CPSC 311 – Term Project Proposal

Expression to Diagram (ETD language)

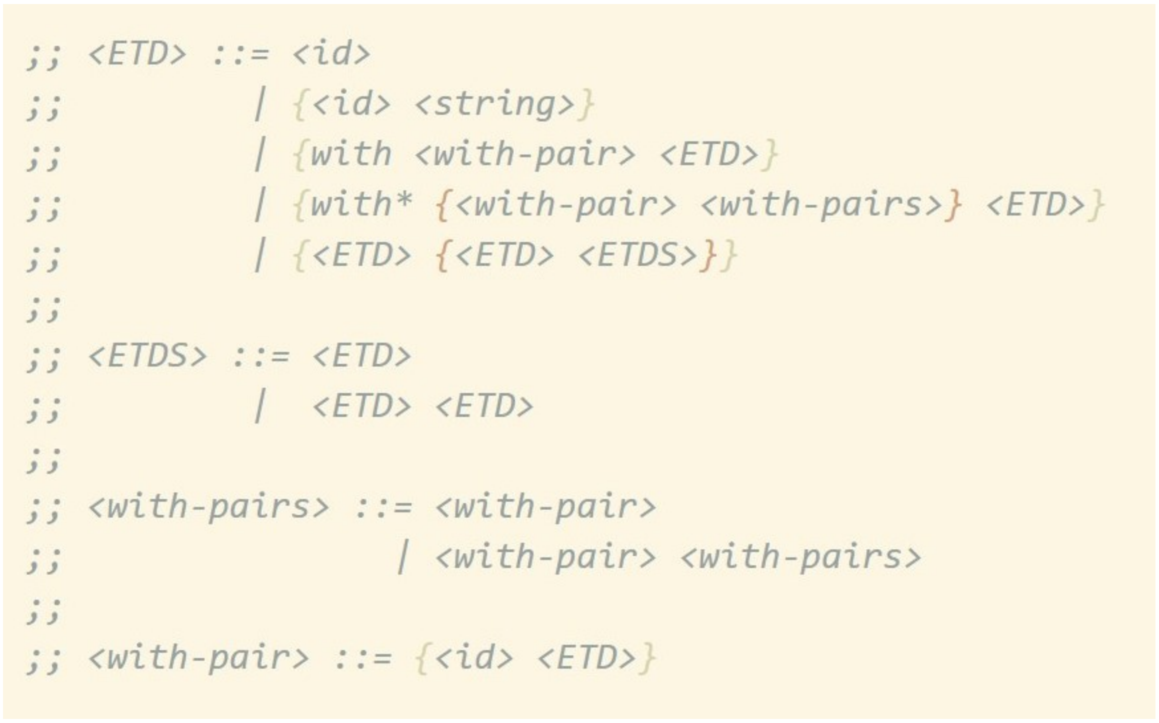
Name	Student #	CS ID	E-Mail
Zhe Li	88792486	d3q1b	lizhe918@alumni.ubc.ca
Maxwell Yang	96245428	r0g2b	shukany@alumni.ubc.ca
Eric Yan	22548424	g5h2b	qfyan@alumni.ubc.ca
Yifei Wang	26530162	w9a2b	yifei.wang.27@gmail.com
Hao Wang	97951503	x9e2b	h.wang.81@alumni.ubc.ca

Introduction

The Expression to Diagram (or simply ETD pronounced “Ed”) is a domain specific language designed to autonomously generate different types of diagrams such as circuits and binary trees from user-provided logical expressions. To elaborate, the user can enter a logical expression in a concrete syntax similar to the Racket language into a Racket file. The ETD will then parse the given expression into an abstract syntax and proceed to interpret it into a corresponding diagram. We opted to use Racket because we think making diagram is a very functional stuff and it is better to choose a language that we are familiar with. The primary motivation behind the development of this language is to provide a clear visual representation of unfamiliar academic concepts for people. The language may also be used by course staff to help the development of new exam/homework questions, as well as solutions to some problems like logical circuit design and binary tree problems.

Project Goals

Our core goal is for the language to be able to convert simple expressions that have only tree structures (without loops) into their corresponding diagram. It is easy to think about them as a recursive structures and process them recursively. Actually, we think the EBNF of this language should be as simple as the Figure. 1 below. The id represents the type of a node such as and-gate and tree-node. It also supports the with expression that allows us to avoid very long duplicated expressions.



```
;; <ETD> ::= <id>
;;         | {<id> <string>}
;;         | {with <with-pair> <ETD>}
;;         | {with* {<with-pair> <with-pairs>} <ETD>}
;;         | {<ETD> {<ETD> <ETDS>}}
;;
;; <ETDS> ::= <ETD>
;;         |  <ETD> <ETD>
;;
;; <with-pairs> ::= <with-pair>
;;                | <with-pair> <with-pairs>
;;
;; <with-pair> ::= {<id> <ETD>}
```

Figure 1. The EBNF of ETD

For our full goal, we want to generate complex diagrams with loops such as graph and circuit that use some outputs as inputs. It seems that it is still easy to just connect the nodes together, but we think it is very complex to leave spaces for these structures and show the user a clear diagram without overlapping or too much crosses. Also, it is not easy to design and deal with the expression for the diagrams with loops, even the normal logical expression does not support the circuit with loops, and we might need to do something like “parse by reference” to achieve this goal. Another idea that we want to achieve is to have different coloring for different lines to show the relationship between nodes which may also show the status such as signal on the line. We think this is not very hard but can be time consuming.

Milestones

To realize the language, we will split the project into multiple milestones. The first step in our strategy is to perform sufficient background research on relevant topics. The central research topic would be ideas of drawing diagrams in functional programming. This would involve researching about the data structure of diagram, how to draw things recursively, diagram generation, image creation for functional programming (or recursive algorithms). All group members will contribute to the research of these core topics above. Furthermore, as the Racket 2htdp/image library is the only way we use to draw on screen, we will likely dedicate 1-2 group members to specially focus on the documentations, tutorials and demos involving the use of the library. Some other topics that we might also conduct research on include syntax choices, impact in educational settings and possible visual designs etc. Once we have completed most of the research (1 week before background research report submission), we will compile our results into a single background research document highlighting our research on the topic.

The next milestone of our project is the proof of concept and plan. At this stage, we will have clarified our "minimal" core goals and provided some of our more "ambitious" full goals. To demonstrate our capability for accomplishing the core goals, we will elaborate on how to achieve some key components of the project by providing written explanation, diagrams and code snippets. Some of the interesting parts that currently comes to us includes how the parser will be written, how each node is defined and stored, how we might recursively draw the diagram. We will also provide possible solutions to achieve the more "ambitious" full goals. They will be in a similar format as the minimal goals. Considering the importance of achieving the core goals first, all members in the group will initially collaborate on providing methods to achieve them. We will also attempt to build a language that supports most of the features that we specified in the main goals. After completing this phase, we will then move on to the full goals.

For the poster session, we will make a poster that illustrates how our project works. We will show the several types of diagrams generated by our project and describe step by step on how we approach the final version. The first part of the poster is parsing and desugaring. We won't spend much time on this part since the parsing tool we use is the same as it was taught in the lecture. The next part is interpretation. We will give examples on how we convert the expression to a special data structure State. This data structure is used to record all the print stuff in the lecture, but here we will store the location of nodes and relationships between them. The last part will be rendering. If the State stores a string, it can be printed with a very simple expression, but what we have is information of a diagram, so what we need is to draw every node and line depend on their types on the screen. Our poster will mostly be composed by the diagrams such as logic circuits and a flow chart of how we approach the final project. We will try to make our poster presentation clear and interesting.

The concluding milestone of our project is the actual implementation of our language. The parse step of the language will use the classical "match" method provided in Racket to parse the input instead of inputting a .txt file. This is because the latter might require extra time researching on topics such as I/O streams and parser generators. However, the majority of our effort will be spent on the interpreter, which is supposed to output images of the corresponding logical circuit. Having completed our background research and proof of concept, we imagine our actual implementation will not differ too much from the plans we laid in the earlier steps. The more challenging aspects of our core goals like the wiring and position of gates and inputs should also be mostly resolved by this point. Moreover, we will add the implementation for some of our full goals depending on the circumstances. We anticipate the final ETD language to be practically usable and helpful in real-life settings.

Literature Review

In the article about graphical data flow programming language (Hunt, 1990), the author created a language that can generate flow charts including a full set of data types and control flow operators. Similar to logic circuit, the language of data flow chart involves distinct definition of each operator and the way to connect them together. We can obtain some useful design concepts from this language.

Another article (Ren, 2002) is about the language of generating abstract syntax trees. The author introduced the design and implementation of the parser and implementation of the tree generator. The design of the language was beginning with the general architecture of the language, and then the author introduced the implementation of the parser detailed with examples. He also mentioned some important issues to consider about the parser such as supporting grammar, which gives us some inspiration for designing our own language. The author also provided source codes of this language, which is considered to be a good reference.

References

Hunt, N. (1990, November). IDF: A graphical data flow programming language for image processing and computer vision. In 1990 IEEE International Conference on Systems, Man, and Cybernetics Conference Proceedings (pp. 351-360). IEEE.

Ren, C. L. (2002). Parsing and abstract syntax tree generation in the GIPSY system (Doctoral dissertation, Concordia University).

(n.d.). Retrieved from https://docs.racket-lang.org/teachpack/2htdpimage.html?q=2htdp.

(n.d.). Retrieved from https://futtetennismo.me/posts/algorithms-and-data-structures/2017-12-08-functional-graphs.html.