COMP 520 Milestone 3

Rajveer Gandhi, Archit Agnihotri, Dipanjan Dutta

Abstract

This report will talk about the language semantics of Golite and our chosen target language (Python) in order to sufficiently implement a fully functional code generator.

1 Introduction

For the last phase of our project, we have chosen to generate Python code with our compiler.

Advantages of our target language

- Python is dynamically typed, meaning we can leave object types to be handled at runtime.
- Lack of extra characters like semicolons and curly braces.
- We can assign multiple variables on a single line, like Golite
- All members of the group have working knowledge of Python

Disadvantages of our target language

- Python is strongly typed, so Golite expressions like 'a' + 5 will require a workaround.
- Python has multiple types of loops unlike Golite which has only one loop (for) which can take several forms.
- Golite is a smaller, simpler language as compared to Python.
- Python is a slow with respect to speed and execution.

2 Golite Semantics

2.1 Identifiers

The names of Golite identifiers must start with a letter or an underscore and may be followed by additional letters, digits and underscores. These names are case-sensitive.
Since Go does not allow unused local variables (results in a compilation error), we use blank identifiers as an alternative (denoted by _). These blank identifiers are mostly used when program syntax may require a use but the program logic doesn't.

Golite also has 28 keywords which cannot be used as variable names. These keywords are:

break	default	func	interface	select	case	defer	go
map	struct	chan	else	goto	package	switch	const
fallthrough	if	range	type	continue	for	import	return
var	print	println	append				

Golite also has 36 predeclared names for built-in constants, types and functions. These names are:

Constants:

true false iota nil

Types:

int	int8	int16	int32	int64	uint	uint8	uint16
uint32	uint64	uintptr	float32	float64	complex128	complex64	bool
byte	rune	string	error				

Functions:

make	len	cap	new	copy	close
delete	complex	real	imag	panic	recover

These names are not reserved, so we may use them as identifiers.

Identifiers in Python are defined in the same way as Golite, and blank identifiers are a viable way to use unused variables so we may not have to make any changes on this front.
On the other hand, Python has 33 keywords. However, they will not be re-used during codegen because of our mapping strategy. Basically, since we are using a large prefix, it will not conflict with any Python keywords.
Mapping strategy:

1. For all non-blank variable names in Golite, we will keep the base name and prefix it with __GOLITE__
2. Blank identifiers: because we're using a dynamic typically language, we don't any special handling of blank identifiers in Golite

2.2 If Statements

If statements in Golite may or may not have an else statement. They may include an optional **simple statement** - a short variable declaration, an increment or assignment statement or a function call that can be used to set a value before the conditional is tested and is scoped within this if statement and available for use within all branches.
Note that you don't need parentheses around conditions in Golite but braces are required.
Conditionals in If statements must be of the type **boolean**. If the condition is true, the lines of code between the braces is executed.

```
if num := 5; num < 0 {
        fmt.Println("negative")
} else if num < 10 {
        fmt.Println("1 digit")
} else {
        fmt.Println("multiple digits")
}
```

There may also be multiple *else if* statements wherein the condition is evaluated for truth from the top to bottom. Whichever if or else if's condition evaluates to true, the corresponding block of code is executed. If none of the conditions are true, then *else* block is executed. It is important in Golite that this else statement is preceded by a closing brace and not on a new line, otherwise a compile time error will occur.

In Python, there may be no optional simple statement before the condition. To map this, we will insert the short delcaration immediately above the "if" statement.

```
num = 5;
if num < 0:
        print "negative"
elif num < 10:
        print "1 digit"
else:
        print "multiple digits"
```

2.3 Loops

The *for* loop is the only loop statement in Golite. It can have multiple forms. One form is the 3-part loop which is illustrated like:

```
for initialization; condition; post {
        // zero or more statements
}
```

Parentheses are never used around the three-part loop. The braces are mandatory, and the opening brace must be on the same line as the post statement.
The optional *initialization* statement is executed before the loop starts. If it is present, it must be a simple statement. The *condition* is a boolean expression that is evaluated at the beginning of each iteration of the loop; if it evaluates to true, the statements controlled by the loop are executed. The *post* statement is executed after the body of the loop, then the condition is evaluated again. The loop ends when the condition becomes false.
If there is no *initialization* and no *post*, then both semicolons may also be omitted. In this case, the loop acts like a **while** loop.
If the condition is omitted entirely entirely in any of these forms, the loop is **infinite**.
These loops may be terminated by using a **break** or **return** statement.

In Python on the other hand, for loops follow a different syntax to iterate over a list or a string:

```
for item in sequence:
        statement(s)
```

Here each object in sequence is assigned to item one at a time and the following statement(s) are executed.

Mapping strategy:

1. We can redefine an infinite loop to be handled using a while loop:
   ```
   while true:
           statement(s)
   ```

2. Since Python has while loops, all *for* loops in Golite without *initialization* and *post* may be redefined in our code generator to output:
   ```
   while expression:
           statement(s)
   ```

3. For the threepart Golite *for* loop, we will have the *initialization* statement immediately preceding the start of the loop, and the post statement as the last statement inside the while loop:
   ```
   initialization
   while expression:
           statement(s)
           post
   ```

3 Current Progress

So far we have spent some time on fixing typechecking and symbol table. All valid programs (and most invalid programs) are now successfully passing through the typecheck mode. We have also spent time planning out the mapping strategies for most constructs of codegen in order to have a very clear understanding from the beginning.