Review: Scanning & Parsing

- **Scanner** (also known as lexer) breaks the program up into individual words with little interest in structure
  - **Input:** A sequence of characters
  - **Output:** tokens from the language
  - Separate from parsing for historical reasons, but also for optimization and convenience

- **Parser** takes the parts, interprets them **structurally** based on the BNF/EBNF grammar specification
  - **Input:** tokens
  - **Output:** parse (or syntax) tree, giving the essential logical structure of the program

Review: A Grammar for Tokens

- Like overall syntax, the tokenization process begins with a formal specification of legal 'words', including active code and comments
  - Part of scanning process is stripping out comments to simplify the later stages of scanning and further compilation
  - Scanner will identify tokens (right-hand side of grammar), and label them with their roles (left-hand side)

\[
\begin{align*}
\text{assign} & \rightarrow = \\
\text{plus} & \rightarrow + \\
\text{minus} & \rightarrow - \\
\text{times} & \rightarrow * \\
\text{div} & \rightarrow / \\
\text{id} & \rightarrow \text{letter} (\text{letter} \mid \text{digit})* \\
\text{number} & \rightarrow \text{digit} \text{digit}* | \text{digit} (\text{. digit} | \text{digit . digit}) \text{digit}* \\
\text{comment} & \rightarrow */* (\text{non-}*) | */* (\text{non-}*)* */* | \text{/* (non-newline)* newline}
\end{align*}
\]
Given a program in our language, the DFA will:
1. Ignore all comments (returning to Start after the end of any such line/block)
2. Reach an accept state on any legal token (after which it will either terminate or return to Start if more text remains)

A program that uses such a structure will have different return conditions for each possible accepting state it reaches.

For each operator, for example, it will return back the appropriate label for that single basic token.

While it is possible to build a grammar (and so a DFA) that recognizes an id and a keyword as separate tokens, this will tend to be more complex than necessary to handle the ambiguity.

A more practical approach is to supplement the code with a simple ad hoc look-up routine: after reaching state 16 and encountering white-space, we simply check at that point if the token is read, write, or something else, and return the appropriate label.
A DFA to Scan the Calculator Language

While in some languages, scanning complete words can be quite complex, our simple calculator language uses white-space to separate tokens. Our scanner will signal an error in the code if it ever reaches a non-accepting state and encounters white-space.

For example, in this line:
\[ A := 5 + 3 \]
the scanner will:
1. Read the first token (‘A’) correctly
2. Reset to Start
3. Reach state 11 and signal an error, due to misplaced white-space

Thus, the following will be read as 3 distinct and legal tokens:
\[ 33A44 \]
While the following is an error
\[ 33A44 \]
since we will reach state 14 and have nowhere to go on the symbol ‘A’

Table-Driven Scanners

- Instead of the grammar ⇒ NFA ⇒ DFA approach, many tools take in a grammar and generate code that works with an associated set of look-up tables
- This is often more cumbersome for human beings to derive, and harder for them to understand
- From the point of view of a computer, it will work just as easily, if not more so
- Common tools like lex and flex use this approach

Note: the scanner will not do any complex syntax analysis beyond eliminating comments and labeling each of the legal tokens it encounters.

Thus, a line like the following is treated as containing 3 legal tokens:
\[ A + = 2 \]
Later, the parser may determine that this is not a proper line of code, but that is not the scanner’s job.
A scanner will start in state 1 and begin processing characters, switching between states (rows of the table) based upon those inputs.

Given a complete token, properly separated by white-space as before, the table-driven scanner returns that token, along with the label (found in the sub-table on the right) given for its current state/row.

If the scanner encounters an incomplete token, it will end up in a state with no label, and again return an error message.

### Tables for the Calculator Language

<table>
<thead>
<tr>
<th>State</th>
<th>digit</th>
<th>letter</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In some states, multiple different characters generate different transitions. In others, very specific transitions are allowed; all others are errors.

For example, any occurrence of “:” must be followed by “=” to generate a complete assignment operator.
Tables for the Calculator Language

<table>
<thead>
<tr>
<th>State</th>
<th>digit</th>
<th>upper</th>
<th>lower</th>
<th>space</th>
<th>tab</th>
<th>newline</th>
<th>Current input character</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>31</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>31</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>31</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>31</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>31</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>31</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>31</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>31</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>31</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>31</td>
<td>11</td>
</tr>
<tr>
<td>13</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>31</td>
<td>12</td>
</tr>
<tr>
<td>14</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>31</td>
<td>13</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>31</td>
<td>14</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>31</td>
<td>15</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>15</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>31</td>
<td>16</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>15</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>31</td>
<td>17</td>
</tr>
</tbody>
</table>

As with a DFA, a table-driven scanner program will also have supplementary look-up for keywords:
- Upon encountering identifier label, it will check a separate table of keywords (read and write), ensuring it labels the token correctly.

From Scanning to Parsing
- Once basic tokens are identified, complex syntactic structure can more easily be identified.
- Early results in programming languages showed that for any language defined by a CFG, a parser exists that runs in time \(O(n^3)\), where \(n\) is the length of the program.
- Early algorithms (late 60’s and early 70’s) achieved this bound.
- Such cubic-time is too slow for our largest programs.
  - Windows OS is estimated at about 50 million lines of code.
  - At a nanosecond per line to parse, a cubic parser would take just under 4 million years to read the source!
  - Better (linear time) results are possible if grammars are restricted to specific forms.

Efficiently Parsed Grammars
- Two classes of grammar (LL/LR) can be parsed efficiently.
  - In each case, a parser starts with a grammar of the given type.
  - Code is read, and a parse tree that corresponds to a derivation of that piece of code in the grammar:
    1. The first letter (L): indicates that input code is read in a Left-to-right direction.
    2. The second letter (L/R): indicates whether the corresponding derivation will be Left-most or Right-most.

Each algorithm builds a parse tree algorithmically:
1. LL: tree is built top-down, with the algorithm using the grammar and expression to predict which production rules to use wherever there is a choice.
2. LR: tree is built bottom-up, working from the leaves (terminal symbols) and reducing expressions back to the tree root (start symbol).
Top-Down Parsing

- Suppose we have a simple language of basic expressions:
  
  \[
  
  \begin{align*}
  \text{Exp} & \rightarrow a \text{ Tail} \mid b \text{ Tail} \\
  \text{Tail} & \rightarrow + a \text{ Tail} \mid + b \text{ Tail} \mid ;
  \end{align*}
  \]

- We can parse expression: `a + b + a;` in a top-down manner, starting with the root node (Exp) and ending with the full tree, built in a left-most fashion:

```
     Exp
       a
      +
     Tail
     +
  Tail
    +
 a
```

Bottom-Up Parsing

- To build bottom-up parse tree, the algorithm reads the expression left-to-right isolating leaf-nodes (terminals)

```
     Exp
       a
      +
     b
    +
 a
```

- Each of these is a single-node sub-tree of its own (6 total)

- The algorithm looks for the first complete expression (the complete right-hand side of some rule)

```
     Exp
       a
      +
     Tail
    +
 a
```

- Thus the last symbol (`;`), is used to generate a more complex sub-tree

```
     Exp
       a
      +
     b
    +
 a
```

- Now, the last three symbols together are a complete expression

```
     Exp
       a
      +
 b
    +
 a
```

```
     Exp
       a
      +
 b
    +
 a
```

Top-Down Parsing

```
     Exp
       a
      +
     b
    +
 a
```

```
     Exp
       a
      +
 b
    +
 a
```

Bottom-Up Parsing

- Now, a more complex tree can again be built of the 3 right-most sub-trees, to form a complete right-hand side of a rule

```
     Exp
       a
      +
 b
    +
 a
```

- The process will continue, ending with the same parse tree given by the top-down algorithm

```
     Exp
       a
      +
 b
    +
 a
```

```
     Exp
       a
      +
 b
    +
 a
```
Predictive Top-Down Parsing

- The top-down parser uses a predictive algorithm.
- It starts with the ‘Exp’ symbol, and the rule:
  \[ \text{Exp} \rightarrow a \text{Tail} | b \text{Tail} \]
- Given two/more options, the algorithm bases its choice upon the particular expression to be parsed/derived.
  - In this grammar, this is straightforward.
  - The leading symbol of any sub-expression (‘a’ or ‘b’) will determine the correct choice exactly.
- In general, for efficient top-down (LL) parsing to be possible, it must be possible to predict every rule application with only a fixed amount of look-ahead.
  - A top-down parser that can look ahead \( n \) tokens into the input string to make its decisions is designated an LL(\( n \)) parser.
  - Most commercial parsers are LL(1), and use a single token at most.

Non-Predictive Bottom-Up Parsing

\[ \text{Exp} \rightarrow a \text{Tail} | b \text{Tail} \]
\[ \text{Tail} \rightarrow + a \text{Tail} | + b \text{Tail} | ; \]

- The bottom-up parser for this grammar never has to make any predictions as it builds its trees.
  - Right-hand sub-trees either form complete expressions or not.
  - This process of building by scanning the input and collapsing subtrees together when possible is called shift-reduce parsing.
- One disadvantage of this grammar is space inefficiency.
  - As seen for the expression ‘\( a + b + a; \)’, the parser needs to read all the way to the end of the expression before it can do any subtree reduction.
  - On very complex expressions this can be problematic.

This Week

- **Topic:** Scanning/Parsing
- **Read:** Text, 2.2–2.3
- **Homework 01:** Posted to D2L last Friday
  - **Due:** Friday, 22 February, 5:00 PM on D2L
- **Office Hours:** Wing 210
  - Monday, 8:45 AM – 9:45 AM
  - Wednesday: 9:00 AM – 10:30 AM
  - Tuesday: 3:00 PM – 4:00 PM
  - Thursday: 2:00 PM – 3:00 PM
  - Friday: 9:00 AM – 10:30 AM