The Dijkstra Family of Languages

Language Specification

Version 2.0

Copyright © Gary Pollice, 2012 Worcester Polytechnic Institute

Revision History

Version #	Comments	Date
1.0	Initial specification of the three languages: Base Dijkstra, Dijkstra with Functions, and Object Dijkstra	8/25/2012
1.0.1	Add Toy Dijkstra—a simple subset of Base Dijkstra that we use in class for discussion and demonstration. This is in an appendix.	8/26/2012
1.0.2	Fix errors reported by class members.	8/26/2012
1.0.3	Fix errors and make clarification based upon comments by Billy Hawkins.	8/30/2012
1.0.4	Fix grammars to show proper relational expressions.	9/10/2012
1.0.5	Correct problem with LogicalAndExpression rule.	9/15/2012
1.0.6	Modify the grammars so that they read more clearly. This makes them cleaner CFGs but they are no longer right recursive. Students will need to factor the grammar to get it to a form that is usable with ANTLR. Also add corrections and clarifications as submitted by students. Cbars to the page.	9/22/2012
2.0	Minor revisions. Starting point for the Spring 2015 courses.	1/14/2015

Table of Contents

Introduction

This document contains specifications for three versions of the Dijkstra programming language. Base Dijkstra is the minimal language that students must implement to receive undergraduate credit—a grade of C. Dijkstra with Functions adds a procedure abstraction to the language. Object Dijkstra adds an object type system to the Dijkstra with Functions. Each specification consists of two parts. The first part is the syntactic definition and the second part is the semantic definition.

The specifications for each of the advanced languages adds to the base specification. The material from "lower level" languages is not repeated unless it is necessary to aid comprehension. However, the full CFG for each of the languages is included.

The Base Dijkstra Language

This page contains information about the Dijkstra Base language, taken from Edsger W. Dijkstra's book, *A Discipline of Programming* (1976). This book is on reserve at the Gordon Library.

Syntax

The following CFG describes the syntax of Base Dijkstra The base language contains a minimal set of operations and statement types, but is sufficient for implementing several algorithms quite efficiently. Lexical structure is not specified in this grammar. Students are expected to create the appropriate lexical grammar as part of their work. Some lexical hints are given, however.

The typographical conventions used are as follows:

- `Monaco 11pt.` is used to represent syntactic elements such as non-terminals.
- ***`Bold Italicized Monaco 11pt.`*** is used to represent terminal symbols such as reserved words, numbers, characters, etc.
- Operators and separators are shown in **`Bold`**, but not italicized since they do not look natural when italicized. If necessary the operator will be surounded by single quotes if it might be confused with the grammar's metacharacters.
- Lexical non-terminals are all uppercased as in `LETTER`.
- Ɛ is the empty symbol.

Base Dijkstra CFG

```
Program:
          program ID DeclarationOrStatementList EOF

DeclarationOrStatementList:
          DeclarationOrStatement
          | DeclarationOrStatementList DeclarationOrStatement

DeclarationOrStatment:
          Declaration | Statement

Declaration:
          Type IDList Separator

Type:
          float | int | boolean

Separator:
          Ɛ | ;
```

```
IDList:
          ID
          | IDList , ID

Statement:
          AssignStatement Separator
          | AlternativeStatement
          | IterativeStatement
          | InputStatement Separator
          | OutputStatement Separator
          | CompoundStatement

AssignStatment:
          IDList <- ExpressionList

AlternativeStatement:
          if GuardedStatementList fi

IterativeStatement:
          do GuardedStatementList od

InputStatement:
          input IDList

OutputStatement:
          print Expression

CompoundStatement:
          { DeclarationOrStatementList }

GuardedStatementList:
          Guard
          | GuardedStatementList Guard

Guard:
          Expression :: Statement

ExpressionList:
          Expression
          | ExpressionList , Expression

Expression:
          LogicalOrExpression

LogicalOrExpression:
          LogicalAndExpression
          | LogicalOrExpression '|' LogicalAndExpression
```

```
LogicalAndExpression:
          EqualityExpression
          | LogicalAndExpression & EqualityExpression

EqualityExpression:
          RelationalExpression
          | RelationalExpression EqualityOp EqualityExpression

EqualityOp:
          = | ~=

RelationalExpression:
          AdditiveExpression
          | AdditiveExpression RelationalOp AdditiveExpression

RelationalOp:
          < | > | <= | >=

AdditiveExpression:
          MultiplicativeExpression
          | AdditiveExpression AdditiveOp
          MultiplicativeExpression

AdditiveOp:
          + | -

MultiplicativeExpression:
          UnaryExpression
          | MultiplicativeExpression MultiplicativeOp
          UnaryExpression

MultiplicativeOp:
          * | / | mod | div

UnaryExpression:
          PrimaryExpression
          | ~ UnaryExpression
          | - UnaryExpression

PrimaryExpression:
          | INTEGER
          | FloatConstant
          | true
          | false
          | ID
          | ( Expression )
```

```
FloatConstant:
          INTEGER . INTEGER
```

Grammar notes (includes lexical information)

1. From a syntactic viewpoint, integers can be any length. The `INTEGER` lexical class represents only non-negative integers. The case of negative numbers is taken care of in the `UnaryExpression` rule.
2. Floating point constants are expressed in the syntactic, rather than the lexical grammar. Also, there is only one way to represent a floating point constant. Scientific notation is not accepted in this language.
3. Whitespace consists of spaces, new lines, horizontal tabs, and comments.
4. Comments are defined as in several scripting languages like Ruby and Perl. They begin with a '**#**' character and continue to the end of the line.
5. Notice that separators are optional and only valid in certain constructs. They are included for readability.
6. Identifiers (IDs) must begin with a letter and may include letters, digits, the underscore character ('_'). and question mark ('**?**').
7. There is no limit to the size of an identifier.
8. Identifiers may never have the same text as a reserved word in the grammar.

Reserved words

The following is a list of reserved words in Base Dijkstra:

```
boolean div do false fi float if input int mod od print
program true
```

Semantics

In general, anyone familiar with imperative programming languages can infer the meaning of Base Dijkstra programs and most of the language. There are a couple of significant features that differ from more common programming languages. This section explains the semantics of Base Dijkstra.

Assignment

Assignment statements are more than single assignments to variables. There must be as many identifiers to the left of the operator (**<-**) as there are expressions to the right of the operator. The semantics of the language require that *all of the assignments occur simultaneously*. That is, all expressions are evaluated from the state before the statement. This means that the following assignment statement effectively swaps the variables:

```
a, b <- b, a
```

If an identifier appears on both the left and right side of the operator, the value associated with the identifier before the assignment is used for every occurrence of the identifier on the right side and the resulting value of the identifier is the value of the expression on the right side that corresponds to the position of the identifier on the left.

If an identifier appears more than once on the left side of the assignment operator, the results are undefined. The final value of the variable corresponding to the identifier may legally be the value of any of the corresponding expressions on the right. The programmer must not assume any specific value for the variable. The compiler is not required, however, to make the runtime behavior random.

Nondeterminism:

Dijkstra argues for nondeterminism in a programming language. This is embodied in the alternative (if) and iterative (do) statements. Both of these statements contain one or more Guards. You cannot have empty bodies for either of these statements. Guards consist of two parts: a) a boolean expression that is a "guard" that gates the execution of b) a statement that executes if the guard is true. Notice that this is a single statement. If you want multiple statements to execute, you would make the statement a `CompoundStatment`.

The guards do not have to be mutually exclusive. For example, the following alternative statement is valid:

```
if
        x > 0 :: y <- 1
        x > 3 :: y <- 2
fi
```

If the statement is executed in a state where x has the value 5, both guards are true. In this case, either of the statements may execute and the programmer must not assume any deterministic behavior. The compiler is not required to make the run time behavior random.

Guarded statements (conditions)

The alternative and iterative statements contain guarded statements. In an alternative statement, *exactly one* of these statements must execute (selected by the above nondeterminism semantics if appropriate). If no guard expression evaluates to true then the program must abort with an error. This means a program with the following statement will fail at run time.

```
x <- 0;
if
        x < 0:: negative <- true
        x > 0:: negative <- false
fi
```

Iterative statements execute, applying any nondeterministic choice of guards, repeatedly until no guarded expression evaluates to true. For iterative statements, zero or one guarded statement executes each time through the loop.

Arithmetic

There are two arithmetic types: float and int. The operators **+**, **-**, and ***** are polymorphic. If the operands are mixed (one float and one int), the result type is float.

The operators **`div`** and **`mod`** may only be applied to int operands. These two operators are defined as follows:

For int values `x`, `y`, where `x = y * q + r`:

```
x div y = q
x mod y = r
```

The '`/`' operator is polymorphic, but the result is *always* a float. *If one or more operands are ints, they are converted to float before the division occurs.*

Assigning an int value (either computed expression or variable) to a float variable is legal. The value is converted to the underlying float representation.

Assigning a float value to an int variable is also legal. The float is *truncated* towards zero. This means, for example, that the float value 1.5 would become an int value of 1 and -1.5 would become -1.

It is appropriate, but not required, for the compiler to emit a warning message indicating a possible loss of information when truncation occurs.

When int operands are required, and a float is provided, this results in a compiler error. This means that the following statement should produce a compilation error:

```
x, y <- 42, 3.0
z <- x div y    # error here, int required
```

Arithmetic Values and Representation

Integers are represented as *signed long integers*. That is, they are the same as `long` integers in Java. All arithmetic involving integers behaves exactly like Java long integers. If input to the compiler contains an integer constant that specifies an integer value greater than what will fit in a Java long integer, *the results are undefined*. The compiler implementation is free to emit an error or warning. If the compiler completes compilation, however, the behavior of the generated code is undefined.

Floating point numbers are represented as Java *doubles*. All arithmetic involving floating point numbers behaves exactly like Java doubles. If input to the compiler contains an floating point constant that specifies a value greater than what will fit in a Java double, the results are undetermined. The compiler implementation is free to emit an error or warning. If the compiler completes compilation, however, the behavior of the generated code is undefined.

Type Inference

Variables must be defined before they are used. Variables may be defined in three ways:

1. As the result of a declaration statement such as:

```
int x
float y
boolean b
```

2. As a variable on the left hand side of an assignment such as:

```
a <- 1
```

3. As a variable in the list of an input statement such as:

```
input a, b
```

A variable that is used before being assigned a value (for example, on the right side of an assignment), will result in a *compilation warning*. It is not always possible to tell at compilation time whether the variable is used without performing sophisticate dataflow, and even then it may not be decidable. For example:

```
…
input x, y
do
   x < y :: print c
   x >= y :: c = 5
od
…
```

If the values 5, 3 are entered for x and y in the input statement then the program should run fine. However, if value of 3, 5 are entered, the output is not determined. The implementation is allowed to initialize values in any manner, but there is no requirement that a program where a variable is used before being initialized must produce consistent, or correct results.

Every variable has an associated type. Dijkstra infers the type from usage unless the variable is explicitly declared to have a type, as in #1 above. Variables need not be explicitly declared unless the compiler cannot infer the type. In the following program that computes the greatest common divisor using Euclid's algorithm, the variables, `x` and `y` can be inferred to be int variables:

```
program euclid
   input x, y;
   do
         x ~= y::
               if
                     x > y :: x <- x – y
                     x < y :: y <- y – x
               fi
   od
   print x
```

The type of a variable is determined as follows. Some statements, such as input and print provide no information for type inference. Types are inferred from expressions and the appropriate operators, as well as assignment statements. The inference rules are:

1. In the absence of other information, an operand of an arithmetic expression is assumed to be of type int where either a float or int applies, except for the `/` operator. So, in the statement `x <- a * b`, all variables are inferred to be int. In the statement `x <- a / b`, all variables are inferred to be float.

2. If a variable has been inferred to be float, and is the operand of an arithmetic expression where either int or float applies, there is no conflict and the already inferred type applies (that is, float). So, if the variable `a` has already been determined to be `float` in the `x <- a * b`, there is no conflict. All will be inferred to be float.
3. Type inference is performed from the lexical beginning of the program to the lexical end of the program. The following will cause a type inference error:

```
x <- a / b
y <- x mod c
```

   The error occurs because `x` is inferred to be float by the first statement. The following, however, does not cause a type inference error:

```
y <- x mod c
y <- a / b
z <- y div b
```

   Even though y would be inferred to be float by the second statement, the first statement sets the type of y to int. Therefore, the third statement is valid since y is an int. The real value of `a / b` is truncated in the second statement.

4. Types can be inferred from several types of statements and expressions in a program. For example, consider the following code:

```
a <- 1
input b
input c
if
   a = b :: print a
   c :: print b
fi
```

   The type of `b` can be inferred to be int because the comparison of an int with `b` implies that `b` is an int. Likewise, since the guard must be a boolean expression, then `c` must be boolean.

If there is a type conflict that cannot be resolved by other semantic rules, the program is invalid. The following code must not compile due to type conflict.

```
x <- 1
x <- true
```

In this case, the type of `x` is found to be int in the first statement. The second statement implies that `x` is a boolean. Both cannot be true and the program is invalid.

Another example of an invalid program is the following:

```
input a, b
if
      a ~= b :: print 1
      a = b :: print 0
```

```
fi
```

In this case, there is no way to infer the types of a and b and they must be declared explicitly.

Lexical Scope

Variables are defined within a scope. Dijkstra names are lexically scoped. That is, when a variable is defined, explicitly or implicitly, it exists only within the current lexical scope. For Base Dijkstra there are two primary scopes:

1. Program scope. Any variable defined in the program, but not within any compound statement exists throughout the program unless it is hidden by a variable of the same name in an inner scope.
2. Block scope. Any variable defined in a compound statement exists only within the block in which it is defined unless it is hidden by a variable of the same name that is declared in an inner compound statement.

There may never be two variable of the same name defined in a single scope.

When a variable is declared in an inner scope with the same name as a variable in an outer scope it hides the variables in all enclosing scopes with the same name. This can lead to a strange situation as shown in the following example.

Example 1:

```
program example1
   a <- 1
   {
         a <- 2        # outer a
         int a
         a <- a + 1    # inner a = 3
         a <- 5        # inner a
   }
```

If there were other references to a further in the block, they would all refer to the inner a.

Relational Expressions

Relations can only be performed on compatible operands. That is any of the binary operators, (= ~= < <= > >=) *must be performed on that are compatible*. For relational operators, except for the equality operators (**= ~=**) the operands must be numeric (float or int). If there are mixed operands in the expression, that is one float and one int, the int is converted to float before the operation is performed. So, the following is legal:

```
a, b <- 5, 5.1
c <- a < b            # boolean c is true
```

For the equality operators (= and ~=) the operands must be of identical types. The following would be illegal:

```
a, b <- 5, 5.1
```

```
c <- a ~= b
```

Logical Expressions

Logical expressions are evaluated in a short circuit manner. Consider

```
e1 | e2
```

and `e1` evaluates to `true`, the expression `e2` is never evaluated. Similarly, if you have `e1 & e2`, `e2` will not be evaluated if `e1` evaluates to false.

Pragmatics

Compiling to the JVM

Given a source file that begins:

```
program MyProgram
   …
```

this will compile to a Java class called MyProgram. There will be a MyProgram.class file generated. All of the code contained in the program will be in the `main()` function of that class. This is the only function that will be in the class.

*All classes compiled by the Dijkstra compiler will be placed in a **djkcode** package, unless overridden by the compiler's command line or the specific version of the Dijkstra language (i.e., Object Dijkstra).*

Running the compiler

Students will provide a mechanism for entering a command line into a shell (e.g. bash, sh, or a .bat file on Windows. The specifications for the command line invocation of the compiler follows. Not all of the options must be implemented, but you will find it helpful.

NAME: dijkstra

Dijkstra compiler

SYNOPSIS:

dijkstra [options] sourcefile

PARAMETERS:

options	command line options
sourcefile	the path to the source file to be compiled, default extension of .djk

DESCRIPTION:

The dijkstra tool executes the Dijkstra compiler to produce a Dijkstra class file and/or other artifacts produced during compilation.

OPTIONS:

-cp classpath
: If this option is present, the classpath specified is added to the default classpath for compilation. The classpath is specified using standard Java conventions for classpaths.

  The default classpath is the directory containing the source file and the directory specified by the DIJKSTRA_HOME environment variable.

-h Produces a help display with a synopsis of how to run the tool.

-p package
: This option declares the package in which the code will be placed. If this option is missing, then the code is placed in the default **djkcode** package.

-s Output a printable version of the symbol table after the compilation completes. This will be in a file with the same name as the input file with a .symtab extension.

-t{1,2} Output the parse tree. –t1 displays the tree output by the parser to a file with the same name as the input file with a .t1 extension. –t2 displays the final tree before code generation with the same name as the input file with a .t2 extension.

Dijkstra with Functions

Dijkstra with Functions is an incremental change to Base Dijkstra. There are two main features of this language.

1. The language now supports arrays of the base scalar types of *int, float,* and *boolean*. These arrays are homogeneous one-dimensional and cannot have elements that are themselves arrays, which would make them effectively multi-dimensional.
2. The language supports procedures and functions. Functions are simply procedures that return results. Unlike many mainstream programming languages, Dijkstra with Functions allows multiple values to be returned from a function.

Syntax

The full CFG is presented here. Where there are additions or changes, they are highlighted with maroon text and underlined. The same typographical conventions of the previous grammar are maintained.

Dijkstra with Functions CFG

```
Program:
          program ID DeclarationOrStatementList EOF

DeclarationOrStatementList:
          DeclarationOrStatement
          | DeclarationOrStatementList DeclarationOrStatement

DeclarationOrStatment:
          Declaration | Statement

Declaration:
          VariableDeclaration
          | ArrayDeclaration
          | ProcedureDeclaration
          | FunctionDeclaration

VariableDeclaration:
          Type IDList Separator

ArrayDeclaration:
          Type [ Expression ] IDList Separator

Type:
          float | int | boolean
```

```
Separator:
            ε | ;

IDList:
            ID
            | IDList , ID

ProcedureDeclaration:
            proc ID ( ) CompoundStatement
            | proc ID ( ParameterList ) CompoundStatement

ParameterList:
            Parameter
            | ParameterList , Parameter

Parameter:
            ID
            | Type ID

FunctionDeclaration:
            fun ID ( ) : TypeList CompoundStatement
            | fun ID ( ParameterList ) : TypeList
            CompoundStatement

TypeList:
            Type
            | TypeList , Type

Statement:
            AssignStatement Separator
            | AlternativeStatement
            | IterativeStatement
            | InputStatement Separator
            | OutputStatement Separator
            | CompoundStatement
            | ReturnStatment Separator
            | ProcedureCall Separator

AssignStatment:
            VarList <- ExpressionList

VarList:
            Var
            | VarList , Var

Var:
            ID | ArrayAccessor
```

```
AlternativeStatement:
          if GuardedStatementList fi

IterativeStatement:
          do GuardedStatementList od

InputStatement:
          input IDList

OutputStatement:
          print Expression

CompoundStatement:
          { CompoundBody }

CompoundBody:
          CompoundDeclOrStatement
          | CompoundDeclOrStatement CompoundBody

CompoundDeclOrStatement:
          VariableDeclaration
          | ArrayDeclaration
          | Statement

GuardedStatementList:
          Guard
          | GuardedStatementList Guard

Guard:
          Expression :: Statement

ExpressionList:
          Expression
          | ExpressionList , Expression

ReturnStatement:
          return
          | return ExpressionList

ProcedureCall:
          ID ( )
          | ID ( ArgList )

ArgList:
          Argument
          | ArgList , Argument
```

```
Argument:
          Expression

Expression:
          LogicalOrExpression

LogicalOrExpression:
          LogicalAndExpression
          | LogicalOrExpression '|' LogicalAndExpression

LogicalAndExpression:
          EqualityExpression
          | LogicalAndExpression & EqualityExpression

EqualityExpression:
          RelationalExpression
          | RelationalExpression EqualityOp EqualityExpression

EqualityOp:
          = | ~=

RelationalExpression:
          AdditiveExpression
          | AdditiveExpression RelationalOp AdditiveExpression

RelationalOp:
          < | > | <= | >=

AdditiveExpression:
          MultiplicativeExpression
          | AdditiveExpression AdditiveOp
          MultiplicativeExpression

AdditiveOp:
          + | -

MultiplicativeExpression:
          UnaryExpression
          | MultiplicativeExpression MultiplicativeOp
          UnaryExpression

MultiplicativeOp:
          * | / | mod | div

UnaryExpression:
          PrimaryExpression
```

```
            | ~ UnaryExpression
            | - UnaryExpression

PrimaryExpression:
            | INTEGER
            | FloatConstant
            | true
            | false
            | ID
            | ( Expression )
            | FunctionCall
            | ArrayAccessor

FunctionCall:
            ID ( )
            | ID ( ArgList )

ArrayAccessor:
            ID [ Expression ]

FloatConstant:
            INTEGER . INTEGER
```

Reserved words

The following is a list of reserved words in Dijkstra with Functions:

```
boolean div do false fi float fun if input int mod od print
proc program return true
```

Semantics

The semantics of Dijkstra with Functions includes all of the semantics of Base Dijkstra and adds the following.

Arrays

Arrays **must** be declared. The Expression in the array declaration must be an integer-valued expression. If the expression evaluates to a non-integer value or is not a positive integer, an error results.

Array indices begin at 0.

Accessing an array with an index value that is outside of the bounds of an array results in an exception—a `Java ArrayIndexOutOfBoundsException` is reommended. If the invalid value can be identified during compilation, then the compiler may choose to not generate output, as long as the appropriate error message(s) are produced.

Functions that return multiple values

A function returns one or more values. Each value must be one of the three basic types, int, float, or boolean. There are several semantic issues with the use of a function call as an expression. **Note:** you cannot have a function call as a statement, it must be used as an expression.

Functions used on the right-hand side of an assignment

Let `f` be a function that returns `n` variables. When using `f` as an expression in the expression list of an assignment statement, the effect is that the `n` values are used as if there are `n` individual expressions. For example, let `f` be a function that returns two integers. The following is a legal assignment:

```
x, y <- f()
```

The following statements are illegal, and the compiler should emit an error:

```
x <- f()        # need variables on lhs
x, y <- 1, f()  # need 3 variables on lhs
```

Function calls used as arguments to procedure and function calls

When a function call is used to provide arguments to other functions or procedures, the semantics are similar to the use of function calls in assignments. Assume that the function `f` is the function defined in the previous section. Let p be a procedure defined as:

```
proc doSomething(a, b, c) {
   …
```

The following statement is legal:

```
doSomething(5, f())
```

but the following statement is not legal:

```
doSomething(4, 5, f())
```

because the two values returned by `f` would make four arguments to the procedure rather than the required three.

Printing the result of a function

The grammar specifically indicates that a `print` statement requires a single expression. If you have a function, like `f` above, then the following statement is illegal:

```
print f()
```

Parameters and arguments

A parameter in a procedure and function declaration exists only within the scope of the procedure or function. Consider the following program:

```
program p
   fun sum(x, y) : int {
```

```
        z <- x + y
        return z
}

print sum(40, 2)
```

This program will execute as if it were written as follows:

```
program p
   x, y <- 40, 2
   z <- x + y
   return z
```

Now consider the following program:

```
program p
   x, y, z <- 1, 2, 3
   fun sum(x, y) : int {
          z <- x + y   # outer z
          return z
   }

   print sum(40, 2)
   …
```

After the print statement executes, there are three global (program scope) variables with the following values:

x: 1

y: 2

z: 42

Parameter types

Parameters, like variables, do not have to have their types declared unless the type cannot be inferred from the program.

Return statements

The grammar makes no distinction between a return statement with an expression list (function return) and one without (procedure return). Semantically, a return with an expression list in a procedure is illegal as is a return with no expression list in a function. The compilation should fail in either case.

Dijkstra with Functions relaxation for CS4533

Students taking the undergraduate course, CS4533, may modify their implementation of Dijkstra with Functions to only include functions that return a single value.

Object Dijkstra

Object Dijkstra extends Dijkstra with Functions by adding the concept of an abstract type, or class, to the language. This adds the ability to create classes that can be instantiated during the execution of a program.

Syntax

The change to the Dijkstra with Functions grammar is shown below with modifications highlighted with underlined, maroon text. The typographical conventions for this grammar are the same as in the previous grammars.

Object Dijkstra CFG

```
CompilationUnit:
            ClassDeclarationList EOF
            | Program EOF
            | ClassDeclarationList Program EOF

ClassDeclarationList:
            ClassDeclaration
            | ClassDeclarationList ClassDeclaration

ClassDeclaration:
            class ID ClassBody
            | class ID ( PropertyList ) ClassBody

PropertyList:
            Property
            | PropertyList , Property

Property:
            ID
            | ID [ AccessSpec ]
            | Type ID
            | Type ID [ AccessSpec ]

AccessSpec:
            R | W | RW

ClassBody:
            Declaration
            | Declaration ClassBody

Program:
            program ID DeclarationOrStatementList
```

```
DeclarationOrStatementList:
          DeclarationOrStatement
          | DeclarationOrStatementList DeclarationOrStatement

DeclarationOrStatment:
          Declaration | Statement

Declaration:
          VariableDeclaration
          | ArrayDeclaration
          | ProcedureDeclaration
          | FunctionDeclaration

VariableDeclaration:
          Type IDList Separator
          | ClassName IDList Separator

ClassName:
          ID

ArrayDeclaration:
          Type [ Expression ] IDList Separator

Type:
          float | int | boolean

Separator:
          ε | ;

IDList:
          ID
          | IDList , ID

ProcedureDeclaration:
          proc ID ( ) CompoundStatement
          | proc ID ( ParameterList ) CompoundStatement

ParameterList:
          Parameter
          | ParameterList , Parameter

Parameter:
          ID
          | Type ID
          | ClassName ID
```

```
FunctionDeclaration:
          fun ID ( ) : TypeList CompoundStatement
          | fun ID ( ParameterList ) : TypeList
          CompoundStatement

TypeList:
          Type
          | TypeList , Type

Statement:
          AssignStatement Separator
          | AlternativeStatement
          | IterativeStatement
          | InputStatement Separator
          | OutputStatement Separator
          | CompoundStatement
          | ReturnStatment Separator
          | ProcedureCall Separator
          | MethodCall Separator

AssignStatment:
          VarList <- ExpressionList

VarList:
          Var
          | VarList , Var

Var:
          ID
          | ID . ID
          | ArrayAccessor

AlternativeStatement:
          if GuardedStatementList fi

IterativeStatement:
          do GuardedStatementList od

InputStatement:
          input IDList

OutputStatement:
          print Expression

CompoundStatement:
          { CompoundBody }
```

```
CompoundBody:
          CompoundDeclOrStatement
          | CompoundDeclOrStatement CompoundBody

CompoundDeclOrStatement:
          VariableDeclaration
          | ArrayDeclaration
          | Statement

GuardedStatementList:
          Guard
          | GuardedStatementList Guard

Guard:
          Expression :: Statement

ExpressionList:
          Expression
          | Expression , ExpressionList

ReturnStatement:
          return
          | return ExpressionList

ProcedureCall:
          ID ( )
          | ID ( ArgList )

MethodCall:
          ID . ID ( )
          | ID . ID ( ArgList )

ArgList:
          Argument
          | ArgList , Argument

Argument:
          Expression

Expression:
          LogicalOrExpression

LogicalOrExpression:
          LogicalAndExpression
          | LogicalOrExpression '|' LogicalAndExpression
```

```
LogicalAndExpression:
          EqualityExpression
          | LogicalAndExpression & EqualityExpression

EqualityExpression:
          RelationalExpression
          | RelationalExpression EqualityOp EqualityExpression

EqualityOp:
          = | ~=

RelationalExpression:
          AdditiveExpression
          | AdditiveExpression RelationalOp AdditiveExpression

RelationalOp:
          < | > | <= | >=

AdditiveExpression:
          MultiplicativeExpression
          | AdditiveExpression AdditiveOp
          MultiplicativeExpression

AdditiveOp:
          + | -

MultiplicativeExpression:
          UnaryExpression
          | MultiplicativeExpression MultiplicativeOp
          UnaryExpression

MultiplicativeOp:
          * | / | mod | div

UnaryExpression:
          PrimaryExpression
          | ~ UnaryExpression
          | - UnaryExpression

PrimaryExpression:
          | INTEGER
          | FloatConstant
          | true
          | false
          | ID
          | ID . ID
          | ( Expression )
```

```
        | FunctionCall
        | MethodCall
        | Constructor
        | ArrayAccessor

FunctionCall:
        ID ( )
        | ID ( ArgList )

Constructor:
        ClassName ( )
        | ClassName ( ExpressionList )

ArrayAccessor:
        ID [ Expression ]

FloatConstant:
        INTEGER . INTEGER
```

Reserved words

The following is a list of reserved words in Dijkstra with Functions:

```
boolean class div do false fi float fun if input int mod od
print proc program R return RW true W
```

Semantics

Classes and source files

Classes are defined before a program in a source file. Any number of classes may be defined in a single source file. Each class will be compiled to a separate class file named for the class in the source file. Consider the following source file **sample.djk**.

```
class Class1
   …

class Class2
   …

program sample
   …
```

This will produce, assuming the program is correct, three output files: **Class1.class**, **Class2.class**, **sample.class**.

Variables representing objects

Unlike the base types of *int*, *float*, and *boolean*, *variables that represent objects must be declared*. This is similar to declaring arrays. *Type inference does not apply to variables representing objects*.

When a variable representing an object is declared, the compiler must throw an error if that object is not in the compilation class path. This does not mean that the compiler proper must throw the error. The error may be thrown by other tools in the compilation tool chain and reported via the compiler driver's error mechanism. This allows some relaxation on the type of semantic checking that must be done on function calls.

Creating objects

Once an variable representing an object is declared, an instance of the object's class may be assigned to the object. This is done by invoking the constructor. The constructor is a special class method that takes zero or more arguments and returns a class instance (object) of the appropriate type.

The number of arguments must agree with the number of properties specified in the class declaration. They must also agree in type.

Class properties

A class declaration may declare properties for the class. A property is a value in the state (i.e., a field). Properties may only be of the basic types, *int*, *float*, and *boolean*. Properties are specified as a property list in the declaration. Each property has an access specification. The specification indicates whether the property is read-only, write-only, or read/write. These access specifications only specify how property values may be accessed from other class instances.

If no access specification is given for a property, it is made read-only by default.

Properties may typically be used anywhere variables are used in other Dijkstra languages. There are some restrictions however as specified in the remainder of this section.

Initializing properties

When an instance of a class is created, initial property values are provided in the constructor. Even if a property is read-only, it can (and must) be set to its initial value in the constructor. This is shown in the following example:

```
Class TestClass (a, b [RW])
   …

program Sample
   TestClass x
   x <- TestClass(1, true)
```

To use Java as a reference, the class `TestClass` would have two fields:

```
final int a;
boolean b;
```

The class would have the following constructor:

```
public TestClass(int a, boolean b) {
```

```
    this.a = a;
    this.b = b;
}
```

There must be the same number of expressions in the arguments to the constructor as there are properties specified in the class declaration. The types of the expressions must agree with the types of the properties.

Accessing properties

If a property has read access (read-only or read/write), then its value can be obtained using a "dot" operator to create a path to the property as shown here, using the class `TestClass` above.

```
…
program Sample
    TestClass x
    x <- TestClass(1, true)
    z <- x.a
    …
```

If a property does not have read access, the compiler must emit an error and the compilation must fail if another class or the main program tries to read its value.

A property that has write access (write-only or read-write) then that property may have its value set in an assignment or input statement as shown here, using `TestClass` from the previous examples:

```
…
program Sample
    TestClass x
    x <- TestClass(1, true)
    x.b <- false
    …
```

If a property does not have write access, the compiler must emit an error and the compilation must fail if another class or the main program tries to set its value.

Note: Properties are analogous to private fields in Java objects coupled with getters and/or setters to access the values. The implementation in Dijkstra however, does not need to provide such features. The implementation is left to the compiler writer as long as the semantics described here are followed.

Class methods

Methods are simply functions or procedures that are defined in a class declaration and invoked by providing a path to the method. Object Dijkstra allows only one level of indirection. The methods are invoked by prefixing the method name with the variable's ID followed by a "dot." This is shown below:

```
class TestClass
    fun sum(a, b) : int
```

```
   {
      return a+b
   }

program Sample
   TestClass x
   x <- TestClass()
   print x.sum(40, 2)
```

Appendix: Toy Dijkstra—A Language for Demonstration

Toy Dijkstra is a Dijkstra language that we will use in class for demonstration purposes. Toy Dijkstra is mainly a subset of Base Dijkstra. We eliminate several features in order to keep the compiler as simple as possible, while allowing us to illustrate some of the main features and techniques needed in an actual Dijkstra Compiler.

The code we develop for Toy Dijkstra should be useful for students when they build their Dijkstra compilers. Copying code from Toy Dijkstra, or using a Toy Dijkstra compiler as a starting point is perfectly acceptable and is not considered plagiarism.

Toy Dijkstra CFG

```
Program:
          program ID DeclarationOrStatementList EOF

DeclarationOrStatementList:
          DeclarationOrStatement
          | DeclarationOrStatementList DeclarationOrStatement

DeclarationOrStatment:
          Declaration | Statement

Declaration:
          Type ID Separator

Type:
          int | boolean

Separator:
          ε | ;

Statement:
          AssignStatement Separator
          | AlternativeStatement
          | IterativeStatement
          | InputStatement Separator
          | OutputStatement Separator
          | CompoundStatement

AssignStatment:
          ID <- Expression

AlternativeStatement:
          if GuardedStatementList fi
```

```
IterativeStatement:
          loop Expression Statement

InputStatement:
          input ID

OutputStatement:
          print Expression

CompoundStatement:
          { DeclarationOrStatementList }

GuardedStatementList:
          Guard
          | GuardedStatementList Guard

Guard:
          Expression :: Statement

Expression:
          EqualityExpression

EqualityExpression:
          RelationalExpression
          | RelationalExpression EqualityOp EqualityExpression

EqualityOp:
          = | ~=

RelationalExpression:
          AdditiveExpression
          | AdditiveExpression RelationalOp AdditiveExpression

RelationalOp:
          < | >

AdditiveExpression:
          MultiplicativeExpression
          | AdditiveExpression AdditiveOp
          MultiplicativeExpression

AdditiveOp:
          + | -

MultiplicativeExpression:
          UnaryExpression
          | MultiplicativeExpression MultiplicativeOp
          UnaryExpression
```

```
MultiplicativeOp:
          * | /

UnaryExpression:
          PrimaryExpression
          | ~ UnaryExpression
          | - UnaryExpression

PrimaryExpression:
          | INTEGER
          | true
          | false
          | ID
          | ( Expression )
```

Changes from Base Dijkstra

1. There are only *int* and *boolean* base types.
2. Some of the operators have been removed, such as <= and >=.
3. The do-od iterative statement has been changed to a simple loop that iterates a specific number of times.
4. ID lists and expression lists are no longer needed.
5. There are no logical connectors (& and |).
6. The `/` multiplicative operator is used instead of `div`, since there are no floating point numbers, there is no confusion.