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 and goes over all concepts and techniques gradually. The current document is more of a fast run-through, skipping on many details and concentrating on how I built up the grammar required to solve the problem at hand.

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, additionally providing you with all the hooks required 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 through the magic of `SimpleController`).

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 in a grid, made up of a list of rows. Each row is enclosed in round brackets `()` and divided in cells using pipe characters `|`. To merge two adjacent cells on the same row together (i.e. specify a rowspan), use a double pipe character `||`. The row with most cells determines the number of columns in the grid. As this is mostly a proof-of-concept (toy) language, there's no provision for merging adjacent cells in the same column (no way to specify columnspan).

Take a look at an example of a guilang specification. It contains a section with rows, which specify the layout and names each control. 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 `ScpParserFactory`.

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.

General debugging tip: tracing parsers

As you start to make more complicated parsers, you may run into situations where things don't work as expected. In that case, you can always call the parser's `run` method with argument `trace:true`. This causes the parsers to log to the post window when and where they start, and whether they succeed/fail.

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 = ScpSequenceOf([
    ScpRegexParser("[a-z][a-z0-9]*"),
    ScpParserFactory.makeWs,
    ScpStrParser(":"),
    ScpParserFactory.makeWs,
    ScpRegexParser("[^(,\))][^(,\))]*"), // everything except comma and closing bracket
    ScpParserFactory.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 ScpSequenceOf parser, the parse result of the ScpSequenceOf 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 ScpSequenceOf parse results into an event with a \key that contains the result of the first subparser (the key name) in the ScpSequenceOf (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 `ScpParserFactory` proivdes a convenience method `makeSepBy`. The method `makeSepBy` is a bit special: you call it with a ScpParser 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 = ScpSequenceOf([
    ScpRegexParser("[a-z][a-z0-9]*"),
    ScpParserFactory.makeWs,
    ScpStrParser(":"),
    ScpParserFactory.makeWs,
    ScpRegexParser("[^(,\))][^(,\))]*"), // everything except comma and
↪   closing bracket
    ScpParserFactory.makeWs
]).map({|result| (\key : result[0], \value: result[4]) });

var betweenCommas = ScpParserFactory.makeSepBy(ScpSequenceOf([
    ScpStrParser(","),
    ScpParserFactory.makeWs])).(keyval);
)
```

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

```
var spec_contents = ScpParserFactory.makeBetween(
    ScpSequenceOf([ScpStrParser("("), ScpParserFactory.makeWs]),
    ScpSequenceOf([ScpStrParser(")"),
↪   ScpParserFactory.makeWs])).(betweenCommas);
```

A single spec row is defined as an identifier : spec_contents

```
var spec = ScpSequenceOf([
    ScpParserFactory.makeIdentifierParser,
    ScpParserFactory.makeWs,
    ScpStrParser(":"),
    ScpParserFactory.makeWs,
    spec_contents,
    ScpParserFactory.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 = ScpSequenceOf([
    ScpStrParser("specs"),
    ScpParserFactory.makeWs,
    ScpStrParser("{"),
    ScpParserFactory.makeWs,
    ScpMany(spec),
    ScpStrParser("}")
]).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 = ScpSequenceOf([
    ScpParserFactory.makeWs,
    ScpRegexParser("[A-Z][_a-zA-Z0-9]*"),
```

```
    ScpParserFactory.makeWs,
    ScpStrParser(":"),
    ScpParserFactory.makeWs,
    ScpParserFactory.makeIdentifierParser,
    ScpParserFactory.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 = ScpChoice([rowelement, ScpParserFactory.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 = ScpSequenceOf([
    row_contents,
    ScpParserFactory.makeWs,
    ScpMany(ScpSequenceOf([
        ScpChoice([
            ScpStrParser("||").map({|result| (\sep : \extendcol)}),
            ScpStrParser("|").map({|result| (\sep : \newcol)})]),
        row_contents,
        ScpParserFactory.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 = ScpParserFactory.makeBetween(
    ScpSequenceOf([ScpStrParser("("), ScpParserFactory.makeWs]),
    ScpSequenceOf([ScpStrParser(")"),
↪  ScpParserFactory.makeWs])).(separatedByPipes);
```

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

```
var row_section = ScpSequenceOf([
    ScpStrParser("rows"),
    ScpParserFactory.makeWs,
    ScpStrParser("{"),
    ScpParserFactory.makeWs,
    ScpMany(row),
    ScpParserFactory.makeWs,
    ScpStrParser("}"),
    ScpParserFactory.makeWs
]).map({ |result| (\allrows : result[4]) });
```

Parsing a complete guilang specification

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

```
var guilang = ScpSequenceOf([
    ScpParserFactory.makeWs,
    row_section,
```

```
    ScpParserFactory.makeWs,
    spec_section
]).map({ |result| (\rows : result[1][\allrows], \specs: result[3][\allspecs]) });
```

Appendix: parse result from the example

```
(
'rows': [
    [
        ( 'name': "s1", 'uielement': "Slider" ),
        ( 'sep': 'newcol' ),
        ( 'name': "t1", 'uielement': "TextField" ),
        ( 'sep': 'newcol' ),
        ( 'name': "l1", 'uielement': "StaticText" ),
        ( 'sep': 'newcol' ),
        ( 'name': "l2", 'uielement': "StaticText" )
    ],
    [
        ( 'name': "l3", 'uielement': "StaticText" ),
        ( 'sep': 'newcol' ),
        ( 'name': "t2", 'uielement': "TextField" ),
        ( 'sep': 'extendcol' ),
        nil,
        ( 'sep': 'newcol' ),
        ( 'name': "k1", 'uielement': "Knob" )
    ],
    [
        ( 'name': "l4", 'uielement': "StaticText" ),
        ( 'sep': 'newcol' ),
        ( 'name': "t3", 'uielement': "TextField" ),
        ( 'sep': 'newcol' ),
        nil,
        ( 'sep': 'newcol' ),
        ( 'name': "l5", 'uielement': "StaticText" )
    ],
    [
        ( 'name': "b1", 'uielement': "Button" ),
        ( 'sep': 'extendcol' ),
        nil,
        ( 'sep': 'newcol' ),
        ( 'name': "b2", 'uielement': "Button" ),
        ( 'sep': 'extendcol' ),
        nil
    ]
],
'specs': [
    (
        'elementspecs': [ ( 'value': "\\horizontal", 'key': "orientation" )
↪   ],
        'element': "s1" ),
    (
        'elementspecs': [ ( 'value': "text1", 'key': "value" ) ],
        'element': "t1" ),
    (
        'elementspecs': [ ( 'value': "textfield2!!", 'key': "value" ) ],
        'element': "t2" ),
    (
```

```
        'elementspecs': [ ( 'value': "yo yo yo label1", 'key': "string" ) ],
        'element': "l1" ),
    (
        'elementspecs': [ ( 'value': "label2", 'key': "string" ) ],
        'element': "l2" ),
    (
        'elementspecs': [ ( 'value': "label3", 'key': "string" ) ],
        'element': "l3" ),
    (
        'elementspecs': [ ( 'value': "label4", 'key': "string" ) ],
        'element': "l4" ),
    (
        'elementspecs': [ ( 'value': "label5", 'key': "string" ) ],
        'element': "l5" )
]
)
```