Pül: Project Proposal

Jacob Cohen, Kevin Pepin
Video link:

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 one. 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, someone might request a ride to the Albany airport next Saturday to catch their flight or to the Pittsfield Target to get some room supplies. With only a few offers and requests, the requestors 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.

By making this a programming language, it can handle any sized input program that evaluates with consistent results. A program can consist of any number of offers/requests. It has the flexibility to find carpools for a small number of people. As you can see in the four example programs below, this is fairly trivial to also do by hand. But, the same language can find carpools across the entire Williams campus of 2,000 people. Imagine a sample program like the ones below but with hundreds of lines. Trying to keep track of such a large amount of requests and offers by hand is no longer a viable option.

Since a program is just a collection of request and 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. Similarly, changing a request or offer's data is merely a change in the program. Regardless of the size of your program, if you need to make changes to the input, recomputing the result simply involves re-evaluating the new program.

As a programming language, it can be expanded to accept as many formats and include as many constraints as we'd like. If you wanted to accept some other date format, you'd only need to add a new parser for that date format. If you wanted to allow for anonymous offers you could edit the parser to look for zero or more names. Since only the format of the input program changes, none of the interpretation and evaluation breaks, as long as the new parser follows the existing ADT's. Likewise, you may want to add new input fields. Maybe you want to consider the limited baggage space in vehicles? Or you are willing to pay money for your request? Or you are only willing to accept above a certain amount for your offer? Regardless of the extension, this programming language can evolve to include these restrictions. We can add any additional features to our requests/offers in our programming language as long as we parse them, pass them to the evaluator, and modify the evaluator to actually consider them.

0.2 Design Principles

The backbone of this language is that one can represent the carpool match-making space via two main types: offers and requests. An offer embodies all crucial information pertaining to someone who has space to take more people in their vehicle. Likewise, a request represents someone who is searching for a seat to somewhere. Each is a combining form containing several primitive types (or near primitive types) that make each offer or request 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. For example, one could translate the chat below into the program below it:

An Imaginary Group Chat
Cooper: Can anyone give me a ride to Bennington tomorrow?
Jacob: I'm heading home to NYC tonight if anyone needs a ride.
Jack: I'm headed to Boston after practice today.
Oscar: Could I get a ride to the airport tomorrow before my 6am flight?

Represented as a Program

Notice that, accounting for interpreting colloquial terms like "tonight" and "after practice," the program is visually similar to the group chat messages that inspired its creation. Thus, rather than hashing out the carpools with a bunch of texts in the group chat, one could evaluate the program above.

0.3 Examples

You'll find multiple example programs in $code/Project/ExamplePrograms/$. Several key ones are listed here with example outputs. Additional ones are included at the path above and all may be run with
at the $Project$ level.

1) A super basic program where the output matches the only offer and request.

   The expected output is

   Evaluated programs return all the requests in the program organized into a "MATCHED REQUESTS" section and an "UNMATCHED REQUESTS" section (doesn't appear here since there aren't any, see later programs for ones with unmatched requests). Each request is followed by the offer (minus its seat info) that will fulfil it. The returned output is the maximum number of requests that can be fulfilled by the program's offers subject to the constraints in the program.

2) A more complex program where the output matches multiple offers and requests.

   The expected output is

3) This one builds on the last by demonstrating the list of locations feature as well as showing what happens with unmatched offers and requests.

   The expected output is

   This is the first example where we see unmatched and matched requests. Notice that the unmatched requests are listed in their own section with no offer following them. We do not print unmatched offers (AKA Andrew in this example) since the output is intended to help requestors find their ride, so it does not matter to Andrew that no one is riding in his car.

4) This one is much longer and incorporates all of the languages features. One may have noticed that manually computing the answers to the above examples isn't that difficult. The power of this programming language is that in can handle longer programs like the one below which become exponentially more challenging to compute by hand.

The expected output is

0.4 Language Concepts

A program in our language represents a series of events to help organize rides amongst students. A brand new program with nothing in it represents an empty $inputSchedule$. An $inputSchedule$ can contain two kinds of events: a $request$ or an $offer$. In order to write a $request$, the user must write a line in the format below:

Request: *name* to *locationList* at *timeRange*.

The string entered in the place of *name* will be the name by which the request is identified. *locationList* can either contain a single location or a list of locations which the person submitting this request is willing to be dropped off at. *timeRange* is the time and date (or times and dates) at which the person requesting the ride would like to leave on their ride to any of their given locations. An example of a $timeRange$ would be:

at 2 PM on 1/5/2023 to 6 PM on 1/5/2023

If the above $timeRange$ were included in a request, it would signify that the person requesting the ride would be willing to be picked up from 2 PM to 6 PM on January 5th, 2023.

In addition to a request, the other object accepted in the $inputSchedule$ is an $offer$ object. The $offer$ constructor follows the following format:

Offer: *name* to *locations* at *timeRange* with *seatCount seatString*.

Similar to the request constructor, *name* is the string by which the offer is identified, and *location* is the place the offering driver is going. A $timeRange$ in the offer constructor represents the times when a driver is offering to depart for a location.

The primary distinction between the request and offer constructor are the *seatCount* and *seatString* parameters. These two parameters are present in the offer constructor only in order to provide the number of people the person offering a ride is willing to take, with $seatString$ being "seat" or "seats" depending on value of $seatCount$. An offer where the driver is only willing to take one other person would end with the following combination of $seatCount$ and $seatString$:

...with 1 seat.

While an offer to drive more than one person would contain the number of available seats and the word "seats"

...with 3 seats.

We then use these requests and offers to organize carpools. While there is no guarantee that all requests will be satisfied, we can guarantee that the maximum possible number of requests will be satisfied in the output given the program's constraints. The outputs will be organized by those requests which were satisfied, and those which were not. First there will be a line which reads "MATCHED REQUESTS:", below which all of the requests in the given program which are satisfied by the offers listed in the program. Below those requests, there is a line which reads "UNMATCHED REQUESTS". Beneath that line, there will be all of the requests which are unable to be serviced given the offers of the input program.

0.5 Syntax

A complete BNF for programs written in our language. Anywhere there is white space, one or more white space characters are allowed.

0.6 Semantics

0.6.1 BNF Element Semantics and ADT's

- $letter$ is a primitive. We represent letters using the F# data type.
- $digit$ is another primitive in our language. We represent a digit by using the F# data type to store the values from 0 - 9.
- $year$ is a primitive in our language used in the date type. $year$ can only be one value, the 2023.
- day uses the F# type to represent the day a request or offer is being made to. day can only be an integer between 1 and 31 inclusive, as any value within that range should be a valid day of a month.
- $seatStr$ is an F# which can either be "seat" or "seats" depending on the value of $seatNum$.
- $seatNum$ is an F# made of 1 or more digits which represents the number of seats available in an offer.
- $monthNum$ is an F# representation of the month for a specific $date$ that a request/offer is valid for.
- $date$ represents the full date of the offer/request by taking the previously mentioned $monthNum$, day, and $year$. It is not specifically stored by any ADT in our parser but is stored later on in $time$ with other types.
- $hourStr$ is an F# which can only either be "AM" or "PM", and is used in the $hour$ constructor to indicate the time at which the offer/request is relevant to.
- $hourNum$ is an F# between 1 and 12 inclusive which is used in the $hour$ constructor to indicate the time at which the offer/request is relevant to.
- $hour$ represents the hour at which at an offer/request is valid by combining an $hourNum$ and an $hourStr$.
- $location$ is an F# which takes one or more $letter$'s to create the name of where an offer/request goes.
- $locationList$ is an F# which contains one or more $location$ types. It represents a singular location or multiple locations, which accounts for the scenario during which someone is willing to be dropped off at different locations.
- $name$ is an F# which takes one or more $letter$'s to create the name of the individual either requesting a ride or offering a ride to someone else.
- $time$ is an F# type which combines the previously mentioned $hour$ object as well as a $date$ object to give the full time at which an individual is willing to be dropped off/ pick someone else up.
- $timeRange$ is an ADT defined below which takes either a singular time object, or can take two time objects in the case that someone is willing to be picked up between two times, or is willing to pick others up between two times. Its type signature is $DateTime \rightarrow DateTime \rightarrow TimeRange$.

- $request$ is a constructor which takes a $name$, $locationList$, and $timeRange$ to create a record of a request for a ride being made. It is a constructor for the $event$ ADT defined below. Its type signature is $name \rightarrow locationList \rightarrow timeRange \rightarrow Event$.
- $offer$ is a constructor which takes a $name$, $location$, $timeRange$, $seatNum$, and $seatStr$ in order to create record of an offer being made. It is a constructor for the $event$ ADT defined below. Its type signature is $name \rightarrow location \rightarrow timeRange \rightarrow seatNum \rightarrow Event$.
- an $event$ is the main ADT of the entire program. It can either be constructed via an $offer$ or a $request$. Its ADT definition is below:

- $inputSchedule$ is an and is the backbone of the programming language. An input schedule can either be one $event$, or an $event$ and an $inputSchedule$, which is to say, multiple events. Its ADT is defined as follows:

0.6.2 AST For Example Program One

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 width wise.

[(inputSchedule) [(event)+ [(offer) [(name, location, timeRange, seatNum, seatStr) [(name) [Jacob]] [(location) [NYC]] [(timeRange) [(time, time) [(hour, date) [(hourNum)(hou

0.6.3 AST For Example Program Two

For the second program we have

```
{request} {name, location, timeRange} {name}{Oscar} ] {locationList} {location} {NYC} ] ] {timeRange} [{time} {hour, date} {hourNum}{hourStr} {9} [AM] ] {monthNum, day, year} [11] [5] [2023] ] ] {hour, date} {hourNum}{hourStr} [5] [PM] ] {monthNum, day, year} [11] [5] [2023] ] ] ] ] ] ] ]
{request} {name, location, timeRange} {name}{Oscar} ] {locationList} {location} {Boston} ] {timeRange} [{time} {hour, date} {hourNum}{hourStr} {12} [PM] ] {monthNum, day, year} [11] [26] [2023] ] ] ] ] ] ] {request} {name, location, timeRange} {name}{Oscar} ] {locationList} {location} {Boston} ] {timeRange} [{time} {hour, date} {hourNum}{hourStr} {12} [PM] ] {monthNum, day, year} [11] [26] [2023] ] ] ] ] ] ]
```

0.6.4 Evaluation For Example Program Four

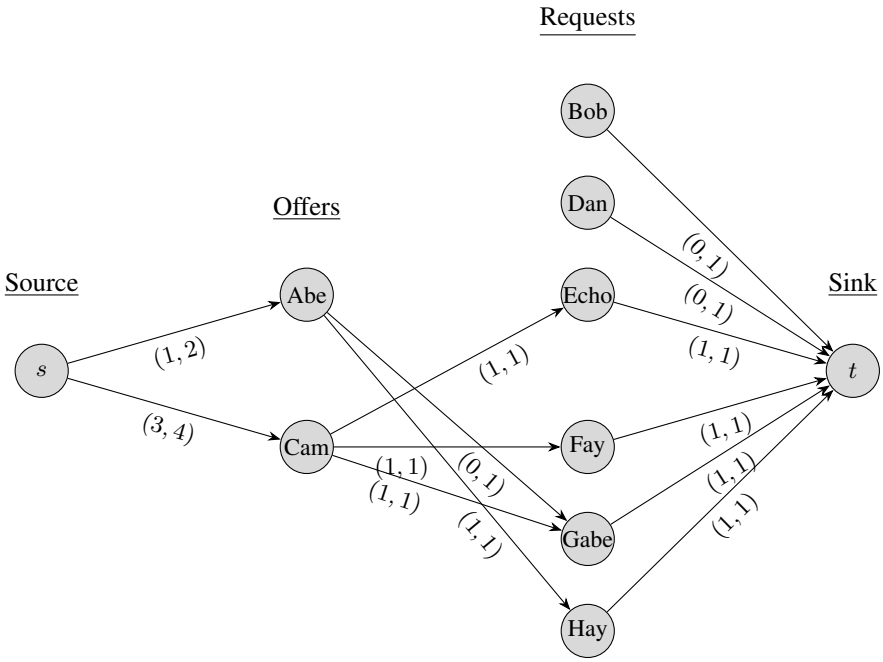
Let's consider the evaluation process for the fourth program reprinted here for convenience.

The parser will convert this program into an $inputSchedule$ which is an . The $inputSchedule$ passed to the evaluator would look like

The evaluator then takes these types and constructs a flow network with them as follows:

1. Add a source node s and sink node t
2. Add nodes and nodes
3. Add an edge from the source s to each offer with capacity of the offer's $seatNum$
4. Add an edge with capacity 1 from each offer to request if the offer satisfies the request
5. Add an edge from each request to the sink t with capacity 1.

For the program above, the flow network would look like the following where the each edge has $(flow, capacity)$ labels.



Then, the evaluator finds the max flow values that can be send across these edges given their capacities via the Ford-Fulkerson/Edmonds-Karp algorithm. The maximal flow and residual graph (represented by the $flow$ value) is the maximal number of requests that can be satisfied by the existing offers. Each unit of flow pairs one offer with one request. The evaluator then recovers the residual graph and converts it to a nicely formatted output like this:

Which would be our final evaluation.

0.6.5 The Benefits of Ford-Fulkerson/Edmonds-Karp

A major benefit of reducing this problem to a flow network rather than iterating over offers and requests to find matches is that Ford-Fulkerson can undo matches it's made previously. Consider example program five.

A naive solution may just match EchoOne and EchoTwo's request with Cam's offer since those appear first in the program and then put GabeOne and GabeTwo requests with Abe's offer. This leaves the remaining four requests EchoThree, EchoFour, Bob and Dan all unmatched. However, this is not the optimal solution. We could have done better by matching EchoThree and EchoFour with Cam and then EchoOne, EchoTwo, GabeOne, and GabeTwo with Abe, leaving the remaining 2 offers unmatched. The problem with the naive solution is that we can't undo a pairing we've made. Ford-Fulkerson, can undo a matching we've already made by sending flow backwards along a path, allowing us to undo a previous paring if we can make a better one.

0.7 Remaining Work

1. In the last edition of the specification, we include BNF types to allow for an additional date format where months are written alphabetically instead of numerically. Thus, programs could include dates written like instead of . We ended up not including this in the final implementation since we already allowed for one date format and wanted to focus our efforts on $Evaluator.fs$. Alternative date formats can be added to the parser and would not change any of the underlying AST types.

2. It's common for people asking for rides to offer snacks, gas money, or compensation. As we demonstrated with the other input fields in this PL, one could add a compensation field to offers/requests. For example maybe $Jacob$'s offer should only be paired with requests that pay for gas in the evaluated result. Given more time we'd like to add an option field for this in the PL.

3. F#'s built-in $DateTime.TryParse$ function is very forgiving. If it can parse but not interpret the date which we have given it defaults to . For example, if we were to write the following date as input to our program:

   at 65 PM on 9999/9999/9999

   Then the following is the date that $DateTime.TryParse$ would return after parsing that line:

   at 12 AM on 1/1/0001

   This means that given a crazy date, the program will still run, but will be incorrect, as offers may be matched to 12AM on 1/1/0001 which should not be matched at all. Therefore, as a future improvement, we would want to place more restrictions on what dates can be inputted, so that the F# $DateTime.TryParse$ does not return any odd dates.