Announcements

• Homework set 3 will be posted by Friday

• Homework set 2 solutions have been posted (sakai → Resources).

• The first programming project new deadline: Monday, October 29.

• Midterm exam, October 23, in class, 80 minutes. No lecture after the exam.
Project 1 – Polymorphic Type Reconstruction

The provided Scheme function \textit{parse} maps a really-TINY program into an AST representation. Your type reconstructor should take as input an AST, a type environment, and a set of constraints:

\begin{verbatim}
(define TR
  (lambda (ast E C)
    ...
  )

(define TRec
  (lambda (m)
    ...
    ;; extract type expression from TR call
    (TR (parse m) init_E init_C) ...
  ))
\end{verbatim}

Good luck, and start early!
Review – Formal Languages

Alphabet: finite set of symbols – \( \{a, b, c\} \)

String (or word): finite sequence of symbols from an alphabet – \( w, x, y \in \Sigma^* \);
\( \epsilon \in \Sigma^* \) denotes the empty word

Language: set of strings over an alphabet – \( L \subseteq \Sigma^* \)

Two problems:

- How to specify a formal language \( L \)?
  Specifications have to be finite.

- How to recognize that string \( w \in L \)?
  Procedure should be efficient (terminate?).

<table>
<thead>
<tr>
<th>specification</th>
<th>recognizing automaton</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular expressions</td>
<td>DFA</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>CFG</td>
<td>PDA</td>
<td>( O(n) / O(n^3) )</td>
</tr>
<tr>
<td>CSG</td>
<td>LBA</td>
<td>P-SPACE COMPLETE</td>
</tr>
<tr>
<td>Arbitrary rewrite systems</td>
<td>TM</td>
<td>r.e.</td>
</tr>
</tbody>
</table>
Compilers

Source code $\rightarrow$ front end $\rightarrow$ opt. $\rightarrow$ back end $\rightarrow$ target code

**front end** produce an intermediate representation (IR) for the program.

**optimizer** transforms the code in IR form into an equivalent program that may run more efficiently.

**back end** transforms the code in IR form into native code for the target machine.

The IR encodes knowledge that the compiler has derived about the source program.
The role of the parser

Parser

• call scanner to get new token
• perform context-free syntax analysis
• guide the context-sensitive analysis
  \[\rightarrow\ SDT / \text{attribute grammars}\]
• construct an intermediate representation
  \[\rightarrow\ SDT / \text{attribute grammars}\]
• produce meaningful error messages
• attempt error correction
Context-free syntax is specified with a grammar.

Formally, a context-free grammar $G$ is a four-tuple $(T, NT, S, P)$

$T$ is the set of terminal symbols in the grammar. For our purposes, the set of terminals is equivalent to the set of tokens returned by the lexical analyzer.

$NT$ is a set of syntactic variables that denote sets of (sub)strings occurring in the language. These are used to impose a structure on the grammar.

$S$ is a distinguished nonterminal ($S \in NT$) that denotes the entire set of strings in $L(G)$. This is sometimes called a goal symbol or start symbol. In an ECFG (extended/augmented CFG), $S$ cannot appear on the right hand side of some production.

$P$ is a set of productions that specify the way that terminals and non-terminals can be combined to form strings in the language. Each production must have a single non-terminal on its left hand side.
Why use context-free grammars?

Many advantages:

• precise syntactic specification of a programming language
• easy to understand, avoids ad hoc definition
• easier to maintain, add new language features
• can automatically construct efficient parser
• parser construction reveals ambiguity, other difficulties
• supports syntax-directed translation (SDT)
Complexity of parsing

Complexity:

- Regular grammars: \( \text{dfas} \) \( \mathcal{O}(n) \)
- LR grammars: Knuth’s algorithm \( \mathcal{O}(n) \)
- Arbitrary CFGs: Early’s algorithm \( \mathcal{O}(n^3) \)
- Arbitrary CSGs: \( \text{lbas} \) \( \text{P-SPACE COMPLETE} \)
Review of some definitions

Parse tree (derivation tree):

“Graphical” representation for a derivation $S \Rightarrow^* w$; filters out the choice regarding non-terminal replacement order

Leftmost (rightmost) derivation

The leftmost (rightmost) non-terminal is replaced at each step

Sentential form:

Any $\alpha \in (T \cup NT)^*$ such that $S \Rightarrow^* \alpha$

Left (Right) sentential form:

Any $\alpha \in (T \cup NT)^*$ such that $S \Rightarrow_{lm}^* \alpha$

$(S \Rightarrow_{rm}^* \alpha)$
Ambiguity

If a grammar has multiple leftmost derivations for a single sentential form, the grammar is *ambiguous*. Similarly, a grammar with multiple rightmost derivations for a single sentential form is *ambiguous*.

Example

\[
<\text{stmt}> ::= \text{if} \ <\text{expr}> \ \text{then} \ <\text{stmt}>
\]
\[
\quad \mid \ \text{if} \ <\text{expr}> \ \text{then} \ <\text{stmt}> \ \text{else} \ <\text{stmt}>
\]
\[
\quad \mid . . . /* \text{other stmts} */
\]

Consider deriving the sentential form:

\[
\text{if } E_1 \ \text{then if } E_2 \ \text{then } S_1 \ \text{else } S_2
\]

It has two derivations.

This ambiguity is purely grammatical. However, there are context-free languages for which every context-free grammar is ambiguous.

Example (see Hopcroft&Ullman, p.99):

\[
\{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}
\]
Top-down parsers

- start at the root of derivation tree and fill in
- picks a production and tries to match the input
- some grammars are backtrack-free (predictive)
- **LL Parsing**: reads input from left to right and constructs leftmost derivation (forwards); LL Parsing is predictive.

\[ S \Rightarrow_{lm}^* xA\beta \Rightarrow_{lm} x\delta\beta \Rightarrow_{lm}^* xy \]

- \( x, y \in T^*; S, A \in NT; \delta, \beta \in (T \cup NT)^*; A \rightarrow \delta \in P \)
Predictive Parsing — LL(1)

Basic idea:

For any two productions \( A ::= \alpha \mid \beta \), we would like a distinct way of choosing the correct production to expand.

For some rhs \( \alpha \in G \), define \( \text{FIRST}(\alpha) \) as the set of terminals that appear as the first symbol in some string derived from \( \alpha \).

That is

\[
\begin{align*}
a \in \text{FIRST}(\alpha) & \iff \alpha \Rightarrow^* a\gamma \text{ for some } \gamma \text{ and } a \in T, \\
\epsilon \in \text{FIRST}(\alpha) & \iff \alpha \Rightarrow^* \epsilon.
\end{align*}
\]

For a non-terminal \( A \), define \( \text{FOLLOW}(A) \) as the set of terminals that can appear immediately to the right of \( A \) in some sentential form.

\[
\begin{align*}
a \in \text{FOLLOW}(A) & \iff S \Rightarrow^* \alpha A a \gamma \text{ for some } \\
\alpha, \gamma & \in (T \cup NT)^*, a \in T.
\end{align*}
\]

Thus, a non-terminal’s FOLLOW set specifies the tokens that can legally appear after it. A terminal symbol has no FOLLOW set.
Predictive Parsing — LL(1) (cont.)

Key Property:
Whenever two productions $A ::= \alpha$ and $A ::= \beta$
both appear in the grammar, we would like

- $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$, and

- if $\alpha \Rightarrow^* \epsilon$ then in addition to above condition:
  $\text{FIRST}(\beta) \cap \text{FOLLOW}(A) = \emptyset$

- Analogue case for $\beta \Rightarrow^* \epsilon$. Note: due to first
  condition, at most one of $\alpha$ or $\beta$ can derive $\epsilon$.

This would allow the parser to make a correct choice
with a lookahead of only one symbol!
LL(1) grammars

Features

- input parsed from left to right
- leftmost derivation (forward)
- one token lookahead

Definition

A grammar $G$ is $LL(1)$ if and only if for each set of productions $A ::= \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$

1. FIRST($\alpha_1$), FIRST($\alpha_2$), $\cdots$, FIRST($\alpha_n$) are all pairwise disjoint, and

2. if $\alpha_i \Rightarrow^* \epsilon$, then in addition
   \[ \text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \emptyset, \text{ for all } 1 \leq j \leq n, i \neq j. \]

What rule to select for a given non-terminal and input token can be represented in a parse table $M$.

Algorithm for $LL(1)$ parse table construction must not result in multiple entries for any $M[A, a]$ or $M[A, eof]$ (Aho, Sethi, and Ullman, Algorithm 4.4).

⇒ Whether a grammar is $LL(1)$ or not is decidable.
Table-driven predictive parser — LL(1)

*Input*: a string $w$ and a parsing table $M$ for $G$

```
push eof
push Start Symbol
token ← next_token()
X ← top-of-stack
repeat
    if $X$ is a terminal then
        if $X = \text{token}$ then
            pop $X$
token ← next_token()
        else error()
    else /* $X$ is a non-terminal */
        if $M[X, \text{token}] = X \rightarrow Y_1Y_2\ldots Y_k$ then
            pop $X$
push $Y_k, Y_{k-1}, \ldots, Y_1$
        else error()

    X ← top-of-stack
until $X = \text{eof}$

if token $\neq$ eof then error()
```

Aho, Sethi, and Ullman, Algorithm 4.3
**FIRST set construction**

For a string of grammar symbols $\alpha$, define $\text{FIRST}(\alpha)$ as

- the set of terminal symbols that begin strings derived from $\alpha$
- if $\alpha \Rightarrow^* \epsilon$, then $\epsilon \in \text{FIRST}(\alpha)$

$\text{FIRST}(\alpha)$ contains the set of tokens valid in the first position of $\alpha$

**STEP 1:** Build $\text{FIRST}(X)$ for all grammar symbols $X$:

1. if $X$ is a terminal, $\text{FIRST}(X)$ is \{X\}
2. if $X ::= \epsilon$, then $\epsilon \in \text{FIRST}(X)$
3. iterate until no more terminals or $\epsilon$ can be added to any $\text{FIRST}(X)$:
   
   if $X ::= Y_1Y_2 \cdots Y_k$ then
   
   $a \in \text{FIRST}(X)$ if $a \in \text{FIRST}(Y_i)$
   and $\epsilon \in \text{FIRST}(Y_j)$ for all $1 \leq j < i$
   
   $\epsilon \in \text{FIRST}(X)$ if $\epsilon \in \text{FIRST}(Y_i)$ for all $1 \leq i \leq k$

   end iterate

(If $\epsilon \notin \text{FIRST}(Y_1)$, then $\text{FIRST}(Y_i)$ is irrelevant, for $1 < i$)
The FIRST set

**STEP 2:** Build FIRST($\alpha$) for $\alpha = X_1X_2 \cdots X_n$:

- $a \in$ FIRST($\alpha$) if $a \in$ FIRST($X_i$) and $\epsilon \in$ FIRST($X_j$) for all $1 \leq j < i$
- $\epsilon \in$ FIRST($\alpha$) if $\epsilon \in$ FIRST($X_i$) for all $1 \leq i \leq n$
**FOLLOW set construction**

For a non-terminal $A$, define $\text{FOLLOW}(A)$ as

the set of terminals that can appear immediately to the right of $A$ in some sentential form

Thus, a non-terminal’s FOLLOW set specifies the tokens that can legally appear after it.

A terminal symbol has no FOLLOW set

To build $\text{FOLLOW}(X)$ for non-terminal $X$:

1. **place eof in** $\text{FOLLOW}(<\text{goal}>)$

   **iterate until** no more terminals or $\epsilon$ can be added to any $\text{FOLLOW}(X)$:

2. if $A ::= \alpha B \beta$ then
   
   put $\{\text{FIRST}(\beta) - \epsilon\}$ in $\text{FOLLOW}(B)$

3. if $A ::= \alpha B$ then
   
   put $\text{FOLLOW}(A)$ in $\text{FOLLOW}(B)$

4. if $A ::= \alpha B \beta$ and $\epsilon \in \text{FIRST}(\beta)$ then
   
   put $\text{FOLLOW}(A)$ in $\text{FOLLOW}(B)$

   end iterate
Predictive, Top-down Parsing - LL(1)

Example:
\[ S ::= aSb | \epsilon \]

\[ FIRST(aSb) = \{a\} \]
\[ FIRST(\epsilon) = \{\epsilon\} \]
\[ FOLLOW(S) = \{eof, b\} \]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>eof</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>aSb</td>
<td>\epsilon</td>
<td>\epsilon</td>
<td>error</td>
</tr>
</tbody>
</table>

INPUT: a a a b b b eof
A larger example

expression grammar with precedence

\[
\langle \text{goal} \rangle ::= \langle \text{expr} \rangle \\
\langle \text{expr} \rangle ::= \langle \text{term} \rangle \langle \text{expr}' \rangle \\
\langle \text{expr}' \rangle ::= \text{+} \langle \text{expr} \rangle \\
| \quad \text{-} \langle \text{expr} \rangle \\
| \quad \epsilon \\
\langle \text{term} \rangle ::= \langle \text{factor} \rangle \langle \text{term}' \rangle \\
\langle \text{term}' \rangle ::= \text{*} \langle \text{term} \rangle \\
| \quad \text{/} \langle \text{term} \rangle \\
| \quad \epsilon \\
\langle \text{factor} \rangle ::= \text{num} \\
| \quad \text{id}
\]

LL(1) parse table

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>num</th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>\langle \text{goal} \rangle</td>
<td>g \rightarrow e</td>
<td>g \rightarrow e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\langle \text{expr} \rangle</td>
<td>e \rightarrow te'</td>
<td>e \rightarrow te'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\langle \text{expr}' \rangle</td>
<td></td>
<td></td>
<td>e' \rightarrow +e</td>
<td>e' \rightarrow -e</td>
<td></td>
<td></td>
<td>e' \rightarrow \epsilon</td>
</tr>
<tr>
<td>\langle \text{term} \rangle</td>
<td>t \rightarrow ft'</td>
<td>t \rightarrow ft'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\langle \text{term}' \rangle</td>
<td></td>
<td></td>
<td>t' \rightarrow \epsilon</td>
<td>t' \rightarrow \epsilon</td>
<td>t' \rightarrow *t</td>
<td>t' \rightarrow /t</td>
<td>t' \rightarrow \epsilon</td>
</tr>
<tr>
<td>\langle \text{factor} \rangle</td>
<td>f \rightarrow \text{id}</td>
<td>f \rightarrow \text{num}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Recursive descent parsing — LL(1)

Recursive descent is one of the simplest parsing techniques used in practical compilers:

- Each non–terminal has an associated parsing procedure that can recognize any sequence of tokens generated by that non–terminal.

- Within a parsing procedure, both non–terminals and terminals can be matched:
  - non–terminal $A$ — call parsing procedure for $A$
  - token $t$ — compare $t$ with current input token; if match, consume input, otherwise ERROR

- Parsing procedures may contain code that performs some useful “computation” (syntax directed translation).
Recursive Descent Parsing (pseudo code)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>eof</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>aSb</td>
<td>ε</td>
<td>ε</td>
<td>error</td>
</tr>
</tbody>
</table>

main: {
    token := next_token( );
    if (S( ) and token == eof) print ‘‘accept’’ else print ‘‘error’’;
}

bool S:
    switch token {
    case a: token := next_token( );
        if (not S( )) return false; // recursive call to S;
        if token == b {
            token := next_token( )
            return true;
        } else
            return false;
    break;
    case b,
    case eof:return true;
    break;
    default: return false;
    }

How to parse input a a a b b b ?
Recursive descent parser

For our larger example grammar

goal:
    token ← next_token();
    if (expr() = ERROR | token ≠ EOF) then
        return ERROR;
    else return OK;

expr:
    if (term() = ERROR) then
        return ERROR;
    else return expr_prime();

expr_prime:
    if (token = PLUS) then
        token ← next_token(); return expr();
    else if (token = MINUS) then
        token ← next_token(); return expr();
    else if (token = eof) then
        return OK;
    else return ERROR;
Recursive Descent Parsing (Cont.)

term:
   if (factor() = ERROR) then
      return ERROR;
   else return term_prime();

term_prime:
   if (token = MULT) then
      token ← next_token(); return term();
   else if (token = DIV) then
      token ← next_token(); return term();
   else if (token = eof) then
      return OK;
   if (token = PLUS) then
      return OK;
   if (token = MINUS) then
      return OK;
   else return ERROR;

factor:
   if (token = NUM) then
      token ← next_token(); return OK;
   else if (token = ID) then
      token ← next_token(); return OK;
   else return ERROR;
LL(1) grammars

Provable facts about LL(1) grammars:

• no left recursive grammar is LL(1)
• no ambiguous grammar is LL(1)
• LL(1) parsers operate in linear time
• an ϵ-free grammar where each alternative expansion for A begins with a distinct terminal is a simple LL(1) grammar

Not all grammars are LL(1)

• $S ::= aS \mid a$
  is not LL(1)
  $\text{FIRST}(aS) = \text{FIRST}(a) = \{a\}$

  $S ::= aS'$

  $S' ::= aS' \mid \epsilon$
  accepts the same language and is LL(1)
LL grammars

LL(1) grammars

- may need to rewrite grammar
  (left recursion removal, left factoring)
- resulting grammar larger, less maintainable

LL(k) grammars

- $k$-token lookahead
- more powerful than LL(1) grammars
- example:

  \[ S ::= ac \mid abc \text{ is } LL(2) \]

Not all grammars are LL(k)

- example:

  Set of productions of form:  \[ S ::= a^i b^j \text{ for } i \geq j \]
- problem - must choose production after $k$ tokens of lookahead

Bottom-up parsers avoid some of these problems
Bottom-up parsers

- start at the leaves and fill in
- construct rightmost derivation in reverse
- find the next right-hand side of a production (handle) such that its replacement by left-hand side nonterminal will yield previous right-sentential form
- as input is consumed, change state to encode possibilities (recognize valid prefixes); if handle is found, REDUCE, otherwise SHIFT (or ERROR)

\[ S \Rightarrow^{*}_{rm} \alpha By \Rightarrow_{rm} \alpha \gamma y \Rightarrow^{*}_{rm} xy \]

- **LR parsing**: Reads input from *left to right* and constructs *rightmost* derivation in reverse
Example:
S ::= a S b | c

How can we parse (automatically construct a right-most derivation backwards) the input string $a a a c b b b$ using a PDA (push-down automaton) and only the first symbol of the remaining input?

INPUT: $a a a c b b b\text{ eof}$
Larger Example

Consider the context-free grammar (in BNF notation)

\[
\begin{align*}
1 & : \langle \text{goal} \rangle ::= a \langle A \rangle \langle B \rangle e \\
2 & : \langle A \rangle ::= \langle A \rangle b c \\
3 & : | b \\
4 & : \langle B \rangle ::= d \\
\end{align*}
\]

and the input string \texttt{abbcde}.

<table>
<thead>
<tr>
<th>Prod’n.</th>
<th>Sentential Form</th>
<th>Handle†</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>\texttt{abbcde}</td>
<td>(3,2)</td>
</tr>
<tr>
<td>3</td>
<td>\texttt{a\langle A\rangle bcde}</td>
<td>(2,4)</td>
</tr>
<tr>
<td>2</td>
<td>\texttt{a\langle A\rangle de}</td>
<td>(4,3)</td>
</tr>
<tr>
<td>4</td>
<td>\texttt{a\langle A\rangle \langle B\rangle e}</td>
<td>(1,4)</td>
</tr>
<tr>
<td>1</td>
<td>\langle \text{goal} \rangle</td>
<td>—</td>
</tr>
</tbody>
</table>

Why is (3,3) not a handle for \texttt{a\langle A\rangle bcde}?

The trick appears to be scanning the input and finding valid right-sentential forms.

† (rule, position of right end of handle in input string).
Handles

We trying to find a substring $\alpha$ of the current right-sentential form where:

- $\alpha$ matches some production $A ::= \alpha$
- reducing $\alpha$ to $A$ is one step in the reverse of a rightmost derivation.

We will call such a string a handle.

Formally,

- a handle of a right-sentential form $\gamma$ is a production $A ::= \beta$ and a position in $\gamma$ where $\beta$ may be found. Convention: position sepcifies the right end of handle.
- If $(A ::= \beta, k)$ is a handle, then replacing the $\beta$ in $\gamma$ at position $k$ with $A$ produces the previous right-sentential form in a rightmost derivation of $\gamma$. 
Handles

Provable fact:

The substring to the right of a handle contains only terminal symbols.

Proof: Follows from the fact that all $\gamma_i$ are right-sentential forms.

Corollary

The right end of a handle is to the right of the previously reduced variable.
Shift-reduce parsing

One scheme to implement a handle-pruning, bottom-up parser is called a *shift-reduce* parser.

Shift-reduce parsers use a *stack* and an *input buffer*

1. initialize stack with $\$

2. Repeat until the top of the stack is the goal symbol and the input token is *eof*

   a) *find the handle*
      
      if we don’t have a handle on top of the stack, *shift* an input symbol onto the stack

   b) *prune the handle*
      
      if we have a handle ($A ::= \beta, k$) on top of the stack, *reduce*
      
      i) pop $| \beta |$ symbols off the stack
      
      ii) push $A$ onto the stack
Example

Left-recursive expression grammar

(original form, before left recursion removal and factoring
→ our “larger” example LL(1) grammar)

1 \langle goal \rangle ::= \langle expr \rangle
2 \langle expr \rangle ::= \langle expr \rangle + \langle term \rangle
3 | \langle expr \rangle - \langle term \rangle
4 | \langle term \rangle
5 \langle term \rangle ::= \langle term \rangle \ast \langle factor \rangle
6 | \langle term \rangle / \langle factor \rangle
7 | \langle factor \rangle
8 \langle factor \rangle ::= num
9 | id
“x – 2 * y”

1. **Shift until top of stack is the right end of a handle**

2. **Find the left end of the handle and reduce**

5 shifts + 9 reduces + 1 accept
Viable prefix

A viable prefix is

1. a prefix of a right-sentential form that does not continue past the right end of the handle of that sentential form†, or

2. a prefix of a right-sentential form that can appear on the stack of a shift-reduce parser.

It is always possible to add terminals onto the end of a viable prefix to obtain a right-sentential form.

As long as the prefix represented by the stack is viable, the parser has not seen a detectable error.

† If the grammar is unambiguous, there is a unique rightmost handle. LR(k) grammars are unambiguous.
Shift-reduce parsing

Grammars that are often used to construct shift-reduce parsers:

• operator grammars (will not discuss here → Aho, Sethi, Ullman p.203)
• LR(1) grammars
  – canonical LR(1) grammars
  – simple LR(1) grammars (SLR(1))
  – lookahead LR(1) grammars (LALR(1))

Grammars use different methods or levels of ”context” information to detect handle.

LR(1), SLR(1) and LALR(1)) grammars use finite automata (NFA or DFA) together with a PDA to recognize viable prefixes and store ”context” information.
**LR(k) grammars**

Informally, we say that a grammar $G$ is LR(k) if,

given a rightmost derivation

$$S' = \gamma_0 \Rightarrow_{rm} \gamma_1 \Rightarrow_{rm} \gamma_2 \Rightarrow_{rm} \cdots \Rightarrow_{rm} \gamma_n = w,$$

we can, for each right-sentential form in the derivation,

1. *isolate the handle of each right-sentential form*,

   and

2. *determine the production by which to reduce*

by scanning $\gamma_i$ from left to right, going at most $k$ symbols beyond the right end of the handle of $\gamma_i$. 
Table-driven LR parsing

A table-driven LR(k) parser looks like

Stack two items per state: state and symbol
Why study LR(1) grammars?

- All context-free, deterministic languages have an LR(1) grammar. Therefore LR grammars describe a proper superset of the languages recognized by LL (predictive) parsers.
- LR grammars are the most general grammars that can be parsed by a non-backtracking, shift-reduce parser
- Efficient shift-reduce parsers can be implemented for LR(1) grammars — time proportional to tokens + reductions
- Easy to build since table construction can be automated
- LR parsers detect an error as soon as possible in a left-to-right scan of the input
- Everyone’s favorite parser (EFP) — tools widely available (example: yacc).
**LR(1) parsing**

The skeleton parser:

```plaintext
token = next_token()
repeat forever
    s = state on top of stack
    if action[s,token] = "shift $s_i$" then
        push token
        push state $s_i$
        token = next_token()
    else if action[s,token] = "reduce $A ::= \beta$" then
        pop 2 * |$\beta$| symbols
        s = state on top of stack
        push $A$
        push goto[s,A]
    else if action[s,token] = "accept" then
        return
    else error()
```

This takes $k$ shifts, $l$ reduces, and 1 accept, where $k$ is the length of the input string and $l$ is the length of the reverse rightmost derivation.

**Note:** See Figure 4.36, Aho, Lam, Sethi, and Ullman
**LR(0) parsing: our example**

<table>
<thead>
<tr>
<th>Stack (without states)</th>
<th>Input</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id - num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$id</td>
<td>- num * id</td>
<td>9,1</td>
<td>reduce 9</td>
</tr>
<tr>
<td>$\langle factor \rangle</td>
<td>- num * id</td>
<td>7,1</td>
<td>reduce 7</td>
</tr>
<tr>
<td>$\langle term \rangle</td>
<td>- num * id</td>
<td>4,1</td>
<td>reduce 4</td>
</tr>
<tr>
<td>$\langle expr \rangle</td>
<td>- num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$\langle expr \rangle -</td>
<td>num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$\langle expr \rangle - num</td>
<td>* id</td>
<td>8,3</td>
<td>reduce 8</td>
</tr>
<tr>
<td>$\langle expr \rangle - \langle factor \rangle</td>
<td>* id</td>
<td>7,3</td>
<td>reduce 7</td>
</tr>
<tr>
<td>$\langle expr \rangle - \langle term \rangle</td>
<td>* id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$\langle expr \rangle - \langle term \rangle *</td>
<td>id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$\langle expr \rangle - \langle term \rangle * id</td>
<td></td>
<td>9,5</td>
<td>reduce 9</td>
</tr>
<tr>
<td>$\langle expr \rangle - \langle term \rangle * \langle factor \rangle</td>
<td></td>
<td>5,5</td>
<td>reduce 5</td>
</tr>
<tr>
<td>$\langle expr \rangle - \langle term \rangle</td>
<td></td>
<td>3,3</td>
<td>reduce 3</td>
</tr>
<tr>
<td>$\langle expr \rangle</td>
<td></td>
<td>1,1</td>
<td>reduce 1</td>
</tr>
<tr>
<td>$\langle goal \rangle</td>
<td></td>
<td>none</td>
<td>accept</td>
</tr>
</tbody>
</table>

The corresponding grammar and language is not LR(0).

**Theorem:** A language $L$ has an LR(0) grammar iff

- $L$ is deterministic
- no proper prefix of a word in $L$ is in $L$ (*prefix property*)
LR parsing

There are three commonly used algorithms to build tables for an “LR” parser:

1. $SLR(1) = LR(0) + FOLLOW$
   - smallest class of grammars
   - smallest tables (number of states)
   - simple, fast construction

2. $LR(1)$
   - full set of LR(1) grammars
   - largest tables (number of states)
   - slow, large construction

3. $LALR(1)$
   - intermediate sized set of grammars
   - same number of states as SLR(1)
   - canonical construction is slow and large
   - better construction techniques exist

An $LR(1)$ parser for either ALGOL or PASCAL has several thousand states, while an $SLR(1)$ or $LALR(1)$ parser for the same language may have several hundred states.
**SLR(1) parsing**

**Viable prefix** of a right-sentential form:

- contains both terminals and nonterminals
- can be recognized using a DFA

Building a *SLR* parser

- construct DFA for recognizing handles
- augment with FOLLOW to disambiguate actions

States in the DFA are sets of *LR(0)* items (subset construction)

**Note:** An “augmented/extended grammar” (ECFG) is one where the start symbol appears only on the *lhs* of productions. For the rest of LR parsing, we will assume the grammar is augmented with a production $S' ::= S$
**LR(0) items**

An *LR(0) item* is a string \([\alpha]\), where

\(\alpha\) is a production from \(G\) with a • at some position in the rhs

The • indicates how much of an item we have seen at a given state in the parsing process.

\([A ::= \bullet XYZ]\) indicates that the parser is looking for a string that can be derived from \(XYZ\)

\([A ::= XY \bullet Z]\) indicates that the parser has seen a string derived from \(XY\) and is looking for one derivable from \(Z\)

**LR(0) Items**  
(no lookahead)

\(A ::= XYZ\) generates 4 \(LR(0)\) items.

1. \([A ::= \bullet XYZ]\)
2. \([A ::= X \bullet YZ]\)
3. \([A ::= XY \bullet Z]\)
4. \([A ::= XYZ\bullet]\)
**Canonical $LR(0)$ items**

The $SLR(1)$ table construction algorithm uses a specific set of sets of $LR(0)$ items.

These sets are called the *canonical collection of sets of $LR(0)$ items* for a grammar $G$.

The canonical collection corresponds to the set of states of the DFA that recognizes viable prefixes. Each state is the set of valid LR(0) items at a particular point in the parse.

The LR(0) item $[A ::= \beta_1 \cdot \beta_2]$ is valid for a viable prefix $\alpha\beta_1$ if there is a derivation $S' \Rightarrow^*_rm \alpha Aw \Rightarrow_{rm} \alpha\beta_1\beta_2w$. In general, an item will be valid for many viable prefixes.
**SLR(1) parser example**

### The Grammar

1. \( E ::= T + E \)
2. \( E ::= T \)
3. \( T ::= \text{id} \)

### The Augmented Grammar

0. \( S' ::= E \)
1. \( E ::= T + E \)
2. \( E ::= T \)
3. \( T ::= \text{id} \)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>\text{FIRST}</th>
<th>\text{FOLLOW}</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S' )</td>
<td>{ id }</td>
<td>{ \text{eof} }</td>
</tr>
<tr>
<td>( E )</td>
<td>{ id }</td>
<td>{ \text{eof} }</td>
</tr>
<tr>
<td>( T )</td>
<td>{ id }</td>
<td>{ +, \text{eof} }</td>
</tr>
</tbody>
</table>
Canonical Collection of LR(0) items

To construct the canonical collection we need two functions:

- **closure(I)**
  
  if $[A ::= \alpha \bullet B\beta] \in I_j$, then, in state $j$, the parser might next see a string derivable from $B\beta$  
  $\Rightarrow$ to form its closure, add all items of the form 
  $[B ::= \bullet \gamma] \in G$

- **GOTO(I, X)**
  
  If $I$ is the set of items that are valid for some viable prefix $\gamma$, then $\text{GOTO}(I, X)$ is the set of items that are valid for the viable prefix $\gamma X$. 
Closure($I$)

Given an item $[A ::= \alpha \bullet B\beta]$, its closure contains the item and any other items that can generate legal substrings to follow $\alpha$.

Thus, if the parser has viable prefix $\alpha$ on its stack, the input should reduce to $B\beta$ (or $\gamma$ for some other item $[B ::= \bullet \gamma]$ in the closure).

To compute closure($I$)

function closure($I$)  
repeat  
    new_item ← false  
    for each item $[A ::= \alpha \bullet B\beta] \in I$,  
    each production $B ::= \gamma \in G'$  
    if $[B ::= \bullet \gamma] \notin I$ then  
        add $[B ::= \bullet \gamma]$ to $I$  
        new_item ← true  
    endif  
    until (new_item = false)  
return $I$
**Goto**\((I, X)\)

Let \(I\) be a set of \(LR(0)\) items and \(X\) be a grammar symbol.

Then, \(\text{GOTO}(I, X)\) is the closure of the set of all items

\[[A ::= \alpha X \bullet \beta] \text{ such that } [A ::= \alpha \bullet X \beta] \in I\]

If \(I\) is the set of valid items for some viable prefix \(\gamma\), then \(\text{goto}(I, X)\) is the set of valid items for the viable prefix \(\gamma X\).

\(\text{goto}(I, X)\) represents state after recognizing \(X\) in state \(I\).

To compute \(\text{goto}(I, X)\)

```plaintext
function goto(I, X)
    J ← set of items \([A ::= \alpha X \bullet \beta]\)
    such that \([A ::= \alpha \bullet X \beta] \in I\)
    J' ← closure(J)
    return J'
```

Collection of sets of $LR(0)$ items

We start the construction of the collection of sets of $LR(0)$ items with the item $[S' ::= \bullet S]$, where

- $S'$ is the start symbol of the augmented grammar $G'$
- $S$ is the start symbol of $G$

To compute the collection of sets of LR(0) items

```plaintext
procedure items($G'$)
    $S_0 \leftarrow \text{closure}([S' ::= \bullet S])$
    Items $\leftarrow \{ S_0 \}$
    ToDo $\leftarrow \{ S_0 \}$
    while ToDo not empty do
        remove $S_i$ from ToDo
        for each grammar symbol $X$ do
            $S_{new} \leftarrow \text{goto}(S_i, X)$
            if $S_{new}$ is a new state then
                Items $\leftarrow$ Items $\cup \{ S_{new} \}$
                ToDo $\leftarrow$ ToDo $\cup \{ S_{new} \}$
            endif
        endfor
    endwhile
    return Items
```
**LR(0) machines**

**LR(0) DFA**

- states – canonical sets of LR(0) items
- edges – goto transitions
- recognizes handles and viable prefixes using stack
- no lookahead

Reducing a handle (rhs of production) to a nonterminal can be viewed as:

- returning to state at beginning of handle
- making transition on nonterminal for this state

To return to state at beginning of the handle, we must use the stack to store the state!
**SLR(1) tables**

**SLR(1) parser**

- augment **LR(0)** machine
- add FOLLOW information using one token of lookahead
- encoded as **ACTION, GOTO** tables

**ACTION table**

- for each [state, lookahead] pair
- have we reached end of handle?
- if not, shift
- if at end of handle, reduce
- may also accept or error
- use lookahead to guide decision

**GOTO table**

- for each [state, nonterminal] pair
- pick state to go to after reduction
**SLR(1) table construction**

**The Algorithm**

1. construct the collection of sets of LR(0) items for \( G' \).

2. State \( i \) of the parser is constructed from \( I_i \).

   (a) if \([A ::= \alpha \cdot a\beta] \in I_i\) and \( \text{goto}(I_i, a) = I_j \), then set \( \text{ACTION}[i, a] \) to “shift \( j \)”.

   (a must be a terminal)

   (b) if \([A ::= \alpha\. \in I_i]\), then set \( \text{ACTION}[i, a] \) to “reduce \( A ::= \alpha \)” for all \( a \) in \( \text{FOLLOW}(A) \).

   (c) if \([S' ::= S\. \in I_i]\), then set \( \text{ACTION}[i, \text{eof}] \) to “accept”.

3. If \( \text{goto}(I_i, A) = I_j \), then set \( \text{GOTO}[i, A] \) to \( j \).

4. All other entries in \( \text{ACTION} \) and \( \text{GOTO} \) are set to “error”

5. The initial state of the parser is the state constructed from the set containing the item \([S' ::= \cdot S]\).
SLR(1) parser example

DFA for handles / viable prefixes based on LR(0) cannonical collection
Example LR(0) states

$S_0$:  
\[
S' ::= \bullet E, \\
E ::= \bullet T + E, \\
E ::= \bullet T, \\
T ::= \bullet \text{id}
\]

$S_1$:  
\[
S' ::= E \bullet
\]

$S_2$:  
\[
E ::= T \bullet + E, \\
E ::= T \bullet
\]

$S_3$:  
\[
T ::= \text{id} \bullet
\]

$S_4$:  
\[
E ::= T + \bullet E, \\
E ::= \bullet T + E, \\
E ::= \bullet T, \\
T ::= \bullet \text{id}
\]

$S_5$:  
\[
E ::= T + E \bullet
\]
Example GOTO function

Start

\[ S_0 \leftarrow \text{closure ( } \{ [ S ::= \bullet E ] \} \) \]

Iteration 1

\[ \text{goto}(S_0, E) = S_1 \]
\[ \text{goto}(S_0, T) = S_2 \]
\[ \text{goto}(S_0, \text{id}) = S_3 \]

Iteration 2

\[ \text{goto}(S_2, +) = S_4 \]

Iteration 3

\[ \text{goto}(S_4, \text{id}) = S_3 \]
\[ \text{goto}(S_4, E) = S_5 \]
\[ \text{goto}(S_4, T) = S_2 \]
Example ACTION and GOTO tables

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>id +</td>
<td>E</td>
</tr>
<tr>
<td>eof</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>eof</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$</td>
<td>shift 3</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$S_1$</td>
<td>—</td>
<td>—</td>
<td>accept</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>$S_2$</td>
<td>—</td>
<td>shift 4</td>
<td>reduce 2</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>$S_3$</td>
<td>—</td>
<td>reduce 3</td>
<td>reduce 3</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>$S_4$</td>
<td>shift 3</td>
<td>—</td>
<td>—</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>$S_5$</td>
<td>—</td>
<td>—</td>
<td>reduce 1</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>id + id eof</td>
<td>shift 3</td>
</tr>
<tr>
<td>$0$ id $3$</td>
<td>+ id eof</td>
<td>reduce 3 (T := id)</td>
</tr>
<tr>
<td>$0$ T $2$</td>
<td>+ id eof</td>
<td>shift 4</td>
</tr>
<tr>
<td>$0$ T $2 + 4$</td>
<td>id eof</td>
<td>shift 3</td>
</tr>
<tr>
<td>$0$ T $2 + 4$ id $3$</td>
<td>eof</td>
<td>reduce 3 (T := id)</td>
</tr>
<tr>
<td>$0$ T $2 + 4$ T $2$</td>
<td>eof</td>
<td>reduce 2 (E := T)</td>
</tr>
<tr>
<td>$0$ T $2 + 4$ E $5$</td>
<td>eof</td>
<td>reduce 1 (E := T + E)</td>
</tr>
<tr>
<td>$0$ E $1$</td>
<td>eof</td>
<td>accept</td>
</tr>
</tbody>
</table>
What can go wrong?

Example: A simple grammar

1. $S' ::= S$
2. $S ::= L = R$
3. $S ::= R$
4. $L ::= *R$
5. $L ::= \text{id}$
6. $R ::= L$

Canonical LR(0) collection

$I_0 : \{ \ [S' ::= \bullet S], [S ::= \bullet L = R], [S ::= \bullet R],$

$[L ::= \bullet * R], [L ::= \bullet \text{id}], [R ::= \bullet L] \ \}$

$I_1 : \{ \ [S' ::= S\bullet] \ \}$

$I_2 : \{ \ [S ::= L\bullet = R], [R ::= L\bullet] \ \}$

$I_3 : \{ \ [S ::= R\bullet] \ \}$

$I_4 : \{ \ [L ::= * \bullet R], [R ::= \bullet L], [L ::= \bullet * R],$

$[L ::= \bullet \text{id}] \ \}$

$I_5 : \{ \ [L ::= \text{id}\bullet] \ \}$

$I_6 : \{ \ [S ::= L = \bullet R], [R ::= \bullet L], [L ::= \bullet * R],$

$[L ::= \bullet \text{id}] \ \}$

$I_7 : \{ \ [L ::= *R\bullet] \ \}$

$I_8 : \{ \ [R ::= L\bullet] \ \}$

$I_9 : \{ \ [S ::= L = R\bullet] \ \}$
SLR(1) table construction

<table>
<thead>
<tr>
<th>Symbol</th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>S'</td>
<td>{ id, * }</td>
<td>{ eof }</td>
</tr>
<tr>
<td>S</td>
<td>{ id, * }</td>
<td>{ eof }</td>
</tr>
<tr>
<td>L</td>
<td>{ id, * }</td>
<td>{ =, eof }</td>
</tr>
<tr>
<td>R</td>
<td>{ id, * }</td>
<td>{ =, eof }</td>
</tr>
</tbody>
</table>

Consider the set of items \( I_2 \). The action table is defined as follows:

\[
[S \ ::= \ L \cdot = R] \implies \text{ACTION}[2, =] = "shift 6"
\]

\[
[R \ ::= \ L \cdot] \implies \text{ACTION}[2, =] = "reduce 6"
\]

Due to multiple definitions of the position in the action table, the grammar is not SLR(1).
What can go wrong?

Two cases arise

\textit{shift/reduce}

This is called a \textit{shift/reduce} conflict. In general, it indicates an ambiguous construct in the grammar.

- can modify the grammar to eliminate it
- can resolve in favor of shifting

\textbf{classic example:} dangling else

\textit{reduce/reduce}

This is called a \textit{reduce/reduce} conflict. Again, it indicates an ambiguous construct in the grammar.

- often, no simple resolution
- parse a nearby language

\textbf{classic example:} PL/I call and subscript
**LR(1)**

SLR(1) parsers may not be able to parse some LR grammars.

Problem is that lookahead information is added to LR(0) parser at the end of construction.

We can get more powerful parser by keeping track of lookahead information in the states of the parser.

If, in a single left-to-right scan, we can construct a reverse rightmost derivation, while using at most a single token lookahead to resolve ambiguities, then the grammar is LR(1).
**LR(k) items**

The table construction algorithms use LR(k) items to represent the set of possible states in a parse.

An LR(k) item is a pair \([\alpha, \beta]\), where

- \(\alpha\) is a production from \(G\) with a \(\bullet\) at some position in the rhs
- \(\beta\) is a lookahead string containing \(k\) symbols (terminals or \texttt{eof})

What about LR(1) items?

- example LR(1) item: \([A ::= X \bullet YZ, a]\)
- LR(1) items have lookahead strings of length 1
- several LR(1) items may have the same core

\[
[A ::= X \bullet YZ, a] \\
[A ::= X \bullet YZ, b]
\]

we represent this as

\[
[A ::= X \bullet YZ, \{a, b\} ]
\]
**LR(1) lookahead**

What’s the point of all these lookahead symbols?

- carry them along to allow us to choose correct reduction when there is any choice
- lookaheads are bookkeeping unless item has • at right end.
  - in \([A ::= X \bullet YZ, a], a\) has no direct use
  - in \([A ::= XYZ\bullet, a], a\) is useful

Recall, the SLR(1) construction uses LR(0) items!

---

**The point**

For \([A ::= \alpha\bullet, a]\) and \([B ::= \alpha\bullet, b]\), we can decide between reducing to \(A\) and to \(B\) by looking at limited right context!
Canonical \( LR(1) \) items

The canonical collection of sets of \( LR(1) \) items:

- sets of valid items for viable prefixes of the grammar
- sets of items derivable from \([S' ::= \bullet S, \text{eof}]\) using \texttt{goto} and \texttt{closure} functions — both functions preserve validity.

A \( LR(1) \) item \([A ::= \alpha \bullet \beta, a]\) is \textit{valid} for a viable prefix \( \gamma \) if there is a derivation \( S \Rightarrow^*_{rm} \delta Aw \Rightarrow_{rm} \delta \alpha \beta w \), where

- \( \gamma = \delta \alpha \), and

- either \( a \) is the first symbol of \( w \), or \( w \) is \( \epsilon \) and \( a \) is \texttt{eof}.

Essentially,

- Each \( LR(1) \) item in a set in the canonical collection represents a state in an NFA that recognizes viable prefixes.

- Grouping these items together is really the DFA subset construction.
**LR(1) closure**

Given an item \([A := \alpha \cdot B\beta, a]\), its closure contains the item and any other items that can generate legal substrings to follow \(\alpha\).

Thus, if the parser has viable prefix \(\alpha\) on its stack, a substring of the input should reduce to \(B\beta\) (or \(\gamma\) for some other item \([B := \bullet\gamma, b]\) in the closure).

To compute \(\text{closure}(I)\)

```plaintext
function closure(I)
repeat
    new_item ← false
    for each item \([A := \alpha \cdot B\beta, a]\) ∈ I,
        each production \(B ::= \gamma \in G'\),
        and each terminal \(b \in \text{FIRST}(\beta a)\),
        if \([B := \bullet\gamma, b]\) \(\notin I\) then
            add \([B := \bullet\gamma, b]\) to I
            new_item ← true
    endif
    until (new_item = false)
return I
```

Aho, Sethi, and Ullman, Figure 4.38
**LR(1) goto**

Let $I$ be a set of LR(1) items and $X$ be a grammar symbol. Then, $\text{goto}(I, X)$ is the closure of the set of all items

$$[A ::= \alpha X \bullet \beta, a] \text{ such that } [A ::= \alpha \bullet X\beta, a] \in I$$

If $I$ is the set of valid items for some viable prefix $\gamma$, then $\text{goto}(I, X)$ is the set of valid items for the viable prefix $\gamma X$.

goto($I, X$) represents state after recognizing $X$ in state $I$.

To compute goto($I, X$)

```plaintext
function goto(I, X)
    J ← set of items $[A ::= \alpha X \bullet \beta, a]$ such that $[A ::= \alpha \bullet X\beta, a] \in I$
    J' ← closure(J)
    return J'
```

Aho, Sethi, and Ullman, Figure 4.38
Collection of sets of $LR(1)$ items

We start the construction of the canonical collection of $LR(1)$ items with the item $[S' ::= \bullet S, \text{eof}]$, where

- $S'$ is the start symbol of the augmented grammar $G'$
- $S$ is the start symbol of $G$, and
- $\text{eof}$ is the right end of string marker

To compute the collection of sets of $LR(1)$ items

```
procedure items($G'$)
    C ← {closure({[S' ::= \bullet S, \text{eof}]})}
    repeat
        new_item ← false
        for each set of items $I$ in C and each grammar symbol $X$ such that
            goto($I, X$) $\neq \emptyset$ and
            goto($I, X$) $\notin C$
            add goto($I, X$) to C
        new_item ← true
    endfor
    until (new_item = false)
```

Aho, Sethi, and Ullman, Figure 4.38
LR(1) table construction

The Algorithm

1. construct the collection of sets of LR(1) items for $G'$.

2. State $i$ of the parser is constructed from $I_i$.
   
   (a) if $[A ::= \alpha \bullet a \beta, b] \in I_i$ and \text{goto}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to "shift $j$". ($a$ must be a terminal)
   
   (b) if $[A ::= \alpha\bullet, a] \in I_i$, then set $\text{ACTION}[i, a]$ to "reduce $A ::= \alpha$".
   
   (c) if $[S' ::= S\bullet, \text{eof}] \in I_i$, then set $\text{ACTION}[i, \text{eof}]$ to "accept".

3. If \text{goto}(I_i, A) = I_j$, then set $\text{GOTO}[i, A]$ to $j$.

4. All other entries in $\text{ACTION}$ and $\text{GOTO}$ are set to "error"

5. The initial state of the parser is the state constructed from the set containing the item $[S' ::= \bullet S, \text{eof}]$.

Aho, Sethi, and Ullman, Algorithm 4.10
**Example**

Our simple grammar

1. \( S' ::= S \)
2. \( S ::= L = R \)
3. \( S ::= R \)
4. \( L ::= *R \)
5. \( L ::= \text{id} \)
6. \( R ::= L \)
Canonical LR(1) collection

$I_0 : \{ [S' ::= \bullet S, \text{eof}], [S ::= \bullet L = R, \text{eof}],$
\[ [S ::= \bullet R, \text{eof}], [L ::= \bullet \ast R, \{=, \text{eof}\}], \]
\[ [L ::= \bullet \text{id}, \{=, \text{eof}\}], [R ::= \bullet L, \text{eof}] \} \]

$I_1 : \{ [S' ::= S \bullet, \text{eof}] \} \]

$I_2 : \{ [S ::= L \bullet = R, \text{eof}], [R ::= L \bullet, \text{eof}] \} \]

$I_3 : \{ [S ::= R \bullet, \text{eof}] \} \]

$I_4 : \{ [L ::= \ast \bullet R, \{=, \text{eof}\}], [R ::= \bullet L, \{=, \text{eof}\}],$
\[ [L ::= \bullet \ast R, \{=, \text{eof}\}], [L ::= \bullet \text{id}, \{=, \text{eof}\}] \} \]

$I_5 : \{ [L ::= \text{id} \bullet, \{=, \text{eof}\}] \} \]

$I_6 : \{ [S ::= L = \bullet R, \text{eof}], [R ::= \bullet L, \text{eof}],$
\[ [L ::= \bullet \ast R, \text{eof}], [L ::= \bullet \text{id}, \text{eof}] \} \]

$I_7 : \{ [L ::= \ast R \bullet, \{=, \text{eof}\}] \} \]

$I_8 : \{ [R ::= L \bullet, \{=, \text{eof}\}] \} \]

$I_9 : \{ [S ::= L = R \bullet, \text{eof}] \} \]

$I_{10} : \{ [R ::= L \bullet, \text{eof}] \} \]

$I_{11} : \{ [L ::= \ast \bullet R, \text{eof}], [R ::= \bullet L, \text{eof}],$
\[ [L ::= \bullet \ast R, \text{eof}], [L ::= \bullet \text{id}, \text{eof}] \} \]

$I_{12} : \{ [L ::= \text{id} \bullet, \text{eof}] \} \]

$I_{13} : \{ [L ::= \ast R \bullet, \text{eof}] \} \]
So, does it work now?

Example: Consider the set of items $I_2$. The action table is defined as follows:

\[
\begin{align*}
[S ::= L\bullet = R, \text{eof}] & \Rightarrow \text{ACTION}[2,=] = "shift 6" \\
[R ::= L\bullet, \text{eof}] & \Rightarrow \text{ACTION}[2,\text{eof}] = "reduce 6"
\end{align*}
\]

Yes, the table construction defines only single entries for each position in the ACTION table.

The price we pay: more states

- construction of table takes longer
- table is larger

Here is an idea: LALR(1) parsing: Union states that are sets of $LR(1)$ items with the same core.
**LALR(1) parsing**

LR(1) parsers have many more states than SLR(1) parsers (approximately factor of ten for Pascal).

LALR(1) parsers have same number of states as SLR(1) parsers, but with more power due to lookahead in states.

Define the core of a set of LR(1) items to be the set of LR(0) items derived by ignoring the lookahead symbols.

Thus, the two sets

- \{[A ::= \alpha \cdot \beta, a], [A ::= \alpha \cdot \beta, b]\}, and
- \{[A ::= \alpha \cdot \beta, c], [A ::= \alpha \cdot \beta, d]\}

have the same core.

**Key Idea:**

If two sets of LR(1) items, $I_i$ and $I_j$, have the same core, we can merge the states that represent them in the ACTION and GOTO tables.
Back to our example grammar

1. $S' ::= S$
2. $S ::= L = R$
3. $S ::= R$
4. $L ::= *R$
5. $L ::= id$
6. $R ::= L$

$I_0 : \{ [S' ::= \bullet S, \text{eof}], [S ::= \bullet L = R, \text{eof}],$
       $[S ::= \bullet R, \text{eof}], [L ::= \bullet * R, \{=, \text{eof}\}],$
       $[L ::= \bullet id, \{=, \text{eof}\}], [R ::= \bullet L, \text{eof}] \}$

$I_1 : \{ [S' ::= S \bullet, \text{eof}] \}$

$I_2 : \{ [S ::= L \bullet = R, \text{eof}], [R ::= L \bullet, \text{eof}] \}$

$I_3 : \{ [S ::= R \bullet, \text{eof}] \}$

$I_{4/11} : \{ [L ::= * \bullet R, \{=, \text{eof}\}], [R ::= \bullet L, \{=, \text{eof}\}],$
          $[L ::= \bullet * R, \{=, \text{eof}\}], [L ::= \bullet id, \{=, \text{eof}\}] \}$

$I_{5/12} : \{ [L ::= id \bullet, \{=, \text{eof}\}] \}$

$I_6 : \{ [S ::= L = \bullet R, \text{eof}], [R ::= \bullet L, \text{eof}],$
       $[L ::= \bullet * R, \text{eof}], [L ::= \bullet id, \text{eof}] \}$

$I_{7/13} : \{ [L ::= * \bullet R, \{=, \text{eof}\}] \}$

$I_{8/10} : \{ [R ::= L \bullet, \{=, \text{eof}\}] \}$

$I_9 : \{ [S ::= L = R \bullet, \text{eof}] \}$
### Example

1. \( S' ::= S \)
2. \( S ::= L = R \)
3. \( S ::= R \)
4. \( L ::= *R \)
5. \( L ::= \text{id} \)
6. \( R ::= L \)

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{id} )</td>
<td>S</td>
</tr>
<tr>
<td>( * )</td>
<td>s5/12</td>
</tr>
<tr>
<td>( = )</td>
<td>—</td>
</tr>
<tr>
<td>( \text{eof} )</td>
<td>—</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_0 )</td>
<td>—</td>
</tr>
<tr>
<td>( S_1 )</td>
<td>—</td>
</tr>
<tr>
<td>( S_2 )</td>
<td>—</td>
</tr>
<tr>
<td>( S_3 )</td>
<td>—</td>
</tr>
<tr>
<td>( S_{4/11} )</td>
<td>s5/12</td>
</tr>
<tr>
<td>( S_{5/12} )</td>
<td>—</td>
</tr>
<tr>
<td>( S_6 )</td>
<td>s5/12</td>
</tr>
<tr>
<td>( S_{7/13} )</td>
<td>—</td>
</tr>
<tr>
<td>( S_{8/10} )</td>
<td>—</td>
</tr>
<tr>
<td>( S_9 )</td>
<td>—</td>
</tr>
</tbody>
</table>
**LALR(1) properties**

LALR(1) parsers have same number of states as SLR(1) parsers (core LR(0) items are the same)

In case of error, LALR(1) parser may perform more reductions than corresponding LR(1) parser, but will catch error before more input is processed.

Example grammar with input "id = id =":

\[
\begin{align*}
LR(1): & \quad S_0 \rightarrow^s S_5 \rightarrow^r S_2 \rightarrow^s S_6 \rightarrow^s S_{12} \Rightarrow error \\
LALR(1): & \quad S_0 \rightarrow^s S_{5/12} \rightarrow^r S_2 \rightarrow^s S_6 \rightarrow^s S_{5/12} \rightarrow^r S_{8/10} \rightarrow^r S_9 \Rightarrow error
\end{align*}
\]
Summary: Resolving parse conflicts

Parse conflicts possible when certain LR items are found in the same state.

Depending on parser, may choose between LR items using lookahead.

Legal lookahead for LR items must be disjoint, else conflict exists.

<table>
<thead>
<tr>
<th>LR(0)</th>
<th>Shift-Reduce</th>
<th>Reduce-Reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[ A ::= α • , ∆ ]</td>
<td>[ A ::= α • , ∆ ]</td>
</tr>
<tr>
<td></td>
<td>[ B ::= β • γ , Ω ]</td>
<td>[ B ::= β • , Ω ]</td>
</tr>
<tr>
<td></td>
<td>conflict</td>
<td>conflict</td>
</tr>
<tr>
<td>SLR(1)</td>
<td>FOLLOW(A) ∩ FIRST(γ)</td>
<td>FOLLOW(A) ∩ FOLLOW(B)</td>
</tr>
<tr>
<td></td>
<td>LR(1)</td>
<td>Δ ∩ FIRST(γ)</td>
</tr>
</tbody>
</table>
Next class: Syntax Directed Translation (SDT)

We will finish up with bottom-up parsing.

For SDT, please read: Scott: Chapters 4.1-4.4;
ALSU: Chapters 5.1-5.3;

Homework will be out by Friday. Please see our course website.