Programming Project 8 – extra credit
Final Project Phase B
EE 312 Fall 2016

General: For our final project we will write our own little toy programming language. The language will have functions, loops, conditional statements, arithmetic and will even support recursion. For Phase B of the project, we're introducing real program statements (well, I suppose "real" here is a relative term). In addition to output and assignment statements from Phase A, we'll have conditional statements (if), loops (do) and functions (defun and call). We'll also introduce the concept of statement blocks, since loops, functions and conditions have "bodies" consisting of a sequence of statements (i.e., a block).

All the code you produce in this project must be original. You may not seek outside help (other than the assistance of the instructor, TA and authorized tutors). Please keep in mind that we will be using a plagiarism checker with this project.

The Blip do statement
While if statements are conceptually easier than loops, it's perhaps best to start with the do construct. At the very least, the do statement is more straightforward to parse than an if. The syntax in Blip for do looks like this:

```
do <expr>
        <statement1>
        <statement2>
        …
od
```

As you can see, the syntax consists of the do keyword followed by a loop condition expression, followed by a block of statements followed by the od keyword. As always in Blip, the spacing, indenting and new lines are all irrelevant. Please note that the block of statements can consist of zero or more statements. In Blip, a do loop with zero statements in the block is always silly (an infinite loop with no output). However, technically that is a legal loop.

The semantics of a do statement in Blip are essentially the same as a **while** loop in C. First evaluate the expression. If the expression is true (i.e., any integer value other than 0), then execute the block of statements. Then re-evaluate the expression and continue repeating the block of statements until the expression evaluates to false.

Parsing the do statement

Parsing do loops in Blip is not incredibly hard. Once you recognize the do keyword, you know the next thing in the input will be an expression, so that's easy (about the same as an output statement in Blip). The challenge comes when you start parsing the block of statements. Depending on how you wrote your parser during Phase A, you may have no problems at all with the statement block, but if your code is highly structured, you might find it a pain in the neck to deal with the od. The issue is that by the time you realize that the next token is an od, you've already read the token (duh). But, if the token is not od, then that token is the first token in a regular statement, by which time you may wish that you hadn't already read it. To help with this situation, the Input.cpp file now has a function "peek_next_token", which returns a C string representing the next token in the stream. To first approximation, peek_next_token works a lot like the sequence read_next_token(); next_token(). However, peek_next_token does not actually remove the token from the input stream. So, after you have "peeked" at the token, you still have to actually read the token to remove it. I used peek_next_token a lot in my solution. Some of you will find it useful, some of you will have no idea what it's supposed to be for. If it doesn't appear really, really useful to you, then your code probably doesn't benefit from it – it will be obvious how to use this function if it's useful to you. And if it's not useful, please don't feel obligated to use it.

Delaying and Repeating execution

The big change from Phase A to Phase B is that when you implement Phase B you're forced to parse and save Blip statements. In Phase A, you could parse, execute and throw away each statement as you executed it. Statements could only be executed once, so there was no need to save them. However, in Phase B we have loops that force us to repeatedly execute the same statement(s). Obviously, the statements appear only once in the input stream even though we're executing them repeatedly in the loop. So, you're forced to save those statements somehow and then re-execute them later. We strongly recommend that you use a parse tree to store your statements. In fact, we recommend that you use a parse tree to store ALL your statements, even those statements that are not inside a loop. Read/parse with one (recursive) function, and then execute with another (recursive) function. If you used a parse tree in Phase A, you're probably in good shape to extend that parse tree for Phase B. Please note that you will probably have two different types of parse trees: trees of statements and trees of expressions. A do statement is a perfect example. The condition for the do is an expression. If you have an expression parse tree, that's a great way to store that expression. The body of the loop is a block (hint, a vector) of statements. Each of those statements can be any Blip statement, including do or if statements! So, a vector of statement parse trees is the right way to store the body of a do loop.

If statements

Once you've figured out how to do a loop (no pun intended), if statements are pretty easy. The syntax of a basic if statement is as follows

```
if <expr>
        <stmt>
        <stmt>
fi
```

where, as before, <expr> is an expression and there is a block of statements in between the start of the if and the end of the if. Note that the end of the if is marked with "fi" (if spelled backwards). The semantics of a basic if statement is the same as in C. If expr evaluates to any integer other than zero, then all the statements in the body are executed. If expr evaluates to 0, then none of the statements in the body are executed.

The else statement

Just as in C, if statements can have an optional else attached. The syntax for an if-then-else statement in Blip is as follows

```
if  <expr>
        <stmt>
        <stmt>
else
        <stmt>
        <stmt>
fi
```

The semantics of an if-then-else in Blip are just like those in C. If expr evaluates to true then execute all the statements inside the first statement block, but none of the statements in the else block. If expr evaluates to false, then evaluate none of the statements in the first block, but evaluate all the statements in the else block. Please keep in mind that the else construct will not appear on every if statement. Some if statements will look like this

```
if < x 0
        set x ~ x
fi
```

and some if statements will look like this

```
if < x 0
        text negative
else
        text non-negative
fi
```

And that's all there is for Project 8

The rest of this document describes additions to Blip. Please note that the effort implied by the requirements below (i.e., how much time it takes to finish everything) is outside the bounds of what's reasonable for a programming project. So, it really is extra credit, and I would not expect very many (if any) students to undertake the entire project. With that caveat, we will test the following requirements, including the quality of your solution (e.g., testing for memory leaks) and assign up to five additional extra credit points for extraordinarily good solutions.

Functions (oh my)

The complicated part of Phase B starts with functions. Adding functions to Blip means adding three new commands, two statements and one expression type. But the hard part

of functions is implementing their semantics. We suggested you approach functions as follows.

1. Implement basic functions with no parameters. These functions, when called, always run to the end of their function bodies – i.e., the return statement will always be the last line in the function.
2. Implement functions with parameters. Note that parameters require that you have a new "local" scope for your variables, so if there's a variable "x" defined and then there's a parameter "x", the parameter and the original variable are different. We'll approximate the rules that C uses for variable scoping. Once you implement parameters, implementing local variables are relatively easy.
3. Implement return statements that happen in the middle of your function. A return statement can appear anywhere inside a function (or even anywhere inside the main program), including inside loops or if statements. When the return statement is executed, the function immediately stops executing and control returns to the calling function.

You can actually swap the order you implement #2 and #3 above, depending on how you're feeling. I just strongly recommend that you do #1 to completion, and then pick either #2 or #3 to work on next. Don't try to do 2 and 3 simultaneously – build the project incrementally!

Basic Functions and defun/nufed

A function can be defined anywhere, at any time. In fact, we can nest functions in functions in Blip (you can't do that in C). When you define a function (with the defun keyword), you create a list of statements (the function body) that are not executed yet, but will be executed when the function is called (with the call expression). The syntax for a function definition is as follows:

```
defun <name> params <p1> <p2> smarap
        <stmt1>
        <stmt2>
nufed
```

For example, here's a simple function

```
defun sayHello params smarap
        text "Hello World\n"
nufed
```

In this example, a new function called "sayHello" is created. The function has zero parameters (there is nothing between the params and smarap markers), and one statement (there is one statement between the parameter list and the nufed marker). When this function is parsed, you should build a parse tree for it. As a hint, since my implementation always parses and then executes statements, my defun statements are also both parsed and executed. However, "executing" a defun statement does not execute the body. Rather executing a defun statement simply adds the function to the symbol table –

once my defun has been executed, the symbol "sayHello" is bound to the function. There are many other ways to implement the semantics, but however you do it, you must not execute the body of the function until it is actually called.

Call expressions

The only way to invoke a function in Blip is to use a call expression. This is actually similar to what C does. However, in Blip, every function has a return value. If you don't include a return statement in your Blip function, then the value 0 must be returned by that function (when it ends). Most importantly, the return value cannot be ignored by the caller – C allows the function return values to be ignored. So, calling the sayHello function requires that we do something with the returned value (zero for that example since there's no explicit return statement).

```
var not_used call sayHello args sgra
```

Here I've used a var statement. I have a bogus variable "not_used" and I'm initializing this variable to the value returned by sayHello. I use this technique extensively in my Blip programs to simulate "void" functions. Anyway, since Blip functions always return a value and since that value can never be ignored, our task to execute them is relatively easy. We have a new expression type "call". The call expression invokes the function using the arguments from the argument list and then evaluates to the value returned by the function. The syntax for a call expression is:

```
call <funName> args <a1> <a2> sgra
```

Note that the argument list is marked with "args" and "sgra". For functions like sayHello that have no arguments, then the list is empty. Otherwise, there are arbitrary expressions inside the list – <a1>, <a2>, etc. can be any expression, such as "+ x y" or even call expressions! Note that there are no commas or other punctuation between the argument expressions.

If you implement basic defun statements and call expressions, you can write simple Blip functions. I strongly encourage you to do this much (and test/debug it) before moving on to the more complex aspects of functions.

Params and variable scoping

Blip functions can have an arbitrary number of parameters. The parameter list is marked with params/smarap in the defun statement. The corresponding set of arguments is marked with args/sgra. Just like in C, the first argument is bound to the first parameter, the second argument is bound to the second parameter and so forth. The first step in implementing parameters is to modify your call expression so that, when it is executed, each of the arguments is evaluated and the value is assigned to a variable with the name of the parameter. Consider the following Blip program

```
var x 1
defun fun params x smarap
```

```
        set x + x 1
        return x
nufed

var y call fun args x sgra
text "x is " output x text \n
text "y is " output y text \n
```

This program has a global variable x and a parameter to the function also called x. If you have only one symbol table entry for the variable x, the program will (probably) print out "2" for both x and y. That's actually a pretty good amount of progress towards functions, but we still have a long way to go. We really want the parameter x to be a "local variable" to the function fun. That is, there should be two different variables both named x. Within fun, x refers to the parameter x, but the output statements from the main program refer to the global variable x. In other words, the correct output for the program above is

```
x is 1
y is 2
```

There are many possible implementations of variable scoping. You're free to choose any implementation strategy that you would like. However, my personal favorite involves using multiple symbol tables. When the parameter x comes into existence, it is placed into its own symbol table – the local variables for function fun. When the function returns, its local-variable symbol table is completely erased. However you decide to implement local variables, the correct behavior in Blip is defined as follows

- Global statements (statements that are outside of any function) have access only to variables that are also outside of any function. Variables defined outside of any function are known as "global variables"
- Statements inside a function have access to both local variables and global variables. Local variables are variables defined with var statements that appear within the function's body, or parameters that appear within the statement's parameter list. In Blip, parameters and local variables are the same scope. Also, new scopes are only created for functions.
    - If a variable appears in an expression inside a function body, and that variable is defined as a local variable, then the value of the local variable must be used for the expression. It does not matter if the variable is also a global variable.
    - If a variable is used as the target of a var statement inside a function, then a new "local" variable is being created. If there is a global variable with the same name, then the new local variable hides the global. If there is a parameter with the same name, or if there is a previously executed var statement in the same function with the same name, then a warning is generated (same as for Phase A warnings).
        - Note that parameters and locals are considered the same scope.

        - Note that only functions create new scopes – if and do blocks do not create new scopes (this is different from C).
    - If a variable is used as the target of a set statement, and the variable has previously been set by an executed var statement from the same function, then the local variable is used, even if a global variable exists by the same name.
    - If a variable is used as the target of a set statement, and the function has not previously executed a var statement with the same name, then the set statement assigns to the global variable. If there is no global variable, then a warning is generated (same as described for Phase A).
    - If a function is nested inside other functions, then statements and expressions within that function can access only the local variables for that specific function or globals. A function cannot access local variables defined by an outer function.

A couple of quick examples:

```
var x 1 // global variable x
var y 1 // global variable y

defun out_fun params x smarap
        set x 42 // assigns to param, not global variable x
        var x 10 // generates a warning, sets param x to 10
        set y 10 // changes global variable y to 10
        defun in_fun params smarap // an inner function, oh my!
                set y 3 // assigns 3 to global variable y
                set x 1 // assigns to GLOBAL variable x
        nufed
        var not_used call in_fun args sgra
nufed

var not_used call out_fun args y sgra
// using y as the argument to out_fun doesn't affect any of the comments above
```

Note how there are exactly two variables named x, one is global and one is local to out_fun. The inner function in_fun has access to only the global, while the outer function out_fun only has access to the local. Another example, this one demonstrating something subtle:

```
var x 1
defun foo params smarap
        if ! x
                var x 42
        fi
        output x // prints GLOBAL variable x
nufed
```

```
var not_used call foo args sgra
```

In Blip, var statements take effect only when they are executed, not when they are parsed. This actually makes our implementation easier, but it's certainly different from C. So, since the global variable x is set initially to 1, the if statement evaluates to false. That means the var statement is NOT executed, and no local variable x is ever created. Thus this program prints 1 when the output statement is reached.

Early return statements

Blip function can have multiple return statements. If there are zero return statements, then the function returns the value 0. Otherwise, the first return statement that is executed supplies the returned value. Additionally, as soon as a function executes a return statement, the function must immediately stop executing and return control to its caller. To make things consistent, return statements can also appear in the "main program" (i.e., the Blip statements that appear outside of any functions constitute the main program). The first of these return statements that are executed forces the Blip program to immediately stop. The returned value is evaluated, but is not used for anything. For example:

```
text "Hello World\n"
return + 42 42
text "never seen\n"
```

In this Blip program, the first message is displayed but then the return statement is executed. The return statement evaluates its expression (the result is 84) and then the Blip program terminates without printing the second message. The value 84 is simply discarded.

How you go about implementing the return statement depends entirely on how you structured your program – how you execute statements and functions. I have no specific advice. There are no fancy data structures or complex semantics to worry about. Please do keep in mind that we prefer clean and well-designed code over sloppy code and that your style is part of your grade for this project.

Memory leaks are "free" for this Project, and you need not worry about them.

Good luck and have fun!

CHECKLIST – Did you remember to:

- ☐ Re-read the requirements after you finished your program to ensure that you meet all of them?
- ☐ Make sure that your program passes all the testcases?
- ☐ Make up your own testcases?
- ☐ Upload your solution (all file) to Canvas in .zip or gzip format?
- ☐ Download your uploaded solution into a fresh directory and re-run all testcases?