Compiler Project | 04 PARSER

By: Mónica AYALA, Davide AVESANI, Théodore PRÉVOT

Parse overview

In the previous phases of the project we succeeded in defining the grammar, creating a scanner able to recognize the tokens on the grammar and creating the Abstract Syntax Tree (AST) used to store the tokens.

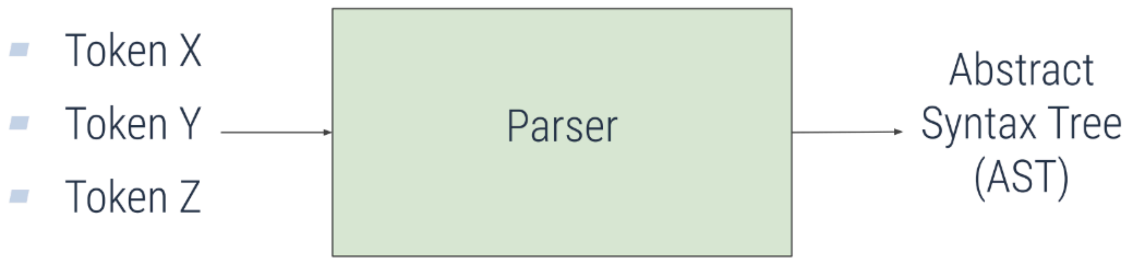


Fig 1. Representation of a Parser

The objective of this phase is to make these two components interact together, which means generating the AST starting from the list of tokens returned by the scanner. The output will be used in the next step to generate the HTML code.

For this purpose we used a parser generator called Bison.

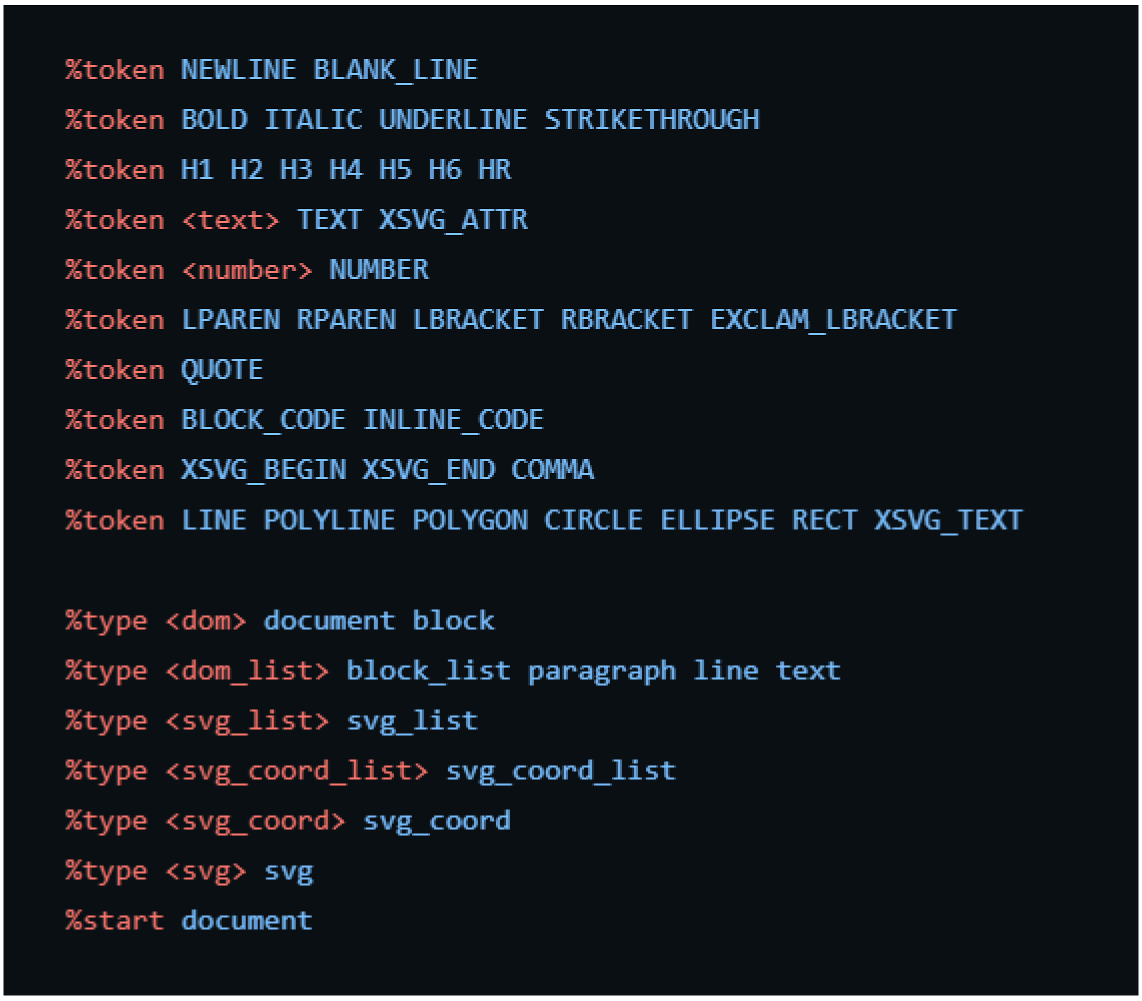
Bison Parser

Bison is a general-purpose parser generator that converts an annotated context-free grammar into a parser for that grammar. It is part of the GNU Compiler Collection (GCC) and is often used in combination with the Flex lexical analyzer generator.
It is based on Yacc, a programming tool that generates parsers (syntax analyzers) for formal languages.

In our case, we were provided with an established environment, and the objective is to complete the **parser.y** file. The aim is to ensure that the parser correctly constructs the Abstract Syntax Tree (AST) based on the tokens yielded by the scanner.

Parser.y

The parser.y file is composed of a section of C Code followed by definitions, production rules and ending with more C Code. For the C Code parts we didn't have to modify anything in the file, so we will focus on the other two middle parts. For the **definition** section we focus on defining all of the tokens that our Abstract Syntax Tree will have, these have to be in concordance with the scanner's return tokens. We also declare the non-terminal symbols and associate them with a type/semantic value.

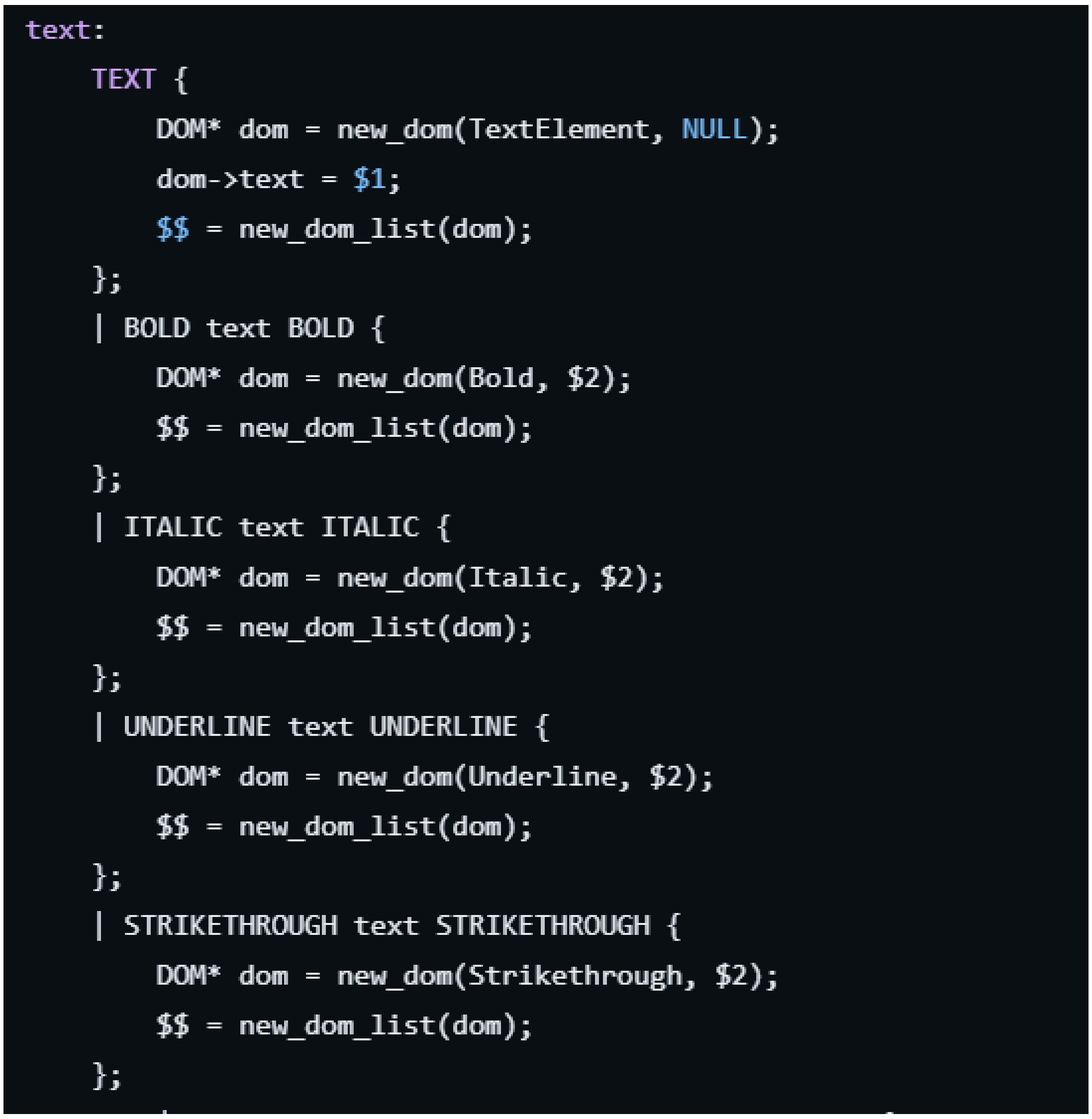


```
%token NEWLINE BLANK_LINE
%token BOLD ITALIC UNDERLINE STRIKETHROUGH
%token H1 H2 H3 H4 H5 H6 HR
%token <text> TEXT XSVG_ATTR
%token <number> NUMBER
%token LPAREN RPAREN LBRACKET RBRACKET EXCLAM_LBRACKET
%token QUOTE
%token BLOCK_CODE INLINE_CODE
%token XSVG_BEGIN XSVG_END COMMA
%token LINE POLYLINE POLYGON CIRCLE ELLIPSE RECT XSVG_TEXT

%type <dom> document block
%type <dom_list> block_list paragraph line text
%type <svg_list> svg_list
%type <svg_coord_list> svg_coord_list
%type <svg_coord> svg_coord
%type <svg> svg
%start document
```

Fig 2. Code for the definitions part in parser.y

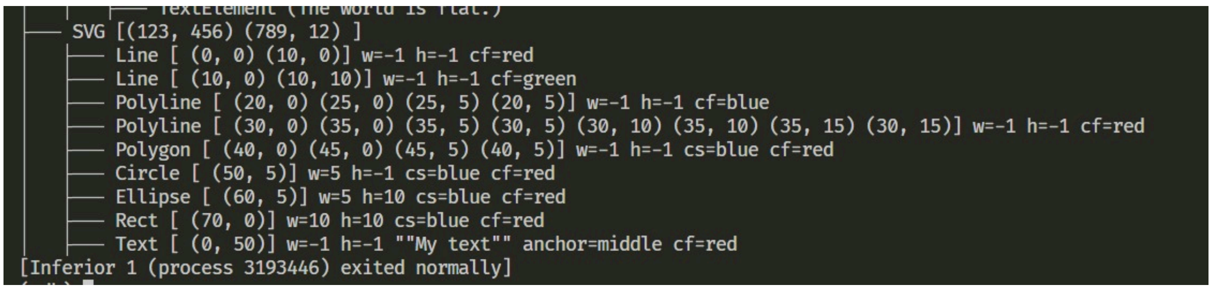
After this we can move on to the production rules part of the code, where we had to define the production rules (patterns) for all non-terminal symbols with the tokens we had previously declared, it is also where we fill the AST's nodes variables with the token's values.



```
text:
    TEXT {
        DOM* dom = new_dom(TextElement, NULL);
        dom->text = $1;
        $$ = new_dom_list(dom);
    };
    | BOLD text BOLD {
        DOM* dom = new_dom(Bold, $2);
        $$ = new_dom_list(dom);
    };
    | ITALIC text ITALIC {
        DOM* dom = new_dom(Italic, $2);
        $$ = new_dom_list(dom);
    };
    | UNDERLINE text UNDERLINE {
        DOM* dom = new_dom(Underline, $2);
        $$ = new_dom_list(dom);
    };
    | STRIKETHROUGH text STRIKETHROUGH {
        DOM* dom = new_dom(Strikethrough, $2);
        $$ = new_dom_list(dom);
    };
```

Fig 3. Example of text production rules in parser.y

We do this for all of the non-terminal symbols which were defined in our grammar. Some of which are not given a proper name, like we see in the example where we do not define “bold text” but are successfully parsed into the Abstract Syntax Tree.



```
│    │    ├── TextElement (The world is flat.)
├── SVG [(123, 456) (789, 12) ]
│    ├── Line [ (0, 0) (10, 0)] w=-1 h=-1 cf=red
│    ├── Line [ (10, 0) (10, 10)] w=-1 h=-1 cf=green
│    ├── Polyline [ (20, 0) (25, 0) (25, 5) (20, 5)] w=-1 h=-1 cf=blue
│    ├── Polyline [ (30, 0) (35, 0) (35, 5) (30, 5) (30, 10) (35, 10) (35, 15) (30, 15)] w=-1 h=-1 cf=red
│    ├── Polygon [ (40, 0) (45, 0) (45, 5) (40, 5)] w=-1 h=-1 cs=blue cf=red
│    ├── Circle [ (50, 5)] w=5 h=-1 cs=blue cf=red
│    ├── Ellipse [ (60, 5)] w=5 h=10 cs=blue cf=red
│    ├── Rect [ (70, 0)] w=10 h=10 cs=blue cf=red
│    ├── Text [ (0, 50)] w=-1 h=-1 ""My text"" anchor=middle cf=red
[Inferior 1 (process 3193446) exited normally]
```

Fig 4. Part of the output AST with a test svg declaration.

Summary

By successfully completing this phase we are able to obtain the final output that will allow us to generate the HTML code. Moreover, this phase gave us the possibility to check the correctness of the scanner phase. With a validated syntax tree in hand, subsequent compiler phases can confidently proceed, hopefully leading to the generation of the final output.