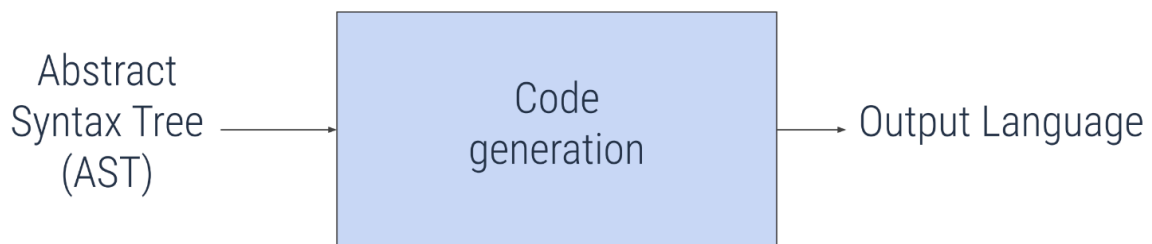


Compiler Project | 05 CODE GENERATION

By: Mónica AYALA, Davide AVESANI, Théodore PRÉVOT

Introduction

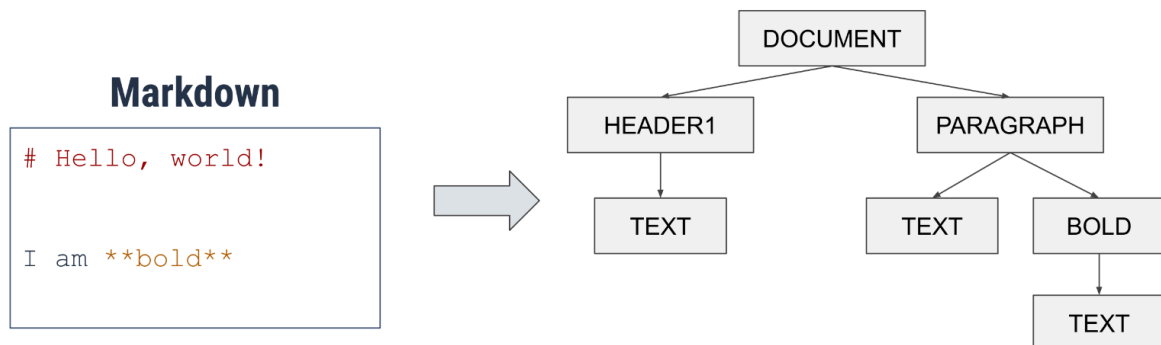
The objective of this last phase of code generation is to generate the output language, in our case HTML, starting from the AST created in the previous phase.



AST structure recap

In the previous phases we successfully generated the AST, a three data structure that organizes the tokens returned by the scanner. These tokens are the codification of the initial Markdown document that allows the Parser to create the AST.

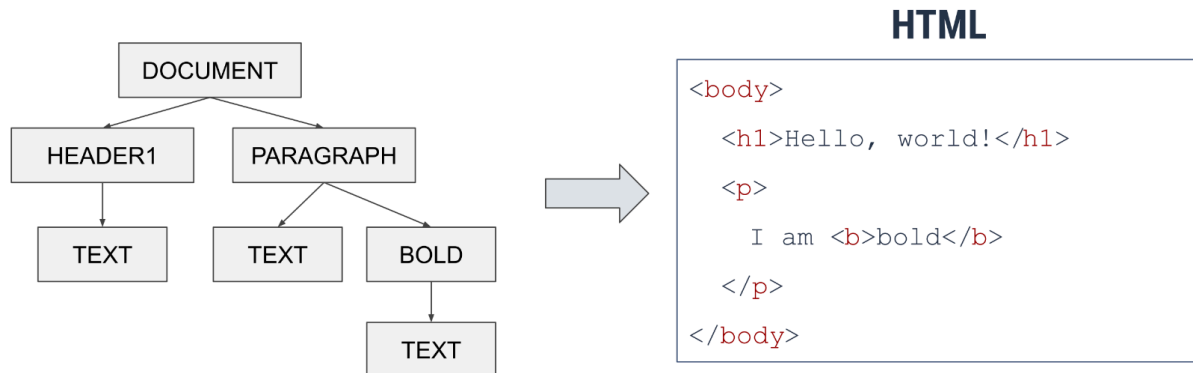
An example of the resulting AST is given in the following image.



Code generation

The final step is the one of translating the generated EST in the HTML document. In other words we have to traverse the generated tree and translate its nodes into HTML tags or text elements.

An example is given below.



Code Structure

As we said in the previous paragraph we traverse the tree, by using a loop, and we translate the nodes in the respective HTML tag or text element.

We perform the translation by matching each node to its respective HTML tag with a switch-case statement. The code example shows how when a Header 2 node is found on the AST, we proceed to add to the HTML document an h2 tag containing the respective text.

```
string code_generation_from_dom(DOM *dom, unsigned int indent)
{
    if (dom == NULL)
        return STR("");

    switch (dom->dom_el)
    {
        case Document:
        { ...
        case Header1:
        { ...
        case Header2:
        {
            string html = STR("");
            add_indentation(html, indent);

            APPEND_ARR(html, "<h2>");
            APPEND_ARR(html, dom->text);
            APPEND_ARR(html, "</h2>\n");

            return html;
        }
    }
}
```

SVG

For the SVG tokens, we created some ad hoc functions to proceed with the translation in HTML language, avoiding code duplication and making the code more reusable.

```
void append_int_prop(string buffer, const char *prop, int value) ...
void append_svg_element(string buffer, SvgInst *element)
{
    switch (element->kind)
    {
    case Line:
    { ...
    case Polyline:
    { ...
    case Polygon:
    { ...
    case Circle:
    { ...
    case Ellipse:
    { ...
    case Rect:
    { ...
    case Text:
    { ...
    }

    APPEND_ARR(buffer, "\n");
}
```

In particular, when processing an SVG element in the AST, we traverse through each item within the SVG (such as lines, rectangles, ellipses, etc.) along with their associated properties (size, color, coordinates, etc.). At this point, we invoke the functions we've defined to convert them into corresponding HTML elements.

```
case SVG:
{
    string html = STR("");
    if (dom->svg_coords == NULL || dom->svg_coords->next == NULL)
        return html;

    add_indentation(html, indent);
    APPEND_ARR(html, "<svg viewBox=\");
    // Add viewBox coordinates
    append_coord_list(html, dom->svg_coords, " ", " ");

    APPEND_ARR(html, "\">\n");

    SvgList *child = dom->svg_children;
    while (child != NULL)
    {
        add_indentation(html, indent + 1);
        append_svg_element(html, child->svg);
        child = child->next;
    }

    add_indentation(html, indent);
    APPEND_ARR(html, "</svg>\n");
}
```

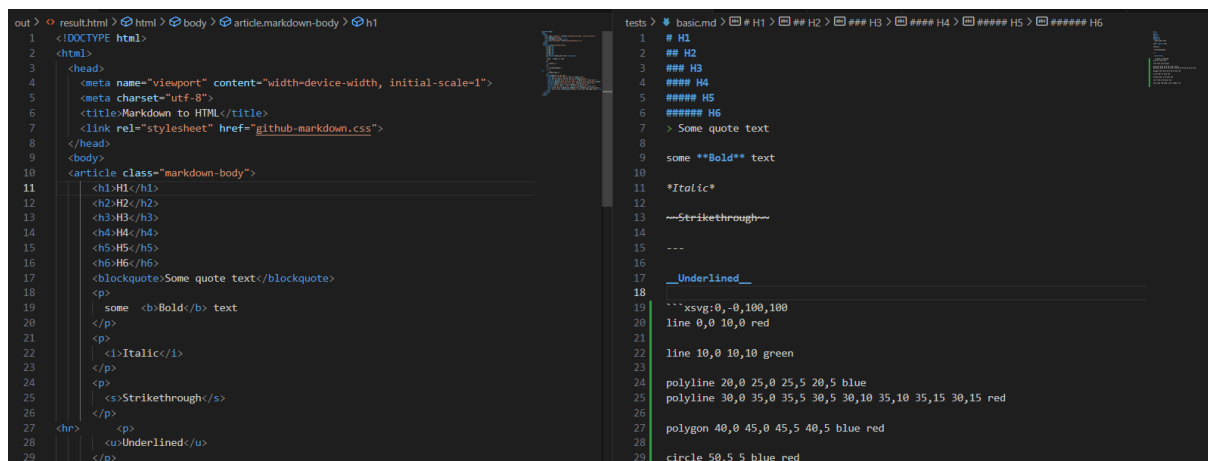
Markdown to HTML results

The HTML document is initially generated with the standard HTML structure

```
const char HTML_HEADER[] = "<!DOCTYPE html>\n"
                             "<html>\n"
                             "  <head>\n"
                             "    <meta name=\"viewport\" content=\"width=device-width, initial-scale=1\">\n"
                             "    <meta charset=\"utf-8\">\n"
                             "    <title>Markdown to HTML</title>\n"
                             "    <link rel=\"stylesheet\" href=\"github-markdown.css\">\n"
                             "  </head>\n"
                             "  <body>\n"
                             "    <article class=\"markdown-body\">\n"

const char HTML_FOOTER[] = "  </article>\n"
                           "  </body>\n"
                           "</html>";
```

As demonstrated earlier, the tags are incorporated into this document. The final outcome is the complete HTML document that contains the translation of the initial markdown file.



```
out > result.html > html > body > article.markdown-body > h1
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta name="viewport" content="width=device-width, initial-scale=1">
5   <meta charset="utf-8">
6   <title>Markdown to HTML</title>
7   <link rel="stylesheet" href="github-markdown.css">
8 </head>
9 <body>
10  <article class="markdown-body">
11    <h1>H1</h1>
12    <h2>H2</h2>
13    <h3>H3</h3>
14    <h4>H4</h4>
15    <h5>H5</h5>
16    <h6>H6</h6>
17    <blockquote>Some quote text</blockquote>
18    <p>
19      some <b>Bold</b> text
20    </p>
21    <p>
22      <i>Italic</i>
23    </p>
24    <p>
25      <del>Strikethrough</del>
26    </p>
27    <p>
28      <u>Underlined</u>
29    </p>
30  </article>
31 </body>
32 </html>
```

```
tests > basic.md > # H1 > ## H2 > ### H3 > #### H4 > ##### H5 > ##### H6
1 # H1
2 ## H2
3 ### H3
4 #### H4
5 ##### H5
6 ##### H6
7 > Some quote text
8
9 some Bold text
10
11 *Italic*
12
13 ~Strikethrough~
14
15 ---
16
17 _Underlined_
18
19 ```xsvg:0,-0,100,100
20 line 0,0 10,0 red
21
22 line 10,0 10,10 green
23
24 polyline 20,0 25,0 25,5 30,5 blue
25 polyline 30,0 35,0 35,5 30,5 30,10 35,10 35,15 30,15 red
26
27 polygon 40,0 45,0 45,5 40,5 blue red
28
29 circle 50,5 5 blue red
```

Summary

With the completion of this phase, we achieved HTML code generation from a provided markdown file. Although the compiler is now fully functional, there are numerous opportunities for improvement and expansion.

Future developments could include implementing robust error recognition and handling mechanisms, expanding the spectrum of recognized tokens, and optimizing the entire translation process for greater efficiency and performance.