Use case: guilang

Before you start...

If you didn't go through the panola tutorial yet, please do that first and then come back here. It is much more gentle paced.

What is guilang?

Guilang is a toy wysiwym (what-you-see-is-what-you-mean) user interface specification language for sclang. The idea is that you can simply draw your ui as a kind of ascii art and have the system turn it into a fully functioning user interface providing you with all the hooks to do something useful with it.

More specifically, from the same specification, the code derives all code required to generate the ui controls, as well as the code to generate a datamodel and the code to connect both together (where the ui automatically observes changes in the model, and the model is automatically updated by interaction with the ui).

The full code is available in the guilang.scd file, part of scparco's examples.

How is a guilang interface specified?

A guilang user interface is layed out as a list of rows. Each row is divided in cells. Cells are delimited by pipe ("|") characters. To merge two adjacent cells on the same row together, use a double pipe ||. As this is a toy language, there's no provision for merging two adjacent cells in the same column.

Here's an example of a guilang specification. It contains a section with rows, which specify the layout. Each row is delimited by (and). Within a row, columns are specified by drawing the cell borders using |. Inside a cell one can add a Control:name pair. Control must be a class of a control that exists in sclang (things like Slider, TextField, Knob, StaticText, Button, …). Name can by any valid identifier as defined in the `ParserFactory`. After the grid definition, there's a specs section. This contains for every name the specifications in the form of (key : value, key : value, …) pairs. Each key must be the name of a method that exist in the control class. While setting up the user interface, these methods will be called on the controls they belong to.

```
(
var example = """
rows {
(Slider:s1      | TextField:t1 | StaticText:l1  | StaticText:l2)
(StaticText:l3  | TextField:t2 ||               | Knob:k1      )
(StaticText:l4  | TextField:t3 |                | StaticText:l5)
(Button:b1      ||             | Button:b2      ||             )
}
specs {
s1 : (orientation:\\horizontal)
t1 : (value: text1)
t2 : (value: textfield2!!)
l1 : (string: yo yo yo label1)
l2 : (string: label2)
l3 : (string: label3)
l4 : (string: label4)
l5 : (string: label5)
}
)
""";
```

Parsing the guilang specification

In scparco, parsers for big specifications are built by combining together many smaller parsers for small parts of the specification. The parser for guilang will be built bottom-up.

Parsing the specs section

Parsing a key-value pair from the specs section

Let's first have a look at parsing a single key-value pair.

```
(
var keyval = SequenceOf([
    RegexParser("[a-z][a-z0-9]*"),
    ParserFactory.makeWs,
    StrParser(":"),
    ParserFactory.makeWs,
    RegexParser("[^(,\))][^(,\))]*"), // everything except comma and closing bracket
    ParserFactory.makeWs
]).map({|result| (\key : result[0], \value: result[4]) });
)
```

A keyvalue as used in the specs section consists of a sequence of things:

- first a Regex containing a lowercase letter followed by 0 or more lowercase letters or digits
- then some optional white space
- then a colon ":"
- then again some optional white space
- then as many characters as wanted except for a comma or a closing bracket
- and again some optional white space

When you combine a sequence of parsers using a SequenceOf parser, the parse result of the SequenceOf parser is a list of parse results of each of the parsers in the sequence. A parse result of any parser can be transformed using a map function. In the above code, the map function transforms the list of SequenceOf parse results into an event with a \key that contains the result of the first subparser (the key name) in the SequenceOf (the "result[0]"), as well as the result of the fifth parser (the value) ("result[4]"), and throws away the result[1] (optional whitespace), result[2] (colon), result[3] (optional whitespace) and result[5] (optional whitespace).

Parsing multiple key-value pairs

Multiple key-value pairs can be present, separated by a comma. To parse "things" separated by "other things", scparco's `ParserFactory` proivdes a convenience method `makeSepBy`. The method `makeSepBy` is a bit special: you call it with a Parser P as argument. P should match the separator (in this case, a comma followed by optional whitespace). It returns a function (not a parser!). The function can be called with another parser Q as argument. Q should match whatever is between the separators (in this case, a single key-value pair).

```
(
var keyval = SequenceOf([
    RegexParser("[a-z][a-z0-9]*"),
    ParserFactory.makeWs,
    StrParser(":"),
    ParserFactory.makeWs,
    RegexParser("[^(,\))][^(,\))]*"), // everything except comma and closing
↪   bracket
    ParserFactory.makeWs
```

```
]).map({|result| (\key : result[0], \value: result[4]) });

var betweenCommas = ParserFactory.makeSepBy(SequenceOf([
    StrParser(","),
    ParserFactory.makeWs])).(keyval);
)
```

The list of key-value pairs is enclosed in brackets "(" and ")", optionally followed by whitespace.

```
var spec_contents = ParserFactory.makeBetween(
    SequenceOf([StrParser("("), ParserFactory.makeWs]),
    SequenceOf([StrParser(")"), ParserFactory.makeWs])).(betweenCommas);
```

A single spec row is defined as an identifier : spec_contents

```
var spec = SequenceOf([
    ParserFactory.makeIdentifierParser,
    ParserFactory.makeWs,
    StrParser(":"),
    ParserFactory.makeWs,
    spec_contents,
    ParserFactory.makeWs
]).map({ |result| (\element: result[0], \elementspecs: result[4]) });
```

The map function is used to remove the useless information like the parsed colon or the parsed whitespaces.

The complete spec section consists of the keyword spec followed by {, many spec rows, and }.

```
var spec_section = SequenceOf([
    StrParser("specs"),
    ParserFactory.makeWs,
    StrParser("{"),
    ParserFactory.makeWs,
    Many(spec),
    StrParser("}")
]).map({ |result| (\allspecs : result[4]) });
```

Parsing the rows section

Each cell (row element) in the grid contains a Control : name. The Control is parsed with a regex that stars with a capital (since all class names in supercollider must start with a capital). The name is just a normal "identifier" as used in many programming languages.

```
var rowelement = SequenceOf([
    ParserFactory.makeWs,
    RegexParser("[A-Z][_a-zA-Z0-9]*"),
    ParserFactory.makeWs,
    StrParser(":"),
    ParserFactory.makeWs,
    ParserFactory.makeIdentifierParser,
    ParserFactory.makeWs
]).map({ | result | (\uielement : result[1], \name : result[5]) });
```

Cells can either contain a rowelement as defined above, or they can be empty. For easier post-processing, when an empty rowelement is detected, it maps the parse result to nil.

```
var row_contents = Choice([rowelement, ParserFactory.makeWsOne.map({ |result|
↪  nil })]);
```

Each row is a list of Control:identifier separated by pipe chars. Note that parsing a double pipe character (used to merge two adjacent cells on the same row together) must be tried before parsing a single pipe character. The reason is that the single pipe character parser would also succeed on ||, after which the double pipe character parser will fail since the single pipe character parser already consumed the first |.

```
var separatedByPipes = SequenceOf([
    row_contents,
    ParserFactory.makeWs,
    Many(SequenceOf([
        Choice([
            StrParser("||").map({|result| (\sep : \extendcol)}),
            StrParser("|").map({|result| (\sep : \newcol)})]),
        row_contents,
        ParserFactory.makeWs]).map({|result| [result[0], result[1]] }))
]).map({|result| [result[0], result[2] ].flat; });
```

To model a single row, use the separatedByPipes parser enclosed in () brackets (and handle whitespace too)

```
var row = ParserFactory.makeBetween(
    SequenceOf([StrParser("("), ParserFactory.makeWs]),
    SequenceOf([StrParser(")"), ParserFactory.makeWs])).(separatedByPipes);
```

The complete rows section consists of the word “rows” followed by {, followed by a collection of Many row}.

```
var row_section = SequenceOf([
    StrParser("rows"),
    ParserFactory.makeWs,
    StrParser("{"),
    ParserFactory.makeWs,
    Many(row),
    ParserFactory.makeWs,
    StrParser("}"),
    ParserFactory.makeWs
]).map({ |result| (\allrows : result[4]) });
```

A complete guilang spec is modeled as a sequence of a row_section and a spec_section.

```
var guilang = SequenceOf([
    ParserFactory.makeWs,
    row_section,
    ParserFactory.makeWs,
    spec_section
]).map({ |result| (\rows : result[1][\allrows], \specs: result[3][\allspecs]) });
```