IMPLEMENTING A CUSTOM LANGUAGE

SUCCINCTLY

BY **VASSILI KAPLAN**

SUCCINCTLY E-BOOK SERIES

Syncfusion®

Implementing a Custom Language Succinctly

By
Vassili Kaplan

Foreword by Daniel Jebaraj

Technical Reviewer: Ed Freitas
Copy Editor: Courtney Wright
Acquisitions Coordinator: Morgan Weston, social media marketing manager, Syncfusion, Inc.
Proofreader: Darren West, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn’t everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors’ tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Vassili Kaplan is a former Microsoft Lync developer. He has been studying and working in a few countries, such as Russia, Mexico, the United States, and Switzerland.

He has a B.S. in Applied Mathematics from Instituto Tecnológico Autónomo de México and an M.S. in Applied Mathematics with Specialization in Computational Sciences from Purdue University, West Lafayette, Indiana.

Currently he lives with his wife and two kids in Zurich, Switzerland. He works as a freelancer for various Swiss banks. He is mostly programming in C++, C#, and Python.

In his spare time, he works on a free iPhone app called iLanguage and on the CSCS language, which is the topic of this book. His other hobbies are traveling, biking, badminton, and enjoying a glass of a good red wine.

You can contact him either at vassilik@gmail.com or at his website: http://www.iLanguage.ch.

Acknowledgements

Many thanks to all the people who contributed to this e-book, especially Tres Watkins and Darren West, and all the Syncfusion team that helped make it a reality.

Ed Freitas, the technical editor, thoroughly reviewed the e-book, its organization, code listings, and gave excellent advice that tremendously improved the quality of this e-book. He even downloaded the accompanying source code, compiled it, ran scripts mentioned in this book, and gave excellent tips on how to improve them.

This book is dedicated to my parents, Larissa Popova and Ilya Kaplan.

Chapter 1 Introduction

"If someone claims to have the perfect programming language, he is either a fool or a salesman or both."
Bjarne Stroustrup[1]

In this book we are going to develop a customizable, interpreted programming language. We are going to write a parser from scratch without using any external libraries. But why would someone want to write yet another parser and another programming language? To answer these questions, we first need a few definitions and a brief history.

Terminology and a history of parsing in an elevator pitch

The following list of definitions is drawn primarily from Wikipedia; citations have been provided.

A **compiler**[2] is a program that transforms source code written in a programming language into another computer language, very often being in binary form, ready to be run on a particular operating system. Examples of such languages are C, C++, and Fortran. Java and .NET compilers translate programs into some intermediate code. Just-in-time (JIT) compilers perform some of the compilation steps dynamically at run-time. They are used to compile Java and Microsoft .NET programs into the machine code.

An **interpreter**[3] is a program that directly executes instructions written in a programming or a scripting language, without previously compiling them. The instructions are interpreted line by line. Some of the examples of such languages are Python, Ruby, PHP, and JavaScript.

A compiler and an interpreter usually consist of two parts: a lexical analyzer and a parser.

A **lexical analyzer**[4] is a program that converts a sequence of characters into a sequence of tokens (strings with an identified "meaning"). An example would be converting the string **`"var = 12 * 3;"`** into the following sequence of tokens: **`{"var", "12", "*", "3", ";"}`**.

A **parser**[5] is a program that takes some input data (often the sequence produced by the lexical analyzer) and builds a data structure based on that data. At the end, the parser creates some form of the internal representation of the data in the target language.

The parsing process can be done in one pass or multiple passes, leading to a one-pass compiler and to a multi-pass compiler. Usually the language to be parsed conforms to the rules of some grammar.

[1] See https://www.wired.com/2010/10/1014cplusplus-released/
[2] See http://en.wikipedia.org/wiki/Compiler
[3] See http://en.wikipedia.org/wiki/Interpreter_(computing)
[4] See http://en.wikipedia.org/wiki/Lexical_analysis
[5] See http://en.wikipedia.org/wiki/Parsing - Computer_languages

Edsger Dijkstra invented the **Shunting-Yard** algorithm to parse a mathematical expression in 1961. In 1965 Donald Knuth invented the **LR** parser (**L**eft-to-Right, **R**ightmost Derivation). Its only disadvantages are potentially large memory requirements.[6]

As a remedy to this disadvantage, the **LALR** parser (**L**ook-**A**head **L**eft-to-**R**ight) was invented by Frank DeRemer in 1969 in his Ph.D. dissertation. It is based on "**bottom-up**" parsing. Bottom-up parsing is like solving a jigsaw puzzle—start at the bottom of the parse tree with individual characters,[7] then use the rules to connect these characters together into larger tokens as you go. At the end a big single sentence is produced. If not, backtrack and try combining tokens in some other ways.

The LALR parser can be automatically generated by **Yacc** (**Y**et **A**nother **C**ompiler **C**ompiler). Yacc was developed in the early 1970s by Stephen Johnson at AT&T Bell Laboratories. The Yacc input is based on an analytical grammar with snippets of C code.

A lexical analyzer, **Lex**, was described in 1975 by Mike Lesk and Eric Schmidt (Schmidt was able later to parse his way up and become the CEO of Google). Lex and Yacc are commonly used together for parsing.

The **top-down**[8] parsers construct the parse tree from the top and incrementally work from the top downwards and rightwards. It is mostly done using recursion and backtracking, if necessary. An example of such a parser is a Recursive Descent Parser.

The **LL(k)** parser is a top-down parser (**L**eft-to-right, **L**eftmost derivation), where **k** stands for the number of look-ahead tokens. The higher the **k** value is, the more languages can be generated by the LL(k) parsers, but the time complexity increases significantly with a larger **k**.

The **Bison** parser was originally written by Robert Corbett in the 1980s while doing his Ph.D. at the University of California, Berkeley. Bison converts a context-free grammar into an LALR parser. A **context-free grammar**[9] is a set of production rules that describe all possible strings of a given formal language. These rules govern string replacements that are applied regardless of the context.

The **Flex** (**F**ast **lex**ical analyzer) is an alternative to Lex. It was written by Vern Paxson in 1987 in the University of California, Berkeley. Flex with Bison are replacements of Lex with Yacc.

The framework **ANTLR** (**An**other **T**ool for **L**anguage **R**ecognition)[10] was invented by Terence Parr in 1992, while doing his Ph.D. at Purdue University. It is now widely used to automatically generate parsers based on the LL(k) grammar. In particular, this framework can even generate the Recursive Descent Parser. ANTLR is specified using a context-free grammar with EBNF (**E**xtended **B**ackus-**N**aur **F**orm; it was first developed by Niklaus Wirth from ETH Zurich).

[6] See https://en.wikipedia.org/wiki/Canonical_LR_parser

[7] See "The difference between top-down and bottom-up parsing", http://qntm.org/top

[8] See http://en.wikipedia.org/wiki/Top-down_parsing

[9] See http://en.wikipedia.org/wiki/Context-free_grammar

[10] See http://en.wikipedia.org/wiki/ANTLR

Why yet another programming language?

It is commonly assumed that you need an advanced degree in Computer Science, a few sleepless nights, and a lot of stubbornness to write a compiler.

This book will show you how to write your own compiler without an advanced degree and sleep deprivation. Besides of being an interesting exercise on its own, here are some of the advantages of writing your own language instead of using an existing language or using tools that help you create new languages:

- The language that we are going to develop turns around functions. Besides common functions (defined with the **`function`** keyword, as in many languages), all of the local and global variables are implemented as functions as well. Not only that, but all of the language constructs: **`if`** – **`else`**, **`try`** – **`catch`**, **`while`**, **`for`**, **`break`**, **`continue`**, **`return`**, etc., are all implemented as functions as well. Therefore, **changing the language is very easy and will involve just modifying one of the associated functions**. A new construct (**`unless`**, for example) can be added with just a few lines of code.

- **The functional programming** in the language that we are going to develop is straightforward since all of the language constructs are implemented internally as functions.

- All of the **keywords** of this language (**`if`**, **`else`**, **`while`**, **`function`**, and so on) **can be easily replaced by any non-English keywords** (they even don't have to be in the ASCII code, unlike most other languages). Only configuration changes are needed for replacing the keywords.

- The language that we are going to develop can be used as both as **a scripting language and as a shell program** with many useful command-promt commands, like bash on Unix or PowerShell on Windows (but we will be able to make it more user-friendly than PowerShell).

- Even Python does not have the prefix and postfix operators "++" and "--", just because Guido van Rossum decided so. There will be no Guido standing behind your back when you write your own language, so **you can decide by yourself what you need**.

- The number of method parameters is limited to 255 in Java because the Java Execution Committee thought that you would never need more (and if you would, you are not a good software architect in their eyes). This Java "feature" hit me pretty hard once when I used a custom code generator and suddenly nothing compiled anymore. It goes without saying that with your own language, **you set the limits—**not a remote committee who has no clue about the problem you are working on.

- **Any custom parsing can be implemented on the fly**. Having full control over parsing means less time searching for how to use an external package or a RegEx library.

- Using parser generators, like ANTLR, Bison, or Yacc, has a few limitations. The requirement for many tools is to have a distinct first lexical analysis phase, meaning the source file is first converted to a stream of tokens, which have to be context-free. Even C++ itself cannot be created using such tools because C++ has ambiguities, where parsing depends on the context. So there are just too many things that the generators do not handle. Our parser won't have any of these requirements—you will decide how your language should look. Furthermore, the **parsing errors can be completely customized**: something that standard parsing tools do not do.

- **You won't be using any Regular Expressions at all!** This is a showstopper for a few people who desperately need to parse an expression but are averse to the pain and humiliation induced by the regex rules.

What will be covered in this book?

I am going to show you how to write an interpreter for a custom language. This interpreter is based on a custom parsing algorithm, which I call the **Split-and-Merge** algorithm.

Using this algorithm, we can combine the lexical analysis and the parser, meaning the parser will do the lexical analysis as well. There will be no formal definition of the grammar—we will just add the new functionality as we go. All the keywords and symbols can be changed ad-hoc in a configuration file.

For brevity and simplicity, I am going to call the language that we are going to develop CSCS (**C**ustomized **S**cripting in **C#**). In CSCS we are going to develop the following features:

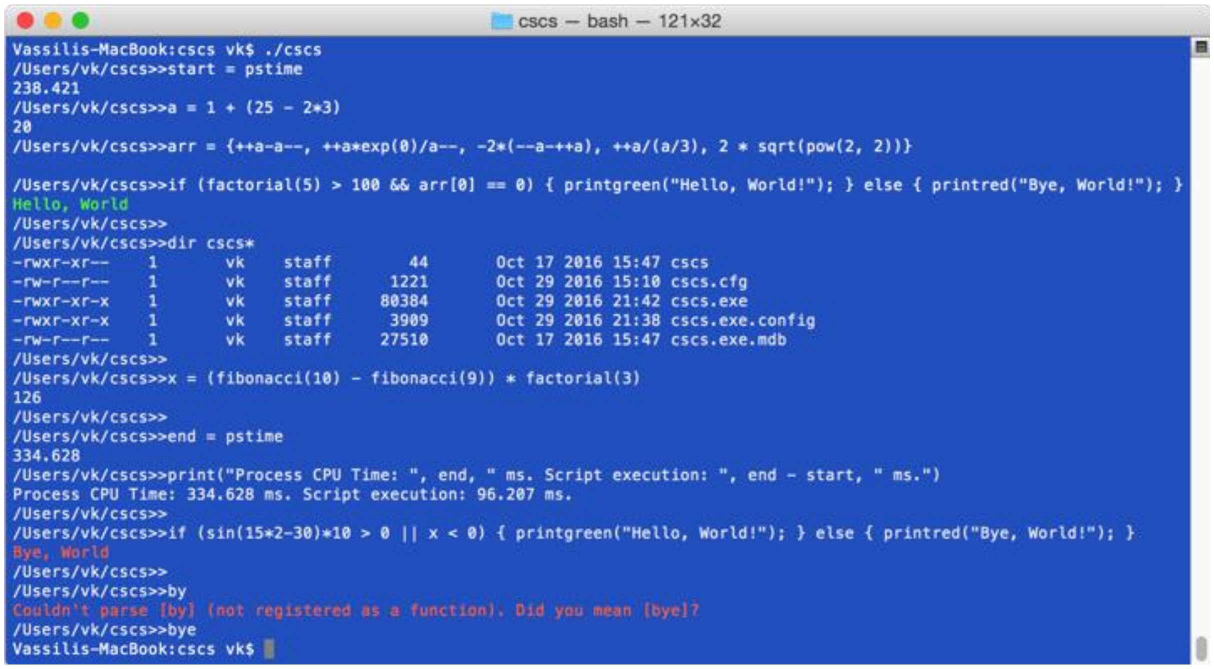
- Mathematical and logical expressions, including multiple parentheses and compound math functions
- Basic control flow statements: **`if`** – **`else`**; **`while`**; **`for`**; **`break`**; **`continue`**, etc.
- Exception handling: **`try`** – **`catch`** and **`throw`** statements
- Prefix and Postfix **`++`** and **`--`** operators
- Compound assignment operators
- Custom functions written in CSCS, including recursive functions
- Including modules from other files with the **`include`** keyword
- Using CSCS as a shell language (a command-line prompt)
- Implementing command-line functions
- Arrays and dictionaries with an unlimited number of dimensions
- Mixing different types as keys in arrays and dictionaries
- Dynamic programming
- Creating and running new threads
- Supporting keywords in multiple languages
- Possibility to translate function bodies from one language to another

In short, we are going to develop not a new programming language, but rather a framework with which it will be easy to create a new programming language with all the custom features you might need.

We are going to write code that will run on both Windows and macOS. For the former, we'll be using Microsoft Visual Studio, and for the latter, Xamarin Studio Community.

Hello World in CSCS

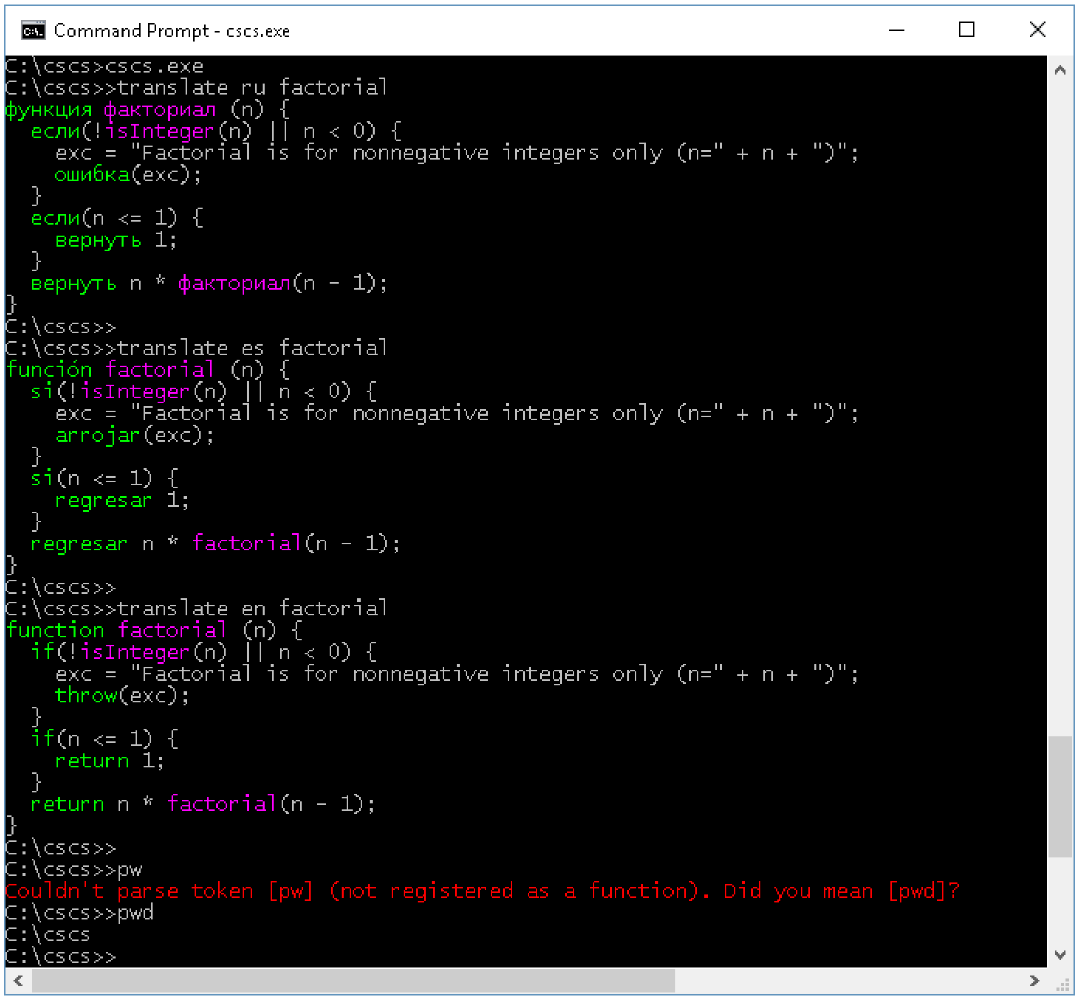
The following two screenshots will give you an idea of some of the features we are going to cover in this book. Both screenshots show the command line (or shell mode) of the CSCS operation. Figure 1 has a screenshot of a "Hello, World!" session with the CSCS language on a Mac.



```
Vassilis-MacBook:cscs vk$ ./cscs
/Users/vk/cscs>>start = pstime
238.421
/Users/vk/cscs>>a = 1 + (25 - 2*3)
20
/Users/vk/cscs>>arr = {++a-a--, ++a*exp(0)/a--, -2*(--a-++a), ++a/(a/3), 2 * sqrt(pow(2, 2))}

/Users/vk/cscs>>if (factorial(5) > 100 && arr[0] == 0) { printgreen("Hello, World!"); } else { printred("Bye, World!"); }
Hello, World
/Users/vk/cscs>>
/Users/vk/cscs>>dir cscs*
-rwxr-xr--    1      vk    staff        44       Oct 17 2016 15:47 cscs
-rw-r--r--    1      vk    staff      1221       Oct 29 2016 15:10 cscs.cfg
-rwxr-xr-x    1      vk    staff     80384       Oct 29 2016 21:42 cscs.exe
-rwxr-xr-x    1      vk    staff      3909       Oct 29 2016 21:38 cscs.exe.config
-rw-r--r--    1      vk    staff     27510       Oct 17 2016 15:47 cscs.exe.mdb
/Users/vk/cscs>>
/Users/vk/cscs>>x = (fibonacci(10) - fibonacci(9)) * factorial(3)
126
/Users/vk/cscs>>
/Users/vk/cscs>>end = pstime
334.628
/Users/vk/cscs>>print("Process CPU Time: ", end, " ms. Script execution: ", end - start, " ms.")
Process CPU Time: 334.628 ms. Script execution: 96.207 ms.
/Users/vk/cscs>>
/Users/vk/cscs>>if (sin(15*2-30)*10 > 0 || x < 0) { printgreen("Hello, World!"); } else { printred("Bye, World!"); }
Bye, World
/Users/vk/cscs>>
/Users/vk/cscs>>by
Couldn't parse [by] (not registered as a function). Did you mean [bye]?
/Users/vk/cscs>>bye
Vassilis-MacBook:cscs vk$
```

Figure 1: Sample execution of CSCS on a Mac

Figure 2 has a sample session on a Windows PC. There you can see how the keywords of a function, implemented in CSCS, can be translated to another language.



```
C:\cscs>cscs.exe
C:\cscs>>translate ru factorial
функция факториал (n) {
  если(!isInteger(n) || n < 0) {
    exc = "Factorial is for nonnegative integers only (n=" + n + ")";
    ошибка(exc);
  }
  если(n <= 1) {
    вернуть 1;
  }
  вернуть n * факториал(n - 1);
}
C:\cscs>>
C:\cscs>>translate es factorial
función factorial (n) {
  si(!isInteger(n) || n < 0) {
    exc = "Factorial is for nonnegative integers only (n=" + n + ")";
    arrojar(exc);
  }
  si(n <= 1) {
    regresar 1;
  }
  regresar n * factorial(n - 1);
}
C:\cscs>>
C:\cscs>>translate en factorial
function factorial (n) {
  if(!isInteger(n) || n < 0) {
    exc = "Factorial is for nonnegative integers only (n=" + n + ")";
    throw(exc);
  }
  if(n <= 1) {
    return 1;
  }
  return n * factorial(n - 1);
}
C:\cscs>>
C:\cscs>>pw
Couldn't parse token [pw] (not registered as a function). Did you mean [pwd]?
C:\cscs>>pwd
C:\cscs
C:\cscs>>
```

Figure 2: Sample execution of CSCS on Windows

The complete source code of the CSCS language developed in this book is available here.

Even though the implementation language will be C#, you'll see that it will be easy to implement this scripting language in any other object-oriented language. As a matter of fact, I also implemented CSCS in C++, the implementation is available here.

Chapter 2 The Split-and-Merge Algorithm

"To iterate is human, to recurse divine."
L. Peter Deutsch

All the parsing of the CSCS language that we are going to develop in this book is based on the **Split-and-Merge** algorithm to parse a custom string. We will start with pure mathematical expressions, and then generalize them into any custom CSCS language expression.

Description of the Split-and-Merge algorithm

The description that I provide here is based on the **Split-and-Merge** algorithm previously published in the MSDN Magazine,[11] ACCU Overload,[12] and in the CODE Magazine.[13]

Note that in the discussion that follows, we suppose that the parsing string doesn't contain any extra blank spaces, tabs, new lines, etc. If this is not the case, we need first to get rid of all these extra characters. (The code that gets rid of these extra characters is available in the accompanying source code).

We could have removed these unneeded blanks at the same time that we parsed the script, but this would have unnecessarily made the algorithm look more complex than it should be.

The algorithm consists of two steps: as the algorithm name implies, the first part is a **Split**, and the second one is a **Merge**.

Splitting an expression

In the first step, the string containing an expression is split into a list of so-called "variables." Each variable consists, primarily, of a number or a string and an action that must be applied to it. The numbers are internally represented as doubles, so in particular, they can be integers or Booleans.

For numbers, the actions can be all possible operations we can do with the numbers, for example, **+, –, *, /, %, or ^** (power, 2^3 means 2 to the power of 3; that is, 8) and Boolean comparisons (**&&**, **||**).

For strings, the actions can be a **+** (meaning string concatenation) or Boolean comparisons.

[11] See http://msdn.microsoft.com/en-us/magazine/mt573716.aspx

[12] See http://accu.org/index.php/journals/2252

[13] See http://www.codemag.com/Article/1607081

For convenience, we denote the action of the last variable in the list as `)`. It could've been any character, but we chose `)` to be sure that it doesn't represent any other action. We call this special action a "null" action. It has the lowest priority.

The separation criteria for splitting the string into tokens are mathematical and logical operators or parentheses. Note that the priority of the operators does not matter in the first step.

Let's see an example of applying the first step to some expressions:

`Split ("3+5") = {Variable(3, "+"), Variable(5, ")")}`

`Split ("11-2*5") = {Variable(11, "-"), Variable(2, "*"), Variable(5, ")")}`

Easy! Now, what about parentheses? As soon as we get an opening parentheses in the expression string, we recursively apply the whole **Split-and-Merge** algorithm to the expression in parentheses and replace the expression in parentheses with the calculated result. For example:

```
Split ("3*(2+2)") = {Variable(3, "*"),Variable(SplitAndMerge("2+2"), ")")}
={Variable(3, "*"), Variable(4, ")")}
```

We apply the same procedure for any function in the expression to be parsed: first we apply recursively the whole **Split-and-Merge** algorithm to the function, replacing it with the calculated result.

For example, consider parsing an expression **`1+sin(3-3)`**. As soon as we get the opening parenthesis in the **`(3-3)`** sub-expression, we apply the whole algorithm to **`3-3`**, which yields **`0`** as a result. Then our expression is reduced to **`1+sin(0)`**. Then we calculate the sine of **`0`**, which is **`0`**, reducing the expression to **`1+0`**. Splitting this expression into a list of variables leads to two variables: **`Variable(1, "+")`** and **`Variable (0, ")")`** as a result.

Note that at the end of the first step, all of the parentheses and functions will be eliminated (by possibly recursive calls to the whole **Split-and-Merge** algorithm).

Merging the list of variables

The priorities of the actions start counting in the second step, when we merge the list of variables created in the previous step. Code Listing 1**Error! Reference source not found.** contains the priorities of some actions that I defined.

Code Listing 1: Priorities of the actions

```
static int GetPriority(string action)
{
  switch (action) {
    case "++":
    case "--": return 10;
    case "^" : return 9;
    case "%" :
```

```
    case "*" :
    case "/" : return 8;
    case "+" :
    case "-" : return 7;
    case "<" :
    case ">" :
    case ">=":
    case "<=": return 6;
    case "==":
    case "!=": return 5;
    case "&&": return 4;
    case "||": return 3;
    case "+=":
    case "-=":
    case "*=":
    case "/=":
    case "%=":
    case "=" : return 2;
  }
  return 0;
}
```

Note that the last variable action `)` has the lowest priority of `0`, since it isn't included in the `switch` statement.

The list of variables is merged one by one, starting from the first variable in the list. Merging means applying the action of the left variable to the values of the left and right variables.

The resulting variable will have the same action as the right variable. The variables can only be merged if the priority of the action of the left variable is greater or equal than the priority of the action of the right variable. As soon as we have only one variable left, its value will be the final result of the whole calculation.

For example, since the priority of the **+** action is greater than the priority of the null action, we have:

`Merge {Variable(3, “+“), Variable(5, “)“)} = Variable(3 + 5, “)”) = Variable(8, “)”) = 8.`

Now what happens if two variables cannot be merged because the priority of the left variable is lower than the priority of the right variable? Then we temporarily move to the next-right variable, in order to try to merge it with the variable next to it, and so on, recursively. As soon as the right variable has been merged with the variable on its right, we return back to the original, left variable, and try to remerge it with the newly created right variable.

Eventually we will be able to merge the whole list, since sooner or later we will reach the last variable of the list that has the lowest priority, and therefore, can be merged with any variable on its left.

Let’s see an example of merging a list of variables **`(11, “-“)`**, **`(2, “*“)`**, **`(5, “)“)`** from the **Split** section. Note that the priority of **`*`** is greater than the priority of **`-`** and therefore, we cannot merge the first and second variables right away, but must first merge the second and the third variables:

`Merge {Variable(11, “-“), Variable(2, “*“), Variable(5, “)“)} =`

`Merge {Variable(11, “-“), Merge {Variable(2, “*“), Variable(5, “)“)} =`

`Merge {Variable(11, “-“), Variable(2 * 5, “)“) } =`

`Merge {Variable(11, “-“), Variable(10, “)“) } =`

`Variable(11 - 10, “)“) } = Variable(1, “)“) } = 1.`

The implementation of the Split

Let’s see the implementation of the **Split-and-Merge** algorithm in C#.

First we need to define a few auxiliary data structures. Code Listing 2 contains the **`ParsingScript`** class (the definition is incomplete there—we are going to see more functions and methods from the **`ParsingString`** class later on). It is a wrapper over the expression being parsed. It contains the actual string being parsed (**`m_data`**) and the pointer to the character where we currently are (**`m_from`**). It also has a few convenience functions for accessing and manipulating the string being parsed.

Code Listing 2: The ParsingScript class

```
public class ParsingScript
{
  private string m_data; // Contains the whole script.
  private int    m_from; // A pointer to the char being processed.

  public ParsingScript(string data, int from = 0)
  {
    m_data = data;
    m_from = from;
  }

  public char Current { get { return m_data[m_from]; } }

  public bool StillValid()          { return m_from < m_data.Length; }
  public char CurrentAndForward()   { return m_data[m_from++]; }
  public void Forward(int delta = 1) { m_from += delta; }

  public string FromPrev(int backChars = 1,
                int maxChars = Constants.MAX_CHARS_TO_SHOW) {
    return Substr(m_from - backChars, maxChars);
  }
  public char TryCurrent() {
    return m_from < m_data.Length ? m_data[m_from] : Constants.EMPTY;
  }

  public char TryPrev() {
    return m_from >= 1 ? m_data[m_from - 1] : Constants.EMPTY;
  }

  public char TryPrevPrev() {
    return m_from >= 2 ? m_data[m_from - 2] : Constants.EMPTY;
  }

  public void MoveForwardIf(char expected,
                            char expected2 = Constants.EMPTY) {
    if (StillValid() && (Current == expected || Current == expected2)) {
      Forward();
    }
  }
}
```

The **`Constants.Empty`** character is defined as **`'\0'`**.

The main task of the first step is to split the string into a list of variables. The **`Variable`** class is defined in Code Listing 3.

Basically, a variable is a wrapper over either a number (including integers, doubles, and Booleans), a string, or a list of other variables.

Code Listing 3: The Variable class

```
public class Variable
{
  public enum VarType { NONE, NUMBER, STRING, ARRAY };

  public Variable()         { Type = VarType.NONE; }
  public Variable(double d) { Value = d; }
  public Variable(bool d)   { Value = d ? 1.0 : 0.0; }
  public Variable(string s) { String = s; }

  public double          Value  {
    get { return m_value; }
    set { m_value = value;  Type = VarType.NUMBER; } }

  public string          String {
    get { return m_string; }
    set { m_string = value; Type = VarType.STRING; } }

  public List<Variable>  Tuple  {
    get { return m_tuple; }
    set { m_tuple = value; Type = VarType.ARRAY; } }

  public void            SetAsArray() {
    Type = VarType.ARRAY;
    if (m_tuple == null) { m_tuple = new List<Variable>(); }
  }

  public string          Action   { get; set; }
  public VarType         Type     { get; set; }
  public bool            IsReturn { get; set; }

  public static Variable EmptyInstance = new Variable();

  private double         m_value;
  private string         m_string;
  private List<Variable> m_tuple;
}
```

Code Listing 4 shows the main function to split an expression into a list of variables.

Code Listing 4: The Split part of the algorithm

```
static List<Variable> Split(ParsingScript script, char[] to)
{
  List<Variable> listToMerge = new List<Variable>();
  StringBuilder item = new StringBuilder();

  do { // Main processing cycle of the first part.
    char ch = script.CurrentAndForward();
    string action = null;

    bool keepCollecting = StillCollecting(item.ToString(),
                                       to, script, ref action);
    if (keepCollecting)  {
    // The char still belongs to the token being collected.
      item.Append(ch);

      bool goForMore = script.StillValid() && !to.Contains(script.Current);
      if (goForMore) {
        continue;
      }
    }

    string token = item.ToString();
    if (action != null && action.Length > 1) {
      script.Forward(action.Length - 1);
    }
    // We are done getting the next token. The GetValue() call may
    // recursively call this method. This can happen if the extracted item
    // is a function or if the next item is starting with START_ARG '('.
    ParserFunction func = new ParserFunction(script, token,
                                            ch, ref action);
    Variable current    = func.GetValue(script);

    if (action == null)  {
      action = UpdateAction(script, to);
    }
    current.Action = action;
    listToMerge.Add(current);

    item.Clear();
  } while (script.StillValid() && !to.Contains(script.Current));

  // This happens when called recursively inside of a math expression:
  script.MoveForwardIf(Constants.END_ARG); // END_ARG = ')'
  return listToMerge;
}
```

The **item** variable holds the current token, adding characters to it one by one as soon as they are read from the string being parsed. To figure out whether we need to add the next character to the current token, the **StillCollecting** method is invoked. We show it in Code Listing 5.

Code Listing 5: The StillCollecting method to check if we are still collecting the current token

```
static bool StillCollecting(string item, char[] to,
                            ParsingScript script, ref string action)
{
  char prev = script.TryPrevPrev();
  char ch   = script.TryPrev();
  char next = script.TryCurrent();

  if (to.Contains(ch) || ch == Constants.START_ARG ||
                         ch == Constants.START_GROUP ||
                       next == Constants.EMPTY)      {
    return false;
  }

  // Otherwise, if it's an action (+, -, *, etc.) or a space
  // we're done collecting current token.
  if ((action = Utils.ValidAction(script.FromPrev())) != null ||
      (item.Length > 0 && ch == Constants.SPACE)) {
    return false;
  }
  return true;
}
```

The **StillCollecting** method uses the following constants defined in the **Constants** class:

```
const char START_ARG     = '(';
const char START_GROUP   = '{';
```

Obviously, you are free to redefine them as you wish. The condition **next == Constants.EMPTY** means that there are no more characters in the script being parsed; that is, the **ch** argument is already the last character in the script.

The **StillCollecting** method in Code Listing 5 also invokes the **Utils.VaildAction** method to find the action that follows the token that we just extracted.

We show the **VailidAction** and **UpdateAction** methods in Code Listing 6.

Code Listing 6: The methods to return an action if the passed string starts with it

```
public static string ValidAction(string rest)
{
  foreach (string action in Constants.ACTIONS) {
    if (rest.StartsWith(action)) {
      return action;
    }
  }
  return null;
}

static string UpdateAction(ParsingScript script, char[] to)
{
  // We look for a valid action till we get to the END_ARG ')'
  // or we pass the end of the string.
  if (!script.StillValid() || script.Current == Constants.END_ARG ||
                              to.Contains(script.Current)) {
    return Constants.NULL_ACTION;
  }

  string action = Utils.ValidAction(script.Rest);

  int advance = action == null ? 0 : action.Length;
  script.Forward(advance);
  return action == null ? Constants.NULL_ACTION : action;
}
```

The **Constants.ACTIONS** is a string array containing all actions we want to define:

```
public static string[] OPER_ACTIONS = { "+=", "-=", "*=", "/=",
                                            "%=", "&=", "|=", "^="};
public static string[] MATH_ACTIONS = { "&&", "||", "==", "!=", "<=", ">=",
                        "++", "--", "%", "*", "/", "+", "-", "^", "<", ">", "="};
// Actions: always decreasing by the number of characters.
public static string[] ACTIONS=(OPER_ACTIONS.Union(MATH_ACTIONS)).ToArray();
```

The actual dealing with every extracted token in the **`Split`** method (Code Listing 4) is in the following 2 lines of code:

```
    ParserFunction func = new ParserFunction(script, token, ch, ref action);
    Variable current    = func.GetValue(script);
```

If the extracted token is not a function previously registered with the parser, this code will try to convert the extracted token to a double (and throw an exception if the conversion is not possible).

Otherwise, a previously registered function will be called, which can in turn recursively invoke the the whole **Split-and-Merge** algorithm. The appropriate **`ParserFunction`** will be selected based on the token being processed. How do we select the appropriate function? This is what we'll explore next.

Virtual constructor idiom

To choose the appropriate function, I decided to use the *virtual constructor idiom*. Virtual constructor? Is it legal? James Coplien described a trick that simulates a virtual constructor. He described this idiom to be used in C++, but it can be easily used in any other object-oriented language as well, so here we are going to use this idiom in C#.

In C# the same functionality is often implemented using a factory method that uses an additional factory class to produce the required object. But Coplien's design pattern doesn't need an extra class, and produces instead the required object on the fly. According to Coplien:[14]

"The virtual constructor idiom is used when the type of an object needs to be determined from the context in which the object is constructed."

This is exactly the case when we associate each token being parsed with a function at runtime. The main advantage of this idiom is that we can easily add a new function to the parser without modifying the main algorithm implementation code.

The main idea of this idiom is to use the **`ParserFunction`** constructor to create the **`m_impl`** member of the object, which is a **`ParserFunction`** itself:

```
   private ParserFunction m_impl;
```

So the constructor of **`ParserFunction`** initializes the **`m_impl`** member with the appropriate class, deriving from **`ParserFunction`**. See the implementation in Code Listing 7.

[14] James Coplien, *Advanced C++ Programming Styles and Idioms* (p. 140), Addison-Wesley, 1992.

Code Listing 7: The ParserFunction base class

```
public class ParserFunction
{
  // A "normal" Constructor
  public ParserFunction() { m_impl = this; }

  // A "virtual" Constructor
  internal ParserFunction(ParsingScript script, string token,
                          char ch, ref string action) {
    if (token.Length == 0 &&
       (ch == Constants.START_ARG || !script.StillValid ())) {
      // There is no function, just an expression in parentheses.
      m_impl = s_idFunction;
      return;
    }

    m_impl = GetFunction(token);
    if (m_impl != null) {
      return;
    }

    if (m_impl == s_strOrNumFunction &&
        string.IsNullOrWhiteSpace(token))  {
      string restData = ch.ToString () + script.Rest;
      throw new ArgumentException("Couldn't parse [" + restData + "]");
    }

    // Function not found, parse this as a string in quotes or a number.
    s_strOrNumFunction.Item = token;
    m_impl = s_strOrNumFunction;
  }

  public Variable GetValue(ParsingScript script)
  {
    return m_impl.Evaluate(script);
  }

  protected virtual Variable Evaluate(ParsingScript script)
  {
    // The real implementation will be in the derived classes.
    return new Variable();
  }

  public static ParserFunction GetFunction(string item)
  {
    ParserFunction impl;
    if (s_functions.TryGetValue(item, out impl)) {
    // Function exists and is registered (e.g. pi, exp, or a variable)
```

```
        return impl.NewInstance();
      }
      return null;
    }

    // Derived classes may want to return a new instance in order to
    // not to use the same object in calculations.
    public virtual ParserFunction NewInstance() {
      return this;
    }

    public static void RegisterFunction(string name,
                                        ParserFunction function) {
      s_functions[name] = function;
    }

    private static Dictionary<string, ParserFunction> s_functions =
               new Dictionary<string, ParserFunction>();

    private static StringOrNumberFunction s_strOrNumFunction =
               new StringOrNumberFunction();
    private static IdentityFunction s_idFunction =
               new IdentityFunction();
```

Notice that the **ParserFunction** has a missing curly brace. This is because the class listing continues later on, in particular in Listings 8 and 9.

The **Evaluate** method (called from the base class **GetValue** method) will use the actual **ParserFunction** object.

So each class deriving from the **ParserFunction** class is expected to override the **Evaluate** method: this is where the real action takes place. We'll see plenty examples later on.

Special parser functions

Different parser functions can be registered with the parser using the **RegisterFunction** method in Code Listing 7. A dictionary is used to map all of the parser functions to their names:

```
  private static Dictionary<string, ParserFunction> s_functions =
             new Dictionary<string, ParserFunction>();
```

There are two special functions deriving from the **ParserFunction** class:

```
  private static StringOrNumberFunction s_strOrNumFunction =
             new StringOrNumberFunction();
  private static IdentityFunction s_idFunction =
             new IdentityFunction();
```

They are used explicitly in the **`ParserFunction`** constructor, and their implementation is provided in Code Listing 8.

Code Listing 8: IdentityFunction and StringOrNumberFunction implementations

```
class IdentityFunction : ParserFunction
{
  protected override Variable Evaluate(ParsingScript script)
  {
    return script.ExecuteTo(Constants.END_ARG); // END_ARG == ')'
  }
}

class StringOrNumberFunction : ParserFunction
{
  protected override Variable Evaluate(ParsingScript script)
  {
    // First check if the passed expression is a string between quotes.
    if (Item.Length > 1 &&
      Item[0] == Constants.QUOTE &&
      Item[Item.Length - 1]  == Constants.QUOTE) {
      return new Variable(Item.Substring(1, Item.Length - 2));
    }

    // Otherwise this should be a number.
    double num;
    if (!Double.TryParse(Item, out num)) {
      Utils.ThrowException(script, "parseToken");
    }
    return new Variable(num);
  }

  public string Item { private get; set; }
}
```

The **`IdentityFunction`** is used when we have an expression in parentheses. The whole **Split-and-Merge** algorithm is called on that expression in parentheses, because the **`ParsingScript.ExecuteTo`** function is just a wrapper over, it as you can see in Code Listing 9.

Code Listing 9: The implementation of ParsingScript Execute, ExecuteTo, and ExecuteFrom functions

```
public Variable ExecuteTo(char to = '\0')
{
  return ExecuteFrom(Pointer, to);
}

public Variable ExecuteFrom(int from, char to = '\0')
{
  Pointer = from;
  char[] toArray = to == '\0' ? Constants.END_PARSE_ARRAY :
                                to.ToString().ToCharArray();
  return Execute(toArray);
}

public Variable Execute(char[] toArray)
{
  if (!m_data.EndsWith(Constants.END_STATEMENT.ToString())) {
    m_data += Constants.END_STATEMENT;
  }
  return Parser.SplitAndMerge(this, toArray);
}
```

The **`StringOrNumberFunction`** is a “Catch All” function that is called when no function is found that corresponds to the current token. Therefore, it yields either a number or a string in quotes. Otherwise, an exception is thrown.

Registering a function with the parser

A typical implementation of a function deriving from the **`ParserFunction`** is shown in Code Listing 10.

In order to use this function with the parser, one needs to register it using the **`RegisterFunction`** static method (refer to Code Listing 7).

Code Listing 10: An example of a ParserFunction: the exponential function

```
class ExpFunction : ParserFunction
{
  protected override Variable Evaluate(ParsingScript script)
  {
    Variable result = script.ExecuteTo(Constants.END_ARG);
    result.Value = Math.Exp(result.Value);
    return result;
  }
}
```

Basically, to register a function with the parser, there are three steps:

1. Find an appropriate function name and define it a s a constant (I define all the constants in the **`Constants`** class):

   ```
   public const string EXP = "exp";
   ```

2. Implement a new class, having **`ParserFunction`** as its base class, and override its **`Evaluate`** method (as in Code Listing 10).

3. Register this function with the Parser:
   `ParserFunction.RegisterFunction(Constants.EXP, new ExpFunction());`

That's it! After completing these steps, you can add any function you wish and use it with the parser. For example, we can now use the exponential function that we defined previously in expressions like **`1+exp(10)`**, and so on.

The Implementation of the Merge

In the second part of the algorithm, we merge the list of variables created in the previous step one by one, applying recursion if we can't merge items right away.

See Code Listing 11 for the implementation details.

From the outside, the **`Merge`** method is called with the **`mergeOneOnly`** argument set to **`false`**, so it will not return before completing the whole merging part.

When the **`Merge`** method is called recursively (when the left and right cells cannot be merged with each other because of their priorities), the **`Merge`** method will be called recursively with the **`mergeOneOnly`** parameter set to **`true`**.

This is because we want to return to where we were as soon as we complete one merge in the **`MergeCells`** method ("cell" is a synonym of "variable" in our context).

Code Listing 11: The implementation of the Merge part of the algorithm

```
static Variable Merge(Variable current, ref int index,
               List<Variable> listToMerge, bool mergeOneOnly = false)
{
  while (index < listToMerge.Count)  {
    Variable next = listToMerge[index++];

    while (!CanMergeCells(current, next)) {
      // If we cannot merge cells yet, go to the next cell and merge
      // next cells first. E.g. if we have 1+2*3, we first merge next
      // cells, i.e. 2*3, getting 6, and then we can merge 1+6.
      Merge(next, ref index, listToMerge, true /* mergeOneOnly */);
    }

    MergeCells(current, next);
    if (mergeOneOnly)  {
      break;
    }
  }
  return current;
}
```

We merge variables one by one, and for each pair we invoke the **CanMergeCells** method, shown in Code Listing 12. This method just compares priorities of the actions of each cell. We saw the priorities of each action in Code Listing 1.

Code Listing 12 also shows the implementation of the **MergeCells** method. There is a special treatment of the variables **Break** and **Continue**. It consists in not doing nothing. We will see the **Break** and **Continue** variables in more detail in the next chapter.

Code Listing 12: The implementation of the CanMerge and MergeCells methods

```
static bool CanMergeCells(Variable leftCell, Variable rightCell)
{
  return GetPriority(leftCell.Action) >= GetPriority(rightCell.Action);
}

static void MergeCells(Variable leftCell, Variable rightCell)
{
  if (leftCell.Type == Variable.VarType.BREAK ||
      leftCell.Type == Variable.VarType.CONTINUE) {
    // Done!
    return;
  }

  if (leftCell.Type  == Variable.VarType.NUMBER) {
    MergeNumbers(leftCell, rightCell);
  }  else {
    MergeStrings(leftCell, rightCell);
  }

  leftCell.Action = rightCell.Action;
}
```

Depending on the cells being merged, they will be merged either as strings or as numbers. Code Listing 13 shows the implementation of the **MergeNumbers** method.

Code Listing 13: The implementation of the MergeNumbers method

```
static void MergeNumbers(Variable leftCell, Variable rightCell)
{
  switch (leftCell.Action) {
    case "^": leftCell.Value = Math.Pow(leftCell.Value, rightCell.Value);
      break;
    case "%": leftCell.Value %= rightCell.Value;
      break;
    case "*": leftCell.Value *= rightCell.Value;
      break;
    case "/":
      if (rightCell.Value == 0.0) {
        throw new ArgumentException("Division by zero");
      }
      leftCell.Value /= rightCell.Value;
      break;
    case "+":
      if (rightCell.Type != Variable.VarType.NUMBER) {
        leftCell.String = leftCell.AsString() + rightCell.String;
      } else {
        leftCell.Value += rightCell.Value;
      }
      break;
    case "-":  leftCell.Value -= rightCell.Value;
      break;
    case "<":  leftCell.Value = Convert.ToDouble(
                                  leftCell.Value < rightCell.Value);
      break;
    case ">":  leftCell.Value = Convert.ToDouble(
                                  leftCell.Value > rightCell.Value);
      break;
    case "<=": leftCell.Value = Convert.ToDouble(
                                  leftCell.Value <= rightCell.Value);
      break;
    case ">=": leftCell.Value = Convert.ToDouble(
                                  leftCell.Value >= rightCell.Value);
      break;
    case "==": leftCell.Value = Convert.ToDouble(
                                  leftCell.Value == rightCell.Value);
      break;
    case "!=": leftCell.Value = Convert.ToDouble(
                                  leftCell.Value != rightCell.Value);
      break;
    case "&&": leftCell.Value = Convert.ToDouble(
                                  Convert.ToBoolean(leftCell.Value) &&
                                  Convert.ToBoolean(rightCell.Value));
      break;
    case "||": leftCell.Value = Convert.ToDouble(
                                  Convert.ToBoolean(leftCell.Value) ||
                                  Convert.ToBoolean(rightCell.Value));
      break;
  }
}
```

Code Listing 14 shows the implementation of the **`MergeStrings`** method. Note that this method may result in a variable of the **`Number`** type, in case the action is a comparison operator (for example, **`“a” > “z”`** leads to a variable of type **`Number`** and value **`0`**). Note that we represent Boolean values internally as integers with **`0`** corresponding to “false” and **`1`** corresponding to “true.”

Code Listing 14: The implementation of the MergeStrings method

```
static void MergeStrings(Variable leftCell, Variable rightCell)
{
  switch (leftCell.Action) {
    case "+": leftCell.String = leftCell.AsString() + rightCell.AsString();
         break;
    case "<":  leftCell.Value = Convert.ToDouble(
         string.Compare(leftCell.AsString(), rightCell.AsString()) < 0);
         break;
    case ">":  leftCell.Value = Convert.ToDouble(
         string.Compare(leftCell.AsString(), rightCell.AsString()) > 0);
         break;
    case "<=": leftCell.Value = Convert.ToDouble(
         string.Compare(leftCell.AsString(), rightCell.AsString()) <= 0);
         break;
    case ">=": leftCell.Value = Convert.ToDouble(
         string.Compare(leftCell.AsString(), rightCell.AsString()) >= 0);
         break;
    case "==": leftCell.Value = Convert.ToDouble(
         string.Compare(leftCell.AsString(), rightCell.AsString()) == 0);
         break;
    case "!=": leftCell.Value = Convert.ToDouble(
         string.Compare(leftCell.AsString(), rightCell.AsString()) != 0);
         break;
    default: throw new ArgumentException("Can't perform action [" +
                       leftCell.Action + "] on strings");
  }
}
```

The **MergeStrings** method heavily relies on the **Variable.AsString** method, which will be shown later on (you can jump to Code Listing 26 to check it out).

Note that at the end of the **Merge** method, only one **Variable** will be left and returned back. That **Variable** will contain the final result of the calculation.

Conclusion

We have seen how to parse a mathematical expression, containing any number of functions (previously registered with the parser) and any number of nested parentheses.

Note that the code we developed in this chapter is still not quite bullet-proof. Some of the items I took away in order to make the algorithm look easier and concentrate on the remaining items in later chapters. Some of these missing items that we are going to catch later on are:

- The “short-circuit” evaluation (when we stop calculating a complex logical condition once we know for sure that it will be true or false, regardless of the remaining terms)
- Processing logical negation

- Processing a string in quotes
- Processing arrays and dictionaries
- Error handling

Prefix and postfix operators, assignments, etc.

Chapter 3 Basic Control Flow Statements

"Talk is cheap. Show me the code."
Linus Torvalds

In the previous chapter we created basic building blocks for our programming language, CSCS. Namely, we saw how to parse a basic mathematical expression and how to define a function and add it to CSCS.

In this chapter we are going to develop the basic control flow statements: **`if – else`**, **`while`**, and some others. You will see that they are all implemented almost identically—as functions (very similar to the implementation of the exponential function in Code Listing 10). We are going to delay the implementation of the **`for`** loop until *Chapter 6 Operators, Arrays, and Dictionaries*, where we are going to see it together with arrays.

We are also going to muscle up the basic parsing algorithm that we started in the previous chapter by adding to it some missing features that we mentioned at the end of the last chapter.

Implementing the If – Else If – Else functionality

To implement the **`if`** statement functionality, we use the same algorithm we used in the "Registering a function with the " section in Chapter 2.

1. Define the **`if`** constant in the **`Constants`** class:
   ```
   public const string IF = "if";
   ```

2. Implement a new class, **`IfStatement`**, having **`ParserFunction`** as its base class, and override its **`Evaluate`** method. There is going to be a bit more work implementing the **`if`** statement; the real implementation is forwarded to the **`Interpreter`** class (see Code Listing 15).

3. Register this function with the parser:
   ```
   ParserFunction.RegisterFunction(Constants.IF, new IfStatement());
   ```

Code Listing 15: The implementation of the if-statement

```
class IfStatement : ParserFunction
{
  protected override Variable Evaluate(ParsingScript script)
  {
    Variable result = Interpreter.Instance.ProcessIf(script);
    return result;
  }
}
```

Note that we defined only **`if`** as a function, and did not do anything with the **`else if`** and **`else`** control flow statements. The reason is that **`else if`** and **`else`** are going to be processed all together in the **`if`** statement processing.

The main reason of implementing the **`if`** statement in a different class is that the implementation will use a few functions and methods that we will be sharing with other control flow statements (for example, with **`while`** and **`for`**). See Code Listing 16 for the implementation of the **`if`** statement.

Code Listing 16: The processing of the if statement

```
internal Variable ProcessIf(ParsingScript script)
{
  int startIfCondition = script.Pointer;

  Variable result = script.ExecuteTo(Constants.END_ARG);
  bool isTrue = Convert.ToBoolean(result.Value);

  if (isTrue)  {
    result = ProcessBlock(script);

    if (result.Type == Variable.VarType.BREAK ||
        result.Type == Variable.VarType.CONTINUE) {
      // Got here from the middle of the if-block. Skip it.
      script.Pointer = startIfCondition;
      SkipBlock(script);
    }

    SkipRestBlocks(script);

    return result;
  }

  // We are in Else. Skip everything in the If statement.
  SkipBlock(script);

  ParsingScript nextData = new ParsingScript(script);
  string nextToken = Utils.GetNextToken(nextData);

  if (Constants.ELSE_IF_LIST.Contains(nextToken))  {
    script.Pointer = nextData.Pointer + 1;
    result = ProcessIf(script);
  }
  else if (Constants.ELSE_LIST.Contains(nextToken))  {
    script.Pointer = nextData.Pointer + 1;
    result = ProcessBlock(script);
  }

  return Variable.EmptyInstance;
}
```

First, we note where we are in the parsing script within the **startIfCondition** variable, and then we apply the **Split-and-Merge** algorithm to the whole expression in parentheses after the **if** keyword in order to evaluate the **if** condition.

Constants.END_ARG_ARRAY is an array consisting of a **)** character, meaning we are going to apply the **Split-and-Merge** algorithm to the passed string until we get the closing parenthesis character that matches this opening parenthesis. The important thing here is the statement "that matches this opening parenthesis."

It means that if we get another opening parenthesis before we get a closing parenthesis, we will recursively apply the **Split-and-Merge** algorithm to the whole expression between the next opening parenthesis and its corresponding closing parenthesis.

If the condition in the **if** statement is true, then we will be within the **if (isTrue)** block. There we call the **ProcessBlock** method to process all the statements of the **if** block. See Code Listing 17.

Code Listing 17: The implementation of the Interpreter.ProcessBlock method

```
Variable ProcessBlock(ParsingScript script)
{
  int blockStart = script.Pointer;
  Variable result = null;

  while(script.StillValid()) {
    int endGroupRead = script.GoToNextStatement();
    if (endGroupRead > 0) {
      return result != null ? result : new Variable();
    }

    if (!script.StillValid()) {
      throw new ArgumentException("Couldn't process block [" +
        script.Substr(blockStart, Constants.MAX_CHARS_TO_SHOW) + "]");
    }

    result = script.ExecuteTo();

    if (result.Type == Variable.VarType.BREAK ||
        result.Type == Variable.VarType.CONTINUE) {
      return result;
    }
  }
  return result;
}
```

Inside of the **ProcessBlock** method, we recursively apply the **Split-and-Merge** algorithm to all of the statements between the **Constants.START_GROUP** (defined as **{** in the **Constants** class) and **Constants.END_GROUP** (defined as **}** in the **Constants** class) characters. **ParsingScript.GoToNextStatement** is an auxiliary method that moves the current script pointer to the next statement and returns an integer greater than zero if the **`Constants.END_GROUP`** has been reached. See Code Listing 18 for details.

Code Listing 18: The implementation of the ParsingScript.GoToNextStatement method

```
public int GoToNextStatement()
{
  int endGroupRead = 0;

  while (StillValid()) {
    char currentChar = Current;
    switch (currentChar) {
      case Constants.END_GROUP:       // '}'
        endGroupRead++;
        Forward();
        return endGroupRead;
      case Constants.START_GROUP:     // '{'
      case Constants.QUOTE:           // '"'
      case Constants.SPACE:           // ' '
      case Constants.END_STATEMENT:   // ';'
      case Constants.END_ARG:         // ')'
        Forward();
        break;
      default: return endGroupRead;
    }
  }

  return endGroupRead;
}
```

Let's continue with the processing of the **`if`** statement in Code Listing 16.

If one of the statements inside of the **`if`** block is a **`break`** or a **`continue`** statement, we must finish the processing and propagate this "break" or "continue" up the stack (the **`if`** statement that we are processing may be inside of a **`while`** or a **`for`** loop).

We'll see how to implement the **`break`** and **`continue`** control flow statements in the next section.

So basically, we need to skip the rest of the statements in the **`if`** block. For that we return back to the beginning of the **`if`** block and skip everything inside it. The implementation is shown in Code Listing 19.

Note that we also skip all nested blocks inside of the curly braces and throw an exception if there is no match between opening and closing curly braces.

Code Listing 19: The implementation of the Interpreter.SkipBlock method

```
void SkipBlock(ParsingScript script)
{
  int blockStart = script.Pointer;
  int startCount = 0;
  int endCount = 0;

  while (startCount == 0 || startCount > endCount) {
    if (!script.StillValid())  {
      throw new ArgumentException("Couldn't skip block [" +
        script.Substr(blockStart, Constants.MAX_CHARS_TO_SHOW) + "]");
    }
    char currentChar = script.CurrentAndForward();
    switch (currentChar) {
      case Constants.START_GROUP: startCount++; break;
      case Constants.END_GROUP:   endCount++; break;
    }
  }

  if (startCount != endCount) {
    throw new ArgumentException("Mismatched parentheses");
  }
}
```

Once we know that the **`if`** condition is true and we have executed all statements that belong to that **`if`** condition, we must skip the rest of the statements that are part of the **`if`** control flow statement (the rest are also the **`else if`** and **`else`** conditions and their corresponding statements). This is done in Code Listing 20: basically we call the **`SkipBlock`** method in a loop.

Code Listing 20: The implementation of the Interpreter.SkipRestBlocks method

```
void SkipRestBlocks(ParsingScript script)
{
  while (script.StillValid())  {
    int endOfToken = script.Pointer;
    ParsingScript nextData = new ParsingScript(script);
    string nextToken = Utils.GetNextToken(nextData);

    if (!Constants.ELSE_IF_LIST.Contains(nextToken) &&
        !Constants.ELSE_LIST.Contains(nextToken))  {
      return;
    }

    script.Pointer = nextData.Pointer;
    SkipBlock(script);
  }
}
```

We already saw what happens if the condition in the **`if`** statement is true. When it is false, we just skip the whole **`if`** block and start looking at what comes next. If an **`else if`** statement comes next, then we call recursively the **`ProcessIf`** method.

If it is an **else** statement, then we just need to execute all of the statements inside of the **else** block. We can do that by calling the **ProcessBlock** method that we already saw in Code Listing 19.

Note: It's important to note that the `if` statement always expects curly braces. You can change this functionality in `ProcessBlock` and `SkipBlock` methods in case there is no curly brace character after the `if` condition. The same applies to the `while` and `for` loops that follow.

Now let's see the implementation of the **break** and **continue** control flow statements.

Break and continue control flow statements

As you saw in Code Listing 17, there are two additional variable types in the **if** statement processing:

```
if (result.Type == Variable.VarType.BREAK ||
    result.Type == Variable.VarType.CONTINUE) {
  return result;
}
```

They are not present in Code Listing 3, but they are used for the typical **break** and **continue** control flow statements. How can we implement the **break** and **continue** statements in CSCS?

The **break** and **continue** control flow statements are implemented analogous to the exponential function and to the **if** statement in the previous section, using the three steps we described before. But first, let's also extend the definition of the **enum VarType** in Code Listing 3:

```
public enum VarType { NONE, NUMBER, STRING, ARRAY, BREAK, CONTINUE };
```

Then, according to our three steps:

1. We define the corresponding constants in the **Constants** class:

   ```
   public const string BREAK    = "break";
   public const string CONTINUE = "continue";
   ```

2. We introduce the **Break** and **Continue** functions, deriving from the **ParserFunction** class (see Code Listing 21).

3. We register the newly created functions with the parser:

```
ParserFunction.RegisterFunction(Constants.BREAK,    new BreakStatement());
ParserFunction.RegisterFunction(Constants.CONTINUE, new ContinueStatement());
```

Code Listing 21: The implementation of the Break and Continue functions

```
class BreakStatement : ParserFunction
{
  protected override Variable Evaluate(ParsingScript script)
  {
    return new Variable(Variable.VarType.BREAK);
  }
}

class ContinueStatement : ParserFunction
{
  protected override Variable Evaluate(ParsingScript script)
  {
    return new Variable(Variable.VarType.CONTINUE);
  }
}
```

That's it! So how do these **`Break`** and **`Continue`** functions work?

As we saw in Code Listing 16, as soon as we get a **`break`** or a **`continue`** statement, we stop all the processing and return (propagate it) to whoever called this statement.

`Break` and **`Continue`** make sense in **`while`** and **`for`** loops. Note that here we won't throw any error if there is no **`while`** or **`for`** loop, which can use these **`break`** and **`continue`** statements—we just stop processing the current block.

If this bothers you, you can change this functionality and throw an exception if there is no **`while`** or **`for`** loop (we would probably need an additional variable to keep track of nested **`while`** and **`for`** loops).

In the next two sections we are going to see how the **`break`** and **`continue`** statements are used in **`while`** and **`for`** loops.

Implementing the while loop

The implementation of the **`while`** loop is analogous to the implementation of the **`if`** control flow statement. The **`while`** loop is also implemented as a function object deriving from the **`ParserFunction`** class. I hate being repetitive, but again, there are three steps: defining the corresponding constant, implementing an **`Evaluate`** method of a class deriving from the **`ParserFunction`** class, and gluing it all together by registering this newly created class with the parser. So we skip these steps.

Now we can also reuse a few methods that we saw in the **`if`**-statement implementation.

Analogous to the **ProcessIf** method in Code Listing 16, we implement the **ProcessWhile** class in Code Listing 22. They do look similar—we just check for the condition to be true not once, but in a while loop.

Code Listing 22: The implementation of the While Loop

```
internal Variable ProcessWhile(ParsingScript script)
{
  int startWhileCondition = script.Pointer;
  int cycles = 0;
  bool stillValid = true;

  while (stillValid) {
    script.Pointer = startWhileCondition;
    Variable condResult = script.ExecuteTo(Constants.END_ARG);
    stillValid = Convert.ToBoolean(condResult.Value);
    if (!stillValid) {
      break;
    }

    // Check for an infinite loop if we are comparing same values:
    if (MAX_LOOPS > 0 && ++cycles >= MAX_LOOPS) {
      throw new ArgumentException("Looks like an infinite loop after " +
                                  cycles + " cycles.");
    }

    Variable result = ProcessBlock(script);
    if (result.IsReturn || result.Type == Variable.VarType.BREAK) {
      script.Pointer = startWhileCondition;
      break;
    }
  }

  // The while condition is not true anymore: must skip the whole while
  // block before continuing with next statements.
  SkipBlock(script);
  return Variable.EmptyInstance;
}
```

Note that we also check the total number of loops and compare it with the **MAX_LOOPS** variable.

This number can be set in the configuration file. If it’s greater than zero, then we stop after this number of loops. No programming language defines a number after which it is likely to suppose that you have an infinite loop. The reason is that the architects of a programming language never know in advance what would be an “infinite” number for you. However, you know your own needs better than those architects, so you can set this limit to an amount that is reasonable for you—say to 100,000—if you know that you’ll never have as many loops, unless you have a programming mistake.

We also break from the **while** loop when having this condition:

```
    if (result.IsReturn || result.Type == Variable.VarType.BREAK) {
      script.Pointer = startWhileCondition;
      break;
    }
```

Note that before breaking out of the **`while`** loop, we reset the pointer of the parsing script to the beginning of the **`while`** condition. We are doing this because it's just easier to skip the whole block between the curly braces by calling the **`SkipBlock`** method, but you could also skip just the rest of the block; then you would need to keep track of the number of opening and closing curly braces, already processed inside of the **`while`** block.

We saw how to trigger the type of the result **`Variable.VarType.BREAK`** at the end of last section. The **`IsReturn`** property will be set to true when a **`return`** statement is executed inside a function. We'll see how it's done in the next chapter.

Short-circuit evaluation

The short-circuit evaluation is used when calculating Boolean expressions. It simply means that only necessary conditions need to be evaluated. For example, if **condition1** is true, then in "**condition1 || condition2**", the **condition2** does not need to be evaluated since the whole expression will be true regardless of the **condition2**.

Similarly, if **condition1** is false, then in "**condition1 && condition2**", the **condition2** does not need to be evaluated as well since the whole expression will be false anyway. This is implemented in Code Listing 23.

Code Listing 23: The implementation of the short-circuit evaluation

```
static void UpdateIfBool(ParsingScript script, Variable current)
{
  if ((current.Action == "&&" && current.Value == 0.0) ||
      (current.Action == "||" && current.Value != 0.0)) {
    // Short-circuit evaluation: don't need to evaluate anymore.
    Utils.SkipRestExpr(script);
    current.Action = Constants.NULL_ACTION;
  }
}
```

The method **`Utils.SkipRestExpr`** just skips the rest of the expression after **"&&"** or **"||"**. The implementation is analogous to the implementation of the **`SkipBlock`** and **`SkipRestBlocks`** we saw before, so we won't show it here—nevertheless, it can be consulted in the accompanying source code.

Where do we call this new **`UpdateIfBool`** method from? It's called from the **`Split`** method, shown in Code Listing 4, right before adding the new item to the list:

```
        UpdateIfBool(script, cell);
        listToMerge.Add(cell);
```

UML class diagrams

Next we show the URL diagrams of the parser classes that we have seen so far, and that we are going to see in the rest of this book. Figure 3 contains the overall structure of the main classes used.

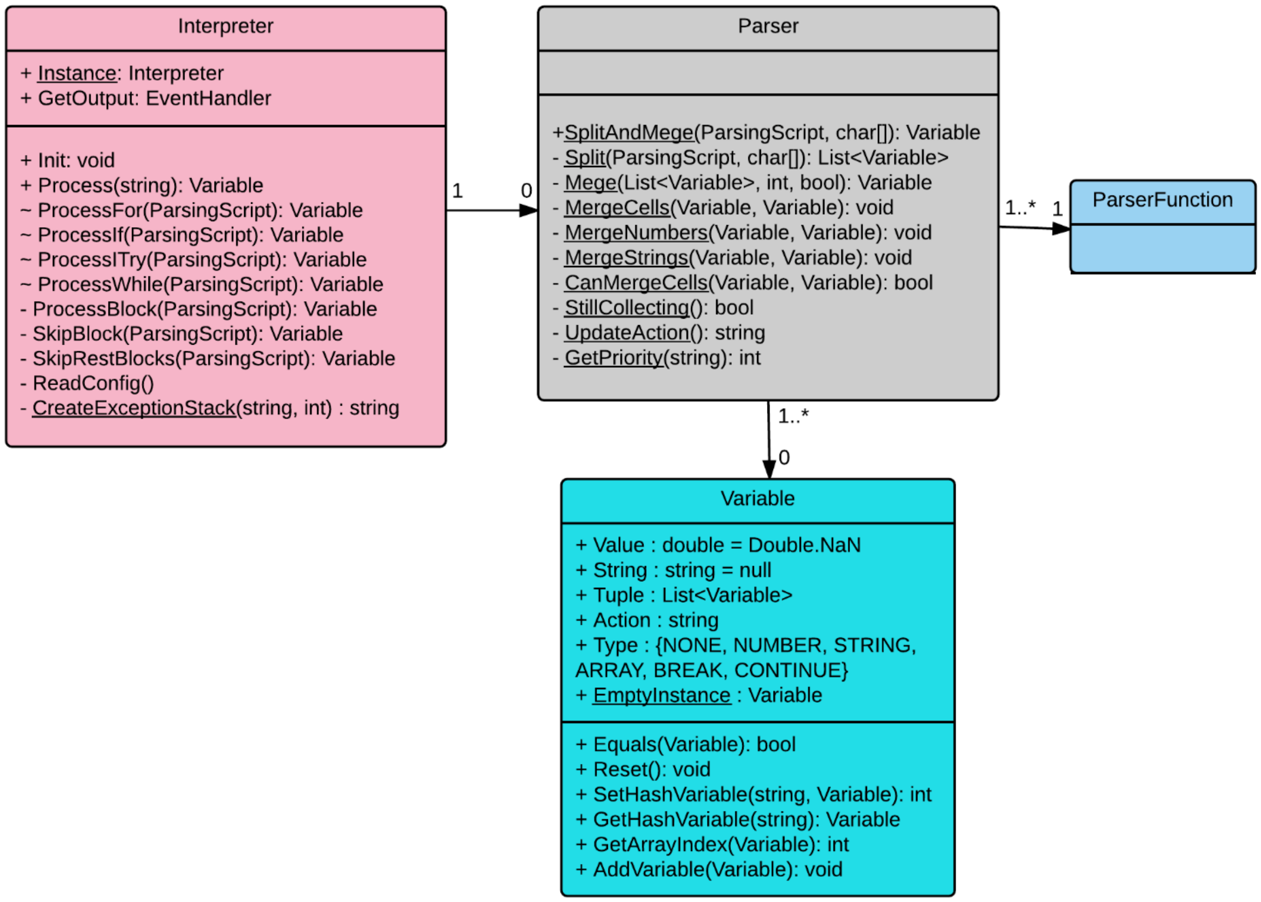

Interpreter
+ Instance: Interpreter
+ GetOutput: EventHandler
+ Init: void
+ Process(string): Variable
~ ProcessFor(ParsingScript): Variable
~ ProcessIf(ParsingScript): Variable
~ ProcessITry(ParsingScript): Variable
~ ProcessWhile(ParsingScript): Variable
- ProcessBlock(ParsingScript): Variable
- SkipBlock(ParsingScript): Variable
- SkipRestBlocks(ParsingScript): Variable
- ReadConfig()
- CreateExceptionStack(string, int) : string
1
0
Parser
+SplitAndMege(ParsingScript, char[]): Variable
- Split(ParsingScript, char[]): List<Variable>
- Mege(List<Variable>, int, bool): Variable
- MergeCells(Variable, Variable): void
- MergeNumbers(Variable, Variable): void
- MergeStrings(Variable, Variable): void
- CanMergeCells(Variable, Variable): bool
- StillCollecting(): bool
- UpdateAction(): string
- GetPriority(string): int
1..*
1
ParserFunction
1..*
0
Variable
+ Value : double = Double.NaN
+ String : string = null
+ Tuple : List<Variable>
+ Action : string
+ Type : {NONE, NUMBER, STRING, ARRAY, BREAK, CONTINUE}
+ EmptyInstance : Variable
+ Equals(Variable): bool
+ Reset(): void
+ SetHashVariable(string, Variable): int
+ GetHashVariable(string): Variable
+ GetArrayIndex(Variable): int
+ AddVariable(Variable): void

Figure 3: Overall structure of parser classes

Figure 4 contains the **`ParserFunction`** class and all the classes deriving from it. It's not complete since the number of descendants can be virtually limitless; for example, all possible mathematical functions, like power, round, logarithm, cosine, and so on, belong all there as well. In order to add a new functionality to the parser, you extend the **`ParserFunction`** class, and the new class will belong to the Figure 4.

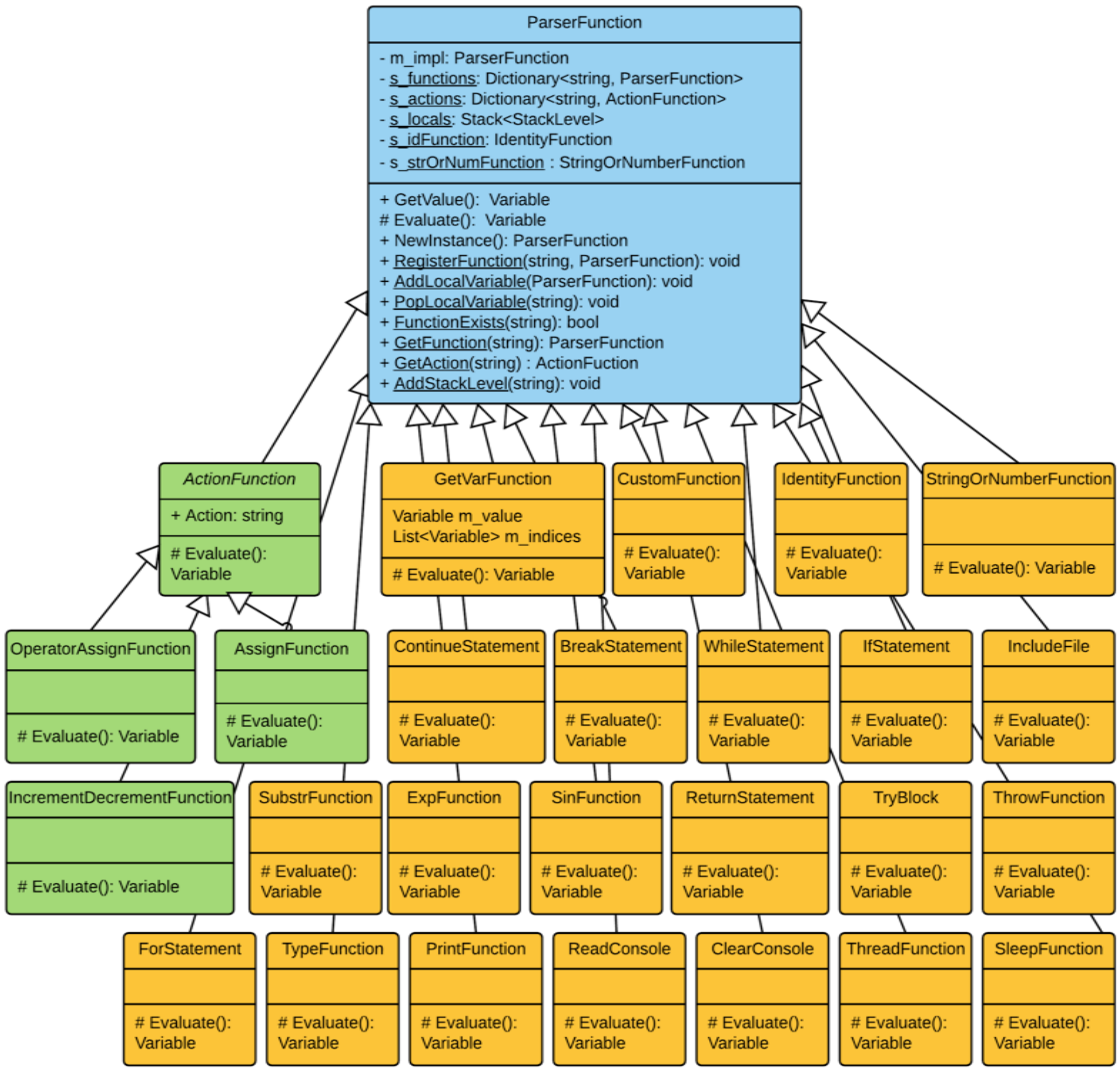


Figure 4: The ParserFunction class and its descendants

Conclusion

In this chapter we implemented some of the basic control flow statements. We implemented all of them as functions, using the **Split-and-Merge** algorithm developed in the previous chapter.

In the next chapter we are going to add some more power to our language, implementing different functions.

Chapter 4 Functions, Functions, Functions

"Developers, developers, developers!"
Steve Ballmer, encouraging more developers for Windows Phone

As you saw in the previous chapter, implementing a class deriving from the **`ParserFunction`** class is the main building block of the programming language we are creating.

In this chapter we are going to see how to implement some popular language constructs, all using the same technique: creating a new class, deriving from the **`ParserFunction`** class, registering it with the parser, and so on. The more functions we can create using this approach, the more power and flexibility will be added to the language.

Writing to the console

Code Listing 24 shows an example of a very useful function implemented in C#, which can be called from the CSCS code.

Code Listing 24: The implementation of the PrintFunction class

```
class PrintFunction : ParserFunction
{
  internal PrintFunction(bool newLine = true)
  {
      m_newLine = newLine;
  }

  internal PrintFunction(ConsoleColor fgcolor)
  {
    m_fgcolor = fgcolor;
    m_changeColor = true;
  }

  protected override Variable Evaluate(ParsingScript script)
  {
    bool isList;
    List<Variable> args = Utils.GetArgs(script,
        Constants.START_ARG, Constants.END_ARG, out isList);

    string output = string.Empty;
    for (int i = 0; i < args.Count; i++) {
      output += args[i].AsString();
    }

    output += (m_newLine ? Environment.NewLine : string.Empty);
    if (m_changeColor) {
      Utils.PrintColor(output, m_fgcolor);
    } else {
      Console.Write(output);
    }

    return Variable.EmptyInstance;
  }

  private bool m_newLine     = true;
  private bool m_changeColor = false;
  private ConsoleColor m_fgcolor;
}

public static void PrintColor(string output, ConsoleColor fgcolor)
{
  ConsoleColor currentForeground = Console.ForegroundColor;
  Console.ForegroundColor = fgcolor;

  Console.Write(output);

  Console.ForegroundColor = currentForeground;
}
```

To extract the arguments (what to print), the **`Utils.GetArgs`** auxiliary function is called (see Code Listing 25). It returns a list of variables whose values have already been calculated by calling the whole **Split-and-Merge** algorithm recursively, if necessary. For example, if the argument list is **`“25, “fu” + “bar”, sin(5*2 – 10)”`**, the **`Utils.GetArgs`** function will return the following list: **`{25, “fubar”, 0}`** (because **`sin(0) = 0`**).

Code Listing 25: The implementation of the Utils.GetArgs method

```
public static List<Variable> GetArgs(ParsingScript script,
                             char start, char end, out bool isList) {
  List<Variable> args = new List<Variable>();
  isList = script.StillValid() && script.Current == Constants.START_GROUP;

  if (!script.StillValid() || script.Current == Constants.END_STATEMENT) {
    return args;
  }

  ParsingScript tempScript = new ParsingScript(script.String,
                                               script.Pointer);
  Utils.GetBodyBetween(tempScript, start, end);
  // After the statement above, tempScript.Parent will point to the last
  // character belonging to the body between start and end characters.
  while (script.Pointer < tempScript.Pointer)  {
    Variable item = Utils.GetItem(script);
    args.Add(item);
  }

  if (script.Pointer <= tempScript.Pointer) {
    // Eat closing parenthesis, if there is one, but only if it closes
    // the current argument list, not one after it.
    script.MoveForwardIf(Constants.END_ARG);
  }

  script.MoveForwardIf(Constants.SPACE);
  return args;
}
```

To print a CSCS variable, we have to represent it as a string. It's easy to do that if the variable is a string or a number. But what if it's an array? Then we convert each element of the array to a string, possibly using recursion (see Code Listing 26). If the parameter **`isList`** is set to **`true`**, opening and closing curly braces will be added to the result.

Code Listing 26: The implementation of the Variable.AsString method

```
public string AsString(bool isList   = true,
                       bool sameLine = true) {
  if (Type == VarType.NUMBER) {
      return Value.ToString();
  }

  if (Type == VarType.STRING) {
    return String;
  }

  if (Type == VarType.NONE || m_tuple == null) {
    return string.Empty;
  }

  StringBuilder sb = new StringBuilder();

  if (isList) {
    sb.Append(Constants.START_GROUP.ToString() +
             (sameLine ? "" : Environment.NewLine));
  }

  for (int i = 0; i < m_tuple.Count; i++) {
    Variable arg = m_tuple[i];
    sb.Append(arg.AsString(isList, sameLine));
    if (i != m_tuple.Count - 1) {
      sb.Append(sameLine ? " " : Environment.NewLine);
    }
  }

  if (isList) {
    sb.Append(Constants.END_GROUP.ToString() +
              (sameLine ? " " : Environment.NewLine));
  }

  return sb.ToString();
}
```

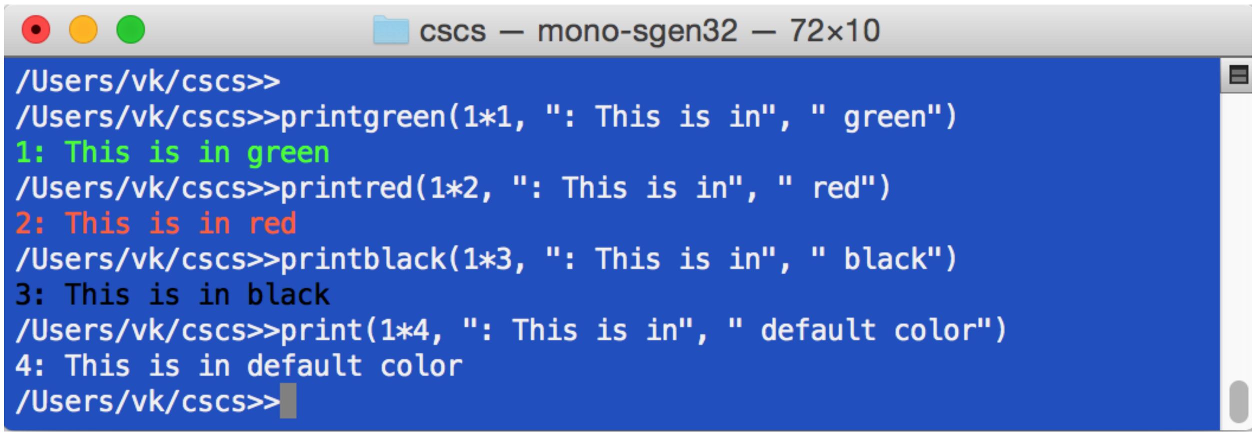
This is how we register printing functions, printing in a specified or in a default color:

```
ParserFunction.RegisterFunction(Constants.PRINT,
                                new PrintFunction());
ParserFunction.RegisterFunction(Constants.PRINT_BLACK,
                                new PrintFunction(ConsoleColor.Black));
ParserFunction.RegisterFunction(Constants.PRINT_GREEN,
                                new PrintFunction(ConsoleColor.Green));
ParserFunction.RegisterFunction(Constants.PRINT_RED,
                                new PrintFunction(ConsoleColor.Red));
```

The constants are defined as follows:

```
public const string PRINT       = "print";
public const string PRINT_BLACK = "printblack";
public const string PRINT_GREEN = "printgreen";
public const string PRINT_RED   = "printred";
```

Analogously, you can add printing any other color you wish. Figure 5 contains a sample session with CSCS playing with different colors.



```
/Users/vk/cscs>>
/Users/vk/cscs>>printgreen(1*1, ": This is in", " green")
1: This is in green
/Users/vk/cscs>>printred(1*2, ": This is in", " red")
2: This is in red
/Users/vk/cscs>>printblack(1*3, ": This is in", " black")
3: This is in black
/Users/vk/cscs>>print(1*4, ": This is in", " default color")
4: This is in default color
/Users/vk/cscs>>
```

Figure 5: Printing in different colors

Reading from the console

Now let’s see a sister of the writing to the console function—reading from the console. Details below in Code Listing 27.

Code Listing 27: The implementation of the ReadConsole class

```
class ReadConsole : ParserFunction
{
  internal ReadConsole(bool isNumber = false)
  {
    m_isNumber = isNumber;
  }

  protected override Variable Evaluate(ParsingScript script)
  {
    script.Forward(); // Skip opening parenthesis.
    string line = Console.ReadLine();

    if (!m_isNumber) {
      return new Variable(line);
    }

    double number = Double.NaN;

    if (!Double.TryParse(line, out number)) {
      throw new ArgumentException("Couldn't parse number [" + line + "]");
    }
    return new Variable(number);
  }

  private bool m_isNumber;
}
```

There are two cases: when we read a number, and when we read a string. Correspondingly, we register two functions with the parser:

```
public const string READ       = "read";
public const string READNUMBER = "readnum";

ParserFunction.RegisterFunction(Constants.READ,       new ReadConsole());
ParserFunction.RegisterFunction(Constants.READNUMBER, new ReadConsole(true));
```

Analogously, we can add functions to read different data structures, for example, to read the whole array of numbers or strings.

Next we’ll see an example of a CSCS script using different constructs that we’ve implemented so far.

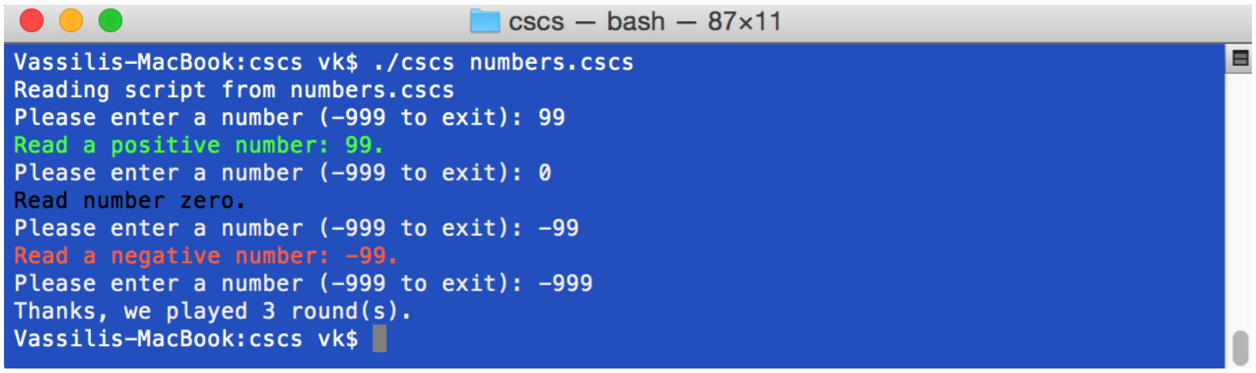
An example of the CSCS code

Code Listing 28 shows an example of the CSCS code using **`while`**, **`break`**, and **`if`** control flow structures that we discussed in the previous chapter, and printing in different colors.

Code Listing 28: CSCS code to print in different colors depending on the user input

```
round = 0;

while(1) {
  write("Please enter a number (-999 to exit): ");
    number = readnum();

  if (number == -999) {
    break;
  } elif (number < 0) {
    printred("Read a negative number: ", number, ".");
  } elif (number > 0) {
    printgreen("Read a positive number: ", number, ".");
  } else {
    printblack("Read number zero.");
  }
  round++;
}

print("Thanks, we played ", round, " round(s).");
```

To test our script, we save the contents of Code Listing 28 to the **numbers.cscs** file and then run it, as shown in Figure 6.



```
Vassilis-MacBook:cscs vk$ ./cscs numbers.cscs
Reading script from numbers.cscs
Please enter a number (-999 to exit): 99
Read a positive number: 99.
Please enter a number (-999 to exit): 0
Read number zero.
Please enter a number (-999 to exit): -99
Read a negative number: -99.
Please enter a number (-999 to exit): -999
Thanks, we played 3 round(s).
Vassilis-MacBook:cscs vk$
```

Figure 6: Sample run of the numbers.cscs script

Working with strings in CSCS

In this section we are going to see some of the examples of classes deriving from the **`ParserFunctions`** class that implement string-related functionality. We are going to see that most of this functionality is already implemented in C#, so our classes are just wrappers over the existing C# functions and methods.

Code Listing 29: The implementation of the ToUpperFunction class

```
class ToUpperFunction : ParserFunction
{
  protected override Variable Evaluate(ParsingScript script)
  {
    // 1. Get the name of the variable.
    string varName = Utils.GetToken(script, Constants.END_ARG_ARRAY);
    Utils.CheckNotEmpty(script, varName, m_name);

    // 2. Get the current value of the variable.
    ParserFunction func = ParserFunction.GetFunction(varName);
    Variable currentValue = func.GetValue(script);

    // 3. Take either the string part if it is defined,
    // or the numerical part converted to a string otherwise.
    string arg = currentValue.AsString();

    Variable newValue = new Variable(arg.ToUpper());
    return newValue;
  }
}
```

Code Listing 30 provides an implementation of converting all characters of a string to the upper case. The **`Utils.GetToken`** is a convenience method that extracts the next string token from the parsing script (it doesn't call the **Split-and-Merge** algorithm). It's a relatively straightforward function, and you can check out its implementation in the accompanying source code.

Code Listing 30 implements the **`IndexOfFunction`** class to search for a substring in a string. The C# **`String.IndexOf`** function, which is used there, has many more optional parameters (for example, from which character to search or which type of comparison to use). These options can be easily added to the **`IndexOfFunction`**.

Code Listing 30: The implementation of the IndexOfFunction class

```
class IndexOfFunction : ParserFunction
{
  protected override Variable Evaluate(ParsingScript script)
  {
    // 1. Get the name of the variable.
    string varName = Utils.GetToken(script, Constants.NEXT_ARG_ARRAY);
    Utils.CheckNotEmpty(script, varName, m_name);

    // 2. Get the current value of the variable.
    ParserFunction func = ParserFunction.GetFunction(varName);
    Variable currentValue = func.GetValue(script);
    // 3. Get the value to be looked for.
    Variable searchValue = Utils.GetItem(script);

    // 4. Apply the corresponding C# function.
    string basePart = currentValue.AsString();
    string search = searchValue.AsString();

    int result = basePart.IndexOf(search);
    return new Variable(result);
  }
}
```

Code Listing 31 shows the implementation of the substring function. Note that it has two arguments: the user can optionally supply the length of the substring.

Code Listing 31: The implementation of the SubstrFunction class

```
class SubstrFunction : ParserFunction
{
  protected override Variable Evaluate(ParsingScript script)
  {
    // 1. Get the name of the variable.
    string varName = Utils.GetToken(script, Constants.NEXT_ARG_ARRAY);
    Utils.CheckNotEmpty(script, varName, m_name);

    // 2. Get the current value of the variable.
    ParserFunction func = ParserFunction.GetFunction(varName);
    Variable currentValue = func.GetValue(script);

    // 3. Take either the string part if it is defined,
    // or the numerical part converted to a string otherwise.
    string arg = currentValue.AsString();
    // 4. Get the initial index of the substring.
    Variable init = Utils.GetItem(script);
    Utils.CheckNonNegativeInt(init);

    // 5. Get the length of the substring if available.
    string substring;
    bool lengthAvailable = Utils.SeparatorExists(script);

    if (lengthAvailable) {
      Variable length = Utils.GetItem(script);
      Utils.CheckPosInt(length);

      if (init.Value + length.Value > arg.Length) {
        throw new ArgumentException("The total  length is larger han [" +
                                     arg + "]");
      }
      substring = arg.Substring((int)init.Value, (int)length.Value);
    } else {
      substring = arg.Substring((int)init.Value);
    }

    Variable newValue = new Variable(substring);

    return newValue;
  }
}
```

Then we register **ToUpperFunction**, **IndexOfFunction**, and **SubstrFunction** with the parser as follows:

```
public const string TOUPPER  = "toupper";
public const string INDEX_OF = "indexof";
public const string SUBSTR   = "substr";
ParserFunction.RegisterFunction(Constants.TOUPPER,  new ToUpperFunction());
ParserFunction.RegisterFunction(Constants.INDEX_OF, new IndexOfFunction());
ParserFunction.RegisterFunction(Constants.SUBSTR,   new SubstrFunction());
```

Code Listing 31 calls a few auxiliary functions that are used extensively in our implementation of CSCS, mostly to save on typing. Some of these functions are shown inCode Listing 32.

Code Listing 32: Functions to check for correct input

```
public static void CheckNonNegativeInt(Variable variable)
{
  CheckInteger(variable);
  if (variable.Value < 0) {
    throw new ArgumentException(
        "Expected a non-negative integer instead of [" +
         variable.Value + "]");
  }
}

public static void CheckInteger(Variable variable)
{
  CheckNumber(variable);
  if (variable.Value % 1 != 0.0) {
    throw new ArgumentException("Expected an integer instead of [" +
                                variable.Value + "]");
  }
}

public static void CheckNumber(Variable variable)
{
  if (variable.Type != Variable.VarType.NUMBER) {
      throw new ArgumentException ("Expected a number instead of [" +
                                   variable.AsString() + "]");
  }
}

public static void CheckNotEmpty(ParsingScript script, string varName,
                                 string name) {
  if (!script.StillValid() || string.IsNullOrWhiteSpace(varName)) {
    throw new ArgumentException("Incomplete arguments for [" + name + "]");
  }
}
```

Code Listing 33 shows examples of using some of the string functions in CSCS.

Code Listing 33: A sample CSCS code for working with strings

```
str = "Perl - The only language that looks the same before and after " +
      "RSA encryption. -- Keith Bostic";
index = indexof(str, "language");
res   = "cscs " + substr(str, index, 8);
print(toupper(res)); // prints "CSCS LANGUAGE"
```

Mathematical functions in CSCS

Implementing a mathematical function in CSCS is even easier than implementing a string function. See Code Listing 34, which shows how to implement a function to round a number to the closest integer.

The only missing part is to define a name for this function and register it with the parser:

```
public const string ROUND = "round";
ParserFunction.RegisterFunction(Constants.ROUND, new RoundFunction());
```

Code Listing 34: The implementation of the RoundFunction class

```
class RoundFunction : ParserFunction
{
  protected override Variable Evaluate(ParsingScript script)
  {
    Variable arg = script.ExecuteTo(Constants.END_ARG);
    arg.Value = Math.Round(arg.Value);
    return arg;
  }
}
```

Finding variable types in CSCS

All of the variables in CSCS have a corresponding **`Variable`** object in C# code. It would be convenient to be able to extract some properties of this variable, such as its type. Code Listing 35 implements this functionality. The **`Evaluate`** method of the **`TypeFunction`** class has a few method invocations that deal with arrays. We are going to look more closely at arrays in the “Using arrays in CSCS” section in Chapter 6 Operators, Arrays, and Dictionaries.

The only missing part is to define a name for this function and register it with the parser:

```
public const string TYPE = "type";
ParserFunction.RegisterFunction(Constants.TYPE, new TypeFunction());
```

Code Listing 35: The implementation of the TypeFunction class

```
class TypeFunction : ParserFunction
{
  protected override Variable Evaluate (ParsingScript script)
  {
    // 1. Get the name of the variable.
    string varName = Utils.GetToken(script, Constants.END_ARG_ARRAY);
    Utils.CheckNotEmpty(script, varName, m_name);

    List<Variable> arrayIndices = Utils.GetArrayIndices(ref varName);

    // 2. Get the current value of the variable.
    ParserFunction func = ParserFunction.GetFunction(varName);
    Utils.CheckNotNull(varName, func);
    Variable currentValue = func.GetValue(script);
    Variable element = currentValue;

    // 2b. Special case for an array.
    if (arrayIndices.Count > 0) {// array element
      element = Utils.ExtractArrayElement(currentValue, arrayIndices);
      script.MoveForwardIf(Constants.END_ARRAY);
    }

    // 3. Convert type to string.
    string type = Constants.TypeToString(element.Type);
    script.MoveForwardIf (Constants.END_ARG, Constants.SPACE);

    Variable newValue = new Variable(type);
    return newValue;
  }
}
```

As you can see, basically, you can implement in CSCS anything that can be implemented in C#.

Also note that the CSCS language we are creating can be easily converted to a functional programming language. In a functional language, the output value of a function depends only on the arguments, so the change would be not to have global variables that can have different values between two function calls.

Working with threads in CSCS

"Giving pointers and threads to programmers is like giving whisky and car keys to teenagers."
P. J. O'Rourke

In this section we are going to see on some of the thread functions. In particular, you'll see that the syntax of creating a new thread can be very simple.

In Code Listing 36 we define three thread-related functions: **`ThreadFunction`** (to create and start a new thread; returns the id of the newly created thread), **`ThreadIDFunction`** (to get and return the thread id of the current thread), and **`SleepFunction`** (to put thread execution to sleep for some number of milliseconds).

Code Listing 36: The implementation of the Thread-related classes

```
class ThreadFunction : ParserFunction
{
  protected override Variable Evaluate(ParsingScript script)
  {
    string body = Utils.GetBodyBetween(script, Constants.START_GROUP,
                                               Constants.END_GROUP);
    Thread newThread = new Thread(ThreadFunction.ThreadProc);
    newThread.Start(body);
    return new Variable(newThread.ManagedThreadId);
  }

  static void ThreadProc(Object stateInfo)
  {
    string body = (string)stateInfo;
    ParsingScript threadScript = new ParsingScript(body);
    threadScript.ExecuteAll();
  }
}

class ThreadIDFunction : ParserFunction
{
  protected override Variable Evaluate(ParsingScript script)
  {
    int threadID = Thread.CurrentThread.ManagedThreadId;
    return new Variable(threadID);
  }
}

class SleepFunction : ParserFunction
{
  protected override Variable Evaluate(ParsingScript script)
  {
    Variable sleepms = Utils.GetItem(script);
    Utils.CheckPosInt(sleepms);
    Thread.Sleep((int)sleepms.Value);
    return Variable.EmptyInstance;
  }
}
```

As you can see, all of these classes are thin wrappers over their C# counterparts. The argument of the thread function can be any CSCS script, including a CSCS function (we'll see functions in CSCS in

Chapter 5 Exceptions and Custom Functions). This is how we register thread functions with the parser:

```
public const string THREAD    = "thread";
public const string THREAD_ID = "threadid";
public const string SLEEP     = "sleep";
ParserFunction.RegisterFunction(Constants.THREAD, new ThreadFunction());
ParserFunction.RegisterFunction(Constants.THREAD_ID, new ThreadIDFunction());
ParserFunction.RegisterFunction(Constants.SLEEP, new SleepFunction());
```

Code Listing 37 contains sample code using the three functions above. It has a `for` loop, which we are going to see in the "Implementing the for-" section in Chapter 6.

Code Listing 37: Playing with threads in CSCS

```
for (i = 0; i < 5; i++) {
  id = thread(
         print("-->> Starting in thread", threadid());
         sleep(1000);
         print("<<-- Finishing in thread", threadid());
       );
  print("Started thread", id, " in main");
}

sleep(2000);
print("Main completed in thread", threadid());

// Sample run of the script above:
Started thread3 in main
-->> Starting in thread3
Started thread4 in main
-->> Starting in thread4
Started thread5 in main
-->> Starting in thread5
Started thread6 in main
-->> Starting in thread6
```

```
Started thread7 in main
-->> Starting in thread7
<<-- Finishing in thread3
<<-- Finishing in thread5
<<-- Finishing in thread6
<<-- Finishing in thread4
<<-- Finishing in thread7
Main completed in thread1
```

Inter-thread communication in CSCS

Once you know how to start a new thread, you would probably want to be able to share resources and send signals among threads (for example, when a thread finishes processing of a task).

The former can be implemented in C# using locks, and the latter using event wait handles. The advantage of using custom locking and synchronization mechanisms in our scripting language is that you can have an even simpler syntax than the one in C#, as you'll see soon.

Code Listing 38 defines **`LockFunction,`** which implements the thread-locking functionality, and **`SignalWaitFunction`**, which implements signaling between threads (this functionality is also commonly known as a "condition variable" in other languages).

From here you can get more fancy and implement, for instance, a readers-writer lock, when you allow a read-access to a resource to multiple threads and a write access to just one thread.

Code Listing 38: The implementation of the LockFunction and SignalWaitFunction classes

```
class LockFunction : ParserFunction
{
  static Object lockObject = new Object();

  protected override Variable Evaluate(ParsingScript script)
  {
    string body = Utils.GetBodyBetween(script, Constants.START_ARG,
                                              Constants.END_ARG);
    ParsingScript threadScript = new ParsingScript(body);

    lock(lockObject) {
      threadScript.ExecuteAll();
    }
    return Variable.EmptyInstance;
  }
}

class SignalWaitFunction : ParserFunction
{
  static AutoResetEvent waitEvent = new AutoResetEvent(false);
  bool m_isSignal;

  public SignalWaitFunction(bool isSignal)
  {
    m_isSignal = isSignal;
  }
  protected override Variable Evaluate(ParsingScript script)
  {
    bool result = m_isSignal ? waitEvent.Set() :
                               waitEvent.WaitOne();
    return new Variable(result);
  }
}
```

This is how we register these functions with the parser:

```
public const string LOCK   = "lock";
public const string SIGNAL = "signal";
public const string WAIT   = "wait";
ParserFunction.RegisterFunction(Constants.LOCK, new LockFunction());
ParserFunction.RegisterFunction(Constants.SIGNAL,
                                new SignalWaitFunction(true));
ParserFunction.RegisterFunction(Constants.WAIT,
                                new SignalWaitFunction(false));
```

39 contains sample code using the **`SignalWaitFunction`** function. It uses a custom CSCS function **`threadWork`**. We'll see custom functions in CSCS in Chapter 5: Exceptions and Custom Functions.

Code Listing 39: Thread synchronization example in CSCS

```
function threadWork()
{
  print("  Starting thread work in thread", threadid());
  sleep(2000);
  print("  Finishing thread work in thread", threadid());
  signal();
}

print("Main, starting new thread from ", threadid());
thread(threadWork());
print("Main, waiting for thread in ", threadid());
wait();
print("Main, wait returned in ", threadid());

// Sample run of the script above:
Reading script from scripts/temp.cscs
```

```
Main, starting new thread from 1
Main, waiting for thread in 1
  Starting thread work in thread3
  Finishing thread work in thread3
Main, wait returned in 1
```

Conclusion

In this chapter we implemented a few useful functions: writing to the console in different colors, reading from the console, implementing mathematical functions, working with strings, threads, and some others. The goal was to show that anything that can be implemented in C#, can be implemented in CSCS as well.

In the next chapter we are going to see how to implement exceptions and custom functions, how to add local and global files, and a few examples.

Chapter 5 Exceptions and Custom Functions

“The primary duty of an exception handler is to get the error out of the lap of the programmer and into the surprised face of the user.”
Verity Stob

In this chapter we are going to see how to implement custom functions and methods in CSCS. We’ll also see how to throw and catch an exception, since when implementing exception stack, we need to use the information about functions and methods being called (“on the stack”).

Related functions and methods are usually implemented together, in the same file, so it would be a nice feature to include code from different files. We’ll see how to do that next.

Including files

To implement including file functionality, we use the same algorithm we already used with other functions—making the implementation in a class deriving from the **`ParserFunction`** class.

The **`IncludeFile`** derives from the **`ParserFunction`** class (see also the UML diagram in Figure 4). Check out its implementation in Code Listing 40.

Code Listing 40: The implementation of the IncludeFile class

```
class IncludeFile : ParserFunction
{
  protected override Variable Evaluate(ParsingScript script)
  {
    string filename = Utils.GetItem(script).AsString();
    string[] lines = File.ReadAllLines(filename);

    string includeFile = string.Join(Environment.NewLine, lines);
    Dictionary<int, int> char2Line;
    string includeScript = Utils.ConvertToScript(includeFile,
                                                 out char2Line);
    ParsingScript tempScript = new ParsingScript(includeScript, 0,
                                                 char2Line);
    tempScript.Filename = filename;
    tempScript.OriginalScript = string.Join(
                             Constants.END_LINE.ToString(), lines);

    while (tempScript.Pointer < includeScript.Length) {
      tempScript.ExecuteTo();
      tempScript.GoToNextStatement();
    }
    return Variable.EmptyInstance;
  }
}
```

The **Utils.GetItem** method is, on one hand, just a wrapper over the **Parser.SplitAndMerge** method. Additionally, it takes care of the string expressions between quotes. On the other hand, it also converts an expression between curly braces to an array.

Code Listing 41: The implementation of the GetItem method

```
public static Variable GetItem(ParsingScript script)
{
  script.MoveForwardIf(Constants.NEXT_ARG, Constants.SPACE);
  Utils.CheckNotEnd(script);

  Variable value = new Variable();
  bool inQuotes = script.Current == Constants.QUOTE;

  if (script.Current == Constants.START_GROUP)  {
    // We are extracting a list between curly braces.
    script.Forward(); // Skip the first brace.
    bool isList = true;
    value.Tuple = GetArgs(script, Constants.START_GROUP,
                          Constants.END_GROUP, out isList);
    return value;
  }
  else {
    // A variable, a function, or a number.
    Variable var = script.Execute(Constants.NEXT_OR_END_ARRAY);
    value.Copy(var);
  }

  if (inQuotes) {
    script.MoveForwardIf(Constants.QUOTE);
  }

  script.MoveForwardIf(Constants.SPACE);
  return value;
}
```

In addition, the **IncludeFile.Evaluate** method invokes the **Utils.ConvertToScript** method, shown in Code Listing 42.

It's actually one of the key methods in understanding how the CSCS language works. It shows the first, preprocessing step of the script to be parsed. Basically, the method translates the passed string to another string, which our parser can understand. Among other things, this method removes all the text in comments and all unnecessary blanks, like spaces, tabs, or new lines.

Code Listing 42: The implementation of the Utils.ConvertToScript method

```
public static string ConvertToScript(string source,
                                     out Dictionary<int, int> char2Line)
{
  StringBuilder sb = new StringBuilder(source.Length);
  char2Line = new Dictionary<int, int>();

  bool inQuotes         = false;
  bool spaceOK          = false;
  bool inComments       = false;
  bool simpleComments   = false;
  char previous         = Constants.EMPTY;
  int parentheses       = 0;
  int groups            = 0;
  int lineNumber        = 0;
  int lastScriptLength = 0;

  for (int i = 0; i < source.Length; i++)  {
    char ch = source[i];
    char next = i + 1 < source.Length ? source[i + 1] : Constants.EMPTY;
    if (ch == '\n') {
      if (sb.Length > lastScriptLength) {
        char2Line[sb.Length - 1] = lineNumber;
        lastScriptLength = sb.Length;
      }
      lineNumber++;
    }
    if (inComments && ((simpleComments && ch != '\n') ||
                      (!simpleComments && ch != '*'))) {
      continue;
    }

    switch (ch) {
      case '/':
        if (inComments || next == '/' || next == '*')  {
          inComments = true;
          simpleComments = simpleComments || next == '/';
          continue;
        }
        break;
      case '*':
        if (inComments && next == '/') {
          i++; // skip next character
          inComments = false;
          continue;
        }
        break;
      case '“':
```

```
      case '”':
      case '"':
        ch = '"';
        if (!inComments && previous != '\\') {
          inQuotes = !inQuotes;
        }
        break;
      case ' ':
        if (inQuotes) {
          sb.Append (ch);
        } else {
          spaceOK = KeepSpace(sb, next)|| (previous != Constants.EMPTY &&
                              previous != Constants.NEXT_ARG && spaceOK);
          bool spaceOKonce = KeepSpaceOnce(sb, next);
          if (spaceOK || spaceOKonce) sb.Append(ch);
        }
        continue;
      case '\t':
      case '\r':
        if (inQuotes) sb.Append(ch);
        continue;
      case '\n':
        if (simpleComments) {
          inComments = simpleComments = false;
        }
        spaceOK     = false;
        continue;
      case Constants.END_ARG:
        if (!inQuotes) {
          parentheses--;
          spaceOK = false;
        }
        break;
      case Constants.START_ARG:
        if (!inQuotes) parentheses++;
        break;
      case Constants.END_GROUP:
        if (!inQuotes) {
          groups--;
          spaceOK = false;
        }
        break;
      case Constants.START_GROUP:
        if (!inQuotes) groups++;
        break;
      case Constants.END_STATEMENT:
        if (!inQuotes) spaceOK = false;
        break;
      default: break;
    }
    if (!inComments) {
      sb.Append(ch);
    }
    previous = ch;
  }
  // Here we can throw an exception if the “parentheses” is not 0.
  // Same for “groups”. Nonzero means there are some unmatched
  // parentheses or curly braces.
  return sb.ToString();
}
```

The **`Utils.CovertToScript`** method uses an auxiliary **`char2Line`** dictionary. It’s needed to have a reference to the original line numbers in the parsing script in case an exception is thrown (or the code is just wrong), so that the user knows in which line the problem occurred. We’ll see it in more detail in the "Getting line numbers where errors occur” section in Chapter 7: Localization.

If you want to define a new style for comments, it will be in this method. This method also allows the user to have different characters for a quote character: “ and ” characters are both replaced by the " character, which is the only one that our parser understands.

How do we know which spaces must be removed and which not? First of all, if we are inside of quotes, we leave everything as is, because the expression is just a string value.

In other cases, we leave at most one space between tokens. For some functions we need spaces to separate tokens, but for others, spaces aren’t needed and tokens are separated by other characters, for example, by an opening parenthesis.

An example of a function that requires a space as a separation token is a change directory function: **`cd C:\Windows`**. A function that doesn’t require a space to separate tokens is any mathematical function, for instance, in **`sin(2*10)`**. A space between the sine function and the opening parenthesis is never needed.

Code Listing 43 contains auxiliary functions used in the **`CovertToScript`** method that determine whether we need to keep a space or not.

Code Listing 43: The implementation of the keeping space functions

```
public static bool EndsWithFunction(string buffer, List<string> functions)
{
  foreach (string key in functions) {
    if (buffer.EndsWith(key, StringComparison.OrdinalIgnoreCase)) {
      char prev = key.Length >= buffer.Length ?
        Constants.END_STATEMENT :
        buffer [buffer.Length - key.Length - 1];
      if (Constants.TOKEN_SEPARATION.Contains(prev)) {
        return true;
      }
    }
  }
  return false;
}

public static bool SpaceNotNeeded(char next)
{
  return (next == Constants.SPACE || next == Constants.START_ARG ||
    next == Constants.START_GROUP || next == Constants.START_ARRAY ||
    next == Constants.EMPTY);
}

public static bool KeepSpace(StringBuilder sb, char next)
{
  if (SpaceNotNeeded(next)) {
    return false;
  }

  return EndsWithFunction(sb.ToString(), Constants.FUNCT_WITH_SPACE);
}

public static bool KeepSpaceOnce(StringBuilder sb, char next)
{
  if (SpaceNotNeeded(next)) {
    return false;
  }
  return EndsWithFunction(sb.ToString(), Constants.FUNCT_WITH_SPACE_ONCE);
}
```

As you can see, we distinguish between two types of functions that require a space as a separation token. See Code Listing 44.

Code Listing 44: Functions allowing spaces as separation tokens

```
// Functions that allow a space separator after them, on top of the
// parentheses. The function arguments may have spaces as well,
// e.g. “copy a.txt b.txt”
public static List<string> FUNCT_WITH_SPACE = new List<string> {
  APPENDLINE, CD, CONNECTSRV, COPY, DELETE, DIR, EXISTS,
  FINDFILES, FINDSTR, FUNCTION, MKDIR, MORE, MOVE, PRINT, READFILE, RUN,
  SHOW, STARTSRV, TAIL, TRANSLATE, WRITE, WRITELINE, WRITENL
};

// Functions that allow a space separator after them, on top of the
// parentheses, but only once, i.e. function arguments are not allowed
// to have spaces between them e.g. return a*b; throw exc;
public static List<string> FUNCT_WITH_SPACE_ONCE = new List<string> {
  RETURN, THROW
};
```

Basically, we want to maintain two modes of operation for our language—a command-line language (or a “shell” language, in Unix terms), which uses mostly spaces as a separation criterion between tokens, and a regular, scripting language, which uses parentheses as a separation criterion.

Let’s return to the implementation of the **`IncludeFile`** class in Code Listing 40. Once we’ve converted the string containing the script to a format that we understand (in Code Listing 42), we go over each statement of the script and apply the whole **Split-and-Merge** algorithm to each statement.

To move to the next statement in the script, we use the **`ParsingScript.GoToNextStatement`** auxiliary method (as seen in Code Listing 18). In particular, it deals with cases when the last processed statement is also the last one in a group of statements (between the curly braces), or when we need to get rid of the character separating different statements (defined as **`END_STATEMENT = ';'`** in the **`Constants`** class).

Throwing an exception

To throw an exception, we use the same approach we already used with other control flow functions: we implement the **`ThrowFunction`** class as a **`ParserFunction`** class. The **`ThrowFunction`** class is included in Figure 4, and its implementation is in Code Listing 45.

Code Listing 45: The implementation of the ThrowFunction class

```
class ThrowFunction : ParserFunction
{
  protected override Variable Evaluate(ParsingScript script)
  {
    // 1. Extract what to throw.
    Variable arg = Utils.GetItem(script);

    // 2. Convert it to a string.
    string result = arg.AsString();

    // 3. Throw it!
    throw new ArgumentException(result);
  }
}
```

This class is registered with the parser as follows:

```
public const string THROW = "throw";
ParserFunction.RegisterFunction(Constants.THROW, new ThrowFunction());
```

This means that as soon as our parser sees something like:

```
throw "Critical exception!";
```

In CSCS code, our C# code will throw an exception. How can we catch it?

Catching exceptions

To catch an exception, we must have a try block. The processing of the catch module will follow the processing of the try block. The class **`TryBlock`** is also derived from the **`ParserFunction`** class; see Figure 4. Its implementation is in Code Listing 46. The main functionality is in the **`Interpreter`** class, where we can reuse the already implemented **`ProcessBlock`** (Code Listing 17), **`SkipBlock`** (Code Listing 19), and **`SkipRestBlocks`** (Code Listing 20) methods.

Code Listing 46: The implementation of the try and catch functionality

```
class TryBlock : ParserFunction
{
  protected override Variable Evaluate(ParsingScript script)
  {
    return Interpreter.Instance.ProcessTry(script);
  }
}

internal Variable ProcessTry(ParsingScript script)
{
  int startTryCondition = script.Pointer - 1;
  int currentStackLevel = ParserFunction.GetCurrentStackLevel();

  Exception exception   = null;
  Variable result = null;
  try {
    result = ProcessBlock(script);
  } catch(ArgumentException exc) {
    exception = exc;
  }

  if (exception != null ||
      result.Type == Variable.VarType.BREAK ||
      result.Type == Variable.VarType.CONTINUE) {
    // We are here from the middle of the try-block either because
    // an exception was thrown or because of a Break/Continue. Skip it.
    script.Pointer = startTryCondition;
    SkipBlock(script);
  }

  string catchToken = Utils.GetNextToken(script);
  script.Forward(); // skip opening parenthesis
  // The next token after the try block must be a catch.
  if (!Constants.CATCH_LIST.Contains(catchToken))  {
    throw new ArgumentException("Expecting a 'catch()' but got [" +
                                catchToken + "]");
  }

  string exceptionName = Utils.GetNextToken(script);
  script.Forward(); // skip closing parenthesis

  if (exception != null) {
    string excStack = CreateExceptionStack(exceptionName,
                                           currentStackLevel);
    ParserFunction.InvalidateStacksAfterLevel(currentStackLevel);
    GetVarFunction excFunc = new GetVarFunction(
                             new Variable(exception.Message + excStack));
    ParserFunction.AddGlobalOrLocalVariable(exceptionName, excFunc);

    result = ProcessBlock(script);
    ParserFunction.PopLocalVariable(exceptionName);
  } else {
    SkipBlock (script);
  }

  SkipRestBlocks(script);
  return result;
}
```

The **`TryBlock`** class is registered with the parser as follows:

```
public const string TRY = "try";
ParserFunction.RegisterFunction(Constants.TRY, new TryBlock());
```

When we catch an exception, we then also create an exception stack; see Code Listing 47.

Code Listing 47: The implementation of the Interpreter.CreateExceptionStack method

```
static string CreateExceptionStack(string exceptionName,
                                   int lowestStackLevel)
{
  string result = "";
  Stack<ParserFunction.StackLevel> stack = ParserFunction.ExecutionStack;
  int level = stack.Count;

  foreach (ParserFunction.StackLevel stackLevel in stack) {
    if (level-- < lowestStackLevel) {
      break;
    }
    if (string.IsNullOrWhiteSpace(stackLevel.Name)) {
      continue;
    }
    result += Environment.NewLine + "  " + stackLevel.Name + "()";
  }

  if (!string.IsNullOrWhiteSpace (result)) {
    result = " --> " + exceptionName + result;
  }

  return result;
}
```

In order to use the exception data in CSCS, we add a variable containing exception information. This variable is the **GetVarFunction** class, which we add to the parser:

```
  ParserFunction.AddGlobalOrLocalVariable(exceptionName, excFunc);
```

The **excFunc** variable is of type **GetVarFunction**; see its implementation in Code Listing 48. The **GetVarFunction** class is just a wrapper over the exception being thrown. We register it with the parser using the exception name, so that as soon as the CSCS code accesses the exception by its name, it gets the exception information that we supplied. You can easily add more fancy things to the exception information there, like having separate fields for the exception name and the exception stack. We'll see some examples of exceptions at the end of this chapter.

Code Listing 48: The implementation of the GetVarFunction class

```
class GetVarFunction : ParserFunction
{
  internal GetVarFunction(Variable value)
  {
    m_value = value;
  }

  protected override Variable Evaluate(ParsingScript script)
  {
    return m_value;
  }

  private Variable m_value;
}
```

To make everything work, we have defined a few new data structures to the **ParserFunction** class. In particular, the **StackLevel** class contains all the local variables used inside of a CSCS function; see Code Listing 49.

The **Stack<StackLevel> s_locals** member holds a stack having the local variables for each function being called on the stack. The **AddLocalVariable** and **AddStackLevel** methods add a new local variable and, correspondingly, a new **StackLevel**.

Note: This is the place where you want to disallow local names if there is already a global variable or function with the same name.

The **`Dictionary<string, ParserFunction> s_functions`** member holds all the global variables and functions (in CSCS all variables and functions are the same; both derive from the **`ParserFunction`** class). The keys to the dictionary are the function or variable names. The **`RegisterFunction`** and **`AddGlobal`** methods both add a new variable or a function. There is also the **`isNative`** Boolean flag, which indicates whether the function is implemented natively in C# or is a custom function implemented in CSCS.

When trying to associate a function or a variable name to the actual function or variable, the **`GetFunction`** is called. Note that it first searches the local names—they have a precedence over the global names.

Code Listing 49: The implementation of the global and local variables in the ParserFunction class

```
public class ParserFunction
{
  public class StackLevel {
    public StackLevel(string name = null) {
      Name = name;
      Variables = new Dictionary<string, ParserFunction>();
    }
    public string Name { get; set; }
    public Dictionary<string, ParserFunction> Variables { get; set; }
  }

  // Global functions and variables:
  private static Dictionary<string, ParserFunction> s_functions =
              new Dictionary<string, ParserFunction>();

  // Local variables:
  // Stack of the functions being executed:
  private static Stack<StackLevel> s_locals = new Stack<StackLevel>();
  public  static Stack<StackLevel> ExecutionStack {
                                   get { return s_locals; } }

  public static ParserFunction GetFunction(string item)
  {
    ParserFunction impl;
    // First search among local variables.
    if (s_locals.Count > 0) {
      Dictionary<string, ParserFunction> local = s_locals.Peek().Variables;
      if (local.TryGetValue(item, out impl)) {
        // Local function exists (a local variable).
        return impl;
      }
    }
    if (s_functions.TryGetValue(item, out impl)) {
      // A global function exists and is registered
      // (e.g. pi, exp, or a variable).
      return impl.NewInstance();
    }

    return null;
  }
  public static bool FunctionExists(string item)
  {
    bool exists = false;
    // First check if the local function stack has this variable defined.
    if (s_locals.Count > 0) {
      Dictionary<string, ParserFunction> local = s_locals.Peek().Variables;
      exists = local.ContainsKey(item);
    }

    // If it is not defined locally, then check globally:
    return exists || s_functions.ContainsKey(item);
  }

  public static void AddGlobalOrLocalVariable(string name,
                                              ParserFunction function) {
    function.Name = name;
    if (s_locals.Count > 0) {
      AddLocalVariable(function);
    } else {
      AddGlobal(name, function, false /* not native */);
    }
  }

  public static void RegisterFunction(string name, ParserFunction function,
                                      bool isNative = true) {
    AddGlobal(name, function, isNative);
  }

  static void AddGlobal(string name, ParserFunction function,
                        bool isNative = true) {
    function.isNative = isNative;
    s_functions[name] = function;
    if (string.IsNullOrWhiteSpace(function.Name)) {
      function.Name = name;
    }
    if (!isNative) {
      Translation.AddTempKeyword(name);
    }
  }

  public static void AddLocalVariables(StackLevel locals)
  {
    s_locals.Push(locals);
  }

  public static void AddStackLevel(string name)
  {
    s_locals.Push(new StackLevel(name));
  }
  public static void AddLocalVariable(ParserFunction local)
  {
    local.m_isGlobal = false;
    StackLevel locals = null;
    if (s_locals.Count == 0) {
      locals = new StackLevel();
      s_locals.Push(locals);
    } else {
      locals = s_locals.Peek();
    }
    locals.Variables[local.Name] = local;
    Translation.AddTempKeyword(local.Name);
  }

  public static void PopLocalVariables()
  {
    s_locals.Pop();
  }

  public static int GetCurrentStackLevel()
  {
    return s_locals.Count;
  }

  public static void InvalidateStacksAfterLevel(int level)
  {
    while (s_locals.Count > level) {
      s_locals.Pop();
    }
  }

  public static void PopLocalVariable(string name)
  {
    if (s_locals.Count == 0) {
      return;
    }
    Dictionary<string, ParserFunction> locals = s_locals.Peek().Variables;
    locals.Remove(name);
  }
}
```

Implementing custom methods and functions

To implement custom methods and functions in CSCS, we need two classes, **`FunctionCreator`** and **`CustomFunction`**, both deriving from the **`ParserFunction`** class; see Figure 4. The **`FunctionCreator`** class is shown in Code Listing 50. We register it with the parser as follows:

```
public const string FUNCTION = "function";
ParserFunction.RegisterFunction(Constants.FUNCTION, new FunctionCreator());
```

Note: No worries, we'll see very soon how to redefine a function name in the configuration file.
So a typical function in our language looks like:

```
function functionName(param1, param2, ..., paramN) {
    // Function Body;
}
```

Code Listing 50: The implementation of the FunctionCreator class

```
class FunctionCreator : ParserFunction
{
  protected override Variable Evaluate(ParsingScript script)
  {
    string funcName = Utils.GetToken(script, Constants.TOKEN_SEPARATION);

    string[] args = Utils.GetFunctionSignature(script);
    if (args.Length == 1 && string.IsNullOrWhiteSpace(args[0])) {
      args = new string[0];
    }

    script.MoveForwardIf(Constants.START_GROUP, Constants.SPACE);
    int parentOffset = script.Pointer;

    string body = Utils.GetBodyBetween(script, Constants.START_GROUP,
                                               Constants.END_GROUP);

    CustomFunction customFunc = new CustomFunction(funcName, body, args);
    customFunc.ParentScript = script;
    customFunc.ParentOffset = parentOffset;

    ParserFunction.RegisterFunction(funcName, customFunc,
                                    false /* not native */);

    return new Variable(funcName);
  }
}
```

First, the **`FunctionCreator.Evaluate`** method calls an auxiliary **`Utils.GetToken`** method, which extracts the **`functionName`** in the function definition. Then, the

`Utils.GetFunctionSignature` auxiliary function gets all the function arguments, see Code Listing 51.

Note that we do not have explicit types in our language: **the types are deduced on the fly from the context.** Therefore, the result of the **`Utils.GetFunctionSignature`** function is an array of strings, like **`arg1`**, **`arg2`**, …, **`argN`**. An example of a function signature is: **`function power(a, n)`**.

Code Listing 51: The implementation of the GetFunctionSignature method

```
public static string[] GetFunctionSignature(ParsingScript script)
{
  script.MoveForwardIf(Constants.START_ARG, Constants.SPACE);

  int endArgs = script.FindFirstOf(Constants.END_ARG.ToString());
  if (endArgs < 0) {
    throw new ArgumentException("Couldn't extract function signature");
  }

  string argStr = script.Substr(script.Pointer, endArgs - script.Pointer);
  string[] args = argStr.Split(Constants.NEXT_ARG_ARRAY);

  script.Pointer = endArgs + 1;
  return args;
}
```

The auxiliary **`Utils.GetBodyBetween`** method extracts the actual body of the function in Code Listing 52. As method arguments, we pass the open character as **`Constants.START_GROUP`** (which I defined as **{**), and the close character as **`Constants.END_GROUP`** (which I defined as **}**).

Code Listing 52: The implementation of the GetBodyBetween method

```
public static string GetBodyBetween(ParsingScript script,
                                    char open, char close)
{
  // We are supposed to be one char after the beginning of the string,
  // so we must not have the opening char as the first character.
  StringBuilder sb = new StringBuilder(script.Size());
  int braces = 0;

  for (; script.StillValid(); script.Forward())  {
    char ch = script.Current;

    if (string.IsNullOrWhiteSpace(ch.ToString()) && sb.Length == 0) {
      continue;
    } else if (ch == open) {
      braces++;
    } else if (ch == close) {
      braces--;
    }

    sb.Append(ch);
    if (braces == -1)  {
      if (ch == close) {
        sb.Remove(sb.Length - 1, 1);
      }
      break;
    }
  }
  return sb.ToString();
}
```

Basically, the **FunctionCreator** class creates a new instance of the **CustomFunction** class (see its implementation in Code Listing 53) and registers it with the parser.

Code Listing 53: The implementation of the CustomFunction class

```
class CustomFunction : ParserFunction
{
  internal CustomFunction(string funcName,
                          string body, string[] args) {
    m_name = funcName;
    m_body = body;
    m_args = args;
  }

  protected override Variable Evaluate(ParsingScript script)
  {
    bool isList;
    string parsing = script.Rest;
    List<Variable> functionArgs = Utils.GetArgs(script,
        Constants.START_ARG, Constants.END_ARG, out isList);

    script.MoveBackIf(Constants.START_GROUP);
    if (functionArgs.Count != m_args.Length) {
      throw new ArgumentException("Function [" + m_name +
         "] arguments mismatch: " + m_args.Length + " declared, " +
         functionArgs.Count + " supplied");
    }

    // 1. Add passed arguments as local variables to the Parser.
    StackLevel stackLevel = new StackLevel(m_name);
    for (int i = 0; i < m_args.Length; i++) {
      stackLevel.Variables[m_args[i]] =
          new GetVarFunction(functionArgs[i]);
    }
    ParserFunction.AddLocalVariables(stackLevel);

    // 2. Execute the body of the function.
    Variable result = null;
    ParsingScript tempScript = new ParsingScript(m_body);
    tempScript.ScriptOffset = m_parentOffset;
    if (m_parentScript != null) {
      tempScript.Char2Line      = m_parentScript.Char2Line;
      tempScript.Filename       = m_parentScript.Filename;
      tempScript.OriginalScript = m_parentScript.OriginalScript;
    }

    while (tempScript.Pointer < m_body.Length - 1 &&
          (result == null || !result.IsReturn)) {
      string rest = tempScript.Rest;
      result = tempScript.ExecuteTo();
      tempScript.GoToNextStatement();
    }

    ParserFunction.PopLocalVariables();
    result.IsReturn = false;
    return result;
  }

  public ParsingScript ParentScript { set { m_parentScript = value; } }
  public int           ParentOffset { set { m_parentOffset = value; } }
  public string        Body         { get { return m_body; } }
  public string        Header       { get {
    return Constants.FUNCTION + " " + Name + " " +
           Constants.START_ARG + string.Join (", ", m_args) +
           Constants.END_ARG + " " + Constants.START_GROUP;
    }
  }

  private string        m_body;
  private string[]      m_args;
  private ParsingScript m_parentScript = null;
  private int           m_parentOffset = 0;
}
```

We call the **`Utils.GetArgs`** auxiliary function to extract the arguments (defined in Code Listing 25).

We'll see the usage of the **`m_parentOffset, m_parentScript`** (and its properties: **`Char2Line`**, **`Filename`**, and **`OriginalScript`**) in the "Getting line numbers where errors " section in Chapter 7 Localization.

An example of a CSCS custom function: factorial

Let's now see custom functions and exceptions in action. As an example, we'll create a factorial. The factorial function, denoted as **`n!`**, is defined as follows:

n! = 1 * 2 * 3 * … * (n - 1) * n.

There is a special definition when **`n = 0: 0! = 1`**. The factorial is not defined for negative numbers. Note that we can also define the factorial recursively: **`n! = (n - 1)! * n`**.

Code Listing 54: The CSCS code for the factorial recursive implementation

```
function factorial(n)
{
  if (!isInteger(n) || n < 0) {
    exc = "Factorial is for nonnegative integers only (n=" + n + ")";
    throw (exc);
  }
  if (n <= 1) {
    return 1;
  }

  return n * factorial(n - 1);
}

function isInteger(candidate) {
  return candidate == round(candidate);
}

function factorialHelper(n)
{
  try {
    f = factorial(n);
    print("factorial(", n, ")=", f);
  } catch (exc) {
    print("Caught exception: ", exc);
  }
}

factorialHelper(0);
factorialHelper(10);
factorialHelper("blah");
```

The implementation of the recursive version of the factorial in CSCS is shown in Code Listing 54. The factorial function uses the **`isInteger`** CSCS function to check if the passed parameter is an integer. This function is implemented in Code Listing 54 as well. It calls the **`round`** function, which was already implemented in C# (see Code Listing 34).

These are the results of running the CSCS script of Code Listing 54:

```
factorial(0)=1
factorial(10)=3628800
Caught exception: Factorial is for nonnegative integers only (n=blah) --> exc
  factorial()
  factorialHelper()
```

In the **`Caught exception`** clause you can see the exception stack, which was produced by the **`CreateExceptionStack`** method (see Code Listing 47). One can also add the line numbers where the exception occurred. We'll see more about that in Chapter 7 *Localization*.

Conclusion

In this chapter we continued adding functionality to the CSCS language: we saw how to include files, to throw and to catch an exception, how to add local and global variables, and how to implement custom functions in CSCS. We also saw some examples of writing custom functions.

In the next chapter we are going to see how to develop some data structures to be used in CSCS.

Chapter 6 Operators, Arrays, and Dictionaries

"Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration."
Stan Kelly-Bootle

In this chapter we are going to see how to implement custom data structures in CSCS. These data structures are added to the Variable class. In particular, we are going to see how to add multidimensional arrays and Dictionaries.

We'll start by looking at *actions*. We call an action a special class, also deriving from the **`ParserFunction`** class, that adds the functionality of using the assignment, increment, and decrement operators.

Actions in CSCS

All of the functionality of CSCS that we've seen so far revolves around functions—all of the features of the language that we develop have been by using classes deriving from the **`ParserFunction`** class.

But what about simple statements, like **`a = 1`**, **`a++`**, or **`a +=5`**? It's not immediately clear how we can use the **`ParserFunction`** class here, so we need to do a bit more of work here.

First, we define an abstract **`ActionFunction`** class, deriving from the **`ParserFunction`** class, in Code Listing 55. Then we create three concrete implementations of this class. Check out the UML diagram in Figure 4.

Code Listing 55: The abstract ActionFunction class

```
public abstract class ActionFunction : ParserFunction
{
  protected string m_action;
  public string Action { set { m_action = value; } }
}
```

We register the new functions with the parser as follows:

```
  public const string ASSIGNMENT = "=";
  public const string INCREMENT  = "++";
  public const string DECREMENT  = "--";
  ParserFunction.AddAction(Constants.ASSIGNMENT, new AssignFunction());
  ParserFunction.AddAction(Constants.INCREMENT,
                           new IncrementDecrementFunction());
  ParserFunction.AddAction(Constants.DECREMENT,
                                    new IncrementDecrementFunction());
```

Each time we extract a **=**, **++**, or **--** token, its corresponding function will be called. For that to work, we need to introduce actions in the **ParserFunction** class. Code Listing 56 extends the "virtual" constructor of the **ParserFunction** class that we previously saw in Code Listing 7.

There are two additions to the constructor: when working with extracted token, we first check whether it's a registered action, and then, whether it's an array. We'll be dealing with arrays in the next section.

Code Listing 56: The ParserFunction "virtual" constructor revisited and Actions

```
public class ParserFunction
{
  // Global actions - function:
  private static Dictionary<string, ActionFunction> s_actions =
             new Dictionary<string, ActionFunction>();

  // A "virtual" constructor
  internal ParserFunction(ParsingScript script, string token,
                          char ch, ref string action) {
    if (token.Length == 0 &&
       (ch == Constants.START_ARG || !script.StillValid ())) {
      // There is no function, just an expression in parentheses.
      m_impl = s_idFunction;
      return;
    }

    // New code:
    m_impl = GetRegisteredAction(item, ref action);
    if (m_impl != null) {
      return;
    }
    m_impl = GetArrayFunction(item, script, action);
    if (m_impl != null) {
      return;
    }
    // As before...
  }

  public static void AddAction(string name, ActionFunction action)
  {
    s_actions[name] = action;
  }

  public static ActionFunction GetAction(string action)
  {
    if (string.IsNullOrWhiteSpace (action)) {
      return null;
    }
    ActionFunction impl;
    if (s_actions.TryGetValue(action, out impl)) {
      // Action exists and is registered (e.g. =, +=, --, etc.)
      return impl;
    }
    return null;
  }

  public static ParserFunction GetRegisteredAction(string name,
                                                   ref string action) {
    ActionFunction actionFunction = GetAction(action);
    if (actionFunction == null) {
      return null;
    }

    ActionFunction theAction = actionFunction.NewInstance() as
                                              ActionFunction;

    theAction.Name = name;
    theAction.Action = action;

    action = null;
    return theAction;
  }
}
```

Implementing new instances of the ParserFunction objects

The **`GetRegisteredAction`** method in Code Listing 56 is just a wrapper over the **`GetAction`** method, which checks if the passed token corresponds to an action that was registered before with the parser. In addition, the **`GetRegisteredAction`** method creates a new instance of the object that implements the action.

Why would we need a new instance of an object? Because actions can be called recursively one from another, and if we used the same object, then there could be discrepancies with its class member variables. If we used the same instance of the object for the assignment operator, we could have the following problem. For the expression **`a = (b = 1)`**, the member variable **`m_name`** will be first assigned the value of **`a`**, but then **`m_name`** will be changed to **`b`** (when parsing the expression **`b=1`**. See Code Listing 57).

I actually spent some time on this issue (using the same object in the assignment operator), so I hope you won't step on the same rake.

The same problem can occur in general with any class deriving from the **`ParserFunction`** class; that's why the **`ParserFunction`** class has a method called **`NewInstance`**, which by default returns the same object. So by default, all **`ParserFunction`** use the same instance of an object. Generally, it's not a problem—even if we have an expression like **`sin(sin(sin(x)))`**, the same **`SinFunction`** object instance will be called three times. But **`SinFunction`** class has no internal members that can be modified by calling the **`Evaluate`** method, so we are safe when using the default version of the **`NewInstance`** method. If you are going to use and modify member variables inside of the **`Evaluate`** method, then you should also return a new instance of the object in the **`NewInstance`** member, analogous to Code Listing 56.

Implementing the assignment operator

Code Listing 57 shows an implementation of the assignment operator. There are two cases. If an assignment looks like **`a = b`** (if the variable **`b`** does not have any array indices), we just register (or, possibly, re-register) the variable named **`a`** with the parser, matching it with the value of **`b`**. The value of **`b`** can be any complex expression. We get it in this statement:

```
  Variable varValue = Utils.GetItem(script);
```

The **`Utils.GetItem`** can recursively call the whole **Split-and-Merge** algorithm.
Another case is when the assignment looks like **`a = b[i]`**, where we are dealing with an array element on the right-hand side. We are going to see this case in the next section.

Code Listing 57: The implementation of the AssignFunction class

```
class AssignFunction : ActionFunction
{
  protected override Variable Evaluate(ParsingScript script)
  {
    Variable varValue = Utils.GetItem(script);

    // Check if the variable to be set has the form of x[a][b]...,
    // meaning that this is an array element.
    List<Variable> arrayIndices = Utils.GetArrayIndices(ref m_name);
    if (arrayIndices.Count == 0) { // Not an array.
      ParserFunction.AddGlobalOrLocalVariable(m_name,
                                   new GetVarFunction(varValue));
      return varValue;
    }

    Variable array;
    ParserFunction pf = ParserFunction.GetFunction(m_name);
    if (pf != null) {
      array = pf.GetValue(script);
    } else {
      array = new Variable();
    }

    ExtendArray(array, arrayIndices, 0, varValue);
    ParserFunction.AddGlobalOrLocalVariable(m_name,
                                                 new GetVarFunction(array));
    return array;
  }
  override public ParserFunction NewInstance()
  {
    return new AssignFunction();
  }
}
```

Code Listing 57 doesn’t have a definition of the **`ExtendArray`** method. We are going to look at it when we study arrays in the next section.

Using arrays in CSCS

“Programming without arrays is like swimming without trunks. It works, but for most people, it's ugly.”
Unknown

Arrays in CSCS don’t have to be declared—everything will be deduced from the context. Also, when assigning a value to an array element, the previous elements don’t have to exist. Why should they? Code Listing 3 already contains a class member variable, which we called **`m_tuple`**:

```
  private List<Variable> m_tuple;
```

This is how we internally represent an array. But we haven’t seen yet how to use arrays in our language. Here we are going to explore how to define and use arrays with an unlimited number of dimensions.

First of all, here are two definitions from the **`Constants`** class, which tell the parser how to define an index of an array:

```
  public const char START_ARRAY   = '[';
  public const char END_ARRAY     = ']';
```

With these definitions, we define how to access an array element: **`a[0][1]`** accesses the first element of the variable **`a`**, which must be of type **`ARRAY`**, and then the second element of the variable **`a[0]`**, which also must be of type **`ARRAY`**. (We decided to start array indices at 0, not 1, nor Stan Kelly-Bottle’s compromise of 0.5. Besides, CSCS is based on C#, so starting at zero is best).

There are a few places where we need to do some changes. One of them is the **`GetVarFunction`** class, which we saw in Code Listing 48. Code Listing 58 shows a more complete version of the **`GetVarFunction`** class, now also for the array elements.

Code Listing 58: The implementation of the GetVarFunction class

```
class GetVarFunction : ParserFunction
{
  internal GetVarFunction(Variable value)
  {
    m_value = value;
  }

  protected override Variable Evaluate(ParsingScript script)
  {
    // First check if this element is a part of an array:
    if (script.TryPrev() == Constants.START_ARRAY) {
      // There is an index given - it must be for an element of the tuple.
      if (m_value.Tuple == null || m_value.Tuple.Count == 0) {
        throw new ArgumentException("No tuple exists for the index");
      }

      if (m_arrayIndices == null) {
        string startName = script.Substr(script.Pointer - 1);
        m_arrayIndices = Utils.GetArrayIndices(ref startName, ref m_delta);
      }
      script.Forward(m_delta);

      Variable result = Utils.ExtractArrayElement(m_value, m_arrayIndices);
      return result;
    }

    // Otherwise just return the stored value.
    return m_value;
  }

  public int Delta {
    set { m_delta = value; }
  }

  public List<Variable> Indices {
    set { m_arrayIndices = value; }
  }

  private Variable m_value;
  private int m_delta = 0;
  private List<Variable> m_arrayIndices = null;
}
```

Code Listing 58**Error! Reference source not found.** uses an auxiliary **`Utils.GetArrayIndices`** method, which is shown in Code Listing 59. It works as follows: if the argument **`varName`** is, for instance, **`a[0][1]`**, then it will return a list of two variables (one for each index); it will also set argument **`varName`** to string **`a`** (the name of the array), and the argument **`end`** to **`6`** (the index of the last closing square bracket).

An empty list will be returned if the **`varName`** argument isn't an array element. Note that the **Split-and-Merge** algorithm can be called recursively when an index of an array is an expression itself.

Code Listing 59: The implementation of the Utils.GetArrayIndices method

```
public static List<Variable> GetArrayIndices(ref string varName,
                                             ref int end)
{
  List<Variable> indices = new List<Variable>();

  int argStart = varName.IndexOf(Constants.START_ARRAY);
  if (argStart < 0) {
    return indices;
  }
  int firstIndexStart = argStart;

  while (argStart < varName.Length &&
         varName[argStart] == Constants.START_ARRAY) {
    int argEnd = varName.IndexOf(Constants.END_ARRAY, argStart + 1);
    if (argEnd == -1 || argEnd <= argStart + 1) {
      break;
    }

    ParsingScript tempScript = new ParsingScript(varName, argStart);
    tempScript.MoveForwardIf(Constants.START_ARG, Constants.START_ARRAY);

    Variable index = tempScript.ExecuteTo(Constants.END_ARRAY);
    indices.Add(index);
    argStart = argEnd + 1;
  }

  if (indices.Count > 0) {
    varName = varName.Substring(0, firstIndexStart);
    end = argStart - 1;
  }

  return indices;
}
```

The argument **`end`** in the **`GetArraIndices`** method will be assigned to the **`m_delta`** member variable of the **`GetVarFunction`** class. We need **`delta`** in order to know the length of the variable, so we can skip that many characters, once the assignment is done.

The **`ParserFunction`** virtual constructor in Code Listing 56**Error! Reference source not found.** also calls the **`GetArrayFunction`** method. The implementation of this method is shown in Code Listing 60. This method checks if the passed argument **`name`** is an array element, and, if it is, returns the corresponding **`GetVarFunction`** object. Remember that the **`GetVarFunction`** class is a wrapper over a CSCS variable, and it's used whenever we register a local or a global variable with the parser. We are going to see more examples of using this class later in this chapter.

Code Listing 60: ParserFunction.GetArrayFunction method

```
public static ParserFunction GetArrayFunction(string name,
                    ParsingScript script, string action)
{
  int arrayStart = name.IndexOf(Constants.START_ARRAY);
  if (arrayStart < 0) {
     return null;
  }
  string arrayName = name;
  int delta = 0;

  List<Variable> arrayIndices = Utils.GetArrayIndices(ref arrayName,
                                                      ref delta);
  if (arrayIndices.Count == 0) {
    return null;
  }

  ParserFunction pf = ParserFunction.GetFunction(arrayName);
  GetVarFunction varFunc = pf as GetVarFunction;
  if (varFunc == null) {
    return null;
  }

  script.Backward(name.Length - arrayStart - 1);
  script.Backward(action != null ? action.Length : 0);
  delta -= arrayName.Length;

  varFunc.Indices = arrayIndices;
  varFunc.Delta = delta;
  return varFunc;
}
```

When assigning values to an element of an array in the **`AssignFunction.Evaluate`** method (see Code Listing 57), the **`ExtendArray`** method is invoked. It's a crucial method for dealing with multidimensional arrays; check it out in Code Listing 61. This method may call itself recursively and permits creating a new array with any number of dimensions on the fly!

Code Listing 61: The implementation of the AssignFunction.ExtendArray method

```
public static void ExtendArray(Variable parent,
                               List<Variable> arrayIndices,
                               int indexPtr,
                               Variable varValue)
{
  if (arrayIndices.Count <= indexPtr) {
    return;
  }
  Variable index = arrayIndices[indexPtr];
  int currIndex = ExtendArrayHelper(parent, index);

  if (arrayIndices.Count - 1 == indexPtr) {
    parent.Tuple[currIndex] = varValue;
    return;
  }

  Variable son = parent.Tuple[currIndex];
  ExtendArray(son, arrayIndices, indexPtr + 1, varValue);
}

static int ExtendArrayHelper(Variable parent, Variable indexVar)
{
  parent.SetAsArray();
  int arrayIndex = parent.GetArrayIndex(indexVar);
  if (arrayIndex < 0) {
  // This is not a "normal index" but a new string for the dictionary.
    string hash = indexVar.AsString();
    arrayIndex  = parent.SetHashVariable(hash, Variable.NewEmpty());
    return arrayIndex;
  }

  if (parent.Tuple.Count <= arrayIndex) {
    for (int i = parent.Tuple.Count; i <= arrayIndex; i++) {
      parent.Tuple.Add(Variable.NewEmpty());
    }
  }
  return arrayIndex;
}
```

For example, if you have an expression in CSCS:

    x[3][7][4] = 10;

Note: *A three-dimensional array will be created on the fly (even if you never mentioned the variable* **x** *before in the CSCS code)*.

The **`ExtendArray`** method would be invoked recursively three times for this example, each time for each of the dimensions. The **`ExtendArrayHelper`** method will initialize all the unused array elements to an **`EmptyVariable`** (with variable type **`NONE`**).

For example, the array element **`x[1][1][1]`** will be used as an empty string, if used in a string manipulation, or as a number 0, if used in a numerical calculation. The **`ExtendArrayHelper`** method also invokes the **`Variable.GetArrayIndex`** and **`SetHashVariable`** methods. We'll show them later on, when learning how to implement dictionaries in CSCS (see Code Listing 66).

Another thing you could do is to use the array indexes as the hash keys to a dictionary—even though the access to an element of a dictionary is a bit slower, you could save quite a bit on memory usage, especially if you do assignments like **`a[100]=x`**, and never access elements of the array having indices less than 100. We'll discuss dictionaries at the end of this chapter.

So one can start using an array even without declaring it. Note that the array elements don't have to be of the same type. It's perfectly legal in CSCS to have statements like:

```
x[3][7][5] = "blah";
```

If you think this treatment of arrays is way too bold, you can add your restrictions in the following **`AssignFunction`** methods: **`Evaluate`**, **`ExtendArray`**, and **`ExtendArrayHelper`**, shown in Code Listing 57 and Code Listing 61.

Implementing the prefix and postfix operators in CSCS

The **`IncrementDecrementFunction.Evaluate`** method shown in Code Listing 62 implements both the prefix and the postfix operators (each one using either the **`++`** or the **`--`** form).

In case of a prefix form (when the operator looks like **`++a`**), we have just collected the **`++`** token, so we need to collect the variable name next. On the other hand, in the postfix form (when the operator looks like **`a++`**), we already know the variable name that has been assigned to the **`m_name`** member variable. This happened in the **`ParserFunction.GetRegisteredAction`** function (which was invoked from the **`ParserFunction`** virtual constructor in Code Listing 56).

Code Listing 62: The implementation of the IncrementDecrementFunction.Evaluate method

```
protected override Variable Evaluate(ParsingScript script)
{
  bool prefix = string.IsNullOrWhiteSpace(m_name);

  if (prefix) {
    // If it is a prefix we do not have the variable name yet.
    m_name = Utils.GetToken(script, Constants.TOKEN_SEPARATION);
  }

  // Value to be added to the variable:
  int valueDelta = m_action == Constants.INCREMENT ? 1 : -1;
  int returnDelta = prefix ? valueDelta : 0;

  // Check if the variable to be set has the form of x[a][b],
  // meaning that this is an array element.
  double newValue = 0;
  List<Variable> arrayIndices = Utils.GetArrayIndices(ref m_name);

  ParserFunction func = ParserFunction.GetFunction(m_name);
  Utils.CheckNotNull(m_name, func);
  Variable currentValue = func.GetValue(script);

  if (arrayIndices.Count > 0 ||
      script.TryCurrent () == Constants.START_ARRAY) {
    if (prefix) {
      string tmpName = m_name + script.Rest;
      int delta = 0;
      arrayIndices = Utils.GetArrayIndices(ref tmpName, ref delta);
        script.Forward(Math.Max(0, delta - tmpName.Length));
    }

    Variable element = Utils.ExtractArrayElement(currentValue,
                                                 arrayIndices);
    script.MoveForwardIf(Constants.END_ARRAY);
    newValue = element.Value + returnDelta;
    element.Value += valueDelta;
  }
  else { // A normal variable.
    newValue = currentValue.Value + returnDelta;
    currentValue.Value += valueDelta;
  }

  ParserFunction.AddGlobalOrLocalVariable(m_name,
                   new GetVarFunction(currentValue));
  return new Variable(newValue);
}
```

Implementing compound assignment operators in CSCS

By compound assigning operators, we mean all possible variants of `a += b`, `a -= b`, `a *= b`, and so on. Code Listing 63 implements the **`OperatorAssignFunction`** class. The **`OperatorAssignFunction.Evaluate`** method will be invoked as soon as we get any of **`+=`**, **`-=`**, **`*=`**, etc. tokens. Therefore, we must register each of these tokens with the parser:

```
for (int i = 0; i < Constants.OPER_ACTIONS.Length; i++) {
  ParserFunction.AddAction(Constants.OPER_ACTIONS[i],
                           new OperatorAssignFunction());
}
```

Where the operator actions string array is defined in the **`Constants`** class as:

```
public static string[] OPER_ACTIONS = { "+=", "-=", "*=", "/=",
                                        "%=", "&=", "|=", "^="};
```

The implementation of all these actions is in the **`OperatorAssignFunction.NumberOperator`** method in Code Listing 63. The **`StringOperator`** contains just the implementation of the **`+=`**

operator since the rest of the operators did not make much sense to me, but you can add as many fancy operators as you like.

Code Listing 63: The implementation of the OperatorAssignFunction class

```
class OperatorAssignFunction : ActionFunction
{
  protected override Variable Evaluate(ParsingScript script)
  {
    // Value to be added to the variable:
    Variable right = Utils.GetItem(script);

    List<Variable> arrayIndices = Utils.GetArrayIndices(ref m_name);

    ParserFunction func = ParserFunction.GetFunction(m_name);
    Utils.CheckNotNull(m_name, func);

    Variable currentValue = func.GetValue(script);
    Variable left = currentValue;

    if (arrayIndices.Count > 0) { // array element
      left = Utils.ExtractArrayElement(currentValue, arrayIndices);
      script.MoveForwardIf(Constants.END_ARRAY);
    }

    if (left.Type == Variable.VarType.NUMBER) {
      NumberOperator(left, right, m_action);
    } else {
      StringOperator(left, right, m_action);
    }

    if (arrayIndices.Count > 0) { // array element
      AssignFunction.ExtendArray(currentValue, arrayIndices, 0, left);
      ParserFunction.AddGlobalOrLocalVariable(m_name,
                                new GetVarFunction(currentValue));
    } else { // normal variable
      ParserFunction.AddGlobalOrLocalVariable(m_name,
                                new GetVarFunction(left));
    }
    return left;
  }

  static void NumberOperator(Variable valueA,
                             Variable valueB, string action) {
    switch (action) {
      case "+=":
        valueA.Value += valueB.Value;
        break;
      case "-=":
        valueA.Value -= valueB.Value;
        break;
      case "*=":
        valueA.Value *= valueB.Value;
        break;
      case "/=":
        valueA.Value /= valueB.Value;
        break;
      case "%=":
        valueA.Value %= valueB.Value;
        break;
      case "&=":
        valueA.Value = (int)valueA.Value & (int)valueB.Value;
        break;
      case "|=":
        valueA.Value = (int)valueA.Value | (int)valueB.Value;
        break;
      case "^=":
        valueA.Value = (int)valueA.Value ^ (int)valueB.Value;
        break;
    }
  }

  static void StringOperator(Variable valueA,
                             Variable valueB, string action) {
    switch (action) {
      case "+=":
        if (valueB.Type == Variable.VarType.STRING) {
          valueA.String += valueB.AsString();
        } else {
          valueA.String += valueB.Value;
        }
        break;
    }
  }

  override public ParserFunction NewInstance()
  {
    return new OperatorAssignFunction();
  }
}
```

Implementing the for-loops

Let's see how to implement the **`for`** control flow statement. We are going to look at it here, since besides the canonical loop **`for (initStep; loopCondition; loopUpdate)`**, we're also going to implement a simpler **`for`** loop for arrays, already present in many languages: **`for (item : array)`**. Code Listing 64 implements both **`for`** loops.

Code Listing 64: The implementation of the Internal.ProcessFor method

```
internal Variable ProcessFor(ParsingScript script)
{
  string forString = Utils.GetBodyBetween(script, Constants.START_ARG,
                                                  Constants.END_ARG);
  script.Forward();
  if (forString.Contains(Constants.END_STATEMENT.ToString())) {
    // Looks like: "for(i = 0; i < 10; i++)".
    ProcessCanonicalFor(script, forString);
  } else {
    // Otherwise looks like: "for(item : array)"
    ProcessArrayFor(script, forString);
  }
  return Variable.EmptyInstance;
}

void ProcessArrayFor(ParsingScript script, string forString)
{
  int index = forString.IndexOf(Constants.FOR_EACH);
  if (index <= 0 || index == forString.Length - 1) {
    throw new ArgumentException("Expecting: for(item : array)");
  }

  string varName = forString.Substring (0, index);
  ParsingScript forScript = new ParsingScript(forString);
  Variable arrayValue = forScript.Execute(index + 1);

  int cycles = varValue.TotalElements();
  int startForCondition = script.Pointer;

  for (int i = 0; i < cycles; i++) {
    script.Pointer = startForCondition;
    Variable current = arrayValue.GetValue(i);
    ParserFunction.AddGlobalOrLocalVariable(forString,
                       new GetVarFunction(current));
    Variable result = ProcessBlock(script);
    if (result.IsReturn || result.Type == Variable.VarType.BREAK) {
      script.Pointer = startForCondition;
      SkipBlock(script);
      return;
    }
  }
}

void ProcessCanonicalFor(ParsingScript script, string forString)
{
  string[] forTokens = forString.Split(Constants.END_STATEMENT);
  if (forTokens.Length != 3) {
    throw new ArgumentException(
              "Expecting: for(init; condition; loopStatement)");
  }

  int startForCondition = script.Pointer;
  ParsingScript initScript = new ParsingScript(forTokens[0] +
                                          Constants.END_STATEMENT);
  ParsingScript condScript = new ParsingScript(forTokens[1] +
                                          Constants.END_STATEMENT);
  ParsingScript loopScript = new ParsingScript(forTokens[2] +
                                          Constants.END_STATEMENT);
  initScript.Execute();
  int cycles = 0;
  bool stillValid = true;

  while(stillValid) {
    Variable condResult = condScript.Execute();
    stillValid = Convert.ToBoolean(condResult.Value);
    if (!stillValid) {
      return;
    }

    if (MAX_LOOPS > 0 && ++cycles >= MAX_LOOPS) {
      throw new ArgumentException ("Looks like an infinite loop after " +
                                   cycles + " cycles.");
    }

    script.Pointer = startForCondition;
    Variable result = ProcessBlock(script);

    if (result.IsReturn || result.Type == Variable.VarType.BREAK) {
      script.Pointer = startForCondition;
      SkipBlock(script);
      return;
    }

    loopScript.Execute();
  }
}
```

The declaration of the **for** loop and registering it with the parser is very similar to what we did in "Implementing the **while** " section in Chapter 3 *Basic Control Flow Statements.*

Code Listing 65 shows examples of using the **for** loops in CSCS. It also has a degraded version of a for loop, where all of the arguments are empty.

Code Listing 65: Different for loops, implemented in CSCS

```
for (i = 10; i >= 0; i--) {
  write(i, " ");
  arr[i] = 2 * i;
}
print;

i = 10;
for (;;) {
  write(i, " ");
  arr[i] = 2 * i;
  i--;
  if (i < 0) { break; }
}
print;

for (item : arr) {
  write (item, " ");
}

// Output:
// 10 9 8 7 6 5 4 3 2 1 0
// 10 9 8 7 6 5 4 3 2 1 0
// 0 2 4 6 8 10 12 14 16 18 20
```

Adding dictionaries to the CSCS

Recall that a dictionary is a data structure that maps a key to a value. A hash table is a particular case of a dictionary, where each key is mapped to a unique value using hash codes. We'll use the C# dictionary data structure that is implemented as a hash table. We'll use strings as keys, and delegate the work of converting a string to a hash code to C#. So basically, our dictionary will be a thin wrapper over the C# dictionary. We are going to reuse the same data structure we used for arrays to store the dictionary values. The dictionary will just map the key to the array index where the value is stored. Code Listing 66 shows how to do this.

Code Listing 66: The implementation of the dictionary in the Variable class

```
private Dictionary<string, int> m_dictionary =
                                new Dictionary<string, int>();
public int SetHashVariable(string hash, Variable var)
{
  int retValue = m_tuple.Count;
  if (m_dictionary.TryGetValue(hash, out retValue)) {
    // Already exists, change the value:
    m_tuple[retValue] = var;
    return retValue;
  }
  m_tuple.Add(var);
  m_dictionary[hash] = m_tuple.Count - 1;

  return m_tuple.Count - 1;
}

public int GetArrayIndex(Variable indexVar)
{
  if (this.Type != VarType.ARRAY) {
    return -1;
  }

  if (indexVar.Type == VarType.NUMBER) {
    Utils.CheckNonNegativeInt(indexVar);
    return (int)indexVar.Value;
  }

  string hash = indexVar.AsString();
  int ptr = m_tuple.Count;

  if (m_dictionary.TryGetValue(hash, out ptr) &&
      ptr < m_tuple.Count) {
    return ptr;
  }

  return -1;
}
```

It defines a dictionary **`m_dictionary`** with the key being a string and the value being an integer, which is a pointer to the index of the **`List<Variable> m_tuple`** data structure for storing the array elements that we defined earlier in Code Listing 3. So everything is stored in the **`m_tuple`** list. **`m_dictionary`** is just an auxiliary data structure to keep track of where a particular variable is stored in **`m_tuple`**.

If **`indexVar`** is a number passed to **`GetArrayIndex`**, it's treated as an index in **`m_tuple`**, and the corresponding value is returned. When **`indexVar`** is a string, we assume it's a key to **`m_dictionary`** and get its corresponding value first. Then we use that value as an index to access **`m_tuple`**. The method **`SetHashValue`** does the opposite of what **`GetArrayIndex`** does. Both methods were already used within the **`ExtendArrayHelper`** method in Code Listing 61.

Using this implementation, we can combine string keys and integer indices in the same data structure in CSCS, for example:

```
a["bla"][1] = 100;
```

Note that we can use string keys and integer indices interchangeably, even on the same dimension in an array. So, for example, it's also legal to declare:

```
    b["bla"] = 1; b[5] = 2;
```

Isn't that cool?

Example: Dynamic programming in CSCS

Dynamic programming is a special software engineering technique that consists in splitting a large problem into smaller pieces, solving each small piece and storing its result somewhere to be reused later on. As an example of applying this technique in CSCS, let's see the implementation of the Fibonacci numbers. Recall that each Fibonacci number is equal to the sum of the two previous Fibonacci numbers (the first two Fibonacci numbers are defined as 1). The recursive calculation of the Fibonacci numbers in CSCS is shown in Code Listing 67.

Code Listing 67: The implementation of the Fibonacci numbers in CSCS

```
cache["fib"] = 0;

function fibonacci(n)
{
  if (contains(cache["fib"], n)) {
    result = cache["fib"][n];
    return result;
  }
  if (!isInteger(n) || n < 0) {
    exc = "Fibonacci is for nonnegative integers only (n=" + n + ")";
    throw (exc);
  }
  if (n <= 1) {
    return n;
  }

  result = fibonacci(n - 2) + fibonacci(n - 1);
  cache["fib"][n] = result;
  return result;
}

a=fibonacci(10);
print(a);
```

We use the **`cache`** variable as a dictionary on the first dimension, and as an array on the second dimension. It stores intermediate results in memory for later use. Note that without using the **`cache`** data structure, the calculation would've taken much longer, because for each number **`n`** we split calculation in two: **`fibonacci (n – 2)`** and **`fibonacci (n - 1)`**, each of them using same previous Fibonacci numbers. But if we store all of the intermediate results, those values should be already in cache, so we do not have to recursively calculate each of them.

Conclusion

In this chapter we saw how to implement some of the data structures in CSCS. Partularly, we saw the implementation of arrays with an unlimited number of dimensions, and how to combine an array and a dictionary in the same data structure.

In the next chapter we are going to see how to write a CSCS script in any human language.

Chapter 7 Localization

"Programs must be written for people to read, and only incidentally for machines to execute."
Abelson & Sussman

In this chapter we are going to see how to write CSCS programs so that they can be localized in any human language. You'll see how we can supply keyword translations in a configuration file so that the keywords from different languages can be used interchangeably. Not only that, we'll also see how to translate a CSCS program written with the keywords in one human language to another.

Adding translations for the keywords

To add translations for keywords, we use the configuration file. You can start with the one that is automatically created by Visual Studio (or Xamarin Studio); it is usually called **App.config**. Code Listing 68 shows an excerpt from the CSCS configuration file.

Code Listing 68: An excerpt from the configuration file

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="Languages"
             type="System.Configuration.NameValueSectionHandler" />
    <section name="Synonyms"
             type="System.Configuration.NameValueSectionHandler" />
    <section name="Spanish"
             type="System.Configuration.NameValueSectionHandler" />
    <section name="Russian"
             type="System.Configuration.NameValueSectionHandler" />
  </configSections>
  <appSettings>
    <add key="maxLoops" value="100000" />
    <add key="dictionaryPath" value="scripts/" />
    <add key="errorsPath" value="scripts/errors.txt" />
    <add key="language" value="ru" />
  </appSettings>
  <Languages>
    <add key="languages" value="Synonyms,Spanish,Russian" />
  </Languages>
  <Synonyms>
    <add key="copy"     value ="cp" />
    <add key="del"      value ="rm" />
    <add key="dir"      value ="ls" />
    <add key="include"  value ="import" />
    <add key="move"     value ="mv" />
    <add key="print"    value ="writenl" />
    <add key="read"     value ="scan" />
  </Synonyms>
  <Spanish>
    <add key="if"       value ="si" />
    <add key="else"     value ="sino" />
    <add key="for"      value ="para" />
    <add key="while"    value ="mientras" />
    <add key="function" value ="función" />
    <add key="size"     value ="tamaño" />
    <add key="print"    value ="imprimir" />
  </Spanish>
  <Russian>
    <add key="if"       value ="если" />
    <add key="else"     value ="иначе" />
    <add key="for"      value ="для" />
    <add key="while"    value ="пока" />
    <add key="return"   value ="вернуть" />
    <add key="function" value ="функция" />
    <add key="include"  value ="включить" />
    <add key="return"   value ="вернуться" />
  </Russian>
</configuration>
```

The configuration file has translations of the keywords to Spanish and Russian, and also to "Synonyms." It lets us use synonyms for the keywords in English. Anyone who worked with both Windows and the macOS (or any other Unix-based system) would often confuse **`dir`** with **`ls`**, **`copy`** with **`cp`**, and **`grep`** with **`findstr`**. The Synonyms section in the configuration file allows them to be used interchangeably.

To make it work, we introduce a new module, Translation, to support all of the localization concepts. For each keyword that we want to add in the configuration file (but it doesn't have to be added), we call the **`Translation.Add`** static method. See the implementation of the **`Interpreter.ReadConfig`** and a short version of the **`Translation.Add`** methods in Code Listing 69.

Code Listing 69: The implementation of the Interpreter.ReadConfig and Translation.Add methods

```
void ReadConfig()
{
  if (ConfigurationManager.GetSection("Languages") == null) {
    return;
  }
  var languagesSection = ConfigurationManager.GetSection("Languages") as
                         NameValueCollection;
  if (languagesSection.Count == 0) {
    return;
  }

  string errorsPath = ConfigurationManager.AppSettings["errorsPath"];
  Translation.Language = ConfigurationManager.AppSettings["language"];
  Translation.LoadErrors(errorsPath);

  string dictPath = ConfigurationManager.AppSettings["dictionaryPath"];
  string baseLanguage = Constants.ENGLISH;
  string languages = languagesSection["languages"];
  string[] supportedLanguages = languages.Split(",".ToCharArray());

  foreach(string lang in supportedLanguages) {
    string language = Constants.Language(lang);
    Dictionary<string, string> tr1 =
        Translation.KeywordsDictionary(baseLanguage, language);
    Dictionary<string, string> tr2 =
        Translation.KeywordsDictionary(language, baseLanguage);

    Translation.TryLoadDictionary(dictPath, baseLanguage, language);
    var languageSection = ConfigurationManager.GetSection(lang) as
                          NameValueCollection;

    Translation.Add(languageSection, Constants.IF, tr1, tr2);
    Translation.Add(languageSection, Constants.FOR, tr1, tr2);
    Translation.Add(languageSection, Constants.WHILE, tr1, tr2);
    Translation.Add(languageSection, Constants.BREAK, tr1, tr2);
    Translation.Add(languageSection, Constants.CONTINUE, tr1, tr2);
    // More keywords go here...

    // Special dealing for else, catch, etc. since they are not separate
    // functions but are part of the if and try statement blocks.
    Translation.AddSubstatement(languageSection, Constants.ELSE,
                                Constants.ELSE_LIST, tr1, tr2);
    Translation.AddSubstatement(languageSection, Constants.ELSE_IF,
                                Constants.ELSE_IF_LIST, tr1, tr2);
    Translation.AddSubstatement(languageSection, Constants.CATCH,
                                Constants.CATCH_LIST, tr1, tr2);
  }
}

// Incomplete definition of the Translation.Add method:
public static void Add(NameValueCollection langDictionary, string origName,
       Dictionary<string, string> tr1, Dictionary<string, string> tr2)
{
  ParserFunction origFunction = ParserFunction.GetFunction(origName);
  ParserFunction.RegisterFunction(translation, origFunction);
}
```

The **`Translation.Add`** method registers a function, previously (already) registered with the parser (using the **`origName`** string variable) but under a new name (**`translation`**). Therefore, if we get either the **`origName`** token, or a **`translation`** token, the same function will be invoked.

Code Listing 70 contains an example of using the keywords in Spanish in CSCS. The CSCS code there contains a size function ("tamaño" in Spanish). We haven't shown the implementation of the size function, but it just returns the number of elements in an array.

Code Listing 70: An example of using Spanish keywords in CSCS

```
números = {"uno", "dos", "tres", "quatro", "cinco", "seis"};
para (i = 1; i <= tamaño(números); i++) {
  si (i % 2 == 0) {
    imprimir(números[i - 1], " es par");
  } sino {
    imprimir(números[i - 1], " es impar");
  }
}

// Output:
uno es impar
dos es par
tres es impar
quatro es par
cinco es impar
seis es par
```

Adding translations for function bodies

Now let's see how to add translations to the parser to translate error messages and words other than CSCS keywords. A full version of the **`Translation.Add`** method shows it in Code Listing 71.

Code Listing 71: Adding translations to the parser

```
public class Translation
{
  private static HashSet<string> s_nativeWords = new HashSet<string>();
  private static HashSet<string> s_tempWords   = new HashSet<string>();

  private static Dictionary<string, string>
    s_spellErrors   = new Dictionary<string, string>();
  private static Dictionary<string, Dictionary<string, string>>
    s_keywords      = new Dictionary<string, Dictionary<string, string>>();
  private static Dictionary<string, Dictionary<string, string>>
    s_dictionaries = new Dictionary<string, Dictionary<string, string>>();
  private static Dictionary<string, Dictionary<string, string>>
```

```
    s_errors =       new Dictionary<string, Dictionary<string, string>>();

  // The default user language. Can be changed in settings.
  private static string s_language = Constants.ENGLISH;
  public static string Language { set { s_language = value; } }

  public static void Add(NameValueCollection langDictionary,
                         string origName,
                         Dictionary<string, string> translations1,
                         Dictionary<string, string> translations2) {
    AddNativeKeyword(origName);

    string translation = langDictionary[origName];
    if (string.IsNullOrWhiteSpace(translation)) {
      // No translation is provided for this function.
      translations1[origName] = origName;
      translations2[origName] = origName;
      return;
    }

    AddNativeKeyword(translation);
    translations1[origName] = translation;
    translations2[translation] = origName;

    if (translation.IndexOfAny((" \t\r\n").ToCharArray()) >= 0) {
      throw new ArgumentException("Translation of [" + translation +
                                  "] contains white spaces");
    }
    ParserFunction origFunction = ParserFunction.GetFunction(origName);
    Utils.CheckNotNull(origName, origFunction);
    ParserFunction.RegisterFunction(translation, origFunction);

    // Also add the translation to the list of functions after which
    // there can be a space (besides a parenthesis).
    if (Constants.FUNCT_WITH_SPACE.Contains(origName)) {
        Constants.FUNCT_WITH_SPACE.Add(translation);
    }
    if (Constants.FUNCT_WITH_SPACE_ONCE.Contains(origName)) {
        Constants.FUNCT_WITH_SPACE_ONCE.Add(translation);
    }
  }

  public static void AddNativeKeyword(string word) {
    s_nativeWords.Add(word);
    AddSpellError(word);
  }

  public static void AddTempKeyword(string word) {
    s_tempWords.Add(word);
    AddSpellError(word);
  }

  public static void AddSpellError(string word) {
    if (word.Length > 2) {
      s_spellErrors[word.Substring(0, word.Length - 1)] = word;
      s_spellErrors[word.Substring(1)] = word;
    }
  }
}
```

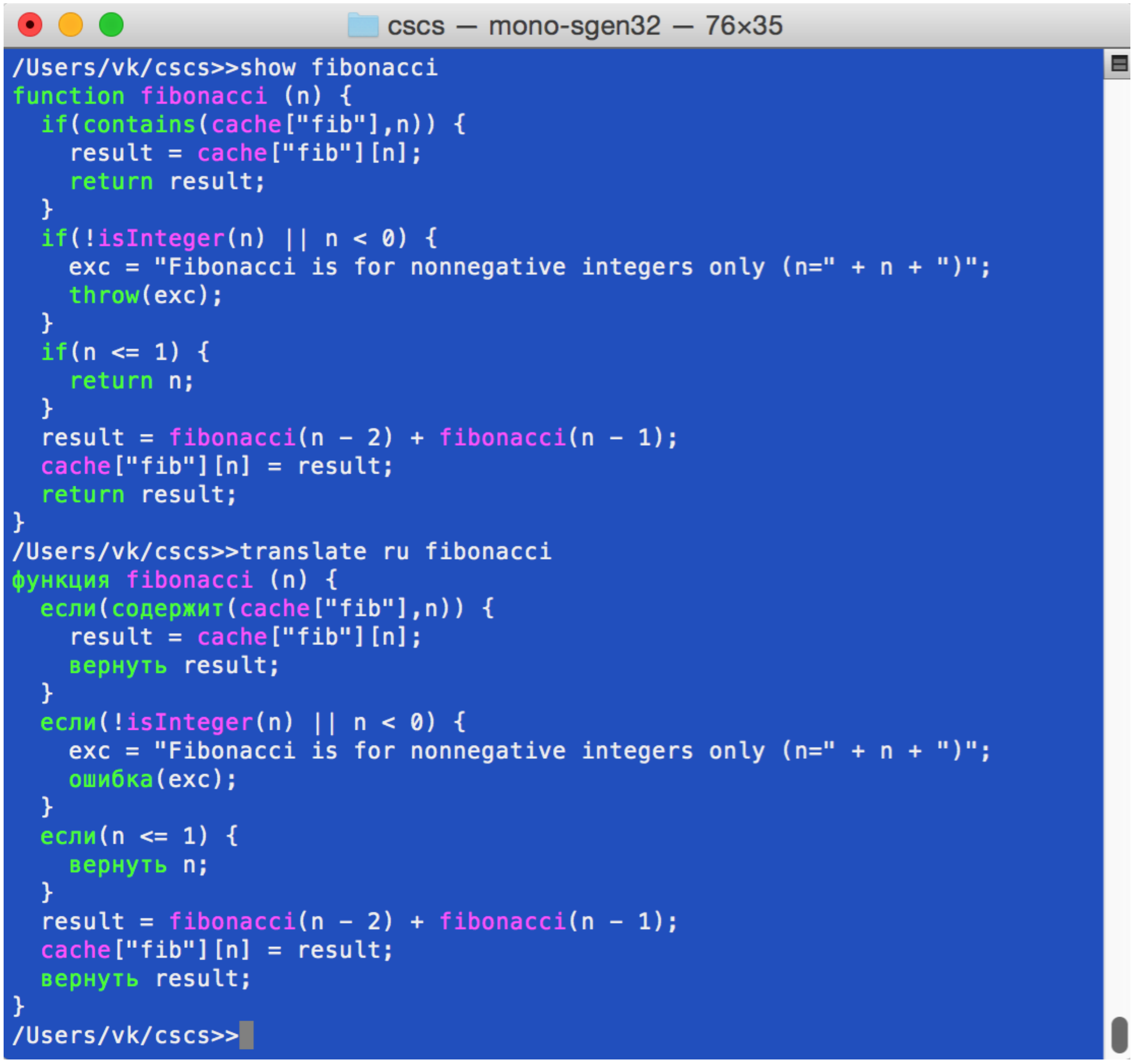
For each pair of languages, we have two dictionaries, each one mapping words from one language to another. In addition to the keywords, you can add translations to any words in the configuration file. We skip loading custom translations, but it can be consulted in the accompanying source code.

Code Listing 72 shows the implementation of the **`TranslateFunction`**, which translates any custom function to the language supplied.

Code Listing 72: The implementation of the function translations

```
class TranslateFunction : ParserFunction
{
  protected override Variable Evaluate (ParsingScript script)
  {
    string language = Utils.GetToken(script, Constants.TOKEN_SEPARATION);
    string funcName = Utils.GetToken(script, Constants.TOKEN_SEPARATION);

    ParserFunction function = ParserFunction.GetFunction(funcName);
    CustomFunction custFunc = function as CustomFunction;
    Utils.CheckNotNull(funcName, custFunc);

    string body = Utils.BeautifyScript(custFunc.Body, custFunc.Header);
    string translated = Translation.TranslateScript(body, language);
    Translation.PrintScript(translated);

    return new Variable(translated);
  }
}

public class Translation
{
  public static string TranslateScript(string script, string fromLang,
                                        string toLang) {
    StringBuilder result = new StringBuilder();
    StringBuilder item = new StringBuilder();

    Dictionary<string, string> keywordsDict =
                                KeywordsDictionary(fromLang, toLang);
    Dictionary<string, string> transDict =
                                TranslationDictionary(fromLang, toLang);
    bool inQuotes = false;

    for (int i = 0; i < script.Length; i++) {
      char ch = script [i];
      inQuotes = ch == Constants.QUOTE ? !inQuotes : inQuotes;

      if (inQuotes) {
        result.Append (ch);
        continue;
      }

      if (!Constants.TOKEN_SEPARATION.Contains(ch)) {
        item.Append(ch);
        continue;
      }

      if (item.Length > 0) {
        string token = item.ToString();
        string translation = string.Empty;
        if (toLang == Constants.ENGLISH) {
          ParserFunction func = ParserFunction.GetFunction(token);
          if (func != null) {
            translation = func.Name;
          }
        }

        if (string.IsNullOrEmpty(translation) &&
            !keywordsDict.TryGetValue(token, out translation) &&
            !transDict.TryGetValue(token, out translation)) {
          translation = token;
        }
        result.Append(translation);
        item.Clear();
      }

      result.Append(ch);
    }

    return result.ToString();
  }
}
```

Figure 7 shows an example of running the **show** and **translate** commands in CSCS. You can see different colors there; this is done in the **Translate.PrintScript** method. This is where **Variable.IsNative** property is used: all “native” (implemented in C#) functions are printed in one color, and all other functions and variables (implemented in CSCS) are printed in another color. The implementation of the **show** function is very similar to the implementation of the **translate** function, shown in Code Listing 72, so we skip it as well.



```
/Users/vk/cscs>>show fibonacci
function fibonacci (n) {
  if(contains(cache["fib"],n)) {
    result = cache["fib"][n];
    return result;
  }
  if(!isInteger(n) || n < 0) {
    exc = "Fibonacci is for nonnegative integers only (n=" + n + ")";
    throw(exc);
  }
  if(n <= 1) {
    return n;
  }
  result = fibonacci(n - 2) + fibonacci(n - 1);
  cache["fib"][n] = result;
  return result;
}
/Users/vk/cscs>>translate ru fibonacci
функция fibonacci (n) {
  если(содержит(cache["fib"],n)) {
    result = cache["fib"][n];
    вернуть result;
  }
  если(!isInteger(n) || n < 0) {
    exc = "Fibonacci is for nonnegative integers only (n=" + n + ")";
    ошибка(exc);
  }
  если(n <= 1) {
    вернуть n;
  }
  result = fibonacci(n - 2) + fibonacci(n - 1);
  cache["fib"][n] = result;
  вернуть result;
}
/Users/vk/cscs>>
```

Figure 7: Show and translate functions run in CSCS

Adding translations for error messages

Now let's see how to add translations to the parser for the error messages. We'll also add possible errors in spelling. Here we use a simplified version, where there is an incorrect or missing first or last letter of the word. Code Listing 73 shows the loading of error messages in different languages, and a sample file in English and German.

Code Listing 73: Loading error messages in different languages

```
public class Translation
{
  public static void LoadErrors(string filename)
  {
    if (!File.Exists(filename)) {
      return;
    }

    Dictionary<string, string> dict = GetDictionary(Constants.ENGLISH,
                                                    s_errors);
    string [] lines = Utils.GetFileLines (filename);
    foreach (string line in lines) {
      string[] tokens = line.Split("=".ToCharArray (),
                                   StringSplitOptions.RemoveEmptyEntries);
      if (tokens.Length < 1 || tokens[0].StartsWith("#")) {
        continue;
      }
      if (tokens.Length == 1) {
        dict = GetDictionary(tokens[0], s_errors);
        continue;
      }
      dict[tokens[0].Trim()] = tokens[1].Trim();
    }
  }
}

// Sample contents of the errors.txt file in English and German.
en
parseToken        = Couldn't parse [{0}] (not registered as a function).
parseTokenExtra   = Did you mean [{0}]?
errorLine         = Line {0}: [{1}]
errorFile         = File: {0}.
de
parseToken        = [{0}] konnte nicht analysiert werden (nicht als
Funktion registriert).
parseTokenExtra   = Meinen Sie [{0}]?
errorLine         = Zeile {0}: [{1}]
errorFile         = Datei: {0}.
```

Code Listing 8 uses the **`Utils.ThrowException`** function, which throws an exception in a language that is configured as user language in the properties file. The implementation of the **`Utils.ThrowException`** function is in Code Listing 74.

Code Listing 74: Implementations of Utils.ThorowException and Translation.GetErrorString

```
public class Utils
{
  public static void ThrowException(ParsingScript script, string excName1,
                                    string errorToken = "", string excName2 = "")
  {
    string msg = Translation.GetErrorString(excName1);

    if (!string.IsNullOrWhiteSpace(errorToken)) {
      msg = string.Format(msg, errorToken);
      string candidate = Translation.TryFindError(errorToken, script);

      if (!string.IsNullOrWhiteSpace(candidate) &&
          !string.IsNullOrWhiteSpace(excName2)) {
        string extra = Translation.GetErrorString(excName2);
        msg += " " + string.Format(extra, candidate);
      }
    }

    if (!string.IsNullOrWhiteSpace(script.Filename)) {
      string fileMsg = Translation.GetErrorString("errorFile");
      msg += Environment.NewLine + string.Format(fileMsg, script.Filename);
    }

    int lineNumber = -1;
    string line = script.GetOriginalLine(out lineNumber);
    if (lineNumber >= 0) {
      string lineMsg = Translation.GetErrorString("errorLine");
      msg += string.IsNullOrWhiteSpace(script.Filename) ?
                                  Environment.NewLine : " ";
      msg += string.Format(lineMsg, lineNumber + 1, line.Trim());
    }
    throw new ArgumentException(msg);
  }
}

public class Translation
{
  public static string GetErrorString(string key)
  {
    string result = null;
    Dictionary<string, string> dict = GetDictionary(s_language, s_errors);
    if (dict.TryGetValue (key, out result)) {
        return result;
    }
    if (s_language != Constants.ENGLISH) {
      dict = GetDictionary(Constants.ENGLISH, s_errors);
      if (dict.TryGetValue(key, out result)) {
        return result;
      }
    }
    return key;
  }
}
```

Consider the following script with a typo in “fibonacci” (an additional “i” at the end):

```
  b = 10;
  c = fibonaccii(b);
```

Here is what our parser prints when running that script with the user language configured as German (see the “language” parameter in the configuration file in Code Listing 68) and loading the **errors.txt** file shown in Code Listing 73:

```
[fibonaccii] konnte nicht analysiert werden (nicht als Funktion registriert).
Meinen Sie [fibonacci]?
Zeile 2: [c = fibonaccii(b);]
```

You can implement catching more advanced spelling errors—not only problems with the first and last letters—for example, by using the Levenshtein distance[15] in strings.

Getting line numbers where errors occur

How do we know which line the error occurred on? (“**Zeile 2**” means “**Line 2**” in German). Most of the information is already in the **`char2Line`** data structure that was loaded in the **`Utils.CovertToScript`** method (see Code Listing 42). But we still need to know what line we are on, only knowing at what character in the script we stopped when the error occurred. Code Listing 75 implements this, using the binary search.

I am not particularly proud of this method of finding the line numbers (even though it works), so hopefully you can come out with a better idea.

Code Listing 75: Implementation of ParsingScript.GetOriginalLineNumber function

```
public int GetOriginalLineNumber()
{
  if (m_char2Line == null || m_char2Line.Count == 0) {
    return -1;
  }
```

[15] See https://en.wikipedia.org/wiki/Levenshtein_distance

```
  int pos = m_scriptOffset + m_from;
  List<int> lineStart = m_char2Line.Keys.ToList();
  int lower = 0;
  int index = lower;

  if (pos <= lineStart[lower]) { // First line.
    return m_char2Line[lineStart[lower]];
  }

  int upper = lineStart.Count - 1;
  if (pos >= lineStart[upper]) { // Last line.
    return m_char2Line[lineStart[upper]];
  }

  while (lower <= upper) {
    index = (lower + upper) / 2;
    int guessPos = lineStart[index];

    if (pos == guessPos) {
      break;
    }

    if (pos < guessPos) {
      if (index == 0 || pos > lineStart[index - 1]) {
        break;
      }
      upper = index - 1;
    } else {
      lower = index + 1;
    }
  }

  return m_char2Line[lineStart[index]];
}
```

Conclusion

In this chapter we saw how to write the CSCS scripts in any language by using different configuration files. Also we saw how to have error messages for programming mistakes in different languages.

In the next chapter we are finally going to talk about testing and how to run CSCS from a shell prompt.

Chapter 8 Testing and Advanced Topics

"Trying to improve software quality by increasing the amount of testing is like trying to lose weight by weighing yourself more often. Don't buy a new scale; change your diet."
Steve McConnell

In this chapter we are finally going to talk about testing. Sure, testing by itself doesn't improve the software quality. But it helps to make sure that the functionality is still correct after any new code changes.

Testing scripts in CSCS

Code Listing 76 shows a testing script written in CSCS. I use it myself to do a unit testing of different language features.

Code Listing 76: A test function in CSCS

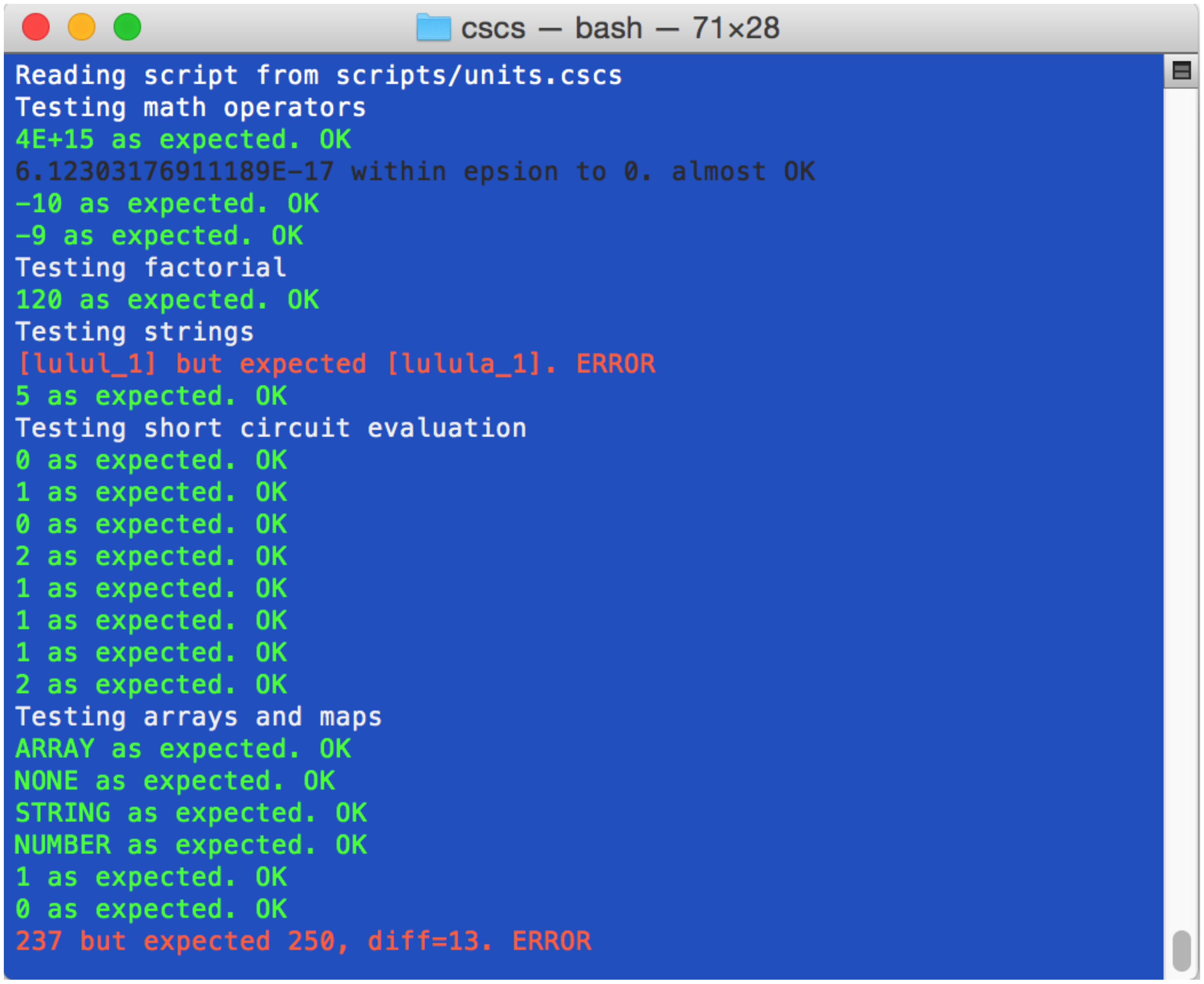
```
function test (x, expected)
{
  if (x == expected) {
    printgreen (x, " as expected. OK");
    return;
  }
  if (type (expected) != "NUMBER") {
    printred ("[", x, "] but expected [", expected, "]. ERROR");
    return;
  }

  epsilon = 0.000001;
  if ((expected == 0 && abs(x) <= epsilon) ||
       abs(x - expected) / expected <= epsilon) {
    printgray (x, " within epsion to ", expected, ". almost OK");
  } else {
    diff = expected - x;
    printred (x, " but expected ", expected, ", diff=", diff, ". ERROR");
  }
}
```

Code Listing 77 shows a fragment of my testing script. It heavily uses the test function defined in Code Listing 76. Basically, for any new feature that I introduce in CSCS, I am trying to write at least one unit test, very much like the tests shown in Code Listing 77.

Code Listing 77: A test script in CSCS using the test function

```
print ("Testing math operators");
test (2.0E+15 + 3e+15 - 1.0e15, 4e+15);
test (cos(pi/2), 0);
a = 10;
test ((a++)-(--a)-a--, a - 2 * a - 1);
test ((a++)-(--a)-a--, a - 2 * a - 1);

print ("Testing factorial");
test (factorial(5), 120);

print ("Testing strings");
txt = "lu";
txt += txt + substr (txt, 0, 1) + "_" + 1;
test (txt, "lulula_1");
ind = indexof (txt, "_");
test (ind, 5);

print ("Testing short circuit evaluation");
function f (x) {
  counter++;
  return x;
}
counter = 0; test (f(0) && f(1), 0); test (counter, 1);
counter = 0; test (f(1) && f(0), 0); test (counter, 2);
counter = 0; test (f(1) || f(2), 1); test (counter, 1);
counter = 0; test (f(0) || f(3), 1); test (counter, 2);

print ("Testing arrays and maps");
arr[2] = 10; arr[1] = "str";
test (type(arr),     "ARRAY");
test (type(arr [0]), "NONE");
test (type(arr [1]), "STRING");
test (type(arr [2]), "NUMBER");

x["bla"]["blu"]=113;
test (contains (x["bla"], "blu"), 1);
test (contains (x["bla"], "bla"), 0);
x["blabla"]["blablu"]=125;
test (--x["bla"]["blu"] + x["blabla"]["blablu"]--, 250);
```

Figure 8 shows the results of running Code Listing 77**Error! Reference source not found.** on a Mac.



```
Reading script from scripts/units.cscs
Testing math operators
4E+15 as expected. OK
6.12303176911189E-17 within epsion to 0. almost OK
-10 as expected. OK
-9 as expected. OK
Testing factorial
120 as expected. OK
Testing strings
[lulul_1] but expected [lulula_1]. ERROR
5 as expected. OK
Testing short circuit evaluation
0 as expected. OK
1 as expected. OK
0 as expected. OK
2 as expected. OK
1 as expected. OK
1 as expected. OK
1 as expected. OK
2 as expected. OK
Testing arrays and maps
ARRAY as expected. OK
NONE as expected. OK
STRING as expected. OK
NUMBER as expected. OK
1 as expected. OK
0 as expected. OK
237 but expected 250, diff=13. ERROR
```

Figure 8: Running CSCS testing script

Logical negation

Another case that needs to be looked at separately is a negation of a Boolean expression. The definition of a logical negation is simple: applied to an expression, it coverts it to false if it is true, and to true, if it is false. In most programming languages this negation is represented by signs **`!`** or **`¬`**, or by a **`NOT`** keyword. We are going to use the exclamation sign, but you can redefine it as you wish in the **`Constants`** class:

```
public const string NOT = "!";
```

We add the implementation of the Boolean negation to the **`Parser.Split`** method (shown in Code Listing 4). See details in Code Listing 78**Error! Reference source not found.**. Note that our implementation supports multiple negations. For example, **`!!!a`** is equivalent to **`!a`**. But **`!!a`** is not necessarily equivalent to **`a`**, since applying the logical negation converts number to a Boolean, represented as either 0 or 1 in CSCS, so **`!!a`** will be either 0 or 1.

Code Listing 78: The implementation of the Boolean negation in Parser.Split method

```
int negated = 0;

do { // Main processing cycle of the first part.
  string negateSymbol = Utils.IsNotSign(script.Rest);
  if (negateSymbol != null) {
    negated++;
    script.Forward(negateSymbol.Length);
    continue;
  }
  // Code as before...

  Variable current = func.GetValue(script);

  if (negated > 0 && current.Type == Variable.VarType.NUMBER) {
    // If there has been a NOT sign, this is a Boolean.
    // Use XOR (true if exactly one of the arguments is true).
    bool neg = !((negated % 2 == 0) ^ Convert.ToBoolean(current.Value));
    current = new Variable(Convert.ToDouble(neg));
    negated = 0;
  }
  // Code as before...
}

// Implementation of the Utils.IsNotSign:
public static string IsNotSign(string data)
{
  return data.StartsWith(Constants.NOT) ? Constants.NOT : null;
}
```

The complete code for the **`Parser.Split`** method can be consulted in the accompanying source code. It also deals with the quotes (the original version of the **`Parser.Split`** method in Code Listing 4 processes all expressions inside of quotes, instead of taking them as a part of a string).

Running CSCS as a shell program

Code Listing 79 shows an excerpt from Main.cs. There are two modes of operation—in case you supply a file, the whole script containing the file will be executed. It will be done in the **`Main.ProcessScript`** method, which is very similar to the **`IncludeFile`** functionality in Code Listing 40. Otherwise, the **`Main.RunLoop`** method is called, which will process entered commands one by one.

An excerpt from the **`Main.RunLoop`** is shown in Code Listing 79, but there are much more than you see there. In particular, there is a functionality to auto-complete file or directory names if you press the Tab key. It's beyond the scope of this book, but you're cordially invited to check it out in the accompanying source code.

Code Listing 79: An Excerpt from Main.cs

```
static void Main(string[] args)
{
  Console.OutputEncoding = System.Text.Encoding.UTF8;
  ProcessScript("include(\"scripts/functions.cscs\");");
  string script = null;

  if (args.Length > 0) {
    if (args[0].EndsWith(EXT)) {
      string filename = args[0];
      Console.WriteLine("Reading script from " + filename);
      script = GetFileContents(filename);
    } else {
      script = args[0];
    }
  }

  if (!string.IsNullOrWhiteSpace(script)) {
    ProcessScript(script);
    return;
  }

  RunLoop();
}

private static void RunLoop()
{
  List<string> commands = new List<string>();
  StringBuilder sb = new StringBuilder();
  int cmdPtr = 0;
  int tabFileIndex = 0;
  bool arrowMode = false;
  string start = "", baseCmd = "", startsWith = "", init = "", script;
  string previous = "";

  while (true) {
    sb.Clear();
    script =  GetConsoleLine(ref nextCmd, init).Trim();

    if (!script.EndsWith(Constants.END_STATEMENT.ToString())) {
      script += Constants.END_STATEMENT;
    }

    ProcessScript(script);
  }
}
```

Running CSCS on non-Windows systems

"I think Microsoft named .NET so it wouldn't show up in a Unix directory Code Listing."
Oktal

Using the Mono Framework[16] and Xamarin Studio Community,[17] you can now develop .NET applications on Unix systems, in particular on macOS. Things became even brighter for non-Windows .NET developers after Microsoft acquired Xamarin Inc. in 2016. The Xamarin Studio with Mono framework is what I used to port CSCS on macOS. The code is basically identical on macOS and on Windows, unless you are writing GUI applications. There are a few quirks even for non-GUI development, some of which we are going to see in this section.

Sometimes we need to know whether the code being executed is on Windows or on another platform. It's easy to figure out at runtime—you can use the **`Environment.OSVersion`** property. But what do you do if you want to know which OS your .NET code is being compiled on? You might want to know that in order to compile different code on different OS. Previously you could use **`#ifdef __MonoCS__`** macro for that, but with latest Mono releases, it appears that this macro is no longer set. What I did is the following: I defined the **`__MonoCS__`** symbol in the project properties in Xamarin Studio, so I can still use it when compiling on macOS.

Once you compile your project with Xamarin Studio, how do you run it from the command line? Code Listing 80 contains a wrapper script; I called it CSCS, so you can just run that script.

Code Listing 80: cscs file: the CSCS starting script on Unix

```
#!/bin/sh
/usr/local/bin/mono cscs.exe "$@"
```

The script should be placed in the same directory as the **cscs.exe** file. It assumes that the Mono Framework was installed at `/usr/local/bin/mono.` Note that this script will pass on all the arguments to **cscs.exe**.

Listing directory entries

As an example of a different C# code on different operating systems, let's see the directory listing function. The directory listing is commonly known as **`dir`** on Windows and **`ls`** on Unix.

The file system and user permissions are implemented differently on Windows and on Unix. For example, on Unix there is a concept of a user, users of a group to which the user belongs, and all other users. Each of these entities has a combination of read, write, and execution permissions on a file or a directory (for instance, write permissions don't automatically imply read permissions). You can't translate such concepts one-to-one to Windows.

[16] Mono Framework is an open source multi-platform implementation of the .Net Framework. See http://www.mono-project.com

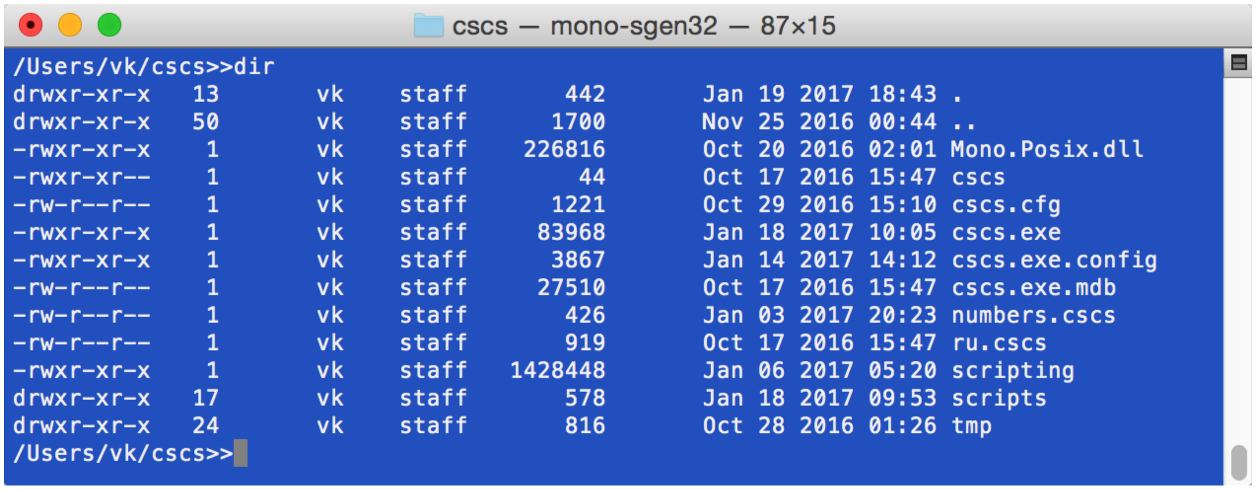
[17] Xamarin Studio Community is a free IDE to develop C# applications on a Mac using Mono Framework. See https://www.xamarin.com/download

We can have both function names, **ls** and **dir**, pointing to the same implementation function (Code Listing 68 shows how to do that). Code Listing 81 implements getting details for a file or a directory on either Windows or macOS. We call this function in a loop for each directory entry. To the curious reader: we are implementing here the **ls -la** Unix command, not just **ls**.

Code Listing 81: Implementation of Utils.GetPathDetails on Windows and macOS

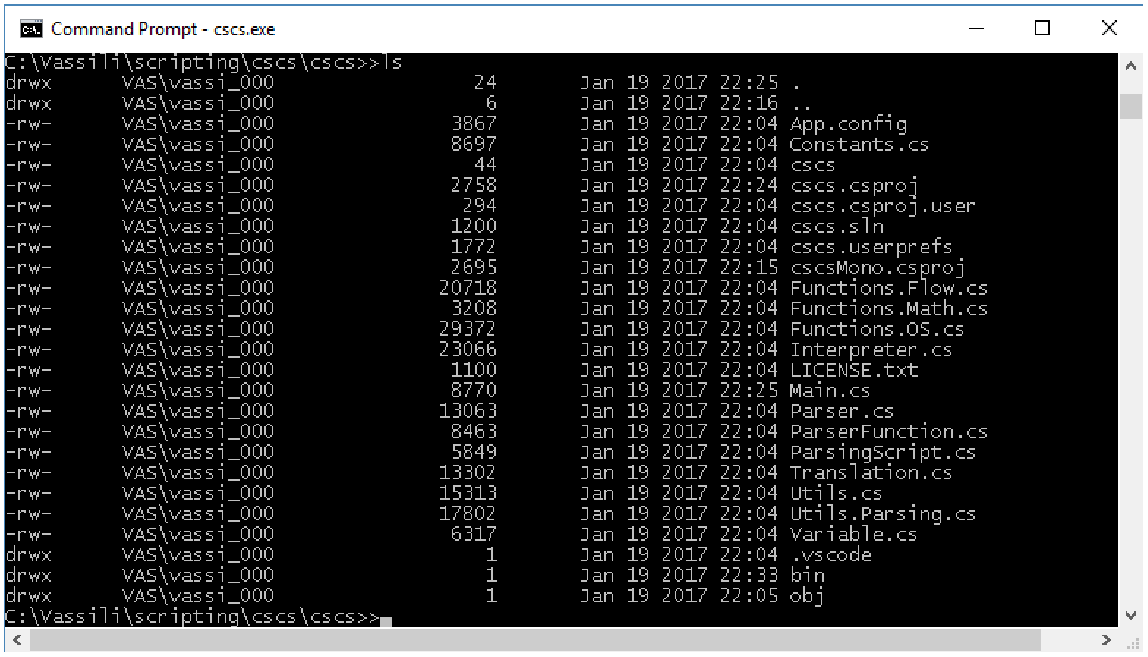
```
public static string GetPathDetails(FileSystemInfo fs, string name)
{
  string pathname = fs.FullName;
  bool isDir = (fs.Attributes & FileAttributes.Directory) != 0;
  char d = isDir ? 'd' : '-';

  string last  = fs.LastWriteTime.ToString("MMM dd yyyy HH:mm");
  string user = string.Empty;
  string group = string.Empty;
  string links = null;
  string permissions = "rwx";
  long size = 0;

  #if __MonoCS__
    Mono.Unix.UnixFileSystemInfo info;
    if (isDir) {
      info = new Mono.Unix.UnixDirectoryInfo(pathname);
    } else {
      info = new Mono.Unix.UnixFileInfo(pathname);
    }
    char ur = (info.FileAccessPermissions &
         Mono.Unix.FileAccessPermissions.UserRead)     != 0 ? 'r' : '-';
    char uw = (info.FileAccessPermissions &
         Mono.Unix.FileAccessPermissions.UserWrite)    != 0 ? 'w' : '-';
    char ux = (info.FileAccessPermissions &
         Mono.Unix.FileAccessPermissions.UserExecute)  != 0 ? 'x' : '-';
    char gr = (info.FileAccessPermissions &
         Mono.Unix.FileAccessPermissions.GroupRead)    != 0 ? 'r' : '-';
    char gw = (info.FileAccessPermissions &
         Mono.Unix.FileAccessPermissions.GroupWrite)   != 0 ? 'w' : '-';
    char gx = (info.FileAccessPermissions &
         Mono.Unix.FileAccessPermissions.GroupExecute) != 0 ? 'x' : '-';
    char or = (info.FileAccessPermissions &
         Mono.Unix.FileAccessPermissions.OtherRead)    != 0 ? 'r' : '-';
    char ow = (info.FileAccessPermissions &
         Mono.Unix.FileAccessPermissions.OtherWrite)   != 0 ? 'w' : '-';
    char ox = (info.FileAccessPermissions &
         Mono.Unix.FileAccessPermissions.OtherExecute) != 0 ? 'x' : '-';

    permissions = string.Format("{0}{1}{2}{3}{4}{5}{6}{7}{8}",
        ur, uw, ux, gr, gw, gx, or, ow, ox);

    user  = info.OwnerUser.UserName;
    group = info.OwnerGroup.GroupName;
    links = info.LinkCount.ToString();

    size = info.Length;

    if (info.IsSymbolicLink) {
      d = 's';
    }
  #else
    if (isDir) {
      user = Directory.GetAccessControl(fs.FullName).GetOwner(
        typeof(System.Security.Principal.NTAccount)).ToString();
      DirectoryInfo di = fs as DirectoryInfo;
      size = di.GetFileSystemInfos().Length;
    } else {
      user = File.GetAccessControl(fs.FullName).GetOwner(
        typeof(System.Security.Principal.NTAccount)).ToString();
      FileInfo fi = fs as FileInfo;
      size = fi.Length;

      string[] execs = new string[] { "exe", "bat", "msi"};
      char x = execs.Contains(fi.Extension.ToLower()) ? 'x' : '-';
      char w = !fi.IsReadOnly ? 'w' : '-';
      permissions = string.Format("r{0}{1}", w, x);
    }
  #endif

  string data = string.Format("{0}{1} {2,4} {3,8} {4,8} {5,9} {6,23} {7}",
                     d, permissions, links, user, group, size, last, name);
  return data;
}
```

To use Unix-specific functions, you have to reference the Mono.Posix.dll library. One of the solutions is to have two project configuration files—one with the `__MonoCS__` symbol and Mono.Posix.dll library references (to be used on Unix systems), and another one, without them, to be used on Windows. You may want to have a look at how I did this in the accompanying source code.



```
/Users/vk/cscs>>dir
drwxr-xr-x    13        vk     staff       442        Jan 19 2017 18:43 .
drwxr-xr-x    50        vk     staff      1700        Nov 25 2016 00:44 ..
-rwxr-xr-x     1        vk     staff    226816        Oct 20 2016 02:01 Mono.Posix.dll
-rwxr-xr--     1        vk     staff        44        Oct 17 2016 15:47 cscs
-rw-r--r--     1        vk     staff      1221        Oct 29 2016 15:10 cscs.cfg
-rwxr-xr-x     1        vk     staff     83968        Jan 18 2017 10:05 cscs.exe
-rwxr-xr-x     1        vk     staff      3867        Jan 14 2017 14:12 cscs.exe.config
-rw-r--r--     1        vk     staff     27510        Oct 17 2016 15:47 cscs.exe.mdb
-rw-r--r--     1        vk     staff       426        Jan 03 2017 20:23 numbers.cscs
-rw-r--r--     1        vk     staff       919        Oct 17 2016 15:47 ru.cscs
-rwxr-xr-x     1        vk     staff   1428448        Jan 06 2017 05:20 scripting
drwxr-xr-x    17        vk     staff       578        Jan 18 2017 09:53 scripts
drwxr-xr-x    24        vk     staff       816        Oct 28 2016 01:26 tmp
/Users/vk/cscs>>
```

Figure 9: Running dir (ls) command on macOS



```
C:\Vassili\scripting\cscs\cscs>>ls
drwx       VAS\vassi_000                24        Jan 19 2017 22:25 .
drwx       VAS\vassi_000                 6        Jan 19 2017 22:16 ..
-rw-       VAS\vassi_000              3867        Jan 19 2017 22:04 App.config
-rw-       VAS\vassi_000              8697        Jan 19 2017 22:04 Constants.cs
-rw-       VAS\vassi_000                44        Jan 19 2017 22:04 cscs
-rw-       VAS\vassi_000              2758        Jan 19 2017 22:24 cscs.csproj
-rw-       VAS\vassi_000               294        Jan 19 2017 22:04 cscs.csproj.user
-rw-       VAS\vassi_000              1200        Jan 19 2017 22:04 cscs.sln
-rw-       VAS\vassi_000              1772        Jan 19 2017 22:04 cscs.userprefs
-rw-       VAS\vassi_000              2695        Jan 19 2017 22:15 cscsMono.csproj
-rw-       VAS\vassi_000             20718        Jan 19 2017 22:04 Functions.Flow.cs
-rw-       VAS\vassi_000              3208        Jan 19 2017 22:04 Functions.Math.cs
-rw-       VAS\vassi_000             29372        Jan 19 2017 22:04 Functions.OS.cs
-rw-       VAS\vassi_000             23066        Jan 19 2017 22:04 Interpreter.cs
-rw-       VAS\vassi_000              1100        Jan 19 2017 22:04 LICENSE.txt
-rw-       VAS\vassi_000              8770        Jan 19 2017 22:25 Main.cs
-rw-       VAS\vassi_000             13063        Jan 19 2017 22:04 Parser.cs
-rw-       VAS\vassi_000              8463        Jan 19 2017 22:04 ParserFunction.cs
-rw-       VAS\vassi_000              5849        Jan 19 2017 22:04 ParsingScript.cs
-rw-       VAS\vassi_000             13302        Jan 19 2017 22:04 Translation.cs
-rw-       VAS\vassi_000             15313        Jan 19 2017 22:04 Utils.cs
-rw-       VAS\vassi_000             17802        Jan 19 2017 22:04 Utils.Parsing.cs
-rw-       VAS\vassi_000              6317        Jan 19 2017 22:04 Variable.cs
drwx       VAS\vassi_000                 1        Jan 19 2017 22:04 .vscode
drwx       VAS\vassi_000                 1        Jan 19 2017 22:33 bin
drwx       VAS\vassi_000                 1        Jan 19 2017 22:05 obj
C:\Vassili\scripting\cscs\cscs>>
```

Figure 10: Running ls (dir) command on Windows

Figure 9 shows the CSCS code running on macOS, and Figure 10 shows the CSCS code running on Windows. As you can see, the results are platform dependent.

Command line commands in CSCS

"It won't be covered in the book. The source code has to be useful for something, after all..."
Larry Wall

Analogously to the directory listing command, developed in the previous section, many other commands can be implemented.
 shows an implementation of a command to clear the console. It also has also necessary steps to register this command with the parser and allow translations for this command in the configuration file.

Code Listing 82: Implementation of the ClearConsole class

```
// 1. Definition in Constants class.
public const string CONSOLE_CLR = "clr";

// 2. Registration with the parser in Interpreter.Init().
ParserFunction.RegisterFunction(Constants.CONSOLE_CLR, new ClearConsole());

// 3. Registration for keyword translations in Interpreter.ReadConfig().
Translation.Add(languageSection, Constants.CONSOLE_CLR, tr1, tr2);

// 4. Implementation of a class deriving from the ParserFunction class.
class ClearConsole : ParserFunction
{
  protected override Variable Evaluate(ParsingScript script)
  {
    Console.Clear();
    return Variable.EmptyInstance;
  }
}
```

In the same way, we can implement many more commands. I refer you to the source code area, where you can find the implementation of the following command-prompt commands: copying, moving, and deleting files, changing the directory, starting and killing a process, listing running processes, searching for files, etc.

You now have a tool with which you can create your own programming language. To extend the language, you just add your own functions, analogous to Code Listing 82.

To continue the exploration of this subject, I would suggest downloading the accompanying source code and start playing with it: add a few functions, starting with very simple ones (like printing a specific message to a file). Then you can gradually increase complexity and go towards a functional language. For example, you could implement a function that can read and manipulate large Excel files from CSCS. Here is an example of how you can do it in C#.

I am looking very much forward to your feedback about what can be improved and what functions you have implemented in CSCS!