`iDrive Programming Language`
Language Reference Manual
(for COMS W4115)

Parag Jain
`pj2171@columbia.edu`

1 Introduction

`iDrive` is a high level programming language that provides basic constructs to develop algorithms to simulate an environment where cars drive without a need for human intervention. Currently, no such language exists and given the complexity of the paradigm, `iDrive` is extremely useful since it is specialized to accomplish this challenging task. The primary focus of `iDrive` is the abstraction and reduction of non-essential or monotonous tasks so that the researchers no longer need to spend time specifying the characteristics of driving and just need to focus on creating intelligent algorithms for developing fully autonomous cars. `iDrive` provides the researcher with a rich set of tools that may be used intuitively to implement algorithms for driving cars. Every effort has been made to ensure that the language is intuitive and easy to read and understand.

Using `iDrive`, a program first creates a simple car object and subsequently creates other simple objects including cars, traffic signals, stop signs, and pedestrians, and assigns basic properties which characterize them. In the real world, the program would be made aware of the presence of such objects using advanced computer software linked to a range of sensors such as cameras, radar, and GPS. To simulate such input, the program creates these objects randomly and incrementally. As the car heads toward its destination, the program determines the flow of traffic and outputs the outcome of interactions between the objects.

2 Lexical Conventions

`iDrive` uses syntax based on that of C and Java because the idea is to create a development environment that is relatively familiar to those who already know C and Java, hence reducing the learning curve for transitioning to `iDrive`.

`iDrive` uses a standard grammar and character set. Characters in the source code are grouped into tokens, which can be identifiers, keywords, operators, punctuators, or string literals. The compiler forms the longest possible token from a given string of characters; tokens end when white space is encountered, or when it would not be possible for the next character to be part of the token.

2.1 Character Set

`iDrive` accepts standard ASCII characters.

2.2 Identifiers

An identifier is a sequence of letters, digits, and the underscore character and represents the names of user defined variables and functions. All identifiers start with a letter. Identifiers are case sensitive and keywords can not be used as identifiers.

2.3 Keywords

Keywords are identifiers that are reserved words in `iDrive`. They have specific function and can not be used as identifiers. Keywords are case sensitive and valid keywords are:

```
object vehicle pedestrian trafficsignal stopsign
boolean int decimal string
true false
while
for
if else
function
print
random
main
```

2.4 Constants

A constant is used to set value for an identifier that forms an attribute for an object.

2.4.1 Integer constants

Integer constants are represented with whole numbers in decimal format. An integer constant constitutes only of digits; decimal point and exponent are not allowed. A unary - operator is allowed. For example, 4 or -342.

2.4.2 Floating point constants

Floating point constants are represented with a whole part, a decimal point and a fractional part. The whole part and the fractional part are made up only of digits. A unary - operator is allowed. For example, 8.1 or -0.42322.

2.4.3 String constants

String constants are made up of a sequence of zero or more characters that are enclosed in quotes. For example, "John Smith" or "2".

2.5 Operators

Operators are tokens that specify an operation on at least one operand. They are used in expressions, assignments and object dereferencing.

`< <= > >= == !=`	Relational operators
`!` && ||	Logical operators
`+ - * / ^`	Mathematical operators
`-`	Unary operator
`=`	Assignment operator
.	To dereference attribute of an object

The table below shows the precedence the `iDrive` compiler uses to evaluate operators. Operators with the highest precedence appear at the top of the table; those with the lowest precedence appear at the bottom. Operators of equal precedence appear in the same row.

Category	**Operator**	**Associativity**
Dot	`.`	Left to right
Unary	`-`	Right to left
Mathematical	`* /`	Left to right
Mathematical	`+ -`	Left to right
Relational	`< <= > >=`	Left to right
Relational	`== !=`	Left to right
Logical	`!`	Left to right
Logical	&&	Left to right
Logical	||	Left to right
Assignment	`=`	Right to left

Associativity relates to precedence, and resolves any ambiguity over the grouping of operators with the same precedence. Most operators associate left-to-right, so the leftmost expressions are evaluated first. The assignment operator and the unary operators associate right-to-left.

2.6 Punctuators

‘‘ ’’	To enclose string constants
{ }	To enclose a group of statements in a procedure or function
( )	To enclose a group of arguments for a function
;	To separate a statement from another
,	To separate arguments within a function

2.7 Comments

Inline comments begin with the // character sequence and end with a line feed. Alternatively, comments may begin with the opening character sequence /* and close with the sequence */. Comments cannot be nested.

2.8 White Space

White space characters which include spaces, tabs, and line feed characters may used to separate keywords, operators, and code tokens in the input but are discarded during parsing.

2.9 Semicolons and Line Breaks

Semicolons serve as a statement separator, and line breaks serve as a terminator. Multiple statements may be put on a single line of source code using semicolons in between each statement.

3 Data Types

3.1 Simple Types

These can be defined on their own, or they can serve as attributes within a complex type.

`boolean`	Used to hold true/false data
`int`	Used to hold integer data
`decimal`	Used to hold decimal data
`string`	Used to hold string data

3.2 Complex Types

3.2.1 `object`

An `object` is used to define a simple object. An `object` is characterized by one or more user defined fields or attributes holding some data. An `object` has no predefined attributes. Attributes are all custom and could be added at initialization as well as later in the program. For example,

```
object identifier (int identifier1 = 1, string identifier2 = ‘‘John Smith’’,
int identifier3 = 5);
```

or

```
object identifier (int identifier1, string identifier2, int identifier3);
identifier.identifier1 = 1;
identifier.identifier2 = ‘‘John Smith’’;
identifier.identifier3 = 5;
```

3.2.2 `vehicle`

A `vehicle` is used to define a car object. Similar to `object`, a `vehicle` is characterized by one or more user defined fields or attributes holding some data. At this point `vehicle` has no predefined attributes but such predefined attributes may be introduced in a later version of `iDrive`.

3.2.3 `pedestrian`

A `pedestrian` is used to define a car object. Similar to `object`, a `pedestrian` is characterized by one or more user defined fields or attributes holding some data. At this point `pedestrian` has no predefined attributes but such predefined attributes may be introduced in a later version of `iDrive`.

3.2.4 `trafficsignal`

A `trafficsignal` is used to define a car object. Similar to `object`, a `trafficsignal` is characterized by one or more user defined fields or attributes holding some data. At this point `trafficsignal` has no predefined attributes but such predefined attributes may be introduced in a later version of `iDrive`.

3.2.5 `stopsign`

A `stopsign` is used to define a car object. Similar to `object`, a `stopsign` is characterized by one or more user defined fields or attributes holding some data. At this point `stopsign` has no predefined attributes but such predefined attributes may be introduced in a later version of `iDrive`.

4 Statements

Statements are executed in the sequence in which they appear in the code.

4.1 Compound Statements

A compound statement, or block, allows a sequence of statements to be treated as a single statement. A compound statement begins with a {, (optionally) contains statements, and ends with a }. For example,

```
{
statement1
statement2
...
}
```

4.2 Expression Statements

An expression statement can be a combination of operators, identifiers, and literals. Upon evaluation, an expression returns a value. The value type is dependant on the expressions being combined. For example,

```
identifier == ``John Smith''
```

or

identifier1 > *identifier2*.

4.3 Control Statements

To control a program's flow the following control statements are supported much like in C and Java:

4.3.1 `while` loop

The `while` loop has the following syntax:

```
while (expression)
{
statement1
statement2
...
}
```

In a `while` loop, the statements inside the `while` loop are repeatedly executed as long as *expression* is true.

4.3.2 `for` loop

The `for` loop has the following syntax:

```
for (expression1; expression2; expression3)
{
statement1
statement2
...
}
```

In a `for` loop, first *expression1* is executed. Then the statements inside the `for` loop are repeatedly executed followed by *expression3* as long as *expression2* is true.

4.3.3 `if else` condition

The `if else` statement has the following syntax:

```
if (expression)
{
statement1
statement2
...
}
else
{
statement3
statement4
...
}
```

The statements following the control expression are executed if the value of the control expression is true (nonzero). The statements in the `else` clause are executed if the control expression is false (zero).

5 Functions

5.1 User-defined Functions

The `function` keyword is used to define user-defined functions. A function in `iDrive` does not have a return type and all parameters passed to the function are "passed by reference" and the outcome of the function is reflected directly on the input. Functions can access global variables, parameters, and locally declared variables. `function` has the following syntax:

```
function identifier (parameter1, parameter2, ...)
{
statement1
statement2
...
}
```

5.2 Built-in Functions

`iDrive` provides one built-in function which may not be redefined.

5.2.1 `print`

The `print` function is used to output text to the screen. This function takes a combinations of string constants (elcosed in “”) and identifiers (holding some data) as input, resolves the identifiers to text, and outputs the resulting text to the screen. The `print` statement has the following syntax:

```
print(‘‘My name is ’’ identifier ‘‘.  What is your name?’’)
```

5.2.2 `random`

The `random` function is used to randomly pick an integer between 0 (inclusive) and ***`bound`*** (exclusive). This function takes an integer value for ***`bound`*** as input and outputs a randomly chosen integer. The `random` statement has the following syntax:

```
random(bound)
```

6 Scope

There are two types of scope – local and global. Identifiers declared within a function are local only to that function and may not be used outside the funtion. Global identifiers are declared outside any functions and may be used anywhere in the program.

7 Representative Program

Here is a representative program that demonstrates a simple `iDrive` program, which first creates a simple `vehicle` object and subsequently creates other objects. As the car heads from its source position to its destination position, the program determines the flow of traffic and outputs the outcome of interactions between these objects:

```
/* This is the vehicle object that needs to be transported from
source position to destination position */

vehicle myCar(decimal currentXPosition=0, decimal currentYPosition=0,
      decimal currentSpeed=0, string heading=’’North’’,
      decimal destXPosition=0, decimal destYPosition=50,
      decimal distanceToDest, boolean isClearToGo = true);
```

```
/* This is an object that stores global variables */

object global(int rand, decimal acceleration = 5, decimal deceleration = 5);

/* The main() function must always be present.
It is the entry point for a iDrive program much like C and Java.*/

function main()
{
      distanceToDestination(myCar);

      while (myCar.distanceToDest > 0.5) {

            /* myCar.isClearToGo will take into account pedestrians, traffic signals,
            stop signs, and other vehicles on the roadway */

            if (myCar.isClearToGo) {

                  /* To simulate such input in absence of sensors such as cameras,
                  radar, and GPS, the program creates these objects randomly and
                  incrementally.  */

                  if (random(4) == 0) {
                        myCar.isClearToGo = false;

                        global.rand = random(4);

                        if (global.rand == 0) {
                              vehicle newCar();
                              global.deceleration = 20;
                              print(‘‘Decelerating quickly due to an
                                    unexpected approaching car’’);
                        }
                        if (global.rand == 1) {
                              pedestrian newPedestrian();
                              global.deceleration = 20;
                              print(‘‘Decelerating quickly due to an unexpected
                                    pedestrian’’);
                        }
                        if (global.rand == 2) {
```

```
                        trafficsignal newTrafficSignal();
                        global.deceleration = 5;
                        print(``Decelerating due to an traffic signal up
                              ahead'');
                  }
                  if (global.rand == 3) {
                        stopsign newStopSign();
                        global.deceleration = 5;
                        print(``Decelerating due to a stop sign up ahead'');
                  }
            }
      }

      if (myCar.isClearToGo) {
            if (myCar.currentSpeed < 45) {
                  accelerate(myCar, global.acceleration);
            }

            updateCurrentPosition(myCar);
            distanceToDestination(myCar);
      }
      else {
            if (myCar.currentSpeed > 0) {
                  decelerate(myCar, global.deceleration);
            }
            else {
                  global.deceleration = 5;
                  myCar.isClearToGo = true;
                  accelerate(myCar, global.acceleration);
            }

            updateCurrentPosition(myCar);
            distanceToDestination(myCar);
      }
}

if (myCar.distanceToDest <= 0.5 && myCar.currentSpeed = 0) {
      print(``Please walk this distance'');
}
else {
```

```
            print(``Approaching destination'');
            while (myCar.currentSpeed > 0) {
                  decelerate(myCar, global.deceleration);
                  updateCurrentPosition(myCar);
                  distanceToDestination(myCar);
            }
      }
}

/* Below are all the user defined functions */

/* accelerate increases the car speed by a specified amount */

function accelerate(v, howFast)
{
      v.currentSpeed = v.currentSpeed + howFast;
      print(``Current speed increased to '' ++ v.currentSpeed);
}

/* decelerate decreases the car speed by a specified amount */

function decelerate(v, howFast)
{
      v.currentSpeed = v.currentSpeed - howFast;
      print(``Current speed decreased to '' ++ v.currentSpeed);
}

/* updateCurrentPosition updates the current position of the car taking current
speed into account */

function updateCurrentPosition(v)
{
      if (v.heading == "North") {
            currentXPosition = 0;
            currentYPosition = currentYPosition + (v.currentSpeed / 3600) * 10;
      }
      if (v.heading == "East") {
            currentXPosition = currentXPosition + (v.currentSpeed / 3600) * 10;
            currentYPosition = 0;
      }
      if (v.heading == "West") {
            currentXPosition = currentXPosition - (v.currentSpeed / 3600) * 10;
            currentYPosition = 0;
      }
      if (v.heading == "South") {
            currentXPosition = 0;
            currentYPosition = currentYPosition - (v.currentSpeed / 3600) * 10;
      }

      print(‘‘Current position is (’’ ++ v.currentXPosition ++ ‘‘, ’’ ++
            currentYPosition ++ ‘‘)’’);
}

/* distanceToDestination updates the distance left to the destination position
*/

function distanceToDestination(v)
{
      v.distanceToDest = ((v.destXPosition - v.currentXPosition) ^ 2 +
                         (v.destYPosition - v.currentYPosition) ^ 2) ^ 0.5;
      print(‘‘Current distance from destination = ’’ ++ v.distanceToDest);
}
```

■