Review: Parsing Algorithm

- Let $\alpha$ be the current stack and $t$ the next input token.
  - If we've reduced the entire input to the start symbol: success.
  - Otherwise, run the DFA on $\alpha$ (the stack)
    - If it rejects, we've detected an error.
    - If the resulting state contains an item $[\varepsilon \rightarrow \alpha, t]$ then reduce, and repeat.
    - If the resulting state contains an item containing $t$ then shift, and repeat.
    - Otherwise, an error has been detected
Parsing Ambiguous Grammars

- Many languages have grammar rules such as

\[
S \rightarrow \text{if } E \text{ then } S \text{ else } S
\]
\[
S \rightarrow \text{if } E \text{ then } S
\]
\[
S \rightarrow \text{other}
\]

- How should we interpret:

\[
\text{if } a \text{ then if } b \text{ then } s_1 \text{ else } s_2
\]
Dangling Else Problem

• Since this grammar is ambiguous, it cannot be LR(k) for any k.
• Thus we know there will be a state with a conflict.
  - In this case, a shift-reduce conflict.

\[
[S \to \text{if } E \text{ then } S. , \text{else}]
[S \to \text{if } E \text{ then } S. \text{ else } S, \text{any}]
\ldots
\]

Solution 1: Fix the grammar

• We could rewrite the grammar to remove the ambiguity. For example:

\[
S \to M \\
S \to U \\
M \to \text{if } E \text{ then } M \text{ else } M \\
M \to \text{other} \\
U \to \text{if } E \text{ then } S \\
U \to \text{if } E \text{ then } M \text{ else } U
\]

(prohibits else-less if in a then branch)
Solution 2: Change Language

• Add an end marker to conditionals

S \rightarrow \text{if } E \text{ then } S \text{ else } S \text{ fi}
S \rightarrow \text{if } E \text{ then } S \text{ fi}
S \rightarrow \text{other}

• Then forced to say

if a then if b then s1 fi else s2 fi
fi
if a then if b then s1 else s2 fi fi

Solution 3: Hack the Parser

• Original problem was that in this state the parser didn't know whether to shift or reduce.
• So just tell it to always reduce in this case
  - No more ambiguity!

• Implements rule that else must match the most recent possible then.
  - This rule holds in most languages.
  - Yacc will default to shifting when there is a shift/reduce conflict, so you just have to live with the warning.
Parsing Ambiguous Grammars

• How about arithmetic expressions? The following grammar is highly ambiguous:

```
E → Int | E + E | E - E | E * E | (E)
```

• For example, consider \(2 \times 3 + 4\) and \(5 - 6 - 7\)

Arithmetic Precedence

• This grammar will generate several shift-reduce conflicts, including

```
[E → E * E . , +]
[E → E . + E , any]

[E → E + E . , *]
[E → E . * E , any]

[E → E - E . , -]
[E → E . - E , any]
```

Solution 1: Fix the grammar

- We can make the grammar unambiguous by adding new nonterminals.

```
E → E + T | E - T | T
T → T * F | F
F → (E) | Int
```

Solution 2: Change Language

- Change the language to something like

```
E → Int | (E + E) | (E - E) | (E * E) | (E)
```

or

```
E → Int | E E + | E E - | E E *
```
Solution 3: Hack the Parser

• Or we could just resolve the conflicts:

\[
\begin{align*}
E &\to E \cdot E . . + \\
E &\to E . + E , any
\end{align*}
\]
always reduce

\[
\begin{align*}
E &\to E + E . . * \\
E &\to E . * E , any
\end{align*}
\]
always shift

\[
\begin{align*}
E &\to E - E . . - \\
E &\to E . - E , any
\end{align*}
\]
always reduce

Parsing Ambiguous Grammars

• The grammar for the EQN typesetting language includes the following productions:

\[
\begin{align*}
E &\to E sub E sup E \\
E &\to E sub E \\
E &\to E sup E \\
E &\to \{ E \} \\
E &\to other
\end{align*}
\]

• Aside from associativity and precedence issues, how do we handle

\[
E sub E sup E
\]
Special-Case Productions

- We will get shift/reduce conflicts due to associativity and
- Also, a reduce/reduce conflict

\[
\begin{align*}
E &\rightarrow E \text{ sub } E \text{ sup } E, \\
E &\rightarrow E \text{ sup } E, \\
\end{align*}
\]

Solution: Hack Parser

- Always take the first reduction.
  - Simple enough to understand consequences
  - Would be much harder to modify grammar

- Note: ML-Yacc will default to the first production rule in a reduce/reduce conflict.
Parsing Ambiguous Grammars

• Suppose we try to distinguish arithmetic and boolean expressions in the grammar.

\[
\begin{align*}
E & \rightarrow A \mid B \\
A & \rightarrow id \mid A + A \mid A - A \\
B & \rightarrow id \mid B \& B \mid A = A
\end{align*}
\]

• What is the reduce/reduce conflict?

Solution: Give Up

• Difficult to get the parser to do this checking
• Just do a typechecking pass later.

\[
E \rightarrow id \mid E + E \mid E - E \mid E \& E \mid E = E
\]
Conflicts in Practice

• The dangling-else problem and arithmetic precedence problem are well-understood.
• Be very careful with any other conflict.
  – Reduce/reduce conflicts usually signal a problem with the grammar (though not always)
  – Examine any shift/reduce conflicts

Constructing Parsers

• The presentation you have seen is unrealistic in the following ways:
  – DFA's are generated directly, rather than by constructing the NFA and using the subset construction. (See Appel for details)
  – LALR parsers are generated directly, rather than by constructing the LR(1) automaton and compressing it.
Implementing Parsers

• First optimization:
  - After each step, prefix of the stack is unchanged
  - Wasteful to run this through the DFA
  - Solution: label each stack element with the state state one ends up in

<table>
<thead>
<tr>
<th>1 E4 +5 E7 *8 E9</th>
<th>$</th>
<th>Reduce E→E*E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 E4 +5 E7</td>
<td>$</td>
<td>Reduce E→E+E</td>
</tr>
<tr>
<td>...etc...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Implementing Parsers

• For each automaton state and lookahead token we can precompute the action to take:
  - Shift token onto stack, tagged with state n
  - Reduce using Rule n
• Two tables (possibly merged)
  - Parsing table: indexed by top stack state and lookahead symbol
  - Goto table: indexed by stack states and nonterminals
### Parsing Table

- Contains 4 types of entries
  - \( s(n) \): Shift next token, tag with state \( n \)
  - \( r(n) \): Reduce using \( n^{th} \) rewrite rule \( A \rightarrow \beta \). (Pop \( \beta \) off the stack, look up \( A \) in the goto table to get a state, push \( A \) and this state on stack.
  - \( a \): Accept the input and stop parsing
  - \( e \): Error detected.

### Goto Table

- Indexed by states and nonterminals
  - If I'm in state \( n \) and I see \( A \), what is the next state?
- Just contains those edges of the DFA labeled by nonterminals.

- Frequently combined with the parsing table (as in Appel) into one big table.
  - Indexed by state and by stack symbol
ML-Yacc

- An LALR(1) parser generator for SML.
  - Modeled on Yacc, Bison, etc.
- File format:

Simple ML-Yacc Input File

(* User declarations *)
%
$name Sample
%term ID | WHILE | BEGIN | END | DO
  | IF | THEN | ELSE | SEMI | ASSIGN | EOF
%nonterm prog | stm | stmlist
%start prog
%pos int
%eop EOF
%verbose

%...continued on next slide...
Simple ML-Yacc Input File

```
prog : stmlist ()

stm : ID ASSIGN ID ()
    | WHILE ID DO stm ()
    | BEGIN stmlist END ()
    | IF ID THEN stm ()
    | IF ID THEN stm ELSE stm ()

stmlist : stm ()
    | stmlist SEMI stm ()
```

Another Example: Precedence

```
%%
%term INT | PLUS | MINUS | TIMES | UMINUS | EQ | EOF
%nonterm exp
%start exp
...
%nonassoc EQ
%left PLUS MINUS
%right TIMES
%left UMINUS
%
exp : INT ()
    | exp PLUS exp ()
    | exp MINUS exp ()
    | exp TIMES exp ()
    | exp EQ exp ()
    | MINUS exp %prec UMINUS ()
```
Semantic Actions

- We can tell the parser to perform an action every time it does a reduction.
- By default, this must be a piece of code of type `unit`.
- Effectively performed in postorder traversal of parse tree

```
exp : INT (print "saw int
")
| exp PLUS exp     (print "saw +\n")
| exp MINUS exp    (print "saw -\n")
| exp TIMES exp    (print "saw *\n")
| exp EQ exp       (print "saw =\n")
| MINUS exp %prec UMINUS (print "saw unary -\n")
```

Terminals with Values

- Any real lexer won't just return the tokens `ID` or `INT` every time a variable or numeral is found; it also returns a string/integer/etc.

```
%term INT of int | ID of string | ...etc...
%%
exp : INT        (print (Int.toString(INT)))
| ID             (print ID)
| exp MINUS exp  (print "saw -\n")
```

- Value can be referred to as ID or INT within the semantic action.  (Or ID1 and ID2 if ID occurs twice on the left, etc.)
Nonterminals with Values

- Nonterminals can also have values
  - All actions for this nonterminal must have the same type, but this type need not be unit.

```
%%
%term INT of int | PLUS | MINUS | TIMES | UMINUS | EOF
%nonterm exp of int
%start exp
%%
exp : INT (INT)
    | exp PLUS exp (exp1 + exp2)
    | exp MINUS exp (exp1 - exp2)
    | exp TIMES exp (exp1 * exp2)
    | MINUS exp %prec UMINUS (~ exp)
```