Jonathan Smoley
CPSC 326
May 12, 2023

Final Project – Report

Extension – Dictionary (Map)

The dictionary implemented is a collection of key-value pairs. The keys do no repeat and can only be of type string or int. The values can be any primitive type in MyPL (string, int, bool, char, double/float). Dictionaries in MyPL are allocated on a heap. As such, they are created using the *new* keyword. Using the key, one can add a pair to a dictionary or obtain a value from an existing key-value pair in the dictionary. Also, common dictionary methods include contains, keys, values, insert, empty, etc. In MyPL dictionaries only have methods *keys* and *values*. These return arrays of their respective parts of each pair in the dictionary.

```
dict string int test(dict string int p2s) {
  p2s["second"] = 25
  return p2s
}

void main() {
  # instantiation
  dict string int ps = new dict{string, int}

  # dict insert
  ps["first"] = 0

  # dict update
  ps["first"] = 12

  # dict access
  int x = ps["first"]

  # output dict state
  print(ps["first"])
  print(x)

  # output new dict from function
  dict string int ndict = test(ps)
  print(ndict["second"])
}
```

Modifications

In adding a new data structure to MyPl, at least one thing in every part of the suite had to be changed.

With the lexer, I added a reserved word ‘dict’. This aided in explicit creation of dictionary objects as they are stored similarly to structs and arrays (on the heap).

Jonathan Smoley
CPSC 326
May 12, 2023

For the simple parser, I checked for the 'dict' token type in both data_type() and new_rvalue() in the case of instantiation. However, it turns out that I did not have to update much here for parsing set and get operations on dictionaries as this follows what was implemented for arrays in MyPL.

In the AST parser, the basic parser file is not only what I had to change. The AST class had to be modified to store two data types: one for the key and another for the value. Similarly, the NewRValue and VarRef were updated to store expressions, or the result of expressions used to get and set pairs in dictionaries. After this was done, I had to go into the AST parser file and modify the same new_rvalue, expression, and data_type functions to reflect parsing a dictionary type.

The semantic checker is where I faced most of my difficulties. Here, I first added built-in names to the built-ins set for keys and values methods and implemented type checking for those in CallExpr. Following this, I had to update type checking in the following visitors: vardeclstmt, assignstmt, expr, newrvalue, varrvalue. This is so the types of the keys and values being inserted match the dictionary typing scheme.

After adding OpCodes for several operations that may need to happen for dictionaries, I was able to begin changing the VM. I started with the keys and values built-ins, making sure return lists of just one set of values in the dictionary. I then moved to allocating a dictionary, where the struggle was to decide what needed to be pushed to the stack, if anything, before storing a dictionary on the heap. I next finished up my changed by adding operations for adding pairs, updating values, and accessing values.

Wrapping up changes with code generation, I did not get very far into considering what to do here. But what I did discover is that the newrvalue visitor needed to visit each expression used to add types for the dictionary. I assume the varrvalue visitor will also need to be updated for parsing dictionary expressions for getting values from the collection. That said, the assignment visitor will need to be updated for getting values and setting keys to map to values correctly.

Progress

I was able to successfully make meaningful changes up to type checking the values on the right and left sides of statements involving dictionaries. That is, I got caught up in the semantic checker modifications when I went to compare the types being instantiated for keys and values in a dictionary.

Despite this issue, I continued into the VM and Code Generator code to make what changes I could. If there were more time, I would require some external help to decide the best way to type check in cases where multiple types need to be parsed. After deciding upon a suitable structure to store this information in, I would move on to further

implementing VM operations that signify needed parts of dictionaries such as grabbing the key from an index verse grabbing a value from the same index (where this index is done with the key).

Testing

In testing this extension, I started by writing new unit tests covering each component of the MyPL suite that was written this semester. Everything from lexical analysis to code generation has tests for dictionaries. Within these tests, the built-in keys and values methods are tested as well. Apart from these tests, there were a few use cases for dictionaries that needed to be tested.

For unit testing, I tried to stick to basic type creation, access, and update methods. I tested these through each step of the MyPL compilation process. With the semantic checker and VM, however, I decided to also check that the keys and values functions were working as well.

```
TEST(SemanticCheckerTests, StaticLoopDictBuiltIns)
{
  stringstream in(build_string({
      "void main() {",
      "  dict string int kvs = new dict{string, int}",
      "  kvs[\"foo\"] = 5",
      "  kvs[\"bar\"] = 4",
      "  for(int i = 0; i < len(keys(kvs)); i = i + 1) {",
      "     print(keys(kvs)[i])",
      "  }",
      "  for(int i = 0; i < len(values(kvs)); i = i + 1) {"
      "     print(values(kvs)[i])",
      "  }",
      "}"
    }));
  SemanticChecker checker;
  ASTParser(Lexer(in)).parse().accept(checker);
}
```

Jonathan Smoley
CPSC 326
May 12, 2023

```
TEST(CodeGenTests, DictUpdatePair)
{
  stringstream in(build_string({
    "void main() {",
    "  dict string int kvs = new dict{string, int}",
    "  kvs[\"first\"] = 42",
    "  print(kvs[\"first\"])",
    "  kvs[\"first\"] = 0",
    "  print(kvs[\"first\"])",
    "}"
  }));
  VM vm;
  CodeGenerator generator(vm);
  ASTParser(Lexer(in)).parse().accept(generator);
  stringstream out;
  change_cout(out);
  vm.run();
  EXPECT_EQ("420", out.str());
  restore_cout();
}
```

Regarding alternative testing, I made a simple use case file that demonstrates why dictionaries are useful to include in a programming language.

Execution

MyPL still builds and runs the same as before. I simply added a data structure and did not need to create a specific case to run dictionaries apart from the rest of MyPL. These steps include: *'make'* to build the project, *'./mypl +\- <flag> <file>'* to run the project. However, I modified the source *mypl* file so that there must be either a flag, file, or both to properly execute. I did not see any value in running *'./mypl'* alone if the source file works.