Laboratory 11: While statement and Variables

While statement

Grammar

```
Z       ::= E_AS | E_IF | E_WHILE.
E_IF    ::= if '(' E_AS ')' '{' E_AS '}' else '{' E_AS '}'.
E_WHILE ::= while '(' E_AS ')' '{' E_AS '}'.
E_AS    ::= E_MDR ('+' | '-' E_MDR)*.
E_MDR   ::= T ('*' | '/' | '%' T)*.
T       ::= i | '(' E_AS ')'.
```

While statement node

In the while node we will store the condition expression and the body of the while statement.

```
class WhileStatementAST : public GenericASTNode
{
    unique_ptr<GenericASTNode> Cond, Body;

public:
    WhileStatementAST(
        unique_ptr<GenericASTNode> Cond,
        unique_ptr<GenericASTNode> Body
    )
    {
        this->Cond = move(Cond);
        this->Body = move(Body);
    }

    void toString() { return; }
    Value *codegen() { return nullptr; }
};
```

Code generation

For the code generation of the while statement, we will define 3 basic blocks, `condition`, `body`, and `end`. The while statement will start by first jumping from the previous basic block, the `entry` block, to the `condition` block. In this block, we will create the comparison with the given `Cond` node and after that, we will perform a conditional jump to the `body` block if the condition is true, or we will jump to the `end` block if the condition is false.

In the body block, we will generate the code of the `Body` node and we will perform an unconditional jump back to the `cond` block.

Because we will also implement support for variables, when we reach the `end` block for the while statement we will return a constant number, for example 0. This is done because the codegen function of every node needs to return a value.

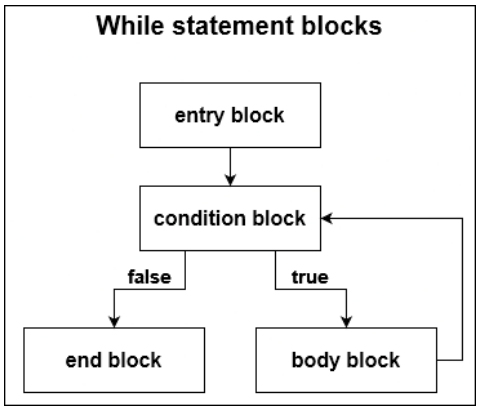

Figure 1: While statement blocks

Variables

Grammar

```
Z           ::= STATEMENTS
STATEMENTS  ::= STATEMENT (';' STATEMENTS)*
STATEMENT   ::=  E_AS | E_IF | E_WHILE | VAR_DECL | VAR_ASSIGN

VAR_DECL    ::= var identifier.
VAR_ASSIGN  ::= assign identifier '=' E_AS.

E_IF        ::= if '(' E_AS ')' '{' E_AS '}' else '{' E_AS '}'.
E_WHILE     ::= while '(' E_AS ')' '{' E_AS '}'.
E_AS        ::= E_MDR ('+' | '-' E_MDR)*.
E_MDR       ::= T ('*' | '/' | '%' T)*.
T           ::= i | '(' E_AS ')'  | identifier.
```

A new variable can be declared using the `var keyword` followed by the name of the variable. Example: `var myVar`

To assign a new value to a variable we will use the `assign` keyword followed by the name of the variable, the equal (`=`) character and, in the end, the value represented as an expression. Example: `assign myVar = 2 * 3 + 1`

Variables node

When working with variables we can perform 3 actions on them. We can **declare** a new variable, we can **assign** a values for a variable and we can **read** the value from a variable. We will have different AST nodes for all these actions.

The first AST node is the variable declaration node. In this node, we will keep the name of the variable that is defined. The constructor of this node will take as a parameter an array of chars and it will be automatically converted to a string variable using the assignation operation `=`.

```
class VariableDeclarationASTNode : public GenericASTNode
{
    string name;

public:
    VariableDeclarationASTNode(char name[])
```

```
    {
        this->name = name;
    }

    void toString() { return; }
    Value *codegen() { return nullptr; }
};
```

The second node is the variable read node. This node will be used in arithmetic operations alongside the number node and it will return the value stored inside the variable identified by the `name` field.

```
class VariableReadASTNode : public GenericASTNode
{
    string name;

public:
    VariableReadASTNode(char name[])
    {
        this->name = name;
    }

    void toString() { return; }
    Value *codegen() { return nullptr; }
};
```

The last AST node for variables is the variable assign node. This node will store the name of the variable that will be assigned a new value and the node that will contain the value. The `value` field can be any AST node.

```
class VariableAssignASTNode : public GenericASTNode
{
    string varName;
    unique_ptr<GenericASTNode> value;

public:
    VariableAssignASTNode(
        char varName[],
        unique_ptr<GenericASTNode> value
    )
    {
        this->varName = varName;
        this->value = move(value);
    }

    void toString() { return; }
    Value *codegen() { return nullptr; }
};
```

Code generation

In order to declare a new variable we will use the `CreateAlloca` provided by the builder object. This function has 3 parameters, the type of the variable, its size(if it is an array declaration), and the name of the variable. This function will return a pointer to an AllocaInst object. This is like a pointer to the memory where the variable is stored and it will be used when we want to read the value of the variable and when we want to assign a new value to that variable.

Example of an integer variable declaration with the name `myVar`:

```
AllocaInst* varPointer = Builder->CreateAlloca(
                            Type::getInt32Ty(*TheContext),
                            nullptr,
                            "myVar"
                         );
```

We will keep all the pointers to the allocated variables in a global map structure that will use the names of the variables as keys and will store the variable pointers as values.

```
map<std::string, AllocaInst *> allocatedVariables;
```

Accessing an element in the map is done using a string variable as the index: `allocatedVariables[varName]`

When we want to read the content of a variable we will use the `CreateLoad` function from the builder object. This needs as parameters the type of the variable, the pointer to the variable allocation, and the name of the variable (this can be omitted, but is recommended to define it because it helps when debugging the IR code generated).

Example of reading the value of the variable previously defined:

```
Value* x = Builder->CreateLoad(
                varPointer->getAllocatedType(),
                varPointer,
                "myVar"
           );
```

To assign a new value for a variable we will use the `CreateStore` function. This function needs 2 parameters, the value to store in the variable and a pointer to the variable.

An example of how to store the value 100 inside the variable defined previously:

```
Value* valToAssign = ConstantInt::get(
                        *TheContext,
                        APInt(32, 100, true)
                     );
Builder->CreateStore(valToAssign, varPointer);
```

Testing

Because we need multiple steps to test the variables, we will introduce a new node, the `StatementASTNode`.

```
class StatementASTNode : public GenericASTNode
{
    unique_ptr<GenericASTNode> node;
    unique_ptr<GenericASTNode> nextNode;

public:
    StatementASTNode(
        unique_ptr<GenericASTNode> node,
        unique_ptr<GenericASTNode> nextNode
    )
    {
        this->node = move(node);
        this->nextNode = move(nextNode);
    }

    void toString() { return; }
```

```
    Value *codegen() { return nullptr; }
};
```

This is used to link multiple nodes one after the other and is needed because when we work with variables we need to first declare the variable then assign a value to it and, in the end, work with the newly defined variable.

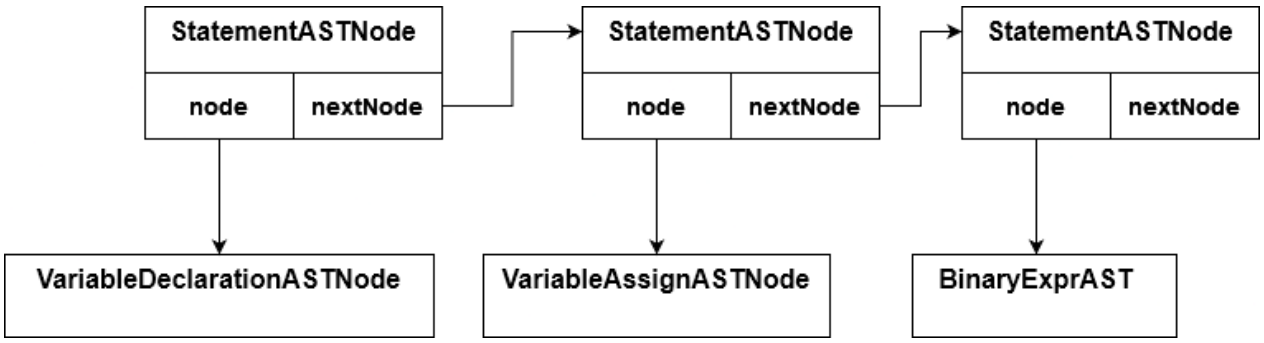

Figure 2: AST example of working with variables

The delimiter character between multiple statements is the semicolon character (`;`). For example `var x; assign x = 100; x + 1`.

When generating the code for a `StatementASTNode` we will first generate the code for the `node` field and after that, we will generate the code for the next node, if it exists. The last node in the linked list will have the `nextNode` field set to `nullptr`. The return value of any `StatementASTNode` codegen function will be the value of the last element in the entire linked list.