CSE443
Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall
Phases of a compiler

Figure 1.7, page 5 of text
Figure 4.35 [p. 248]
### LR parsing table

**ACTION function**

- Inputs: state $i$ and an input symbol $a$ (terminal or $\$$)
- $\text{ACTION}[i,a]$ is:
  - Shift $j$ - shift $a$ onto stack, using state $j$ to represent $a$
  - Reduce $A \rightarrow \beta$
  - Accept
  - Error

**GOTO function** - extend from sets of items to states.

- $\text{GOTO}[I_i,A] = I_j \Rightarrow \text{GOTO}[i,A] = j$
Algorithm 4.44 [p. 250-251]
The LR-parsing algorithm

INPUT: An input string $w$ and an LR-parsing table with functions ACTION and GOTO for a grammar $G$

OUTPUT: If $w$ is in $L(G)$, the reduction steps of a bottom-up parse for $w$; otherwise, an error indication

METHOD: Initially, the parser has $s_0$ on its stack, where $s_0$ is the initial state. The parser then executes the program in Fig. 4.36.
let a be the first symbol of $w$

while (true) {
    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 -> $\beta$) {
        pop |$\beta$| 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 -> $\beta$
    } else if (ACTION[s,a] = accept) break
    else call error-recovery routine
}

REVIEW
Algorithm 4.46 [p. 253]
Constructing an SLR-parsing table

**INPUT:** An augmented grammar $G'$

**OUTPUT:** The SLR-parsing table functions $\text{ACTION}$ and $\text{GOTO}$ for $G'$

**METHOD:**

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(0) items for $G'$
2. State $i$ is constructed from $I_i$. The parsing items for state $i$ are determined as follows:
   
   A. If $[A \rightarrow a \alpha \beta]$ is in $I_i$ and $\text{GOTO}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to "shift $j". Here $a$ must be a terminal.
   
   B. If $[A \rightarrow \alpha \beta]$ is in $I_i$, then set $\text{ACTION}[i, a]$ to "reduce $A \rightarrow \alpha$" for all $a$ in $\text{FOLLOW}(A)$; here $A$ may not be $S'$.
   
   C. If $[S' \rightarrow S \alpha]$ is in $I_i$, then set $\text{ACTION}[i, \$$]$ to "accept."

If conflicting actions result from the above rules, we say the grammar is not SLR(1). The algorithm fails to produce a parser in this case.

3. The goto transitions for state $I$ are constructed for all nonterminals $A$ using the rule: If $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$.

4. All entries not defined by rules (2) and (3) are made "error".

5. The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow S]$
\textbf{FIRST}(X)

- if $X \in T$ then $\text{FIRST}(X) = \{ X \}$

- if $X \in N$ and $X \rightarrow Y_1 Y_2 \ldots Y_k \in P$ for $k \geq 1$, then

  - add $a \in T$ to $\text{FIRST}(X)$ if $\exists i$ s.t. $a \in \text{FIRST}(Y_i)$ and $\varepsilon \in \text{FIRST}(Y_j) \ \forall \ j < i$ (i.e. $Y_1 Y_2 \ldots Y_k \Rightarrow^* \varepsilon$)

- if $\varepsilon \in \text{FIRST}(Y_j) \ \forall \ j < k$ add $\varepsilon$ to $\text{FIRST}(X)$
FOLLOW(X)

Place $ in FOLLOW(S), where S is the start symbol ($ is an end marker)

if A $\rightarrow\alpha B\beta \in P$, then FIRST(\beta) - \{ε\} is in FOLLOW(B)

if A $\rightarrow\alpha B \in P$ or A $\rightarrow\alpha B\beta \in P$ where $\epsilon \in$ FIRST(\beta), then everything in FOLLOW(A) is in FOLLOW(B)
**FIRST(X) and FOLLOW(X)**

<table>
<thead>
<tr>
<th>X</th>
<th>FIRST(X)</th>
<th>FOLLOW(X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S'</td>
<td>id, (</td>
<td>$</td>
</tr>
<tr>
<td>E</td>
<td>id, (</td>
<td>+, ), $</td>
</tr>
<tr>
<td>T</td>
<td>id, (</td>
<td>*, +, ), $</td>
</tr>
<tr>
<td>F</td>
<td>id, (</td>
<td>*, +, ), $</td>
</tr>
<tr>
<td>id</td>
<td>id</td>
<td>*, +, ), $</td>
</tr>
<tr>
<td>(</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>)</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>
2A. If \[A \rightarrow \alpha \cdot a \beta\] is in \(I_i\) and \(\text{GOTO}(I_i,a) = I_j\), then set \(\text{ACTION}[i,a]\) to "shift \(j\)". Here \(a\) must be a terminal.

2B. If \([A \rightarrow \alpha \cdot ]\) is in \(I_i\), then set \(\text{ACTION}[i,a]\) to "reduce \(A \rightarrow a\)" for all \(a\) in \(\text{FOLLOW}(A)\); here \(A\) may not be \(S'\).

2C. If \([S' \rightarrow S\delta]\) 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 \(\text{GOTO}(I_i,A) = I_j\), then \(\text{GOTO}[i,A] = j\).

Production numbers:
1. \(E \rightarrow E + T\)
2. \(E \rightarrow T\)
3. \(T \rightarrow T \ast F\)
4. \(T \rightarrow F\)
5. \(F \rightarrow (E)\)
6. \(F \rightarrow \text{id}\)

See Piazza post @53
### Figure 4.37 [p. 252]
### Parsing table for expression grammar

<table>
<thead>
<tr>
<th>STATE</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>s6</td>
<td>accept</td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td></td>
<td>E-&gt;T</td>
<td>E-&gt;T</td>
</tr>
<tr>
<td>3</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td></td>
<td>T-&gt;F</td>
<td>T-&gt;F</td>
</tr>
<tr>
<td>4</td>
<td>s6</td>
<td>s4</td>
</tr>
<tr>
<td>5</td>
<td>r6</td>
<td>r6</td>
</tr>
<tr>
<td></td>
<td>F-&gt;id</td>
<td>F-&gt;id</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Lecture goof today**

Hi everyone,

I realized, right at the end of class, that I goofed in describing the number used for reductions in the table. When we first encountered the reduction I said it was the state number, which in that example was 2. '2' was correct, but it's not the state number - it's the number of the reducing production in our grammar.

**Production numbers:**

1. E -> E + T
2. E -> T + E
3. T -> T * F
4. T -> F
5. F -> ( E )
6. F -> id

When we reduce by E -> T we enter 'r2' in the table, regardless of which state we're in.

At a later point we reduced by F->id in state 5. The correct entry for those cells is 'r6'.

Sorry for the confusion.
## Figure 4.37 [p. 252]

**Parsing table for expression grammar**

<table>
<thead>
<tr>
<th>STATE</th>
<th>ACTION</th>
<th>GOTO0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>id</td>
<td>s5</td>
</tr>
<tr>
<td>1</td>
<td>s6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>s7</td>
</tr>
<tr>
<td>3</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r6</td>
<td>r6</td>
</tr>
<tr>
<td>6</td>
<td>s6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>s6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s6</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r1</td>
<td>s7</td>
</tr>
<tr>
<td>10</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>11</td>
<td>r5</td>
<td>r5</td>
</tr>
</tbody>
</table>

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T \ast F$
4. $T \rightarrow F$
5. $F \rightarrow ( E )$
6. $F \rightarrow id$
LR parser configuration
(comes up today)

- An LR parser configuration is a pair:
  \((s_0 \ldots s_m, a_1a_2 \ldots a_n\$)\)
- \(s_0 \ldots s_m\) is the stack (bottom to top)
- \(a_1a_2 \ldots a_n\$\) is the (remaining) input

- Represents the right-sentential form
  \(X_1X_2 \ldots X_m a_1a_2 \ldots a_n\)
Textbook Typo:

- On page 254, line-4:
  - "Fig. 4.31" should be "Fig. 4.37".
Project 2 notes

- Make sure you work through remainder of chapter 4, especially 4.9.
- Also consult sections you read for HW1, such as 2.7.
- Symbol table structure and functionality will need to be adapted to meet changing needs as project continues. Write your code with growth/change in mind.
Example 4.51 [p. 260]

Grammar from example 4.48:

\[ S \rightarrow L = R | R \]
\[ L \rightarrow *R | id \]
\[ R \rightarrow L \]

\[ I_0: \quad S' \rightarrow S \]
\[ S \rightarrow \cdot L = R \]
\[ S \rightarrow \cdot R \]
\[ L \rightarrow \cdot * R \]
\[ L \rightarrow \cdot id \]
\[ R \rightarrow \cdot L \]

\[ I_1: \quad S' \rightarrow S. \]

\[ I_2: \quad S \rightarrow L = R \]
\[ R \rightarrow L. \]

\[ I_3: \quad S \rightarrow R. \]

\[ I_4: \quad L \rightarrow \cdot R \]
\[ R \rightarrow \cdot L \]
\[ L \rightarrow \cdot * R \]
\[ L \rightarrow \cdot id \]

\[ I_5: \quad L \rightarrow id. \]

\[ I_6: \quad S \rightarrow L = \cdot R \]
\[ R \rightarrow \cdot L \]
\[ L \rightarrow \cdot * R \]
\[ L \rightarrow \cdot id \]

\[ I_7: \quad L \rightarrow \cdot * R. \]

\[ I_8: \quad R \rightarrow L. \]

\[ I_9: \quad S \rightarrow L = R; \]

Figure 4.39: Canonical LR(0) collection for grammar (4.49)
Example 4.51 [p. 260]

Grammar from example 4.48:

\[
S \rightarrow L = R \mid R \\
L \rightarrow *R \mid id \\
R \rightarrow L
\]

"[This grammar] is not ambiguous. This shift/reduce conflict arises [because] SLR parser construction method [does not] remember enough left context..."
Viable prefix

"Why can LR(0) automata be used to make shift-reduce decisions? The LR(0) automaton for a grammar characterizes the strings of grammar symbols that can appear on the stack... The stack contents must be a prefix of a right-sentential form. If the stack holds α and the rest of the input is x, then a sequence of reductions will take αx to S. In terms of derivations, $S \Rightarrow_{r^m} αx$." [p. 256]
"Not all prefixes of right-sentential forms can appear on the stack...since the parser must not shift past the handle." [p. 256]

\[ E \Rightarrow_{rm*} F \ast id \Rightarrow_{rm} (E) \ast id \]
Viable prefix

"Not all prefixes of right-sentential forms can appear on the stack...since the parser must not shift past the handle." [p. 256]

\[ E \Rightarrow_{\text{rm}*} F \ast \text{id} \Rightarrow_{\text{rm}} ( E ) \ast \text{id} \]

(\( E \)) is a handle of \( F \Rightarrow ( E ) \)
Viable prefix
(parser configurations shown)

\[
(\$, (\text{`, id `}) \ast \text{id} \$)
(\$, ( \text{`id `, `} \ast \text{id} \$
(\$, (\text{`id `, `} \ast \text{id} \$
(\$, (\text{`F `, `} \ast \text{id} \$
(\$, (\text{`T `, `} \ast \text{id} \$
(\$, (\text{`E `, `} \ast \text{id} \$
(\$, (\text{`E `}) \ast \text{id} \$
(\$, \text{`F `} \ast \text{id} \$
(\$, \text{`T `} \ast \text{id} \$
(\$, \text{`T `} \ast \text{id} $
\]
e tc.

Cannot shift `*` here, because 
(``E``)`
is a handle.
Viable prefix

"The prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes." [p. 256]
Viable prefix

Cannot shift '*' here, because


is a handle.

Therefore


is not a viable prefix.
LR(1) items

"...in the SLR method, state I calls for reduction by $A \to \alpha$ if the set of items $I_i$ contains item $[A \to \alpha \epsilon]$ and input symbol a is in FOLLOW(A)." [p. 260]
LR(1) items

"In some situations, however, when state I appears on top of the stack the viable prefix $\beta \alpha$ on the stack is such that $\beta A$ cannot be followed by a in any right-sentential form." [p. 260]
Example 4.51 [p. 260]

Grammar from example 4.48:
\[ S \rightarrow L = R \mid R \]
\[ L \rightarrow *R \mid id \]
\[ R \rightarrow L \]

State I2 from figure 4.39
\[ S \rightarrow L \ast = R \]
\[ R \rightarrow L \ast \]

"Consider the set of items I2. The first item in this set makes \text{ACTION}[2,=] be 'shift 6'. Since \text{FOLLOW}(R) contains = [\ldots] the second items sets \text{ACTION}[2,=] to reduce \( R \rightarrow L \)." [p. 255]

"...the SLR parser calls for reduction by \( R \rightarrow L \) in state 2 with = as the next input (the shift action is also called for ...). However, there is no right-sentential form of the grammar ... that begins \( R = \ldots \) . Thus state 2, which is the state corresponding to viable prefix \( L \) only, should not really call for reduction of that \( L \) to \( R \)." [p. 260]
"By splitting states when necessary, we can arrange to have each state ... indicate exactly which input symbols can follow a handle $\alpha$ for which there is a possible reduction to $A.$" [p. 260]

"The general form of an item becomes

$$[ A \rightarrow \alpha \cdot \beta, a]$$

where $A \rightarrow \alpha \beta$ is a production and $a$ is a terminal or ... $\$.$" [p. 260]
LR(1) items

"The lookahead has no effect in an item of the form \([ A \rightarrow \alpha \cdot \beta, a]\), where \(\beta\) is not \(\varepsilon\), but an item of the form \([ A \rightarrow \alpha \cdot, a]\) calls for reduction by \(A \rightarrow \alpha\) only if the next input symbol is \(a\). [...] The set of such \(a\)'s will always be a subset of FOLLOW(A), but it could be a proper subset ..." [p. 260]
LALR (lookahead LR)

"SLR and LALR tables ... always have the same number of states." [p. 266]

Idea: merge sets of LR(1) items with the same core.

Cannot introduce Shift/Reduce conflicts, may introduce Reduce/Reduce conflicts.

Bison and YACC produce LALR parsers.
Phases of a compiler

Semantic analysis

Figure 1.6, page 5 of text
Semantics

• “Semantics” has to do with the meaning of a program.

• We will consider two types of semantics:

  – Static semantics: semantics which can be enforced at compile-time.

  – Dynamic semantics: semantics which express the run-time meaning of programs.
Static semantics

• Semantic checking which can be done at compile-time

• Type-compatibility is a prime example
  – int can be assigned to double (type coercion)
  – double cannot be assigned to int without explicit type cast

• Type-compatibility can be captured in grammar, but only at expense of larger, more complex grammar
Ex: adding type rules in grammar

• Must introduce new non-terminals which encode types:
• Instead of a generic grammar rule for assignment:
  - \( <\text{stmt}> \rightarrow <\text{var}> \ '==' \ <\text{expr}> \ ';' \)
• we need multiple rules:
  - \( <\text{stmt}> \rightarrow <\text{doubleVar}> \ '==' \ <\text{intExpr}> \ | \ <\text{doubleExpr}> \ ';' \)
  - \( <\text{stmt}> \rightarrow <\text{intVar}> \ '==' \ <\text{intExpr}> \ ';' \)
• Of course, such rules need to handle all the relevant type possibilities (e.g. \text{byte}, \text{char}, \text{short}, \text{int}, \text{long}, \text{float} and \text{double}).
Alternative: attribute grammars

- Attribute grammars provide a neater way of encoding such information.
- Each syntactic rule of the grammar can be decorated with:
  - a set of semantic rules/functions
  - a set of semantic predicates
We can associate with each symbol $X$ of the grammar a set of attributes $A(X)$. Attributes are partitioned into:

- synthesized attributes $S(X)$ – pass info up parse tree
- inherited attributes $I(X)$ – pass info down parse tree
Semantic rules/functions

• We can associate with each rule $R$ of the grammar a set of semantic functions.

• For rule $X_0 \rightarrow X_1 \ X_2 \ \ldots \ \ X_n$
  - synthesized attribute of LHS:
    $S(X_0) = f(A(X_1), A(X_2), \ldots, A(X_n))$
  - inherited attribute of RHS member:
    for $1 \leq j \leq n$, $I(X_j) = f(A(X_0), \ldots, A(X_{j-1}))$
    (note that dependence is on siblings to left only)
Predicates

• We can associate with each rule R of the grammar a set of semantic predicates.

• Boolean expression involving the attributes and a set of attribute values

• If true, node is ok

• If false, node violates a semantic rule
Example

\[
<assign> \rightarrow <var> = <expr>
\]

Start with a production of the grammar
Example

<assign> → <var> = <expr>

<expr>.expType

Associate an attribute with a non-terminal, <expr>, on the right of the production: expType (the expected type of the expression)
Assign to `<expr>.expType` the value of `<var>.actType`, the actual type of the variable (the type the variable was declared as).
Example

<assign> → <var> = <expr>
<expr>.expType ← <var>.actType

In other words, we expect the expression whose value is being assigned to a variable to have the same type as the variable.
Example

<assign> → <var> = <expr>
<expr>.expType ← <var>.actType


Another grammar production
Example

\[ \text{Example} \]

\begin{align*}
\text{<assign>} & \rightarrow \text{<var> } = \text{<expr>} \\
\text{<expr>.expType} & \leftarrow \text{<var>.actType} \\
\text{<expr>} & \rightarrow \text{<var>[2]} + \text{<var>[3]} \\
\text{<expr>.actType} & \leftarrow \text{if (var[2].actType = int) and} \\
& \quad \text{(var[3].actType = int)} \\
& \quad \text{then int} \\
& \quad \text{else real} \\
\end{align*}

This production has a more involved semantic rule: it handles type coercion. This rule assumes that there are only two numeric types (int and real) and that int can be coerced to real.
Here is our first semantic predicate, which enforces a type-checking constraint: the actual type of `<expr>` must match the expected type (from elsewhere in the tree).
Example

<assign>  \rightarrow  <var> = <expr>
<expr>.expType \leftarrow <var>.actType

<expr>.actType \leftarrow \text{if (var[2].actType = int) and (var[3].actType = int)}
then int
else real
<expr>.actType == <expr>.expType

Another production, with a semantic rule and a semantic predicate.
Example

\[\textit{<assign>} \rightarrow \textit{<var>} = \textit{<expr>}\]
\[\textit{<expr>}.\text{expType} \leftarrow \textit{<var>}.\text{actType}\]

\[\textit{<expr>} \rightarrow \textit{<var>}[2] + \textit{<var>}[3]\]
\[\textit{<expr>}.\text{actType} \leftarrow \begin{cases} 
\text{int} & \text{if (var[2].actType = int) and (var[3].actType = int)} \\
\text{real} & \text{else}
\end{cases}\]
\[\textit{<expr>}.\text{actType} == \textit{<expr>}.\text{expType}\]

\[\textit{<expr>} \rightarrow \textit{<var>}\]
\[\textit{<expr>}.\text{actType} \leftarrow \textit{<var>}.\text{actType}\]
\[\textit{<expr>}.\text{actType} == \textit{<expr>}.\text{expType}\]

\[\textit{<var>} \rightarrow \text{A | B | C}\]
\[\textit{<var>}.\text{actType} \leftarrow \text{lookUp(\textit{<var>}.string)}\]

This semantic rule says that the type of an identifier is determined by looking up its type in the symbol table.
All the productions, rules and predicates

<assign>  \rightarrow  <var> = <expr>
<expr>.expType  \leftarrow  <var>.actType

<expr>.actType  \leftarrow  \text{if } (\text{var}[2].actType = \text{int}) \text{ and }
                (\text{var}[3].actType = \text{int})
             \text{then int}
             \text{else real}
<expr>.actType  ==  <expr>.expType

<expr>  \rightarrow  <var>
<expr>.actType  \leftarrow  <var>.actType
<expr>.actType  ==  <expr>.expType

<var>  \rightarrow  A \mid B \mid C
<var>.actType  \leftarrow  \text{lookUp}(<var>.string)
Let's see how these rules work in practice!

In this example A and B are both of type int.

Suppose:
A is int
B is int
Suppose:
A is int
B is int

(actual type = int)

(actual type = int)

(expected type = int)

(actual type = int)

(actual type = int)

Effects of the syntactic rules is shown in red.