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.
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 (\(A, B, \ldots\)).
- **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 (\(\alpha, \beta, \ldots\)).
- **Subscripts** on lower-case greek letters indicate individual symbols within them, so \(\alpha = \alpha_1\alpha_n \ldots \alpha_n\) and each \(\alpha_i\) is a single terminal or nonterminal.

For example,

- \(A : \alpha\) might describe the production \(e : e \ '++' t\),
- \(B \Rightarrow \alpha A\gamma \Rightarrow \alpha\beta\gamma\) might describe the derivation steps \(e \Rightarrow e \ '++' t \Rightarrow e \ '++' \text{ID}\) (\(\alpha\) is \(e \ '++'\); \(A\) is \(t\); \(B\) is \(e\); and \(\gamma\) 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 $S = c_1c_2\ldots c_n$, we’ll define $\text{parse}(A, S)$ to return the length of a valid substring derivable from $A$.
- That is, $\text{parse}(A, c_1c_2\ldots c_n) = k$, where

$$
\underbrace{c_1c_2\ldots c_k}_{A \Rightarrow^*} c_{k+1}c_{k+2}\ldots c_n
$$
Abstract body of parse(A,S)

• Can formulate top-down parsing analogously to NFAs.

\[
\text{parse (A, S):}
\]

"""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)

\[
k = 0
\]

for x in \( \alpha_1, \alpha_2, \ldots, \alpha_m \):

if x is a terminal:

if x == \( c_{k+1} \):

\[
k += 1
\]

else:

GIVE UP

else:

\[
k += \text{parse (x, } c_{k+1} \ldots c_n)\]

return k

• Assume that the grammar contains one production for the start symbol: \( p: \gamma \rightarrow \cdot \).

• 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.
Example

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

\[
\begin{align*}
  p & : e \ '\rightarrow' \\
  e & : t \\
       & | e \ '/' t \\
       & | e \ '*' t \\
  t & : \text{ID}
\end{align*}
\]
Example

Consider parsing $S =$ "ID*ID-1" with a grammar from last time:

\[
\begin{align*}
p : & \ e \ ' \rightarrow ' \\
e : & \ t \\
| & \ e \ '/ ' \ t \\
| & \ e \ '*' \ t \\
t : & \ ID
\end{align*}
\]

$k_i$ means "the variable $k$ in the call to parse that is nested $i$ deep." Outermost $k$ is $k_1$.

A failing path through the program:

```plaintext
parse(p, S):
  Choose p : e '→' :
    parse(e, S):
      Choose e : t:
        parse(t, S):
          choose t : ID:
            check S[1] == ID; OK, so $k_3$ += 1;
            return 1 (= $k_3$; added to $k_2$)
        return 1 (and add to $k_1$)
```
Consider parsing $S = "ID*ID-1"$ with a grammar from last time:

$$
\begin{align*}
p & : e '−' \\
e & : t \\
  & \mid e '/' t \\
  & \mid e '*' t \\
t & : ID \\
\end{align*}
$$

$k_i$ means “the variable $k$ in the call to parse that is nested $i$ deep.” Outermost $k$ is $k_1$. Likewise for $S$.

A successful path through the program:

```
p : e '−'
  parse(p, S):
    Choose p : e '−':
    parse(e, S):
      Choose e : e '*' t:
      parse(e, S):
        choose e : t:
        parse(t, S):
          choose t : ID:
            check $S[1] == ID$; OK, so return 1
            return 1 (so $k_2 += 1$)
            check $S[k_2] == '*'$; OK, $k_2 += 1$
          parse(t, S$: S_3$)
            # $S_3 == "ID −1"
            choose t : ID:
              check $S_3[k_3+1] == S_3[1] == ID$; OK
              $k_3+1; return 1$ (so $k_2 += 1$)
              return 3
            Check $S[k_1+1] == S[4] == '−'$: OK
            $k_1 +=1; return 4$
```
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`:

  ```python
  parse (A: $\alpha \cdot \beta$, s, k):
  """Assumes A: $\alpha \beta$ is a production in the grammar,  
  0 <= s <= k <= n, and $\alpha$ can produce the string $c_{s+1} \cdots c_k$.
  Returns integer j such that $\beta$ can produce $c_{k+1} \cdots c_j$.""
  ```

• Or diagrammatically, `parse` returns an integer $j$ such that:

\[
\begin{align*}
C_1 \cdots C_s &\underbrace{C_{s+1} \cdots C_k}_{\alpha^*} &\underbrace{C_{k+1} \cdots C_j}_{\beta^*} &C_{j+1} \cdots C_n
\end{align*}
\]
Earley's Algorithm: II

parse \((A: \alpha \bullet \beta, s, k)\):

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

if \(\beta\) is empty:
    return \(k\)
Assume \(\beta\) has the form \(x \delta\)
if \(x\) is a terminal:
    if \(x == c_{k+1}\):
        return parse\((A: \alpha x \bullet \delta, s, k+1)\)
    else:
        GIVE UP
else:
    Choose production \('x: \kappa'\) for \(x\) (nondeterministically)
    \(j = \text{parse}(x: \bullet \kappa, k, k)\)
    return parse \((A: \alpha x \bullet \delta, s, j)\)

• 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 \((A: \alpha \cdot \beta, s, k)\).

- We’ll organize table in columns numbered by the \(k\) parameter, so that column \(k\) represents all calls that are looking at \(c_{k+1}\) in the input.

- Each column contains entries with the other two parameters: \([A: \alpha \cdot \beta, s]\), which are called \textit{items}.

- The columns, therefore, are \textit{item sets}.
Example

Grammar

\[
\begin{align*}
p & : e ' - ' \\
e & : s I | e '+' e \\
s & : ' - ' |
\end{align*}
\]

Input String

\[- I + I -\]

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>I</th>
<th>2</th>
<th>+</th>
<th>3</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. p:</td>
<td>( e ' - ' ), 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b. e:</td>
<td>( e '+' e ), 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c. e:</td>
<td>( s I ), 0</td>
<td></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>
<td></td>
</tr>
<tr>
<td>e. s:</td>
<td>( ' - ' ), 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f. e:</td>
<td>( sI ), 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g. e:</td>
<td>( sI ), 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h. e:</td>
<td>( e '+' e ), 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i. e:</td>
<td>( e '+' e ), 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j. e:</td>
<td>( sI ), 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k. s:</td>
<td>( \cdot ), 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l. e:</td>
<td>( sI ), 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>m. e:</td>
<td>( sI ), 3</td>
<td></td>
</tr>
<tr>
<td>n. e:</td>
<td>( e '+' e ), 0</td>
<td></td>
</tr>
<tr>
<td>o. p:</td>
<td>( e '+' ), 0</td>
<td></td>
</tr>
<tr>
<td>p. p:</td>
<td>( e ' - ' ), 0</td>
<td></td>
</tr>
</tbody>
</table>
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>-</th>
<th>1</th>
<th>I</th>
<th>2</th>
<th>+</th>
<th>3</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.</td>
<td>p:</td>
<td>e</td>
<td>’\−’</td>
<td>0</td>
<td>e.</td>
<td>s:</td>
<td>’\−’</td>
<td>0</td>
</tr>
<tr>
<td>b.</td>
<td>e:</td>
<td>e</td>
<td>’+’</td>
<td>e, 0</td>
<td>f.</td>
<td>e:</td>
<td>s</td>
<td>I, 0</td>
</tr>
<tr>
<td>c.</td>
<td>e:</td>
<td>s</td>
<td>I, 0</td>
<td></td>
<td>d.</td>
<td>s:</td>
<td>’\−’</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>e:</td>
<td>s</td>
<td>I, 0</td>
<td></td>
<td></td>
<td>e:</td>
<td>s</td>
<td>I, 0</td>
</tr>
<tr>
<td></td>
<td>e:</td>
<td>e</td>
<td>s</td>
<td></td>
<td></td>
<td>e:</td>
<td>e</td>
<td>I, 0</td>
</tr>
<tr>
<td></td>
<td>e:</td>
<td>e</td>
<td>s</td>
<td></td>
<td></td>
<td>e:</td>
<td>e</td>
<td>I, 0</td>
</tr>
<tr>
<td></td>
<td>e:</td>
<td>e</td>
<td>e</td>
<td></td>
<td></td>
<td>e:</td>
<td>e</td>
<td>s</td>
</tr>
<tr>
<td></td>
<td>e:</td>
<td>e</td>
<td>e</td>
<td></td>
<td></td>
<td>e:</td>
<td>e</td>
<td>e</td>
</tr>
</tbody>
</table>

Last modified: Wed Sep 19 10:34:38 2012
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 $\text{parse}(Y: \kappa, s, k)$ to the nonterminal $Y$ in the algorithm.