Global project definition Marama ADT transpiler

1 TOC

2 Goal

Build a transpiler that is capable of translating algorithmic data types in clean to 3D shapes for the marama program.

3 Requirements

- The output of the transpiler should be easily importable into any 3D game engine.
- The input translation configuration should be readable and changeable by humans, so it is easy to change the output and add extra definitions.

4 Input

A line of clean code. Like example 1.

```
ADTDset =
1 ::WeekendDay = Sat | Sun
2 ::Bool = True | False
3 ::List a = Cons a (List a) | Nil
```

Example 1: algorithmic data types in clean (Cons = :: which is : in haskell)

4.0.1 Discussion

- Chide: please use prefix notation, so Cons instead of ::, and Nil instead of []. That is more elementary, and will make writing your first parser easier. :: and [] are `syntantic sugar', can be added later.

And definitions for the shapes of the output. Containing the shape of every primitive data type (not containing other data types) and the zoom factors for complex data types (containing other data types).

```
________________________
XAML Example
------------------------------------

<Faces>
        <Bool>
        <Shape>
                "triangle"
        </Shape>
        <LineThickness>
                2
        </LineThickness>
        </Bool>
        <List>
        <Shape>
                "circle"
        </Shape>
```

```
       <LineThickness>
              2
       </LineThickness>
       <InnerFace>
              <Reference>
              Bool
              </Reference>
              <ZoomingFactor>
              0.8
              </ZoomingFactor>
       </InnerFace>
       </List>
</Faces>


________________________
Proprietary Example
------------------------------------

Bool {"triangle", 2}
List {"circle", 2, {Bool, 0.8}}


________________________
JSON Example
------------------------------------

Faces {
       "Bool": {
              "Shape": "triangle",
              "LineThickness": 2,
       },
       "List": {
              "Shape": "circle",
              "LineThickness": 2,
              "InnerShape": {
                     "Reference": "Bool",
                     “ZoomingFactor": 0.8
              }
       }
}
```

Example 2: Some ways of storing the information compared

4.1 Discussion

- Douwe: I would go for JSON, because (as seen in this comparison) it uses far less space compared to XAML and it is easier to read for humans than the proprietary version I thought of. The proprietary version is a version which uses the least space possible, as seen here it makes it very hard to get the meaning of the supplied information, it is just the JSON version stripped of the naming.
- Chide: agreed, JSON it is.

4.2 Translation configuration file format

4.2.1 TypeConstructorFaces

- Shape: (SVG like paths, based on the svg path element)
    - M (x, y) (move to, starting point or gaps in the shape)
    - L (x, y) (straight line)
    - C (x1, y1, x2, y2, xe, ye) (a cubic curve from the last point with control points x1,y1 and x2,y2 to point xe,ye)
    - Z () (Closes the last shape (goes to last M))
    - ZF () (Closes the last shape and fills it)
    - No capitalisation requirements (internal everything is handled equally)
    - Spacing is needed between two points (2020 is not the same as 20 20) but not necessary between a digit and a alphabetic character ([MLCZ]).
    - Commas are only for readability
    - The total space on a face is 100x100 (so 0 to 100 in the x and y axis)
    - All the coordinates are interpreted as doubles
    - Example: "M10,10 L10,90 L90,90 L90,10 Z" (a rectangle)
    - Example: "M10 10 C20 20, 40 20, 50 10 C30 30, 50 30, 20 20 Z
- LineThickness:
    - Double
    - It is the thickness of the lines of the shape
- Filled:
    - Boolean
    - If the inner part of the shape is part of the shape or not (depending on the face being male or female the internal is filled or not filled, mainly for polymorphic types) (defaults to false)
    - Maybe this should be able to be set per sub shape (M to Z) this would give more granular control over the end result, I propose ending with ZF to fill a shape and Z to close it without filling.
- VerticalJointSize:
    - Double
    - How deep the shape is “etched” into (Female) or pulled out (Male) of the zeropoint.
- Variables:
    - List of inner shapes (only for types with variables)
    - Place:
        - Double, double
        - This is the placing of the referenced shape (top left corner of the referenced shape) so the referenced shape will be draw relative to this point.
    - ScalingFactor:
        - Double
        - The zooming factor for drawing the referenced shape. Eg 0.5 gives the shape at half its normal width and height.

4.2.1.1 A list of all possible fields

- Shape
- LineThickness
- Filled (depracted?)
- VerticalJointSize
- Variables

4.2.2 ConstructorBlocks

- Shape
- Argument location(s)
- Resulttype location

Douwe:How should the haskell code (transpiler) know which constructorblock to take? If it is required to pass this as an argument the code has not much to do.
Chide: What do you mean? The article specified this: it is a map between a constructor and a constructor block. E.g. “Cons” is translated into the constructor-block that is associated with “Cons”.
Douwe: so for example: Bool Cons List Bool, would only need the Cons constructor block? For all its faces, input and output/result? So this constructor block has to hold the information to map all the applicable typefaces to the correct place. And the constructor block only holds the placement, not the actual typeface, am I right?

- C: Indeed (but please use the word `jointLocation’, also see below).

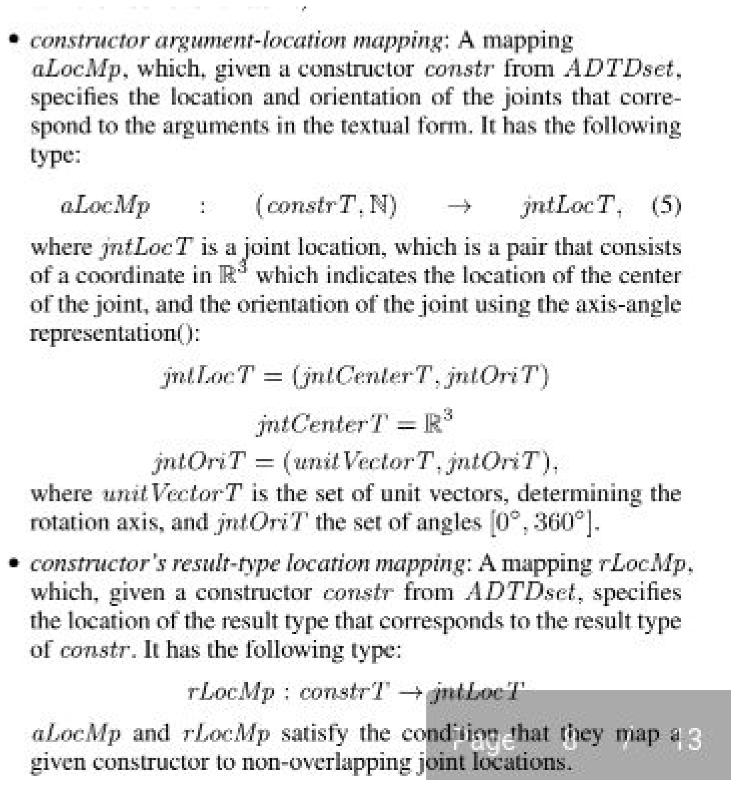
So for example:

```
“Cons”: {
        “Shape”: {“H”: 120, “W”: 120, “D”: 200}, -- Could be more advanced
        “JointLocations”: {
                “Result”: { “Orientation”: {“Angle”:π/2,”UnitVector”:{0,1,0}},
                        “JointCenter”: {10,10,10}},
                “LeftHand”: { “Orientation”: {“Angle”:π,”UnitVector”:{1,0,1}},
                        “JointCenter”: {100,10,20}},
                “RightHand”: { “Orientation”: {“Angle”:π/4,”UnitVector”:{0,0.5,0.5}},
                        “JointCenter”: {10,10,100}}
        }
}
```

- Thijs: I do not quite understand what ‘result’, ‘lefthand’ and ‘righthand’ mean, could you give an example of this json for figure 7 in the visupol paper?

Douwe: How should these things be called? Or should it be a list of JointLocations? Or maybe the names of the specific JointLocation to make it easier to reference it later.
Douwe: You asked to number them which essentially makes it a list. I will rethink if it makes sense to name the joints when the rest is somewhat clearer.
Chide: Good to hear that things became clearer, and your attempt shows you are well on your way of understanding it all! However, please see the article because some things have to be added (I’m aware you left some things out in the above example, but it cannot hurt to

indicate what the next steps are), and, as requested before, please do this (!!). In other words, given the relevant part of the article:



- *constructor argument-location mapping*: A mapping $aLocMp$, which, given a constructor $constr$ from $ADTDset$, specifies the location and orientation of the joints that correspond to the arguments in the textual form. It has the following type:

$$aLocMp \quad : \quad (constrT, \mathbb{N}) \quad \rightarrow \quad jntLocT, \quad (5)$$

where $jntLocT$ is a joint location, which is a pair that consists of a coordinate in $\mathbb{R}^3$ which indicates the location of the center of the joint, and the orientation of the joint using the axis-angle representation():

$$jntLocT = (jntCenterT, jntOriT)$$

$$jntCenterT = \mathbb{R}^3$$

$$jntOriT = (unitVectorT, jntOriT),$$

where $unitVectorT$ is the set of unit vectors, determining the rotation axis, and $jntOriT$ the set of angles $[0°, 360°]$.

- *constructor's result-type location mapping*: A mapping $rLocMp$, which, given a constructor $constr$ from $ADTDset$, specifies the location of the result type that corresponds to the result type of $constr$. It has the following type:

$$rLocMp : constrT \rightarrow jntLocT$$

$aLocMp$ and $rLocMp$ satisfy the condition that they map a given constructor to non-overlapping joint locations.

Page 6 / 13

Make the following adaptations to your designs:

- Instead of ‘TypeFaces’, use ‘jntLocs’ (or ‘jointLocations’) (joint locations)
- Instead of ‘Face’ use ‘jntLoc’ (or ‘jointLocation’) (joint location).
- Instead of ‘left hand’ and ‘right hand’, use a natural number (1, 2, 3, etc.).
- Add joint orientation. Or use another way to express all information needed to fully specify a joint location. See https://en.wikipedia.org/wiki/Axis%E2%80%93angle_representation.

C: The fragment from the paper also suggests the names: ResultLocation (rLocMap), and ArgumentLocations (aLocMap). So, you could represent this information as a list of ArgumentLocations (indeed as you suggest, instead of numbers) and one ResultLocation.

4.3 Discussion

- Chide: could you use as much as possible names the same or similar to those used in the definition of the translation configuration in the article in which it is defined?: https://drive.google.com/file/d/0BwvE2Xxofa7HUm5mTVZqRXhabGM/view?usp=sharing

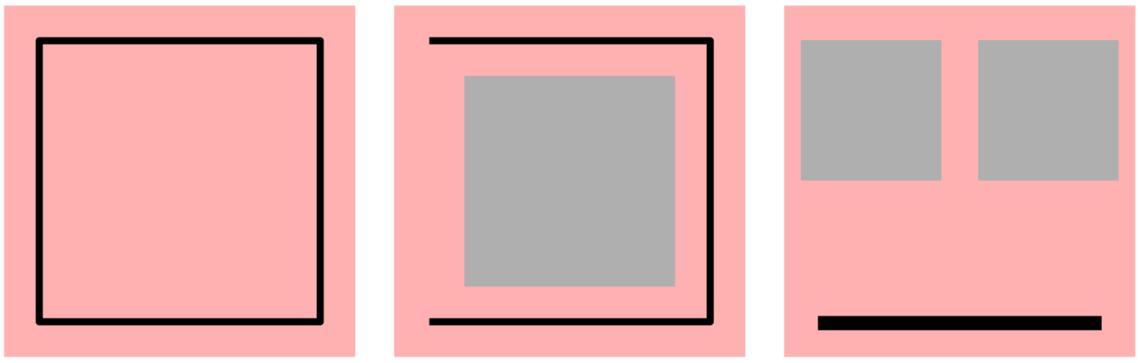
  Then it is easier to see what corresponds with what in the article (see Definition 5). (In fact, you are translating definition 5 into a file format. Every piece of information expressed in definition 5 should become part of the file, of course adapted to the new context, a haskell transpiler and a file that is also reasonably “human” readable.)
  So, for example, instead of “Faces”, use “TypeConstructorFaces”.
  And: “ScalingFactor” instead of “ZoomingFactor”.

And: an item “ConstructorBlocks” should be added which specifies the form of each constructor block (see “constructor block mapping” in the second column of page 7). etc.

- Chide: please, also add the original ADTD definitions. (see “ADTDset” in 2nd column of page 7 of the previous article).
- Chide: in general you really `got’ the gist, I’m happy with your progress.
- Douwe: the translation configuration parser I wrote is able to consume nearly all of the proposed example, the only thing it is not able to parse is lists or objects as value of an element (for example the variables in List). Do you have any ideas how to tackle this problem?
- C: You mean your current implementation produces errors, or did you not yet write the code to parse these lists?
- D: I did try it, but it produced type errors. I wanted to have the possibility of a list or an object or a string. I tried creating a new datatype “value” which could hold all those different options, but it did not work out well (I think it was just me not knowing how to do it).

4.4 Example translation configuration file

```
“TypeConstructorFaces”: {
        “Bool”: {
                “Shape”: “M10 10 L10 90 L90 90 L90 10Z”,
                “LineThickness”: 2,
                “VerticalJointSize”: 10
        },
        “List”: {
                “Shape”: “M10 10 L10 90 L90 90 L90 10”,
                “LineThickness”: 2,
                “VerticalJointSize”: 10,
                “Variables”: [ {“Place”: {20, 20}, “ZoomingFactor”: 0.6} ]
        },
        “Tuple”: {
                “Shape”: “M10 90 L90 90”,
                “LineThickness”: 4,
                “VerticalJointSize”: 10,
                “Variables”: [
                        {“Place”: {5, 10},
                        “ZoomingFactor”: 0.4 },
                        {“Place”: {55, 10},
                        “ZoomingFactor”: 0.4}]
        }
}
```



The faces of the above example of a translation configuration. The first face is the “Bool” type, the second the “List” and the third the “Tuple”.

5 Output

3D objects defined by the input. There are many possible file types (http://edutechwiki.unige.ch/en/3D_file_format). The exact file type could be defined later but it would help in building the system to know where to head to. Maybe .Obj would be a possibility, because it is a universally accepted open file type, many programs are able to handle it.
But I don’t know if it is possible to make moving things with the .obj file type. Also I am not entirely sure the output file type needs to support moving, because it should only output the 3d shape of the datatype in clean.

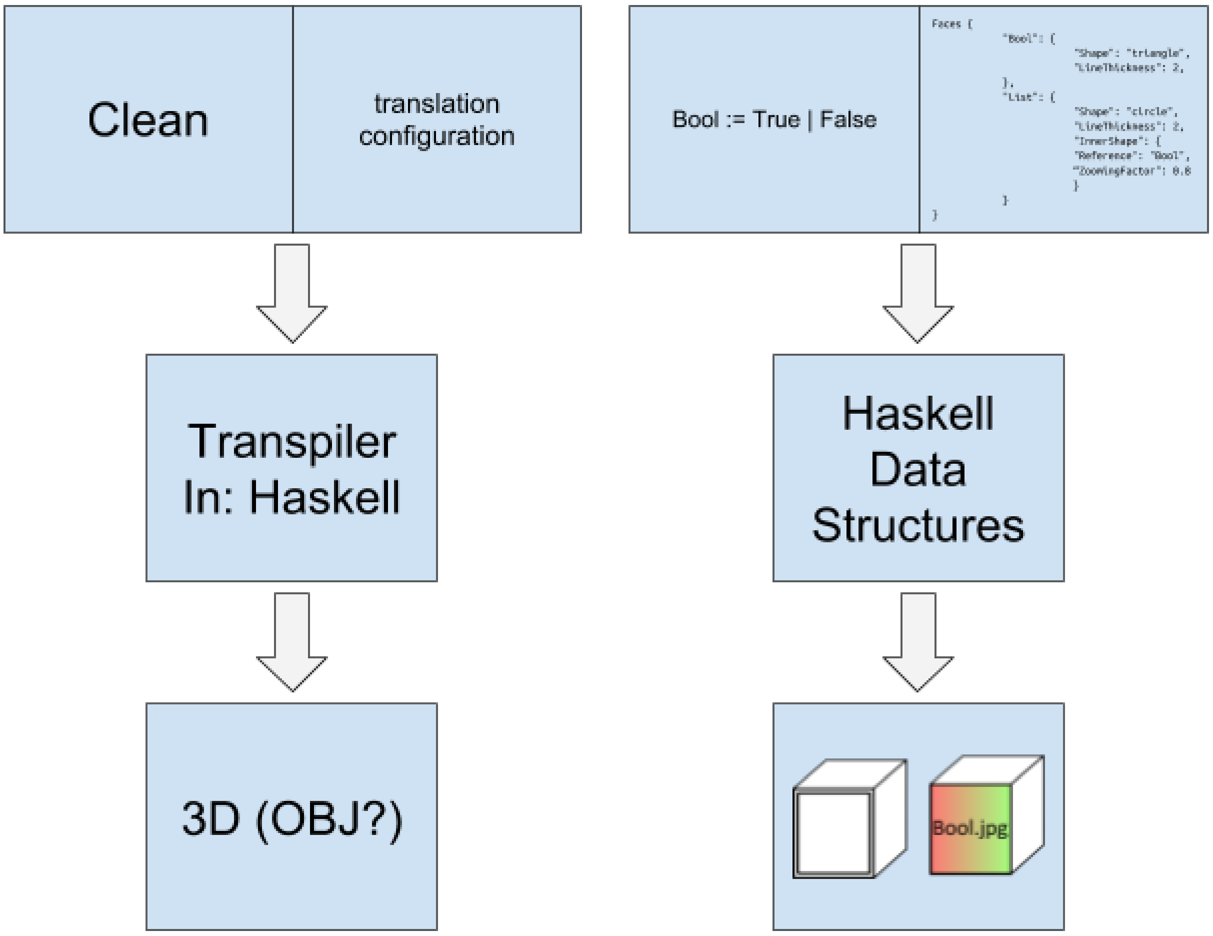
Discussion

- Thijs (Member of the Windesheim group): Douwe, have you thought about what happens when multiple 3D-models or elements are needed for the maramification? Will you output a single .obj with both elements? If yes, how are we supposed to strip them from each other? (It might be an idea to output multiple obj files) I realised this might be an issue after reading this document if you look at figure 4 on page 4, the ball and the tubes are separate 3D-models.
- Douwe: I totally agree. The .obj file format gives support for easy splitting of objects, but this makes it harder for the transpiler and the “game” so I will not do this.

A module which is able to read .OBJ files
https://hackage.haskell.org/package/WaveFront
https://hackage.haskell.org/package/obj

6 Flow diagram

Just a simple action of the program: turning the Bool datatype into a 3D object.



```
Faces {
        "Bool": {
                "Shape": "triangle",
                "LineThickness": 2,
        },
        "List": {
                "Shape": "circle",
                "LineThickness": 2,
                "InnerShape": {
                "Reference": "Bool",
               "ZoomingFactor": 0.8
               }
        }
}
```

7 Implementation

7.1 Which libraries and languages for parser construction

- Chide (02.19.2018): (In the meanwhile I have regained access to the bitbucket repo) see that you have copied the parser combinator library from the Jeroen Fokker paper, and that things now seem to become clearer to you. Good!
  Since there are already parser combinator libraries in Clean that are much more similar to the latter paper, and because I'm already writing another Clean parser in Clean (note the reflectivity of this ;-)) -- I propose to port your code to Clean as well and drop Haskell for now. An additional benefit is that the group in Nijmegen with

whom I collaborate closely, will highly appreciate this, and we can consult them for technical questions about the parser libraries/the Clean language.
(In some later stage we can port the whole project also to Haskell, so that both languages are covered. The advantage of Haskell of course is that it is well-known. Moreover, I read that its parser combinator library is quicker than that of Clean.)

- Chide: Note that Clean is very similar to Haskell, so essentially everything you have learnt transfers directly to Clean.
- Douwe: Okay… I will have to install clean first and check all my code before submitting it. It will be on my to do list for a day with some time.

Parser technology proposed: Clean programming language + ZParsers library
C: In the meanwhile I have contacted the developer of an extensive parser combinator library (Erik Zuurbier), an interesting and very helpful person. It is the library ZParsers in Clean. Therefore I've changed my proposal to ZParsers instead. To port to ZParsers, you need to do some tweaking, because it works a bit differently than default parser combinators. For the time being I think it is better to first finish the first completely working acceptance parser (so only consuming, no transformation yet) in Haskell, and then port that to ZParsers.
D: Sounds okay to me, I have not done anything in Clean yet, but read somethings about it and concluded it was very much like haskell.

7.2 Parsing algebraic data structures

7.2.1 Reflections on current partial implementations

Code produced by Douwe:

```
-- The clean algebraic data structures parser
-- Douwe Schulte 12-2017

---------------------------------------------
-- AlgebraicDataStructure
---------------------------------------------

data AlgebraicDataStructure = ADT Type Constructor

instance Show AlgebraicDataStructure where
    show ( ADT s c ) = show s ++ " = " ++ show c



---------------------------------------------
-- Type
```

```
---------------------------------------------

data Type = Type String [String]    -- The name and the variables
names

instance Show Type where
    show ( Type n [] ) = n
    show ( Type n vars ) = n ++ (foldl append "" vars)
        where
            append::String -> String -> String
            append s s2 = s ++ " " ++ s2



---------------------------------------------
-- Constructor
---------------------------------------------

data Constructor = Value String     -- The fundamental building
block, it contains a possible value for the constructor (like
"False" or "True")
    | Constructor :|: Constructor   -- To give another posibility
   | Constructor :-: Constructor   -- To stitch two Values together
   | Constructor :+: Constructor   -- To Stitch a typevariable to a type
```

- Chide: I'm quite impressed that you found a way to create quite sensible code, while you are at it for the first time! A few remarks about the code above:
  - You are covering algebraic data *type definitions*, while what you need to cover at this stage are algebraic data *structures*. The latter are the values of the algebraic data types. I.e. you need to parse "(Cons True (Cons False Nil))", but you do not (yet) have to be able to parse "::List a = Cons a (List a) | Nil".
  - Try to follow the naming of algebraic data structures as defined in the clean language report as much as possible (did you already install clean? It includes the clean language report).
  - Do not hesitate to look at the source code of the Clean compiler itself - it will also contain a parser to read in Clean code. This may also give you some ideas.
  - In the code above, I think you are deviating too far from the original structure, by inventing your own infix operators. Probably it is easier to keep things close to the original representation. For example:
    ```
    data AlgebraicDataStructure = ADS String
    [AlgebraicDataStrucure] -- The first argument is the
    identifier of the constructor, the list contains its
    arguments.
    ```

    Note that is a recursive definition, needed because algebraic data structures indeed have a recursive structure!

- Chide: I still miss code defining a parser using parser combinators in Haskell. Please, start with this ASAP, otherwise you won't get feeling for this essential part of the project. You can just write an `acceptance' parser - one that does not do any transformation yet (so you do no have to read them into the algebraic data structures you have defined to hold the things you parsed). A very clear introduction into parser combinators is : http://www.staff.science.uu.nl/~fokke101/article/parsers/index.html
  Not in Haskell, but the examples should be easily translatable into Haskell.
- Douwe: I build a parser for algebraic data types.
  it is very simple, it just structures the data type into a tuple, which should make it usable in following code.

  Example output
  ```
  > adtd "Bool = True | False"
  ("Bool",[],["True ","False"])
  > adtd "Thing a b = Help a b | Anotherthing a"
  ("Thing",["a","b"],["Help a b ","Anotherthing a"])
  ```

  So in general it gives (Name, [variables], [constructors])
  Do you think this will work? Or should I create data types to contain this tuple?
  This same question also goes for the translation configuration file parser.
- Chide: always the latter, it is good practice to parse things into customised datatypes. That way the compiler can help you discover errors in your program, and the program will become much more legible. This prevents a “tuple and list” hell, where what everything means is very difficult to see and the compiler can also not make sense of it… ;-)
    - Chide: is it clear to you what I mean with this? THe start would be that you define an algebraic datatype with the name AlgebraicDatatype (or similar) and define it to hold information about algebraic datatypes (fascinating - you probably note the reflectivity in this - this is typical for compiler construction)?
- Chide: a meta-request: can you place your points in the locations in this documents that fit its topic the best (that is why I moved it)? The document grows large, and doing this will become a tremendous help for future reference by us or others joining the project. Everything thought, created and discussed about a topic is then in the same place.
-

```
-------------------------------------
-- This is the actual usefull code --
-------------------------------------
type Name = String
type Variables = [String]
type Definitions = [[String]]

type ADTD = (Name,Variables,Definitions)

adtd:: String -> ADTD
adtd t = snd(head(p_adtd t))

p_adtd = p_name <+> white(p_variables) <+> white (token "=") +> p_definitions <@ f
    where f (a,(b,c)) = (a,b,c)

p_name = many1 (satisfy(/= ' '))

p_variables = many( p_name <+ token " ")

p_definitions = just (dividedBy "|" p_definition)
p_definition = dividedByWhite " " (white(many(dontsatisfy (`elem` " |"))))

-----------------------
-- testing variables --
-----------------------

test = "Bool = True | False"
test2 = "WeekDay = Sat | Sun"
test3 = "List a = Cons a (List a) | Nil"
test4 = "Thing a b = Help a b | Anotherthing a"
-- Output: ("Thing",["a","b"],[["Help","a","b",""],["Anotherthing","a"]])
```

- Chide: Looks very good! A few remarks:
    - What is the empty string doing in the parse result of “Help” (third argument)?
    - You still not have followed up on this.
    - You can now start doing this.

7.3 Parsing the Translation Configuration into Haskell Data Structures

7.3.1 Discussion

- Chide: can you also start doing this? So, define the algebraic data types in Haskell to hold the translation config information?

Discussion

- Chide: I notice that you you have a copy of the same code twice, in Algebraic_data and in (see commit id ea450ab06abb9b9c84f6b9ec42a6582b8522eb6b). It is better to make use of imports.

8 Questions that need an answer

V = done

1. V What are the inputs
2. V What is the output (which filetype)

9 Discussion

- Wilbert: As we are performing a small research about game-engines, we see that .obj is the most widely supported format, but not the most complete one. .fbx and .dae (collada) are the more common formats for storing more data than just a shape. .fbx is a file format owned by Autodesk, whereas .dae is open source and managed by a non profit organisation called the Khronos Group. This does affect the game-engine choice because both these formats are not supported by our initial choice called jMonkeyEngine (Java based). We have found another open source engine called Godot (C++ based) that does support the .dae format. This research is still being made though. What information besides the geometry does need to come out of the transpiler?
    - Douwe: I think the only thing needed to come out of the parser is the geometry. So any format will do as long as it is capable of storing the geometry (including rather difficult shapes, not only rectangles/boxes). Also the .obj was chosen because it is used so much and a friend of mine said it was a good choice, but nothing is really written yet that is specific to any output format (19-04-2018).
    - Wilbert: That clears a lot up. .obj is a format that is support by every engine we came across. If it is okay with you, we will continue our research only taking .obj into account.
    - Douwe: Seems fine to me.
    - Chide: probably more information needs to be expressed in the result of the transpiler (among other things typing information), I discussed this with the Windesheim team, but not yet with you, Douwe, so you could not have known this. However, I do not think that .obj is a problem. This information can also be stored in a custom-made intermediate file that expresses *all* information, which can be further parsed into plain .obj files that can be offered to a 3D engine.
    - Chide: also, I think that an even better approach is not to feed the transpiler-output directly into a 3D engine, but to first read it into your own designed datastructures in the main Scala program, and then communicate that result to the engine through its API (so not with files) - I do not know jMonkey, but I assume it works like this - probably you can call some API-functions and build up your 3D objects. Then you (Wilbert & team) have full control and can read in *any* format into the Scala program, because you

    are writing your own parser. Another option would be to first parse the intermediate format into plain .obj files and offer those to the engine, and then somehow relate the resulting 3D object that is then created by the engine to the richer intermediate datastructure.
    - Douwe: The information from the transpiler should get to the “game” part in one way or another and I think it would be nice to be able to use both programs seperated of each other. But maybe the idea is to use the transpiler as a part of the other programs (using a foreign function interface of some sort) but I haven’t heard about such plans. I am now focussing on building a stand-alone program which can transpile the input files to .obj (like) files. If there is a need for more information to be transferred, it is very easy to extend the .obj file format (possibly while maintaining .obj syntax so any program can read and show it). I can not think of any other information needed to be sent to the “game” only the full type (filename?) and maybe some metadata like author or something and the transpiler version. The object information has to be transferred and .obj has a small file size and is very simple to parse (and create) so maybe it is a good idea to use this structure (general idea) to transfer the spatial information, whether or not it is temporarily stored in a file or not. Is the Scala code involved in the transpiler, or for the “game”?
- Chide: I’m impressed by your methodological approach!
- Chide: Question about the diagram: shouldn’t the XAML be after Haskell, it is the Haskell transpiler that transforms it from Clean code to XAML objects. Moreover, I would rename the Haskell boxes to “Transpiler”, to make its intention clear.
- Chide: answer this: yes the 3D objects should be easy to import into a 3D (game) engine for moving objects. I suggest contacting:
    - Stephan van der Feest, he is a professional game developer, see http://virtualplay.zone/, address: stephan@virtualplay.com
      I know him personally, so just refer to me and pose your question.
    - Jesse Nortier, graphical 3D designer and artist: jessenortier@gmail.com
      He designed the Nanquanu brochure, so the same story as for Stephan!
    - Douwe: I asked a friend, he is very good at 3d work, what the best format would be. He proposed FBX or OBJ, FBX as first choice.
- Chide: moreover, you can already start writing a Haskell program that defines the data structures to read in the translation configuration as defined in [TODO add link to marama article here], and reads it from a plain text file. Hence, you also have to define a domain specific language for this plaintext file (which should be easy to understand for non-programmers as well). This is independent from the output format of your transpiler, as you also indicated yourself.
- Douwe: I was proposing the XAML as the plaintextfile for the configurations because XAML is used in many places (so known to many programmers) and it is a very readable document type for humans and easy to parse for computers.
- Chide: Aaah, now it is clear. You mean using XAML in relation to the *translation configuration*. Can you add this to the diagram (“Translation Configuration”), again this makes the intention clear.
- Douwe: clearly I did not make it as clear as possible. But what do you think of this approach? Your last comment does not answer the question, only gives a comment.

- Douwe: I thought of it and I will stick to JSON (Javascript object notation) because it uses way less space compared to XAML and is (I think) a little bit more readable for humans.
- Douwe: What is the precise Clean syntax this program will get?
    - Chide: see the article https://drive.google.com/open?id=0BwvE2Xxofa7HUm5mTVZqRXhabGM Note that it also defines the translation itself (see definition 16on page 10).
- Douwe: What do you think of the translation configuration file format I came up with?
- Chide: (my answers to this are throughout the document, you've already seen them.)
- Chide: important: try to use the information in the article as precisely and completely as possible. Most aspects are already defined in the article, such as the translation itself (so the core of the transpiler). Please, confirm this by placing your answer below.
- Douwe: I saw it yesterday and read it (about the transpiler) to check if I added everything that is necessary, I will try to keep as close to the article as possible.
- Chide: I believe we already spoke about this, but for constructing the parser (from clean + translation config to the 3D format), use so called "parser combinators", a lot of that can be found on the Web, and Haskell has dedicated libraries for creating parser combinators.

10 Links

Git: https://bitbucket.org/maramawebsite/clean-parser/src (team: Marama so Chide is part of it already)