DotPar
Final Report
Team 3

Logan Donovan - lrd2127@columbia.edu
Justin Hines - jph2149@columbia.edu
Andrew Hitti - aah2147@columbia.edu
Nathan Hwang - nyh2105@columbia.edu Sid Nair - ssn2114@columbia.edu

May 4, 2012

Part I
Introduction

1 Motivation

Moores Law, the trend that the number of transistors that can fit on a chip doubles every 2 years, has held true for half a century, driving innovations in all areas of technology. But recently chip manufacturers have moved away from ever-increasing clock speeds, the usual manifestation of Moores law, and instead increase the number of cores in CPUs to manage power dissipation.

As consumer and business demands for speed grow, programs must now be written to take advantage of multiple cores in order to utilize the power of CPUs. However, the imperative programming paradigm that has prevailed in the programming community isnt conducive to safe, parallel programming. Parallel programming is difficult and synchronization bugs are notoriously difficult to fix.

Many solutions to this problem of parallelization have emerged, but each has its drawbacks. We will briefly survey a few of the most prominent existing solutions.

1.1 Locking

Locking controls access to memory accessible by several threads of execution. However, this requires that the programmer manually lock regions of code, which is tedious and error prone. Historically, it can lead to deadlock and race conditions, both of which are extremely hard to debug.

1.2 Task-based Concurrency and the Actor Model

Task-based concurrency, used in languages like Ada and X10, doesnt expose threads to the programmer. Simple computations, not threads are the unit of parallelization. The actor model, present in SmallTalk and Scala, treats different threads as actors that communicate with each other via message passing. Since the actors dont share state, this is much less error prone than locking since race conditions can be avoided entirely. These models can be elegant, but they still require manual division of labor on the part of the programmer, which introduces errors and overhead, especially when refactoring code. We seek to avoid imposing this burden on programmers.

1.3 Functional Programming

The functional paradigm minimizes side effects in favor of deterministic functions. Under this model, functions could easily be parallelized because they execute independently. Functional languages like Haskell boast compilers that produce a great degree of parallelization.

However, the functional paradigm is an unfamiliar way of coding for many programmers and having state is often a preferable way of modeling a program. In particular, object-oriented approaches have proven useful in building large applications. These factors have limited the adoption of purely functional languages.

Additionally, the benefits of parallelizing a purely functional programming languages is limited because objects must be frequently copied because they are immutable. This means that cache hit rates drop significantly, main memory bandwidth becomes a bottleneck, and garbage collection becomes a pain[1].

1.4 Implicit Parallelization

Some efforts to write implicitly parallel, imperative languages have been made[2]. However, state- of-the-art compilers are not yet capable of good implicit parallelization. While some cases of potential parallelization are easy to recognize, many of them difficult or impossible to find with static analysis[3].

With this in mind, implicit parallelization may sound daunting, but it is important to remember that the compiler can miss many potential opportunities for parallelization without sacrificing performance. Even with a modest ability to find parallelization opportunities, the performance of parallel programs is still limited by the number of processors, not the number of threads that run simultaneously. In fact, if too many threads are created, the overhead of thread creation and execution will adversely affect performance.

[1] http://bit.ly/dotpar1

[2] See http://bit.ly/dotpar2 and http://bit.ly/dotpar3 for two examples.

[3] http://bit.ly/dotpar4

2 Introducing DotPar

With familiar C-like syntax, DotPar is a multi-paradigm language that implicitly parallelizes code with minimal programming effort. By abstracting away parallelization, DotPar enables programmers to focus on designing and and architecting systems instead of on micromanaging performance. Additionally, it provides the expressive power of functional languages and includes some imperative paradigms derived from C to ease the job of the programmer.

2.1 Automatic Parallelization

Based on static analysis of code and user annotations, DotPar can detect a large class of parallelization opportunities, speeding up execution considerably.

2.2 Static and Strong Typing

Keeping DotPar strongly and statically typed provides compile-time check for common mistakes that should never make it to production. This also enhances the compilers ability to parallelize the code.

2.3 First-class Functions and Lexical Closures

By providing first-class functions and closures, DotPar provides a great degree of expressiveness not possible in a language like Java. However, it is not as restrictive as a purely functional language like Haskell because it provides some imperative constructs as well.

2.4 Robustness and Security

Through implicit parallelization, DotPar removes the risk of race conditions and deadlock. And, since it runs on the JVM, the code is highly robust and portable. Furthermore, DotPar raises informative errors that allow the user to debug code efficiently and quickly, reducing development time.

2.5 Memory Management and Garbage Collection

Since DotPar runs on the JVM, garbage collection and memory management are handled automatically, reducing programmer overhead.

Part II
Language Tutorial

Parallel algorithms tend to be described as operations on collections of values. For instance, “find the minimum neighbor for each vertex”, or “sum each row of a matrix”

are data-centric operations that can be done in parallel. [4] This ability to operate in parallel over sets of data is often referred to as data-parallelism. It is important that this data-parallelism can be exploited to its full potential. We want to be able to run parallel functions in parallel to maximize the efficiency of the program. Thus, we seek to provide an effective nested data-parallel language.

The following tutorial will give a quick tour of the basics of the language and build up to defining complex array manipulations to enable you to start writing useful programs as soon as possible. To do that, we'll concentrate on the basics: types, arithmetic expressions, arrays, control flow, functions, list comprehensions, and some nifty built-in functions.

3 Overview

3.1 Hello, World

DotPar was designed with simplicity and power in mind. To illustrate this, let's get started with a traditional "Hello, World" program that prints the words "Hello, World" to stdout.

```
func main:void()
{
    println("Hello, world!");
}
```

Just like in C and Java, the DotPar execution begins with a function named `main`. Thus, every program must contain a main function. In this `main` method, we use the built-in function `print` to print the `string` "Hello, world". A `string` is an `array` of characters. We will explain exactly what `function`s, `string`s, and `char`s in later. But first, let's get this program running.

To compile and run this program there are three steps (assuming a UNIX-like system). First, we should save the function in a file called `helloWorld.par`. DotPar programs are stored in files that have the extension `.par`. Next, we run a compiler on the program file, like so:

```
dotparc helloWorld.par
```

`dotparc` can accept multiple files. So, to compile all the .par files in a directory, you would run:

```
dotparc *.par
```

If the program residing in the file has no syntax errors and is a complete DotPar program then the compiler will produce an executable file:

[4] http://www.cs.cmu.edu/ scandal/cacm/node4.html

```
out.parc
```

If there are errors during compilation those will be printed to your console. In this example, running out.parc in the command line with:

```
./out.parc
```

will produce the output:

```
Hello, World
```

We'll now walk you through some key features of the language. As you go along, try writing and compiling these programs yourself.

3.2 Types And Arithmetic Expressions

DotPar contains the concept of types which are used to define a variable before its use. The definition of a variable will assign a store address for the variable and define the type of data that will be held at that address. DotPar only three contains basic types: `number`, `boolean`, and `char`.

3.2.1 Numbers and Basic Arithmetic Operators

All numbers in DotPar are double-precision 64-bit floating point numbers. This basic type is referred to as `number`. This simplicity allows for computations to be straightforward while maintaining incredible precision. We realized that since numbers were so imporant in the language that simplifying everything to high precision and using the parallelization for performance gains would make the language easier to use.

Our next program will illustrate the use of numbers and some basic arithmetic expressions that can be used to manipulate numbers.

```
func main:void()
{
    number a = 1;
    number b = 2;
    println(b);
    println(a / b);
    number c = a + b;
    println(c);
}
```

The first `println` statement may be counterintuitive to what one might expect. This statement will not return `2`, but rather `2.0`. Following this logic, the second print statement should, and does return `.5`. The last print statement's output should now be obvious as `3.0`.

Other basic arithmetic operators include `-` , `*`, and `/` which represent subtraction, multiplication, and division, respectively. There is also a remainder operator, `\%`, which returns the remainder of the division of two numbers.

Note: `+`, `-`, `*`, `/`, `sqrt`, `ln`, `log(num, base)`, `exp(num, exponent)`, `ceil`, `floor`, `trunc`, `round`, `sin(x)`, `cos(x)`, `tan(x)`, `asin(x)`, `acos(x)`, `atan(x)` are all supported. The trigonometric functions return a value in radians, and the inverse trigonometric functions expect a value in radians.

3.2.2 Booleans and Operators

The boolean data type has only two possible values, `true` and `false`. This uses simple flags that track true/false conditions. While this data type represents one bit of information, but it is treated internally as a 32-bit entity. We decided to have booleans because they are a very useful data type. We implemented them as 32-bit entities because that is how Java does it.

The following program will give an elementary example of the use of booleans, and their uses.

```
func main:void()
{
    boolean b = true;
    println(!b);
    boolean c = false;
    println(b && c);
    println(c || b);
}
```

Since a boolean tracks a single true/false condition, each print statement will only return either `true` or `false`. The ! symbol is the NOT unary operator and evaluates to the complement of its argument. Since `b` is `true`, the first print statement will be "`false`". The `&&` operator signifies the binary operator AND which returns the conjunction of its two arguments. So the second print statement will return false, since true AND false evaluates to false. The `||` operator signifies the binary operator OR which returns the disjunction of its two operators. So the last print statement will return true, since false OR true evaluates to true.

3.2.3 Char

The char data type is a single 16-bit Unicode character.

The following example will given an elementary example of the use of chars.

```
func main:void()
{
    char c = 'a';
    println(c);
}
```

The above will print out “a”, without the quotations. Often chars by themselves are not very useful, as in reality we use a sequence of chars to represent words, or strings. This provides a nice transition to DotPar’s arrays.

3.3 Arrays and Control Flow

There are times when writing programs when you may want to store multiple items of same type together. Arrays provide this functionality. They are contiguous bytes in memory, which facilitates easy searching and manipulation of lists of elements. Like everything else in the language, arrays are typed. You can have an array of any type, such as `number`, `boolean`, `char`, or another array. Let’s take a look at some examples.

3.3.1 Revisiting Hello, World

```
func main:void()
{
    char[] helloWorld = "Hello, World";
    println(helloWorld);
}
```

The first program we learned to write prints the words “Hello, World.” In this equivalent program, we explicitly create an array of type `char` in the first line of `main`. An arrays of chars is also often called a `string`. Arrays are very fundamental building blocks in DotPar. Rather than have many built-in types, DotPar has just a few types and gives users the power to create more complicated types. So, there is no built-in `string` type, so there is instead only an arrays of characters. So, the first line creates a character array with each element being a character in the statement, “Hello, World”.

Arrays are where DotPar leverages its power as it can manipulate large sets of data quickly by manipulating arrays, iterating through arrays, and applying functions to arrays.

3.4 Array Iteration

3.4.1 the each method

Our next example program creates an array of type number and demonstrates a method of iterating through an array. The first line creates an array arr of size 10 and populates it with the numbers 1-10. Notice at the end of the line there is a `//`. This is a single-line comment. `//` will cause the compiler to ignore the rest of the line. Comments are a useful to markup text to explain subtleties of the code. They are ignored by the compiler, so they are not evaluated. The second line demonstrates the use of the built-in `each` function, which is used to iterate over an array element by element. For each loops contain all of its statements within curly brackets. This for each iterates through every number in arr and calls the function print on every element. This prints a list of the elements in arr to standard out.

```
func main:void()
{
    number[] arr = [1,2,3,4,5,6,7,8,9,10];  //create the array
    each(arr, func:void(number n) { println(n); });
}
```

3.4.2 the for loop

Another way to write the program above would be to use a `for` loop, which can be used to iterated over the indices of an array by using a counter. So, the variable `i` of type number is created and is initialized to 0. It then checks to ensure that `i` is less than `len(arr)`. `len` is an example of a built-in function that returns a `number` whose value is the length of the array in its argument. In `for` loop we are again calling the `print` function and are passing in `arr[i]`, which is the value in the array at index `i`. The `for` loop then lastly increments `i` by 1.

```
func main:void()
{
    number[] arr = [1,2,3,4,5,6,7,8,9,10];
    number i;
    for(i = 0; i < len(arr); i = i + 1) {
        println(arr[i]);
    }
}
```

Arrays can be extended to arbitrarily many dimensions. For example, we create a matrix in the example below.

```
func main:void()
{
    number[][] matrix = [ [1,2,3], [4,5,6], [7,8,9] ];
    println(matrix);
}
```

Here we are creating a two-dimensional `3x3` array with the numbers 1 through 9. We have a set of `[ ]` for each dimension we want create. We can fill this array with nested brackets by defining each row within brackets. Try running this program and see what it prints!

3.5 Control Statements

Next, we'll look at conditional statements, random number generation, and the fill statement in DotPar.

Let's take a look at our next example program.

```
func main:void()
{
    /*
    This program creates an array of length 10 and
    fills it with random numbers in the range [0-100).
    It then prints out each element with
    a message declaring it even or odd.
    */

    number[] arr;
    fill(arr, func:number(number index) { return rand(100); }, 10);
    char[] even = is even;
    char[] odd = is odd;

    each(arr, func:void(number element) {
        if (element % 2 == 0) {
            println(element +   + even);
        } else {
            println(element +   + odd);
        }
    });
}
```

First, notice that the first few lines of the function demonstrate another way to comment in DotPar by using traditional C style comments, `/* ... */` to comment out an entire block of text.

Next, we create an number array arr. The next line calls the built-in function `fill(array, function)` which takes in an array and a function as arguments and inserts values into the array according to the passed in function. In this case we are passing in the `rand(number)` function. `rand` generates random numbers in the range [0, cap). Afterwards, we create two character arrays that contain the text "is even" and " is odd" which will be used later.

Next we have a call to our old friend, the `each` function, that iterates through each element in the array. Within the body of the `each` function we have our first example of `if` statements in DotPar. They follow the form:

```
if (conditional) {
   // code
} elif (conditional) {
   // code
} else {
   // code
}
```

There must be exactly one `if` clause, any number of `elif` clauses, and no more than

one `else` clause.

The parenthetical after the `if` is the condition that must be met in order to run the code within the braces. If this condition is true, the code with the braces of the else is run. The conditional we test for in this example is `element \% 2` $= 0=$. `element` is the value in the array, the `\%` is the arithmetic operation for modulus, the

`==`

3.6 List Comprehensions

As a more complicated example, you may want to write something like:

```
number[] foo = [1,2,3,4,5];
number[] squares;
number i;
for(i = 0; i < len(squares) ; i = i + 1){
    if (squares[i] % 2 == 0) {
        squares[i] = exp(foo[i], 2);
    }
}
```

This function creates an array with the square of the even numbers in the original array. But this is rather verbose, which seems unnecessary for a conceptually simple task like this. DotPar provides an easy way to express this logic, borrowing the useful list comprehension from Python and Haskell. We will illustrate the power of list comprehensions by replacing the previous program fragment in one line:

```
number[] squares = [exp(x,2) for number x in [1,2,3,4,5] if (x % 2 == 0)];
```

You can also use an expanded notation for using n arrays, such as

```
number[] foo = [x*y for number x, number y in [1, 3, 9], [1, 2, 3] if (x != 1)];
```

3.7 Working with Functions

In DotPar, functions are first-class, which means they can be assigned to variables or passed to other functions as arguments. A function provides a convenient way to encapsulate some computation, which can then be used without stressing its implementation. All functions are declared with the `func` reserved word, and can be nested within each other.

A function definition has this form:

```
func function-name : return-type(optional parameter declarations)
{
    declarations and statements;
    return return-val;
}
```

3.7.1 Revisiting main

With this in mind, let's take a look at `main` again. As in Java, `main` can accept arguments. However, as in C, the arguments can be omitted.

```
func main: void(char[][] args)
{
    if (len(args) > 0) {
      each(args, func: void(char[] s) { println(s); });
    }
    else {
      println(No arguments passed!);
    }
}
```

Try running this program, passing various arguments and check out the results.

3.7.2 Declaring an Array

Next, let's fill a create a matrix of integers. This example will demonstrate how to pass functions as parameters.

```
func main: void()
{
// This creates an array with 10 rows and 20 columns. The values are not populated,
// so they are random.
    number[10][20] b;
}
```

3.8 Idioms and Parallelism

Our next sample programs will contain more complex array manipulation and introduce some useful functions for arrays. These functions aren't common in imperative languages, and they exhibit some nice properties of DotPar.

3.8.1 Map

`map(array, function)` calls `function(item)` for each of the array's items and returns an array of the return values. For example, let's compute some cubes:

```
func main:void(){
    func cube:number (number value, number index){
        return value*value*value;
    }
    number[] a = [1,2,3,4,5];
    map(a, cube);
}
```

Note that this example could have been computed more succinctly with a list comprehension. However, in some cases, you may find a `map(array, function)` to be clearer. When you run map it returns a transformed list. Whereas when you run each it performs the functions and returns nothing, instead usings the

3.8.2 Reduce

`reduce(array, function, original)` returns a single value constructed by applying the reducing function repeatedly to the reduced value so far and each element of the array. For instance, if you wanted to add all the elements of a numerical array, one could define the function `sum(number a, number b)`, and apply `sum` to the array.

To break things down, we take the first two elements of the array and apply `sum` to them. This returns the sum of the elements, our first result. Then, we take the third element of the array, and apply `sum` to our first result and the third element, obtaining our second result. We continue in this manner, until we have one result and no more elements to reduce.

For a concrete example, let's compute the sum of the numbers 1 to 10000:

```
func main:void(){
   func sum:number (number a, number b){
       return a+b;
   }
   number[] x;
   fill(x, func:number(number index) { return index+1; }, 10000);
   // 0 is the starting value, we do this so that we can return 0
   //if the array is  empty
   //Here we are filling the array with the values 0 - 10000 for
   //the use in the suming function
   println(reduce(x, sum, 0));
}
```

3.8.3 Matrix Multiplication

In this example we introduce nested functions and explore how working with functions can create clearer and better code. We'll also take a look at the feature that gives DotPar its name: implicit parallelism.

```
func matrixMultiply:number[][](number[][] a, number[][] b) {
    func transpose: number[][](number[][] a) {
        number[len(a[0])][len(a)] transposed;
        number i;
        number j;
        for (i<0; i < len(a); i = i + 1){
            for (j<0; j < len(a[0]); j = j+ 1){
                transposed[j][i] = a[i][j];
```

```
            }
        }
        return transposed;
    }

    func sum:number(number a, number b) {
          return a + b;
    }

    func dotprod:number(number[] a, number[] b) {
          return reduce([x * y for number x, number y in a, b], sum );
    }

    if (len(a) != len(b[0])) {
          return nil; // one can always return nil in place of a real value
    }

    return [[dotprod(row, col) for number[] col in transpose(b)] for number[] row in a];
}
```

The function `transpose` returns a transpose of the input matrix. The function `matrixMultiply` takes in arguments which are two two-dimensional arrays, `a` and `b`. The function `dotprod` accepts two one-dimensional arrays and calls the reduce function, which calls the sum function on each `x`, `y` pair which are elements in arrays `a` and `b`. The final line returns a statement that pairs every row in `a` with every column in `b` and calls the function `dotprod`.

3.8.4 Parallelization

These examples demonstrate the power of DotPar's implicit parallelization.

The `map` and `reduce` functions can be parallelized, given certain restrictions. If the functions only access variables passed to them and do not mutate outside state, the mappings and reductions can be automatically parallelized and executed on multiple cores. For instance, the computation done by `map` and `reduce` can be done in $O(n/m)$ time, where n is the size of the array and m the number of processors. In the transpose function from the previous section, each cell in the new matrix can be computed independently of the other, so DotPar will parallelize this multiplication if it is appropriate given the computer's architecture and the nature of the computation.

4 Conclusion

In this tutorial we have covered compiling your programs and writing basic assignment and logical statements all the way through to more complicated features like map, reduce, list comprehensions, and nested functions. With all of the these tools now available, you

should now be able to unleash the performance power that DotPar provides for data manipulation.

Part III
Language Reference Manual

DotPar is a flexible language that provides implicit nested data parallelism with powerful first-class functions while providing a familiar imperative interface. The focus is on parallel performance for arrays and loops, since these are often the source of performance bottlenecks.

Previous implementations of nested data-parallel languages have relied on the use of purely functional paradigms which can make the learning-curve steep and difficult to those who just want to pick up a new tool. DotPar provides a multi-paradigm nested data-parallel language with a friendly imperative style in addition to powerful functions. Its implicit parallelism focuses on the performance of arrays and loops, but its implementation requirements are flexible enough for other forms of parallelism as well.

5 Lexical Convention

A program consists of zero or more input statements, and one or more lines stored in files. These are translated in several phases, which are described in Section 13. The first phases do low-level lexical transformations, which reduce the program to a series of tokens.

lines:
    imports_opt external_declaration
    lines external_declaration
imports:
    imports import_declaration
    import_declaration

import_declarations will be explained in detail in Section 11.

6 Tokens

DotPar has 5 token classes: identifiers, keywords, literals, operators, and other separators. All white space, collectively including blanks, horizontal and vertical tabs, newlines, and comments as described below are ignored except as they separate certain tokens. Sometimes white space is required to separate otherwise adjacent identifiers, keywords, and literals.

6.1 Comments

The characters /* introduce a block comment, which terminates with the characters */. In addition, // adds line comments which converts everything that follows to a comment until the end of the line. Comments do not nest, and they do not occur within string or character literals.

6.2 Constants

Constants are character, number, or string literals, as well as true, false, or nil.
constant

- CHAR_LITERAL
- NUM_LITERAL
- STRING_LITERAL
- TRUE
- FALSE
- NIL

6.2.1 Character Literal

A character constant specifies a single ascii character surrounded by single quotes, such as 'a'. If one is so inclined, he or she can include such control characters as newlines by using traditional backslash delimited escape codes. For instance, a newline is '\n', and '\0' is the null character. A notable exception is the backslash itself, which is merely '\'.

A char literal has the type char and is initialized with the given ascii character. The behavior of a program that attempts to alter a char literal is undefined.

6.2.2 Number Literal

There are two ways to represent literal numbers with DotPar: the first is an integral format, without a decimal point, and the second is a floating point representation, including a decimal point.

The integral representation is merely a series of digits without spaces between them, like 31 or 42. Integral number literals are only available for base 10.

The floating point representation is a series of digits with a period embedded within or prepended to the front, and without spaces anywhere. Examples include 3.14159, or .11235, but not 125. with a trailing period.

The floating point representation is restricted to base 10, and there is no support for

scientific notation.

A number literal has a type number and is initialized with the given value: if the given literal is too large, the behavior of the program will be undefined.

6.2.3 String Constant

A string literal is a sequence of ascii characters surrounded by double quotes as in ". . . ". A string has type 'array of characters' and is initialized with the given characters. One may include traditional backslash escape codes, similar to character constants explained above. The behavior of a program that attempts to alter a string literal is undefined.

6.2.4 Boolean Constant

There are only two values for a boolean constant, `true` or `false`. Their literal value is respectively represented as `true` and `false`, not decorated by any special characters.

6.2.5 Nil Constant

`nil` is a special literal that can stand in for any other type of value. `nil` evaluates to `false`. `nil` and `false` are the only values in the language that evaluate to `false`.

6.3 Identifiers

An identifier is a sequence of letters and digits, and underscores. The first character must be a letter. Identifiers are case-sensitive and may have any length.

6.4 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

- `import`
- `true`
- `false`
- `boolean`
- `char`
- `func`
- `number`
- `void`
- `if`

- `else`
- `elif`
- `for`
- `in`
- `return`

7 Syntax Notation

The syntax in this manual has categories written in italics and literals and characters in plain font. Optional terminal and nonterminal symbols have the subscript "opt," like below:

{expression_opt}

*An non-terminal with the name foo_opt will either go to "the empty string" or foo. Similarly, lists follow the format of imports above. To avoid repetition, rules that follow this rigid structure will not be explained in-depth. However, they will be included in the complete grammar at the end of this reference manual.

8 Meaning of Identifiers

Identifiers are simply the names given to functions and variables. A variable is used to store data and points to a specific location. The interpretation of the data is based on the variable's type. Identifiers are limited to the scope within which they are defined and are only accessible within that scope. This can be only within a specific function or an entire program.

8.1 Basic Types

DotPar contains three basic types: number, boolean and char. Variables of type number are 64-bit double precision floating point numbers. Boolean variables have only two possible values: true and false. They are internally treated as a 32-bit entities. Char variables can store any member of the set of characters. Every character has its value as equivalent to the integer code for the character.

8.2 Derived Types

These include arrays and functions which can be constructed from the basic types. Derived types include:

- *arrays* of elements of a given type
- *functions* accepting variables of certain types, and returning variables of a given type

There are infinitely many derived types. Type declaration syntax are described in more detail below.

9 Objects and lvalues

A variable is a named region of storage and an lvalue is an expression referring to that variable. For example, an lvalue expression can be an identifier with a specified type. Like in C, the term "lvalue" originates from the term left-value which indicates that it was on the left side of the assignment operator. Each operator listed below can expect lvalue operands and yields an lvalue.

10 Expressions

The precedence of expression operators is the same as the order of the major subsections outlined below in this section, with highest precedence first. In all subsections, precedence follows C-style conventions. Left- or right-associativity is specified in each subsection for the operators discussed therein. The grammar given in Section 13 incorporates the precedence and associativity of the operators.

The precedence and associativity of the operators is fully specified, but the order of evaluation of expressions is undefined, even if the subexpressions involve side effects. However, each operator combines the values it produces by its operands in a way that is compatible with the parsing of the containing expression in which it appears. The handling of overflow, divide check, and other exceptions in expression evaluation is undefined by the language spec.

10.1 Primary Expressions

Primary expressions are identifiers, constants, string literals, or expressions in parentheses.

primary_expression:
    IDENTIFIER
    constant
    (expression)

An identifier is a primary expression. Its type is specified by its declaration, which must occur earlier in the program.

A constant is a primary expression. Its type depends on its form as discussed in section 2.2

A string literal is a primary expression. It is actually converted to an array of characters, but string literals are included as part of the grammar for programmer convenience.

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression.

10.2 Postfix Expressions

The operators in postfix expressions group left to right.

postfix_expression:
primary expression
postfix_expression [expression]
postfix_expression (argument_expression_list_pt)

10.2.1 Array References

A postfix expression followed by an expression in square brackets is a postfix expression denoting a subscripted array reference. The postfix expression must evaluate to an array and the expression must evaluate to a type number.

10.2.2 Function call

A postfix expression followed by the function designator, followed by parentheses containing a possibly empty, comma-separated list of assignment expressions, which constitute the arguments to the given function.

The term argument is used for an expression passed by a function call; the term parameter is used for an input object (or its identifier) received by a function definition, or described in function declaration. The scope is lexical.

Arguments are passed by value. A function may change the values of its parameter objects, which are copies of the argument expressions, but these changes cannot affect the values of the arguments.

However, array references are passed by value, so a function may modify the contents of an array passed to it.

Type agreement is strict between the parameters and arguments. Thus, DotPar is strongly and statically typed.

The order of evaluation of arguments is unspecified. In addition, the arguments and their side effects need not be fully evaluated before the function is entered if the compiler can guarantee that this does not affect the correctness of the program.

10.3 Unary Expressions

Expressions with unary operators group right-to-left.

unary_experession:
postfix_expression
NOT unary_expression
SUB unary_expression

The NOT unary_expresssion refers to the ! operator. The operand of the ! operator must have a boolean type, and the result is true if the value of its operand evaluates to false, and false otherwise. The type of the result is boolean.

The SUB unary_expression refers to the - operator. The operand of the - operator must have a number type, and the result is the negation of the value of its operand. The type of the result is number.

10.4 Arithmetic Expressions

Arithmetic Expressions are grouped left to right, and include: %, /, *, + , - but this is infact is not useful, as it is parsed to preserve C style precedence. All operators return number types.

arithmetic_expression:
unary_expression
arithmetic_expression REM arithmetic_expression
arithmetic_expression DIV arithmetic_expression
arithmetic_expression MULT arithmetic_expression
arithmetic_expression ADD arithmetic_expression
arithmetic_expression SUB arithmetic_expression

REM refers to the % operator, the remainder which refers to the remaining value after quotient the first operand by the second. Note that the remainder of a negative value will be negative.

DIV refers to the / operator, the division which refers to quotient of the first operand and the second.

MULT refers to the * operator, the multiplication between its two operands.

ADD refers to the + operator, the addition between its two operands.

SUB refers to the binary - operator, the subtraction between its two operands.

10.5 Relational Expressions

The relational operators group left-to-right.

relational_expression:
arithmetic_expression
relational_expression GEQ relational_expression
relational_expression GT relational_expression
relational_expression LT relational_expression
relational_expression LEQ relational_expression
relational_expression EQ relational_expression
relational_expression NEQ relational_expression

The GEQ refers to >= operators(greater or equal), GT to >(greater), LT to <(less than), LEQ to <= (less than or equal), EQ to == (equality), and NEQ to != (not equality). All yield false if the specified relation is false, and true if it is true. The type of the result is boolean. The usual arithmetic conversions are performed on arithmetic operands. GEQ, GT, LEQ, and LT have higher precedence than EQ and NEQ.

10.6 Conditional Expressions

The conditional operators group left-to-right and include && and ||.

conditional_expression:
relational_expression
conditional_expression OR conditional_expression
conditional_expression AND conditional_expression

OR refers to the ||operator which performs a logical OR on its two operands which must be of type boolean. The result is of type boolean.

AND refers to the && operator which performs a logical AND on its two operands which must be of type boolean. The result is of type boolean.

10.7 Array Expressions

Array Expressions are conditional statements, or list_comprehnesionsand initializer_list_opt enclosed in square brackets '[' ,']'.

array_expression:
conditional_expression
[list_comprehension]
[initializer_list_opt]

An initializer_list is used in the creation of array literals.

10.8 List Comprehension

*List comprehensions are used to succinctly create an array. They are equivalent in power to maps and filters, but are often a more convenient syntax to use. Although the list comprehension may be parallelized during its execution, the ordering of the resulting array is deterministic.

list_comprehension:
array_expression FOR paren_parameter_list_opt IN array_expression if_comp_opt
array_expression FOR paren_parameter_list_opt IN paren_multi_array_expression_list_opt if_comp_opt

The list comprehension syntax and behavior is very similar to Python's.

10.9 Assignment Expressions

There is one assignment operator =, and it is not used more than once per statement.

assignment_expression:
array_expression
anonymous_function_definition
postfix_expression ASSIGN array_expression
postfix_expression ASSIGN function_definition
postfix_expression ASSIGN anonymous_function_definition

The assignment operator requires an lvalue as a left operand, and the lvalue must be mutable.

The assignment operator assigns array_expression, function_definition and anonymous_function_definition to postfix_expressions.

10.10 Anonymous Function Definitions

Anonymous function definitions can be used in assignments. They are identical to regular function definitions, except they lack an identifier.

anonymous_function_definition:
FUNC: type_specifier (parameter_list_opt) compound_statement

11 External Declarations

External declarations form the basic building blocks of lines of code.

external_declaration:
function_definition
declaration

The two top-level declarations are function_definitions and declarations. Function definitions are defined as:

function_definition:
FUNC IDENTIFIER: type_specifier (parameter_list_opt) compound_statement) compound_statement

Unlike C, functions the return type of a function is defined after the name. With potentially complicated return types, having the type come after the name makes reading the code easier. They also have the `func` keyword so that function syntax for types, definitions, and assignments.

Declarations inform the interpretation given to each identifier. Implementations decide when to reserve storage space associated with the identifier. Declarations have the form

declaration:
type_specifier declarator;
type_specifier declarator ASSIGN initializer;

where ASSIGN is =. A declaration must have one and only one declarator.

11.1 Type Specifiers

A type specifier is either a basic or derived type, examples of which are:

- number
- char[]
- func:number[](char[])

* At most one type-specifier may be given in a declaration. Type specifiers have the form:

type_specifier:
type_specifier [arithmetic_expression]
type_specifier [ ]
basic_type
VOID
func_specifier

func_specifier:
func: type_specifier (type_list)
func: type_specifier (parameter_list_opt)

basic_type:
NUMBER
CHAR
BOOLEAN

11.2 Declarators

Declarators have the syntax:

declarator:
IDENTIFIER
(declarator)

11.3 Initializers

* *initializer:*
array_expression
anonymous_function_definition

As some example types, we can see that array types include number[] and number[][]. For example array can be declared initialized as: number[] a = [1, 2];

An example function declaration is func:number[](number){ } foo. This function returns an array of numbers, and accepts a number as an argument, and is named foo. This syntax allows us to concisely declare variables as functions, which is important for a language with first-class functions.

Note that we could have given a name to the number parameter if we chose to using a parameter_list, which is a series of parameter_declarations:

parameter_declaration:
type_specifier declarator

Although this identifier would not be used, it may be desirable to name parameter the so that the declaration is self-documenting.

12 Statements

Statements are executed sequentially and have no value. The types of statements in DotPar are listed below:

statement:
    expression_statement
    compound_statement
    selection_statement
    iteration_statement
    jump_statement

12.1 Expression Statements

Most statements are expressions of the form

expression-statement:
    expression_opt;

These statements are mostly assignments and function calls. The side effects of an expression need not be computed before the execution of the following statement if the compiler can guarantee that this does not affect correctness. This is different from other languages like C.

12.2 Compound Statements

Compound statements are multiple statements executed when one is expected. For example, the body of a function is a compound statement. In DotPar, if statements and for loops also require compound statements, even if they have single line bodies.

compound_statement:
    statement_list_opt
statement_list :
    statement_list_opt statement
    statement_list_opt declaration
    statement_list_opt function_definition

Identifiers can only be declared once within the scope of a function and cannot be the same as the identifiers passed into a function.

12.3 Selection Statements

Selection statements allow for the choice between multiple control flows.

selection_statement:
    if elifs_opt else_opt
if:
    IF '(' expression ')' compound_statement
else_opt:
    else
else:
    ELSE compound_statement
elifs_opt:
    elifs
elifs:
    ELIF '(' expression ')' compound_statement
    elifs ELIF '(' expression ')' compound_statement

Note that `IF`, `ELSE`, and `ELIF`, are used to distinguish the terminal if, else, and elif keywords from the if, else, and elif non-terminals.

The expression within the if statement must evaluate to either true or false. When it is true the substatement is executed. With an if else statement the else substatement is executed when the expression evaluates to false. Every else is connected with the first if statement above it that is unconnected to an else.

12.4 Iteration Statements

Iteration statements specify looping.

iteration-statement:
    for (expression_opt; expression_opt; expression_opt) compound_statement

The first expression is evaluated once and marks the start of the loop. Note that a new variable cannot be declared as part of this operation. The second expression is coerced to a boolean; if false, it terminates the loop. The third expression is evaluated after each iteration so it specifies the re-initialization for the loop.

Again, note that curly braces must be used.

12.5 Jump Statements

Jump statements, once they are reached, always transfer control regardless of any condition.

jump_statement:
RETURN expression_opt ';'

A function provides the value back to its caller via the return statement. It returns the value of the expression when it is evaluated. The expression once evaluated is interpreted as the type specified in the function declaration.

13 Scope

A program can be compiled from several .par files. Scoping is lexical. Thus, identifiers in DotPar have one top-level namespace in which variables and functions are defined, with a shared namespace between the two. Imports made in a file are accessible anywhere else in that file. An identifier declared outside of any function can be accessed anywhere in the program. An identifier declared in a block is available anywhere within the block, including inner functions. Note that this means that DotPar has closures. That is, if an inner function is returned from a function, it maintains access to the variables of the outer function.

14 Preprocessing

A preprocessor performs the inclusion of named files. Lines beginning with import communicate with the preprocessor. The syntax of these lines is independent of the rest of the language; they must only appear at the beginning of the file.

14.1 File Inclusion

Imports have the syntax:

imports:
imports import_declaration
import_declaration

import_declarations:
IMPORT IDENTIFIER

A control line of the form
import module;
means this line will be replaced by the contents of the file named filename, with extension .par. The named file is searched for in the current directory, and each file is imported only once per program. Import statements can be nested, so every file can include them.

15 Built-in Functions

The language includes several built-in functions that provide basic building blocks for more complex user-defined functions. These appear to the user as regular library functions. They can be shadowed by user functions to avoid having an unreasonable amount of reserved words. Many of these functions are self-explanatory and, as such, their formal prototypes are not given. Further explanation is present where warranted.

Finally, note that many useful functions, such as string containment, are not included. These are more appropriate as libraries than language features.

15.1 Array

```
cat(arr, other\_arr) // concatenate two arrays and return the result
each(arr, fn(element, index)) // iterate through an array
fill(dimensions, fn(index)) // fill an array using a function. dimensions
// is an array that has the size of each dimension in the output array
filter(arr, fn(element, index)) // selects certain elements from an
// array; the filter function returns a boolean
len(arr) // array length
map(arr, fn(element, index)) //runs function fn on each element of the array
reduce(sum, nums, 0) // reduce operation. The last argument is the
// initial value.
zip(arr, other\_arr) // combine two arrays into a nested array
```

15.2 String

```
len(str) // string length
```

15.3 Math

Note that all trigonometric functions operate with radians.

```
acos(n)
asin(n)
atan(n)
cos(n)
exp(num, exponent)
ln(n)
log(n, base)
sin(n)
sqrt(n)
tan(n)
ceil(n)
floor(n)
trunc(n)
```

```
round(n)
rand(n)
```

15.4 I\O

```
print(s)
println(s)
printerr(s)
read()
readln()
```

16 Future Goals

We have an ambitious set of possible additions to the language. We do not expect to complete all of them, but we do hope to add at least a few of these features. Also, note that some of these goals are mutually exclusive. For instance, if Java interoperability is implemented, adding some of the library functionality mentioned below will not be necessary. Note that these additions are in no particular order.

1. Add more assignment operators: *=, \=, +=, -=, %=
2. Do basic type inference
3. Add new control flow tools: foreach loops, while, break, and continue
4. Add new container objects, such as a struct or dictionary
5. Introduce shorthand syntax for operations like array concatenation (++)
6. Range selection for arrays and strings, e.g. arr[i:j] or arr[i:j:stride]
7. Java interoperability (this will be hard given static analysis for parallelization)
8. Namespacing
9. Immutable values
10. Library-level functionality
    - String manipulation: trim, contains, split
    - Regexes
    - Arrays: populate (like fill, but for existing arrays), max, min, push, pop
    - Basic data structures
    - Time
    - Char to int function
11. Mutating versions of some functions, such as map! and zip!

12. Function keywords and/or optional params

13. Implement more parallelized aspects in the language

17 Grammar

Note that for the grammar we specify the following precedence following YACC conventions:
\%left OR
\%left AND

\%left EQ NEQ
\%left GT GEQ LT LEQ

\%left ADD SUB
\%left MULT DIV REM

\%right UMINUS

These are the rules for the grammar:

lines	::=	imports_opt external_declaration
lines	::=	lines external_declaration
imports_opt	::=	imports
imports_opt	::=	
imports	::=	imports import_declaration
imports	::=	import_declaration
import_declaration	::=	IMPORT IDENTIFIER “**;**”
constant	::=	CHAR_LITERAL
constant	::=	NUM_LITERAL
constant	::=	STRING_LITERAL
constant	::=	TRUE
constant	::=	FALSE
constant	::=	NIL
argument_expression_list_opt	::=	argument_expression_list
argument_expression_list_opt	::=	
argument_expression_list	::=	assignment_expression
argument_expression_list	::=	argument_expression_list “**,**” assignment_expression
postfix_expression	::=	primary_expression
postfix_expression	::=	postfix_expression “[” expression “]”
postfix_expression	::=	postfix_expression “**(**” argument_expression_list_opt “**)**”
unary_expression	::=	postfix_expression
unary_expression	::=	NOT unary_expression

unary_expression ::= SUB unary_expression
arithmetic_expression ::= unary_expression
arithmetic_expression ::= arithmetic_expression REM arithmetic_expression
arithmetic_expression ::= arithmetic_expression DIV arithmetic_expression
arithmetic_expression ::= arithmetic_expression MULT arithmetic_expression
arithmetic_expression ::= arithmetic_expression ADD arithmetic_expression
arithmetic_expression ::= arithmetic_expression SUB arithmetic_expression
relational_expression ::= arithmetic_expression
relational_expression ::= relational_expression GEQ relational_expression
relational_expression ::= relational_expression GT relational_expression
relational_expression ::= relational_expression LT relational_expression
relational_expression ::= relational_expression LEQ relational_expression
relational_expression ::= relational_expression EQ relational_expression
relational_expression ::= relational_expression NEQ relational_expression
conditional_expression ::= relational_expression
conditional_expression ::= conditional_expression OR conditional_expression
conditional_expression ::= conditional_expression AND conditional_expression
opt_paren_multi_array_expression_list ::= "(" multi_array_expression_list ")"
opt_paren_multi_array_expression_list ::= multi_array_expression_list
multi_array_expression_list ::= array_expression "," array_expression
multi_array_expression_list ::= array_expression "," array_expression "," array_expression_list
array_expression_list ::= array_expression
array_expression_list ::= array_expression_list "," array_expression
array_expression ::= conditional_expression
array_expression ::= "[" list_comprehension "]"
array_expression ::= "[" initializer_list_opt "]"
if_comp_opt ::= if_comp
if_comp_opt ::=
if_comp ::= IF expression

list_comprehension	::=	array_expression FOR paren_parameter_list_opt IN array_expression if_comp_opt
list_comprehension	::=	array_expression FOR paren_parameter_list_opt IN opt_paren_multi_array_expression_list if_comp_opt
assignment_expression	::=	array_expression
assignment_expression	::=	anonymous_function_definition
assignment_expression	::=	postfix_expression ASSIGN array_expression
assignment_expression	::=	postfix_expression ASSIGN function_definition
assignment_expression	::=	postfix_expression ASSIGN anonymous_function_definition
expression	::=	assignment_expression
primary_expression	::=	IDENTIFIER
primary_expression	::=	constant
primary_expression	::=	"(" expression ")"
type_specifier	::=	type_specifier "[" arithmetic_expression "]"
type_specifier	::=	type_specifier "[" "]"
type_specifier	::=	basic_type
type_specifier	::=	VOID
type_specifier	::=	func_specifier
func_specifier	::=	FUNC "**:**" type_specifier "(" type_list ")"
func_specifier	::=	FUNC "**:**" type_specifier "(" parameter_list_opt ")"
basic_type	::=	NUMBER
basic_type	::=	CHAR
basic_type	::=	BOOLEAN
declaration	::=	type_specifier declarator "**;**"
declaration	::=	type_specifier declarator ASSIGN initializer "**;**"
declarator	::=	IDENTIFIER
declarator	::=	"(" declarator ")"
type_list	::=	type_specifier
type_list	::=	type_list "**,**" type_specifier
parameter_list_opt	::=	parameter_list
parameter_list_opt	::=	
paren_parameter_list_opt	::=	"(" parameter_list ")"
paren_parameter_list_opt	::=	parameter_list
parameter_list	::=	parameter_declaration

parameter_list	::=	parameter_list “**,**” parameter_declaration
parameter_declaration	::=	type_specifier declarator
initializer	::=	array_expression
initializer	::=	anonymous_function_definition
initializer_list_opt	::=	initializer_list
initializer_list_opt	::=	
initializer_list	::=	initializer
initializer_list	::=	initializer_list “**,**” initializer
expression_statement	::=	expression_opt “**;**”
expression_opt	::=	expression
expression_opt	::=	
compound_statement	::=	“{” statement_list_opt “}”
statement_list_opt	::=	statement_list
statement_list_opt	::=	
statement_list	::=	statement_list_opt statement
statement_list	::=	statement_list_opt declaration
statement_list	::=	statement_list_opt function_definition
selection_statement	::=	if elifs_opt else_opt
if	::=	IF “(” expression “)” compound_statement
else_opt	::=	else
else_opt	::=	
else	::=	ELSE compound_statement
elifs_opt	::=	elifs
elifs_opt	::=	
elifs	::=	ELIF “(” expression “)” compound_statement
elifs	::=	elifs ELIF “(” expression “)” compound_statement
iteration_statement	::=	FOR “(” expression_opt “**;**” expression_opt “**;**” expression_opt “)” compound_statement
jump_statement	::=	RETURN expression_opt “**;**”
statement	::=	expression_statement
statement	::=	compound_statement
statement	::=	selection_statement
statement	::=	iteration_statement
statement	::=	jump_statement
anonymous_function_definition	::=	FUNC “**:**” type_specifier “(” parameter_list_opt “)” compound_statement
function_definition	::=	FUNC IDENTIFIER “**:**” type_specifier “(” parameter_list_opt “)” compound_statement

external_declaration	::=	function_definition
external_declaration	::=	declaration

Part IV
Project Plan

Project Manager - Logan

- State what process used was used to develop the language and its translator.
- State the roles and responsibilities of each team member.
- Include the implementation style sheet used by the team.
- Show the timeline of what was done and when.
- Include your project log.

Part V
Language Evolution

Language Guru - Sid

- Describe how the language evolved during the implementation and what steps were used to try to maintain the good attributes of the original language proposal.
- Describe the compiler tools used to create the compiler components.
- Describe what unusual libraries are used in the compiler.
- Describe what steps were taken to keep the LRM and the compiler consistent.

Part VI
Translator Architecture

System Architect - Justin

- Show the architectural block diagram of translator.
- Describe the interfaces between the modules.
- State which modules were written by which team members.

Part VII
Development and Run-time Environment

System Integrator - Nathan

- Describe the software development environment used to create the compiler.
- Show the makefile used to create and test the compiler during development.
- Describe the run-time environment for the compiler.

Part VIII
Test Plan

System Tester - Andrew

- Describe the test methodology used during the development.
- Show programs used to test your translator.

Part IX
Conclusions

Whole team

- lessons learned as a team
- lessons for future teams
- suggestions for the instructor on what topics to keep, drop or add in future courses

Individually

- Lessons learned individually

Part X
Appendix

Include a listing of the complete source code with identification of who wrote which module of the compiler. This listing does not have to be included in the paper copy of the final report.