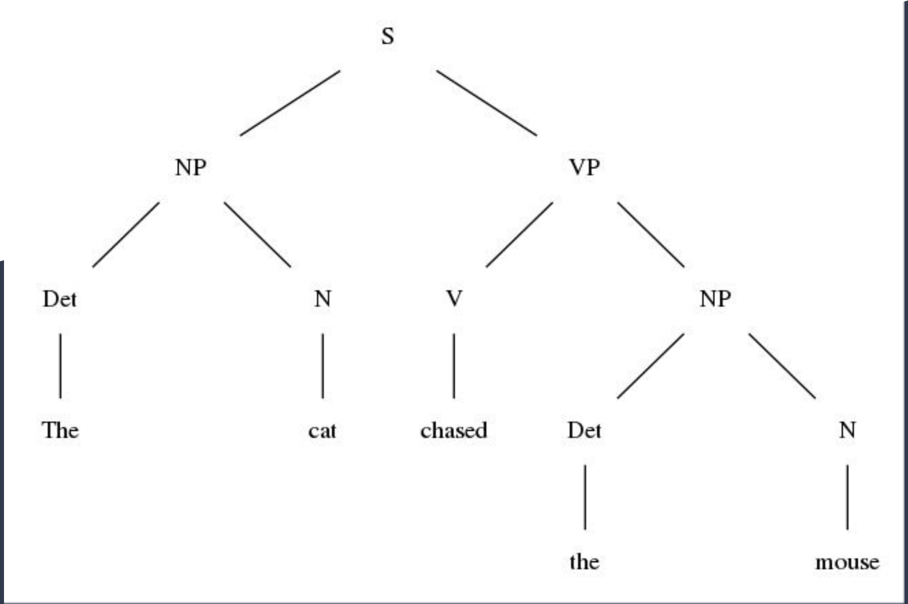
CPTS 483 Project
Syntax Tree Generator

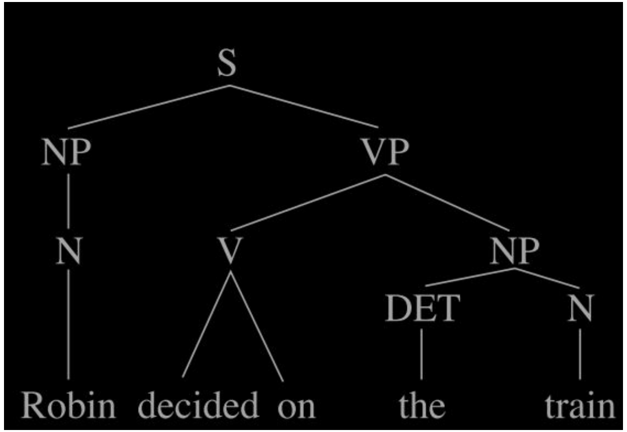
Odeysiuss Tuon

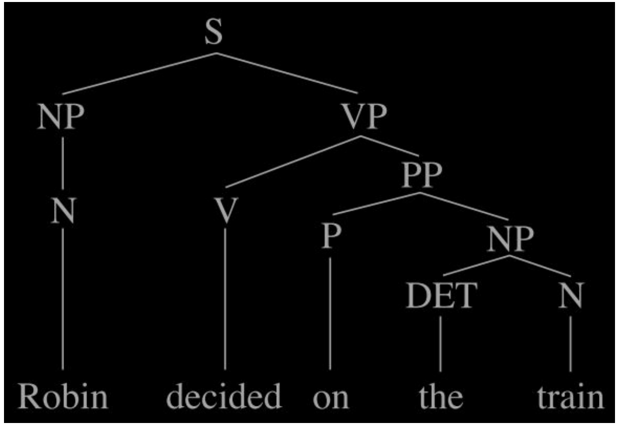


Introduction and Motivation

- Minoring in Linguistics
- Syntax Tree Generation seemed simple enough
- Applied knowledge of Flex/Bison to scan and parse labeled bracket notation to an actual tree
- Syntax trees are used to analyze constituent structure of languages

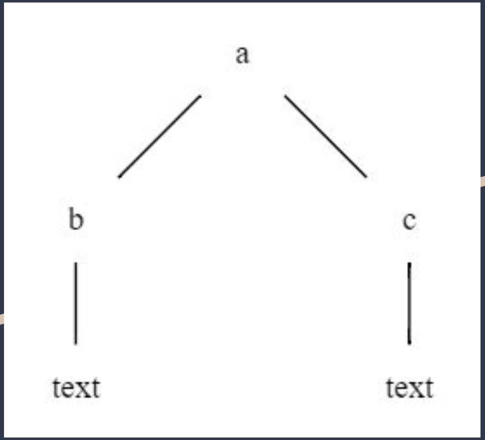
[S [NP [N Robin]] [VP [V decided on] [NP [Det the] [N train]]]]



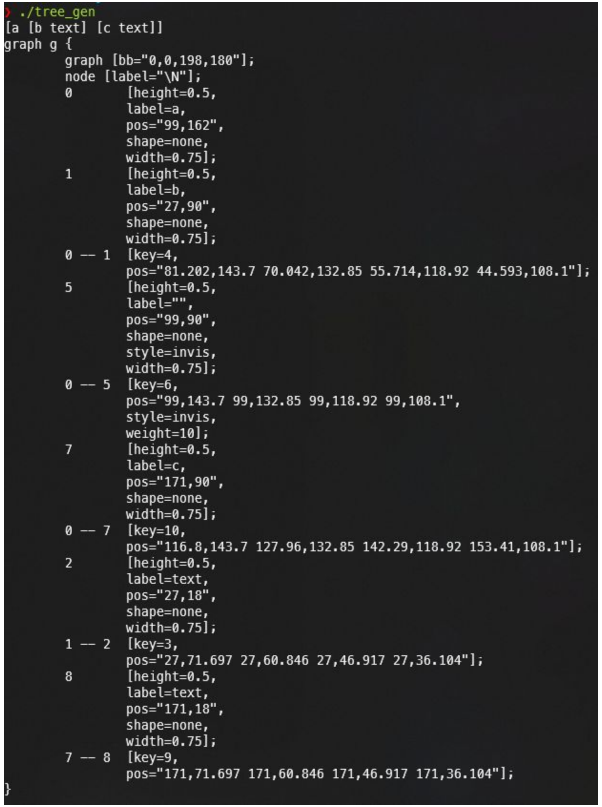


(Images from Van Valin's "An Introduction to Syntax")

Program Description



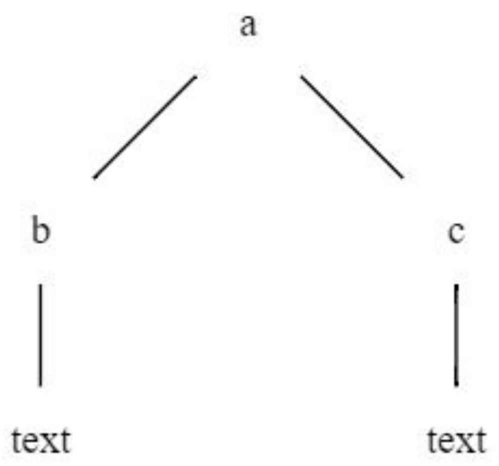
- Flex and Bison to scan and parse input (stdin)
- Graphviz C library (gvc) to draw the trees
- Outputs (stdout) a tree representation in the Graphviz DOT format
- The resulting DOT format can be converted to various graphical formats, such as PNG and SVG



```
❯ ./tree_gen
[a [b text] [c text]]
graph g {
        graph [bb="0,0,198,180"];
        node [label="\N"];
        0       [height=0.5,
                label=a,
                pos="99,162",
                shape=none,
                width=0.75];
        1       [height=0.5,
                label=b,
                pos="27,90",
                shape=none,
                width=0.75];
        0 -- 1  [key=4,
                pos="81.202,143.7 70.042,132.85 55.714,118.92 44.593,108.1"];
        5       [height=0.5,
                label="",
                pos="99,90",
                shape=none,
                style=invis,
                width=0.75];
        0 -- 5  [key=6,
                pos="99,143.7 99,132.85 99,118.92 99,108.1",
                style=invis,
                weight=10];
        7       [height=0.5,
                label=c,
                pos="171,90",
                shape=none,
                width=0.75];
        0 -- 7  [key=10,
                pos="116.8,143.7 127.96,132.85 142.29,118.92 153.41,108.1"];
        2       [height=0.5,
                label=text,
                pos="27,18",
                shape=none,
                width=0.75];
        1 -- 2  [key=3,
                pos="27,71.697 27,60.846 27,46.917 27,36.104"];
        8       [height=0.5,
                label=text,
                pos="171,18",
                shape=none,
                width=0.75];
        7 -- 8  [key=9,
                pos="171,71.697 171,60.846 171,46.917 171,36.104"];
}
```

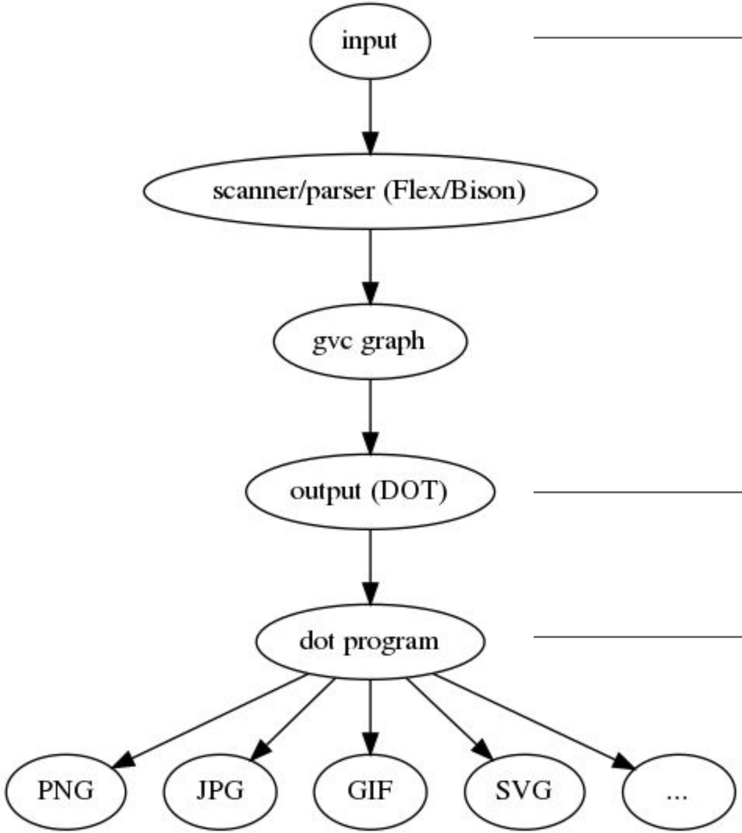
DOT format

dot -Tpng



Example with input: “[a [b text] [c text]]”

Framework

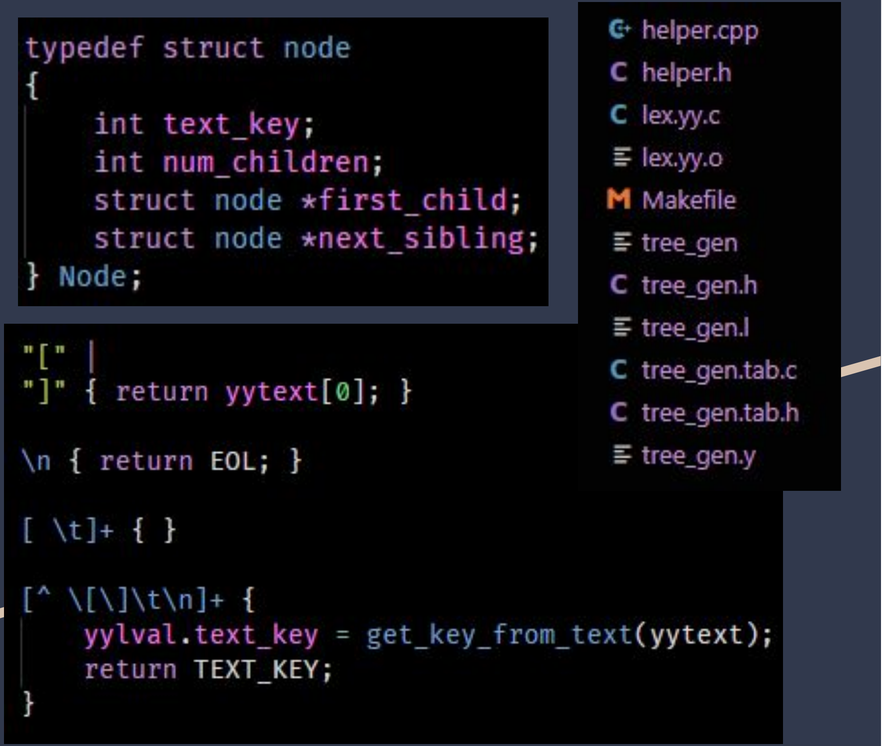


```
[a [b text] [c text]]
```

```
graph g {
        graph [bb="0,0,198,180"];
        node [label="\N"];
        ...
```

```
dot -T<format> -o output.<format>
```

Structure of Program



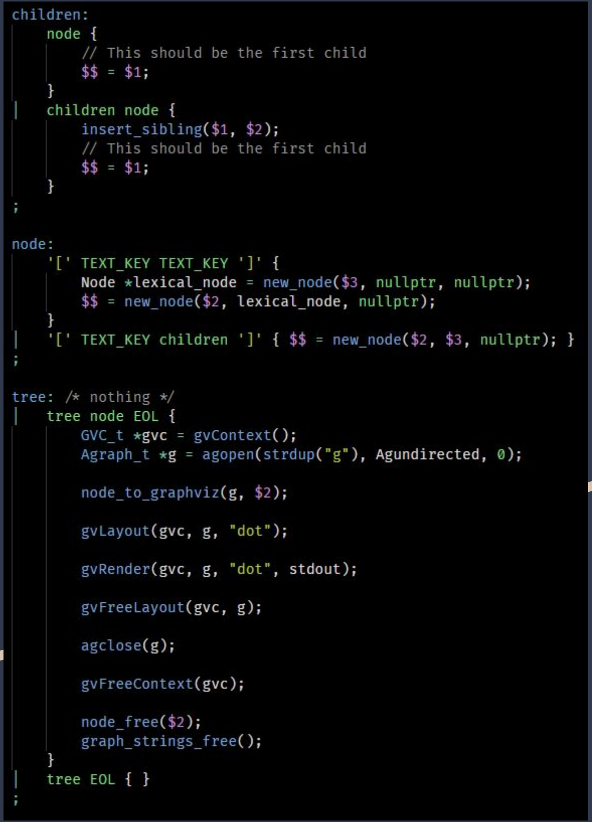
```
typedef struct node
{
    int text_key;
    int num_children;
    struct node *first_child;
    struct node *next_sibling;
} Node;
```

```
helper.cpp
helper.h
lex.yy.c
lex.yy.o
Makefile
tree_gen
tree_gen.h
tree_gen.l
tree_gen.tab.c
tree_gen.tab.h
tree_gen.y
```

```
"[" |
"]" { return yytext[0]; }

\n { return EOL; }

[ \t]+ { }

[^ \[\]\t\n]+ {
    yylval.text_key = get_key_from_text(yytext);
    return TEXT_KEY;
}
```

- Flex code (tree_gen.l)
  - Only reserved characters are ‘[‘, ‘]’, and whitespace (‘\n’ ,’\t’)
  - Any other characters can be used
- Bison code (tree_gen.y)
  - Constructs m-ary (k-way) tree from input
  - Uses GVC and node_to_graphviz() to convert to final DOT format output
- Helper code (helper.cpp)
  - Functions for allocating nodes
  - node_to_graphviz()

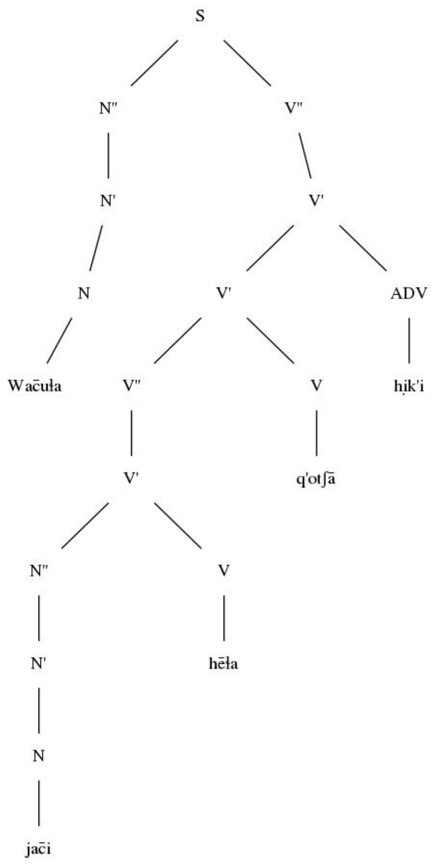
Grammar and Implementation



```
children:
    node {
        // This should be the first child
        $$ = $1;
    }
|   children node {
        insert_sibling($1, $2);
        // This should be the first child
        $$ = $1;
    }
;

node:
    '[' TEXT_KEY TEXT_KEY ']' {
        Node *lexical_node = new_node($3, nullptr, nullptr);
        $$ = new_node($2, lexical_node, nullptr);
    }
|   '[' TEXT_KEY children ']' { $$ = new_node($2, $3, nullptr); }
;

tree: /* nothing */
|   tree node EOL {
        GVC_t *gvc = gvContext();
        Agraph_t *g = agopen(strdup("g"), Agundirected, 0);

        node_to_graphviz(g, $2);

        gvLayout(gvc, g, "dot");

        gvRender(gvc, g, "dot", stdout);

        gvFreeLayout(gvc, g);

        agclose(g);

        gvFreeContext(gvc);

        node_free($2);
        graph_strings_free();
    }
|   tree EOL { }
;
```

- In the grammar, a node is represented as:
    - [text <text | NODE>]
- Root is the very first node
- In the program, a node may have multiple children
    - Node *next_sibling
- Store text in dictionary (map)
    - Store a text’s integer key in a node
- “tree” is analogous to the “calclist” from the simple calculator
- Graphviz Nodes are created by calling gvc’s “agnode(g, ...)”
    - Likewise, “agedge(g, ...)”

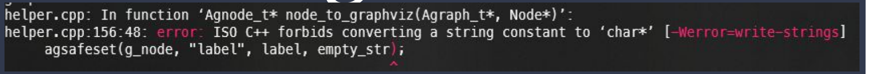
[S [N" [N' [N Wac̄ułа]]] [V" [V' [V' [V" [V' [N" [N' [N jac̄i]]] [V hēła]]] [V q'otʃā]] [ADV hịk'i]]]]



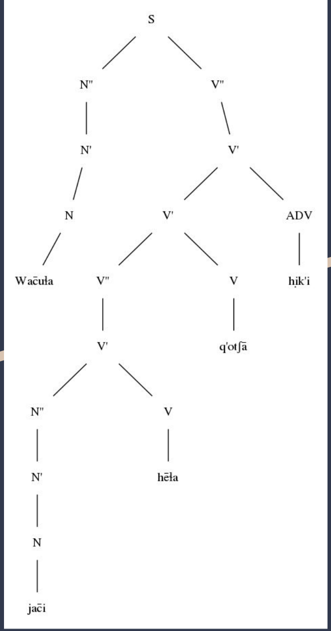
(Example is from Van Valin’s “An Introduction to Syntax”)

Example with Unicode characters. The example is in Tindi, a Northeast Caucasian language.

Problems and Challenges



```
helper.cpp: In function ‘Agnode_t* node_to_graphviz(Agraph_t*, Node*)’:
helper.cpp:156:48: error: ISO C++ forbids converting a string constant to ‘char*’ [-Werror=write-strings]
     agsafeset(g_node, "label", label, empty_str);
                                              ^
```



- Had to navigate an entire C library (GVC)
- GVC accepts “char *” when example problems pass in “const char *”
    - C++ does not like that, so I had to do work around it
- Difficult to make trees (graphs) look nice and balanced
- Grammar implementation was comparatively easier

Live Demo (if time)

End of Presentation