

CSE 131A
Lecture 10

LR Parser Table Construction

Scott B. Baden

Department of Computer Science & Engineering

University of California, San Diego

Winter 2007

Today's lecture

- Constructing LR Parser tables
- Today's reading continues with §4.7 of the text

Announcements

- Correction to error test case outputs so they are consistent with the A2 specification

if (true) then 4 else 5

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<errors>
```

```
  <syntaxError column="11" line="1"/>
```

```
</errors>
```

- Xml outputs for all error test cases were reloaded

kw01.onyx

punctuation.onyx

e1.onyx

iv.onyx

literals.onyx

strvar.onyx

integers.onyx

dot.onyx

COMMENTS for NEXT TIME

- The discussions about the goto() function and Canonical LR(0) collection are muddled esp the Canonical collection
- Use the discussion from the Midterm review to clarify things

Recapping from last time

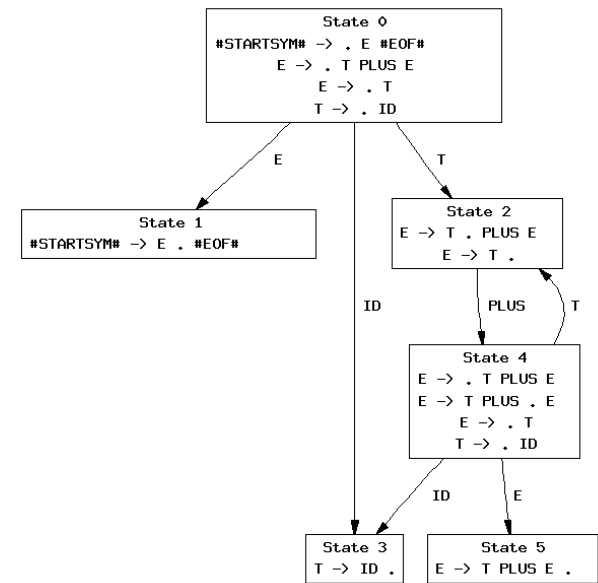
- The LR parser pushes symbols onto the top of stack..
- ... performing reductions when it encounters a handle
- We solved the problem of how to identify handles ..
- ... by recasting it as the problem of identifying viable prefixes
- Invariant: the TOS always contains a viable prefix, which will eventually become a handle
- The state encodes the history of past shifts and reductions enabling the parser to recognize a handle simply by consulting a table and the lookahead
- The parse tables tell the parser what action to take next and what state to enter terminating in an **accept** or an **error**

Constructing the DFA

- The set of viable prefixes form a regular language
- A DFA can be used to recognize the viable prefixes
- We'll think of the parser in terms of a NFA
 - The states of the NFA will correspond to a recognized handle or viable prefix
 - State transitions correspond to increasingly closer matches with RHSs of the grammar
- We'll construct a DFA to recognize the viable prefixes and use it to fill in the tables for a LR parser
- We will concentrate now on how to construct SLR (“Simple” LR) parsing tables

NFA

- The NFA is completely determined by the grammar
- Each accept state corresponds to a grammar rule, ensuring that RHS matches the symbols on the TOS
- There is one accept state for each handle
- The remaining states correspond to partial matches between TOS and an emerging handle
- State transitions are on symbols
 - Lead to states indicating further progress in identifying the handle
 - Epsilon transitions are also possible



$E' \rightarrow E$
 $E \rightarrow T + E \quad | \quad T$
 $T \rightarrow \text{id}$

Matching Right Hand Sides

- Consider the grammar

$$E \rightarrow E + T \mid T$$

$$T \rightarrow (E) \mid \text{id}$$

- We mark each RHS with a dot • showing how much has been matched with TOS
 - dot at left end: nothing matched yet
 - dot at right end: entire RHS matched
- Marking rules this way gives the set of *items* of the grammar
- There are 4 items for the production $E \rightarrow E + T$

$$\bullet E + T, E \rightarrow E \bullet + T, E \rightarrow E + \bullet T, E \rightarrow E + T \bullet$$

An example

- Our simple calculator

$$\begin{aligned} E &\rightarrow E + T \quad | \quad T \\ T &\rightarrow T * F \quad | \quad F \\ F &\rightarrow (E) \quad | \quad \mathbf{id} \end{aligned}$$

- The set of items of this grammar

$$\begin{aligned} E &\rightarrow \bullet E + T, E \rightarrow E \bullet + T, E \rightarrow E + \bullet T, E \rightarrow E + T \bullet, \\ E &\rightarrow \bullet T, E \rightarrow T \bullet, \\ T &\rightarrow \bullet T * F, T \rightarrow T \bullet * F, T \rightarrow T * \bullet F, T \rightarrow T * F \bullet, \\ T &\rightarrow \bullet F, T \rightarrow F \bullet, \\ F &\rightarrow \bullet (E), F \rightarrow (\bullet E), F \rightarrow (E \bullet), F \rightarrow (E) \bullet, \\ F &\rightarrow \bullet \mathbf{id}, F \rightarrow \mathbf{id} \bullet \end{aligned}$$

Grammar items and NFA states

- A grammar item corresponds to a match, possibly partially complete, with symbols on TOS
- Each grammar item corresponds to a state of the NFA that recognizes the handles
- Items with dot all the way to the right will correspond to accept states

$$\begin{aligned} F &\rightarrow \bullet \text{id} , & F &\rightarrow \text{id} \bullet \\ T &\rightarrow \bullet F , & T &\rightarrow F \bullet , \end{aligned}$$

Example

- When in state $E \rightarrow \bullet E+T$
transition on symbol E
to state $E \rightarrow E \bullet +T$
- Transition on symbol $+$ to state $E \rightarrow E+ \bullet T$
- Transition on symbol T to state $E \rightarrow E+ T \bullet$
- **Accept** (reduce $E \rightarrow E+ T$)

Multiple transitions

- On the item's RHS, the symbols to the left of the dot must match symbols at the top of stack
- Transitions depend on the symbol, and move the dot over that symbol
- We can have multiple items matching TOS, with same symbol just to the right of the dot:
 - $E \rightarrow \bullet E+T$ matches anything on TOS, transition on E
 - $F \rightarrow (\bullet E)$ matches $($ on TOS, transition on E
- Multiple transitions on the same symbol imply an NFA

State transitions

- State transitions occur on symbols
- A state transition on a terminal symbol occurs when we shift a token onto the stack

$F \rightarrow \bullet (E)$ $\gg \gg \gg \gg \gg \gg \gg$ $F \rightarrow (\bullet E)$
 $\$...$ $\$... ($

$E \rightarrow E + T \quad | \quad T$
 $T \rightarrow T * F \quad | \quad F$
 $F \rightarrow (E) \quad | \quad \text{id}$

- A state transition on a nonterminal symbol occurs when we detect a handle, shifting a nonterminal onto the stack

$E \rightarrow \bullet E + T$ $\gg \gg \gg \gg \gg \gg \gg$ $E \rightarrow E \bullet + T$

- We need ϵ -transitions to handle reductions
- The new states will ultimately enable us to shift terminal symbols, or enter an **accept** or **error** state

ϵ -transitions

- We need ϵ -transitions to handle reductions
- The new states will ultimately enable us to shift terminal symbols, or enter an **accept** or **error** state
- Shifts a nonterminal onto the stack
- For example, the parse of **(id)**

$F \rightarrow \mathbf{id} \quad (E) \quad T \rightarrow F \quad E \rightarrow T$

$E \rightarrow E + T \quad | \quad T$
 $T \rightarrow T * F \quad | \quad F$
 $F \rightarrow (E) \quad | \quad \mathbf{id}$

ϵ -transitions

- For example, consider the state $E \rightarrow \bullet T$
- When we shift T onto the stack we arrive at $E \rightarrow T \bullet$
- Thus, T is pushed onto the stack
- This is possible because there is a grammar rule $T \rightarrow F$
 - If there was an F on TOS, the reduction would put T on top
 - We add an ϵ -transition on $T \rightarrow \bullet F$ (what else?)
- This implies that an F could be on the top of stack, so we add ϵ -transitions from $T \rightarrow \bullet F$ to $F \rightarrow \bullet (E)$ and $F \rightarrow \bullet \mathbf{id}$

$$\begin{array}{l} E \rightarrow E + T \quad | \quad T \\ T \rightarrow T * F \quad | \quad F \\ F \rightarrow (E) \quad | \quad \mathbf{id} \end{array}$$

Viabke-prefix NFA to DFA

- We can think of each grammar item as corresponding to a state in some NFA that ensures the stack contains a viable prefix, and can recognize handles at TOS
 - the part of the RHS to the left of the dot represents the end of the prefix on the stack; the part to the right of the dot represents the possible remainder of a handle
 - “nondeterministic” because more than one item with same symbol to right of dot may be consistent with what is currently on the stack and with initial part of the unread input
- We can convert the NFA into a DFA using subset construction: organize *sets of grammar items* corresponding to states in the DFA

Viable prefix DFA states

- In the operation of the parser, at every point we are interested in what items could be useful in continuing the parse: call these the “alive” or “valid” items
- What are these items?
- Depends on where we are in the parse
- The history of the parse is summarized by the contents of the stack, and the current parser state
- Assuming we can keep the stack holding a viable prefix, we can identify the state of the parser with the set of items valid in that state

Valid items

- An item is valid if:
 - the stack contains a viable prefix, and
 - the symbols to the left of the dot on the item's RHS exactly match the symbols on the top of the stack, and
 - replacing only those symbols with the item's LHS would leave the stack still holding a viable prefix
- Intuitively, the valid items are ones that could be useful in continuing the parse from the current point
- We will identify states of the parser with sets of items, namely the ones that are valid in that state

What items are in the start state?

$E \rightarrow E + T \quad | \quad T$
 $T \rightarrow T * F \quad | \quad F$
 $F \rightarrow (E) \quad | \quad \mathbf{id}$

$E \rightarrow \bullet E + T$
 $E \rightarrow \bullet T$
 $T \rightarrow \bullet T * F$
 $T \rightarrow \bullet F$
 $F \rightarrow \bullet (E)$
 $F \rightarrow \bullet \mathbf{id}$

Closure() and goto()

- We will define a *closure()* function to enumerate all the items valid in each state
- Similar to the ϵ - closure function using in the DFA to NFA construction
- We will define a *goto()* function to identify transitions between states
- We begin by ..
 - Enumerating the items that should be in the start state
 - Describing the transitions out of the start state

Viabale prefix DFA start state

- The parser stack starts out empty
- Which items are valid?
- Every item of the form $S \rightarrow \bullet \gamma$, where S is the start symbol, is valid: matches an empty stack, and the lone start symbol is a viable prefix
- But if γ is of the form $A\alpha$, then ...
 - any item of the form $A \rightarrow \bullet \beta$ is also valid (matches with an empty stack)
 - A is a viable prefix
- How do we generate the rest of the items?

Finding all valid items in a DFA state

- In any state, $A \rightarrow \alpha \bullet B\beta$ must be valid if..
 - the stack currently contains a viable prefix
 - α is at the top of the stack
(symbol(s) to the left of the \bullet)
 - if we replaced α with A the stack would still contain a viable prefix
(only those symbols with the item's LHS)

Finding all valid items in a DFA state

- But if $A \rightarrow \alpha \bullet B \beta$ is valid then *any* item of the form $B \rightarrow \bullet \gamma$ is also valid
 - $B \rightarrow \bullet \gamma$ matches any TOS
 - if we placed B on top of α on the stack, we would also have a viable prefix
 - we could follow that B with β , reduce to A , and have the previous case

The *closure* function

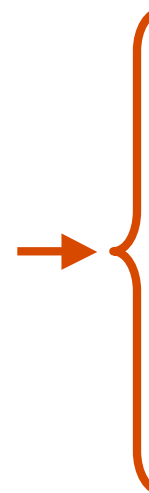
- The *closure* of a set of items in a state enables us to collect all the valid items in a state
- $C(I)$ of a set of items I is constructed as follows
 - Put all of I in $C(I)$
 - If $A \rightarrow \alpha \bullet B \beta$ is in $C(I)$ and $B \rightarrow \gamma$ is a rule, add $B \rightarrow \bullet \gamma$ to $C(I)$. Continue until no new items can be added
 - Repeat until no new items can be added

closure: an example

- Consider the simple expression grammar with start symbol E

$$\begin{array}{l} E \rightarrow E + T \quad | \quad T \\ T \rightarrow T * F \quad | \quad F \\ F \rightarrow (E) \quad | \quad \text{id} \end{array}$$

- Let I be the set $\{[E \rightarrow \bullet E + T], [E \rightarrow \bullet T]\}$
- Closure(I) is the set of items valid in the parser start state


$$\begin{array}{l} E \rightarrow \bullet E + T \\ E \rightarrow \bullet T \\ T \rightarrow \bullet T * F \\ T \rightarrow \bullet F \\ F \rightarrow \bullet (E) \\ F \rightarrow \bullet \text{id} \end{array}$$

How did this work?

- Initially, we include I
- The T after \bullet in $E \rightarrow \bullet T$ matches the LHS of the productions

$$T \rightarrow T * F \quad | \quad F$$

so we include both

- The F after \bullet in $T \rightarrow \bullet F$ matches the LHS of productions

$$F \rightarrow (E) \quad | \quad \text{id}$$

- (These are sets: they don't contain duplicates)

$$\begin{array}{l} E \rightarrow E + T \quad | \quad T \\ T \rightarrow T * F \quad | \quad F \\ F \rightarrow (E) \quad | \quad \text{id} \end{array}$$

$$\begin{array}{l} E \rightarrow \bullet E + T \\ E \rightarrow \bullet T \\ T \rightarrow \bullet T * F \\ T \rightarrow \bullet F \\ F \rightarrow \bullet (E) \\ F \rightarrow \bullet \text{id} \end{array}$$

closure in pseudocode

function *closure* (*I*) **begin**

J := *I*

repeat

for each item $A \rightarrow \alpha \cdot B \beta \in J$ and
each production $B \rightarrow \gamma \in G$
such that $B \rightarrow \cdot \gamma \notin J$

do **add** $B \rightarrow \cdot \gamma$ to *J* **end for**

until no more items can be added to *J*

return *J*

end *closure*

Transitions from the start state

- So we have determined what the start state of the DFA should be
- What should the transitions from this state be? What should the transitions be labeled with, and what states should they transition to?
- Keep in mind the important thing is to maintain a viable prefix on the stack, and to have states contain only valid items
- We want to do this while making progress in the parse, i.e. while shifting and reducing

Transitions when shifting

- Suppose a DFA state contains a valid item $A \rightarrow \alpha \bullet a \beta$
- And suppose a is the next token in the input
- We can make progress in the parse by shifting the terminal a onto the stack; this maintains a viable prefix there (why?)
- Clearly the item $A \rightarrow \alpha a \bullet \beta$ has become valid, and so should be in the new state we transition to (along with possibly other items)
- This transition is labeled with the terminal a

Transitions when reducing

- Suppose a DFA state contains a valid item
 $A \rightarrow \alpha \bullet B \beta$
- Here B is a nonterminal, so we can't put B on the stack just by shifting... but we can get it there by reducing
- If α was at the top of the stack holding a viable prefix in this state, and we return to this state after a reduction that placed B on top of α , the stack still holds a viable prefix (why?)
- Now clearly the item $A \rightarrow \alpha B \bullet \beta$ has become valid, and so should be in the new state we transition to (along with possibly other items)
- This transition is labeled with the nonterminal B

Closure again

- If we transition from a state containing item $A \rightarrow \alpha \cdot X \beta$ on a transition labeled X (where X is either terminal or nonterminal), we get to a state containing item $A \rightarrow \alpha X \cdot \beta$
- Clearly $A \rightarrow \alpha X \cdot \beta$ is valid in that new state, but we want the state to contain *all* items valid in the state
- We can find these items by taking the closure of the state
- We have arrived at the *goto* function which determines transitions among the DFA states

The *goto* function

- The $goto(I, X)$ of a set of items I with respect to a grammar symbol X is the *closure* of the set of items $A \rightarrow \alpha X \cdot B$ such that $A \rightarrow \alpha \cdot XB \in I$
- Intuitively, if I is the set of items that are valid given some viable prefix γ , then $goto(I, X)$ is the set of items valid given viable prefix γX
- The *goto* function lets us extend viable prefixes by one grammar symbol (either terminal or nonterminal)

goto: An example

- Consider the grammar

$$\begin{array}{l} E' \rightarrow E \\ E \rightarrow E + T \quad | \quad T \\ T \rightarrow T * F \quad | \quad F \\ F \rightarrow (E) \quad | \quad \text{id} \end{array}$$

- Let I be the set of items $\{ [E \rightarrow E \bullet + T] \}$
- $\text{goto}(I, +)$ is the closure of $\{ [E \rightarrow E + \bullet T] \}$
- The set of items:

$$\{ E \rightarrow E + \bullet T, \quad T \rightarrow \bullet T * F, \\ T \rightarrow \bullet F, \quad F \rightarrow \bullet (E), \quad F \rightarrow \bullet \text{id} \}$$

The canonical LR(0) collection

- Given a context free grammar, we can now construct the DFA states for recognizing the handles in the grammar
- Each state is a set of items valid in the state
- The transition function maintains the invariant that a viable prefix is always on the stack
- We use a construction called the ***Canonical LR(0) Collection***, which generates sets of items
- LR(0) because we will construct the states without the use of lookahead
- But, the action and goto tables will use lookahead

Canonical LR(0) construction

- Consider a grammar G with start symbol S
- We first augment G (to G') with a new start symbol S'
- Add a corresponding production
 $S' \rightarrow S$
- Ensures that the start production unique is, so we can tell when we are done with the parse
- Define a function called *collection()*, that constructs the canonical LR(0) collection of states for G'
- Need two helper functions: *closure()* and *goto()*

Constructing the canonical LR(0) collection

function *collection* (G') **begin**

$C := \{ \text{closure} (\{ [S' \rightarrow \cdot S] \}) \}$

repeat

for each set of items $I \in C$ and

 each grammar symbol X

 such that $\text{goto}(I, X) \neq \emptyset$ and $\notin C$

do **add** $\text{goto}(I, X)$ to C **end for**

until no more sets of items can be added to C

return C

end *collection*

Constructing the canonical LR(0) collection

- Consider the simple augmented expression grammar
- The canonical LR(0) collection of states is shown next

$$\begin{array}{l} E' \rightarrow E \\ E \rightarrow E + T \quad | \quad T \\ T \rightarrow T * F \quad | \quad F \\ F \rightarrow (E) \quad | \quad \mathbf{id} \end{array}$$

The canonical LR(0) collection

$I_0:$ $E' \rightarrow \cdot E$
 $E \rightarrow \cdot E + T$
 $E \rightarrow \cdot T$
 $T \rightarrow \cdot T * F$
 $T \rightarrow \cdot F$
 $F \rightarrow \cdot (E)$
 $F \rightarrow \cdot \mathbf{id}$

$I_5:$ $F \rightarrow \mathbf{id} \cdot$

$I_6:$ $E \rightarrow E + \cdot T$
 $T \rightarrow \cdot T * F$
 $T \rightarrow \cdot F$
 $F \rightarrow \cdot (E)$
 $F \rightarrow \cdot \mathbf{id}$

$I_1:$ $E' \rightarrow E \cdot$
 $E \rightarrow E \cdot + T$

$I_7:$ $T \rightarrow T * \cdot F$
 $F \rightarrow \cdot (E)$
 $F \rightarrow \cdot \mathbf{id}$

$I_2:$ $E \rightarrow T \cdot$
 $T \rightarrow T \cdot * F$

$I_8:$ $F \rightarrow (E \cdot)$
 $E \rightarrow E \cdot + T$

$I_3:$ $T \rightarrow F \cdot$

$I_9:$ $E \rightarrow E + T \cdot$
 $T \rightarrow T \cdot * F$

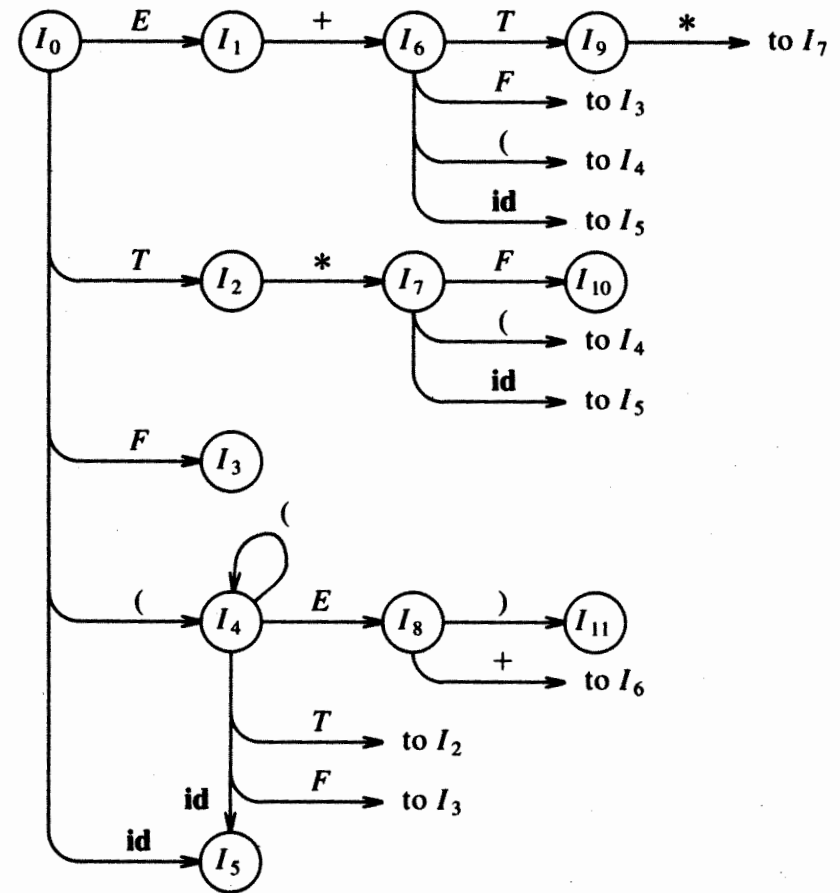
$I_4:$ $F \rightarrow (\cdot E)$
 $E \rightarrow \cdot E + T$
 $E \rightarrow \cdot T$
 $T \rightarrow \cdot T * F$
 $T \rightarrow \cdot F$
 $F \rightarrow \cdot (E)$
 $F \rightarrow \cdot \mathbf{id}$

$I_{10}:$ $T \rightarrow T * F \cdot$

$I_{11}:$ $F \rightarrow (E) \cdot$

The DFA

- Each set in the canonical collection corresponds to a state in the DFA
- Indices I_x designate sets
- $goto(I, X)$ is the state transition function from state I on symbol X
- This DFA recognizes viable prefixes (every state is an accepting state)
- Use CUP with `-dump_state:`
public/examples/expr



The link between *goto*, *closure* and *collection*

- Each NFA state corresponds to an individual item
- In that NFA there is a transition labeled X from a state of the form $A \rightarrow \alpha \bullet X \beta$, to the state $A \rightarrow \alpha X \bullet \beta$
- And there is an ϵ -transition from $A \rightarrow \alpha \bullet B \beta$ to $B \rightarrow \bullet \gamma$
- $\mathit{closure}(I)$ is the ϵ -closure of a set of NFA states (see §3.6)
- $\mathit{goto}(I, X)$ gives the transition from I on symbol X in the DFA built using subset construction
- Thus, $\mathit{collection}(G')$ is the subset construction applied to the viable-prefix recognizing NFA constructed from the augmented grammar G'

Finally: recognizing handles

- The canonical collection construction gives us a viable-prefix preserving DFA
- So long as we follow its state transitions the contents of the stack will be a viable prefix
- But how can we know if the current viable prefix ends in a handle?
- If the current state of the parser contains an item of the form $A \rightarrow \alpha \bullet$
- ... then it is an accepting state for handle α on TOS..
- ... to be reduced by the rule $A \rightarrow \alpha$
- If the grammar is SLR(0), that is enough to know we have a handle!
- But we can handle more grammars with lookahead

Followsets and lookahead

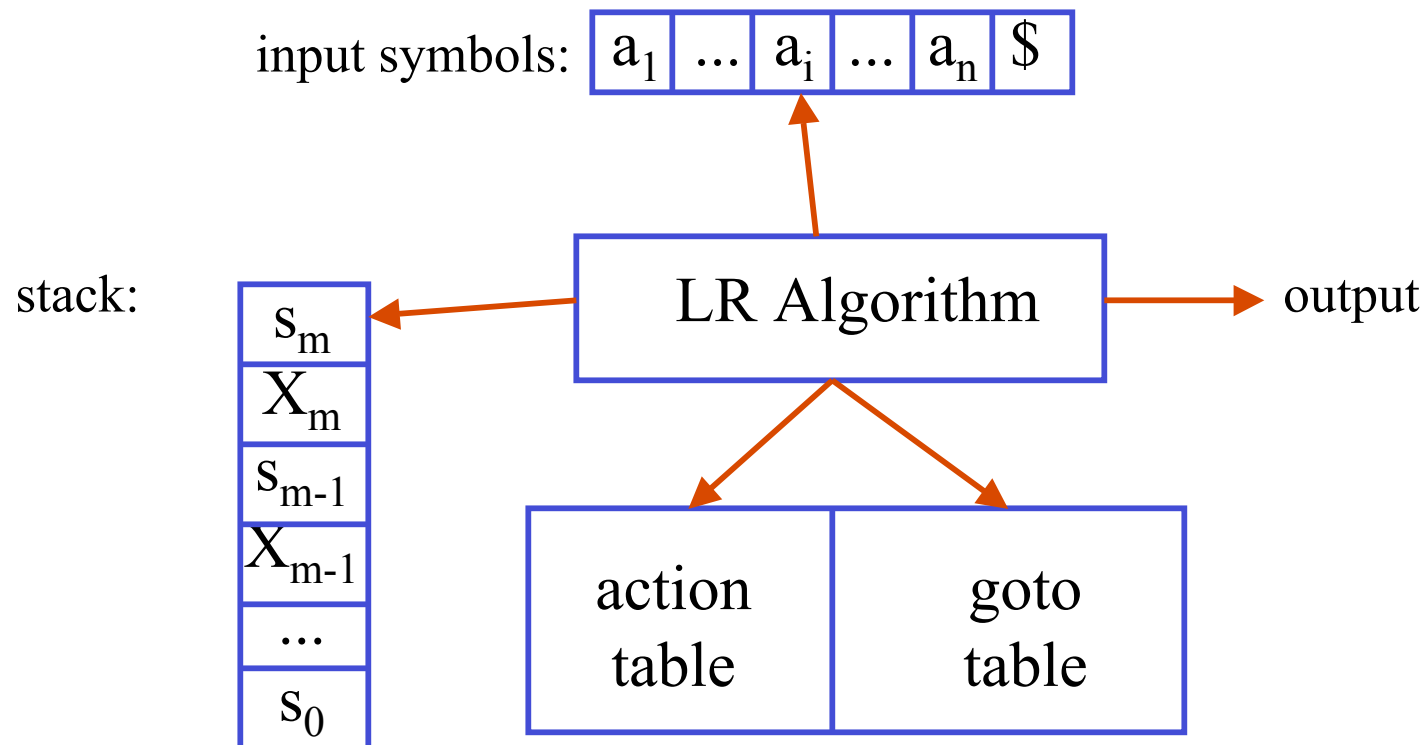
- If we know the followset of the nonterminals, then we can employ lookahead
- $follow_k(A)$ is the k -followset of a nonterminal symbol A
- It is the set of sequences of k terminal symbols that can immediately follow A in a right-sentential form
- Thus, if a state in the DFA has an item $A \rightarrow \alpha \bullet$, we will reduce according to that rule if the next k lookahead symbols appear in $follow_k(A)$
- Typically we will assume $k=1$, and just write $follow(A)$

SLR(1) parser table construction

- Now we know everything we need to construct the tables for an LR parser
- We will actually construct a Simple LR grammar with 1 token lookahead: SLR(1)
- CUP uses LALR(1) tables, and these are similar

Structure of a LR parser

- All LR parsers have this form; variants only differ in the content of the tables:



Recalling the parsing tables

- There are two parts to the table:
 - *action* maps [state, terminal symbol] \rightarrow action
 - *goto* maps [state, nonterminal symbol] \rightarrow state
- The tables implement the transition function of a DFA that recognizes handles on the stack so they can be reduced at the right time

The action table

- $action[s_m, a_i]$ tells the parser what action to take when state s_m is on top of stack and lookahead symbol is a_i
- Possible actions:
 - **shift** a new state onto the stack (along with the input symbol, consuming the input symbol)
 - **reduce** some symbols on the top of the stack by a particular rule; this exposes a state at the TOS; use goto table to decide new state to push (along with symbol on LHS of the rule)
 - **accept** the parse, or report an **error**

The goto table

- $goto[s_m, X_i]$ tells the parser what state to push on the stack when a reduction to symbol X_i exposes state s_m on the top of the stack
- The *goto* table is different from the *goto* function
 - the *goto* table maps state and nonterminal symbol to state
 - the *goto* function maps state and terminal *or* nonterminal symbol to state

Constructing SLR(1) tables for a grammar G

- Create augmented grammar G' , construct canonical LR(0) collection $C = \text{collection}(G') = \{I_0, \dots, I_n\}$
- For each item set I_i in C , construct the *action* table entry for parser state i and terminal symbol \mathbf{a} :
 - If $A \rightarrow \alpha \bullet \mathbf{a} \beta$ is in I_i and $\text{goto}(I_i, \mathbf{a}) = I_j$, then set $\text{action}[i, \mathbf{a}]$ to “**shift j** ”
 - If $A \rightarrow \alpha \bullet$ is in I_i and \mathbf{a} is in $\text{follow}(A)$, then set $\text{action}[i, \mathbf{a}]$ to “**reduce $A \rightarrow \alpha$** ”
 - If $S' \rightarrow S \bullet$ is in I_i , then set $\text{action}[i, \$]$ to “**accept**”
 - Else leave $\text{action}[i, \$]$ blank, meaning “**error**”

Continued....

- For each item set I_i in C , construct the goto table entry for parser state i and nonterminal A by:
 - If $goto(I_i, A) = I_j$, then set $goto[i, A]$ to j
- All table entries not filled in by the above procedure are marked **error**
- The parser start state is the state i constructed from the item set I_i that contains $S' \rightarrow S \bullet$

Shift-reduce Conflicts

- It is possible that some set I_i in the construction has two items $A \rightarrow \alpha \bullet a \beta$ and $B \rightarrow \gamma \bullet$ such that:
 - a is a terminal symbol and $A \rightarrow \alpha \bullet a \beta \in I_i$
 - a is a terminal symbol in $\text{follow}(B)$ and $B \rightarrow \gamma \bullet \in I_i$
- This is a *shift-reduce conflict* in state i over a
- But there can be only one table entry
- CUP resolves this in favor of a shift

Reduce-reduce Conflicts

- It is possible that some set I_i in the construction has two items $A \rightarrow \alpha \bullet$ and $B \rightarrow \beta \bullet$ such that:
 - \mathbf{a} is a terminal symbol in $follow(A)$ and $A \rightarrow \alpha \bullet \in I_i$
 - \mathbf{a} is a terminal symbol in $follow(B)$ and $B \rightarrow \beta \bullet \in I_i$
- This is a *reduce-reduce conflict* in state i over \mathbf{a}
- Again, we must resolve it
- CUP defaults to the first listed rule

A grammar not in SLR(1)

- Consider the following grammar

$$S \rightarrow L = R$$

$$S \rightarrow R$$

$$L \rightarrow \mathbf{id}$$

$$R \rightarrow L$$

- And one of the sets in the canonical LR(0) collection for the grammar

$$I_2: S \rightarrow L \bullet = R$$

$$R \rightarrow L \bullet$$

A shift/reduce conflict

- The first item sets action[2,=] to ‘shift 6’
- The second item sets action[2,=] to “reduce $R \rightarrow L$,” because $follow(R)$ contains =
- This is a *shift/reduce conflict*
- But this grammar is not ambiguous...!
- The problem arises because the SLR parser table construction does not carry enough information in its state
- LALR(k) tables are somewhat better
- But in general, every LR parser can have conflicts with unambiguous grammars

LR parsers and grammars

- If a grammar leads to a conflict when constructing a SLR(1) parser, that grammar is not SLR(1)
- Other LR techniques may be more general than SLR in that they can avoid some SLR conflicts
- LALR(1) grammars \supseteq SLR(1) grammars
- But no ambiguous grammar is LR(k) for any k, and some unambiguous grammars are also not LR(k) for any k
- But LR(k) parsers are in general more powerful than LL(k) (i.e., top-down) parsers...
- ...a LL(k) parser makes decisions based just on k lookahead symbols, but a LR(i) parser uses that plus its stack history
- But all conflicts must be dealt with no matter what the grammar

Putting it all together

- Let's build the SLR(1) table for the previous expression grammar

$$(1) E \rightarrow E + T$$

$$(2) E \rightarrow T$$

$$(3) T \rightarrow T * F$$

$$(4) T \rightarrow F$$

$$(5) F \rightarrow (E)$$

$$(6) F \rightarrow \text{id}$$

Building an SLR parse table

- Consider the start state set of items I_0 for the augmented grammar

- (0) $E' \rightarrow E$
- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow T * F$
- (4) $T \rightarrow F$
- (5) $F \rightarrow (E)$
- (6) $F \rightarrow \text{id}$

$$I_0: E' \rightarrow \bullet E$$
$$E \rightarrow \bullet E + T$$
$$E \rightarrow \bullet T$$
$$T \rightarrow \bullet T * F$$
$$T \rightarrow \bullet F$$
$$F \rightarrow \bullet (E)$$
$$F \rightarrow \bullet \text{id}$$

action table entry: shift

- Consider item $F \rightarrow \bullet (E) \in I_0$
- The applicable rule in the table construction algorithm for terminal a

If $A \rightarrow \alpha \bullet a \beta \in I_i$ and $goto(I_i, a) = I_j$,
then set $action[i, a]$ to “shift j”

- Thus, $i=0$, $a = (\quad \alpha = \varepsilon \quad \beta = E)$
- $goto(I_0, ()) = closure(\{ F \rightarrow \bullet (E) \})$

action table entry: shift

$$\begin{aligned}
 & \text{closure}(\{ F \rightarrow \bullet (E) \}) \\
 &= \{ F \rightarrow (\bullet E), E \rightarrow \bullet E + T, \\
 & \quad E \rightarrow \bullet T, \quad E' \rightarrow E \\
 & \quad T \rightarrow \bullet T * F, \quad E \rightarrow E + T \quad | \quad T \\
 & \quad T \rightarrow \bullet F, \quad T \rightarrow T * F \quad | \quad F \\
 & \quad F \rightarrow \bullet (E), \quad F \rightarrow (E) \quad | \quad \text{id} \\
 & \quad F \rightarrow \bullet \text{id}) \} = I_4
 \end{aligned}$$

Thus, $\text{action}[0, (] := \text{shift } 4$

$$\begin{aligned}
 I_4: & F \rightarrow (\bullet E) \\
 & E \rightarrow \bullet E + T \\
 & E \rightarrow \bullet T \\
 & T \rightarrow \bullet T * F \\
 & T \rightarrow \bullet F \\
 & F \rightarrow \bullet (E) \\
 & F \rightarrow \bullet \text{id}
 \end{aligned}$$

goto table entry

- Consider item $T \rightarrow \bullet F \in I_0$
- The applicable rule in the table construction algorithm:
 - For each nonterminal A , if $goto(I_i, A) = I_j$, then set $goto[i, A]$ to j .

- $goto(I_0, F) = closure(T \rightarrow F \bullet)$

$$= \{T \rightarrow F \bullet\} = I_3$$

$$I_3: T \rightarrow F \bullet$$

- Thus, $goto[0, F] = 3$

Another example

- Consider items $E' \rightarrow \bullet E$, $E \rightarrow \bullet E + T \in I_0$
- In the table construction algorithm, for nonterminal A :

If $goto(I_i, A) = I_j$, then set $goto[i, A]$ to j

- $goto(I_0, E) = closure(\{E' \rightarrow E \bullet, E \rightarrow E \bullet + T\})$
 $= \{E' \rightarrow E \bullet, E \rightarrow E \bullet + T\}$
 $= I_1$
- Thus, $goto[0, E]$ is set to 1

action table entry: accept

- Consider item $E' \rightarrow E \bullet \in I_1$
- The applicable rule in the table construction algorithm
 - If $S' \rightarrow S \bullet$ is in I_i , then $action[i, \$] := \text{“accept”}$
- We are about to reduce to the start symbol so we check if the input is exhausted
- Thus, $action[1, \$] := \text{“accept”}$.

action table entry: reduce

- Consider item $E \rightarrow T \bullet \in I_2$
- The applicable rule in the table construction algorithm
 If a is a terminal symbol $\in FOLLOW(A)$ & $A \rightarrow \alpha \bullet \in I_i$,
 then $action[i, a] := \text{“reduce } A \rightarrow \alpha \text{”}$
- $FOLLOW(A)$ is the set of terminal symbols that can immediately follow A in a right-sentential form
- $A = E$, $FOLLOW(E) = \{ \$, +,) \}$, $\alpha = T$

$I_2: E \rightarrow T \bullet$

$T \rightarrow T \bullet * F$

$E' \rightarrow E$

$E \rightarrow E + T \quad | \quad T$

$T \rightarrow T * F \quad | \quad F$

$F \rightarrow (E) \quad | \quad id$

action table entry: reduce

- ... if $A \rightarrow \alpha \bullet \in I_i$, $action[i, \mathbf{a}] := \text{“reduce } A \rightarrow \alpha \text{”}$
- $A = E$, $FOLLOW(E) = \{ \$, +,) \}$, $i=2$, $\alpha = T$
- Thus, $action[2, \$] = action[2, +] = action[2,)]$
reduce $E \rightarrow T$

$$\begin{array}{l} E' \rightarrow E \\ E \rightarrow E + T \quad | \quad T \\ T \rightarrow T * F \quad | \quad F \\ F \rightarrow (E) \quad | \quad id \end{array}$$

action table entry: reduce

- Consider item $T \rightarrow F \bullet \in I_3$
- The applicable rule in the table construction algorithm
If \mathbf{a} is a terminal symbol $\in FOLLOW(A)$ &
 $A \rightarrow \alpha \bullet \in I_i$,
then set $action[i, \mathbf{a}]$ to “**reduce $A \rightarrow \alpha$** ”
- $FOLLOW(A)$ is the set of terminal symbols that can immediately follow A in a right-sentential form

action table entry: reduce

- if $\mathbf{a} \in FOLLOW(T)$ & $A \rightarrow \alpha \bullet \in I_i$,
 $action[i, \mathbf{a}] := \text{“reduce } A \rightarrow \alpha \text{”}$
- $A = T$ $FOLLOW(T) = \{ +, *,), \$ \}$, $i=3$, $\alpha = F$
- Thus, $action[3, +] = action[3, *] = action[3,)] =$
 $action[3, \$] = \text{reduce by } T \rightarrow F, \text{ i.e. rule 4}$

$$\begin{array}{l} E' \rightarrow E \\ E \rightarrow E + T \quad | \quad T \\ T \rightarrow T * F \quad | \quad F \\ F \rightarrow (E) \quad | \quad \text{id} \end{array}$$

The completed SLR parse table

- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow T * F$
- (4) $T \rightarrow F$
- (5) $F \rightarrow (E)$
- (6) $F \rightarrow \text{id}$

s_i	<i>action</i>						<i>goto</i>		
	id	+	*	()	\$	<i>E</i>	<i>T</i>	<i>F</i>
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			