西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY

Compiler Design: NGEBSKI Programming Language

Professor: 林奕

.....

Designed by:

Cecillia Alexandra Badaruddin (2021380098)

Rheina Trudy Williams (2021380101)

Table of Contents

I. Introduction

a) Background

The Ngebski programming language is inspired by the Indonesian word ‘ngeb’, which is a friendly way to address people in Indonesia. This aligns with the motivation to create a simple user friendly language. The language has similarity to C and Python syntax with necessary language features for a basic compiler.

b) Goal

The main goal of the Ngebski compiler project is to create an efficient compiler for a custom language named Ngebski language.

Other objectives include,

- understanding the fundamental knowledge of compiler design, break down source code into tokens using a lexer, construct a parser that can generate parse trees or abstract syntax trees (AST) from tokens, implement semantic checks to ensure the program adheres to the language rules, generate intermediate code, and then translate that into machine code or an executable.
- implement essential language features like variable declarations, arithmetic operations and boolean operations.
- use LLVM for backend code generation to produce optimised machine code.
- develop extensive test cases to ensure the compiler handles various language constructs correctly.

c) Features

- **Language Constructs**: Support for assign statements, arithmetic expressions, if-then-else statements, while loops, and boolean expressions integer type support, intermediate code production , string output, and boolean operations.
- **Data Types**: Support for data types, such as integers and floats.
- **Code Generation**: Generation of IR code.
- **Operating system:** Supports Apple macOS and Microsoft Windows.
- **Testing**:
    1. string.ngeb
    2. arithmetic1.ngeb
    3. arithmetic2.ngeb
    4. boolean.ngeb
    5. conditional.ngeb
    6. IfThenElse.ngeb
    7. incDec.ngeb
    8. integration.ngeb
    9. variable.ngeb

10. while.ngeb

d) Prerequisites

Environment:

Python 3.12

Visual Studio Code

MicrosoftWindows users need to add ___chkstk_ms to the OS standard library

Library:

rply

llvm

II. Description of the Language:

a) BNF Grammar

- program ::= statement_list
- statement_list ::= statement | statement_list statement
- statement ::= assign | print | if_statement | while_statement
- assign ::= ID "=" expression ";" | ID "=" ID "++" ";" | ID "=" ID "--" ";"
- print ::= "ngeb" "(" expression ")" "ski"
- if_statement ::= "if" condition "then" statement_list "else" statement_list "end"
- while_statement ::= "while" condition "do" statement_list "end"
- condition ::= expression comp_operator expression | boolean_expression
- comp_operator ::= "==" | "!=" | "<" | ">" | "<=" | ">="
- boolean_expression ::= boolean_expression "&&" boolean_term | boolean_expression "||" boolean_term | boolean_term
- boolean_term ::= "!" boolean_term | term
- expression ::= expression "+" term | expression "-" term | term
- term ::= term "*" factor | term "/" factor | factor
- factor ::= INTEGER | FLOAT | LONG | SHORT | ID | "(" expression ")" | "true" | "false" | STRING_LITERAL
- ID ::= letter id_tail*
- letter ::= "a" | "b" | ... | "z" | "A" | "B" | ... | "Z" | "_"
- id_tail ::= letter | digit | "_"
- digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

b) Lexical model

- Control Flow:

  IF for "if" statements.

THEN for "then" statements.

ELSE for "else" statements.

END for "end" statements.

WHILE for "while" loops.

DO for "do" statements.

- Print:

  PRINT for the "ngeb" print function.

- Data Types

  FLOAT for floating-point numbers (e.g., -123.45).

  LONG for long integers (e.g., 1234567890).

  INTEGER for integers (e.g., 123).

  SHORT for short integers (e.g., 123).

- Boolean Literals

  TRUE for the boolean literal true.

  FALSE for the boolean literal false.

- Boolean Operators

  AND for logical and (and).

  OR for logical or (or).

  NOT for logical not (not).

- Comparison Operators

  EQ for equality (==).

  NEQ for inequality (!=).

  LEQ for less than or equal (<=).

GEQ for greater than or equal (>=).

LT for less than (<).

GT for greater than (>).

- Parentheses

  OPEN_PAREN for opening parenthesis (().

  CLOSE_PAREN for closing parenthesis ()).

- Symbols

  SEMI_COLON for the statement terminator (ski).

- Increment and Decrement

  INCREMENT for incrementing (++).

  DECREMENT for decrementing (--).

- Arithmetic Operators

  SUM for addition (+).

  SUB for subtraction (-).

  MUL for multiplication (*).

  DIV for division (/).

- Identifier

  ID for identifiers, which include variable names (e.g., my_var).

- Assignment

  EQUALS for assignment operator (=).

- String Literal

  STRING_LITERAL for string values (e.g., "hello ngeb").

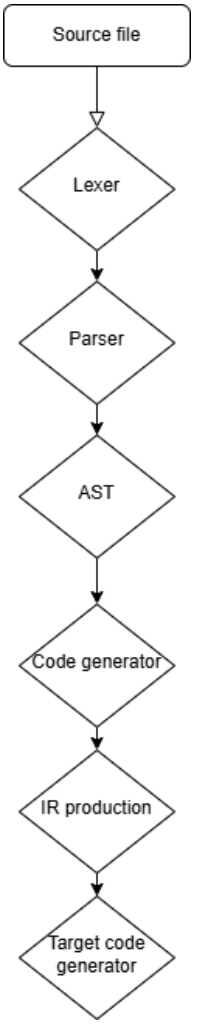
- Ignored Characters

Whitespace: The lexer ignores spaces, tabs, and other whitespace characters.

III. Design and Implementation:

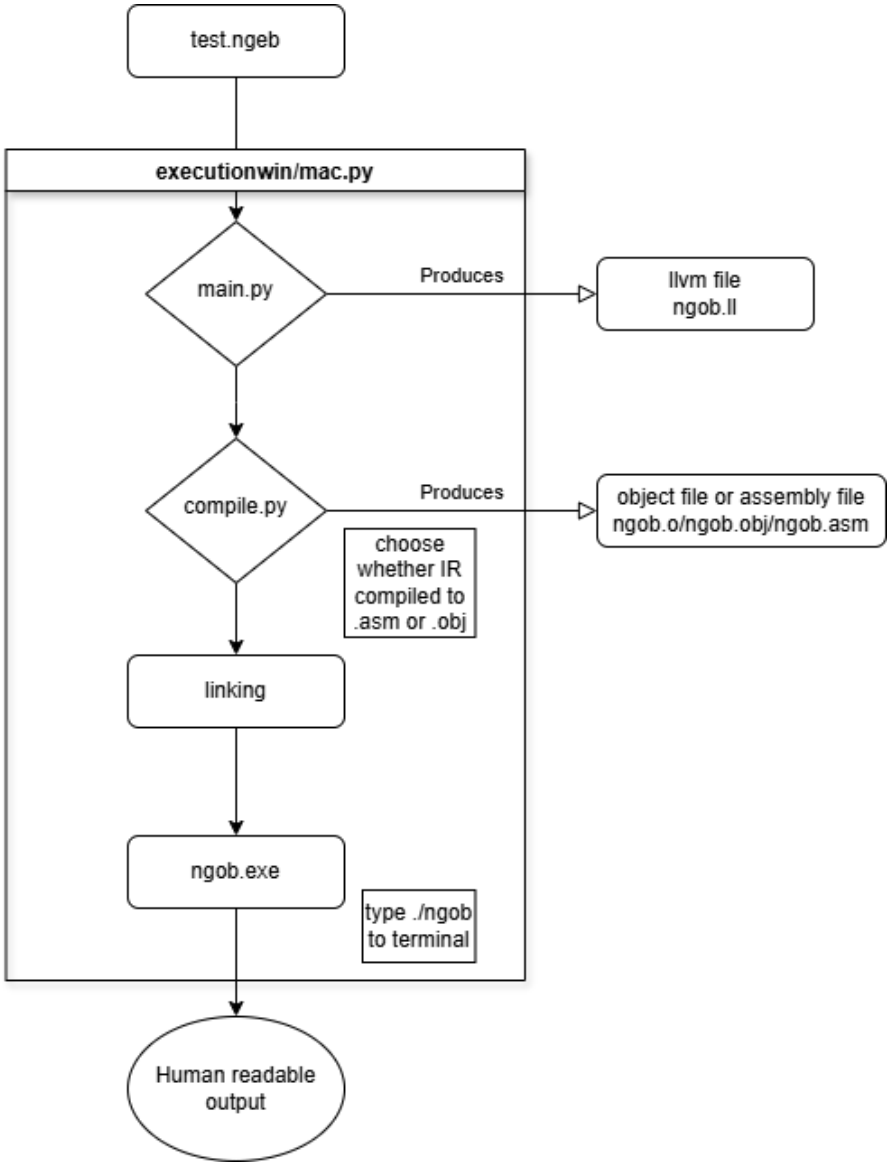
a) Compiler Architecture

Architecture:



The overall architecture of the compiler has lexer as the token scanner, parser for syntax check, AST(Abstract Syntax Tree) to contain the functions for parser and create the LLVM AST, code generator to transform machine code into intermediate representation (IR), and target code generator for compiling the IR code into either assembly or object file.

The compiler architecture is designed to be compatible with Microsoft Windows and Apple macOS. The program has two versions of target code generator that can be run based on the target OS.



1. main.py is used as the body of the compiler for lexer, parser, AST and code generator for intermediate representation (IR file).
2. compile.py is used for object and assembly file generators.
3. ngob.exe is the executable of the machine code.
4. ./ngob is used to execute the file into human readable output.

b) Lexer (lexx.py)

1. Define a class Lexx:

- Initialize the class with a LexerGenerator object.

2. Inside Lexx class:

- Define a method _add_tokens():

- Add tokens to the lexer using self.lexer.add().
- Tokens are added in the form (token_name, regex_pattern).
- Each token represents a lexical element of the programming language.

3. Tokens added include:
   - Keywords and control flow: IF, THEN, ELSE, END, WHILE, DO.
   - Number types: FLOAT, LONG, INTEGER, SHORT.
   - Boolean literals: TRUE, FALSE.
   - Boolean operators: AND, OR, NOT.
   - Print statement: PRINT.
   - Comparison operators: EQ (==), NEQ (!=), LEQ (<=), GEQ (>=), LT (<), GT (>).
   - Parentheses: OPEN_PAREN, CLOSE_PAREN.
   - Semicolon: SEMI_COLON.
   - Increment and decrement: INCREMENT (++), DECREMENT (--).
   - Arithmetic operators: SUM (+), SUB (-), MUL (*), DIV (/).
   - Identifier (variable names): ID.
   - Assignment operator: EQUALS (=).
   - String literal: STRING_LITERAL.
   - Ignore whitespace characters using self.lexer.ignore().
4. Define a method get_lexer():
   - Call _add_tokens() to populate the lexer with defined tokens.
   - Build and return the lexer using self.lexer.build().
5. End of Lexx class definition.
6. Instantiate Lexx class to create a lexer object.
7. Use get_lexer() method to obtain the lexer for tokenizing input code.
8. The lexer created can tokenize input code according to the defined tokens, identifying keywords, numbers, operators, identifiers, and literals, while ignoring whitespace.

c) Parser (parser.py)

- Import necessary classes from libraries: ParserGenerator and AST nodes (Number, Sum, Sub, Mul, Div, Print, Assign, Variable, Condition, IfThenElse, While, Boolean, BooleanOp, BooleanNot, Inc, Dec, StringLiteral).
- Define a class Parser:

  Constructor `__init__(self, module, builder, printf)`:

  Initialize a ParserGenerator `pg` with predefined tokens and operator precedence.

  Store references to `module`, `builder`, `printf`, and initialize an empty `variables` dictionary.

- Method `parse(self)`:

  - `program : statement_list`: Defines the top-level production for the program, consisting of a list of statements.

  - `statement_list : statement_list statement` and `statement_list : statement`: Handles recursive lists of statements within the program.

  - `statement : PRINT OPEN_PAREN expression CLOSE_PAREN SEMI_COLON`: Handles the statement for printing expressions.

  - `statement : assign`, `statement : if_statement`, `statement : while_statement`, `statement : expression SEMI_COLON`: Handles different types of statements including assignments, if-then-else, while loops, and standalone expressions.

  - `assign : ID EQUALS expression SEMI_COLON`, `assign : ID EQUALS boolean_expression SEMI_COLON`, `assign : ID EQUALS ID INCREMENT SEMI_COLON`, `assign : ID EQUALS ID DECREMENT SEMI_COLON`: Handles various forms of assignment statements.

  - `if_statement : IF condition THEN statement_list optional_else END`: Handles if-then-else statements including optional else parts.

  - `optional_else : ELSE statement_list`, `optional_else : `: Handles the optional else part of if-then-else statements.

  - `while_statement : WHILE condition DO statement_list END`: Handles while loop statements.

  - `condition : boolean_expression`: Handles conditions used in if and while statements.

  - `boolean_expression : boolean_expression AND boolean_expression`, `boolean_expression : boolean_expression OR boolean_expression`, `boolean_expression : comparison`, `boolean_expression : term`: Handles boolean expressions and their precedence.

- `comparison : expression EQ expression`, `comparison : expression NEQ expression`, `comparison : expression LT expression`, `comparison : expression GT expression`, `comparison : expression LEQ expression`, `comparison : expression GEQ expression`: Handles comparison operations between expressions.

- `expression : expression SUM term`, `expression : expression SUB term`, `expression : term`: Handles arithmetic expressions including addition, subtraction, and unary operators.

- `expression : boolean_expression`, `expression : NOT expression`, `expression : SUB term`, `expression : SUM term`: Handles boolean expressions and their unary operators.

- `term : term MUL factor`, `term : term DIV factor`, `term : factor`: Handles multiplication and division operations within expressions.

- `term : factor`: Handles factors such as literals, identifiers, and parenthesized expressions within expressions.

- `factor : INTEGER`, `factor : FLOAT`, `factor : LONG`, `factor : SHORT`, `factor : ID`, `factor : OPEN_PAREN expression CLOSE_PAREN`, `factor : TRUE`, `factor : FALSE`, `factor : STRING_LITERAL`: Handles different types of factors that can appear in expressions.

- `error_handle(token)`: Handles syntax errors during parsing and error reporting.

- Method `get_parser(self)`:

Build and return the parser using `self.pg.build()`.

d) AST (astree.py)

- Number: Represents numeric literals in the AST, evaluating to LLVM IR constants based on their type (integer or float).
- StringLiteral: Handles string literals, converting them to LLVM IR global variables with appropriate formatting and handling null termination.
- BinaryOp: Base class for binary operations (Sum, Sub, Mul, Div), ensuring operands are converted to the same type before performing operations.
- Sum: Performs addition operation between two operands, handling type conversions and generating LLVM IR.
- Sub: Performs subtraction operation between two operands, handling type conversions and generating LLVM IR.
- Mul: Performs multiplication operation between two operands, handling type conversions and generating LLVM IR.
- Div: Performs division operation between two operands, handling type conversions and generating LLVM IR.

- UnaryOp: Base class for unary operations (Inc, Dec), allowing increment and decrement operations on variables, updating LLVM IR accordingly.
- Inc: Represents the increment operation, evaluating to LLVM IR instructions for adding 1 to a variable.
- Dec: Represents the decrement operation, evaluating to LLVM IR instructions for subtracting 1 from a variable.
- Variable: Represents variables in the AST, handling allocation, storage, and loading of values in LLVM IR.
- Assign: Handles assignment operations, storing evaluated values into variables.
- Print: Represents print statements, formatting and printing values based on their type (integer, float, string) using LLVM IR.
- Condition: Handles conditional expressions (IfThenElse), evaluating relational operations and generating LLVM IR for comparisons.
- IfThenElse: Represents if-then-else statements in the AST, branching based on evaluated conditions and executing appropriate blocks.
- Boolean: Represents boolean literals (True, False), evaluating to LLVM IR constants for boolean values.
- BooleanOp: Represents boolean operations (AND, OR), performing logical operations on boolean values and generating LLVM IR.
- BooleanNot: Represents the logical NOT operation (NOT), evaluating to LLVM IR instructions for negating boolean values.
- While: Represents while loop constructs in the AST, evaluating loop conditions and executing loop bodies repeatedly in LLVM IR.

e) Intermediate Code Generation (CodeGen.py)

- init(self): Initializes the LLVM environment and prepares for code generation. It initializes LLVM bindings, targets, and printers, configures LLVM for the current platform, creates a main function in LLVM IR, and sets up an IR builder for generating LLVM instructions.
- _config_llvm(self): Configures the LLVM module by setting its name, triple (target architecture), and creating a base function main with no parameters and void return type. It also initializes an entry basic block and an IR builder associated with it.
- _create_execution_engine(self): Creates an execution engine for LLVM JIT compilation. It retrieves the target machine based on the module's triple, parses an empty assembly module, and initializes an MCJIT compiler with the target machine.
- _declare_print_function(self): Declares the printf function in LLVM IR. It defines the function signature (printf_ty) accepting a void pointer (voidptr_ty) as an argument with variable arguments (var_arg=True). The function is added to the LLVM module as printf.
- _compile_ir(self): Compiles the LLVM IR code generated so far. It finalizes the main function by appending a return void instruction, verifies the module's integrity, adds it to the execution engine, runs static constructors, and returns the compiled module.

- create_ir(self): Calls _compile_ir() to compile the LLVM IR code and returns the resulting module.
- save_ir(self, filename): Saves the LLVM IR code to a specified file. It converts the LLVM module to a string representation and writes it to the output file.
- _declare_global_string(self, name, value): Declares a global string variable in LLVM IR. It converts the string value to UTF-8 bytes, creates an LLVM constant array (str_const), declares a global variable (global_var) with internal linkage and constant initializer, and returns the global variable object.

f) main.py

- Importing Modules: Imports necessary modules (Lexx, Parser, CodeGen) for lexical analysis, parsing, and code generation.
- Reading Input: Prompts the user to enter an input file name (assumed to be a .ngeb file), reads its contents, and stores it in text_input.
- Lexical Analysis: Initializes a lexer from Lexx and performs lexical analysis on text_input, generating tokens.
- Token List Creation: Iterates through the tokens generated by the lexer, printing each token for debugging purposes, and adds each token to token_list. Handles any exceptions that may occur during tokenization.
- Code Generation: Initializes a CodeGen object (codegen) for generating LLVM IR code.
- Parsing: Initializes a Parser (pg) with references to module, builder, and printf from the CodeGen object. Calls parse() method to define production rules and get_parser() method to obtain the parser. Parses the token_list to obtain parsed_program, which contains parsed statements.
- Evaluation: Iterates through parsed_program and calls eval() on each statement to evaluate and generate LLVM IR code.
- IR Code Compilation and Saving: Calls create_ir() on codegen to compile the generated LLVM IR code and save_ir() to save it to a file named "ngob.ll".
- Printing Target Triple: Prints the target triple (module.triple) set in the CodeGen object to ensure it is correctly configured.

g) compile.py

- Compiles LLVM IR code (ir_filename) into either an object file (filetype='obj') or an assembly file (filetype='asm'), and optionally links it to produce an executable.
- Error Handling: Checks if the IR file exists and validates the filetype. Prints errors if the file doesn't exist or if the filetype is unsupported.
- LLC Command: Constructs an LLVM LLC command (llc_command) based on the filetype and executes it using subprocess.run(). Handles errors using subprocess.CalledProcessError.
- GCC Command (if obj): If filetype is 'obj', constructs a GCC command (gcc_command) to link the object file and create the executable. Executes it using subprocess.run() and handles errors similarly.
- Output: Prints relevant messages during the process, indicating successful generation of the assembly file or the executable.

- Main Execution: If executed as a script (__name__ == "__main__"), prompts the user to enter the desired output file type (obj or asm). Calls compile_to_executable with predefined filenames (ir_filename, output_filename) and the user-provided filetype. Prints a message if compilation fails.

h) execution.py

This file encapsulates main.py, compile.py and file execution running for easier run and compile.

IV. Test Cases:

a) Compiling test

The test case was used to ensure the compiler was able to detect the basic tokens and grammars.

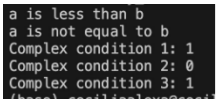
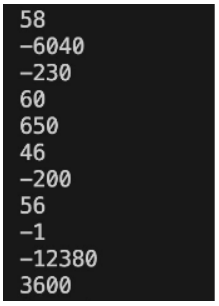
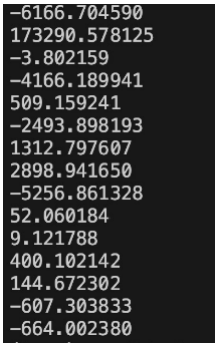
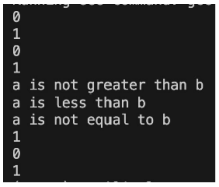
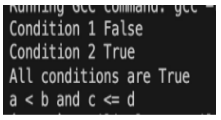
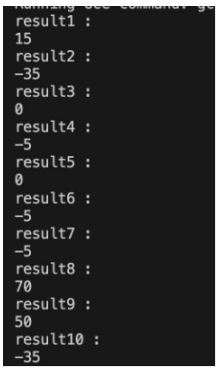
test.py : ensuring the tokens are detected

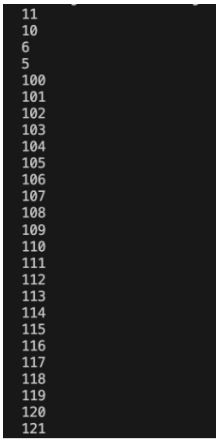
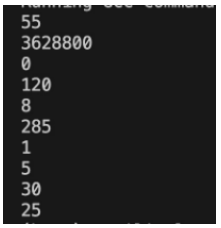
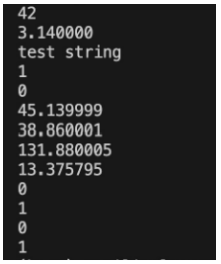
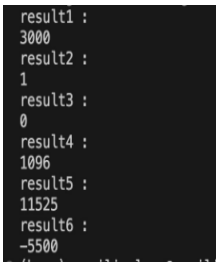
```
from lexx import Lexx

text_input = """

ngeb(5 + 5)ski

"""

lexx = Lexx().get_lexer()

tokens = lexx.lex(text_input)

for token in tokens:

    print(token)
```

b) Language feature test

The test cases used to ensure the integrated language features are correct and can give expected output.

Test case type	Description	Test step	Expected result	Status

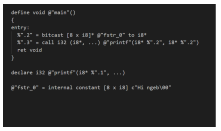
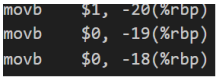
string.ngeb	Printing string, characters and sentences.	test/string		Pass
arithmetic1.ngeb	Variable usage for arithmetic expressions with complex multiplications, divisions, sums and subs in integers.	test/arithmetic1		Pass
arithmetic2.ngeb	Variable usage for arithmetic expressions with complex multiplications, divisions, sums and subs in floats.	test/arithmetic2		Pass
boolean.ngeb	Implementation of boolean expressions and operators.	test/boolean		Pass
conditional.ngeb	Implementation of If, else. then with boolean expressions and variables.	test/conditional		Pass
IfThenElse.ngeb	If then else usage with simple variables.	test/IfThenElse		Pass

incDec.ngeb	Increment and decrement usage	test/incDec		Pass
integration.ngeb	Complex case consists of assigning statements, variables. arithmetic, data types. conditionals and loops.	test/integration		Pass
variable.ngeb	Variable usage in numerical data and strings.	test/variable		Pass
while.ngeb	While loop implementation	test/while		Pass

string.ngeb:

```
Condition 1 False
Condition 2 True
All conditions are True
Result of x_bool and y_bool: 0
Result of x_bool or y_bool: 1
Result of not x_bool: 0
Result of not y_bool: a is not greater than b
a is less than b
a is not equal to b
Complex condition 1: 1
Complex condition 2: 0
Complex condition 3: 1
```

Additional feature:

Feature type	Description	Test step	Expected result	Status
IR code generator	Generate intermediate representational language llvm (ngob.ll)	run main.py		Pass
Assembly code generator	Generate assembly code with .s and .asm extension (ngob.s/ngob.asm)	run compile.py, type asm		Pass

Sample of IR code generated:

```
; ModuleID = "C:\Users\Rheina Trudy\Documents\UNI\SEMESTER 6\Compiler exp\selesai\jam3\CodeGen.py"
target triple = "x86_64-pc-windows-msvc"
target datalayout = ""

define void @"main"()
{
entry:
  %".2" = bitcast [8 x i8]* @"fstr_0" to i8*
  %".3" = call i32 (i8*, ...) @"printf"(i8* %".2", i8* %".2")
  ret void
}

declare i32 @"printf"(i8* %".1", ...)

@"fstr_0" = internal constant [8 x i8] c"Hi ngeb\00"
```

Sample of ASM code generatyed:

ngob.s

```
1       .text
2       .def    @feat.00;
3       .scl    3;
4       .type   0;
5       .endef
6       .globl  @feat.00
7   .set @feat.00, 0
8       .file   "ngob.ll"
9       .def    main;
10      .scl    2;
11      .type   32;
12      .endef
13      .globl  main                            # -- Begin function main
14      .p2align    4, 0x90
15  main:                                   # @main
16  .seh_proc main
17  # %bb.0:                                # %entry
18      pushq   %rbp
19      .seh_pushreg %rbp
20      pushq   %rsi
21      .seh_pushreg %rsi
22      pushq   %rdi
23      .seh_pushreg %rdi
24      subq    $32, %rsp
25      .seh_stackalloc 32
26      leaq    32(%rsp), %rbp
27      .seh_setframe %rbp, 32
28      .seh_endprologue
29      movb    $1, -20(%rbp)
30      movb    $0, -19(%rbp)
31      movb    $0, -18(%rbp)
32      movb    $1, -5(%rbp)
33      movb    $0, -4(%rbp)
34      movb    $1, -3(%rbp)
35      leaq    fstr_0(%rip), %rcx
36      subq    $32, %rsp
37      xorl    %edx, %edx
```

TERMINAL OUTPUT DEBUG CONSOLE PORTS

Test cases:

1. string.ngeb

```
x = 42;
y = 100;
z = 150;
```

```
if x >= 40 and y < 200 then
    if z > 100 then
        ngeb("Condition 1 True, Nested Condition True\n")ski
    else
        ngeb("Condition 1 True, Nested Condition False\n")ski
    end
else
    ngeb("Condition 1 False\n")ski
end

if x < 40 or y > 200 then
    ngeb("Condition 2 True\n")ski
else
    if z == 150 then
        ngeb("Condition 2 False, Nested Condition True\n")ski
    else
        ngeb("Condition 2 False, Nested Condition False\n")ski
    end
end

if x == 42 then
    if y == 100 then
        if z == 150 then
            ngeb("All conditions are True\n")ski
        end
    end
end

x_bool = true;
y_bool = false;

result1 = x_bool and y_bool;
result2 = x_bool or y_bool;
result3 = not x_bool;
result4 = not y_bool;

ngeb("Result of x_bool and y_bool: ")ski
ngeb(result1)ski
ngeb("Result of x_bool or y_bool: ")ski
ngeb(result2)ski
ngeb("Result of not x_bool: ")ski
ngeb(result3)ski
```

```
ngeb("Result of not y_bool: ")ski

a = 10;
b = 20;
condition1 = a > b;
condition2 = a < b;
condition3 = a == b;

if condition1 then
    ngeb("a is greater than b\n")ski
else
    ngeb("a is not greater than b\n")ski
end

if condition2 then
    ngeb("a is less than b\n")ski
else
    ngeb("a is not less than b\n")ski
end

if condition3 then
    ngeb("a is equal to b\n")ski
else
    ngeb("a is not equal to b\n")ski
end

complex_condition1 = (a < b) and (a != b) or (b == 20);
complex_condition2 = not ((a + b) > (b - a));
complex_condition3 = (a == 10) and ((b > a) or (a < 5));

ngeb("Complex condition 1: ")ski
ngeb(complex_condition1)ski
ngeb("Complex condition 2: ")ski
ngeb(complex_condition2)ski
ngeb("Complex condition 3: ")ski
ngeb(complex_condition3)ski
```

2. arithmetic1.ngeb

```
a = -10;
b = 20;
c = 30;
d = 40;
e = 50;
f = 60;
```

```
result1 = (a + b) * (c - d) / e + f;
result2 = a * b * c - d + e / f;
result3 = (a * b - c) + (d / e * f);
result4 = ((a + b) / c) * (d - e) + f;
result5 = a + b * c - d / e + f;

result6 = a + b * (c - d) / e + f;
result7 = (a * b) + (c - d) / e;
result8 = (b + c) - (a * d) / f;

complex_result1 = ((a + b) * (c - d) + (e / f) * (b - a) + c) / d;
complex_result2 = ((a * b + c) * d - e / f + (b * c) - a) * 2;
complex_result3 = (a + b - c) * (d / e) + f * (b - a + c - d / e);

ngeb(result1)ski
ngeb(result2)ski
ngeb(result3)ski
ngeb(result4)ski
ngeb(result5)ski
ngeb(result6)ski
ngeb(result7)ski
ngeb(result8)ski
ngeb(complex_result1)ski
ngeb(complex_result2)ski
ngeb(complex_result3)ski
```

3. arithmetic2.ngeb

```
a = -10.5;
b = 20.75;
c = -30.125;
d = 40.5;
e = 50.875;
f = -60.25;
g = 70.875;
h = -80.125;
i = 15.5;
j = -25.75;

result1 = ((a - b * c + d) / (e - f) + g) * h - i;
result2 = (a / b + c * d - e) * (f - g + h / i);
result3 = ((a - b) * (c + d) / (e - f)) + (g / h);
result4 = (a * b + c) / (d - e) + (f * g) - h;
result5 = ((a + b) / c * (d - e) + f) * (g + h) - i;

result6 = (a / b - c * d + e) * (f / g + h) / (i - j);
result7 = (a - b) / (c + (d * e) - f) + g - h * i;
result8 = ((b * c) - (d / e) + (f * g) - h) * (i / j);
result9 = (a + b - c) * (d / e + f - g) - (h / i);
result10 = ((a / b * c) - (d + e) * f) / (g - h) + i;

complex_result1 = ((a - b) * (c + d) / (e - f) + (g / h) * i) - j;
complex_result2 = ((a * b - c + d) / e * (f - g)) + (h / i) - j;
complex_result3 = ((a + b) - (c / d) * (e + f - g) / h) * i + j;
complex_result4 = (a * (b - c) + (d / e) * f - (g + h) / i) + j;
complex_result5 = ((a - b) * (c + d) - (e / f) + (g * h) / i) - j;

ngeb(result1)ski
ngeb(result2)ski
ngeb(result3)ski
ngeb(result4)ski
```

```
ngeb(result5)ski
ngeb(result6)ski
ngeb(result7)ski
ngeb(result8)ski
ngeb(result9)ski
ngeb(result10)ski
ngeb(complex_result1)ski
ngeb(complex_result2)ski
ngeb(complex_result3)ski
ngeb(complex_result4)ski
ngeb(complex_result5)ski
```

4. boolean.ngeb

```
x = true;
y = false;

result1 = x and y;
result2 = x or y;
result3 = not x;
result4 = not y;

ngeb(result1)ski
ngeb(result2)ski
ngeb(result3)ski
ngeb(result4)ski

a = 10;
b = 20;
condition1 = a > b;
condition2 = a < b;
condition3 = a == b;

if condition1 then
    ngeb("a is greater than b\n")ski
else
    ngeb("a is not greater than b\n")ski
end

if condition2 then
    ngeb("a is less than b\n")ski
else
    ngeb("a is not less than b\n")ski
end

if condition3 then
    ngeb("a is equal to b\n")ski
else
```

```
    ngeb("a is not equal to b\n")ski
end

complex_condition1 = (a < b) and (a != b) or (b == 20);
complex_condition2 = not ((a + b) > (b - a));
complex_condition3 = (a == 10) and ((b > a) or (a < 5));

ngeb(complex_condition1)ski
ngeb(complex_condition2)ski
ngeb(complex_condition3)ski
```

5. conditional.ngeb

```
x = 42;
y = 100;
z = 150;

if x >= 40 and y < 200 then
    if z > 100 then
        ngeb("Condition 1 True, Nested Condition True\n")ski
    else
        ngeb("Condition 1 True, Nested Condition False\n")ski
    end
else
    ngeb("Condition 1 False\n")ski
end

if x < 40 or y > 200 then
    ngeb("Condition 2 True\n")ski
else
    if z == 150 then
        ngeb("Condition 2 False, Nested Condition True\n")ski
    else
        ngeb("Condition 2 False, Nested Condition False\n")ski
    end
end

if x == 42 then
    if y == 100 then
        if z == 150 then
            ngeb("All conditions are True\n")ski
        end
    end
```

```
end

a = 10;
b = 20;
c = 30;
d = 40;

if a < b then
    if c > d then
        ngeb("a < b and c > d\n")ski
    else
        ngeb("a < b and c <= d\n")ski
    end
else
    if c > d then
        ngeb("a >= b and c > d\n")ski
    else
        ngeb("a >= b and c <= d\n")ski
    end
end
```

6. IfThenElse.ngeb

```
a = 5;
b = 10;
c = -15;
d = 20;
e = 25;
f = -30;
g = 35;
h = -40;
i = 45;
j = -50;
k = 55;
l = -60;
m = 65;
n = -70;
o = 75;

result1 = 0;
if a < b then
    result1 = a + b;
else
```

```
    result1 = a - b;
end

result2 = 0;
if c < 0 then
    result2 = c + d;
else
    result2 = c - d;
end

result3 = 0;
if e > f then
    result3 = e * f;
else
    result3 = e / f;
end

result4 = 0;
if g < h then
    result4 = g + h;
else
    result4 = g - h;
end

result5 = 0;
if i > j then
    result5 = i * j;
else
    result5 = i / j;
end

result6 = 0;
if k > l then
    result6 = k - l;
else
    result6 = k + l;
end

result7 = 0;
if m < n then
    result7 = m + n;
else
    result7 = m - n;
```

```
end

result8 = 0;
if o > a then
    result8 = o - a;
else
    result8 = o + a;
end

result9 = 0;
if a > 0 and b > 0 then
    result9 = a * b;
else
    result9 = a / b;
end

result10 = 0;
if c < 0 or d < 0 then
    result10 = c + d;
else
    result10 = c - d;
end

ngeb("result1 : \n")ski
ngeb(result1)ski

ngeb("result2 : \n")ski
ngeb(result2)ski

ngeb("result3 : \n")ski
ngeb(result3)ski

ngeb("result4 : \n")ski
ngeb(result4)ski

ngeb("result5 : \n")ski
ngeb(result5)ski

ngeb("result6 : \n")ski
ngeb(result6)ski

ngeb("result7 : \n")ski
ngeb(result7)ski
```

```
ngeb("result8 : \n")ski
ngeb(result8)ski

ngeb("result9 : \n")ski
ngeb(result9)ski

ngeb("result10 : \n")ski
ngeb(result10)ski
```

7. incDec.ngeb

```
x = 10;
x = x++;
ngeb(x)ski
x = x--;
ngeb(x)ski

y = 5;
y = y + 1;
ngeb(y)ski
y = y - 1;
ngeb(y)ski

a = 100;
b = 200;

while a < b do
    ngeb(a)ski
    a = a++;
end

while b > a do
    ngeb(b)ski
    b = b--;
end

p = 50;
q = 50;
p = p++;
q = q--;
ngeb(p)ski
ngeb(q)ski
```

```
nested_inc = 0;
nested_dec = 100;
while nested_inc < 10 do
    ngeb(nested_inc)ski
    nested_inc = nested_inc++;
    while nested_dec > 90 do
        ngeb(nested_dec)ski
        nested_dec = nested_dec--;
    end
end
```

8. integration.ngeb

```
a = 0ski
b = 10ski
sum = 0ski
product = 1ski
factorial = 1ski
max_num = 0ski

while a <= b do
    sum = sum + a ski
    a = a ++ ski
end
ngeb(sum)ski

i = 1ski
while i <= 10 do
    product = product * i ski
    i = i ++ ski
end
ngeb(product)ski

num = 15ski
is_even = false ski
if num - (num / 2 * 2) == 0 then
    is_even = true ski
else
    is_even = false ski
end
ngeb(is_even)ski
```

```
n = 5ski
i = 1ski
while i <= n do
    factorial = factorial * i ski
    i = i ++ ski
end
ngeb(factorial)ski

i = 0ski
current_num = 3ski
max_num = current_num ski
i = i ++ ski
current_num = 5ski
if current_num > max_num then
    max_num = current_num ski
end
i = i ++ ski
current_num = 1ski
if current_num > max_num then
    max_num = current_num ski
end
i = i ++ ski
current_num = 8ski
if current_num > max_num then
    max_num = current_num ski
end
i = i ++ ski
current_num = 7ski
if current_num > max_num then
    max_num = current_num ski
end
i = i ++ ski
current_num = 2ski
if current_num > max_num then
    max_num = current_num ski
end
ngeb(max_num)ski

i = 0ski
sum_squares = 0ski
while i < 10 do
    sum_squares = sum_squares + i * i ski
    i = i ++ ski
```

```
end
ngeb(sum_squares)ski

x = 5ski
y = 10ski
z = 15ski
result = false ski

if x < y then
    result = true ski
else
    result = false ski
end

if y >= z then
    result = false ski
else
    result = true ski
end
ngeb(result)ski

i = 1ski
total = 0ski
while i <= 10 do
    if i - (i / 2 * 2) == 0 then
        total = total + i ski
    else
        total = total - i ski
    end
    i = i ++ ski
end
ngeb(total)ski

counter = 0ski
n = 10ski
sum_even = 0ski
sum_odd = 0ski

while counter <= n do
    if counter - (counter / 2 * 2) == 0 then
        sum_even = sum_even + counter ski
    else
        sum_odd = sum_odd + counter ski
```

```
    end
    counter = counter ++ ski
end

ngeb(sum_even)ski
ngeb(sum_odd)ski
```

9. variable.ngeb

```
x = 42;
y = 3.14;
z = "test string";
w = true;
a = false;

temp1 = x + y;
temp2 = x - y;
temp3 = x * y;
temp4 = x / y;

bool_and = w and a;
bool_or = w or a;
bool_not_w = not w;
bool_not_a = not a;

ngeb(x)ski
ngeb(y)ski
ngeb(z)ski
ngeb(w)ski
ngeb(a)ski
ngeb(temp1)ski
ngeb(temp2)ski
ngeb(temp3)ski
ngeb(temp4)ski
ngeb(bool_and)ski
ngeb(bool_or)ski
ngeb(bool_not_w)ski
ngeb(bool_not_a)ski
```

10. while.ngeb

```
a = 5;
b = 10;
c = -15;
```

```
d = 20;
e = 25;
f = -30;
g = 35;
h = -40;
i = 45;
j = -50;
k = 55;
l = -60;
m = 65;
n = -70;
o = 75;

result1 = 0;
while a <= 50 do
    result1 = result1 + a * b - c / d + e;
    a = a + 5;
end

result2 = 1;
while b >= -50 do
    result2 = result2 * (f / g + h - i * j);
    b = b - 10;
end

result3 = 0;
while c < 0 do
    result3 = result3 + (k + l) / m - n * o;
    c = c + 5;
end

result4 = 1000;
while d > 0 do
    result4 = result4 - (a + b) * (c - d) / e + f;
    d = d - 10;
end

result5 = 0;
while e < 100 do
    result5 = result5 + (g / h) - (i * j) + k;
    e = e + 15;
end
```

```
result6 = 0;
while f < -10 do
    result6 = result6 + (l - m) / (n + o) * a;
    f = f + 5;
end

ngeb("result1 : \n")ski
ngeb(result1)ski

ngeb("result2 : \n")ski
ngeb(result2)ski

ngeb("result3 : \n")ski
ngeb(result3)ski

ngeb("result4 : \n")ski
ngeb(result4)ski

ngeb("result5 : \n")ski
ngeb(result5)ski

ngeb("result6 : \n")ski
ngeb(result6)ski
```

V. Results and Future Work

a) Test Results

The test results from the test cases pass successfully. The compiler is only tested for integers, boolean, floats and strings. Other types of data may result in lexer error. The compiler is also able to output IR code and assembly code based on the input. However, the user needs to give command to the compiler whether to output assembly or object files.

```
Target triple: x86_64-pc-windows-msvc
Enter the file type to generate (obj/asm):
```

asm or obj should be inserted in this prompt to continue into the code generator.

b) Performance Analysis

- Compilation time

  The Ngebski compiler has seven phases of compilation, including lexer, parser, abstract syntax tree generator, code generator for intermediate code, IR optimization for .obj and .s and machine code generator. Ngebski compiler favours runtime performance over compilation time to ensure performance for the targeted code. Therefore, the runtime complexity is linear O(1).

- Memory usage

  The memory for Ngebski compiler depends on the nodes in AST and does not depend on the input. Therefore the space complexity of this compiler is O(1) which is quite efficient.

- Generated code quality

  There are three types of codes generated by Ngebski compiler:

  - Assembly code
  - Intermediate representation code
  - Human readable output

  Generally, the code quality follows the syntax of IR and assembly. For human readable output, it follows the high-level language style output.

- Error handling

  There are several error handling inserted in the compiler, defined as error messages to tell the user which error results in compilation failure.

  - Syntax error

```
@self.pg.error
def error_handle(token):
    if token is None:
        raise ValueError('Unexpected end of input')
    raise ValueError(f'Syntax Error at token {token.gettokentype()} ({token.getstr()}) at line {token.getsourcepos().lineno}.')
```

  - Undefined variable error

```
        return Number(self.builder, self.module, p[0].getstr())
    elif token_type == 'ID':
        var_name = p[0].getstr()
        if var_name in self.variables:
            return self.variables[var_name]
        else:
            raise ValueError(f"Undefined variable {var_name}")
    elif token_type == 'TRUE':
```

  - Token error/undefined tokens

```
token_list = []
try:
    for token in tokens:
        token_list.append(token)
except Exception as e:
    print(f"Error: {e}")
```

- Variable initialization error

```
    if self.pointer is None:
        raise ValueError(f"Variable {self.name} has not been initialized")
    return self.builder.load(self.pointer, name=f"{self.name}.load")
```

- Unsupported variable error

```
            self.pointer = self.builder.alloca(ir.IntType(32), name=self.name)
    else:
        raise ValueError(f"Unsupported type for variable {self.name}")
```

- File type error

```
def compile_to_executable(ir_filename, output_filename, filetype='obj'):
    if filetype not in ['obj', 'asm']:
        print(f"Unsupported file type: {filetype}")
        return False
```

- Linking error

```
    try:
        subprocess.run(gcc_command, check=True)
    except subprocess.CalledProcessError as e:
        print(f"Error linking object file to executable: {e}")
        return False
else:
```

- Compilation error

```
if run_main():
    if os.path.isfile(ir_filename):
        if run_compile(ir_filename, output_filename):
            run_executable(output_filename)
        else:
            print("Compilation failed.")
    else:
        print(f"Error: {ir_filename} not created.")
else:
    print("Main script failed.")
```

- File location error

```
executable_path = f"{output_filename}.exe"
if not os.path.isfile(executable_path):
    print(f"Error: {executable_path} does not exist.")
    return False
```

c) Future Enhancements

There are some features that can be added to improve the Ngebski language, including

- Function calls
- For loop
- Integration with other operating systems aside from Windows and MacOS
- Type system enhancement
- Complex mathematical functions

This concludes the Ngebski programming language compiler project. By applying compiler principles from lexer analysis, LL tables, syntax definer and BNF, the Ngebski compiler is finished successfully. We applied rigorous testing to ensure there are no ambiguity and errors in the compiler. Additionally, We designed the compiler to be available in both Windows and MacOS to ensure its flexibility.

VI. References

1. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley.
2. Grune, D., & Jacobs, C. J. H. (1990). *Parsing Techniques: A Practical Guide*. Ellis Horwood.
3. Intermediate Code Generation in Compiler Design. (2024, January 19). Retrieved June 12, 2024, from [Intermediate Code Generation in Compiler Design - GeeksforGeeks]
4. LLVM Project. (n.d.). *LLVM Language Reference Manual*. LLVM Documentation. Retrieved June 12, 2024, from https://llvm.org/docs/LangRef.html

APPENDIX:

lexx.py

```
# /mnt/data/lexx.py



from rply import LexerGenerator



class Lexx():

    def __init__(self):

        self.lexer = LexerGenerator()



    def _add_tokens(self):

        # If-Then-Else

        self.lexer.add('IF', r'if')

        self.lexer.add('THEN', r'then')

        self.lexer.add('ELSE', r'else')

        self.lexer.add('END', r'end')
```

```
        # Number types
        self.lexer.add('FLOAT', r'-?\d+\.\d+')
        self.lexer.add('LONG', r'-?\d{10,}')
        self.lexer.add('INTEGER', r'-?\d+')
        self.lexer.add('SHORT', r'-?\d{1,4}')
        # Boolean literals (prioritized over ID)
        self.lexer.add('TRUE', r'true')
        self.lexer.add('FALSE', r'false')
        # Boolean operators
        self.lexer.add('AND', r'and')
        self.lexer.add('OR', r'or')
        self.lexer.add('NOT', r'not')
        # Print
        self.lexer.add('PRINT', r'ngeb')
        # While loop
        self.lexer.add('WHILE', r'while')
        self.lexer.add('DO', r'do')
        # Comparison operators
        self.lexer.add('EQ', r'==')
        self.lexer.add('NEQ', r'!=')
        self.lexer.add('LEQ', r'<=')
        self.lexer.add('GEQ', r'>=')
        self.lexer.add('LT', r'<')
```

```
        self.lexer.add('GT', r'>')

        # Parenthesis

        self.lexer.add('OPEN_PAREN', r'\(')

        self.lexer.add('CLOSE_PAREN', r'\)')

        # Semi Colon

        self.lexer.add('SEMI_COLON', r'ski')

        self.lexer.add('SEMI_COLON', r';')

        # Increment and decrement

        self.lexer.add('INCREMENT', r'\+\+')

        self.lexer.add('DECREMENT', r'\-\-')

        # Operators

        self.lexer.add('SUM', r'\+')

        self.lexer.add('SUB', r'\-')

        self.lexer.add('MUL', r'\*')

        self.lexer.add('DIV', r'\/')

        # Identifier (variable name)

        self.lexer.add('ID', r'[a-zA-Z_][a-zA-Z0-9_]*')

        # Assignment

        self.lexer.add('EQUALS', r'\=')

        # String literal

        self.lexer.add('STRING_LITERAL', r'"[^"]*"')

        # Ignore spaces

        self.lexer.ignore(r'\s+')
```

```
    def get_lexer(self):
        self._add_tokens()
        return self.lexer.build()
```

parser.py

```
from rply import ParserGenerator
from astree import Number, Sum, Sub, Mul, Div, Print, Assign, Variable,
Condition, IfThenElse, While, Boolean, BooleanOp, BooleanNot, Inc, Dec,
StringLiteral


class Parser():
    def __init__(self, module, builder, printf):
        self.pg = ParserGenerator(
                    ['INTEGER', 'FLOAT', 'LONG', 'SHORT', 'PRINT',
'OPEN_PAREN', 'CLOSE_PAREN',
             'SEMI_COLON', 'SUM', 'SUB', 'MUL', 'DIV', 'EQUALS', 'ID',
                'IF', 'THEN', 'ELSE', 'END', 'EQ', 'NEQ', 'LT', 'GT',
'LEQ', 'GEQ',
                 'WHILE', 'DO', 'TRUE', 'FALSE', 'AND', 'OR', 'NOT',
'INCREMENT', 'DECREMENT', 'STRING_LITERAL'],
            precedence=[
                ('left', ['AND', 'OR']),
                ('left', ['EQ', 'NEQ', 'LT', 'GT', 'LEQ', 'GEQ']),
                ('left', ['SUM', 'SUB']),
```

```
                ('left', ['MUL', 'DIV']),
                ('right', ['NOT'])
            ]
        )
        self.module = module
        self.builder = builder
        self.printf = printf
        self.variables = {}

    def parse(self):
        @self.pg.production('program : statement_list')
        def program(p):
            return p[0]

                @self.pg.production('statement_list  :  statement_list
statement')
        @self.pg.production('statement_list : statement')
        def statement_list(p):
            if len(p) == 1:
                return [p[0]]
            else:
                p[0].append(p[1])
                return p[0]
```

```
        @self.pg.production('statement : PRINT OPEN_PAREN expression
CLOSE_PAREN SEMI_COLON')
        def statement_print(p):
            return Print(self.builder, self.module, self.printf, p[2])


        @self.pg.production('statement : assign')
        @self.pg.production('statement : if_statement')
        @self.pg.production('statement : while_statement')
        @self.pg.production('statement : expression SEMI_COLON')
        def statement(p):
            return p[0]


        @self.pg.production('assign : ID EQUALS expression SEMI_COLON')
            @self.pg.production('assign : ID EQUALS boolean_expression
SEMI_COLON')
                @self.pg.production('assign : ID EQUALS ID INCREMENT
SEMI_COLON')
                @self.pg.production('assign : ID EQUALS ID DECREMENT
SEMI_COLON')
        def assign(p):
            var_name = p[0].getstr()
            if var_name not in self.variables:
                    self.variables[var_name] = Variable(self.builder,
self.module, var_name)
```

```
            variable = self.variables[var_name]

            if len(p) == 4:

                value = p[2]

            else:

                id_name = p[2].getstr()

                if id_name not in self.variables:

                    raise ValueError(f"Undefined variable {id_name}")

                id_variable = self.variables[id_name]

                if p[3].gettokentype() == 'INCREMENT':

                    value = Inc(self.builder, self.module, id_variable)

                else:

                    value = Dec(self.builder, self.module, id_variable)

            return Assign(self.builder, self.module, variable, value)



            @self.pg.production('if_statement : IF condition THEN
statement_list optional_else END')

        def if_statement(p):

            condition = p[1]

            then_body = p[3]

            else_body = p[4]

                return IfThenElse(self.builder, self.module, condition,
then_body, else_body)
```

```
        @self.pg.production('optional_else : ELSE statement_list')
        @self.pg.production('optional_else : ')
        def optional_else(p):
            if len(p) == 0:
                return []
            return p[1]


            @self.pg.production('while_statement : WHILE condition DO
statement_list END')
        def while_statement(p):
            condition = p[1]
            body = p[3]
            return While(self.builder, self.module, condition, body)


        @self.pg.production('condition : boolean_expression')
        def condition(p):
            return p[0]


          @self.pg.production('boolean_expression : boolean_expression
AND boolean_expression')
        @self.pg.production('boolean_expression : boolean_expression OR
boolean_expression')
        @self.pg.production('boolean_expression : comparison')
        @self.pg.production('boolean_expression : term')
```

```
        def boolean_expression(p):

            if len(p) == 3:

                left = p[0]

                operator = p[1].gettokentype()

                right = p[2]

                      return BooleanOp(self.builder, self.module, left,
right, operator)

            else:

                return p[0]



        @self.pg.production('comparison : expression EQ expression')

        @self.pg.production('comparison : expression NEQ expression')

        @self.pg.production('comparison : expression LT expression')

        @self.pg.production('comparison : expression GT expression')

        @self.pg.production('comparison : expression LEQ expression')

        @self.pg.production('comparison : expression GEQ expression')

        def comparison(p):

            left = p[0]

            operator = p[1].gettokentype()

            right = p[2]

               return Condition(self.builder, self.module, left, right,
operator)
```

```
        @self.pg.production('expression : expression SUM term')
        @self.pg.production('expression : expression SUB term')
        def expression(p):
            left = p[0]
            right = p[2]
            operator = p[1]
            if operator.gettokentype() == 'SUM':
                return Sum(self.builder, self.module, left, right)
            elif operator.gettokentype() == 'SUB':
                return Sub(self.builder, self.module, left, right)


        @self.pg.production('expression : term')
        @self.pg.production('expression : boolean_expression')
        @self.pg.production('expression : NOT expression')
        @self.pg.production('expression : SUB term')
        @self.pg.production('expression : SUM term')
        def expression_single_term(p):
            if len(p) == 2:
                if p[0].gettokentype() == 'SUB':
                                return Sub(self.builder, self.module,
Number(self.builder, self.module, '0'), p[1])
                if p[0].gettokentype() == 'SUM':
```

```
                                return Sum(self.builder, self.module,
Number(self.builder, self.module, '0'), p[1])

                elif p[0].gettokentype() == 'NOT':

                    return BooleanNot(self.builder, self.module, p[1])

            return p[0]



        @self.pg.production('term : term MUL factor')

        @self.pg.production('term : term DIV factor')

        def term(p):

            left = p[0]

            right = p[2]

            operator = p[1]

            if operator.gettokentype() == 'MUL':

                return Mul(self.builder, self.module, left, right)

            elif operator.gettokentype() == 'DIV':

                return Div(self.builder, self.module, left, right)



        @self.pg.production('term : factor')

        def term_factor(p):

            return p[0]



        @self.pg.production('factor : INTEGER')

        @self.pg.production('factor : FLOAT')
```

```
        @self.pg.production('factor : LONG')

        @self.pg.production('factor : SHORT')

        @self.pg.production('factor : ID')

                @self.pg.production('factor  :  OPEN_PAREN  expression
CLOSE_PAREN')

        @self.pg.production('factor : TRUE')

        @self.pg.production('factor : FALSE')

        @self.pg.production('factor : STRING_LITERAL')

        def factor(p):

            if len(p) == 1:

                token_type = p[0].gettokentype()

                if token_type == 'INTEGER':

                                return  Number(self.builder,  self.module,
p[0].getstr())

                elif token_type == 'FLOAT':

                                return  Number(self.builder,  self.module,
p[0].getstr())

                elif token_type == 'LONG':

                                return  Number(self.builder,  self.module,
p[0].getstr())

                elif token_type  == 'SHORT':

                                return  Number(self.builder,  self.module,
p[0].getstr())

                elif token_type == 'ID':

                    var_name = p[0].getstr()
```

```
                    if var_name in self.variables:
                        return self.variables[var_name]
                    else:
                                raise ValueError(f"Undefined variable
{var_name}")
                elif token_type == 'TRUE':
                    return Boolean(self.builder, self.module, 1)
                elif token_type == 'FALSE':
                    return Boolean(self.builder, self.module, 0)
                elif token_type == 'STRING_LITERAL':
                        return StringLiteral(self.builder, self.module,
p[0].getstr())
            elif len(p) == 3:
                return p[1]


        @self.pg.error
        def error_handle(token):
            if token is None:
                raise ValueError('Unexpected end of input')
                        raise ValueError(f'Syntax Error at token
{token.gettokentype()} ({token.getstr()}) at line
{token.getsourcepos().lineno}.')


    def get_parser(self):
```

```
        return self.pg.build()

 # type: ignore
```

astree.py

```
import itertools

from llvmlite import ir



class Number():

    def __init__(self, builder, module, value):

        self.builder = builder

        self.module = module

        self.value = value



    def eval(self):

        if '.' in str(self.value):

            return ir.Constant(ir.FloatType(), float(self.value))

        else:

            return ir.Constant(ir.IntType(32), int(self.value))



class StringLiteral():

    def __init__(self, builder, module, value):

        self.builder = builder

        self.module = module

        self.value = value.strip('"')
```

```
    def eval(self):

        str_value = self.value.replace("\\n", "\n")

        str_value += "\0"

        str_bytes = bytearray(str_value.encode("utf8"))

        str_len = len(str_bytes)

            str_fmt = ir.Constant(ir.ArrayType(ir.IntType(8), str_len),
str_bytes)



             global_str = ir.GlobalVariable(self.module, str_fmt.type,
name="str")

        global_str.linkage = 'internal'

        global_str.global_constant = True

        global_str.initializer = str_fmt



        voidptr_ty = ir.IntType(8).as_pointer()

        return self.builder.bitcast(global_str, voidptr_ty)



class BinaryOp():

    def __init__(self, builder, module, left, right):

        self.builder = builder

        self.module = module

        self.left = left
```

```
        self.right = right



    def _convert_to_same_type(self, left_val, right_val):

                        if   isinstance(left_val.type,   ir.IntType)   and
isinstance(right_val.type, ir.FloatType):

            left_val = self.builder.sitofp(left_val, ir.FloatType())

                    elif   isinstance(left_val.type,   ir.FloatType)   and
isinstance(right_val.type, ir.IntType):

            right_val = self.builder.sitofp(right_val, ir.FloatType())

        return left_val, right_val



class Sum(BinaryOp):

    def eval(self):

        left_val = self.left.eval()

        right_val = self.right.eval()



            left_val,  right_val  =  self._convert_to_same_type(left_val,
right_val)



        if isinstance(left_val.type, ir.FloatType):

                        result  =  self.builder.fadd(left_val,  right_val,
name="sumtmp")

        else:

                         result  =  self.builder.add(left_val,  right_val,
name="sumtmp")
```

```
        return result


class Sub(BinaryOp):
    def eval(self):
        left_val = self.left.eval()
        right_val = self.right.eval()

            left_val, right_val = self._convert_to_same_type(left_val,
right_val)

        if isinstance(left_val.type, ir.FloatType):
                    result = self.builder.fsub(left_val, right_val,
name="subtmp")
        else:
                     result = self.builder.sub(left_val, right_val,
name="subtmp")

        return result


class Mul(BinaryOp):
    def eval(self):
        left_val = self.left.eval()
        right_val = self.right.eval()
```

```
            left_val, right_val = self._convert_to_same_type(left_val,
right_val)

        if isinstance(left_val.type, ir.FloatType):
                    result = self.builder.fmul(left_val, right_val,
name="multmp")
        else:
                     result = self.builder.mul(left_val, right_val,
name="multmp")

        return result

class Div(BinaryOp):
    def eval(self):
        left_val = self.left.eval()
        right_val = self.right.eval()

            left_val, right_val = self._convert_to_same_type(left_val,
right_val)

        if isinstance(left_val.type, ir.FloatType):
                    result = self.builder.fdiv(left_val, right_val,
name="divtmp")
        else:
```

```
                    result = self.builder.sdiv(left_val, right_val,
name="divtmp")



        return result



class UnaryOp():

    def __init__(self, builder, module, operand, variable=None):

        self.builder = builder

        self.module = module

        self.operand = operand

        self.variable = variable



class Inc(UnaryOp):

    def eval(self):

        var = self.operand.eval()

          inc_value = self.builder.add(var, ir.Constant(ir.IntType(32),
1))

        if self.variable:

            self.variable.store(inc_value)

        return inc_value



class Dec(UnaryOp):

    def eval(self):
```

```
        var = self.operand.eval()

          dec_value = self.builder.sub(var, ir.Constant(ir.IntType(32),
1))

        if self.variable:

            self.variable.store(dec_value)

        return dec_value



class Variable():

    def __init__(self, builder, module, name):

        self.builder = builder

        self.module = module

        self.name = name

        self.pointer = None



    def allocate(self, value):

        if isinstance(value, ir.Constant):

            if value.type == ir.IntType(1):

                    self.pointer = self.builder.alloca(ir.IntType(1),
name=self.name)

            elif value.type == ir.FloatType():

                    self.pointer = self.builder.alloca(ir.FloatType(),
name=self.name)

                      elif isinstance(value.type, ir.ArrayType) and
value.type.element == ir.IntType(8):
```

```
                        self.pointer = self.builder.alloca(value.type,
name=self.name)

            else:

                    self.pointer = self.builder.alloca(ir.IntType(32),
name=self.name)

        elif isinstance(value, (Boolean, BooleanNot, BooleanOp)):

                    self.pointer = self.builder.alloca(ir.IntType(1),
name=self.name)

        elif isinstance(value, ir.instructions.Instruction):

            if value.type == ir.IntType(1):

                     self.pointer = self.builder.alloca(ir.IntType(1),
name=self.name)

            elif value.type == ir.FloatType():

                    self.pointer = self.builder.alloca(ir.FloatType(),
name=self.name)

                     elif isinstance(value.type, ir.PointerType) and
value.type.pointee == ir.IntType(8):

                        self.pointer = self.builder.alloca(value.type,
name=self.name)

            else:

                    self.pointer = self.builder.alloca(ir.IntType(32),
name=self.name)

        else:

                    raise ValueError(f"Unsupported type for variable
{self.name}")



    def store(self, value):
```

```
        if self.pointer is None:
            self.allocate(value)
        self.builder.store(value, self.pointer)

    def load(self):
        if self.pointer is None:
            raise ValueError(f"Variable {self.name} has not been
initialized")
        return self.builder.load(self.pointer,
name=f"{self.name}.load")

    def eval(self):
        return self.load()

class Assign():
    def __init__(self, builder, module, variable, value):
        self.builder = builder
        self.module = module
        self.variable = variable
        self.value = value

    def eval(self):
        value_to_store = self.value.eval()
```

```
        self.variable.store(value_to_store)



class Print():

    counter = itertools.count()



    def __init__(self, builder, module, printf_func, value):

        self.builder = builder

        self.module = module

        self.printf_func = printf_func

        self.value = value

        self.fstr_name = f"fstr_{next(self.counter)}"



    def eval(self):

        voidptr_ty = ir.IntType(8).as_pointer()



        if isinstance(self.value, StringLiteral):

            str_value = self.value.value

            str_value = str_value.replace("\\n", "\n")

            str_value += "\0"

            str_bytes = bytearray(str_value.encode("utf8"))

            str_len = len(str_bytes)

             str_fmt = ir.Constant(ir.ArrayType(ir.IntType(8), str_len),
str_bytes)
```

```
            global_str = ir.GlobalVariable(self.module, str_fmt.type,
name=self.fstr_name)
            global_str.linkage = 'internal'
            global_str.global_constant = True
            global_str.initializer = str_fmt

            fmt_str = "%s\n\0"
            fmt_arg = self.builder.bitcast(global_str, voidptr_ty)
            self.builder.call(self.printf_func, [fmt_arg, fmt_arg])

        else:
            value_to_print = self.value.eval()
            fmt_str = ""
            fmt_arg = None

            if isinstance(value_to_print.type, ir.IntType):
                fmt_str = "%i\n\0"
                if value_to_print.type.width == 1:
                    value_to_print = self.builder.zext(value_to_print,
ir.IntType(32))
                fmt_arg = value_to_print
            elif isinstance(value_to_print.type, ir.FloatType):
```

```
                fmt_str = "%f\n\0"

                    value_to_print = self.builder.fpext(value_to_print,
ir.DoubleType())

                fmt_arg = value_to_print

              elif isinstance(value_to_print.type, ir.PointerType) and
value_to_print.type.pointee == ir.IntType(8):

                fmt_str = "%s\n\0"

                fmt_arg = value_to_print



           if fmt_str:

                       c_fmt = ir.Constant(ir.ArrayType(ir.IntType(8),
len(fmt_str)), bytearray(fmt_str.encode("utf8")))

                global_fmt = ir.GlobalVariable(self.module, c_fmt.type,
name=self.fstr_name)

                global_fmt.linkage = 'internal'

                global_fmt.global_constant = True

                global_fmt.initializer = c_fmt

                fmt_arg = self.builder.bitcast(global_fmt, voidptr_ty)

                           self.builder.call(self.printf_func, [fmt_arg,
value_to_print])



class Condition():

    def __init__(self, builder, module, left, right, operator):

        self.builder = builder

        self.module = module
```

```
        self.left = left

        self.right = right

        self.operator = operator



    def eval(self):

        left_val = self.left.eval()

        right_val = self.right.eval()

        if self.operator == 'EQ':

                    return  self.builder.icmp_unsigned('==',  left_val,
right_val)

        elif self.operator == 'NEQ':

                    return  self.builder.icmp_unsigned('!=',  left_val,
right_val)

        elif self.operator == 'LT':

            return self.builder.icmp_unsigned('<', left_val, right_val)

        elif self.operator == 'GT':

            return self.builder.icmp_unsigned('>', left_val, right_val)

        elif self.operator == 'LEQ':

                    return  self.builder.icmp_unsigned('<=',  left_val,
right_val)

        elif self.operator == 'GEQ':

                    return  self.builder.icmp_unsigned('>=',  left_val,
right_val)
```

```
class IfThenElse():

        def __init__(self, builder, module, condition, then_body,
else_body):

        self.builder = builder

        self.module = module

        self.condition = condition

        self.then_body = then_body

        self.else_body = else_body



    def eval(self):

        cond_val = self.condition.eval()

        then_block = self.builder.append_basic_block('then')

        else_block = self.builder.append_basic_block('else')

        merge_block = self.builder.append_basic_block('ifcont')

        self.builder.cbranch(cond_val, then_block, else_block)



        self.builder.position_at_start(then_block)

        for stmt in self.then_body:

            stmt.eval()

        self.builder.branch(merge_block)

        then_block = self.builder.block



        self.builder.position_at_start(else_block)
```

```
        for stmt in self.else_body:
            stmt.eval()
        self.builder.branch(merge_block)
        else_block = self.builder.block

        self.builder.position_at_start(merge_block)


class Boolean():
    def __init__(self, builder, module, value):
        self.builder = builder
        self.module = module
        self.value = value

    def eval(self):
        return ir.Constant(ir.IntType(1), int(self.value))


class BooleanOp(BinaryOp):
    def __init__(self, builder, module, left, right, operator):
        super().__init__(builder, module, left, right)
        self.operator = operator

    def eval(self):
        left_val = self.left.eval()
```

```
        right_val = self.right.eval()

        if self.operator == 'AND':

            return self.builder.and_(left_val, right_val)

        elif self.operator == 'OR':

            return self.builder.or_(left_val, right_val)



class BooleanNot():

    def __init__(self, builder, module, value):

        self.builder = builder

        self.module = module

        self.value = value



    def eval(self):

        val = self.value.eval()

        return self.builder.not_(val)



class While():

    def __init__(self, builder, module, condition, body):

        self.builder = builder

        self.module = module

        self.condition = condition

        self.body = body
```

```
    def eval(self):

        loop_cond_block = self.builder.append_basic_block('loop_cond')

        loop_body_block = self.builder.append_basic_block('loop_body')

        loop_end_block = self.builder.append_basic_block('loop_end')



        self.builder.branch(loop_cond_block)



        self.builder.position_at_start(loop_cond_block)

        cond_val = self.condition.eval()

        self.builder.cbranch(cond_val, loop_body_block, loop_end_block)



        self.builder.position_at_start(loop_body_block)

        for stmt in self.body:

            stmt.eval()

        self.builder.branch(loop_cond_block)



        self.builder.position_at_start(loop_end_block)
```

CodeGen.py

```
from llvmlite import ir, binding
```

```
class CodeGen():

    def __init__(self):

        self.binding = binding

        self.binding.initialize()

        self.binding.initialize_all_targets()

        self.binding.initialize_all_asmprinters()

        self._config_llvm()

        self._create_execution_engine()

        self._declare_print_function()



    def _config_llvm(self):

        self.module = ir.Module(name=__file__)

          self.module.triple = binding.get_default_triple()   # Get the
default triple

        if "arm64" in self.module.triple:

            self.module.triple = "arm64-apple-macosx11.0.0"

        func_type = ir.FunctionType(ir.VoidType(), [], False)

        base_func = ir.Function(self.module, func_type, name="main")

        block = base_func.append_basic_block(name="entry")

        self.builder = ir.IRBuilder(block)



    def _create_execution_engine(self):

        target = self.binding.Target.from_triple(self.module.triple)
```

```
        target_machine = target.create_target_machine()

        backing_mod = self.binding.parse_assembly("")

          self.engine = self.binding.create_mcjit_compiler(backing_mod,
target_machine)



    def _declare_print_function(self):

        voidptr_ty = ir.IntType(8).as_pointer()

            printf_ty = ir.FunctionType(ir.IntType(32), [voidptr_ty],
var_arg=True)

        printf = ir.Function(self.module, printf_ty, name="printf")

        self.printf = printf



    def _compile_ir(self):

        self.builder.ret_void()

        llvm_ir = str(self.module)

        mod = self.binding.parse_assembly(llvm_ir)

        mod.verify()

        self.engine.add_module(mod)

        self.engine.finalize_object()

        self.engine.run_static_constructors()

        return mod



    def create_ir(self):
```

```
        self._compile_ir()



    def save_ir(self, filename):

        with open(filename, 'w') as output_file:

            output_file.write(str(self.module))



    def _declare_global_string(self, name, value):

        str_val = bytearray(value.encode('utf8'))

                str_const  =  ir.Constant(ir.ArrayType(ir.IntType(8),
len(str_val)), str_val)

           global_var = ir.GlobalVariable(self.module, str_const.type,
name=name)

        global_var.linkage = 'internal'

        global_var.global_constant = True

        global_var.initializer = str_const

        return global_var
```

main.py

```
import warnings

import sys

from lexx import Lexx

from parser import Parser  # type: ignore

from CodeGen import CodeGen

import io
```

```
# Suppress specific warnings
warnings.filterwarnings("ignore")

# Redirect stderr to suppress warnings
stderr = sys.stderr
sys.stderr = io.StringIO()

# Read the input file
fname = input("Please enter the input file name: ") + ".ngeb"
with open(fname) as f:
    text_input = f.read()

# Lexical analysis
lexx = Lexx().get_lexer()
tokens = lexx.lex(text_input)

token_list = []
try:
    for token in tokens:
        token_list.append(token)
except Exception as e:
    print(f"Error: {e}")
```

```
# Code generation

codegen = CodeGen()



module = codegen.module

builder = codegen.builder

printf = codegen.printf



# Parsing

pg = Parser(module, builder, printf)

pg.parse()

parser = pg.get_parser()

parsed_program = parser.parse(iter(token_list))



# Evaluate all statements in the parsed program

for statement in parsed_program:

    statement.eval()



# Create and save the IR code

codegen.create_ir()

codegen.save_ir("ngob.ll")



# Ensure the target triple is set correctly
```

```
print(f"Target triple: {codegen.module.triple}")



# Restore stderr

sys.stderr = stderr
```