Final Exam Review

Administrative

• Final Exam on Thursday, April 28 @ 10:00 a.m. – 12:50 p.m. in HEC 118

Final Exam Review

• Please check out Dr. Hughes’ structure of the final exam:
  o It contains the complete outline of the final exam (at least in regards to the style of questions asked)
  o Very strong hints about what you should focus your studies on

Dr. Hughes’ Promises

1. An expression grammar that incorporates precedence and associativity  ⇒ Recitation #4
2. Distinction between languages and grammars in a particular class.  ⇒ Recitation #13
3. Ambiguity  ⇒ Recitation #4
4. FLEX type answer to a regular expression problem.  ⇒ Recitation #3
5. EBNF / Railroad chart question  ⇒ Recitations #1 and #5 for EBNF
6. Creation of a recursive descent parser for some simple construct.  ⇒ Recitations #8 and #12
7. Creation of FIRST, FOLLOW and an LL(1) parse table.  ⇒ Recitations #5 and #6 and see notes below
8. Removal of left recursion and common prefixes.  ⇒ Recitation #4
9. CKY Parsing Table.  ⇒ Recitations #6 and #8 and #13
10. Bottom-Up and Top-Down stack manipulation  ⇒ Recitations #5 and #9
11. Adding actions to Bison grammar, e.g., code generation, semantic error checks  ⇒ Recitation #7 and #13
12. Completion of the states, actions and gotos for an SLR(1) parser.  ⇒ Recitation #9 and #13
13. Completion of canonical LR(1) parser.  ⇒ See notes below
14. LALR(1) parser by doing merges on a canonical LR(1) parser's states.  ⇒ See notes below
15. Evaluation of attributes (inherited and synthesized) for some attributed translation grammar.  ⇒ Recitation #10 and notes below
16. Data flow algorithm based on one of the four discussed in class.  ⇒ See notes below
Regular Expressions

- Write a Flex-style regular expression for the following sets:
  - \( A = \{ w \mid w \) is over the alphabet \( \{a,b\} \) and \( w \) contains multiples of 3 b’s \}
    - You should realize that 0 is a multiple of 3, so the empty string is allowed
    - **Solution:**
      \[ a^*| (a^*ba^*ba^*)^* \]
  - \( B = \{ w \mid w \) is over the alphabet \( \{0,1\} \) and \( w \) contains the substring 110 \}
    - This is fairly trivial; whenever the question talks about a verbatim substring, it will definitely show up in the regular expression
    - Since there are no other constraints, the solution comes easy
    - **Solution:**
      \[ (0|1)^*110 (0|1)^*=[01]^*110[01]^* \]
  - \( C = \{ w \mid w \) is over the alphabet \( \{a,b\} \) and \( w \) does not end with \( ab \) \}
    - Any string in language over \( \{a,b\} \) must end in \( a \) or \( b \)
    - If it doesn’t end with \( ab \), then it ends with \( a \), or if it ends with \( b \) then the last \( b \)
    - must be preceded by another
    - The string in front of the last \( a \) or \( bb \) can be arbitrary
    - **Solution:**
      \[ (a|b)^*(a|bb) \]
  - \( D = \{ w \mid w \) is over the alphabet \( \{a,b\} \) and has an even length \}
    - Any string of even length can be expressed by concatenation of strings of length 2
    - With such a small alphabet, the strings of length 2 are \( aa,ab,ba,bb \).
    - Note that 0 is an even number → the empty string has to be in the language
    - **Solution:**
      \[ (aa|ab|ba|bb)^* \]
  - \( E = \{ w \mid w \) is over the alphabet \( \{a,b\} \) and \( w \) contains no more than one occurrence of \( aa \) \}
    - The one \( aa \) can be followed by any number of \( b \). If \( a \) comes after \( aa \), it must be preceded by a \( b \) (otherwise we would have two occurrences of \( aa \) → Any string that follows \( aa \) is represented by \( (b|ba)^* \)
    - If \( a \) comes before \( aa \), it must be followed by \( b \) → String preceding \( aa \) is \( (b|ab)^* \)
    - Hence if string contains exactly one \( aa \) it corresponds to \( (b|ab)^* aa(b|ba)^* \)
    - It there is no \( aa \), similar arguments for one \( a \) and \( \varepsilon \)
    - **Solution:**
      \[ (b|ab)^*(\varepsilon|a|aa)(b|ba)^* \]
**CKY Parser**

- Quickly go through example from recitation #13
- One more example from Dr. Hughes’ sample questions:
  - Present the CKY recognition matrix for the string “a – a + a – a” assuming the Chomsky Normal Form grammar specified by the rules

\[
\begin{align*}
E & \to EF \mid ME \mid PE \mid a \\
F & \to MF \mid PF \mid ME \mid PE \\
P & \to + \\
M & \to -
\end{align*}
\]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>–</th>
<th>a</th>
<th>+</th>
<th>a</th>
<th>–</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>E → a</td>
<td>M → –</td>
<td>E → a</td>
<td>P → +</td>
<td>E → a</td>
<td>M → –</td>
<td>E → a</td>
</tr>
<tr>
<td>2</td>
<td>E → ME</td>
<td>E → PE</td>
<td>E → ME</td>
<td>E → ME</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>E → EF</td>
<td>E → EF</td>
<td>E → EF</td>
<td>E → EF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>E → EF</td>
<td>E → ME</td>
<td>E → PE</td>
<td>E → ME</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>E → EF</td>
<td>E → EF</td>
<td>E → EF</td>
<td>E → EF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>E → EF</td>
<td>E → ME</td>
<td>E → PE</td>
<td>E → ME</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>E → EF</td>
<td>E → EF</td>
<td>E → EF</td>
<td>E → EF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The abbreviated form of the table, only containing the LHS of the grammar rules, is also acceptable.

**The string is accepted** since the start symbol $E$ appears in the final cell.
**LL(1) Parser**

- Consider the following grammar $G = (\{S, T\}, \{\alpha, ',', (, )\}, S, P)$ where $P$ is:
  
  $S \rightarrow a \mid (T)$
  
  $T \rightarrow T, S \mid S$

- Remove the **left recursion** from the given grammar:
  
  o Remember that general productions of the form $A \rightarrow A\alpha \mid \beta$ will be replaced by the non-left-recursive productions: $A \rightarrow \beta A'$ and $A' \rightarrow \alpha A' \mid \epsilon$
  
  o In our case: $A = T; \quad \alpha = S; \quad \beta = S$
  
  o Thus, the non-left-recursive grammar will take the following form:
    
    $S \rightarrow a \mid (T)$
    
    $T \rightarrow ST'$
    
    $T' \rightarrow ST' \mid \epsilon$

- Construct the **predictive parsing table** for the non-left-recursive grammar

  ![](algorithm.png)

  o Let’s calculate the FIRST and FOLLOW sets (see recitation #5 for more details):
    
    FIRST($S$) = FIRST($a \cup$ FIRST($T$)) = \{ $a, ( \}$
    
    FIRST($T$) = FIRST($ST'$) = FIRST($S$) = \{ $a, ( \}$
    
    FIRST($T'$) = FIRST($, ST'$) $\cup$ FIRST($\epsilon$) = \{ $', \epsilon \}$
    
    FOLLOW($T$) = \{ $) \}$
    
    FOLLOW($T'$) = FOLLOW($T$) = \{ $) \}$
    
    FOLLOW($S$) = \{ $\} \cup$ FIRST($T'$)$\setminus \{ \epsilon \}$ $\cup$ FOLLOW($T$) = \{ $, ', , ) \}$
Now we can build the parsing table according to the algorithm:

<table>
<thead>
<tr>
<th>Non-Terminals</th>
<th>Input Symbol</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$S \rightarrow a$</td>
<td>$(a, a, a)$</td>
<td>Output $S \rightarrow (T)$</td>
</tr>
<tr>
<td>$T$</td>
<td>$T \rightarrow ST'$</td>
<td>$S \rightarrow a'$</td>
<td></td>
</tr>
<tr>
<td>$T'$</td>
<td>$T' \rightarrow ST'$</td>
<td>$T' \rightarrow \epsilon$</td>
<td></td>
</tr>
</tbody>
</table>

- Is the grammar LL(1)? Yes, because there are no conflicts in the parsing table.

- How will the input string “(a, a, a)” be processed on the stack?

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$(a, a, a)$</td>
<td>Output $S \rightarrow (T)$</td>
</tr>
<tr>
<td>$(T)$</td>
<td>$(a, a, a)$</td>
<td>Match $( $</td>
</tr>
<tr>
<td>$T)$</td>
<td>$a, a, a$</td>
<td>Output $T \rightarrow ST'$</td>
</tr>
<tr>
<td>$ST')$</td>
<td>$a, a, a$</td>
<td>Output $S \rightarrow a$</td>
</tr>
<tr>
<td>$aT')$</td>
<td>$a, a, a$</td>
<td>Match $a$</td>
</tr>
<tr>
<td>$T')$</td>
<td>$a, a, a$</td>
<td>Output $T' \rightarrow ST'$</td>
</tr>
<tr>
<td>$,ST')$</td>
<td>$a, a$</td>
<td>Match ,</td>
</tr>
<tr>
<td>$ST')$</td>
<td>$a, a$</td>
<td>Output $S \rightarrow a$</td>
</tr>
<tr>
<td>$aT')$</td>
<td>$a$</td>
<td>Match $a$</td>
</tr>
<tr>
<td>$T')$</td>
<td>$\epsilon$</td>
<td>Output $T' \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$)$</td>
<td>$)$</td>
<td>Match )</td>
</tr>
<tr>
<td>$$</td>
<td>$$</td>
<td>Accept</td>
</tr>
</tbody>
</table>

- How is the string “(aa)” processed?

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$(aa)$</td>
<td>Output $S \rightarrow (T)$</td>
</tr>
<tr>
<td>$(T)$</td>
<td>$(aa)$</td>
<td>Match $( $</td>
</tr>
<tr>
<td>$T)$</td>
<td>$(aa)$</td>
<td>Output $T \rightarrow ST'$</td>
</tr>
<tr>
<td>$ST')$</td>
<td>$(aa)$</td>
<td>Output $S \rightarrow a$</td>
</tr>
<tr>
<td>$aT')$</td>
<td>$(aa)$</td>
<td>Match $a$</td>
</tr>
<tr>
<td>$T')$</td>
<td>$a$</td>
<td>Error</td>
</tr>
</tbody>
</table>
SLR(1) Parser

- See recitation #13 for a complete construction of an SLR parser
- As a reminder, the **augmented grammar** $G' = (\{S', S, E, B\}, \{; , s , if , exp\}, S', P)$ looked as follows:
  1. $S' \rightarrow S$
  2. $S \rightarrow E; | if E s | B s | if B$
  3. $E \rightarrow exp$
  4. $B \rightarrow exp$

LR(1) Parser

- Recall that in SLR method, state $i$ calls for reduction by $A \rightarrow \alpha$ if the set of items $I_i$ contains item $A \rightarrow \alpha \cdot a$ and $a$ is in FOLLOW($A$).
- It’s possible to carry more information in the state to try to rule out some of those
- Each state of LR parser can indicate exactly which input symbols can follow a handle $\alpha$ for which there is a possible reduction to $A$
  
  > Items are redefined as including a terminal symbol (or $\$$) as a second component $A \rightarrow \alpha \cdot \beta$, $a \rightarrow LR(1)$ items

- 1 refers to the length of the second component = lookahead
- Lookahead only has effect for items of the form $[A \rightarrow \alpha \cdot a]$, which calls for a reduction by $A \rightarrow \alpha$
- only if the next input symbol is $a$
- $a \subseteq FOLLOW(A)$ and could even be $a \subseteq FOLLOW(A)$; $a \equiv FOLLOW(A)$ for SLR(1) parser

- CLOSURE and GOTO functions operate similarly to SLR(1); main differences in red:

```
SetOfItems CLOSURE(I) {
    repeat
        for ( each item $[A \rightarrow \alpha \cdot \beta [a] \text{ in } I]$
            for ( each production $B \rightarrow \gamma \text{ in } G'$
                for ( each terminal $b \text{ in FIRST(} \beta a)\text{) }
                    add $[B \rightarrow \gamma [b]] \text{ to set } I$;
            )
        )
        until no more items are added to $I$;
    return $I$;
}
```
SetOfItems \text{GOTO}(I, X) \{ \\
\quad \text{initialize } J \text{ to be the empty set; } \\
\quad \text{for ( each item } A \rightarrow \alpha.X\beta[a] \text{ in } I \) \\
\quad \quad \text{add item } A \rightarrow \alpha.X\beta[a] \text{ to set } J; \\
\quad \text{return } \text{CLOSURE}(J); \\
\}

- **Construct Canonical LR(1) Collection**

\begin{verbatim}
void items(G') \{ \\
\quad \text{initialize } C \text{ to } \text{CLOSURE}([S' \rightarrow \cdot S, \$]); \\
\quad \text{repeat} \\
\quad \quad \text{for ( each set of items } I \text{ in } C \) \\
\quad \quad \quad \text{for ( each grammar symbol } X \) \\
\quad \quad \quad \quad \text{if ( } \text{GOTO}(I, X) \text{ is not empty and not in } C \) \\
\quad \quad \quad \quad \quad \text{add } \text{GOTO}(I, X) \text{ to } C; \\
\quad \quad \text{until no new sets of items are added to } C; \\
\}
\end{verbatim}

\begin{itemize}
\item $C = \text{CLOSURE}(\{[S' \rightarrow \cdot S, \$]\})$
\begin{itemize}
\item For $[S' \rightarrow \cdot S, \$]$, $a = \$ $ and $\beta = e \Rightarrow \text{FIRST}(\beta a) = \text{FIRST}(\$) = \{\$\}$, thus $[S \rightarrow \cdot E; , \$], [S \rightarrow \cdot \text{if } E s , \$], [S \rightarrow \cdot \text{if } B s , \$] \text{ are added to } \text{CLOSURE}$
\item For $[S \rightarrow \cdot E; , \$]$, $a = \$ $ and $\beta = ; \Rightarrow \text{FIRST}(;\$) = \{;\}$, thus $[E \rightarrow \cdot \text{exp};, ]$ is added to $\text{CLOSURE}$
\item For $[S \rightarrow \cdot \text{if } E s , \$]$, $a = \$ $ and $\beta = s \Rightarrow \text{FIRST}(s\$) = \{s\}$, thus $[B \rightarrow \cdot \text{exp}, s]$ don’t add any more items, because there is a terminal to the right of the dot
\end{itemize}
\end{itemize}

$\Rightarrow \text{Add the following state to } C: \ I_0 = \{[S' \rightarrow \cdot S, \$], [S \rightarrow \cdot E; , \$], [S \rightarrow \cdot \text{if } E s , \$], [S \rightarrow \cdot \text{if } B s , \$], [E \rightarrow \cdot \text{exp};, ], [B \rightarrow \cdot \text{exp}, s]\}$

- **Round 1:** For set $I_0$
\begin{itemize}
\item $\text{GOTO}(I_0, S') = \emptyset$, because $S'$ is not to the right of any dot
\item $\text{GOTO}(I_0, S) = \text{CLOSURE}([S' \rightarrow \cdot S, \$]) = \{[S' \rightarrow \cdot S, \$]\}$
$\Rightarrow \text{Add } I_1 = \{[S' \rightarrow \cdot S, \$]\}$
\item $\text{GOTO}(I_0, E) = \text{CLOSURE}([S \rightarrow \cdot E; , \$]) = \{[S \rightarrow \cdot E; , \$]\}$
$\Rightarrow \text{Add } I_2 = \{[S \rightarrow \cdot E; , \$]\}$
\item $\text{GOTO}(I_0, B) = \text{CLOSURE}([S \rightarrow \cdot B s , \$]) = \{[S \rightarrow \cdot B s , \$]\}$
$\Rightarrow \text{Add } I_3 = \{[S \rightarrow \cdot B s , \$]\}$
\end{itemize}
• GOTO($I_0, ;$) = $\emptyset$, because ; is not to the right of any dot
• GOTO($I_0, s$) = $\emptyset$, because "s" is not to the right of any dot
• GOTO($I_0, \text{if }$) = CLOSURE($\{ [S \rightarrow \text{if } E \ s, S], [S \rightarrow \text{if } *B; , S] \}$)
  $\rightarrow$ Add $I_4 = \{ [S \rightarrow \text{if } *E \ s, S], [S \rightarrow \text{if } *B; , S], [E \rightarrow \exp, s], [B \rightarrow \exp, :] \}$
• GOTO($I_0, \exp)$ = CLOSURE($\{ [E \rightarrow \exp;:], [B \rightarrow \exp\s, s] \}$)
  $\rightarrow$ Add $I_5 = \{ [E \rightarrow \exp\s, s], [B \rightarrow \exp\s, s] \}$

  ▪ Round 2: For set $I_1 = \{ [S' \rightarrow S\s, S]\}$
    • GOTO($I_1, S$) = GOTO($I_1, S'$) = GOTO($I_1, B$) = GOTO($I_1, \text{E}$) = GOTO($I_1, ;$) = GOTO($I_1, s$) = GOTO($I_1, \text{if }$) = GOTO($I_1, \exp$) = $\emptyset$

  ▪ Round 2: For set $I_2 = \{ [S \rightarrow E\s, S]\}$
    • GOTO($I_2, S$) = GOTO($I_2, S'$) = GOTO($I_2, B$) = GOTO($I_2, \text{E}$) = GOTO($I_2, s$) = GOTO($I_2, \text{if }$) = GOTO($I_2, \exp$) = $\emptyset$
    • GOTO($I_2, ;$) = CLOSURE($\{ [S \rightarrow E;\s, S]\}$)
      $\rightarrow$ Add $I_6 = \{ [S \rightarrow E;\s, S]\}$

  ▪ Round 2: For set $I_3 = \{ [S \rightarrow B \ s, S]\}$
    • GOTO($I_3, S$) = GOTO($I_3, S'$) = GOTO($I_3, B$) = GOTO($I_3, \text{E}$) = GOTO($I_3, ;$) = GOTO($I_3, s$) = GOTO($I_3, \text{if }$) = GOTO($I_3, \exp$) = $\emptyset$
    • GOTO($I_3, s$) = CLOSURE($\{ [S \rightarrow B \ s, S]\}$)
      $\rightarrow$ Add $I_7 = \{ [S \rightarrow B \ s, S]\}$

  ▪ Round 2: For set $I_4 = \{ [S \rightarrow \text{if } E \ s, S], [S \rightarrow \text{if } *B; , S], [E \rightarrow \exp, s], [B \rightarrow \exp, :] \}$
    • GOTO($I_4, S$) = GOTO($I_4, S'$) = GOTO($I_4, ;$) = GOTO($I_4, s$) = GOTO($I_4, \text{if }$) = GOTO($I_4, \exp$) = $\emptyset$
    • GOTO($I_4, B$) = CLOSURE($\{ [S \rightarrow \text{if } B\s, S]\}$) = $\{ [S \rightarrow \text{if } B\s, S]\}$
      $\rightarrow$ Add $I_8 = \{ [S \rightarrow \text{if } B\s, S]\}$
    • GOTO($I_4, \text{E}\s$) = CLOSURE($\{ [S \rightarrow \text{if } E \ s, S]\}$)
      $\rightarrow$ Add $I_9 = \{ [S \rightarrow \text{if } E \ s, S]\}$
    • GOTO($I_4, \exp$) = CLOSURE($\{ [E \rightarrow \exp\s, s], [B \rightarrow \exp\s, :]\}$)
      $\rightarrow$ Add $I_{10} = \{ [E \rightarrow \exp\s, s], [B \rightarrow \exp\s, :]\}$
      $\rightarrow$ This state is main difference to SLR Parser; Note that $I_5 \neq I_{10}$

  ▪ Round 2: For set $I_5 = \{ [E \rightarrow \exp\s, :], [B \rightarrow \exp\s, s]\}$
    • GOTO($I_5, S$) = GOTO($I_5, S'$) = GOTO($I_5, B$) = GOTO($I_5, \text{E}$) = GOTO($I_5, ;$) = GOTO($I_5, s$) = GOTO($I_5, \text{if }$) = GOTO($I_5, \exp$) = $\emptyset$
Round 3: For set $I_6 = \{ \{S \rightarrow E ::, S\} \}$

- $\text{GOTO}(I_6, S) = \text{GOTO}(I_6, S) = \text{GOTO}(I_6, B) = \text{GOTO}(I_6, E) = \text{GOTO}(I_6, : ) = \text{GOTO}(I_6, s) = \text{GOTO}(I_6, \text{if } ) = \text{GOTO}(I_6, \text{exp }) = \emptyset$

Round 3: For set $I_7 = \{ \{S \rightarrow B ::, S\} \}$

- $\text{GOTO}(I_7, S) = \text{GOTO}(I_7, S) = \text{GOTO}(I_7, B) = \text{GOTO}(I_7, E) = \text{GOTO}(I_7, : ) = \text{GOTO}(I_7, s) = \text{GOTO}(I_7, \text{if } ) = \text{GOTO}(I_7, \text{exp }) = \emptyset$

Round 3: For set $I_8 = \{ \{S \rightarrow \text{if } B ::, S\} \}$

- $\text{GOTO}(I_8, S) = \text{GOTO}(I_8, S) = \text{GOTO}(I_8, B) = \text{GOTO}(I_8, E) = \text{GOTO}(I_8, s) = \text{GOTO}(I_8, \text{if } ) = \text{GOTO}(I_8, \text{exp }) = \emptyset$

- $\text{GOTO}(I_8, :) = \text{CLOSURE}(\{\{S \rightarrow \text{if } B ::, S\}\})$

- $\rightarrow \text{Add } I_{11} = \{\{S \rightarrow \text{if } B ::, S\}\}$

Round 3: For set $I_9 = \{ \{S \rightarrow E ::, S\} \}$

- $\text{GOTO}(I_9, S) = \text{GOTO}(I_9, S) = \text{GOTO}(I_9, B) = \text{GOTO}(I_9, E) = \text{GOTO}(I_9, : ) = \text{GOTO}(I_9, s) = \text{GOTO}(I_9, \text{if } ) = \text{GOTO}(I_9, \text{exp }) = \emptyset$

- $\text{GOTO}(I_9, :) = \text{CLOSURE}(\{\{S \rightarrow E ::, S\}\})$

- $\rightarrow \text{Add } I_{12} = \{\{S \rightarrow E ::, S\}\}$

Round 3: For set $I_{10} = \{ \{E \rightarrow \text{exp } S, [B \rightarrow \text{exp } : ]\} \}$

- $\text{GOTO}(I_{10}, S) = \text{GOTO}(I_{10}, S) = \text{GOTO}(I_{10}, B) = \text{GOTO}(I_{10}, E) = \text{GOTO}(I_{10}, : ) = \text{GOTO}(I_{10}, s) = \text{GOTO}(I_{10}, \text{if } ) = \text{GOTO}(I_{10}, \text{exp }) = \emptyset$

Round 4: For set $I_{11} = \{ \{S \rightarrow B ::, S\} \}$

- $\text{GOTO}(I_{11}, S) = \text{GOTO}(I_{11}, S) = \text{GOTO}(I_{11}, B) = \text{GOTO}(I_{11}, E) = \text{GOTO}(I_{11}, : ) = \text{GOTO}(I_{11}, s) = \text{GOTO}(I_{11}, \text{if } ) = \text{GOTO}(I_{11}, \text{exp }) = \emptyset$

Round 4: For set $I_{12} = \{ \{S \rightarrow E ::, S\} \}$

- $\text{GOTO}(I_{12}, S) = \text{GOTO}(I_{12}, S) = \text{GOTO}(I_{12}, B) = \text{GOTO}(I_{12}, E) = \text{GOTO}(I_{12}, : ) = \text{GOTO}(I_{12}, s) = \text{GOTO}(I_{12}, \text{if } ) = \text{GOTO}(I_{12}, \text{exp }) = \emptyset$
The Canonical LR(1) Collection $C$ has 13 states:

$$C = \{ I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9, I_{10}, I_{11}, I_{12} \} =$$

$$= \{ \{ S' \rightarrow S, \$$, $[S \rightarrow E; $, $[S \rightarrow \text{if } E \ s, \$$, $[S \rightarrow \text{if } B \ s, \$$, $[S \rightarrow \text{if } B; , \$$, $[E \rightarrow \text{exp}; , \$$, $[B \rightarrow \text{exp}, s] \},$$

$$\{ [S \rightarrow S*, $],$$

$$\{ [S \rightarrow E*, $],$$

$$\{ [S \rightarrow B * s, $],$$

$$\{ [S \rightarrow \text{if } E s, $],$$

$$\{ [S \rightarrow \text{if } B; ; , $],$$

$$\{ [E \rightarrow \text{exp}; , $],$$

$$\{ [B \rightarrow \text{exp}; , $],$$

$$\{ [E \rightarrow \text{exp}; , $],$$

$$\{ [B \rightarrow \text{exp}; , $],$$

$$\{ [E \rightarrow \text{exp}; , $],$$

$$\{ [B \rightarrow \text{exp}; , $],$$

$$\{ [E \rightarrow \text{exp}; , $],$$

$$\{ [B \rightarrow \text{exp}; , $] \} \}$$

Resulting GOTO graph
• Canonical LR(1) Parsing Table
  
  - The construction is very similar to construction of parsing table for SLR(1) parser
  - Note that I don’t need to separately compute the FOLLOW sets, because they are implicitly encoded in the LR(1) items
  - Main differences are highlighted in red

  **Algorithm 4.56:** Construction of canonical-LR parsing tables.

  - **INPUT:** An augmented grammar $G'$.
  - **OUTPUT:** The canonical-LR parsing table functions `ACTION` and `GOTO` for $G'$.
  - **METHOD:**
    1. Construct $G' = \{I_0, I_1, \cdots, I_n\}$, the collection of sets of LR(1) items for $G'$.
    2. State $i$ of the parser is constructed from $I_i$. The parsing action for state $i$ is determined as follows.
      - (a) If $[A \rightarrow \alpha \cdot]$ is in $I_i$ and $\text{GOTO}(i, a) = I_j$, then set $\text{ACTION}(i, a) = \text{to "shift j." Here } a \text{ must be a terminal.}$
      - (b) If $[A \rightarrow \alpha .]$ is in $I_i$, $A \neq S'$, then set $\text{ACTION}(i, a) = \text{to "reduce } A \rightarrow \alpha ."$
      - (c) If $[S' \rightarrow S .]$ is in $I_i$, then set $\text{ACTION}(i, \$)$ to "accept."
    3. The goto transitions for state $i$ are constructed for all nonterminals $A$ using the rule: If $\text{GOTO}(i, A) = I_j$, then $\text{GOTO}(i, A) = j$.
    4. All entries not defined by rules (2) and (3) are marked as "error."
    5. The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow S .]$.

<table>
<thead>
<tr>
<th>STATE</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>;</td>
<td>s</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>Shift 4</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Accept</td>
</tr>
<tr>
<td>2</td>
<td>Shift 6</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Shift 7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Shift 10</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Reduce (6)</td>
<td>Reduce (7)</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Reduce (2)</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Reduce (4)</td>
</tr>
<tr>
<td>8</td>
<td>Shift 11</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Shift 12</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Reduce (7)</td>
<td>Reduce (6)</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>Reduce (5)</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>Reduce (3)</td>
</tr>
</tbody>
</table>

- The grammar is definitely LR(1), because there are no multiply defined entries, i.e. there are no shift/reduce or reduce/reduce conflicts.
Parse the input string “if exp ;” with the given LR(1) parser

- Given any LR – parsing table (no matter if from SLR(1), LR(1), or LALR(1) parser, you can use the following algorithm to process a given input string:

```
METHOD: Initially, the parser has s0 on its stack, where s0 is the initial state, and w$ in the input buffer. The parser then executes the program in Fig. 4.36.

let a be the first symbol of w$;
while(1) { /* repeat forever */
    let s be the state on top of the stack;
    if ( ACTION[s,a] = shift t ) {
        push t onto the stack;
        let a be the next input symbol;
    } else if ( ACTION[s,a] = reduce A → β ) {
        pop |β| symbols off the stack;
        let state t now be on top of the stack;
        push goto[t,A] onto the stack;
        output the production A → β;
    } else if ( ACTION[s,a] = accept ) break; /* parsing is done */
    else call error-recovery routine;
}
```

- Note that in addition to the state numbers, we also push the actual input terminals onto the stack
- For our example:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>if exp ;$</td>
<td>Shift 4</td>
<td></td>
</tr>
<tr>
<td>$0 4</td>
<td>exp ;$</td>
<td>Shift 10</td>
<td></td>
</tr>
<tr>
<td>$0 4 10</td>
<td>;$</td>
<td>Reduce B → exp</td>
<td>B → exp</td>
</tr>
<tr>
<td>$0 4 8</td>
<td>;$</td>
<td>Shift 11</td>
<td></td>
</tr>
<tr>
<td>$0 4 8 11</td>
<td>$</td>
<td>Reduce S → if B;</td>
<td>S → if B;</td>
</tr>
<tr>
<td>$0 1</td>
<td>$</td>
<td>Accept</td>
<td></td>
</tr>
</tbody>
</table>

Parse the input string “if exp s;”

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>if exp s;$</td>
<td>Shift 4</td>
<td></td>
</tr>
<tr>
<td>$0 4</td>
<td>exp ;$</td>
<td>Shift 10</td>
<td></td>
</tr>
<tr>
<td>$0 4 10</td>
<td>s;$</td>
<td>Reduce E → exp</td>
<td>E → exp</td>
</tr>
<tr>
<td>$0 4 9</td>
<td>s;$</td>
<td>Shift 12</td>
<td></td>
</tr>
<tr>
<td>$0 4 8 12</td>
<td>;$</td>
<td>Error</td>
<td></td>
</tr>
</tbody>
</table>
**LALR(1) Parser**

- “Lookahead-LR”
- Often used in practice, because tables are considerably smaller than the canonical LR tables
- SLR and LALR tables always have same number of states
- Idea is to merge states from LR(1) parser with common first component
- This merging could introduce a reduce/reduce conflict, but never a shift/reduce conflict

- Construction of LALR parsing table

```
Algorithm 4.59: An easy, but space-consuming LALR table construction.

INPUT: An augmented grammar $G'$.

OUTPUT: The LALR parsing-table functions ACTION and GOTO for $G'$.

METHOD:

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(1) items.

2. For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union.

3. Let $C' = \{J_0, J_1, \ldots, J_m\}$ be the resulting sets of LR(1) items. The parsing actions for state $i$ are constructed from $J_i$ in the same manner as in Algorithm 4.56. If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(1).

4. The GOTO table is constructed as follows. If $J$ is the union of one or more sets of LR(1) items, that is, $J = I_1 \cap I_2 \cap \cdots \cap I_k$, then the cores of GOTO($I_1, X$), GOTO($I_2, X$), $\ldots$, GOTO($I_k, X$) are the same, since $I_1, I_2, \ldots, I_k$ all have the same core. Let $K$ be the union of all sets of items having the same core as GOTO($I_1, X$). Then GOTO($J, X$) = $K$.
```

- Since we already have $C$, we can skip step (1)

- In step (2), we notice that $I_5$ and $I_{10}$ have common cores. Their union is the new state $J_5 = I_5 \cup I_{10} = \{[E \rightarrow \text{exp}, ;], [B \rightarrow \text{exp}, s], [E \rightarrow \text{exp}, s], [B \rightarrow \text{exp}, ;]\}$

- All other states don’t have any cores in common, hence $J_k = I_k$ for $k \in \{0, 4, 6, \ldots, 9\}$ and $J_{10} = I_{11}$ and $J_{11} = I_{12}$

- The resulting parsing table is constructed in step (3)
<table>
<thead>
<tr>
<th>STATE</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>s</td>
<td>exp</td>
</tr>
<tr>
<td>0</td>
<td>Shift 4</td>
<td>Shift 5</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Shift 6</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Shift 7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Shift 5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Reduce (6)</td>
<td>Reduce (6)</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Shift 10</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Shift 11</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Reduce (3)</td>
<td></td>
</tr>
</tbody>
</table>

- The grammar is NOT LALR(1), because through the merger of states we are re-introducing the two reduce/reduce conflicts that look familiar from the SLR construction.

- There are more efficient ways of constructing an LALR parsing table by using kernel items → see your text book on pages 342 – 347
Hierarchy of languages and grammars
- Gathered from Dr. Hughes’ class notes
- Describes power of languages commonly encountered in top-down and bottom-up parsing

Languages
- $LL(k-1) \subset LL(k)$ for $k \geq 1$ → Looking ahead farther increases the number of languages a top-down parser recognizes.
- $LL(k) \subset LR(1)$ for $k \geq 0$ → All languages recognized by top-down parsers with arbitrary lookaheads are a proper subset of $LR(1)$ languages.
- $LR(1) = LR(k)$ for $k \geq 1$ → Looking ahead farther does not increase the power of bottom-up LR-style parsers

Grammars
- Hierarchy of LR grammars: $LR(0) \subset SLR(1) \subset LALR(1) \subset LR(1) \subset LR(k)$
- Especially not the last part: $LR(k-1) \subset LR(k)$ for $k \geq 1$, which is different from the languages
- Relationship between $LL$ and $LR$-style grammars:
  - $LL(0) \subset LR(0)$ and $LL(1) \subset LR(1)$
  - $LL(k) \subset LR(k)$
Data Flow Analysis

- Reaching Definitions (Forward Dataflow Analysis):
  - Definitions \( d \) that may reach a program point along some execution path, such that \( d \) is not “killed” along that path
  - We kill a definition of a variable \( x \) if there is any other definition of \( x \) anywhere along the path
- Code is organized into execution blocks \( \rightarrow \) data flow in terms of what is entering and leaving blocks
- Consider definition: \( d : u = v + w \)
  - Generates a definition \( d \) of variable \( u \), while leaving the remaining incoming definitions unaffected
  - Transfer function of definition \( d \) is
    \[
    f_d(x) = \text{gen}(d) \cup (x - \text{kill}(d))
    \]
    Where \( \text{gen}(d) = \{d\} \), the set of definitions generated by the statement, and \( \text{kill}(d) \) is set of all other definitions of \( u \) in the program
- Suppose block \( B \) has \( n \) statements, with transfer functions \( f_i(d) = \text{gen}(i) \cup (x - \text{kill}(i)) \) for \( i = 1, 2, ..., n \). Then transfer function for block \( B \) may be written as:
  \[
  f_B(x) = \text{gen}(B) \cup (x - \text{kill}(B))
  \]
  - Here \( \text{kill}(B) = \text{kill}(1) \cup \text{kill}(2) \cup \ldots \cup \text{kill}(n) \) is the union of all the definitions killed by the individual statements
  - \( \text{gen}(B) = \text{gen}(n) \cup (\text{gen}(n-1) - \text{kill}(n)) \cup \ldots \cup (\text{gen}(1) - \text{kill}(2) - \text{kill}(3) - \ldots - \text{kill}(n)) \) contains all the definitions inside the block that are “visible” immediately after the block (downwards exposed)

- Example Block \( B_1 \):
  \[
  d_1 : a = 3 \\
  d_2 : a = 4
  \]
  - \( \text{kill}(d_1) = \{d_2\} \), \( \text{kill}(d_2) = \{d_1\} \)
  - \( \text{gen}(d_1) = \{d_1\} \), \( \text{gen}(d_2) = \{d_2\} \)
  - \( \text{kill}(B_1) = \text{kill}(d_1) \cup \text{kill}(d_2) = \{d_1, d_2\} \)
  - \( \text{gen}(B_1) = \text{gen}(d_2) \cup (\text{gen}(d_1) - \text{kill}(d_2)) = \{d_2\} \)
  - \( d_1 \) is not downwards exposed

- Control Flow Equations
  - Set of constraints derived from control flow between basic blocks
  - If there is a control-flow edge from \( P \) to \( B \) then \( \text{OUT}(P) \subseteq \text{IN}(B) \)
o Since a definition cannot reach a point unless there is a path along which it reaches \( IN[B] \) needs to be no larger than the union of the reaching definitions of all the predecessor blocks:

\[
IN[B] = \bigcup_{P \text{ is predecessor of } B} OUT[P] \rightarrow \text{union is the meet operator}
\]

o \( OUT[B] = gen(B) \cup (IN[B] - kill(B)) \)

o Algorithm for computing \( IN[B] = REACH \_ IN[B] \) and \( OUT[B] = REACH \_ OUT[B] \)
  - Computes the least fixed-point of the equations, i.e. the solution whose assigned values to the \( IN \)'s and \( OUT \)'s is contained in the corresponding values for any other solution to the equations

**Algorithm 9.11: Reaching definitions.**

**INPUT:** A flow graph for which \( kill_B \) and \( gen_B \) have been computed for each block \( B \).

**OUTPUT:** \( IN[B] \) and \( OUT[B] \), the set of definitions reaching the entry and exit of each block \( B \) of the flow graph.

**METHOD:** We use an iterative approach, in which we start with the “estimate” \( OUT[B] = \emptyset \) for all \( B \) and converge to the desired values of \( IN \) and \( OUT \). As we must iterate until the \( IN \)'s (and hence the \( OUT \)'s) converge, we could use a boolean variable \( change \) to record, on each pass through the blocks, whether any \( OUT \) has changed. However, in this and in similar algorithms described later, we assume that the exact mechanism for keeping track of changes is understood, and we elide those details.

The algorithm is sketched in Fig. 9.14. The first two lines initialize certain data-flow values. Line (3) starts the loop in which we iterate until convergence, and the inner loop of lines (4) through (6) applies the data-flow equations to every block other than the entry.

1) \( OUT[\text{ENTRY}] = \emptyset \);
2) for (each basic block \( B \) other than \( \text{ENTRY} \)) \( OUT[B] = \emptyset \);
3) while (changes to any \( OUT \) occur)
4) for (each basic block \( B \) other than \( \text{ENTRY} \)) {
5) \( IN[B] = \bigcup_{P \text{ a predecessor of } B} OUT[P] \);
6) \( OUT[B] = gen_B \cup (IN[B] - kill_B) \);
}

Figure 9.14: Iterative algorithm to compute reaching definitions.
• From Sample Exam:

\[
\begin{align*}
\text{B1} & \quad = d_1 : i := m-1 \quad = \\
& \quad = d_2 : j := n \quad = \\
& \quad = d_3 : a := t \quad = \\
\end{align*}
\]

\[
\begin{align*}
\text{B2} & \quad = d_4 : i := i+1 \quad = \\
& \quad = d_5 : j := j-1 \quad = \\
\end{align*}
\]

\[
\begin{align*}
\text{B3} & \quad = d_6 : a := u \quad = \\
\text{B4} & \quad = d_7 : i := v \quad = \\
\end{align*}
\]

- What are the values of KILL and GEN for each of the above four blocks?

\[
\begin{align*}
\text{kill}(B_1) & = \text{kill}(d_1) \cup \text{kill}(d_2) \cup \text{kill}(d_3) \\
& = \{d_4, d_5\} \cup \{d_6\} \\
& = \{d_4, d_5, d_6, d_7\}
\end{align*}
\]

\[
\begin{align*}
\text{gen}(B_1) & = \text{gen}(d_3) \cup (\text{gen}(d_3) - \text{kill}(d_3)) \cup (\text{gen}(d_3) - \text{kill}(d_2) - \text{kill}(d_1)) \\
& = \{d_3\} \cup (\{d_4\} - \{d_6\}) \cup (\{d_4\} - \{d_5\} - \{d_6\}) \\
& = \{d_1, d_2, d_3\}
\end{align*}
\]

\[
\begin{align*}
\text{kill}(B_2) & = \text{kill}(d_4) \cup \text{kill}(d_5) \\
& = \{d_1, d_7\} \cup \{d_2\} = \{d_1, d_2, d_7\}
\end{align*}
\]

\[
\begin{align*}
\text{gen}(B_2) & = \text{gen}(d_3) \cup (\text{gen}(d_4) - \text{kill}(d_3)) \\
& = \{d_5\} \cup (\{d_4\} - \{d_5\}) = \{d_4, d_5\}
\end{align*}
\]

\[
\begin{align*}
\text{kill}(B_3) & = \text{kill}(d_6) = \{d_3\} \\
\text{gen}(B_3) & = \text{gen}(d_6) = \{d_6\}
\end{align*}
\]

\[
\begin{align*}
\text{kill}(B_4) & = \text{kill}(d_7) = \{d_1, d_4\} \\
\text{gen}(B_4) & = \text{gen}(d_7) = \{d_7\}
\end{align*}
\]
What are the recurrence relations that can be used to compute \textit{REACH\_IN} and \textit{REACH\_OUT} for each of these blocks?

\begin{align*}
\text{\textit{OUT}[B_i]} &= \text{\textit{gen}(B_i)} \cup (\text{\textit{IN}[B_i]} - \text{\textit{kill}(B_i)}) = \{d_1, d_2, d_3\} \cup (\text{\textit{IN}[B_i]} - \{d_4, d_5, d_6, d_7\}) \\
\text{\textit{IN}[B_i]} &= \bigcup_{p \text{ is predecessor of } B_i} \text{\textit{OUT}[P]} = \emptyset
\end{align*}

\begin{align*}
\text{\textit{OUT}[B_2]} &= \text{\textit{gen}(B_2)} \cup (\text{\textit{IN}[B_2]} - \text{\textit{kill}(B_2)}) = \{d_4, d_5\} \cup (\text{\textit{IN}[B_2]} - \{d_1, d_2, d_7\}) \\
\text{\textit{IN}[B_2]} &= \bigcup_{p \text{ is predecessor of } B_2} \text{\textit{OUT}[P]} = \text{\textit{OUT}[B_1]} \cup \text{\textit{OUT}[B_3]} \cup \text{\textit{OUT}[B_4]} = \emptyset
\end{align*}

\begin{align*}
\text{\textit{OUT}[B_3]} &= \text{\textit{gen}(B_3)} \cup (\text{\textit{IN}[B_3]} - \text{\textit{kill}(B_3)}) = \{d_6\} \cup (\text{\textit{IN}[B_3]} - \{d_3\}) \\
\text{\textit{IN}[B_3]} &= \bigcup_{p \text{ is predecessor of } B_3} \text{\textit{OUT}[P]} = \text{\textit{OUT}[B_2]}
\end{align*}

\begin{align*}
\text{\textit{OUT}[B_4]} &= \text{\textit{gen}(B_4)} \cup (\text{\textit{IN}[B_4]} - \text{\textit{kill}(B_4)}) = \{d_7\} \cup (\text{\textit{IN}[B_4]} - \{d_1, d_4\}) \\
\text{\textit{IN}[B_4]} &= \bigcup_{p \text{ is predecessor of } B_4} \text{\textit{OUT}[P]} = \text{\textit{OUT}[B_2]}
\end{align*}

What are the values of \textit{REACH\_IN} and \textit{REACH\_OUT}?

- **Iteration 0:**
  \begin{align*}
  \text{\textit{OUT}[B_1]} &= \text{\textit{OUT}[B_2]} = \text{\textit{OUT}[B_3]} = \text{\textit{OUT}[B_4]} = \emptyset
  
\end{align*}

- **Iteration 1:**
  \begin{align*}
  \text{\textit{IN}[B_1]} &= \emptyset \\
  \text{\textit{OUT}[B_1]} &= \text{\textit{gen}(B_1)} \cup (\text{\textit{IN}[B_1]} - \text{\textit{kill}(B_1)}) = \{d_1, d_2, d_3\} \\
  \text{\textit{IN}[B_2]} &= \text{\textit{OUT}[B_1]} \cup \text{\textit{OUT}[B_1]} \cup \text{\textit{OUT}[B_4]} = \emptyset \\
  \text{\textit{OUT}[B_2]} &= \text{\textit{gen}(B_2)} \cup (\text{\textit{IN}[B_2]} - \text{\textit{kill}(B_2)}) = \{d_4, d_5\} \\
  \text{\textit{IN}[B_3]} &= \text{\textit{OUT}[B_2]} = \emptyset \\
  \text{\textit{OUT}[B_3]} &= \text{\textit{gen}(B_3)} \cup (\text{\textit{IN}[B_3]} - \text{\textit{kill}(B_3)}) = \{d_6\} \\
  \text{\textit{IN}[B_4]} &= \text{\textit{OUT}[B_2]} = \emptyset \\
  \text{\textit{OUT}[B_4]} &= \text{\textit{gen}(B_4)} \cup (\text{\textit{IN}[B_4]} - \text{\textit{kill}(B_4)}) = \{d_7\}
  
\end{align*}

- **Iteration 2:**
  \begin{align*}
  \text{\textit{IN}[B_1]} &= \emptyset \\
  \text{\textit{OUT}[B_1]} &= \text{\textit{gen}(B_1)} \cup (\text{\textit{IN}[B_1]} - \text{\textit{kill}(B_1)}) = \{d_1, d_2, d_3\} \\
  \rightarrow & \text{ Done with } B_1 \text{ because no change in } \text{\textit{OUT}[B_1]} \\
  \text{\textit{IN}[B_2]} &= \text{\textit{OUT}[B_1]} \cup \text{\textit{OUT}[B_1]} \cup \text{\textit{OUT}[B_4]} = \{d_1, d_2, d_3\} \cup \{d_6\} \cup \{d_7\} = \{d_1, d_2, d_3, d_6, d_7\} \\
  \text{\textit{OUT}[B_2]} &= \text{\textit{gen}(B_2)} \cup (\text{\textit{IN}[B_2]} - \text{\textit{kill}(B_2)}) = \{d_4, d_5\} \cup \{d_3, d_6\} = \{d_3, d_4, d_5, d_6\} \\
  \text{\textit{IN}[B_3]} &= \text{\textit{OUT}[B_2]} = \{d_3, d_4, d_5, d_6\} \\
  \text{\textit{OUT}[B_3]} &= \text{\textit{gen}(B_3)} \cup (\text{\textit{IN}[B_3]} - \text{\textit{kill}(B_3)}) = \{d_6\} \cup \{d_4, d_5, d_6\} = \{d_4, d_5, d_6\}
  
\end{align*}
$IN[B_4] = OUT[B_2] = \{d_3, d_4, d_5, d_6\}$

$OUT[B_2] = gen(B_4) \cup (IN[B_4] - kill(B_4)) = \{d_7\} \cup \{d_3, d_5, d_6\} = \{d_3, d_5, d_6, d_7\}$

- **Iteration 3:**
  
  $IN[B_2] = OUT[B_1] \cup OUT[B_3] \cup OUT[B_4]$
  
  $\quad = \{d_1, d_2, d_3\} \cup \{d_4, d_5, d_6\} \cup \{d_3, d_5, d_6, d_7\} = \{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$

  $OUT[B_2] = gen(B_2) \cup (IN[B_2] - kill(B_2))$
  
  $\quad = \{d_4, d_5\} \cup \{d_4, d_5, d_6\} = \{d_4, d_5, d_6\}$

  $\rightarrow$ Done with $B_2$ because no change in $OUT[B_2]$

  $IN[B_3] = OUT[B_2] = \{d_3, d_4, d_5, d_6\}$

  $OUT[B_3] = gen(B_3) \cup (IN[B_3] - kill(B_3)) = \{d_6\} \cup \{d_4, d_5, d_6\} = \{d_4, d_5, d_6\}$

  $\rightarrow$ Done with $B_3$ because no change in $OUT[B_3]$

  $IN[B_4] = OUT[B_2] = \{d_3, d_4, d_5, d_6\}$

  $OUT[B_4] = gen(B_4) \cup (IN[B_4] - kill(B_4)) = \{d_7\} \cup \{d_3, d_5, d_6\} = \{d_3, d_5, d_6, d_7\}$

  $\rightarrow$ Done with $B_4$ because no change in $OUT[B_4]$
Example:

- All the definitions in block B1 reach the beginning of block B2.
- The definition \(d_5: j = j-1\) in block B2 also reaches the beginning of block B2, because no other definitions of \(j\) can be found in the loop leading back to B2.
- This definition, however, kills the definition \(d_2: j = n\), preventing it from reaching B3 or B4.
- The statement \(d_4: i = i+1\) in B2 does not reach the beginning of B2 though, because the variable \(i\) is always redefined by \(d_7: i = u3\).
- Finally, the definition \(d_6: a = u2\) also reaches the beginning of block B2.