Dot Language Specification

David Vieten

May 9, 2024

Introduction

Fortunately, I have had the privilege of being around the sport of hockey for the majority of my lifetime. Over the years, it has become clear to me that the current method of drawing face-off plays is outdated. A face-off takes place starting a game or following a whistle. The ref drops the puck and two center-men battle for the puck. Face-offs play an important role in hockey, as it is an opportunity to gain possession of the puck. As a result coaches draw up routes for each of the 5 guys on the each, dictating what each player should do on a face-off win. Each team often has multiple of these set routes called ”face-off plays.” These faceoff plays may change on any given week depending on who the opponent is and what their weaknesses are. As a result, the players are responsible for memorizing their routes. Currently, coaches either draw up the faceoff plays on a white board and hope the players can remember it, or they draw them up on paper and distribute photocopies for the players to study.
The programming language ”Dot” solves this problem. ”Dot” allows coaches to create face-off plays in an svg that can be emailed to players. This language saves coaches a headache by providing a way easily distribute face-off plays without wasting paper and ink to do so. Coaches will also be able to save face-off plays, allowing them to keep track of what plays they have used against who in the past. This ability to save face-off plays will also allow coaches to save scouting reports. Every week, coaches watch video on the opposing team to get an idea of what they are in for come game time. While watching video, coaches can create and save the opposing team’s faceoff plays to determine the best counter plays. ”Dot” will save coaches time and allow players to study their routes in an efficient manner.

Design Principles

Several design principles guide Dot’s development to address the problem of creating and distributing faceoff plays. With accessibility and usability at the forefront of priorities, Dot produces an SVG allowing coaches to easily distribute visual representations of plays. Efficiency and scalability are two factors that Dot seeks to achieve. The ability to save and organize face-off plays supports scalability by allowing coaches to implement libraries of strategies/pre-scouts over time. Adaptability and extensibility are also fundamental design principles in the development of the language. Dot allows for the creation of custom plays designed for specific opponents, providing an advantage in the hockey world. Furthermore, Dot’s support for saving and analyzing opponent’s face-off plays contributes to its strategic use as a tool for planning and adapting. Moreover, Dot is built with accessibility, efficiency, and adaptability in mind.

Examples

lefthash right offense corner
righthash right offense backdoor
dot right offense slot
stackinside right offense walkline
stackoutside right offense upwall

lefthash left offense backdoor
righthash left offense slot
dot left offense upwall
rightpoint left offense net
leftpoint outside left offense walkline

stackoutside left defense net
righthash left defense walkline
dot left defense net
leftpoint left defense slot
stackoutside left defense backdoor

Language Concepts

To write programs in this language, users need to have a grasp on the concepts of primitives and combining forms as well as knowledge of the sport of hockey. The primitives help define areas on the hockey rink through the use of side and zone indicators to specify the context of the location. End routes represent endpoints that players will move to from their original position once the puck drops. Combining forms involve creating these routes by pairing start and end points. These routes can be organized into a board, which is a list of routes. A sequence of strategic moves can then be represented off a faceoff. By understanding these core concepts, users can effectively design player movements that will produce an advantage on game day.

Formal Syntax

```
     <expr> ::= <route>+
     <route>::= <routedef><dotplace>
 <routedef> ::= <startroute><endroute>
 <dotplace> ::= <side><zone>
 <endroute> ::= net
              | walkline
              | downwall
              | upwall
              | corner
              | hold
              | slot
              | backdoor
     <zone> ::= offense
              |  defense
     <side> ::= right
              |  left
<startroute> ::= lefthash
              |  righthash
              |  dot
              |  rightpoint
              |  leftpoint
              |  stackinside
              |  stackoutside
```

Table 1: Language Semantics.

Syntax	Abstract Syntax	Prec./Assoc	Meaning
<routeef>	1110.1	a	
<zone>	10.1	n/a	zone is primative
3	23.113231	c	
4	woooo		

Semantics

The basic building blocks of Dot are locations and end routes. Possible locations include the left hash, right hash, dot, etc. End routes consist of options like net, walkline, corner, etc. The combined forms are made up of these basic values using different constructs such as tuples and variant types. For instance, I combine a side and zone to define a "dotplace". A "start" is created by combining a location with a dot place. Routes are formed by combining a start with an endroute. Programs are run by inputting a txt. This can be done on the command line. When evaluating a program, an svg. file is produced with all the routes specified through the txt. file.