News & Info

- Who’s Hiring May 2016
  - [https://news.ycombinator.com/item?id=11611867](https://news.ycombinator.com/item?id=11611867)

- SoCal Code Camp | San Diego, CA 6/25-6/26
Administrivia

- Lab 06
  - Due Thursday
Predictive Parsers

- Like recursive-descent but parser can “predict” which production to use
  - By looking at the next few tokens
  - No backtracking

- Predictive parsers accept LL(k) grammars
  - L means “left-to-right” scan of input
  - L means “leftmost derivation”
  - k means “predict based on k tokens of lookahead”
  - In practice, LL(1) is used
LL(1) vs Recursive Descent

- In recursive-descent,
  - At each step, many choices of production to use
  - Backtracking used to undo bad choices

- In LL(1),
  - At each step, only one choice of production
  - That is
    - When a non-terminal $A$ is leftmost in a derivation
    - The next input symbol is $\mathbf{t}$
    - There is a unique production $A \rightarrow \alpha$ to use
      - Or no production to use (an error state)

- LL(1) is a recursive descent variant without backtracking
Predictive Parsing and Left Factoring

• Recall the grammar
  
  \[ E \rightarrow T + E \mid T \]
  
  \[ T \rightarrow \text{int} \mid \text{int} \times T \mid (E) \]

• Hard to predict because
  
  - For \( T \) two productions start with \text{int}
  
  - For \( E \) it is not clear how to predict

• We need to \text{left-factor} the grammar
Left-Factoring Example

- Recall the grammar
  \[E \to T + E \mid T\]
  \[T \to \text{int} \mid \text{int} \ast T \mid ( E )\]

- Factor out common prefixes of productions
  \[E \to T X\]
  \[X \to + E \mid \varepsilon\]
  \[T \to ( E ) \mid \text{int} Y\]
  \[Y \to \ast T \mid \varepsilon\]
LL(1) Parsing Table Example

- Left-factored grammar
  \[ E \rightarrow TX \]
  \[ T \rightarrow (E) \mid \text{int } Y \]
  \[ X \rightarrow +E \mid \varepsilon \]
  \[ Y \rightarrow *T \mid \varepsilon \]

- The LL(1) parsing table:

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>E</strong></td>
<td>TX</td>
<td></td>
<td></td>
<td>TX</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>X</strong></td>
<td></td>
<td></td>
<td>+E</td>
<td></td>
<td>\varepsilon</td>
<td>\varepsilon</td>
</tr>
<tr>
<td><strong>T</strong></td>
<td>int Y</td>
<td></td>
<td></td>
<td>(E)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Y</strong></td>
<td></td>
<td>*T</td>
<td></td>
<td>\varepsilon</td>
<td>\varepsilon</td>
<td>\varepsilon</td>
</tr>
</tbody>
</table>

- leftmost non-terminal
- rhs of production to use
- next input token
LL(1) Parsing Table Example

- **Consider the [E, int] entry**
  - “When current non-terminal is E and next input is int, use production E → T X”
  - This can generate an int in the first position

- **Consider the [Y,+] entry**
  - “When current non-terminal is Y and current token is +, get rid of Y”
  - Y can be followed by + only if Y → ε
LL(1) Parsing Tables. Errors

- Blank entries indicate error situations

- Consider the \([E,*]\) entry
  - “There is no way to derive a string starting with \(*\) from non-terminal \(E\)”
Using Parsing Tables

- Method similar to recursive descent, except
  - For the leftmost non-terminal $S$
  - We look at the next input token $a$
  - And choose the production shown at $[S,a]$

- A stack records frontier of parse tree
  - Non-terminals that have yet to be expanded
  - Terminals that have yet to matched against the input
  - Top of stack = leftmost pending terminal or non-terminal

- Reject on reaching error state
- Accept on end of input & empty stack
LL(1) Parsing Algorithm

initialize stack = <S $> and next
repeat
  case stack of
    <X, rest> : if T[X,*next] = Y_1...Y_n
      then stack ← <Y_1... Y_n rest>;
    else error ()
    <t, rest> : if t == *next ++
      then stack ← <rest>;
    else error ()
  until stack == < >
LL(1) Parsing Algorithm

initialize stack = <S $> and next
repeat
case stack of
  <X, rest> : if T[X,*next] = Y_1...Y_n
    then stack ← <Y_1... Y_n rest>;
    else error ();
  <t, rest> : if t == *next ++
    then stack ← <rest>;
    else error ();
until stack == < >

$ marks bottom of stack
For non-terminal X on top of stack, lookup production
For terminal t on top of stack, check t matches next input token.
Pop X, push production rhs on stack. Note leftmost symbol of rhs is on top of the stack.
## LL(1) Parsing Example

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>E $</td>
<td>int * int $</td>
<td>T X</td>
</tr>
<tr>
<td>T X $</td>
<td>int * int $</td>
<td>int Y</td>
</tr>
<tr>
<td>int Y X $</td>
<td>int * int $</td>
<td>terminal</td>
</tr>
<tr>
<td>Y X $</td>
<td>* int $</td>
<td>* T</td>
</tr>
<tr>
<td>* T X $</td>
<td>* int $</td>
<td>terminal</td>
</tr>
<tr>
<td>T X $</td>
<td>int $</td>
<td>int Y</td>
</tr>
<tr>
<td>int Y X $</td>
<td>int $</td>
<td>terminal</td>
</tr>
<tr>
<td>Y X $</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>X $</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>$</td>
</tr>
</tbody>
</table>

**ACCEPT**
Constructing Parsing Tables

- Consider non-terminal $A$, production $A \to \alpha$, & token $t$
- $T[A,t] = \alpha$ in two cases:
  - If $\alpha \to^* t \beta$
    - $\alpha$ can derive a $t$ in the first position
    - We say that $t \in \text{First}(\alpha)$
  - If $A \to \alpha$ and $\alpha \to^* \varepsilon$ and $S \to^* \beta A \vdash \delta$
    - Useful if stack has $A$, input is $t$, and $A$ cannot derive $t$
    - In this case only option is to get rid of $A$ (by deriving $\varepsilon$)
      - Can work only if $t$ can follow $A$ in at least one derivation
    - We say $t \in \text{Follow}(A)$
Computing First Sets

Definition

\[ \text{First}(X) = \{ \tau \mid X \rightarrow^* \tau \alpha \} \cup \{ \varepsilon \mid X \rightarrow^* \varepsilon \} \]

Algorithm sketch:

1. \( \text{First}(\tau) = \{ \tau \} \)

2. \( \varepsilon \in \text{First}(X) \)
   - if \( X \rightarrow \varepsilon \)
   - if \( X \rightarrow A_1 ... A_n \) and \( \varepsilon \in \text{First}(A_i) \) for \( 1 \leq i \leq n \)

3. \( \text{First}(\alpha) \subseteq \text{First}(X) \) if \( X \rightarrow A_1 ... A_n \alpha \)
   - and \( \varepsilon \in \text{First}(A_i) \) for \( 1 \leq i \leq n \)
First Sets. Example

• Recall the grammar
  
  \[
  E \rightarrow TX \\
  T \rightarrow (E) | \text{int } Y \\
  X \rightarrow +E | \varepsilon \\
  Y \rightarrow *T | \varepsilon
  \]

• First sets
  
  \[
  \text{First( ( ) } = \{ ( ) \} \\
  \text{First( ) } = \{ ) \} \\
  \text{First( int ) } = \{ \text{int } \} \\
  \text{First( + ) } = \{ + \} \\
  \text{First( * ) } = \{ * \}
  \]

  \[
  \text{First( T ) } = \{ \text{int, ( )} \} \\
  \text{First( E ) } = \{ \text{int, ( )} \} \\
  \text{First( X ) } = \{ +, \varepsilon \} \\
  \text{First( Y ) } = \{ *, \varepsilon \}
  \]
Computing Follow Sets

• Definition:

\[ \text{Follow}(X) = \{ \dagger \mid S \xrightarrow{\ast} \beta X \dagger \delta \} \]

• Intuition

  - If \( X \rightarrow A B \) then \( \text{First}(B) \subseteq \text{Follow}(A) \) and \( \text{Follow}(X) \subseteq \text{Follow}(B) \)
  
  - if \( B \rightarrow \ast \varepsilon \) then \( \text{Follow}(X) \subseteq \text{Follow}(A) \)

  - If \( S \) is the start symbol then \( \$ \in \text{Follow}(S) \)
Algorithm sketch:

1. $\$ \in \text{Follow}(S)$
2. $\text{First}(\beta) - \{\varepsilon\} \subseteq \text{Follow}(X)$
   - For each production $A \rightarrow \alpha X \beta$
3. $\text{Follow}(A) \subseteq \text{Follow}(X)$
   - For each production $A \rightarrow \alpha X \beta$ where $\varepsilon \in \text{First}(\beta)$
Follow sets. Example

Recall the grammar

\[ E \rightarrow TX \]
\[ T \rightarrow (E) \mid \text{int } Y \]
\[ X \rightarrow +E \mid \epsilon \]
\[ Y \rightarrow *T \mid \epsilon \]

Follow sets

\[
\begin{align*}
\text{Follow}(+) &= \{ \text{int}, () \} \\
\text{Follow}(&)*) &= \{ \text{int}, () \} \\
\text{Follow}(()) &= \{ \text{int}, () \} \\
\text{Follow}(X) &= \{ $, ) \} \\
\text{Follow}(T) &= \{ +, ) , $ \} \\
\text{Follow}(Y) &= \{ +, ) , $ \} \\
\text{Follow}(\text{int}) &= \{ *, +, ) , $ \}
\end{align*}
\]
Constructing LL(1) Parsing tables

- Construct a parsing table $T$ for CFG $G$

- For each production $A \rightarrow \alpha$ in $G$ do:
  - For each terminal $t \in \text{First}(\alpha)$ do
    • $T[A, t] = \alpha$
  - If $\epsilon \in \text{First}(\alpha)$, for each $t \in \text{Follow}(A)$ do
    • $T[A, t] = \alpha$
  - If $\epsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(A)$ do
    • $T[A, \$] = \alpha$
Notes on LL(1) Parsing Tables

- If any entry is multiply defined then $G$ is not LL(1)
  - If $G$ is ambiguous
  - If $G$ is left recursive
  - If $G$ is not left-factored
  - And in other cases as well

- Most programming language CFGs are not LL(1)
Bottom-Up Parsing

- Bottom-up parsing is more general than top-down parsing
  - And just as efficient
  - Builds on ideas in top-down parsing

- Bottom-up is the preferred method

- Concepts today, algorithms next time
An Introductory Example

- Bottom-up parsers don’t need left-factored grammars

- Revert to the “natural” grammar for our example:
  
  \[
  E \rightarrow T + E \mid T \\
  T \rightarrow \text{int} \ast T \mid \text{int} \mid (E)
  \]

- Consider the string: \text{int} \ast \text{int} + \text{int}
The Idea

Bottom-up parsing reduces a string to the start symbol by inverting productions:

\[
\begin{align*}
\text{int} \ast \text{int} + \text{int} & \\
\text{int} \ast \text{T} + \text{int} & \\
\text{T} + \text{int} & \\
\text{T} + \text{T} & \\
\text{T} + \text{E} & \\
\text{E} & \\
\text{T} & \rightarrow \text{int} \\
\text{T} & \rightarrow \text{int} \ast \text{T} \\
\text{T} & \rightarrow \text{int} \\
\text{E} & \rightarrow \text{T} \\
\text{E} & \rightarrow \text{T} + \text{E}
\end{align*}
\]
Observation

- Read the productions in reverse (from bottom to top)
- This is a rightmost derivation!

\[
\begin{align*}
\text{int} \times \text{int} + \text{int} & \quad \text{T} \rightarrow \text{int} \\
\text{int} \times \text{T} + \text{int} & \quad \text{T} \rightarrow \text{int} \times \text{T} \\
\text{T} + \text{int} & \quad \text{T} \rightarrow \text{int} \\
\text{T} + \text{T} & \quad \text{E} \rightarrow \text{T} \\
\text{T} + \text{E} & \quad \text{E} \rightarrow \text{T} + \text{E} \\
\text{E} & \\
\end{align*}
\]
Important Fact #1 about bottom-up parsing:

A bottom-up parser traces a rightmost derivation in reverse.
A Bottom-up Parse
A Bottom-up Parse in Detail (1)

\[ \text{int } \ast \text{ int } \ast \text{ int } + \text{ int } \]
A Bottom-up Parse in Detail (2)

int * int + int

int * T + int

T

int * int + int

int
A Bottom-up Parse in Detail (3)

int * int + int
int * T + int
T + int
A Bottom-up Parse in Detail (4)
A Bottom-up Parse in Detail (5)

```
int * int + int
int * T + int
T + int
T + T
T + E
```

![Diagram](image)

```
T

E
```

```
int
*
int
+
int
```
A Bottom-up Parse in Detail (6)
A Bottom-up Parsing Algorithm

Let $I =$ input string

repeat

pick a non-empty substring $\beta$ of $I$

where $X \rightarrow \beta$ is a production

if no such $\beta$, backtrack

replace one $\beta$ by $X$ in $I$

until $I =$ "S" (the start symbol) or all possibilities are exhausted
Where do Reductions Happen?

**Important Fact #1 has an interesting consequence:**
- Let $\alpha\beta\omega$ be a step of a bottom-up parse
- Assume the next reduction is by $X \rightarrow \beta$
- Then $\omega$ is a string of terminals

Why? Because $\alpha X \omega \rightarrow \alpha \beta \omega$ is a step in a rightmost derivation.
Notation

- Idea: Split string into two substrings
  - Right substring is as yet unexamined by parsing (a string of terminals)
  - Left substring has terminals and non-terminals

- The dividing point is marked by a | 
  - The | is not part of the string

- Initially, all input is unexamined |x_1x_2 \ldots x_n|
Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

- **Shift**
- **Reduce**
Shift

• *Shift*: Move | one place to the right
  - Shifts a terminal to the left string

  $ABC|xyz \Rightarrow ABCx|yz$
Reduce

- Apply an inverse production at the right end of the left string
  - If $A \rightarrow xy$ is a production, then

$$Cbxy|ijk \Rightarrow CbA|ijk$$
The Example with Reductions Only

\[
\begin{align*}
\text{int} \times \text{int} &\mid + \text{int} \\
\text{int} \times \text{T} &\mid + \text{int} \\
\text{T} + \text{int} &\mid \\
\text{T} + \text{T} &\mid \\
\text{T} + \text{E} &\mid \\
\text{E} &\mid \\
\end{align*}
\]

\[
\begin{align*}
\text{reduce } \text{T} &\rightarrow \text{int} \\
\text{reduce } \text{T} &\rightarrow \text{int} \times \text{T} \\
\text{reduce } \text{T} &\rightarrow \text{int} \\
\text{reduce } \text{E} &\rightarrow \text{T} \\
\text{reduce } \text{E} &\rightarrow \text{T} + \text{E} \\
\end{align*}
\]
The Example with Shift-Reduce Parsing

```
|int * int + int          | shift |
int | * int + int            | shift |
int * | int + int            | shift |
int * int | + int         | reduce T → int |
int * T | + int     | reduce T → int * T |
T | + int    | shift |
T+ | int    | shift |
T + int |     | reduce T → int |
T + T |     | reduce E → T |
T + E |     | reduce E → T + E |
E |         |
```
A Shift-Reduce Parse in Detail (1)

\[ \text{int} \ast \text{int} + \text{int} \]

\[ \text{int} \ast \text{int} + \text{int} \]
A Shift-Reduce Parse in Detail (2)

\[
\begin{align*}
\text{int} &\quad \ast \quad \text{int} &\quad + &\quad \text{int} \\
\text{int} &\quad | &\quad \ast \quad \text{int} &\quad + &\quad \text{int}
\end{align*}
\]
A Shift-Reduce Parse in Detail (3)

\[ \text{int} \times \text{int} + \text{int} \]
\[ \text{int} \mid \times \text{int} + \text{int} \]
\[ \text{int} \times \mid \text{int} + \text{int} \]

\[ \text{int} \times \text{int} + \text{int} \]

A Shift-Reduce Parse in Detail (4)

```
| int * int + int |
| int | * int + int |
| int * | int + int |
| int * int | + int |
```

```
int * int + int
```

↑
A Shift-Reduce Parse in Detail (5)
A Shift-Reduce Parse in Detail (6)

| int * int + int
| int | * int + int
| int * | int + int
| int * int | + int
| int * T | + int
| T | + int

```
+---+
| T |
+---+
|    +---+
|      |   +---+
|       |     |     +---+
|       |     |     int  +---+
|       |     |     int
```

```
A Shift-Reduce Parse in Detail (7)

\[
\begin{align*}
| & \text{int } * \text{ int } + \text{ int} \\
\text{int} & | \text{ int } * \text{ int } + \text{ int} \\
\text{int} & | \text{ int } * \text{ int } | + \text{ int} \\
\text{int} & | \text{ int } * \text{ T } | + \text{ int} \\
\text{T} & | + \text{ int} \\
\text{T} & | + \text{ int} \\
\text{int} & | + \text{ int} \\
\text{int} & | + \text{ int} \\
\end{align*}
\]
A Shift-Reduce Parse in Detail (8)
A Shift-Reduce Parse in Detail (9)
A Shift-Reduce Parse in Detail (10)

\[
\begin{align*}
| & \text{int} \ast \text{int} + \text{int} \\
\text{int} & | \ast \text{int} + \text{int} \\
\text{int} & \ast \text{int} + \text{int} \\
\text{int} & \ast \text{T} + \text{int} \\
\text{T} & + \text{int} \\
\text{T} & + \text{int} \\
\text{T} + \text{T} & | \\
\text{T} + \text{E} & | \\
\end{align*}
\]

\[\text{T} \quad \ast \quad \text{int} \quad + \quad \text{int} \quad \text{E}\]
A Shift-Reduce Parse in Detail (11)
The Stack

- Left string can be implemented by a stack
  - Top of the stack is the |

- Shift pushes a terminal on the stack

- Reduce pops 0 or more symbols off of the stack (production rhs) and pushes a non-terminal on the stack (production lhs)
Conflicts

- In a given state, more than one action (shift or reduce) may lead to a valid parse

- If it is legal to shift or reduce, there is a \textit{shift-reduce} conflict

- If it is legal to reduce by two different productions, there is a \textit{reduce-reduce} conflict

- You will see such conflicts in your project!
  - More next time...