Project Proposal

Jacob Cohen, Kevin Pepin

0.1 Introduction

This scheduling language is motivated by a common ride-share problem that Williams students face. Amongst students groups, friend groups, and even campus wide, it is common for those with cars at Williams to offer rides to those that don't have. On the track and field team, someone might offer a ride to NYC at 2pm next Friday or to Boston the Tuesday before Thanksgiving. On the other side, I might request a ride to the Albany airport next Saturday to catch my flight or to the Pittsfield target to get some room supplies. With only a few offers and requests, the requesters can just check to see if there's an offer that fits their needs. However, during school breaks when the offer/request list is massive, finding a valid matching of offers and requests can be quite tedious. A language composed of standardized offer and request entries could allow programmers to model any rapidly evolving offer/request list given any number of constraints.

There are a few major benefits of making this a programming language. First, a program can consist of any number of offers/requests, so it has the flexibility to find carpools for a small number of people but with the same language can find carpools across the entire Williams campus of 2,000 people. Next, since a program is just a collection of request/offers, if someone decides they won't be driving and revokes their offer, then finding a new carpool setup is just a matter of removing a line of the program and re-evaluating it. Changing a request or offer is merely a change in the program. As a programming language, it can be expanded to model as many constraints as we like as long as we can parse them and work them into the evaluation process. For example, requests and offers should be at a specific date/time. Offers should also only contain a fixed number of seats. We can add any additional features to our requests/offers in our programming language as long as we parse them and pass them to the evaluator.

0.2 Design Principles

The guiding technicality of this language is that there are two main types it models: offers and requests. Each is a combining form containing several primitive types that make each offer or requests unique. A program in this language is some example scenario of offers and requests which looks like a list of them. This design choice allows individual offers/requests to be easily modified in a program and allows for a program to be infinitely long. This language syntax is intuitive because a program resembles the group chat that motivated the programming language where we can imagine multiple people would have communicated unstandardized offers and requests for rides.

0.3 Examples

One example program:

```
Offer: Jacob to NYC at 14-16 on 11/5-11/6 with 3 seats
Request: Oscar to NYC at 15 on 11/5
```

A second example program:

```
Offer: Jacob to NYC at 14-16 on 11/5-11/6 with 3 seats
Offer: Jack to Boston at 12 on 11/26 with 1 seats
Request: Cooper to Boston at 12 on 11/26
Request: Oscar to NYC at 9-17 on 11/5
```

A third example program:

```
Request: Cooper to Springfield, Pittsfield, Boston at 15 on 11/26
Request: Oscar to NYC, Westchester at 9 on 11/5
Offer: Jacob to NYC at 14-16 on 11/5-11/6 with 3 seats
Offer: Jack to Boston at 12 on 11/26 with 1 seats
```

0.4 Language Concepts

A programming contains one or more of the main combining forms: Offers and Requests. Offers contains the following:

1. A name which is a primitive
2. A location which is a primitive
3. A TimeRange which is a combining form containing two times or just a time primitive
4. A DateRange which is a combining form containing two dates or just a date primitive
5. A seatCount which is a primitive int

Requests contains the following:

1. A name which is a primitive
2. A locationList which is one or more primitive locations
3. A TimeRange which is a combining form containing two times or just a time primitive
4. A DateRange which is a combining form containing two dates or just a date primitive

The key idea here is that our program is composed of several Offers and Requests which are each a combination of a few primitives that make them unique. A program may model a real-world set of requests/offers or may be some fictional representation. The flexibility of Offers and Requests mean that a program can represent endless scenarios. We also have a few other combining forms to allow us to express ranges of times, dates, and cities that way the programming language can model flexibility in someone's schedule.

0.5 Syntax

The syntax in the examples above fits the following BNF. It contains some shorthand notation for representing time in 24 hour format and calendar dates.

```
<inputSchedule>    ::= <event><ws><inputSchedule>
                    |  <event>
<event>            ::= <offer>
                    |  <request>
<offer>            ::= Offer:<ws><name><ws>to<ws><location><ws>at<ws><timeRange><ws>
                               on<ws><dateRange><ws>with<ws><seatCount><ws>seats
<Request>          ::= Request:<ws><name><ws>to<ws><locationList><ws>at<ws><timeRange>
                               <ws>on<ws><dateRange>
<timeRange>        ::= <time>
                    |  <time>-<time>
<dateRange>        ::= <date>
                    |  <date>-<date>
<locationList>     ::= <location>
                    |  <location><ws>,<ws><locationList>
<name>             ::= <letter>+
<location>         ::= <letter>+
<seatCount>        ::= <digit>+
<digit>            ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<time>             ::= 0| 1 | 2 | ... | 23
<date>             ::= 1/1 | 1/2 | ... | 12/30 | 12/31
<letter>           ::= \alpha \in {a...z}
                    |  \alpha \in {A...Z}
<ws>               ::= ␣+
```

0.6 New Semantics (Lab 9)

Syntax	**Abstract Syntax**	**Type**	**Prec./Assoc.**	**Meaning**
l	Letter of char	char	n/a	l is a primitive. We represent letters using the F# char data type.
d	Digit of int	int	n/a	d is a primitive. We represent digits using the F# int data type.
$time$	Time of int	int	n/a	$time$ is a primitive. We represent times using the F# int data type (The existing parser does not limit time to be between 0-23).
$date$	Date of int * int	int $\rightarrow$ int $\rightarrow$ Date	n/a	Creates a $date$ type from two integers.
$seatCount$	seatCount of int	int	n/a	seatCount is an object created from an integer which represents the number of seats available in an offer.
$location$	location of string	string	n/a	location is string which represents where someone is offering to drive/requesting to go to.
$name$	name of string	string	n/a	name is string which represents someone who is either offering to drive somewhere or requesting a ride somewhere.
$locationList$	locationList of location List	location List	n/a	A list of location types.
$dateRange$	dateRange of date * date	dateRange	n/a	A dateRange type storing two date types.
$timeRange$	timeRange of time * time	timeRange	n/a	A timeRange type storing two time types.
$Offer$: name to locationList at timeRange on dateRange with seatCount seats	Event of name * locationList * timeRange * dateRange * seatCount	name $\rightarrow$ locationList $\rightarrow$ timeRange $\rightarrow$ dateRange $\rightarrow$ seatCount $\rightarrow$ Event	n/a	Creates an Event type with Offer constructor the provided fields.
$Request$: name to locationList at timeRange on dateRange	Event of name * locationList * timeRange * dateRange	name $\rightarrow$ locationList $\rightarrow$ timeRange $\rightarrow$ dateRange $\rightarrow$ Event	n/a	Creates an Event type with the Request constructor using the provided fields.
e_1 e_2 e_3 e_4 ...	inputSchedule of Event list	Event list $\rightarrow$ String	n/a (Order of events in the list does not impact the final evaluation)	inputSchedule evaluates all of the Events, returning a string representation of the optimal pairing of Offer and Request events.

0.7 Old Semantics (Lab 8)

i. The primitives in this language are name, location, digit, time, and date. Each of these can not be broken down further into something that has meaning in our programming language. A name is the name of someone offering a requesting. The location is a place that the offer/request goes to. While these can both be broken down into chars, the char's don't mean anything in our language. Digits can not be broken down further since they are already $0 - 9$. Time is a whole number $1 - 24$ so can't be broken down further into anything that has meaning in our programming language. Dates are the date an offer/request is made for so can't be broken down into anything smaller that has meaning in our language.

ii. Our program is a combination of offers and requests. An offer is a combination of a name, location, timeRange, dateRange, and seatCount. A request is a combination of a name, locationList, timeRange, and dateRange. A timeRange can either be a time (which is a primitive) or it can combine two times. A dateRange can either be a date (which is a primitive) or it can combine two dates. A locationList can either be a location (which is a primitive) or it can combine two or more locations. A seatCount combines one or more digits (which are primitive). Name is the last primitive.

iii. We need to define an ADT:

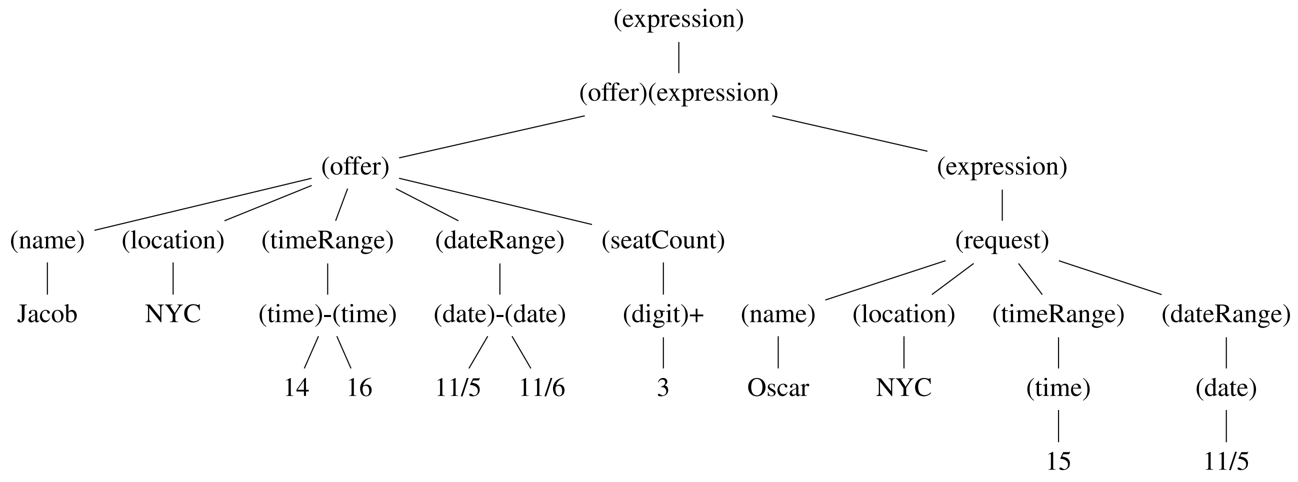
```
type Entry =
| Offer of  name * location * timeRange * dateRange * seatCount
| Request of name * locationList * timeRange * dateRange
```

Since we can have infinitely many requests and offers, the AST would contain:

```
type Expr = Entry list
```

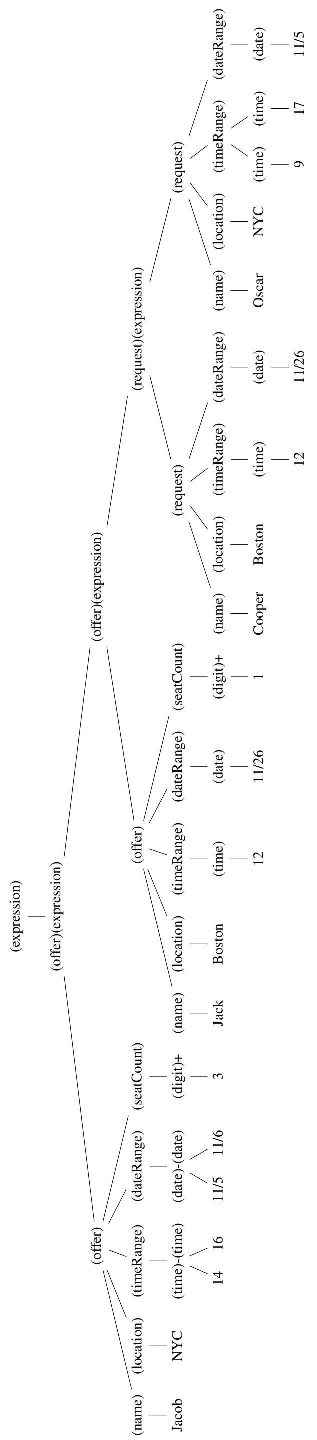
iv.1. We can make an AST for the first example program which is repeated here for reference. Note that this AST uses () in place of <> to save space wdith wise.

```
Offer: Jacob to NYC at 14-16 on 11/5-11/6 with 3 seats
Request: Oscar to NYC at 15 on 11/5
```



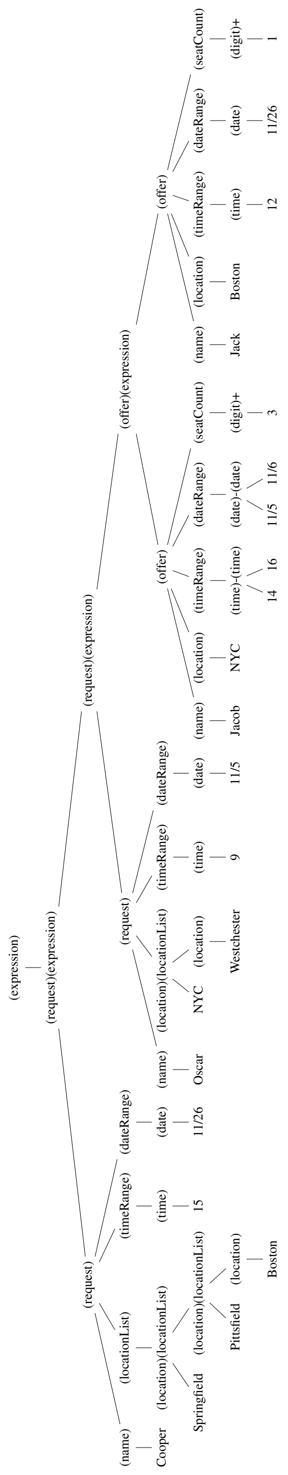
iv.2. For the second program we have

```
Offer: Jacob to NYC at 14-16 on 11/5-11/6 with 3 seats
Offer: Jack to Boston at 12 on 11/26 with 1 seats
Request: Cooper to Boston at 12 on 11/26
Request: Oscar to NYC at 9-17 on 11/5
```



iv.3. For the third program we have

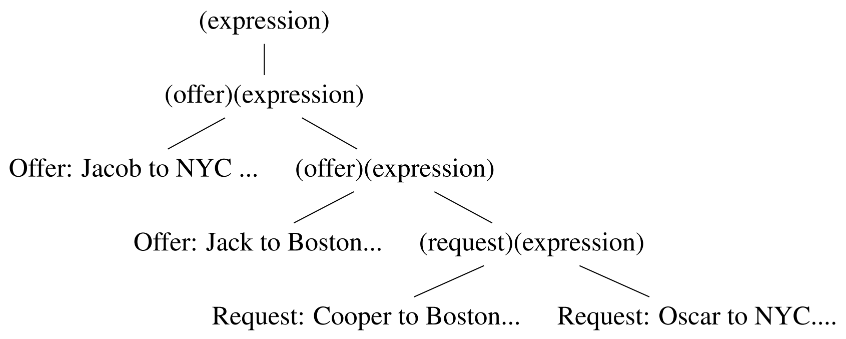
```
Request: Cooper to Springfield, Pittsfield, Boston at 15 on 11/26
Request: Oscar to NYC, Westchester at 9 on 11/5
Offer: Jacob to NYC at 14-16 on 11/5-11/6 with 3 seats
Offer: Jack to Boston at 12 on 11/26 with 1 seats
```



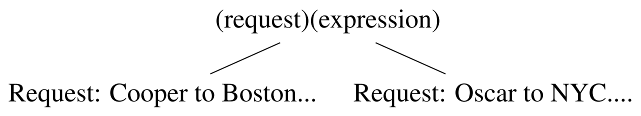
v. Once parsed, a program is evaluated by passing the list of offers/requests to an algorithm that maximizes the number of filed requests subject to the offer constraints. The program does not take any input as the program itself is a complete record of the offers and requests. The evaluated program should return a list of all the offers. Each offer should show which requests it fills if any. And there should also be a returned list of unmatched requests that couldn't be filled by the offers in the program.

   Evaluation remains a post-order traversal of the AST. Consider the following AST of the second example program which is copied here for reference. The AST is abbreviated since we don't need to fully depict all of the primitives).

```
Offer: Jacob to NYC at 14-16 on 11/5-11/6 with 3 seats
Offer: Jack to Boston at 12 on 11/26 with 1 seats
Request: Cooper to Boston at 12 on 11/26
Request: Oscar to NYC at 9-17 on 11/5
```



   Only the expressions in this language can be evaluated. We'll begin the post-order traversal at

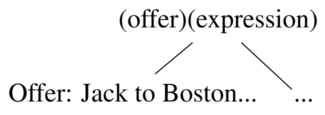


   Since we only have two requests, evaluating them just returns a list of the unmatched requests. Output could look like

```
MATCHES:

UNMATCHED REQUESTS:
Request: Cooper to Boston at 12 on 11/26
Request: Oscar to NYC at 9-17 on 11/5
```

   Then we'll step up to



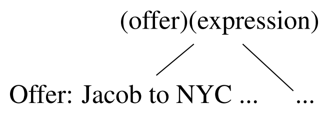
   Now we can match Jack's offer to Cooper's request since the extra conditions can be satisifed for each. The return might look like

```
MATCHES:
Offer: Jack to Boston at 12 on 11/26 with 1 seats
```

```
    FULLFILLS: Request: Cooper to Boston at 12 on 11/26

UNMATCHED REQUESTS:
Request: Oscar to NYC at 9-17 on 11/5
```

And lastly we'll step up to



Now we can match Jacob's offer to Oscar's request since the extra conditions can be satisfied for each. The return might look like

```
MATCHES:
Offer: Jack to Boston at 12 on 11/26 with 1 seats
    FULLFILLS: Request: Cooper to Boston at 12 on 11/26
Offer: Jacob to NYC at 14-16 on 11/5-11/6 with 3 seats
    FULLFILLS: Request: Oscar to NYC at 9-17 on 11/5

UNMATCHED REQUESTS:
```

Which would be our final evaluation.