Assignment 5 - Code Generation

Ethan Benabou

ID: 40032543

1 - Analysis

1. Memory Allocation:

1. ☑ Allocate memory for basic types (integer, float).
2. ☑ Allocate memory for arrays of basic types.
3. ☑ Allocate memory for objects.
4. ☑ Allocate memory for arrays of objects.

2. Functions:

1. ☑ Branch to a function's code block, execute the code block, branch back to the calling function.
2. ☐ Pass parameters as local values to the function's code block.
3. ☐ Upon execution of a return statement, pass the return value back to the calling function.
4. ☐ Call to member functions that can use their object's data members.

3. Statements:

1. ☑ Assignment statement: assignment of the resulting value of an expression to a variable, independently of what is the expression to the right of the assignment operator.
2. ☑ Conditional statement: implementation of a branching mechanism.
3. ☑ Loop statement: implementation of a branching mechanism.
4. ☑ Input/output statement: execution of a keyboard input statement should result in the user being prompted for a value from the keyboard by the Moon program and assign this value to the parameter passed to the input statement. Execution of a console output statement should print to the Moon console the result of evaluating the expression passed as a parameter to the output statement

4. Aggregate data elements access:

1. ☑ For arrays of basic types (integer and float), access to an array's elements.
2. ☐ For arrays of objects, access to an array's element's data members.
3. ☐ For objects, access to members of basic types.
4. ☐ For objects, access to members of array or object types.

5. Expressions:

1. ☑ Computing the value of an entire complex expression.
2. ☐ Expression involving an array factor whose indexes are themselves expressions.
3. ☐ Expression involving an object factor referring to object members.

2 - Design

In this part of the project, we reuse the visitor infrastructure that we created in the previous phase of the project. Visitors are components that are applied to the our *intermediate representation (IR)* to implement further processing phases following the parse.

In the previous phase of the project, we extended our AST data structure to support accepting visitor objects. This included adding symbol table annotations to the AST nodes, as well as adding an `Accept` method to the AST node so that each node can accept visitors. We also created two visitors: a symbol table creation visitor, and a type checking visitor.

These two visitors implement the first two post-parse phases. In this part of the project, we will implement one more visitor - the codegen visitor.

This visitor will traverse the AST using *Depth-First Search (DFS)* and emit moon assembly code along the way as nodes are processed.

We started this part of the project with a `TagsBasedCodeGenVisitor`. The visitor emits simple tags-based assembly code where variables, including temporary variables used to process expressions, are stored in tagged memory areas reserved using the `res` moon instruction. This type of code is does not support more advanced code features such as stack-based function execution, but the advantage is that it is very simple to implement.

This is what the tags-based visitor looks like:

```
// A visitor that emits MOON assembly that uses a tags-based approach to
// variables and memory allocation.
//
// For data declarations, use the `dataOut` callback.
//
// For regular assembly instructions, use the `out` callback.
type TagsBasedCodeGenVisitor struct {
	// Assembly instructions will be printed to this callback, one line at a
	// time
	out func(string)

	// Memory reserved and tagged will be printed to this callback, one line at
	// a time
	dataOut func(string)

	// A prefix that is used internally for logging
	logPrefix string

	*tagPool
	*RegisterPool

	bufEmitted bool
}
```

The `Visit` method of this visitor is called on every node in the AST; it contains a switch statement that dispatches based on the type of the node being processed:

```go
func (v *TagsBasedCodeGenVisitor) Visit(node *token.ASTNode) {
	switch node.Type {
	case token.FINAL_PROG:
		v.prog(node)
	case token.FINAL_VAR_DECL:
		v.varDecl(node)
	case token.FINAL_WRITE:
		v.write(node)
	case token.FINAL_ARITH_EXPR:
		v.arithExpr(node)
	case token.FINAL_PLUS:
		v.plus(node)
	case token.FINAL_MINUS:
		v.minus(node)
	case token.FINAL_MULT:
		v.mult(node)
	case token.FINAL_DIV:
		v.div(node)
	case token.FINAL_INTNUM:
		v.intnum(node)
	case token.FINAL_ASSIGN:
		v.assign(node)
	case token.FINAL_FACTOR:
		v.factor(node)
	default:
		v.propagate(node)
	}
}
```

The node-processing methods of the visitor emit simplified, tags-based code. For example, here is an implementation of the `write` builtin function which writes an expression to the console:

```go
func (v *TagsBasedCodeGenVisitor) write(node *token.ASTNode) {
	v.propagate(node)
	buf := "wbuf"
	bufsize := 32
	v.emitDataf("%v	res	%v			%v Buffer for printing", buf, bufsize, token.MOON_COMMENT)
	top := v.tagPool.pop()
	v.headerComment(fmt.Sprintf("WRITE(%v)", top))
	reg := v.RegisterPool.ClaimAny()
	defer v.Free(reg)
	v.lw(reg, offR0(top))
	v.sw(off(-8, R14), reg, fmt.Sprintf("	%v %v arg1", token.MOON_COMMENT, INTSTR))
	v.addis(reg, R0, buf)
	v.sw(off(-12, R14), reg, fmt.Sprintf("	%v %v arg2", token.MOON_COMMENT, INTSTR))
	v.jl(R15, INTSTR, fmt.Sprintf("	%v Procedure call %v", token.MOON_COMMENT, INTSTR))
	v.sw(off(-8, R14), R13, fmt.Sprintf("	%v %v arg1", token.MOON_COMMENT, PUTSTR))
	v.jl(R15, PUTSTR, fmt.Sprintf("	%v Procedure call %v", token.MOON_COMMENT, PUTSTR))
}
```

The visitor makes use of two important secondary components:

1. `tagPool`

An object that manages all the tags used to reserve memory.

```go
// Used for keeping track of tags used in expressions. Package-private because
// tags are only used in this particular visitor implementation.
You, yesterday | 1 author (You)
type tagPool struct {
	tempc  int      // Count for temporary tags
	active []string // Stack of tags that are in active usage in the program
}
```

2. `RegisterPool`

A simple pool object that keeps track of the available register of the moon machine. When the visitor wants to use a register, it invokes its `RegisterPool` to decide which register will be used. The register pool is aware of the 16 register of the moon machine. It keeps `r0`, `r15`, and `r14` reserved for `0`, jump link, and the stack pointer respectively.

```go
// Starting register set
var registers = []string{
	// R0, // reserved

	R1, R2, R3,
	R4, R5, R6,
	R7, R8, R9,
	R10, R11, R12,

	// R13, R14, R15, // reserved
}

// A simple pooling machine for keeping track of MOON registers in active use
// during a program
You, yesterday | 1 author (You)
type RegisterPool struct {
	registers []string // Free registers
	active    []string // Used registers
}
```

3 - Use of Tools

- `tool.go`: written by me for this project. This is a partial parser-generator that generates a large portion of the code used for the parser. Specifically, it creates the parse table, first, and follow sets, as well as code and function stubs used for the semantic actions of our attribute grammar.