This technical note describes the data structures in the C# and Java implementations of the compiler generator Coco/R. The major data structures are:

- **The Symbol table** (Classes: Symbol). All terminals, pragmas and nonterminals in linear sequence. This data structure is trivial and therefore not further described.

- **The Syntax graph** (Classes: Node, Graph). The productions of the grammar as separate subgraphs. For every nonterminal sym there is a pointer sym.graph to the root of this symbol's syntax graph. A snapshot of this data structure is described in Section 1.

- **The Scanner automaton** (Classes: State, Action, Target, Melted). The DFA generated from token declarations. The token declarations are first translated to a syntax graph which is then transformed into a deterministic finite automaton. These steps are shown in Section 2.

- **The Character classes** (Class: CharClass). The character sets declared in the grammar stored as a linear list. This data structure is trivial and therefore not further explained.

- **The literals table** (Class: Tab). A mapping between token names and their literal representation.

### 1. Syntax Graph

**Production:** $A = (a \ (b) \ c \ | \ d \ [e] \ f \ | \ ) \ g$.

**Graph:**

Gray lines denote `next` pointers that point upwards. For any node $n$, if $n.next$ points upwards, then $n.up$ is true.
Operations to build the syntax graph

Tab.MakeFirstAlt(g)

\[
\begin{array}{c}
\text{g.l} \quad \text{g.r} \\
\circlearrowleft \quad \circlearrowleft
\end{array}
\quad \rightarrow
\quad
\begin{array}{c}
\text{g.l} \\
\text{g.r} \\
\text{alt}
\end{array}
\quad \rightarrow
\quad
\begin{array}{c}
\circlearrowleft
\end{array}
\text{connects right open ends of a graph}
\]

Tab.MakeAlternative(g1, g2)

\[
\begin{array}{c}
\text{g1.l} \\
\text{g1.r} \\
\text{alt}
\end{array}
\quad +
\quad
\begin{array}{c}
\text{g2.l} \\
\text{g2.r} \\
\text{alt}
\end{array}
\quad \rightarrow
\quad
\begin{array}{c}
\text{g1.l} \\
\text{g1.r} \\
\text{alt} \\
\text{alt}
\end{array}
\quad +
\quad
\begin{array}{c}
\text{g2.l} \\
\text{g2.r} \\
\text{alt}
\end{array}
\quad \rightarrow
\quad
\begin{array}{c}
\text{g1.l} \\
\text{g1.r} \\
\text{alt} \\
\text{alt}
\end{array}
\]

Tab.MakeSequence(g1, g2)

\[
\begin{array}{c}
\text{g1.l} \\
\text{g1.r} \\
\text{alt}
\end{array}
\quad +
\quad
\begin{array}{c}
\text{g2.l} \\
\text{g2.r} \\
\text{alt}
\end{array}
\quad \rightarrow
\quad
\begin{array}{c}
\text{g1.l} \\
\text{g1.r} \\
\text{alt} \\
\text{alt}
\end{array}
\quad +
\quad
\begin{array}{c}
\text{g2.l} \\
\text{g2.r} \\
\text{alt}
\end{array}
\quad \rightarrow
\quad
\begin{array}{c}
\text{g1.l} \\
\text{g1.r} \\
\text{alt} \\
\text{alt}
\end{array}
\]

Tab.MakeOption(g)

\[
\begin{array}{c}
\text{g.l} \\
\text{g.r}
\end{array}
\quad \rightarrow
\quad
\begin{array}{c}
\text{g.l} \\
\text{g.r} \\
\text{opt}
\end{array}
\]

Tab.MakeIteration(g)

\[
\begin{array}{c}
\text{g.l} \\
\text{g.r}
\end{array}
\quad \rightarrow
\quad
\begin{array}{c}
\text{g.l} \\
\text{g.r} \\
\text{iter}
\end{array}
\]

Tab.Finish(g)

\[
\begin{array}{c}
\text{g1.l} \\
\text{g1.r} \\
\text{alt}
\end{array}
\quad \rightarrow
\quad
\begin{array}{c}
\text{g1.l} \\
\text{g1.r} \\
\text{alt}
\end{array}
\quad \rightarrow
\quad
\begin{array}{c}
\text{g1.l} \\
\text{g1.r} \\
\text{alt}
\end{array}
\]

2. Scanner automaton

Declarations

CHARACTERS
- digit = '0'..'9'.
- hex = digit + 'a'..'f'.

TOKENS
- number = digit (digit).
- hexnum = digit (digit) 'H'.
- special = "0x".

Syntax graph for the tokens

The bold numbers denote the states that were assigned to the nodes by the method DFA.NumberNodes. They are used to derive the automaton from the graph as follows: if a node for a character or a character class c has the number n and its next pointer points to a node with a number m, then this leads to a transition

\[ n \xrightarrow{c} m \]

If there is no next node, the transition leads to a new state.

Nondeterministic automaton

The automaton is nondeterministic since there are three transitions with '0' in state 0 and two with digit in state 0. The first step in making the automaton deterministic is to split overlapping character ranges. This is done by DFA.MakeUnique.
After MakeUnique

The next step is to melt those states that can be reached by a transition with the same symbol from the same state. This is done in `DFA.MeltStates`.

After MeltStates

The only remaining task now is to delete the redundant states (here 1, 2 and 4).

After DeleteRedundantStates

This is the resulting deterministic finite automaton from which the scanner is generated.
Concrete data structures

Nondeterministic automaton

States

Actions

Targets

After MakeUnique

This means: from state 0 one can go with the characters 1..9 to state 1 and 2, and with the character 0 to state 1, 2 and 4.
After MeltStates

The states 1 and 2 have been "melted" into a new state 6, the states 1, 2 and 4 have been melted into a new state 7. This information is kept in class Melted using the following data structure:

The literals table

If a token is explicitly declared as a string, e.g.:

```
TOKENS
  while = "while".
  ...
```

it can be referenced in the productions both by its name (while) and by its literal representation ("while"). The symbol table just stores the names of such tokens. The hash table Tab.literals is used to map their literal representation to their node in the symbol table.