Caige Middaugh
11/12/2021
Programming Language

Report Part 2

Changes:

- As far as changes to the Grammar goes, I added the ability for the get command to do multiple different IDs. The user can also enter a custom message that would be printed to the terminal. This change is done for the <input> symbol.
    - Example before change: get a
    - Example after change: get a{"Value for a: "}, b{"Value for b: "}, c

  As you can see above the {""} is optional.
- I also added "int" and "str" to be in front of every assignment so that type checking would be possible.
- I also added "(int)" and "(str)" for the return statement to affectively cast the functions return type. I may go back for part 3 and allow recursion which then I would have to make the return type specified in the header of the definition for the function.
- I also changed how I handled the values. I have <value> which does not care about type and just wants to know a value is there. Such as a raw integer or string, or an ID which may be a function call, variable, or array. Before this caused my grammars to not be LL grammars, and I had to reimagine how it's done to both keep the types important and still have an option where I can include both integers and strings. I added <raw-int>, <raw-nonint>, and <ambiguous> to make this possible. Then I have <int-value>, <nonint-value> and <value>. This allows me to easily define types of IDs and type check.

Additions:

- **Scope, type checking, syntax validation:** printf validation of its (%s, and %d) versus number of arguments given. Note that some stuff that's valid under the grammar may not be valid in actuality due to parser rules such as an ID has not been defined previously, or type is different, or functions cannot be nested.
- **Symbol Table:** Most of the challenged with this project was the symbol table. How to design it such that it supports static scoping and type checking. Designing a symbol table that isn't so complex that it's impossible to understand.

Scope

The approach for scope in this project is static scoping. I limited functions to only have the global scope as its parent. The parser also ensures nested functions cannot happen to keep consistency, and to reduce confusion surrounding scope. It is possible technically to have nested function with the grammars I provide, and I can implement them. However, to keep the function's scope rules it may confuse the user, thus I made the decision to not allow nested functions.

I am not sure how scope is usually maintained but I chose to take an approach similar to how the file system handles hierarchy. With a string that contains its parents and children such as:

"globe/main/For1/While1" would be a while loop in a for loop in the main scope of the program. The downside to this is that the function table's key is the scope that means that "globe",

“globe/main”, “globe/main/For1”, and “globe/main/For1/While1” are all entries in the Function table. I have not had the time to reimagine this process but this is one of the better solutions I thought of to maintain scope. If we are in the while loop and want to verify some ID has been defined or check its type, we then have to look up for all those functions mentioned in the function table. We then get each of their variable tables and check if it exists. This process isn’t definitely not slow because both the variable and function tables are dictionaries. However, can take up a lot of memory and ultimately there probably exists a more efficient solution for memory and speed.

The worst case to loop up if an ID exists is if it's in the “globe” scope which scales with the size of functions for its independent scope. Generally speaking, I don’t think there would be a scope as big as 100 functions, but if that is the case this should still be a rather decently fast solution.

NOTE: The function table and scope pretty much treats For, and while loops as a function. Loops are the only “functions” that can actually get nested because of my rule to not allow actual functions to be nested.

Type Checking

This process was surprisingly easy after I was able to get a working symbol table. The grammars I defined made the process extremely simple to define if a variable should be a string or integer. The main grey area was function parameters, which I ended up setting the type as “undefined” then once it's used the type will change to what it's expected to be. However, I should point out that as of right now my parser does not check if the correct number of parameters are given.

I was personally shocked with how decent the type checking was. If you do:

int d = b + c, then b and c must also be integers which I did not have to code for. Due to how my grammars were set up and how I set up type checking that was an inherited feature. Similar with int and string casting.

Symbol Table

The symbol table was the hardest part for this project for me because of scoping. The function symbol table object contains the method to search for an ID within a given scope. The findExistingInstance method will start with given scope and access the variable table for that scope. Then pop off the top name of the scope and now it’s the parent scope that the method then searched the variable table for the parent, and so on. It will then terminate with the globe scope.

The main issue that I ran into is how to set up the symbol table. I tried having just one Variable symbol table, but that wasn’t sufficient for managing scope with how I wanted to do it. Overall, there was shockingly a lot of memory management I had to do considering its python. I was getting weird memory error with accessing the variable tables that were stored inside the function table. I even got a weird error when trying to define variables which all was due to how I defined the variable symbol table.

With for loops the control variable is defined within the scope of the for loop regardless if it's defined in a previous scope or not. This will force the for loop to use that control variable by default because of how my function table searches through scope. I search the scope it's in first then its parent, then grandparents and so on. In some perspectives this can be considered a bug, but I meant it as a feature.

Overall, I did not change as much about my language in this part because I chose to add Functions. Functions are a big task to tackle because I have to then tackle scope, and if I want to return variable then I should tackle some form of type checking. Since these tasks are all challenging and took a long time to develop, I did not make a whole lot of change to my language. The custom message for 'get' will be fun to implement in the interpreter. I made a lot of changes to the grammar for the language also to make sure that its decently thorough and has types built into it.

GRAMMAR:

```
    <prog>  -> <stmt_list>

<stmt_list> -> ∈

            | <stmt>";"<stmt_list>

    <stmt>  -> <print>

            | <printf> // For the people obsessed with C

            | <input>

            | <assign> // arrays also

            | <if>

            | <while>

            | <for>

            | <function-def>

            | <function-call>

    <print> -> "print" "(" <prnt-arg> ")"

<prnt-arg>  -> ∈

            | <value> <prnt-expr>

<prnt-expr> -> ∈

            | <str-operation> <prnt-expr>

            | "," <value> <prnt-expr> // ONLY ALLOW INT+STRING IN PRINT STATEMENT. Will
concat. Leave out of <str-operation> to make that reusable.

    <printf>-> "printf" "(" <prntf-arg> ")"

<prntf-arg> -> STRING

            | FSTRING <prntf-expr> // Regular expression mandates that %d %s are in it.
    <prntf-expr>-> ∈
                | "," <value> <prntf-expr>
        <input> -> "get" ID <input-mult>
<input-mult>    -> ∈
                | "{" STRING "}" <input-mult>
                | "," ID <input-mult>
        <assign>-> "int" ID "=" <assign-intopt>
                | "str" ID "=" <assign-stropt>
        <if>    -> "if" <int-expr> "then" <stmt_list> <if-options> "end"
<if-options>    -> ∈
                | "elif" <int-expr> "then" <stmt_list> <if-options>
                | "else" <stmt_list>
        <while> -> "while" <int-expr> "do" <stmt_list> "end"
        <for>   -> "for" <assign> ";" <int-expr> ";" <int-expr> "do" <stmt_list> "end"
<assign-intopt> -> <int-expr>
                | "[" <int-arg> "]"
<assign-stropt> -> <str-expr>
                | "[" <str-arg> "]"
    <int-arg>   -> ∈
                | <int-expr> <int-arg'>
    <int-arg'>  -> ∈
                |"," <int-expr> <int-arg'>
    <str-arg>   -> ∈
                | <str-expr> <str-arg'>
    <str-arg'>  -> ∈
                | "," <str-expr> <str-arg'>
<function-call> -> ID "(" <params> ")"
<call-params>   -> ∈
                | <value> <params'>
```

```
<call-params'>  -> ∈
                | "," <value> <params'>
<function-def>  -> "def" ID "(" <params> ")" "begin" <stmt_list> <return> "end"
        <params>-> ∈
                | ID <params'>
    <params'>   -> ∈
                | "," ID <params'>
    <str-expr>  -> <nonint-value> <str-operation>
    <int-expr>  -> <n_expr> <logic> //Note function call is included
        <n_expr>-> <term> <first-degree>
        <term>  -> <factor> <second-degree>
<str-operation> -> ∈
                | ADD <str-operation'> <str-operation>
<str-operation'>-> <nonint-value>
    <compare>   -> ∈
                | ">" <int-value>
                | ">=" <int-value>
                | "<" <int-value>
                | "<=" <int-value>
                | "==" <int-value>
                | "!=" <int-value>
        <logic> -> ∈
                | AND <n_expr> // "and"
                | OR <n_expr> // "or"
<first-degree>  -> ∈
                | ADD <n_expr>// INTs will be tokenized as only positive ints
                | SUBTRACT <n_expr>// interpreter may have to differentiate "-" ID and
Subtract
<second-degree> -> ∈
```

```
                | MULTIPLY <term>//"*"
                | DIVIDE <term>// "/"
                | MODULO <term>// "%"
        <factor>-> <int-value> <compare>
<ID-operation>  -> ∈
                | "[" <int-value> "]"
                | "(" <params> ")"
        <value> -> <raw-int>
                | <raw-nonint>
                | <ambiguous>
<nonint-value>  -> <raw-nonint>
                | <ambiguous>
    <int-value> -> <raw-int>
                | SUBTRACT <int-value>
                | ADD <int-value>
                | <ambiguous>
    <raw-int>   -> INT
                | "int" "(" <value> ")"
    <raw-nonint>-> STRING
                | "str" "(" <value> ")"
    <ambiguous> -> ID <ID-operation>
        <return>-> ∈
                | "return" <return-option>
<return-cast>   -> "(" <return-option>
<return-option  -> "str" ")" <str-expr>
                | "int" ")" <int-expr>
```