Project #1 Notes

- Project involves generating an AST for Python dialect.
- Our tools provide extended BNF (BNF + regular-expression notations like '*', '+', and '?') both for context-free and lexical definitions.
- Tools also provide largely automatic AST building:
  - Tokens double as AST operators.
  - By default, each rule computes the list of all trees built by its right-hand side.
  - The '^' notation allows you to build a tree designating the operator.
  - Or, in an action, you can use '$^ (...) $' to build an AST node, and '$*' to denote the list of children's ASTs.
- We've also provided methods to print nodes.

Project #1 Notes (II)

- In my solution, a majority of grammar rules look like this:
  ```
  attributeref: primary "."! identifier
  { $$ = $^ (ATTRIBUTEREF, $*); } 
  
  and all the printing, etc. is taken care of.
  ```
- Dummy tokens like ATTRIBUTEREF are first defined with
  ```
  %token ATTRIBUTEREF "@attributeref"
  ```
- In a few cases, I can just write
  ```
  expr1 : expr1 "or" expr1
  ```
  and the action is generated automatically.

A Little Notation

Here and in lectures to follow, we'll often have to refer to general productions or derivations. In these, we'll use various alphabets to mean various things:

- Capital roman letters are nonterminals (A, B, ...).
- Lower-case roman letters are terminals (or tokens, characters, etc.)
- Lower-case greek letters are sequences of zero or more terminal and nonterminal symbols, such as appear in sentential forms or on the right sides of productions (α, β, ...).
- Subscripts on lower-case greek letters indicate individual symbols within them, so α = α₁α₂...αₙ and each αᵢ is a single terminal or nonterminal.

For example,

- A : α might describe the production e : e '^' t,
- B ⇒ αAγ ⇒ αβγ might describe the derivation steps e ⇒ e '^' t ⇒ e '^' ID (α is e '^'; A is t; B is e; and γ is empty.)
Fixing Recursive Descent

- First, let’s define an impractical but simple implementation of a top-down parsing routine.
- For nonterminal $A$ and string $S=c_{1}c_{2}\ldots c_{n}$, we’ll define $parse(A,S)$ to return the length of a valid substring derivable from $A$.
- That is, $parse(A,c_{1}c_{2}\ldots c_{n}) = k$, where

$$c_{1}c_{2}\ldots c_{k}c_{k+1}c_{k+2}\ldots c_{n}\overset{A}{\Rightarrow}$$

### Example

Consider parsing $S=\text{"ID*ID-"}$ with a grammar from last time:

- $p : e \ ' - ' t$
- $e : t$
- $| e \ ' / ' t$
- $| e \ ' * ' t$
- $t : \text{ID}$

A successful path through the program:

```
parse(p, S):
  "Assuming $A$ is a nonterminal and $S = c_{1}c_{2}\ldots c_{n}$ is a string, return integer $k$ such that $A$ can derive the prefix string $c_{1}\ldots c_{k}$ of $S$."

  Choose production $A : \alpha_{1}\alpha_{2}\ldots \alpha_{m}$ for $A$ (nondeterministically)
  \begin{itemize}
    \item $k = 0$
    \item for $x \in \alpha_{1}, \alpha_{2}, \ldots, \alpha_{m}$:
      \begin{itemize}
        \item if $x$ is a terminal:
          \begin{itemize}
            \item if $x = c_{k+1}$:
              \begin{itemize}
                \item $k += 1$
              \end{itemize}
            \item else:
              \begin{itemize}
                \item GIVE UP
              \end{itemize}
          \end{itemize}
        \item else:
          \begin{itemize}
            \item $k += parse(x, c_{k+1}\ldots c_{n})$
          \end{itemize}
      \end{itemize}
  \end{itemize}
  return $k$
```

- Assume that the grammar contains one production for the start symbol: $p : \gamma \rightarrow$.
- We’ll say that a call to $parse$ returns a value if some set of choices for productions (the blue step) would return a value (just like NFA).
- Then if $parse(p, S)$ returns a value, $S$ must be in the language.

### Making a Deterministic Algorithm

- If we had an infinite supply of processors, could just spawn new ones at each “Choose” line.
- Some would give up, some loop forever, but on correct programs, at least one processor would get through.
- To do this for real (say with one processor), need to keep track of all possibilities systematically.
- This is the idea behind Earley’s algorithm:
  - Handles any context-free grammar.
  - Finds all parses of any string.
  - Can recognize or reject strings in $O(N^{3})$ time for ambiguous grammars, $O(N^{2})$ time for “nondeterministic grammars”, or $O(N)$ time for deterministic grammars (such as accepted by Bison).
Earley's Algorithm: I

• First, reformulate to use recursion instead of looping. Assume the string \( S = c_1 \cdots c_n \) is fixed.

• Redefine `parse`:
  \[
  \text{parse} (A: \alpha \beta, s, k): \\
  \begin{array}{ll}
  \text{Assumes } A: \alpha \beta \text{ is a production in the grammar,} \\
  0 \leq s \leq k \leq n, \text{ and } \alpha \text{ can produce the string } c_{s+1} \cdots c_k. \\
  \text{Returns integer } j \text{ such that } \beta \text{ can produce } c_k \cdots c_j. \\
  \end{array}
  \]

• Or diagrammatically, `parse` returns an integer \( j \) such that:

\[
\begin{array}{c}
  c_1 \cdots c_s c_{s+1} \cdots c_k c_{k+1} \cdots c_j c_{j+1} \cdots c_n \\
  \alpha \beta \Rightarrow \beta \Rightarrow \\
  0 \leq s \leq k \leq n
\end{array}
\]

Earley's Algorithm: II

\[
\text{parse} (A: \alpha \beta, s, k): \\
\text{Assumes } A: \alpha \beta \text{ is a production in the grammar,} \\
0 \leq s \leq k \leq n, \text{ and } \alpha \text{ can produce the string } c_{s+1} \cdots c_k. \\
\text{Returns integer } j \text{ such that } \beta \text{ can produce } c_k \cdots c_j. \\
\]

```python
if \beta \text{ is empty:} \\
  \text{return } k
```

Assume \( \beta \) has the form \( x\delta \)

```python
if x \text{ is a terminal:} \\
  if x == c_{k+1}: \\
    \text{return parse}(A: \alpha x, s, k+1) \\
  else: \\
    \text{GIVE UP}
else: \\
  \text{Choose production } 'x: \kappa' \text{ for } x \text{ (nondeterministically)} \\
  j = \text{parse}(x: \kappa, k, k) \\
  \text{return parse}(A: \alpha x, s, j)
```

• Now do all possible choices that result in such a way as to avoid redundant work ("nondeterministic memoization").

Chart Parsing

• Idea is to build up a table (known as a chart) of all calls to `parse` that have been made.

• Only one entry in chart for each distinct triple of arguments \( (A: \alpha \beta, s, k) \).

• We’ll organize table in columns numbered by the \( k \) parameter, so that column \( k \) represents all calls that are looking at \( c_k \) in the input.

• Each column contains entries with the other two parameters: \( [A: \alpha \beta, s] \), which are called *items*.

• The columns, therefore, are *item sets*.

Example

**Grammar**

<table>
<thead>
<tr>
<th>( p )</th>
<th>( e ) ( \rightarrow ) ( \cdot )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e )</td>
<td>( s ) ( I )</td>
</tr>
<tr>
<td>( s )</td>
<td>( \rightarrow )</td>
</tr>
</tbody>
</table>

**Input String**

\(- I + I - I\)

**Chart.** Heading are values of \( k \) and \( c_{k+1} \) (raised symbols).

\[
\begin{array}{c|c|c|c|c|c}
0 & - & I & 2 & + & 3 \\
\hline
a. p: & e \rightarrow \cdot, 0 & e. s: & \rightarrow \cdot s, 0 & g. e: & e \rightarrow + e, 0 \\
b. e: & e \rightarrow + e, 0 & f. e: & s \rightarrow I, 0 & h. e: & e \rightarrow + e, 0 \\
c. e: & e \rightarrow e, 0 & k. s: & \rightarrow e, 3 \\
d. s: & \rightarrow I, 0 & i. e: & \rightarrow s \rightarrow I, 3 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c}
4 & - & 5 & p. p: & e \rightarrow \cdot, 0 \\
\hline
m. e: & s \rightarrow I, 3 & p. p: & e \rightarrow \cdot, 0 \\
a. e: & e \rightarrow + e, 0 & a. e: & e \rightarrow e, 0 \\
a. p: & e \rightarrow \cdot, 0 \\
\end{array}
\]
Example, completed

- Last slide showed only those items that survive and get used. Algorithm actually computes dead ends as well (unlettered, in red).

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ap</td>
<td>$e \cdot ' - ' \cdot$, 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ae</td>
<td>$e \cdot ' + ' \cdot e$, 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c e</td>
<td>$e \cdot s \cdot I$, 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d s</td>
<td>$'$-, 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>$e \cdot s \cdot I$, 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>$e \cdot ' + ' \cdot e$, 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>a s</td>
<td>$s \cdot I$, 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p e</td>
<td>$e \cdot ' - ' \cdot$, 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i e</td>
<td>$e \cdot ' + ' \cdot e$, 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j e</td>
<td>$e \cdot s \cdot I$, 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k s</td>
<td>$s \cdot I$, 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l e</td>
<td>$e \cdot ' - ' \cdot$, 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o p</td>
<td>$e \cdot ' + ' \cdot e$, 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adding Semantic Actions

- Pretty much like recursive descent. The call $\text{parse}(A: \alpha \cdot \beta, s, k)$ can return, in addition to $j$, the semantic value of the $A$ that matches characters $c_{s+1} \cdots c_j$.
- This value is actually computed during calls of the form $\text{parse}(A: \alpha' \cdot, s, k)$ (i.e., where the $\beta$ part is empty).
- Assume that we have attached these values to the nonterminals in $\alpha$, so that they are available when computing the value for $A$.

Ambiguity

- Ambiguity only important here when computing semantic actions.
- Rather than being satisfied with a single path through the chart, we look at all paths.
- And we attach the set of possible results of $\text{parse}(Y: \cdot \kappa, s, k)$ to the nonterminal $Y$ in the algorithm.