iDrive Programming Language
COMS W4115 Project Proposal

Parag Jain
pj2171@columbia.edu

1 Introduction

Cars equipped with a system that is capable of driving from one point to another without input from a human operator have got significant attention from academic, commercial, defense, and government sectors over the past few decades. Early prototypes date as far back as late 1970s, however important progress only came to be seen in the late 2000s through programs like DARPA Grand Challenge competitions and Google Driverless Car project. Some proposed systems depend on infrastructure-based guidance systems; while more advanced systems propose to simulate human perception and decision-making during steering of a car via advanced computer software linked to a range of sensors such as cameras, radar, and GPS. Some of the advantages of "Driverless Cars" include managing traffic flow to increase road capacity, avoid accidents by eliminating driver error, relieving vehicle occupants from driving and navigating chores, transporting loads in dangerous zones such as battlefields, and reducing costs of employing drivers.

2 iDrive Programming Language

iDrive is a high level programming language that provides basic constructs to develop algorithms for cars to 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.

3 Language Outline

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.

3.1 Simple Types

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

3.2 Complex Types

`Segment` : Represents a segment out of the driving directions
`Heading` : An attribute of a `Segment` that determines usually direction altering action that the car must perform at the start of a `Segment`. E.g. `TurnRight, TurnLeft, Proceed, BearRight, BearLeft, SwitchToRightLane, SwitchToLeftLane.`
`Distance` : An attribute of a `Segment` that determines how far the car must drive before reaching the next `Segment`
`TargetSpeed` : An attribute of a `Segment` that determines how fast the car may drive along the `Segment`
`Directions` : An array of segments, which form the driving directions from source position to destination position (in the initial implementation data for this variable is read from a tab delimited text file)
`Position` : Repesents position in terms of geographic latitude and longitude

3.3 Control Statements

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

`while` loop: To repeatedly execute a block of code until a certain condition is met
`for` loop: To repeatedly execute a block of code until a certain condition is met
`if / else` condition: To conditionally execute a block of code

3.4 Built-in Functions

Here are the initially proposed keywords and frequently used built-in functions:

`function` : To create user-defined functions
`TurnRight` : To turn the car right
`TurnLeft` : To turn the car left
`Proceed` : To make the car proceed forward or to move the car as per the `Heading`
`Reverse` : To reverse the car
`Stop` : To stop the car as soon as possible
`SlowDown` : To slow down the car
`Accelerate` : To accelerate the car
`BearRight` : To make the car bear right
`BearLeft` : To make the car bear left
`SwitchToRightLane` : To make the car switch to the adjacent right lane
`SwitchToLeftLane` : To make the car switch to the adjacent left lane
`IsClearToGo` : To check if the car is clear to proceed (interrogates sensors attached to the car)
`Park` : To make the car park
`ReadDirections` : To read directions from a tab delimited text file

3.5 Other Features

{ } 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
`/* */` All characters between `/*` and `*/` are treated as comments
`//` To include inline comments
`+ - * /` Mathematical operators
`< <= > >= == !=` Relational operators
`!` && || Logical operators

4 Representative Program

Here is a representative program that demonstrates a simple implementation of `Drive` algortihm, which reads the driving directions passed to it by the navigation system (in the initial implementation driving directions are read from a tab delimited text file) and transports the car from source position to destination position:

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

function init() {
      string directionsFile = ‘‘directions.txt’’;
      Directions d;
      Position srcPosition;
      Position destPosition;
      [d, srcPosition, destPosition] = ReadDirections(directionsFile);
      Drive(srcPosition, destPosition, Directions);
}


/* This is a user defined function that transports the car from
source position to destination position given the driving directions.*/

function Drive(srcPosition, destPosition, Directions) {
      /* Current position of the car */
      Position CurrentPosition = srcPosition;
      /* Current speed of the car */
      decimal CurrentSpeed = 0.0;

      i = 0;
      while (i < Directions.length) {
            /* Current Segment to execute */
            Segment CurrentSegment = Directions[i];

            /* Distance covered along current Segment */
            decimal DistanceCovered = 0.0;

            while (CurrentSegment.Distance > DistanceCovered) {

                  /* IsClearToGo will take into account pedestrians,
                  traffic signals, and other traffic on the roadway by
                  interrogating sensors attached to the car */

                  if (IsClearToGo) {
                        /* Accelerate will increase the car speed if
                        CurrentSpeed < CurrentSegment.TargetSpeed
                        It will also update the CurrentSpeed.  */
```

```
                Accelerate;

                /* Proceed will move the car forward or as per
                CurrentSegment.Heading if this event hasn't already taken
                place.  It will also update the CurrentPosition and
                DistanceCovered accoring to CurrentSpeed.  */

                Proceed;
            }
            else {
                SlowDown;
            }

            if (CurrentPosition == destPosition) {
                Stop;
                Park;
            }
        }
        i++;
    }
}
```

■