Compiler and its NanoScript

Description

The NanoScript Compiler is a custom-built compiler/parser designed as an educational tool to help users and learners explore the inner workings of compilers. It processes **NanoScript(.ns)**, a simple, custom programming language with the following features:

- **Variables**: Supports integers (int), characters (char), and strings (string).
- **Pointers**: Allows pointer operations for advanced memory manipulation.
- **Loops**: Includes while and for loops for iteration.
- **Statements**: Supports if-else conditionals and break statements for control flow.
- **Basic Operations**: Arithmetic (e.g., +, -, *, /), comparison (e.g., ==, <, >), and logical operations (e.g., &&, ||).

Technologies Used

The compiler leverages industry-standard tools:

- **LLVM**: For generating Intermediate Representation (IR) and memory allocation logic.
- **Clang**: For conversion from IR(.ll) to executable(.exe).
- **Flex**: For lexical analysis (breaking source code into tokens).
- **Bison**: For syntactic analysis (building an Abstract Syntax Tree based on grammar rules).

This combination makes NanoScript both powerful and educational, bridging theoretical compiler concepts with practical implementation.

Installation

Prerequisites

To build and run the NanoScript compiler, you'll need:

- **Clang**: A C/C++ compiler compatible with LLVM.
- **LLVM**: Version 15 or higher for IR generation and code compilation.
- **Bison**: For generating the parser from grammar rules.
- **Flex**: For generating the lexer from token definitions.
- **C++ Comp**

Installation Steps

1. **Clone the Repository**:

   ```
   git clone https://github.com/VedaangNarkhede/Compiler.git
   ```

2. **Navigate to the Directory**:

   ```
   cd compiler_directory
   ```

3. **Build the Project**:
   - Clean any existing build files: `make clean`
   - Build the project: `make`
4. This creates the compiler.exe executable.
5. **Test the Compiler**:
   - Compile a sample NanoScript file (e.g., demo.ns):
     `compiler.exe` or

     `./compiler demo.ns`
   - This generates an executable named out.exe.
6. **Run the Output**: `out.exe` or `./out`

These steps ensure the compiler is correctly set up and ready to process NanoScript code.

Usage

Basic Example

Here's a sample NanoScript program demonstrating key features:

```
number a = 5;
word b = "hello";
for number i = 0 to 4, 1
    print(a + i);
end
print(b);
```

Running the Compiler

1. Save the code in a file, e.g., demo.ns.
2. Compile it: `compiler.exe demo.ns`
3. Run the generated executable: `out.exe` or `./out`

Expected Output

```
5
6
7
8
9
hello
```

This demonstrates how NanoScript handles variable declaration, loops, arithmetic, and output.

Code Structure

The NanoScript compiler is modular, reflecting the four core stages of compilation: **lexing**, **parsing**, **semantic analysis**, and **IR generation**. Below is a breakdown of its components and their roles.

1. Lexer (lexer.l)

- **Purpose**: Performs *lexical analysis* by converting NanoScript source code into a stream of tokens.
- **How It Works**: Reads the input character-by-character and matches patterns (e.g., keywords like int, operators like =, identifiers).
- **Example**:
    - Input: number a = 5;
    - Output Tokens: INT, IDENTIFIER("a"), ASSIGN(=), NUMBER(5), SEMICOLON(;).

2. Parser (proj.y)

- **Purpose**: Performs *syntactic analysis* by constructing an Abstract Syntax Tree (AST) based on NanoScript's grammar rules, defined in the Bison file.
- **How It Works**: Takes the token stream from the lexer and ensures it follows the language's syntax, building a tree representation.
- **Example**:
    - Input Tokens: INT, IDENTIFIER("a"), ASSIGN(=), NUMBER(5), SEMICOLON(;).
    - Output AST: A node representing a variable declaration with an assignment.

3. Semantic Analysis

- **Purpose**: Validates the code's meaning, catching errors like undeclared variables or type mismatches.
- **Implementation**: Uses a **symbol table** (defined in symbol_table.h), which maps variable names to their types and LLVM values, tracking scope and usage.
- **Example**:
  - Valid: number a = 5; (type matches).
  - Invalid: number a = "string"; (type mismatch, error flagged).

4. IR Generation (proj.y with llvm_ir_builder.cpp)

- **Purpose**: Translates the AST into LLVM Intermediate Representation (IR), a portable, low-level code format.
- **How It Works**: The parser integrates with LLVM APIs to emit IR instructions, which are later compiled into machine code.
- **Example**:
  - Input: print(a);
  - Output IR: Calls printf with the value of a, after loading it from memory.

Main Workflow (compiler.cpp)

- Coordinates the lexer, parser, semantic analysis, and IR generation.
- Steps:
  1. **Lexing**: Tokenize the input.
  2. **Parsing**: Builds an AST for the for loop structure.
  3. **Semantic Analysis**: Verifies i's usage using the symbol table.
  4. **IR Generation**: Emits LLVM IR for loop initialization, condition checking, and body execution.

This modular design makes the compiler easy to understand and modify, ideal for educational purposes.

Syntax Rules

Integer

- **Declaration:**
  - `number a;`
  - `number a = 5;`
- **Print:**
  - `print(a);`

Operators

- **Arithmetic:**
  - `number result = a + b; (addition)`
  - `number result = a - b; (subtraction)`
  - `number result = a * b; (multiplication)`
  - `number result = a / b; (division)`
- **Comparison:**

```
if (a > b) {
    print("Greater");
}

if (a <= b) {
    print("Less or equal");
}

if (a >= b) {
    print("Greater or equal");
}

if (a == b) {
    print("Equal");
}

if (a != b) {
    print("Not equal");
}
```

- **Logical:**

```
if (a && b) {
      print("Both true");
}
```

```
if (a and b) {
      print("Both true");
}
```

```
if (a || b) {
    print("One true");
}
```

```
if (a or b) {
    print("One true");
}
```

Characters

- **Declaration:**

```
letter c; //(declares a character variable 'c')

letter c = 'A'; //declares and initializes character 'c' to 'A')
```

- **Print:**

```
print(c); //prints the character 'c')
```

Strings

- **Declaration:**

```
# Word s; //declares a string variable 's')

# Word s = "hello"; //declares and initializes 's' to "hello"')
```

- **Print:**

```
print(s); //prints the string)
```

If-Else Statements

Syntax:

```
if (condition) {
    // code if true
} else {
    // code if false

}
```

Example:

```
number a = 5;
if (a > 0) {
    print("Positive");
}
else {
    print("Non-positive");
}
```

While Loops

Syntax:

```
while (condition) {
    // loop body

}
```

Example:

```
number i = 0;
while (i < 5) {
    print(i);
    i = i + 1; //or i++;
}
```

For Loops

Syntax:

```
for number variable = start to end, step
    // loop body

end
```

Example (Ascending):

```
for number a = 1 to 5, 1
    print(a);  // prints 1, 2, 3, 4, 5
end
```

Example (Descending):

```
for number a = 5 to 1, -1
    print(a);  // prints 5, 4, 3, 2, 1
end
```

Switch Statements

Syntax:

```
switch (expression) {
    case value1 {
        // code for value1
    }
    case value2 {
        // code for value2
    }
    default {
        // default code
    }

}
```

Example:

```
number a = 2;
switch (a) {
    case 1 {
        print("One");
    }
    case 2 {
        print("Two");
    }
    default {
        print("Other");
    }
}
```

Functions

- **Syntax:**
  - `function return_type function_name(parameter_type param_name, ...)`
- **Example:**

```
function number sum(number a, number b)
{
    number result = a + b;
    return result;
}

number y = sum(3, 4);
print(y);
```

```
function void func()
{
    print("Hello from void function!");
    return;
}

func();
```

Break Statements

- **Syntax:**
  - `break;` (exits the current loop or switch)
- **Example:**

```
for number i = 0 to 10, 1
    if (i == 5) {
        break;
    }
    print(i);  // prints 0, 1, 2, 3, 4
end
```

Pointers

- **Declaration:**

```
ptr p; //(declares a pointer 'p')
```

- Access the value with **“ *p ”**

Dynamic Memory Allocation

- **Syntax:**

```
# ptr var = dyn("type") //allocates memory for the specified type

# ptr a = dyn("type", initial_value) //allocates and initializes
```

- **Examples:**

```
# ptr a = dyn("number") //allocates an integer

# ptr a = dyn("number", 5) //allocates an integer and sets it to 5

# ptr a = dyn("letter", ‘A’) //allocates a character and sets it to‘A’
```

Structures

- **Declaration:**

```
struct struct_name var; //declares a structure variable 'var'
```

- **Accessing Fields:**
  - Use dot notation : `var.field`
- **Syntax:**

```
struct struct_name {
    number field1;
    letter field2;
};
```

- **Examples:**

```
struct Person{
      Number age;
      letter initial;
};

struct Person p;
P.age = 25;
p.initial = 'J';
print(p.age);   // prints 25
print(p.initial);  // prints J
```

Before Execution

Before running `out.exe`, the terminal shows messages like *initializations*, *declarations*, and *memory allocations*. These are informative logs from the compilation phase, helping learners understand how the code is processed – how variables are declared, memory is allocated, and IR is generated – before actual execution. It offers a quick insight into the compiler's backend workings.

The shown memory locations are **reference representations** with respect to `0x0000...`
While these values may vary based on system and runtime, they confirm space is reserved for each variable and it gets a fixed address in the actual stack.
