Static Site Generator

by Beretta Michele, Crippa Bianca, Toure Pape Alpha

University of Bergamo
School of Engineering
Formal Languages & Compilers
2021-2022

1 Project description

This project is a *Static Site Generator* written in the *Haskell* programming language. As the name suggests, a static site generator (SSG for here on) is a tool that allows the creation static web pages from templates and raw data. More specifically, it consumes files written in a custom-defined language and outputs HTML5 files.

Such tools have long attracted the attention of various players in the web space as they provide an abstraction that achieves two things: the first is a degree of automation in the domain of web development, specifically in making web pages; the second is transpilation as there is a source to source compilation. Combined together, they prove to be an effective approach to either building user interfaces for web site building used by non technical users or styling the same website in vastly different ways.

2 Design goals

The main non functional requirements to be pursued are the following:

- *Simplicity*: the input language ought to have an easy learning curve, prioritizing minimal static sites, valuing user productivity;
- *Extensibility*: with future change in mind, it should be possible to define your custom layouts to be applied in the source files;
- *Correctness*: as any compilation, consistency in the output produced is of utmost importance;
- *Security*: attacker controlled arbitrarily long inputs is an intrinsically hard threat-model to handle;
- *Maintainability*: the resulting code should be easily modifiable;

The programming language chosen (Haskell) aims to address several of the aforementioned points by providing an extremely strong type system, type safety, immutability, memory safety and strong and verifiable formal guarantees about correctness.

3 Usage

Usage is quite straightforward, after installing the binary provide a folder containing the source files written in the DSL. Here is an example of running `haskell-ssg` for a `website` folder, with a `build` output folder. Once the file are parsed and rendered, a web server is spawn at `localhost:4000` so that you can preview the output live.

```
$ ./haskell-ssg website --output build
Reading macros...
Reading source files...
Building...
Server running on port 4000...
>
```

In Table 1 are visible all available command line options. Once the web server is running, the program presents a very simple prompt and accept 3 commands:

- `r` or `reload`, which allows the user to re-parse the files to update the preview;
- `q` or `quit`, to exit the program;
- `h` or `help`, to view which commands are available.

Argument	*Description*	*Default*
`SRC`	The source folder, containing all the main DSL files	`_src`
`--output`	The output folder, which will contain all HTML files	`_build`
`--layouts`	A folder with *layout* macros	`_layouts`
`--macros`	A folder with *generic* macros, not to be used as layouts	`_macros`
`--static`	A folder with static files	`_static`
`--out-static`	Where to copy the static files	`_build/static`

Table 1. All command line arguments and their defaults.

4 DSL

The DSL (Domain Specific Language) used as source language to design the web pages is a LISP-esque formatting language, where each element has a 1 to 1 correspondance to an HTML attributes or tags. It is also possible to define *custom macros* so that custom components can be re-used, or even *custom layouts* for entire pages.

1 Grammar

Here is the abstract syntax of the language for a document definition and a macro definition.

document	::=	*config content**
config	::=	`{` **title** *string* (**custom-css** *string*)? (**layout** *string*)? `}`
content	::=	*unquote* | *string* | *list* | *macro-call*
unquote	::=	`@` *identifier*
list	::=	`(` *identifier attr-list*? *content** `)`
attr-list	::=	`[` *tuple** `]`
tuple	::=	`(` *identifier string*? `)`
macro-call	::=	`(` *identifier macro-arg** `)`
macro-arg	::=	`(` *identifier content** `)`
string	::=	`"` *s-char** `"`
s-char	::=	\ any | any but `"`
identifier	::=	*i-char+*
i-char	::=	none of `()[]"@`, blank or newline
macro-def	::=	`'(` *identifier content** `)`

If the *layout* property in the *config* section of a document is missing, `"default"` will be used.

2 Tags and attributes

All identifiers that are accepted as *list* names are visible in Table 2, while all identifiers that are accepted as *tuple* keys are visible in Table 3.

Identifier	***HTML tag***
par	p
title	h1
subtitle	h2
image	img
b	strong
i	i
nl	br
table	table
trow	tr
tcel	td
link	a
html	html
head	head
pagetitle	title
body	body
div	div
style	style
link_	link
list	ul
enumerate	ol
item	li

Table 2. Identifiers accepted as list names

accept	id	onended	onsubmit	summary
acceptCharset	icon	onerror	onsuspend	tabindex
accesskey	ismap	onfocus	ontimeupdate	target
action	itemprop	onformchange	onundo	title
alt	itemscope	onforminput	onunload	type
async	keytype	onhaschange	onvolumechange	usemap
autocomplete	label	oninput	onwaiting	value
autofocus	lang	oninvalid	open	width
autoplay	list	onkeydown	optimum	wrap
challenge	loop	onkeyup	pattern	xmlns
charset	low	onload	ping	
checked	manifest	onloadeddata	placeholder	
cite	max	onloadedmetadata	preload	
class	maxlength	onloadstart	pubdate	
cols	media	onmessage	radiogroup	
colspan	method	onmousedown	readonly	
content	min	onmousemove	rel	
contenteditable	multiple	onmouseout	required	
contextmenu	name	onmouseover	reversed	
controls	novalidate	onmouseup	role	
coords	onbeforeonload	onmousewheel	rows	
data	onbeforeprint	ononline	rowspan	
datetime	onblur	onpagehide	sandbox	
defer	oncanplay	onpause	scope	
dir	oncanplaythrough	onplay	scoped	
disabled	onchange	onplaying	seamless	
draggable	onclick	onprogress	selected	
enctype	oncontextmenu	onpropstate	shape	
for	ondblclick	onratechange	size	
formaction	ondrag	onreadystatechange	sizes	
formenctype	ondragend	onredo	span	
formmethod	ondragenter	onresize	spellcheck	
formnovalidate	ondragleave	onscroll	src	
formtarget	ondragover	onseeked	srcdoc	
headers	ondragstart	onseeking	start	
height	ondrop	onselect	step	
hidden	ondurationchange	onstalled	style	
high	onemptied	onstorage	subject	

Table 3. Identifiers accepted as attributes lists' tuple key names

3 Example

What follows is a simple example of a *document* written in the DSL.

```
{ title "Home page" custom-css "/static/custom.css" layout "default" }

(title "Home page")

(subtitle "Section 1")
(par
  (list [(class "custom-css-class")]
    (item "First item")
    (item "Second item"))
```

5 Parser's details

Haskell SSG makes uses of *Megaparsec*, an industrial strength monadic parser combinator library derived from the simpler *Parsec* library, bundled with GHC. Proper installation of the library requires a working installation of `cabal` and `ghc`, which are part of any Haskell installation.

1 Monad

In functional programming, a **monad** is a software design pattern with a structure that combines functions and wraps their return values in a type with additional information (usually something about computation).

In addition to defining a wrapping **monadic type**, monads define two operators: one to wrap a value in the monad type, called `return` in Haskell, and another to compose together functions that output values of the monad type, called `bind` or `>>=` in Haskell - these are known as **monadic functions**. These functions must also obey some laws, namely `return` must act as an identity and `>>=` must be "associative".

Functional languages use monads to turn complicated sequences of functions into succinct pipelines that abstract away control flow and side-effects.

In *Megaparsec*, parsers are monad, and as such can be combined using a uniform abstraction, common throughout all Haskell code. Moreover, parser can also be *monad transformers*, which means they can combine with other monads to enhance functionality. For example, in this project the parser is combined with a `State Env`, another monad used to keep track of eventual macro declarations.

2 Parser combinators

A *parser combinator* is a higher-order function that takes one or more parsers as input and produces a new parser as its output.

To put it simply, combining parsers means that given two distinct parsers a third one can be made that is the sum of its part, namely, can parse both grammars parsed by the two smaller parsers.

Compared to Bison/yacc/ANTLR which are parsers **generators**, *Megaparsec* is a parser combinator library. In functional terms, parsing combination is akin to treating parsers as first class values and making use of higher order functions to compose them.

Therefore, parser combinators offer a universal and flexible approach to parsing. They follow the structure of an underlying grammar, are modular, well-structured, easy to maintain, and can recognize a large variety of languages including context-sensitive ones.

However, these advantages generally introduce a performance overhead as the same powerful parsing algorithm is used to recognize every language, even the simplest one. Specifically, a parser combinator uses the full power of a Turing-equivalent formalism to recognize even simple languages that could be recognized by finite state machines or pushdown automata. Time-wise, parser combinators cannot compete with parsers generated by well-performing parser generators or optimized hand-written code. Meta-programming approaches such as macros and staging have been applied to Scala parser combinators with significant performance improvements. In general, these approaches remove composition overhead and intermediate representations. Moreover, parser generators can give static guarantees about termination and non-ambiguity which, depending on the specific implementation, parser combinator libraries are not able to give.

6 Errors

Errors are managed entirely by *Megaparsec*. *Megaparsec*, upon parsing failure, returns a *ParseError s e*, which is an abstract data type that represents error messages on a stream of type `s` for errors of type `e`, and is made as follows

```
 data ParseError s e
    = TrivialError Int (Maybe (ErrorItem (Token s))) (Set (ErrorItem (Token s)))
    | FancyError Int (Set (ErrorFancy e))
```

More specifically, a `TrivialError` is generated by Megaparsec's machinery and includes an offset, an unexpected `Token s` (if any) and a set of expected `Token s`. A `FancyError`, on the other way, is used for custom errors, represented by `ErrorFancy e`.

```
data ErrorItem t
    = Tokens (NonEmpty t)
    | Label (NonEmtpy Char)
    | EndOfInput

data ErrorFancy e
    = ErrorFail String
    | ErrorIndentation Ordering Pos Pos
    | ErrorCustom e
```

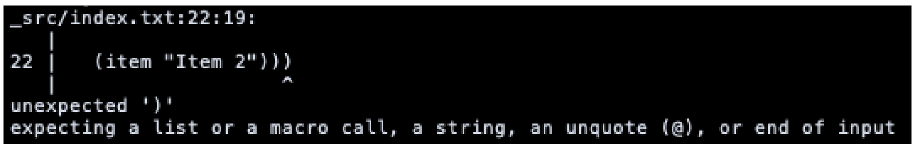
In order to enrich error reporting, three semantic errors have been added to clarify the nature of the error the user, defined by the following type.

```
data CustomError
    = InvalidListName Text
    | InvalidAttrName Text
    | IdentifierAlreadyTaken Text
```

These are represented by the following error messages.

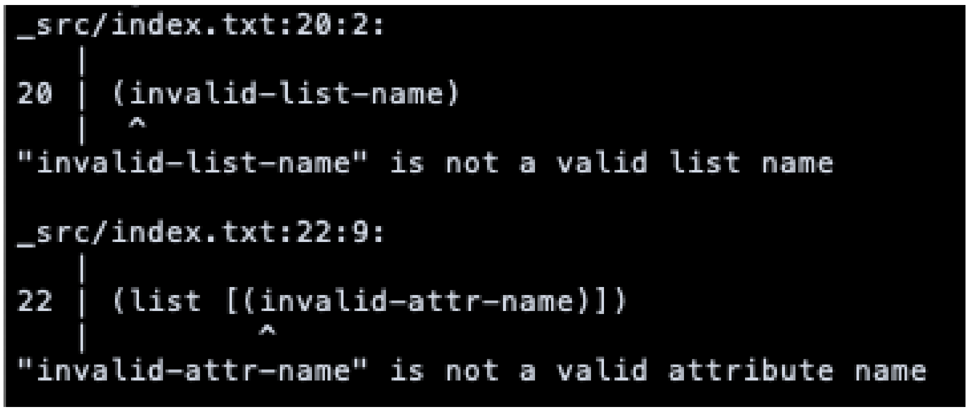
Error	***Message***
`InvalidListName name`	"name" is not a valid list name
`InvalidAttrName name`	"name" is not a valid attribute name
`IdentifierAlreadyTaken name`	"name" is already declared and cannot be used as a macro

Here an example of a syntax error:



```
_src/index.txt:22:19:
   |
22 |    (item "Item 2")))
   |                   ^
unexpected ')'
expecting a list or a macro call, a string, an unquote (@), or end of input
```

And here an example of some semantic errors:



```
_src/index.txt:20:2:
   |
20 | (invalid-list-name)
   |  ^
"invalid-list-name" is not a valid list name

_src/index.txt:22:9:
   |
22 | (list [(invalid-attr-name)])
   |         ^
"invalid-attr-name" is not a valid attribute name
```