Laboratory 08: LLVM introduction

Setup

For this laboratory we will use the Clang compiler in order to compile our source code.

Packages needed:

- `llvm` (version 17)
- `clang` (version 17)
- `flex`

Optional tools:

- `make`
- `visual studio code` + C/C++ extension

Ubuntu installation example:

```
sudo apt install llvm-17 clang-17 make flex
```

If you have a Debian or Ubuntu distribution that does not have the packages for LLVM and Clang version 17 you can use the following commands to install them.

```
sudo apt install wget
wget https://apt.llvm.org/llvm.sh
chmod +x llvm.sh
sudo ./llvm.sh 17 all
```

Basic nodes

Generic node

This node will represent the base node that will be inherited by all the other nodes defined by us.

```
class GenericASTNode {
public:
    virtual ~GenericASTNode() = default;
    virtual void toString() {};
    virtual Value *codegen() = 0;
};
```

Number node

This node extends the `GenericASTNode` and is used to store constant integer numbers.

```
class NumberASTNode : public GenericASTNode {
  int Val;

public:
    NumberASTNode(int Val)
    {
        this->Val = Val;
    }
    void toString() { /* Implementation */ }
    Value* codegen() { /* Implementation */ }
};
```

Binary operations

This node also extends the `GenericASTNode` and is used to represent binary operations. The **Op** field is used to determine the operation and the **LHS** and **RHS** represent the left-hand side and the right-hand side nodes of the binary expression.

```
class BinaryExprAST : public GenericASTNode {
    char Op;
    unique_ptr<GenericASTNode> LHS, RHS;

public:
    BinaryExprAST(char Op, unique_ptr<GenericASTNode> LHS, unique_ptr<GenericASTNode> RHS)
    {
      // Implementation
    }
    void toString() { /* Implementation */ }
    Value* codegen() { /* Implementation */ }
};
```

Code generation

To implement the code generation for the binary operations nodes we will use the `Builder` object. This provides methods that we can use in order to insert instructions represented in the LLVM IR(Intermediate Representation). In this laboratory, we will work with some of the methods provided by the Builder object such as `CreateAdd` and `CreateSub`. These will take as parameters two LLVM Value objects and will return another Value object.

```
// Example for CreateAdd
Value X, Y;
Builder->CreateAdd(X, Y, "addtmp")
```

Numerical integer values can be represented in LLVM using the `ConstantInt` type.

```
// Define a 32-bit signed integer that stores the number 10
ConstantInt::get(*TheContext, APInt(32, 10, true));
```

Running the code

The commands can be run one after the other in order to obtain the executable

```
lex -o lexer.cpp lexer.l
clang++-17 -g -O3 main.cpp lexer.cpp \
  `llvm-config-17 --cxxflags --ldflags --system-libs --libs core` \
  -o main -ll
./main
```

Another method is to use the `make` command with the provided makefile. This will run the previous commands one after the other.

```
make
```

Visual Studio Code setup

When using Visual Studio Code with the C/C++ extension, the following paths must be included in the .vscode/c_cpp_properties.json file in the `"includePath"` array:

```
  "/usr/include/llvm-17",
  "/usr/include/llvm-c-17"
```