Searching for Reserved Words

An important task of the scanner is to distinguish between user-defined identifiers and reserved words. Since identifiers and reserved words account for a large percentage of the symbols in a typical program, it is important that this task be implemented efficiently. Indeed, an analysis of the collection of correct CPRL test programs shows that almost half the symbols fall into one of these two categories.

The basic idea is for the scanner to accumulate the characters for an identifier or reserved word into a string and then use a look-up mechanism to determine if the string is one of the reserved words. We encapsulate the details of the look-up mechanism in a method with the following signature.

```
public Symbol getIdentifierSymbol(String idString)
```

The method will return either `Symbol.identifier` or one of the reserved word symbols – `Symbol.IntegerRW`, `Symbol.ifRW`, `Symbol.loopRW`, etc.

This handout will explore several algorithms for implementing this method, and it will include a benchmark for the performance of each algorithm. Details of the implementations and benchmark analysis are given below, but the results displayed in the following table show that the performance differences can be significant.

Search Algorithm	Benchmark Time (in seconds)	Standard Deviation
Sequential 1	19.57	0.1750
Sequential 2	6.36	0.0796
Binary	2.87	0.1076
By Length	2.07	0.0544
By First Character	1.48	0.0351
Gperf Hash	1.63	0.0442
HashMap	1.08	0.0264

Benchmarking the Search Algorithms

The general approach to benchmarking was to use each algorithm to perform numerous searches for identifiers and reserved words. The times for each algorithm were computed simply by reading the system clock before and after performing the searches and then subtracting. While the timings are subject to some background noise from other processes running in the background, every effort was made to minimize this effect, and multiple runs of the benchmarks produce comparable times. The results above are obtained by running the timed searches six times and averaging the results.

Since not all reserved words are used with the same frequency, an analysis of the correct CPRL test programs was performed to determine a rough order of magnitude. For example, the reserve word `end` was used a lot more than the reserved word `loop`, and `loop` was used a lot more than the reserved word `false`. Among predefined type names, `Integer` was by far the most used. Also, recall that some reserved words are not yet used in CPRL but are reserved for possible future use. In addition, user-defined identifiers occurred almost as frequently as all reserved words combined.

Based on this analysis, a file was created to control the number of times each reserved word or identifier would be searched as part of the benchmarking process. Each line of the file contained

either a reserved word or an identifier followed by the number of times it was to be searched. Below is an outline of the file.

```
Boolean   1000000
Char      1000000
Integer   4000000
String    500000
and       1000000
array     1000000
begin     4000000
...
type      1000000
var       4000000
when      1000000
while     2000000
write     2000000
writeln   6000000
i         15000000
average   10000000
value1    10000000
thisIsAVeryLongIdentifier 10000000
```

The data in the list was used to initialize an array of class `TestId`, where `TestId` is defined as a private, static, nested class of the benchmarking test class as follows.

```
private static class TestId
  {
    public String idString;
    public int    reps;

    public TestId(String idString, int reps)
      {
        this.idString = idString;
        this.reps     = reps;
      }
  }
```

With this structure, the timed performance of each search algorithm was implemented as shown below.

```
timer.start();
for (int i = 0;  i < testIds.length;  ++i)
  {
    for (int j = 0;  j < testIds[i].reps;  ++j)
        searchAlgorithm.getIdentifierSymbol(testIds[i].idString);
  }
timer.stop();
reportResult(methodName, timer.getElapsedTimeInSeconds());
```

Note that this code simply ignores the value returned by the `getIdentifierSymbol()` method.

Sequential Search 1

This is the least efficient approach to searching for reserved words and is useful only for comparison purposes. This approach uses an array list of reserved words and a sequential search through the array list. Its only advantage is that it is very easy to implement.

The code to initialize the list is very simple.

```
ArrayList<Symbol> reservedWords = new ArrayList<>();
for (Symbol symbol : Symbol.values())
  {
    if (symbol.isReservedWord())
        reservedWords.add(symbol);
  }
```

We can visualize the contents of the array list as follows.

```
{
  BooleanRW,
  CharRW,
  IntegerRW,
  StringRW,
  andRW,
  arrayRW,
  beginRW,
  ...
  typeRW,
  varRW,
  whenRW,
  whileRW,
  writeRW,
  writelnRW
}
```

Using this array list, the search method is implemented as shown below.

```
public Symbol getIdentifierSymbol(String idString)
  {
    for (int i = 0;  i < reservedWords.size();  ++i)
      {
        if (idString.equals(reservedWords.get(i).toString()))
            return reservedWords.get(i);
      }

    return Symbol.identifier;
  }
```

Note the use of the `toString()` method for retrieving the reserved word spelling.

Sequential Search 2

Similar to the approach above, this approach also uses a sequential search, so we might expect similar performance. But for this approach we make a couple of changes that greatly improve performance. First, instead of using an array list we use just a plain array. And second, instead of storing simply the symbol and calling `toString()` to get the spelling, we store a pair that has both the spelling (string) and the symbol. We could have used the generic class `Pair` defined in package

`javafx.util`, but since JavaFX is no longer shipped with Java and since this is a very simple class, we elected to just implement it directly as a private, static, nested class.

```
private static class StrSymPair
  {
    public String rwString;
    public Symbol rwSymbol;

    public StrSymPair(String rwString, Symbol rwSymbol)
      {
        this.rwString = rwString;
        this.rwSymbol = rwSymbol;
      }
  }
```

Note that the fields are public to avoid "`getX()`" method calls, but that is acceptable since class is only used in limited ways by the search methods. In addition, it is declared as private within the classes containing the search algorithm, thereby hiding it from other classes. This class will be also be used by some of the other search methods.

This approach requires only slightly more work initializing the array, and after initialization we can visualize the contents of the array of pairs as follows.

```
{
  ("Boolean", BooleanRW),
  ("Char", CharRW),
  ("Integer", IntegerRW),
  ("String", StringRW),
  ("and", andRW),
  ("array", arrayRW),
  ("begin", beginRW),
  ...
  ("type", typeRW),
  ("var", varRW),
  ("when", whenRW),
  ("while", whileRW),
  ("write", writeRW),
  ("writeln", writelnRW)
}
```

Using this array of `(String, Symbol)` pairs, the search method is implemented as shown below.

```
    public Symbol getIdentifierSymbol(String idString)
      {
        for (int i = 0;  i < reservedWordPairs.length;  ++i)
          {
            if (idString.equals(reservedWordPairs[i].rwString))
                return reservedWordPairs[i].rwSymbol;
          }

        return Symbol.identifier;
      }
```

Note that the results shown near the beginning of this handout indicate a run-time performance of roughly a third of that required by the **Sequential Search 1** algorithm, a somewhat surprising outcome.

Binary Search

Since the list of `(String, Symbol)` pairs used in **Sequential Search 2** is already sorted by the first (string) component, we can use a binary search with this list and expect that it would perform faster. Using big-Oh notation, we know that a sequential search has O(n) performance, while a binary search has O(log n) performance.

It is not difficult to implement a binary search algorithm, but class `Arrays` (in package `java.util`) already does this for us. However, there is one minor glitch. There is no `compareTo()` method defined for class `StrSymPair`. We have two choices. We can declare that `StrSymPair` implements interface `Comparable` and add a `compareTo()` method, or we can implement an external `Comparator` and pass it as a second parameter to `Arrays` binary search method. We choose the second option and declare the `Comparator` as a lambda expression.

Thus, the search algorithm can be implemented as follows.

```
private Comparator<StrSymPair> rwComparator =
    (x, y) -> x.rwString.compareTo(y.rwString);

public Symbol getIdentifierSymbol(String idString)
  {
    StrSymPair key = new StrSymPair(idString, Symbol.unknown);
    int index = Arrays.binarySearch(reservedWordPairs, key, rwComparator);
    return index >= 0 ? reservedWordPairs[index].rwSymbol : Symbol.identifier;
  }
```

The benchmark results show that **Binary Search** is about twice as fast as **Sequential Search 2** making it generally acceptable for our purposes. But we can do better.

Search by Length

The general approach here is to sort the `(String, Symbol)` pairs in an array of subarrays according to the string length and then to essentially used the length of the identifier string being searched as a hash to pick out the appropriate subarray. The subarray of reserved words having the desired length is then searched sequentially.

We declare and initialize the array of subarrays as follows.

```
private StrSymPair[][] reservedWordsByLength =
  {
      // first two subarrays (indexes 0 and 1) are empty
      { },
      { },
      {
        // reserved words with two characters
        new StrSymPair("if", Symbol.ifRW),
        new StrSymPair("in", Symbol.inRW),
        new StrSymPair("is", Symbol.isRW),
        new StrSymPair("of", Symbol.ofRW),
        new StrSymPair("or", Symbol.orRW)
      },
      {
        // reserved words with three characters
        new StrSymPair("and", Symbol.andRW),
        new StrSymPair("end", Symbol.endRW),
```

```
        new StrSymPair("for", Symbol.forRW),
        new StrSymPair("mod", Symbol.modRW),
        new StrSymPair("not", Symbol.notRW),
        new StrSymPair("var", Symbol.varRW)
      },
      ...

      {
        // reserved words with nine characters
        new StrSymPair("procedure", Symbol.procedureRW),
        new StrSymPair("protected", Symbol.protectedRW)
      }
  };
```

Using this array of subarrays, the search algorithm can now be implemented as follows.

```
public Symbol getIdentifierSymbol(String idString)
  {
    // quick check based on length
    if (idString.length() > 9)
        return Symbol.identifier;

    // get array of reserved words based on length of idString
    StrSymPair[] reservedWords = reservedWordsByLength[idString.length()];

    // perform a sequential search
    for (int i = 0; i < reservedWords.length;  ++i)
      {
        if (idString.equals(reservedWords[i].rwString))
            return reservedWords[i].rwSymbol;
      }

    return Symbol.identifier;
  }
```

Benchmark results indicate that this search algorithm performs slightly faster than **Binary Search**. It is possible that we could tweak out a few extra milliseconds by reordering the subarrays according to expected frequency of use (e.g., moving `Symbol.loopRW` to the beginning of the subarray for reserved words of length four and moving `Symbol.trueRW` to the end) or by using a binary search within the appropriate subarray, but the subarrays are small enough that such changes are likely to have only minimal effect. Besides, we can do better.

Search by First Character

Rather than organizing the array of subarrays according to string length, we can organize according to the first character in the string. Then we use the first character of the identifier string being searched as a hash to pick out the appropriate subarray. The primary advantage here is that the subarrays are much shorter (average 2.1 items per nonempty subarray versus 5.1 for **Search by Length**), so sequentially searching a subarray takes less time. The primary disadvantage is that it is a little more complicated to initialize the arrays.

We declare and initialize the array of subarrays as follows.

```
private StrSymPair[][] words =
  {
      // first 65 subarrays (indexes 0-64) are empty
      { }, { }, { }, { }, { }, { }, { }, { }, { }, { }, { },
      ...
      { }, { }, { }, { }, { }, { }, { }, { }, { }, { }, { },
      { }, { }, { }, { }, { },
      { },   // 'A'
      { new StrSymPair("Boolean",   Symbol.BooleanRW) },
      { new StrSymPair("Char",      Symbol.CharRW) },
      { },   // 'D'
      ...
      { },   // 'H'
      { new StrSymPair("Integer",   Symbol.IntegerRW) },
      { },   // 'J'
      ...
      { },   // 'R'
      { new StrSymPair("String",    Symbol.StringRW) },
      { },   // 'T'
      ...
      { },   // '`'
      { new StrSymPair("and",       Symbol.andRW),
        new StrSymPair("array",     Symbol.arrayRW) },
      { new StrSymPair("begin",     Symbol.beginRW) },
      { new StrSymPair("class",     Symbol.classRW),
        new StrSymPair("const",     Symbol.constRW) },
      { new StrSymPair("declare",   Symbol.declareRW) },
      { new StrSymPair("else",      Symbol.elseRW),
        new StrSymPair("elsif",     Symbol.elsifRW),
        new StrSymPair("end",       Symbol.endRW),
        new StrSymPair("exit",      Symbol.exitRW) },
      ...
      { },   // 's'
      { new StrSymPair("then",      Symbol.thenRW),
        new StrSymPair("true",      Symbol.trueRW),
        new StrSymPair("type",      Symbol.typeRW) },
      { },   // 'u'
      { new StrSymPair("var",       Symbol.varRW) },
      { new StrSymPair("when",      Symbol.whenRW),
        new StrSymPair("while",     Symbol.whileRW),
        new StrSymPair("write",     Symbol.writeRW),
        new StrSymPair("writeln",   Symbol.writelnRW) },
      { },   // 'x'
      { },   // 'y'
      { }    // 'z'
  };
```

Using this array of subarrays, the search algorithm can now be implemented as follows.

```
public Symbol getIdentifierSymbol(String idString)
  {
    // get array of reserved word pairs based on first char of idString
    StrSymPair[] reservedWordPairs = words[(int) idString.charAt(0)];
```

```
    // perform a sequential search
    for (StrSymPair rwPair : reservedWordPairs)
      {
        if (idString.equals(rwPair.rwString))
            return rwPair.rwSymbol;
      }

    return Symbol.identifier;
  }
```

Benchmark results show that this algorithm is, indeed, faster than searching based on the length of the string.

Search Using Gperf Hash

The need to perform a fast search for a list of words, reserved or otherwise, has been around since the early days of computing, so it should come as no surprise that there is a software utility that can provide assistance. Gperf is a perfect hash function generator for a list of strings. You give gperf a list of strings, and it produces a hash function and hash table in the form of C/C++ code for looking up an arbitrary string to see if it is one of those in the list. The hash function is perfect meaning that the hash table has no collisions and the lookup needs only one single string comparison. Gperf uses extensive analysis of the list of strings (plus a little magic and trial and error) to generate the C/C++ code. Gperf is used for keyword recognition in several production and research compilers including GNU C and GNU C++.

Gperf is available primarily on Linux, but there are versions available for Windows or through Windows Subsystem for Linux (WSL). Gperf on Ubuntu (running on WSL) was used to generate the C code for CPRL reserved words, and the C code was translated to Java for comparison purposes. The output of gperf is a little cryptic. It has three major components.

First, we have `hashCode`, an array that maps characters to integer values. Here is the declaration for this array.

```
private int[] hashCode = new int[256];
```

Most of the positions in the array have value 70, but characters that appeared frequently in CPRL reserved words had different values. Here is an outline of the code used to initialize the `hashCode`.

```
for (int i = 0;  i < hashCode.length;  ++i)
    hashCode[i] = 70;

hashCode[(int)'B'] = 40;
...
hashCode[(int)'e'] = 5;
hashCode[(int)'f'] = 0;
...
hashCode[(int)'i'] = 5;
hashCode[(int)'l'] = 5;
hashCode[(int)'m'] = 25;
hashCode[(int)'n'] = 5;
hashCode[(int)'o'] = 0;
hashCode[(int)'p'] = 5;
...
hashCode[(int)'y'] = 0;
```

The idea is that the letter 'B' is associated with the integer 40 using this array, and the letter 'e' is associated with the integer 5. The selection of integer values is certainly not obvious at first glance. (I told you it was magic.)

Second, we associate a hash value (an integer) with a candidate string by adding the string's length, the hash code for the character at position zero, the hash code for the character at position one, and the hash code at position three. Assuming that str is the name of a candidate string, we compute a hash value for str as follows.

```
hash(str) = str.length + hashCode[str.charAt(0)] + hashCode[str.charAt(1)]
            + hashCode[str.charAt(3)]
```

(More magic!) If the string is short, we omit terms for those character positions. Let's look at a few examples.

The hash value for reserved word "Boolean" is given by 7 + 40 + 0 + 5 = 52.

The hash value for reserved word "loop" is given by 4 + 5 + 0 + 5 = 14.

The hash value for reserved word "of" is given by 2 + 0 + 0 = 2.

The hash value for user defined identifier "john" is given by 4 + 70 + 0 + 5 = 79.
(Note that the letter 'j' has a default hashCode value of 70.)

Third, we create an array of symbols that are indexed by hash values for our reserved words. Since the hash function isn't minimal, there will be entries in the array that do not correspond to a reserved word. For those entries we use the value Symbol.unknown. Our array contains 70 symbols and is defined as follows.

```
private Symbol[] symbolList =
  {
    Symbol.unknown,     Symbol.unknown,     Symbol.ofRW,
    Symbol.forRW,       Symbol.unknown,     Symbol.unknown,
    Symbol.unknown,     Symbol.ifRW,        Symbol.notRW,
    Symbol.typeRW,      Symbol.unknown,     Symbol.StringRW,
    Symbol.inRW,        Symbol.endRW,       Symbol.loopRW,
    Symbol.beginRW,     Symbol.publicRW,    Symbol.orRW,
    Symbol.andRW,       Symbol.elseRW,      Symbol.elsifRW,
    Symbol.unknown,     Symbol.isRW,        Symbol.varRW,
    Symbol.trueRW,      Symbol.writeRW,     Symbol.returnRW,
    Symbol.writelnRW,   Symbol.modRW,       Symbol.protectedRW,
    Symbol.falseRW,     Symbol.unknown,     Symbol.declareRW,
    Symbol.functionRW,  Symbol.exitRW,      Symbol.unknown,
    Symbol.unknown,     Symbol.privateRW,   Symbol.unknown,
    Symbol.readRW,      Symbol.arrayRW,     Symbol.readlnRW,
    Symbol.programRW,   Symbol.unknown,     Symbol.unknown,
    Symbol.constRW,     Symbol.unknown,     Symbol.IntegerRW,
    Symbol.unknown,     Symbol.unknown,     Symbol.classRW,
    Symbol.unknown,     Symbol.BooleanRW,   Symbol.unknown,
    Symbol.procedureRW, Symbol.unknown,     Symbol.unknown,
    Symbol.unknown,     Symbol.unknown,     Symbol.thenRW,
    Symbol.unknown,     Symbol.unknown,     Symbol.unknown,
    Symbol.unknown,     Symbol.whenRW,      Symbol.whileRW,
    Symbol.unknown,     Symbol.unknown,     Symbol.unknown,
    Symbol.CharRW
  };
```

(Yet more magic!) Note that, as expected, `Symbol.BooleanRW` is at index `52`, `Symbol.loopRW` is at index `14`, and `Symbol.ofRW` is at index `2`.

With these three components in place, our search algorithm is implemented as follows.

```
public Symbol getIdentifierSymbol(String idString)
  {
    if (idString.length() >= 2 && idString.length() <= 9)
      {
        int key = hash(idString);

        if (key < symbolList.length)
          {
            Symbol s = symbolList[key];

            if (idString.equals(s.toString()))
                return s;
          }
      }

    return Symbol.identifier;
  }
```

Benchmark results indicate that this search algorithm is roughly comparable in performance to the one in **Search by First Character**, but for the specific benchmark test cases, it was actually slightly slower.

Search Using `HashMap`

Our last search uses class `HashMap` from package `java.util`. `HashMap` maps keys to values, and if the key type has an efficient `hashCode()` method, `HashMap` can be very fast. In our case we want to map objects of type `String` to object of type `Symbol`, so we declare our map as follows.

```
private HashMap<String, Symbol> rwMap;
```

Fortunately for us, `String` is predefined in Java, and it has a very efficient `hashCode()` method.

The remaining details are straightforward and are left as an exercise, but initialization of the `HashMap` is just as simple and straightforward as it was to initialize the `ArrayList` in **Sequential Search 1**, and based on the benchmark analysis, using **HashMap** is the fastest search algorithm of all.

Easy to implement and fastest performance, it is the best of both worlds.