3 src/nanoyacc.py -i samples/ParserTest.c
Abstract Syntax Tree:

ProgNode(
DecNode( TypeNode(int) IDNode(a) = IntNode(0) )
DecNode( TypeNode(long) IDNode(l) = IntNode(9999999) )
DecNode( TypeNode(double) IDNode(pi) = FloatNode(3.141592) )
DecNode( TypeNode(float) IDNode(f) = FloatNode(0.03) )
DecNode( TypeNode(char) IDNode(ch) = CharNode(a) )
DecNode( TypeNode(char) IDNode(b) = CharNode(1) )
DecNode( TypeNode(char) IDNode(e) )
DecNode( TypeNode(PTR)(TypeNode(char)) IDNode(str) )
DecNode( TypeNode(char) IDNode(thisIsAnArray)[19][20][21] )
|   FuncNode(27, 1)( TypeNode(int) IDNode(main)(  ) { BlockNode(2
|   |   DecNode( TypeNode(double) IDNode(pi2) = BinaryNode( IDNode(pi) * IntNode(2) ) )
|   |   DecNode( TypeNode(double) IDNode(da)[10] )
|   |   DecNode( TypeNode(int) IDNode(i) )
|   |   DecNode( TypeNode(int) IDNode(j) )
|   |   DecNode( TypeNode(int) IDNode(t) )
|   |   DecNode( TypeNode(int) IDNode(controller) = IntNode(1) )
|   |   DecNode( TypeNode(int) IDNode(condition) = IntNode(0) )
|   |   LoopNode( DecNode( TypeNode(int) IDNode(i) = IntNode(0) ) LOOP(BinaryNode( IDNode(i) < IntNode(10) )) { BlockNode(3
|   |   |   AssNode( ArrSubNode( IDNode(da)[IDNode(i)] ) = BinaryNode( IDNode(i) * IDNode(pi) ) )
|   |   )EndBlock3, AssNode( IDNode(i) = BinaryNode( IDNode(i) + IntNode(1) ) ) } )
|   |   LoopNode( EmptyStmtNode() LOOP(IntNode(1)) { BlockNode(3
|   |   |   EmptyStmtNode()
|   |   )EndBlock3, EmptyStmtNode() } )
|   |   LoopNode( BlockNode(3
|   |   |   EmptyStmtNode()
|   |   )EndBlock3 LOOP(while) { IntNode(0), EmptyStmtNode() } )
|   |   LoopNode( EmptyExpNode() LOOP(EmptyExpNode()) { BlockNode(3
|   |   |   EmptyStmtNode()
|   |   )EndBlock3, EmptyExpNode() } )
|   |   LoopNode( EmptyStmtNode() LOOP(IntNode(1)) { BlockNode(3
|   |   |   LoopNode( DecNode( TypeNode(int) IDNode(n) = IntNode(100) ) LOOP(BinaryNode( IDNode(n) >= IntNode(0) )) { BlockNode(4
|   |   |   |   LoopNode( EmptyStmtNode() LOOP(IDNode(controller)) { BlockNode(5
|   |   |   |   |   LoopNode( BlockNode(6
|   |   |   |   |   |   AssNode( IDNode(str) = StringNode("inside while loop") )
|   |   |   |   |   )EndBlock6 LOOP(while) { BinaryNode( IDNode(condition) == StringNode("OK") ), EmptyStmtNode() } )
|   |   |   |   )EndBlock5, EmptyStmtNode() } )
|   |   |   )EndBlock4, AssNode( IDNode(n) = BinaryNode( IDNode(n) - IntNode(1) ) ) } )
|   |   )EndBlock3, EmptyStmtNode() } )
|   |   LoopNode( AssNode( IDNode(j) = IntNode(0) ) LOOP(BinaryNode( IDNode(j) < IntNode(10) )) { BlockNode(3
|   |   |   IfStmtNode( IF (BinaryNode( ArrSubNode( IDNode(da)[IDNode(j)] ) < IntNode(10) )) { BlockNode(4
|   |   |   |   ContinueNode()
|   |   |   )EndBlock4 } ELSE { BlockNode(4
|   |   |   |   CallNode( IDNode(PrintHello)() )
|   |   |   |   BreakNode()
|   |   |   )EndBlock4 } )
|   |   )EndBlock3, AssNode( IDNode(j) = BinaryNode( IDNode(j) + IntNode(1) ) ) } )
|   |   IfStmtNode( IF (IntNode(1)) { BlockNode(3
|   |   |   IfStmtNode( IF (IntNode(2)) { BlockNode(4
|   |   |   |   AssNode( IDNode(l) = IntNode(222) )
|   |   |   )EndBlock4 } ELSE { BlockNode(4
|   |   |   |   AssNode( IDNode(l) = IntNode(223) )
|   |   |   )EndBlock4 } )
|   |   )EndBlock3 } ELSE { BlockNode(3
|   |   |   IfStmtNode( IF (IntNode(3)) { BlockNode(4
|   |   |   |   AssNode( IDNode(l) = IntNode(333) )
|   |   |   )EndBlock4 } ELSE { BlockNode(4
|   |   |   |   AssNode( IDNode(l) = IntNode(444) )
|   |   |   )EndBlock4 } )
|   |   )EndBlock3 } )
|   |   BlockNode(3
|   |   |   DecNode( TypeNode(double) IDNode(t) = FloatNode(6.06) )
|   |   |   BlockNode(4
|   |   |   |   DecNode( TypeNode(char) IDNode(t) = CharNode(0) )
|   |   |   |   BlockNode(5
|   |   |   |   |   DecNode( TypeNode(int) IDNode(t) = IntNode(0) )
|   |   |   |   |   AssNode( IDNode(t) = BinaryNode( IDNode(pi) * IDNode(f) ) )
|   |   |   |   )EndBlock5
|   |   |   )EndBlock4
|   |   )EndBlock3
|   |   EmptyStmtNode()
|   |   EmptyStmtNode()
|   |   EmptyStmtNode()
|   |   EmptyStmtNode()
|   |   EmptyStmtNode()
|   |   EmptyStmtNode()
|   |   AssNode( IDNode(t) = IntNode(1) )
```

2. Structed AST

```
1    {'name': 'Prog', '_children': [{'name': 'Dec', '_children': [{'name':
     'Type{int}'}, {'name': 'ID{a}'}, {'name': 'Int{0}'}]}, {'name':
     'Dec', '_children': [{'name': 'Type{long}'}, {'name': 'ID{l}'},
     {'name': 'Int{9999999}'}]}, {'name': 'Dec', '_children': [{'name':
     'Type{double}'}, {'name': 'ID{pi}'}, {'name': 'Float{3.141592}'}]},
     {'name': 'Dec', '_children': [{'name': 'Type{float}'}, {'name':
     'ID{f}'}, {'name': 'Float{0.03}'}]}, {'name': 'Dec', '_children':
     [{'name': 'Type{char}'}, {'name': 'ID{ch}'}, {'name': 'Char{a}'}]},
     {'name': 'Dec', '_children': [{'name': 'Type{char}'}, {'name':
```

```
'ID{b}'}, {'name': 'Char{1}'}]}, {'name': 'Dec', '_children':
[{'name': 'Type{char}'}, {'name': 'ID{e}'}, {'name': 'None'}]},
{'name': 'Dec', '_children': [{'name': 'Type{Type{char}}'}, {'name':
'ID{str}'}, {'name': 'None'}]}, {'name': 'Dec', '_children':
[{'name': 'Type{char}'}, {'name': 'ID{thisIsAnArray}'}, {'name':
'19'}, {'name': '20'}, {'name': '21'}, {'name': 'None'}]}, {'name':
'Func', '_children': [{'name': 'Type{int}'}, {'name': 'ID{main}'},
{'name': 'Block', '_children': [{'name': 'Dec', '_children':
[{'name': 'Type{double}'}, {'name': 'ID{pi2}'}, {'name': 'Binary{*}',
'_children': [{'name': 'ID{pi}'}, {'name': 'Int{2}'}]}]}, {'name':
'Dec', '_children': [{'name': 'Type{double}'}, {'name': 'ID{da}'},
{'name': '10'}, {'name': 'None'}]}, {'name': 'Dec', '_children':
[{'name': 'Type{int}'}, {'name': 'ID{i}'}, {'name': 'None'}]},
{'name': 'Dec', '_children': [{'name': 'Type{int}'}, {'name':
'ID{j}'}, {'name': 'None'}]}, {'name': 'Dec', '_children': [{'name':
'Type{int}'}, {'name': 'ID{t}'}, {'name': 'None'}]}, {'name': 'Dec',
'_children': [{'name': 'Type{int}'}, {'name': 'ID{controller}'},
{'name': 'Int{1}'}]}, {'name': 'Dec', '_children': [{'name':
'Type{int}'}, {'name': 'ID{condition}'}, {'name': 'Int{0}'}]},
{'name': 'Loop', '_children': [{'name': 'Dec', '_children': [{'name':
'Type{int}'}, {'name': 'ID{i}'}, {'name': 'Int{0}'}]}, {'name':
'Binary{<}', '_children': [{'name': 'ID{i}'}, {'name': 'Int{10}'}]},
{'name': 'Block{Ass{=}}', '_children': [{'name': 'ArrSub',
'_children': [{'name': 'ID{da}'}, {'name': 'ID{i}'}]}, {'name':
'Binary{*}', '_children': [{'name': 'ID{i}'}, {'name': 'ID{pi}'}]}]},
{'name': 'Ass{=}', '_children': [{'name': 'ID{i}'}, {'name':
'Binary{+}', '_children': [{'name': 'ID{i}'}, {'name':
'Int{1}'}]}]}]}, {'name': 'Loop', '_children': [{'name':
'EmptyStmt'}, {'name': 'Int{1}'}, {'name': 'Block{EmptyStmt}'},
{'name': 'EmptyStmt'}]}, {'name': 'Loop', '_children': [{'name':
'Block{EmptyStmt}'}, {'name': 'while'}, {'name': 'Int{0}'}, {'name':
'EmptyStmt'}]}, {'name': 'Loop', '_children': [{'name': 'EmptyExp'},
{'name': 'EmptyExp'}, {'name': 'Block{EmptyStmt}'}, {'name':
'EmptyExp'}]}, {'name': 'Loop', '_children': [{'name': 'EmptyStmt'},
{'name': 'Int{1}'}, {'name': 'Block{Loop}', '_children': [{'name':
'Dec', '_children': [{'name': 'Type{int}'}, {'name': 'ID{n}'},
{'name': 'Int{100}'}]}, {'name': 'Binary{≥}', '_children': [{'name':
'ID{n}'}, {'name': 'Int{0}'}]}, {'name': 'Block{Loop}', '_children':
[{'name': 'EmptyStmt'}, {'name': 'ID{controller}'}, {'name':
'Block{Loop}', '_children': [{'name': 'Block{Ass{=}}', '_children':
[{'name': 'ID{str}'}, {'name': 'String{inside while loop}'}]},
{'name': 'while'}, {'name': 'Binary{==}', '_children': [{'name':
'ID{condition}'}, {'name': 'String{OK}'}]}, {'name': 'EmptyStmt'}]},
{'name': 'EmptyStmt'}]}, {'name': 'Ass{=}', '_children': [{'name':
'ID{n}'}, {'name': 'Binary{-}', '_children': [{'name': 'ID{n}'},
{'name': 'Int{1}'}]}]}]}, {'name': 'EmptyStmt'}]}, {'name': 'Loop',
'_children': [{'name': 'Ass{=}', '_children': [{'name': 'ID{j}'},
{'name': 'Int{0}'}]}, {'name': 'Binary{<}', '_children': [{'name':
'ID{j}'}, {'name': 'Int{10}'}]}, {'name': 'Block{IfStmt}',
```

```
'_children': [{'name': 'Binary{<}', '_children': [{'name': 'ArrSub',
'_children': [{'name': 'ID{da}'}, {'name': 'ID{j}'}]}, {'name':
'Int{10}'}]}, {'name': 'Block{Continue}'}, {'name': 'Block',
'_children': [{'name': 'Call', '_children': [{'name':
'ID{PrintHello}'}]}, {'name': 'Break'}]}]}, {'name': 'Ass{=}',
'_children': [{'name': 'ID{j}'}, {'name': 'Binary{+}', '_children':
[{'name': 'ID{j}'}, {'name': 'Int{1}'}]}]}]}, {'name': 'IfStmt',
'_children': [{'name': 'Int{1}'}, {'name': 'Block{IfStmt}',
'_children': [{'name': 'Int{2}'}, {'name': 'Block{Ass{=}}',
'_children': [{'name': 'ID{l}'}, {'name': 'Int{222}'}]}, {'name':
'Block{Ass{=}}', '_children': [{'name': 'ID{l}'}, {'name':
'Int{223}'}]}]}]}, {'name': 'Block{IfStmt}', '_children': [{'name':
'Int{3}'}, {'name': 'Block{Ass{=}}', '_children': [{'name': 'ID{l}'},
{'name': 'Int{333}'}]}, {'name': 'Block{Ass{=}}', '_children':
[{'name': 'ID{l}'}, {'name': 'Int{444}'}]}]}]}, {'name': 'Block',
'_children': [{'name': 'Dec', '_children': [{'name': 'Type{double}'},
{'name': 'ID{t}'}, {'name': 'Float{6.06}'}]}, {'name': 'Block',
'_children': [{'name': 'Dec', '_children': [{'name': 'Type{char}'},
{'name': 'ID{t}'}, {'name': 'Char{0}'}]}, {'name': 'Block',
'_children': [{'name': 'Dec', '_children': [{'name': 'Type{int}'},
{'name': 'ID{t}'}, {'name': 'Int{0}'}]}, {'name': 'Ass{=}',
'_children': [{'name': 'ID{t}'}, {'name': 'Binary{*}', '_children':
[{'name': 'ID{pi}'}, {'name': 'ID{f}'}]}]}]}]}]}, {'name':
'EmptyStmt'}, {'name': 'EmptyStmt'}, {'name': 'EmptyStmt'}, {'name':
'EmptyStmt'}, {'name': 'EmptyStmt'}, {'name': 'EmptyStmt'}, {'name':
'Ass{=}', '_children': [{'name': 'ID{t}'}, {'name': 'Int{1}'}]},
{'name': 'Ass{=}', '_children': [{'name': 'ID{t}'}, {'name':
'Ass{=}', '_children': [{'name': 'ID{a}'}, {'name': 'Unary{-}
{ID{j}}'}]}]}, {'name': 'Ternary', '_children': [{'name': 'Int{1}'},
{'name': '?'}, {'name': 'Int{2}'}, {'name': ':'}, {'name':
'Int{3}'}]}, {'name': 'Ternary', '_children': [{'name': 'ID{t}'},
{'name': '?'}, {'name': 'Ternary', '_children': [{'name': 'Int{1}'},
{'name': '?'}, {'name': 'Int{2}'}, {'name': ':'}, {'name':
'Int{3}'}]}, {'name': ':'}, {'name': 'Int{4}'}]}, {'name': 'Ternary',
'_children': [{'name': 'Int{0}'}, {'name': '?'}, {'name': 'Int{2}'},
{'name': ':'}, {'name': 'Ternary', '_children': [{'name': 'Int{3}'},
{'name': '?'}, {'name': 'Ternary', '_children': [{'name': 'ID{t}'},
{'name': '?'}, {'name': 'Int{4}'}, {'name': ':'}, {'name':
'Int{5}'}]}, {'name': ':'}, {'name': 'Int{6}'}]}]}, {'name':
'Binary{||}', '_children': [{'name': 'Binary{&&}', '_children':
[{'name': 'Int{1}'}, {'name': 'Int{2}'}]}, {'name': 'Binary{&&}',
'_children': [{'name': 'Int{3}'}, {'name': 'Int{4}'}]}]}, {'name':
'Binary{||}', '_children': [{'name': 'Int{1}'}, {'name':
'Binary{&&}', '_children': [{'name': 'Binary{||}', '_children':
[{'name': 'ID{t}'}, {'name': 'Int{3}'}]}, {'name': 'Int{4}'}]}]},
{'name': 'Binary{^}', '_children': [{'name': 'ID{e}'}, {'name':
'ID{e}'}]}, {'name': 'Binary{|}', '_children': [{'name': 'Binary{|}',
'_children': [{'name': 'Binary{&}', '_children': [{'name': 'ID{t}'},
{'name': 'ID{e}'}]}, {'name': 'Binary{^}', '_children': [{'name':
```

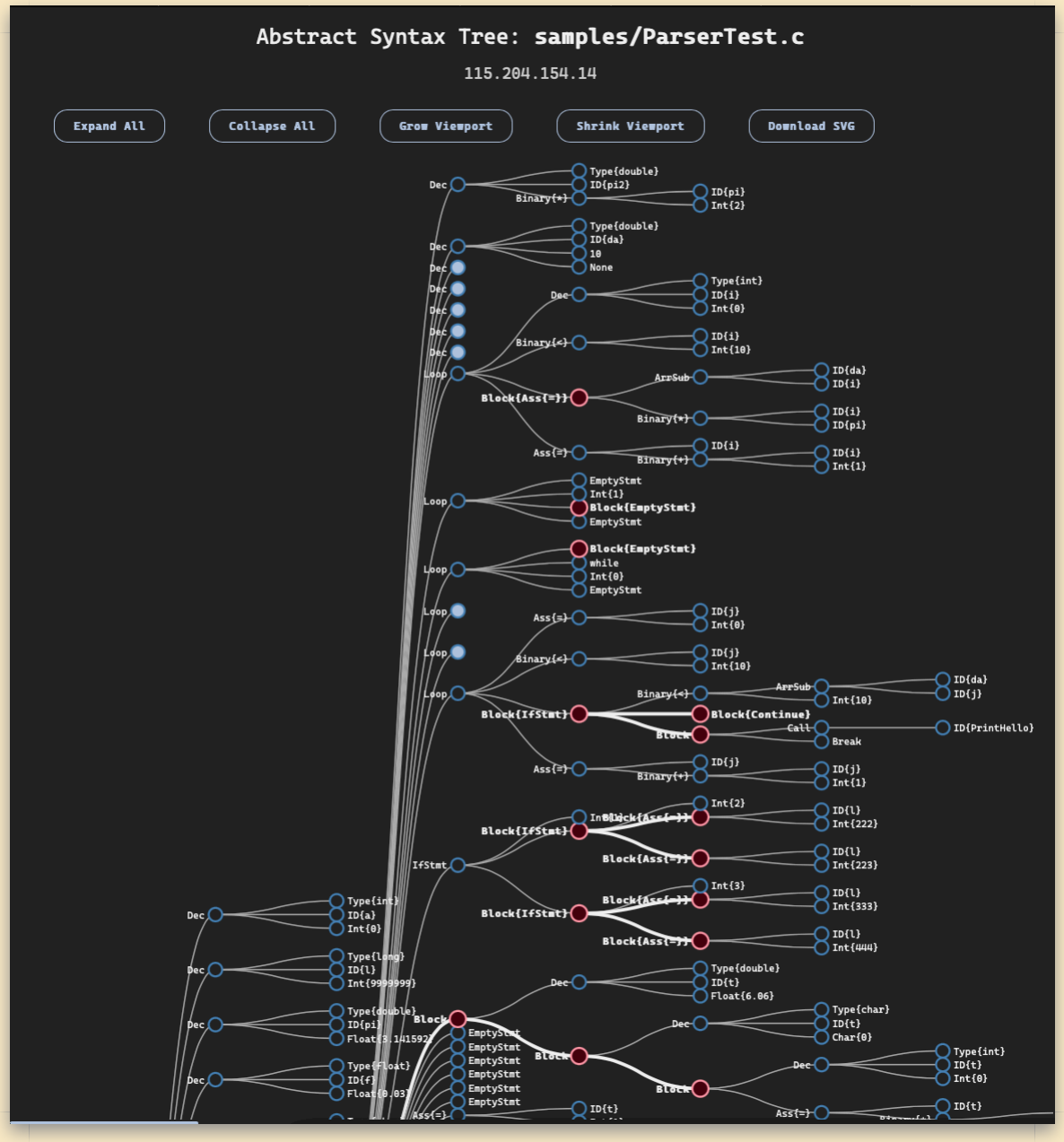
```
'Int{3}'}, {'name': 'Int{5}'}]}]}, {'name': 'Binary{|}', '_children':
[{'name': 'ID{j}'}, {'name': 'ID{e}'}]}]}, {'name': 'Binary{==}',
'_children': [{'name': 'ID{t}'}, {'name': 'Int{1}'}]}, {'name':
'Binary{≥}', '_children': [{'name': 'ID{e}'}, {'name': 'Char{a}'}]},
{'name': 'Binary{≠}', '_children': [{'name': 'Binary{==}',
'_children': [{'name': 'Int{1}'}, {'name': 'Binary{<}', '_children':
[{'name': 'Int{2}'}, {'name': 'Int{3}'}]}]}, {'name': 'Binary{≥}',
'_children': [{'name': 'Binary{>}', '_children': [{'name':
'Binary{≤}', '_children': [{'name': 'Int{4}'}, {'name': 'Int{5}'}]},
{'name': 'Int{6}'}]}, {'name': 'Int{7}'}]}]}, {'name': 'Binary{>>}',
'_children': [{'name': 'ID{a}'}, {'name': 'Int{10}'}]}, {'name':
'Binary{<<}', '_children': [{'name': 'Binary{>>}', '_children':
[{'name': 'Int{2}'}, {'name': 'Int{3}'}]}, {'name': 'Int{4}'}]},
{'name': 'Binary{+}', '_children': [{'name': 'ID{i}'}, {'name':
'Int{1}'}]}, {'name': 'Binary{+}', '_children': [{'name':
'Int{100}'}, {'name': 'Binary{/}', '_children': [{'name': 'Int{30}'},
{'name': 'Float{4.0}'}]}]}, {'name': 'Unary{~}{ID{a}}'}, {'name':
'Unary{-}{Int{100}}'}, {'name': 'Unary{Type{int}}{Unary{!}
{Binary{+}}}', '_children': [{'name': 'Unary{*}{Binary{+}}',
'_children': [{'name': 'Binary{+}', '_children': [{'name': 'Unary{&}
{ID{t}}'}, {'name': 'Unary{+}{Int{1}}'}]}, {'name': 'Unary{-}
{Int{1}}'}]}, {'name': 'Unary{*}{Ass{=}}', '_children': [{'name':
'ID{str}'}, {'name': 'Binary{+}', '_children': [{'name': 'ID{str}'},
{'name': 'Int{1}'}]}]}]}, {'name': 'Binary{||}', '_children':
[{'name': 'Int{1}'}, {'name': 'Binary{&&}', '_children': [{'name':
'Int{0}'}, {'name': 'Binary{==}', '_children': [{'name': 'Int{4}'},
{'name': 'Binary{≥}', '_children': [{'name': 'Int{3}'}, {'name':
'Binary{-}', '_children': [{'name': 'Int{2}'}, {'name': 'Binary{*}',
'_children': [{'name': 'Int{1}'}, {'name': 'Unary{-}
{Int{1}}'}]}]}]}]}]}]}]}]}]}, {'name': 'Ass{=}', '_children': [{'name':
'ID{e}'}, {'name': 'Ternary', '_children': [{'name': 'Binary{<}',
'_children': [{'name': 'ID{i}'}, {'name': 'ID{t}'}]}, {'name': '?'},
{'name': 'Ass{=}', '_children': [{'name': 'ID{a}'}, {'name':
'Binary{+}', '_children': [{'name': 'ID{a}'}, {'name': 'Int{1}'}]}]},
{'name': ':'}, {'name': 'Binary{&&}', '_children': [{'name':
'ID{a}'}, {'name': 'ID{t}'}]}]}]}, {'name': 'Ass{=}', '_children':
[{'name': 'ArrSub', '_children': [{'name': 'ID{da}'}, {'name':
'Int{2}'}]}, {'name': 'Binary{+}', '_children': [{'name': 'ArrSub',
'_children': [{'name': 'ID{da}'}, {'name': 'Int{2}'}]}, {'name':
'Int{1}'}]}]}, {'name': 'Ret{Int{0}}'}]}]}, {'name': 'Func',
'_children': [{'name': 'Type{void}'}, {'name': 'ID{PrintHello}'},
{'name': 'Block', '_children': [{'name': 'Call', '_children':
[{'name': 'ID{printf}'}, {'name': 'String{Hello, Nano C!}'}]},
{'name': 'Ret{EmptyExp}'}]}]}, {'name': 'Func', '_children':
[{'name': 'Type{int}'}, {'name': 'ID{max}'}, {'name': 'Param',
'_children': [{'name': 'Type{int}'}, {'name': 'ID{a}'}]}, {'name':
'Param', '_children': [{'name': 'Type{int}'}, {'name': 'ID{b}'}]},
{'name': 'Block{Ret{Ternary}}', '_children': [{'name': 'Binary{>}',
'_children': [{'name': 'ID{a}'}, {'name': 'ID{b}'}]}, {'name': '?'},
```

```
{'name': 'ID{a}'}, {'name': ':'}, {'name': 'ID{b}'}]}]}], 'size':
[2111.1111111111113, 911.1111111111111], 'filename':
'samples/ParserTest.c'}
```


This structed tree is in `json` format and is sent to our server to create a visualized and interactive AST on the webpage.

3. **Visualized AST** (partial)

   The visualized AST is too long to be screenshotted in a single page. You may see the complete version of the fancy-visualized AST in Introduction of Visitor.



Obtaining the above results with enough complexity, we carefully examined each one of the output. It turned out that **all nodes were parsed as desired**, with satisfying orders and precedence. **We can conclude that our parser passed the test.**

§7.3 IR Generation and Execution

Checkpoint 1

H3

```c
/**
 * checkpoint 1
 * feature:
 *      single function & directly return
 * expected output: 0
 */
int main() {
    return 0;
}
```

```
(pytorch) E:\0PROGRESS\CP\compiler>.\results\checkpoints\c1.exe

(pytorch) E:\0PROGRESS\CP\compiler>echo %errorlevel%
0
```

Checkpoint 2

```c
/**
 * checkpoint 2
 * feature:
 *      single function
 *      single type variable declarations
 * expected output: 0
 */
int main() {
    int a;
    int b;
    int c;
    return 0;
}
```

```
(pytorch) E:\0PROGRESS\CP\compiler>.\results\checkpoints\c2.exe

(pytorch) E:\0PROGRESS\CP\compiler>echo %errorlevel%
0
```

Checkpoint 3

```c
/**
 * checkpoint 3
 * feature:
 *      single function
 *      initilizers & declaration list
 *      assignment expressions
 * expected output: 6
 */
int main() {
    int a = 0, b = 1, c = 2, d = 3;
    a = b*c*d;
    return a;
}
```

```
(pytorch) E:\0PROGRESS\CP\compiler>.\results\checkpoints\c3.exe

(pytorch) E:\0PROGRESS\CP\compiler>echo %errorlevel%
6
```

Checkpoint 4

```c
/**
 * checkpoint 4
 * feature:
 *      single function
 *      initilizers & declaration list
 *      assignment expressions
 *      more complicated expression
 *      implicit type casting from int1 to int32
 * expected output: 3
 */
int main() {
    int a = 0, b = 1, c = 2, d = 3;
    a = ((b*c*d) && (a*d)) + (((c+b)/d) || a) + b*c;
    return a;
}
```

```
(pytorch) E:\0PROGRESS\CP\compiler>.\results\checkpoints\c4.exe

(pytorch) E:\0PROGRESS\CP\compiler>echo %errorlevel%
3
```

Checkpoint 5

```c
/**
 * checkpoint 5
 * feature:
 *      integer type
 *      multiple functions & function call
 * expected output: 2
 */
int two() { return 2; }
int main() {
    return two();
}
```

```
(pytorch) E:\0PROGRESS\CP\compiler>.\results\checkpoints\c5.exe

(pytorch) E:\0PROGRESS\CP\compiler>echo %errorlevel%
2
```

Checkpoint 6

```c
/**
 * checkpoint 6
 * feature:
 *      integer type
 *      multiple functions & function call
 *      if-else statements
 *      loop statements
 * expected output: 20
 */
int sum_up_to(int a) {
    int sum = 0;
    for (int i=0;i<a;i=i+1)
        sum = sum + i;
    return sum;
}
int main() {
    int a = 5;
    if (a % 2) return 2*sum_up_to(a);
    else return sum_up_to(a);
}
```

```
(pytorch) E:\0PROGRESS\CP\compiler>.\results\checkpoints\c6.exe

(pytorch) E:\0PROGRESS\CP\compiler>echo %errorlevel%
20
```

Checkpoint 7

```c
/**
 * checkpoint 7
 * feature:
 *      integer type
 *      multiple functions & function call
 *      nested if-else statements
 *      recursion functions
 * expected output: 8
 */
int fib(int n) {
    if (n ≤ 0) return 0;
    else if (n == 1) return 1;
    else if (n == 2) return 1;
    else return fib(n - 1) + fib(n - 2);
}
int main() {
    int a = 6;
    return fib(a);
}
```

```
(pytorch) E:\0PROGRESS\CP\compiler>.\results\checkpoints\c7.exe

(pytorch) E:\0PROGRESS\CP\compiler>echo %errorlevel%
8
```

Checkpoint 8

```c
/**
 * checkpoint 8
 * feature:
 *      integer type
 *      multiple functions & function call
 *      nested if-else statements
 *      recursion functions
 *      nested loop statements
 *      break & continue
 * expected output: 595
 */
int fib(int n) {
    // 1 1 2 3 5 8 13 21 34
    if (n ≤ 0) return 0;
    else if (n == 1) return 1;
    else if (n == 2) return 1;
    else return fib(n - 1) + fib(n - 2);
```

```c
}
int main() {
    int sum = 0;
    for (int a = 4; a < 10; a++) {
        if (a == 6) continue;
        else if (fib(a) % 2 == 0) {
            for (int b = fib(a); b > 0; b=b-1)
                sum = sum + b;
            break;
        }
    }
    return sum;
}
```

```
(pytorch) E:\0PROGRESS\CP\compiler>.\results\checkpoints\c8.exe

(pytorch) E:\0PROGRESS\CP\compiler>echo %errorlevel%
595
```

Checkpoint 9

```c
/**
 * checkpoint 9
 * feature:
 *      void return type
 *      integer type & pointer type
 *      & and * operator
 *      value swap with pointers
 * expected output: 2
 */
void swap(int *a, int *b) {
    int c = *a;
    *a = *b;
    *b = c;
}
int main() {
    int a=1, b=2;
    int *c = &a;
    int *d = &b;
    swap(c,d);
    return a;
}
```

```
(pytorch) E:\0PROGRESS\CP\compiler>.\results\checkpoints\c9.exe

(pytorch) E:\0PROGRESS\CP\compiler>echo %errorlevel%
2
```

Checkpoint 10

```c
/**
 * checkpoint 10
 * feature:
 *      integer type & pointer type
 *      array type
 *      & and * operator
 *      value swap with pointers
 * expected output: 7
 */
void swap(int* a, int* b) {
    int c = *a;
    *a = *b;
    *b = c;
}
int main() {
    int a[3][3];
    a[0][0] = 0; a[0][1] = 1; a[0][2] = 2;
    a[1][0] = 3; a[1][1] = 4; a[1][2] = 5;
    a[2][0] = 6; a[2][1] = 7; a[2][2] = 8;
    int b=1, c=2;
    int *ptr_to_b = &b;
    int *ptr_to_c = &c;
    swap(ptr_to_b,ptr_to_c);
    return a[b][c];
}
```

```
(pytorch) E:\0PROGRESS\CP\compiler>.\results\checkpoints\c10.exe

(pytorch) E:\0PROGRESS\CP\compiler>echo %errorlevel%
7
```

Checkpoint 11

```c
/**
 * checkpoint 11
 * feature:
 *      integer type & float type & void type
 *      pointer type & array type
 *      & and * operator
```

```
 *      type casting:
 *          xd array -> pointer
 *          int <-> float
 *      gloabl variables
 *      multi-scopes
 *      calculations:
 *          *(pointer + integer)
 * expected output: 12
 */

int n = 10;
int a[10][10];

int main() {
    int i = 3, j = 3;
    for (int i=0; i<n;i=i+1) {
        for (int j=0; j<n; j=j+1) {
            a[i][j] = i+j;
        }
    }
    int * arr_ptr = (int*)a;
    *(arr_ptr + i*n + j) = 2 * *(arr_ptr + i*n + j);
    return a[i][j];
}
```

```
(pytorch) E:\0PROGRESS\CP\compiler>.\results\checkpoints\c11.exe

(pytorch) E:\0PROGRESS\CP\compiler>echo %errorlevel%
12
```

Checkpoint 12

```
/**
 * checkpoint 12
 * feature:
 *      integer type & float type & void type
 *      pointer type & array type
 *      & and * operator
 *      type casting:
 *          xd array -> pointer
 *          int <-> float
 *      calculations:
 *          *(pointer + integer)
 *      quicksort
 *      random integer generation
 * expected output: 1
 */
```

```
16
17  int qsort(int * a, int l, int r)
18  {
19      int i = l;
20      int j = r;
21      int p = *(a + ((l + r)/2));
22      int flag = 1;
23      while (i ≤ j) {
24          while (*(a+i) < p) i = i + 1;
25          while (*(a+j) > p) j = j - 1;
26          if (i > j) break;
27          int u = *(a+i);
28          *(a+i) = *(a+j);
29          *(a+j) = u;
30          i = i + 1;
31          j = j - 1;
32      }
33      if (i < r) qsort(a, i, r);
34      if (j > l) qsort(a, l, j);
35      return 0;
36  }
37
38  // random floating point number distributed uniformly in [0,1]
39  float rand(float *r) {
40      float base=256.0;
41      float a=17.0;
42      float b=139.0;
43      float temp1=a*(*r)+b;
44      float temp2=(float)(int)(temp1/base);
45      float temp3=temp1-temp2*base;
46      *r=temp3;
47      float p=*r/base;
48      return p;
49  }
50
51  int initArr(int* a, int n)
52  {
53      float state = 114514.0;
54      int i =0;
55      while (i < n) {
56          *(a + i) = (int)(255*rand(&state));
57          i = i + 1;
58      }
59  }
60
61  int isSorted(int *a, int n)
62  {
63      int i = 0;
64      while (i < n - 1) {
```

```
65          if ( (*(a+i)) > (*(a+i+1)))
66              return 0;
67          i = i + 1;
68      }
69      return 1;
70  }
71
72  int main()
73  {
74      int n = 100;
75      int arr[100];
76      int * a = (int*)arr;
77      initArr(a, n);
78      qsort(a, 0, n - 1);
79      return isSorted(a, n);
80  }
```

```
(pytorch) E:\0PROGRESS\CP\compiler>.\results\checkpoints\c12.exe

(pytorch) E:\0PROGRESS\CP\compiler>echo %errorlevel%
1
```

Checkpoint 13

```
1   /**
2    * checkpoint 13
3    * feature:
4    *      integer type & float type & void type
5    *      pointer type & array type
6    *      & and * operator
7    *      type casting:
8    *          xd array -> pointer
9    *          int <-> float
10   *      calculations:
11   *          *(pointer + integer)
12   *      multiplication of matrix
13   * expected output: 0
14   */
15
16  int mulMatrix(int n, int *a, int *b, int *c) {
17      int i; int j; int k;
18      i = 0;
19      while (i < n) {
20          j = 0;
21          while (j < n) {
22              *(c + i*n + j) = 0;
23              k = 0;
```

```
            while (k < n) {
                int old = *(c + i*n + j);
                *(c + i*n + j) = old + *(a+i*n + k) * (*(b+k*n + j));
                k = k + 1;
            }
            j = j + 1;
        }
        i = i + 1;
    }
}

int initMatrix(int n, int *a) {
int i; int j; int k;
    k = 0;
    i = 0;
    while (i < 2) {
        j = 0;
        while (j < 2) {
            k = k + 1;
            *(a + i*n + j) = k;
            j = j + 1;
        }
        i = i + 1;
    }
}

int a[2][2]; int b[2][2]; int c[2][2];
int main() {
    initMatrix(2, (int*)a);
    mulMatrix(2, (int*)a, (int*)a, (int*)b);
    mulMatrix(2, (int*)b, (int*)b, (int*)c);
    if (c[0][0] ≠ 199)
        return 1;
    if (c[0][1] ≠ 290)
        return 2;
    if (c[1][0] ≠ 435)
        return 3;
    if (c[1][1] ≠ 634)
        return 4;
    return 0;
}
```

```
(pytorch) E:\0PROGRESS\CP\compiler>.\results\checkpoints\c13.exe

(pytorch) E:\0PROGRESS\CP\compiler>echo %errorlevel%
0
```

Checkpoint 14

```c

/**
 * checkpoint 13
 * feature:
 *      dijkstar shortest path algorithm
 * expected output: 17
 */

int main(void)
{
    int e[10][10], dis[10], book[10], i, j, m, n, t1, t2, t3, u, v, min;
    n = 6;
    m = 9;
    int inf = 99999;
    for (i = 1; i ≤ n; i++)
        for (j = 1; j ≤ n; j++)
            e[i][j] = inf;
    e[1][2] = 1;
    e[1][3] = 12;
    e[2][3] = 9;
    e[2][4] = 3;
    e[3][5] = 5;
    e[4][3] = 4;
    e[4][5] = 13;
    e[4][6] = 15;
    e[5][6] = 4;
    for (i = 1; i ≤ n; i++)
        dis[i] = e[1][i];  //初始化dis数组，表示1号顶点到其他顶点的距离
    for (i = 1; i ≤ n; i++)
        book[i] = 0;
    book[i] = 1;  //记录当前已知第一个顶点的最短路径
    for (i = 1; i ≤ n - 1; i++)
        for (i = 1; i ≤ n - 1; i++) {  //找到离一号顶点最近的点
            min = inf;
            for (j = 1; j ≤ n; j++) {
                if (book[j] == 0 && dis[j] < min) {
                    min = dis[j];
                    u = j;
                }
            }
            book[u] = 1;  //记录当前已知离第一个顶点最近的顶点
            for (v = 1; v ≤ n; v++) {
                if (e[u][v] < inf) {
                    if (dis[v] > dis[u] + e[u][v])
                        dis[v] = dis[u] + e[u][v];
                }
            }
        }
```

```
49        //0 1 8 4 13 17
50        return dis[6];
51    }
```

```
(pytorch) E:\0PROGRESS\CP\compiler>.\results\checkpoints\c14.exe

(pytorch) E:\0PROGRESS\CP\compiler>echo %errorlevel%
17
```