04/21/2011

# Recitation #13

## Administrative

- Final Project (due **Tuesday, April 26 @ 11:59 p.m.; No Lateness allowed**)
- Final Exam on **Thursday, April 28 @ 10:00 a.m. – 12:50 p.m.** in HEC 118
- Extended Office Hours on **Monday, April 25 @ 2:00 p.m. – 5 p.m.**
- Final Exam help session on **Wednesday, April 27**
  - Didn't decide on the time and room yet, but will send that information by week's end

## Final Project

- Maybe some more comments on the project
- **Return type of procedures** is structure `YYSTYPE` (at least for functions that return anything, i.e. they contain action code with `$$` in Bison)
  - That's the same structure used to transmit information from the lexical analyzer to the parser via the variable `yylval`
  - Returned structures are referenced through `$1, $2`, …
- **Pushing lexeme back**
  - New function `yylexback()` as part of skeleton lexical analyzer
  - One situation where this might be necessary is the variable declaration section. Take this example:

    ```
    var a, b : number;
    ```
  - When your parser looks at the semicolon at the end of this line, there are two different rules it could apply to:
    1. It matches the semicolon in the `variable.decl.list` rule and hence there should be more variable declarations after this semicolon.
    2. It matches the semicolon in the `variable.decl.part` rule and hence this is the end of all variable declarations.
  - To decide that one way or the other when you are in your function handling `variable.decl.list`, you could look past the semicolon and see what's next without actually parsing the next token. If it's an identifier, you should stay in `variable.decl.list`; otherwise push lexeme back and return. The semicolon should be matched by `variable.decl.part`.
  - Wirth's Pascal parser solves that problem in a similar way; after reading the semicolon his code checks if there is another identifier. He doesn't have to push back any lexeme, because the end of the variable declarations is implicit if there is anything other than an identifier after the semicolon → only partially helpful if you implemented one function per non-terminal
- **Bison error recovery**:
  - Dr. Hughes had a custom error recovery function "ok()" in the solution for assignment 4

- In the solution for the final project he uses the built-in Bison error recovery methods that work similarly.
- Take the following example:

```
program.heading:
   PROGRAM                     { basicinit(); }
         IDENT SEMICOLON |
   error BEGINN                { yyerrok; yychar = BEGINN;} |
   error SEMICOLON             { yyerrok; yyclearin; }
   ;
```

- If an error occurs while matching the tokens PROGRAM, IDENT, and SEMICOLON, there are two possible error recovery mechanisms:
  1. Read tokens until you find BEGINN, then tell Bison that everything is fine and keep the look-ahead token at BEGINN:

     ```
     error BEGINN                { yyerrok; yychar = BEGINN;}
     ```

  2. Read tokens until you find SEMICOLON, then tell Bison that everything is fine and discard the current look-ahead token, so the parsing can continue after the semicolon:

     ```
     error SEMICOLON         { yyerrok; yyclearin; }
     ```

- Further explanations of Bison's error recovery here: http://www.gnu.org/software/bison/manual/bison.html#Error-Recovery
- More details on all of the Bison macros (e.g. $$, yyerrok, yyclearin, …) can be found here: http://www.gnu.org/software/bison/manual/bison.html#Action-Features

## Final Exam Review

- Final Exam will be comprehensive.
- Questions from the first half of the semester will be similar to the ones that were asked for the midterm exam → Definitely look at solutions for sample and actual midterm

### *Dr. Hughes' Promises*

1. An expression grammar that incorporates precedence and associativity → **Recitation #4**
2. Distinction between languages and grammars in a particular class. → **See notes below**
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**
8. Removal of left recursion and common prefixes. → **Recitation #4**
9. CKY Parsing Table. → **Recitations #6 and #8 and notes below**
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 see notes below**
12. Completion of the states, actions and gotos for an SLR(1) parser. → **Recitation #9**
13. Completion of canonical LR(1) parser.
14. LALR(1) parser by doing merges on a canonical LR(1) parser's states.
15. Evaluation of attributes (inherited and synthesized) for some attributed translation grammar. → **Recitation #10**
16. Data flow algorithm based on one of the four discussed in class.

### *Ambiguous Languages and Grammars*

- Distinction between ambiguous grammars and ambiguous languages
- A grammar G of a language L is ambiguous if there exists some string $w \in L(G)$ for which there are two or more distinct derivation trees
- Ambiguity is a property of a grammar, and it is usually (but not always) possible to find an equivalent unambiguous grammar.
- An inherently ambiguous language is a language for which no unambiguous grammar exists.
- Ambiguous grammar might just be a bad grammar for a given language → doesn't always imply that language is ambiguous as well

### *Bison Basics*

- My notes on Bison are sprinkled throughout the labs, so please see recitations #6, #7, and #8
- Attributes
  - `$$` refers to the value for the left-hand side symbol (the one to the left of the colon)
  - `$1, $2`, etc. refer to elements of the right-hand side
    - Be careful with embedded actions, e.g. for
      `ELSE {$$=triple;} statement {$$=$2+1}`
    - $1 = ELSE, $2 ={$$=triple;}, $3 = statement, $4 = {$$=$2+1}

- Associativity / Precedence

```
%left          PLUS MINUS
%left          TIMES DIVIDE
%nonassoc      RELOP
```

  - Means that PLUS, and MINUS are left-associative and have lower precedence than left-associative TIMES, and DIVIDE tokens
  - RELOP is not associative at all, i.e. it cannot appear with another RELOP token in the same expression

- See error recovery notes in the Final Project section above
- Bison Manual is a great resource for all further questions: [http://www.gnu.org/software/bison/manual/bison.html](http://www.gnu.org/software/bison/manual/bison.html)

### *CKY Parsing Table*

- See recitations #6 and #8 for more explanation and examples
- Should be easy; everybody did really well on assignment #3
- Remember that grammar has to be in Chomsky Normal Form
- Recognize ambiguity when you see it

- Grammar for prefix expressions with + and -, e.g. + d d, or + - - d d d d:

$$E \to OF$$
$$F \to EE$$
$$O \to +$$
$$O \to -$$
$$E \to \mathrm{d}$$

- The CKY table for string **+ d – d d** looks as follows:

| | + | **d** | **-** | **d** | **d** |
|---|---|---|---|---|---|
| **1** | $O \to +$ | $E \to \mathrm{d}$ | $O \to -$ | $E \to \mathrm{d}$ | $E \to \mathrm{d}$ |
| **2** | | | | $F \to EE$ | |
| **3** | | | $E \to OF$ | | |
| **4** | | $F \to EE$ | | | |
| **5** | $E \to OF$ | | | | |

The string **+d – d d** is part of the language, because it can be derived from $E$. There is no ambiguity in that derivation:

$$E \rightarrow OF \rightarrow +F \rightarrow +EE \rightarrow +dE \rightarrow +dOF \rightarrow +d-F \rightarrow +d-EE \rightarrow +d-dE \rightarrow +d-dd$$

## *SLR(1) Parser*

- See recitation #9 for more explanation and examples
- Grammar $G = (\{S, E, B\}, \{; , \text{s} , \text{if} , \text{exp}\}, S, P)$ from Sample Exam:

  $S \rightarrow E; | \text{if } E \text{ s} | B \text{ s} | \text{if } B;$

  $E \rightarrow \text{exp}$

  $B \rightarrow \text{exp}$

- The **augmented grammar** $G' = (\{S', S, E, B\}, \{; , \text{s} , \text{if} , \text{exp}\}, S', P)$ adds a new starting symbol :

  (1) $S' \rightarrow S$

  (2) – (5) $S \rightarrow E; | \text{if } E \text{ s} | B \text{ s} | \text{if } B;$

  (6) $E \rightarrow \text{exp}$

  (7) $B \rightarrow \text{exp}$

- Construct **Canonical LR(0) Collection**:
  - C = CLOSURE( $\{S' \rightarrow \bullet S\}$ )

    → Add $I_0 = \{S' \rightarrow \bullet S, S \rightarrow \bullet E;, S \rightarrow \bullet\text{if } E \text{ s}, S \rightarrow \bullet B \text{ s}, S \rightarrow \bullet\text{if } B;, E \rightarrow \bullet\text{exp}, B \rightarrow \bullet\text{exp}\}$ to C
  - *Round 1*: For set $I_0$
    - Grammar symbol $S'$:

      GOTO( $I_0$, $S'$ ) = $\varnothing$, because $S'$ is not to the right of any dot
    - Grammar symbol $S$ :

      GOTO( $I_0$, $S$ ) = CLOSURE( $\{S' \rightarrow S\bullet\}$ ) = $\{S' \rightarrow S\bullet\}$

      →Add $I_1 = \{S' \rightarrow S\bullet\}$ to C
    - Grammar symbol $E$ :

      GOTO( $I_0$, $E$ ) = CLOSURE( $\{S \rightarrow E\bullet;\}$ ) = $\{S \rightarrow E\bullet;\}$

      → Add $I_2 = \{S \rightarrow E\bullet;\}$ to C
    - Grammar symbol $B$ :

      GOTO( $I_0$, $B$ ) = CLOSURE( $\{S \rightarrow B\bullet \text{ s}\}$ ) = $\{S \rightarrow B\bullet \text{ s}\}$

→ Add $I_3 = \{S \to B \bullet \text{ s}\}$ to C

- Grammar symbol $;$ :

  GOTO( $I_0$ , $;$ ) = $\varnothing$, because $;$ is not to the right of any dot
- Grammar symbol "s" :

  GOTO( $I_0$ , s ) = $\varnothing$, because "s" is not to the right of any dot
- Grammar symbol "if" :

  GOTO( $I_0$ , if ) = CLOSURE( $\{S \to \text{if} \bullet E \text{ s}, S \to \text{if} \bullet B;\}$ )

  → Add $I_4 = \{S \to \text{if} \bullet E \text{ s},\ S \to \text{if} \bullet B;,\ E \to \bullet\text{exp},\ B \to \bullet\text{exp}\}$ to C
- Grammar symbol "exp" :

  GOTO( $I_0$ , exp ) = CLOSURE( $\{E \to \text{exp}\bullet,\ B \to \text{exp}\bullet\}$ )

  → Add $I_5 = \{E \to \text{exp}\bullet,\ B \to \text{exp}\bullet\}$ to C

- *Round 2*: For set $I_1 = \{S' \to S\bullet\}$
  - GOTO( $I_1$ , $S'$ ) = GOTO( $I_1$ , $S$ ) = GOTO( $I_1$ , $B$ ) = GOTO( $I_1$ , $E$ ) =

    GOTO( $I_1$ , $;$ ) = GOTO( $I_1$ , s ) = GOTO( $I_1$ , if ) = GOTO( $I_1$ , exp ) = $\varnothing$
- *Round 2*: For set $I_2 = \{S \to E\bullet;\}$
  - GOTO( $I_2$ , $S'$ ) = GOTO( $I_2$ , $S$ ) = GOTO( $I_2$ , $B$ ) = GOTO( $I_2$ , $E$ ) =

    GOTO( $I_2$ , s ) = GOTO( $I_2$ , if ) = GOTO( $I_2$ , exp ) = $\varnothing$
  - GOTO( $I_2$ , $;$ ) = CLOSURE( $\{S \to E;\bullet\}$ ) = $\{S \to E;\bullet\}$

    → Add $I_6 = \{S \to E;\bullet\}$ to C
- *Round 2*: For set $I_3 = \{S \to B \bullet \text{ s}\}$
  - GOTO( $I_3$ , $S'$ ) = GOTO( $I_3$ , $S$ ) = GOTO( $I_3$ , $B$ ) = GOTO( $I_3$ , $E$ ) =

    GOTO( $I_3$ ,;) = GOTO( $I_3$ , if ) = GOTO( $I_3$ , exp ) = $\varnothing$
  - GOTO( $I_3$ , s ) = CLOSURE( $\{S \to B\text{s}\bullet\}$ ) = $\{S \to B\text{s}\bullet\}$

    → Add $I_7 = \{S \to B\text{s}\bullet\}$ to C
- *Round 2*: For set $I_4 = \{S \to \text{if} \bullet E \text{ s},\ S \to \text{if} \bullet B;,\ E \to \bullet\text{exp},\ B \to \bullet\text{exp}\}$
  - GOTO( $I_4$ , $S'$ ) = GOTO( $I_4$ , $S$ ) = GOTO( $I_4$ ,;) = GOTO( $I_4$ , if ) = GOTO( $I_4$ , s ) = $\varnothing$
  - GOTO( $I_4$ , $E$ ) = CLOSURE( $\{S \to \text{if } E\bullet\text{s}\}$ ) = $\{S \to \text{if } E\bullet\text{s}\}$

    → Add $I_8 = \{S \to \text{if } E\bullet\text{s}\}$ to C
  - GOTO( $I_4$ , $B$ ) = CLOSURE( $\{S \to \text{if } B\bullet;\}$ ) = $\{S \to \text{if } B\bullet;\}$

    → Add $I_9 = \{S \to \text{if } B\bullet;\}$ to C
  - GOTO( $I_4$ , exp ) = CLOSURE( $\{E \to \text{exp}\bullet,\ B \to \text{exp}\bullet\}$ ) = $I_5$
- *Round 2*: For set $I_5 = \{E \to \text{exp}\bullet,\ B \to \text{exp}\bullet\}$

- GOTO( $I_1$, $S'$ ) = GOTO( $I_1, S$ ) = GOTO( $I_1$, $B$ ) = GOTO( $I_1, E$ ) = GOTO( $I_1$, ; ) = GOTO( $I_1, \mathrm{s}$ ) = GOTO( $I_1$, $\mathrm{if}$ ) = GOTO( $I_1, \mathrm{exp}$ ) = $\varnothing$

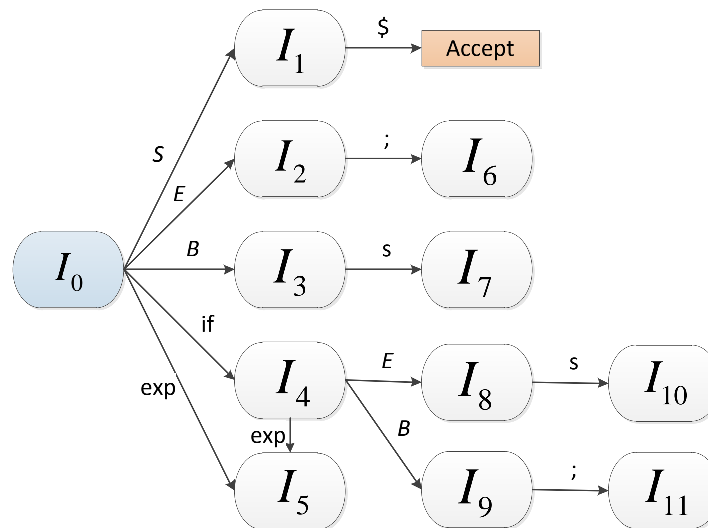
- *Round 3*: For set $I_6 = \{S \rightarrow E;\bullet\}$
  - GOTO( $I_6$, $S'$ ) = GOTO( $I_6, S$ ) = GOTO( $I_6$, $B$ ) = GOTO( $I_6, E$ ) = GOTO( $I_6$, ; ) = GOTO( $I_6, \mathrm{s}$ ) = GOTO( $I_6$, $\mathrm{if}$ ) = GOTO( $I_6, \mathrm{exp}$ ) = $\varnothing$
- *Round 3*: For set $I_7 = \{S \rightarrow B\mathrm{s}\bullet\}$
  - GOTO( $I_7$, $S'$ ) = GOTO( $I_7, S$ ) = GOTO( $I_7$, $B$ ) = GOTO( $I_7, E$ ) = GOTO( $I_7$, ; ) = GOTO( $I_7, \mathrm{s}$ ) = GOTO( $I_7$, $\mathrm{if}$ ) = GOTO( $I_7, \mathrm{exp}$ ) = $\varnothing$
- *Round 3*: For set $I_8 = \{S \rightarrow \mathrm{if}\ E\bullet\mathrm{s}\}$
  - GOTO( $I_8$, $S'$ ) = GOTO( $I_8, S$ ) = GOTO( $I_8$, $B$ ) = GOTO( $I_8, E$ ) = GOTO( $I_8$, ; ) = GOTO( $I_8$, $\mathrm{if}$ ) = GOTO( $I_8, \mathrm{exp}$ ) = $\varnothing$
  - GOTO( $I_8$, $\mathrm{s}$ ) = CLOSURE( $\{S \rightarrow \mathrm{if}\ E\mathrm{s}\bullet\}$ ) = $\{S \rightarrow \mathrm{if}\ E\mathrm{s}\bullet\}$

    → Add $I_{10} = \{S \rightarrow \mathrm{if}\ E\mathrm{s}\bullet\}$ to C
- *Round 3*: For set $I_9 = \{S \rightarrow \mathrm{if}\ B\bullet;\}$
  - GOTO( $I_9$, $S'$ ) = GOTO( $I_9, S$ ) = GOTO( $I_9$, $B$ ) = GOTO( $I_9, E$ ) = GOTO( $I_9$, $\mathrm{s}$ ) = GOTO( $I_9$, $\mathrm{if}$ ) = GOTO( $I_9, \mathrm{exp}$ ) = $\varnothing$
  - GOTO( $I_9$, ; ) = CLOSURE( $\{S \rightarrow \mathrm{if}\ B;\bullet\}$ ) = $\{S \rightarrow \mathrm{if}\ B;\bullet\}$

    → Add $I_{11} = \{S \rightarrow \mathrm{if}\ B;\bullet\}$ to C
- *Round 4*: For set $I_{10} = \{S \rightarrow \mathrm{if}\ E\mathrm{s}\bullet\}$
  - GOTO( $I_{10}$, $S'$ ) = GOTO( $I_{10}, S$ ) = GOTO( $I_{10}$, $B$ ) = GOTO( $I_{10}, E$ ) = GOTO( $I_{10}$, ; ) = GOTO( $I_{10}, \mathrm{s}$ ) = GOTO( $I_{10}$, $\mathrm{if}$ ) = GOTO( $I_{10}, \mathrm{exp}$ ) = $\varnothing$
- *Round 4*: For set $I_{11} = \{S \rightarrow \mathrm{if}\ B;\bullet\}$
  - GOTO( $I_{11}$, $S'$ ) = GOTO( $I_{11}, S$ ) = GOTO( $I_{11}$, $B$ ) = GOTO( $I_{11}, E$ ) = GOTO( $I_{11}$, ; ) = GOTO( $I_{11}, \mathrm{s}$ ) = GOTO( $I_{11}$, $\mathrm{if}$ ) = GOTO( $I_{11}, \mathrm{exp}$ ) = $\varnothing$

- The Canonical LR(0) Collection has 12 states

- Resulting **LR(0) Automaton**




- Construct the **SLR Parsing Table**:
  - Grammar Reminder:

    (1) $S' \to S$

    (2) – (5) $S \to E; | \text{if } E \text{ s} | B \text{ s} | \text{if } B;$

    (6) $E \to \text{exp}$

    (7) $B \to \text{exp}$

  - Calculate FIRST sets:

    FIRST( $S'$ ) = FIRST( $S$ ) = $\{\text{exp}, \text{if}\}$

    FIRST( $E$ ) = FIRST( $B$ ) = $\{\text{exp}\}$
  - Calculate FOLLOW sets:

    FOLLOW( $S'$ ) = FOLLOW ( $S$ ) = $\{\$\}$

    FOLLOW ( $E$ ) = FOLLOW ( $B$ ) = $\{;, \text{ s}\}$

| STATE | ACTION | | | | | GOTO | | | |
|---|---|---|---|---|---|---|---|---|---|
| | ; | s | if | exp | $ | $S'$ | $S$ | $E$ | $B$ |
| **0** | | | Shift 4 | Shift 5 | | | 1 | 2 | 3 |
| **1** | | | | | Accept | | | | |
| **2** | Shift 6 | | | | | | | | |
| **3** | | Shift 7 | | | | | | | |
| **4** | | | | Shift 5 | | | | 8 | 9 |
| **5** | Reduce (6)<br>Reduce (7) | Reduce (6)<br>Reduce (7) | | | | | | | |
| **6** | | | | | Reduce (2) | | | | |
| **7** | | | | | Reduce (4) | | | | |
| **8** | | Shift 10 | | | | | | | |
| **9** | Shift 11 | | | | | | | | |
| **10** | | | | | Reduce (3) | | | | |
| **11** | | | | | Reduce (5) | | | | |

This grammar turns out not to be SLR, because the SLR parsing table contains two reduce/reduce conflicts.