Building LR Tables

How to construct the ACTION and GOTO tables?

• Define “items”
• Define “viable prefix”
• Define the “closure function”
  Set-of-items → Set-of-items
• Define the GOTO function
• Work with a set of sets of items
  A collection of sets of items
  CC = Canonical Collection of LR items
• Describe how to construct CC
• Given all this, describe how to construct the tables

LR(0) Items

Given: A grammar, G
Items look like productions
... augmented with a dot in the righthand side.

Grammar:
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow ( E )$
6. $F \rightarrow id$

The Items:
$E \rightarrow \cdot E + T$
$E \rightarrow E + T$ $T \rightarrow \cdot F$
$E \rightarrow E + T$ $T \rightarrow F$
$E \rightarrow E + T$ $E \rightarrow \cdot T$
$E \rightarrow \cdot T$ $F \rightarrow \cdot ( E )$
$E \rightarrow \cdot T$ $E \rightarrow \cdot T$
$E \rightarrow \cdot T$ $F \rightarrow ( \cdot E )$
$E \rightarrow \cdot T$ $E \rightarrow \cdot T$
$E \rightarrow \cdot T$ $F \rightarrow ( E \cdot )$
$E \rightarrow \cdot T$ $E \rightarrow \cdot T$
$E \rightarrow \cdot T$ $F \rightarrow \cdot ( E \cdot )$
$T \rightarrow \cdot T * F$ $F \rightarrow \cdot id$
$T \rightarrow \cdot T * F$ $T \rightarrow id$
$T \rightarrow \cdot T * F$ $T \rightarrow id$
$T \rightarrow \cdot T F$ $T \rightarrow T$
LR(1) Items

Just like before, except...

• Look-ahead symbol
• Terminal symbol from grammar

Grammar:

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

Examples:

- \( E \rightarrow \bullet E + T , ) \)
- \( E \rightarrow \bullet E + T , \$ \)
- \( E \rightarrow E \ast + T , ) \)
- \( E \rightarrow E \ast + T , \$ \)
- ...
**Intuition behind LR(1) Items**

F → • ( E ), )

It would be legal at this point in the parse to see an F, followed by a ).

Using rule 5, one way to find an F is to find ( E ) next.
So, among other possibilities, we are looking for ( E ), followed by a )).
If a ( comes next, then let’s scan it and keep going, looking for E ), followed by a ).
If we get E ) later, then we will be able to reduce it to F ...
... but we may get something different (although perfectly legal).

E → • T, )

It would be legal at this point in the parse to see an E, followed by a ).

Using rule 2, one way to find an E is to find T next.
So, among other possibilities, we are looking for a T followed by a ).
And how can we find a T followed by a )? 

T → • T * F, )
T → • F, )

**Step 1**

Augment the grammar by adding...

- A new start symbol, S'
- A new rule S' → S

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

Our goal is to find an S', followed by $.

S' → • E, $

Whenever we are about to reduce using rule 0...
Accept! Parse is finished!
The CLOSURE Function

Let’s say we have this item:

\[ E \rightarrow \cdot T, ) \]

What are the ways to find a \( T \)?

\[ T \rightarrow F \]
\[ T \rightarrow T \ast F \]

We are looking for a \( T \), followed by a \( ) \), so we’ll need to add these items:

\[ T \rightarrow \cdot F, ) \]
\[ T \rightarrow \cdot T \ast F, ) \]

We can find a \( T \) followed by a \( ) \) if we find an \( F \) followed by a \( ) \).

How can we find that?

\[ F \rightarrow \ast ( \ E \ ), ) \]
\[ F \rightarrow \ast \text{id}, ) \]

We can also find a \( T \) followed by a \( ) \) if we find an \( T \ast F \) followed by a \( ) \).

To find that, we need to first find another \( T \), but followed by \( \ast \).

\[ T \rightarrow \cdot F, \ast \]
\[ T \rightarrow \cdot T \ast F, \ast \]

So we should also look for a \( F \) followed by a \( \ast \).

\[ F \rightarrow \ast ( \ E \ ), \ast \]
\[ F \rightarrow \ast \text{id}, \ast \]
CLOSURE Function Example

Example: Let $I_1 = \{ \begin{align*}
E &\rightarrow E \cdot + T, \\
T &\rightarrow T \cdot * F, \\
F &\rightarrow \text{id} \cdot , \\
F &\rightarrow (E) \cdot ,
\end{align*} \}$

Compute: CLOSURE $(I_1) = \{ \begin{align*}
0. S' &\rightarrow E \\
1. E &\rightarrow E + T \\
2. E &\rightarrow T \\
3. T &\rightarrow T \cdot F \\
4. T &\rightarrow F \\
5. F &\rightarrow (E) \\
6. F &\rightarrow \text{id}
\end{align*} \}$
Example: Let $I_1 = \{ \ E \rightarrow E \cdot + T, 
\ T \rightarrow T \cdot * F, 
\ F \rightarrow \text{id} \cdot, 
\ F \rightarrow ( E ) \cdot, \}$

Compute: CLOSURE ($I_1$) = 
Start by adding all items in $I_1$...

\[
E \rightarrow E \cdot + T, \\
T \rightarrow T \cdot * F, \\
F \rightarrow \text{id} \cdot, \\
F \rightarrow ( E ) \cdot,
\]

Is the dot in front of a non-terminal? No... no more items are added.

\[
0. S' \rightarrow E \\
1. E \rightarrow E + T \\
2. E \rightarrow T \\
3. T \rightarrow T * F \\
4. T \rightarrow F \\
5. F \rightarrow ( E ) \\
6. F \rightarrow \text{id}
\]
CLOSURE Function Example

Example: Let $I_2 = \{ (T \rightarrow \cdot F, ) \quad T \rightarrow \cdot T \cdot F, ) \}$

Compute: CLOSURE $(I_2)$ = {
Start by adding...

(1) $T \rightarrow \cdot F, )$
(2) $T \rightarrow \cdot T \cdot F, )$

Look at (1) first...
CLOSURE Function Example

Example: Let $I_2 = \{ T \rightarrow \cdot F, \} \cup \{ T \rightarrow \cdot T \cdot F, \}$

Compute: CLOSURE ($I_2$) = 

Start by adding all items in $I$...

(1) $T \rightarrow \cdot F, \$
(2) $T \rightarrow \cdot T \cdot F, \$

Look at (1) first. Look at each $F$ rule. For every $b$ in FIRST ($\varepsilon$) = \{ \}...
**CLOSURE Function Example**

**Example:**  Let \( I_2 = \{ \ T \rightarrow \ast \ F, \ \)  
\( T \rightarrow \ast T \ast F, \ \) \}

**Compute:**  CLOSURE ( \( I_2 \)) = \{

Start by adding all items in \( I_2 \)... 

1. \( T \rightarrow \ast F, \)  
2. \( T \rightarrow \ast T \ast F, \)  

Look at (1) first. Look at each \( F \) rule. For every \( b \) in FIRST (\( \epsilon \)) = \{ \}... 

3. \( F \rightarrow \ast (E), \)  
4. \( F \rightarrow \ast \text{id}, \)  

Look at (2) next. Look at each \( T \) rule. For every \( b \) in FIRST (\( \ast F \)) = \{ \ast \}... 

5. \( T \rightarrow \ast F, \ast \)  
6. \( T \rightarrow \ast T \ast F, \ast \)  

Look at (3) and (4) next...

0. \( S' \rightarrow E \)  
1. \( E \rightarrow E + T \)  
2. \( E \rightarrow T \)  
3. \( T \rightarrow T \ast F \)  
4. \( T \rightarrow F \)  
5. \( F \rightarrow (E) \)  
6. \( F \rightarrow \text{id} \)
CLOSURE Function Example

Example: Let $I_2 = \{ T \rightarrow \cdot F, \) 
        T \rightarrow \cdot T \cdot F, \) \}

Compute: \( \text{CLOSURE} ( I_2 ) = \{
\)

Start by adding all items in $I$...

(1) $T \rightarrow \cdot F, )$
(2) $T \rightarrow \cdot T \cdot F, )$

Look at (1) first. Look at each $F$ rule. For every $b$ in $\text{FIRST} (ε) = \{ \} ...$
(3) $F \rightarrow \cdot ( E ), )$
(4) $F \rightarrow \cdot \text{id} , )$

Look at (2) next. Look at each $T$ rule. For every $b$ in $\text{FIRST} (\cdot F) = \{ \cdot \} ...$
(5) $T \rightarrow \cdot F, \cdot$
(6) $T \rightarrow \cdot T \cdot F, \cdot$

Look at (3) and (4) next. The dot is not in front of a non-terminal.

Look at (5) next. Look at each $F$ rule. For every $b$ in $\text{FIRST} (ε \cdot F) = \{ ε \cdot \} ...$
(7) $F \rightarrow \cdot ( E ), \cdot$
(8) $F \rightarrow \cdot \text{id} , \cdot$

Look at (6) next...

We already added (5) and (6)

Look at (7) and (8) next...
CLOSURE Function Example

Example: Let $I_2 = \{ T \rightarrow \cdot F, \\
T \rightarrow \cdot T \cdot F, \} \}

Compute: CLOSURE ($I_2$) =

Start by adding all items in $I$...

1. $T \rightarrow \cdot F,$
2. $T \rightarrow \cdot T \cdot F,$

Look at (1) first. Look at each $F$ rule. For every $b$ in $\text{FIRST}(b) = \{\}$....

3. $F \rightarrow \cdot (E),$
4. $F \rightarrow \cdot \text{id},$

Look at (2) next. Look at each $T$ rule. For every $b$ in $\text{FIRST}(T) = \{\}$....

5. $T \rightarrow \cdot F,$
6. $T \rightarrow \cdot T \cdot F,$

Look at (3) and (4) next. The dot is not in front of a non-terminal.

Look at (5) next. Look at each $F$ rule. For every $b$ in $\text{FIRST}(F) = \{\}$....

7. $F \rightarrow \cdot (E),$
8. $F \rightarrow \cdot \text{id},$

Look at (6) next. Look at each $T$ rule. For every $b$ in $\text{FIRST}(T) = \{\}$....

We already added (5) and (6).

Look at (7) and (8) next. The dot is not in front of a non-terminal.

}
The GOTO Function

Let I be a set of items...
Let X be a grammar symbol (terminal or non-terminal)...

\begin{verbatim}
function GOTO(I,X) returns a set of items
    result = {}
    look at all items in I...
        if A → α•X δ, a is in I
            then add A → αX•δ, a to result
    result = CLOSURE(result)
\end{verbatim}

In other words, move the dot past the X in any items where it is in front of an X

...and take the CLOSURE of whatever items you get

Intuition:

• I is a set of items indicating where we are so far, after seeing some prefix γ of the input.
• I describes what we might legally see next.
• Assume we get an X next.
• Now we have seen some prefix γX of the input.
• GOTO(I, X) tells what we could legally see after that.
• GOTO(I, X) is the set of all items that are “valid” for prefix γX.
**GOTO Function Example**

**Example:** Let $I_4 = \{ E \rightarrow T \cdot , \) \\
$ T \rightarrow T \ast F , ) \}$

**Compute:** GOTO ($I_4, \ast$) = \{

Is the $\cdot$ in front of $\ast$ in any of the items?

$T \rightarrow T \ast F , )$

Now take the closure...

$F \rightarrow \cdot ( E ) , )$
$F \rightarrow \cdot \text{id} , )$

**Intuition:**

We found a $T$ and then we found a $\cdot$. What are we looking for next?

$E \rightarrow T \cdot , )$

Means: We are now looking for an $F$ followed by $)$

$F \rightarrow \cdot ( E ) , )$

Means: We could find that by finding $( E )$ followed by $)$

$F \rightarrow \cdot \text{id} , )$

Means: We could find that by finding $( E )$ followed by $)$

---

**GOTO Function Example**

**Example:** Let $I_5 = \{ E \rightarrow T \cdot , ) \\
T \rightarrow \cdot T \ast F , ) \}$

**Compute:** GOTO ($I_5, T$) = \{

Is the $\cdot$ in front of $T$ in any of the items?

$E \rightarrow T \cdot , )$
$T \rightarrow T \ast F , )$

Now take the closure...

$E \rightarrow T \cdot , )$

Means: We are now looking for $)$

$T \rightarrow T \ast F , )$

Means: We are now looking for $\ast F$ followed by $)$

---

© Harry H. Porter, 2005
Constructing the Canonical Collection

Each $CC_i$ will be a set of items.
We will build up a collection of these.

$CC = \text{"The Canonical Collection of } LR(1) \text{ items"} = \text{a set of sets of items} = \{ CC_0, CC_1, CC_2, CC_3, \ldots CC_N \}$

Algorithm to construct $CC$, the Canonical Collection of $LR(1)$ Items:

1. let $CC_0 = \text{CLOSURE}( \{ S' \rightarrow \cdot S, \cdot \} )$
2. add $CC_0$ to $CC$
3. repeat
   4. let $CC_i$ be some set of items already in $CC$
   5. for each $X$ (that follows a $\cdot$ in some item in $CC_i$)…
      6. compute $CC_k = \text{GOTO}(CC_i, X)$
      7. if $CC_k$ is not already in $CC$ then
         8. add it
         9. set $\text{MOVE}(CC_i, X) = CC_k$
      10. endIf
   11. endFor
4. until nothing more can be added to $CC$
Example:

\[ CC_0 = CLOSURE( \{ S' \rightarrow \cdot E, \} ) \]

\[ = \{ S' \rightarrow \cdot E, \}
E \rightarrow \cdot E + T, 
E \rightarrow \cdot E + T, +
E \rightarrow \cdot T, 
E \rightarrow \cdot T, +
T \rightarrow \cdot T \cdot F, 
T \rightarrow \cdot T \cdot F, +
T \rightarrow \cdot T \cdot F, *
T \rightarrow \cdot F, 
T \rightarrow \cdot F, *
F \rightarrow \cdot ( E ), 
F \rightarrow \cdot ( E ), +
F \rightarrow \cdot ( E ), *
F \rightarrow \cdot id, 
F \rightarrow \cdot id, +
F \rightarrow \cdot id, * \} \]
GOTO (CC₀ E) = CC₁

Advance • past E in the items containing • E

CC₁ = {
  S' → E • , $
  E → E • + T , $
  E → E • + T , +
}

And take the closure... (Nothing more added.)

Intuition:
We will reduce by S' → E if the next symbol is $.
Otherwise, we will look for a + next.

The • is in front of +. We’ll come back to CC₁ later.

GOTO (CC₀ T) = CC₂

Advance • past T

CC₂ = {
  E → T • , $
  E → T • , +
  T → T • • F , $
  T → T • • F , +
  T → T • • F , *
}

And take the closure...

Intuition:
We will reduce by T → F if the next symbol is $ or +.
Otherwise, we will look for *.

The • is in front of *. We’ll come back to CC₂ later.
GOTO (CC₀, F) = CC₃

Advance • past F
CC₃ = {
  T → F•, $
  T → F•, +
  T → F•, *
}

And take the closure...

Intuition:
We will reduce by T → F if the next symbol is $, +, or *.

The • is not in front of any symbol; no further “GOTO”s.

GOTO (CC₀, id) = CC₅

Advance • past id
CC₅ = {
  F → id•, $
  F → id•, +
  F → id•, *
}

And take the closure...

Intuition:
We will reduce after seeing an id, if the next symbol is +, *, or $.

The • is not in front of any symbol; no further “GOTO”s.
Syntax Analysis - Part 3

GOTO (CC_0, () = CC_4

Advance past (E)
CC_4 = {F → (• E ), $F → (• E ), +F → (• E ), *}

And take the closure...
E → • E + T , )E → • E + T , +E → • T , )E → • T , +T → • T * F , )T → • T * F , +T → • T * F , *T → • F , )T → • F , +T → • F , *F → • ( E ), )F → • ( E ), +F → • ( E ), *F → • id , )F → • id , +F → • id , *}

The • is before E, T, F, (, and id...
Next, we’ll compute...
GOTO (CC_4, E) ⇒ CC_8GOTO (CC_4, T) ⇒ CC_9GOTO (CC_4, F) ⇒ CC_10GOTO (CC_4, ( ) ⇒ CC_11GOTO (CC_4, id ) ⇒ CC_12

© Harry H. Porter, 2005
Syntax Analysis - Part 3

GOTO (CC_{1^*} + ) = CC_6

CC_1 = {  
  S' \rightarrow E^{*}, $  
  E \rightarrow E^{*} + T, $  
  E \rightarrow E^{*} + T, +  
}

Advance \ast past + in the items containing \ast +

CC_6 = {  
  E \rightarrow E^{*} + T, $  
  E \rightarrow E^{*} + T, +  
}

And take the closure...

T \rightarrow \ast T \ast F, $  
T \rightarrow \ast T \ast F, +  
T \rightarrow \ast T \ast F, *  
T \rightarrow \ast F, $  
T \rightarrow \ast F, +  
T \rightarrow \ast F, *  
F \rightarrow \ast ( E ), $  
F \rightarrow \ast ( E ), +  
F \rightarrow \ast ( E ), *  
F \rightarrow \ast id, $  
F \rightarrow \ast id, +  
F \rightarrow \ast id, *  
}

Intuition:
We have found T \ast .

Next, look for a F followed by $, +, or *.
\textbf{GOTO (CC_4, E) = CC_8}

\begin{align*}
CC_8 &= \{ \\
F &\rightarrow (E \cdot), \$
F &\rightarrow (E \cdot), + \\
F &\rightarrow (E \cdot), * \\
E &\rightarrow E \cdot + T, ) \\
E &\rightarrow E \cdot + T, + 
\}
\end{align*}

And take the closure...

\textbf{GOTO (CC_4, T) = CC_9}

\begin{align*}
CC_9 &= \{ \\
E &\rightarrow T \cdot, ) \\
E &\rightarrow T \cdot, + \\
T &\rightarrow T \cdot \ast F, ) \\
T &\rightarrow T \cdot \ast F, + \\
T &\rightarrow T \cdot \ast F, * 
\}
\end{align*}

And take the closure...
GOTO (CC_{4}, F) = CC_{10}

CC_{10} = \{
    T \rightarrow F^*, 
    T \rightarrow F^*, +
    T \rightarrow F^*, *
\}
And take the closure...

GOTO (CC_{4}, \text{id}) = CC_{12}

CC_{12} = \{
    F \rightarrow \text{id}^*, 
    F \rightarrow \text{id}^*, +
    F \rightarrow \text{id}^*, *
\}
And take the closure...
Syntax Analysis - Part 3

\[ \text{GOTO (CC}_{11}, () = \text{CC}_{11} \]

\[ \text{CC}_{11} = \{ \]
\[ \quad F \rightarrow ( \cdot E ), \]
\[ \quad F \rightarrow ( \cdot E ), + \]
\[ \quad F \rightarrow ( \cdot E ), * \]
\[ \}

And take the closure...

\[ E \rightarrow \cdot E + T, \]
\[ E \rightarrow \cdot E + T, + \]
\[ E \rightarrow \cdot T, \]
\[ E \rightarrow \cdot T, + \]
\[ T \rightarrow \cdot T * F, \]
\[ T \rightarrow \cdot T * F, + \]
\[ T \rightarrow \cdot T * F, * \]
\[ T \rightarrow \cdot F, \]
\[ T \rightarrow \cdot F, + \]
\[ T \rightarrow \cdot F, * \]
\[ F \rightarrow \cdot ( E ), \]
\[ F \rightarrow \cdot ( E ), + \]
\[ F \rightarrow \cdot ( E ), * \]
\[ F \rightarrow \cdot \text{id}, \]
\[ F \rightarrow \cdot \text{id}, + \]
\[ F \rightarrow \cdot \text{id}, * \]

\[ \text{CC}_{11} \text{ is similar to, but not quite the same as CC}_0 \]

The \( \cdot \) is before \( E, T, F, \text{id} \), and \( ( \ldots \)

Next, we'll compute...

\[ \text{GOTO (CC}_{11}, E) \Rightarrow \text{CC}_{18} \]
\[ \text{GOTO (CC}_{11}, T) \Rightarrow \text{CC}_9 \]
\[ \text{GOTO (CC}_{11}, F) \Rightarrow \text{CC}_{10} \]
\[ \text{GOTO (CC}_{11}, \text{id}) \Rightarrow \text{CC}_{12} \]
\[ \text{GOTO (CC}_{11}, ( \ldots \Rightarrow \text{CC}_{11} \]

Syntax Analysis - Part 3

\[ \text{GOTO (CC}_{18}, E) = \text{CC}_{18} \]

\[ \text{CC}_{18} = \{ \]
\[ \quad F \rightarrow ( E * ), \]
\[ \quad F \rightarrow ( E * ), + \]
\[ \quad F \rightarrow ( E * ), * \]
\[ \quad E \rightarrow E * + T, \]
\[ \quad E \rightarrow E * + T, + \]
\[ \}

And take the closure...

\[ \]

© Harry H. Porter, 2005
GOTO \( CC_{11, T} = CC_9 \)

\[
CC_9 = \{
\begin{align*}
E & \rightarrow T \cdot, \\
E & \rightarrow T \cdot, + \\
T & \rightarrow T \cdot F, ) \\
T & \rightarrow T \cdot F, + \\
T & \rightarrow T \cdot F, *
\end{align*}
\]

And take the closure...

\}

We have seen \( CC_9 \) before!

---

GOTO \( CC_{11, F} = CC_{10} \)

\[
CC_{10} = \{
\begin{align*}
T & \rightarrow F \cdot, \\
T & \rightarrow F \cdot, + \\
T & \rightarrow F \cdot, *
\end{align*}
\]

And take the closure...

\}

We have seen \( CC_{10} \) before!
Viable Prefixes

Consider a right-sentential form

\[ S \Rightarrow_{RM}^{*} XXA \Rightarrow_{RM}^{*} XXBCDE \Rightarrow_{RM}^{*} \ldots \]

Rule:

\[ A \rightarrow BCDE \]

A viable prefix is a prefix of a right sentential form that does not extend to the right end of the handle.

\[ XXBCDE \]

A Viable prefix
Viable Prefixes

Consider a right-sentential form

\[
S \Rightarrow_{RM}^{*} XXAfff \Rightarrow_{RM} XXBCDEfff \Rightarrow_{RM} \ldots
\]

Rule:

\[A \rightarrow BCDE\]

A viable prefix is a prefix of a right-sentential form that does not extend to the right end of the handle.

\[XXBCDEfff\]

A Viable prefix

Given a viable prefix, we can always add terminals to get a right-sentential form!

Why?

Assume that \[XXBC\] is a viable prefix that we’ve shifted onto the stack.
Assume that we have some more terminals \[dddeeefff\] in the input.

If this string is legal, there must be rules that allow

\[D \Rightarrow^{*} ddd\] and \[E \Rightarrow^{*} eee\]

\[
S \ldots \Rightarrow^{*}_{RM} XXBCDEfff \Rightarrow^{*}_{RM} XXBCDeeef \Rightarrow^{*}_{RM} XXBCdddeeefff
\]

As long as we have a viable prefix, just keep shifting!

The Main Idea of LR Parsing

As long as what is on the stack is a viable prefix...

- The unseen terminals might be what is required to make
  \[STACK \parallel REMAINING-INPUT\]
  into a right-sentential form.
- We are on course to finding a rightmost derivation.

The key ideas of LR parsing:

Construct a DFA to recognize viable prefixes!

Every path in the DFA (from start to any final or non-final state) describes a viable prefix.

Each state is a set of items.
If the DFA has an edge from the current state labeled with a terminal And the edge label = the lookahead symbol Do a shift: Add this terminal to the viable prefix.
When the dot is at the end of one of the items in a state... If the next symbol = the lookahead symbol... Do a reduction. \[A \rightarrow XYZ\]
Example
Consider this viable prefix. Trace through the DFA.
( T * ( ( E ) End up in state 21, a final state.
If next token is +, *, or ) then reduce by F → ( E )

The stack shows our path through the DFA. “How we got to state 21”
Example
Consider this viable prefix. Trace through the DFA.
\( T \ast ( ( E ) \) Ends up in state 21, a final state.
If next token is +, *, or ) then reduce by \( F \rightarrow ( E ) \)

Back up along our path... to where we were before we saw (E).
Then take the F edge to get to state 10.
Example
Consider this viable prefix. Trace throughout the DFA.
(T * ( ( E ) Ends up in state 21, a final state.
   If next token is +, *, or ) then reduce by F→(E)
(T * ( F Ends up in state 10, a final state.
   If next token is +, *, or ) then reduce by T→F

Back up along our path... to where we were before we saw F.
Then take the T edge to get to state 9.
Example

Consider this viable prefix. Trace through the DFA.

( T * ( ( E ) ) Ends up in state 21, a final state.
  If next token is +, *, or ) then reduce by F → (E)
( T * ( F ) Ends up in state 10, a final state.
  If next token is +, *, or ) then reduce by T → F
( T * ( T ) Ends up in state 9, a final state.
  If next token is +, or ) then reduce by E → T
  Assume next token is *... Need to shift and goto state 17

Other Examples

Here are some viable prefixes. Trace through the DFA for each of these!

( E + T * F
  Goes to state 20, a final state.
  If next token is $, +, or * then reduce by T → T * F
E + ( ( T * id
  Goes to state 12, a final state.
  If next token is +, *, or ) then reduce by F → id
( E + ( E + ( E + ( E * ( T * ( T * ( T * ( T *
  Goes to state 17, not a final state.
  We must get F, (, or id next
( E + ( T
  Goes to state 9
  If next token is ( or + then reduce by E → T
  Else okay to see *
Algorithm to Construct ACTION and GOTO Tables

Input: Grammar G, augmented with $S' \rightarrow S$

Output: ACTION and GOTO Tables

Construct “Canonical Collection of LR(1) items” along with MOVE function.

$\text{CC} = \{ CC_0, CC_1, CC_2, CC_3, \ldots CC_N \}$

There will be $N$ states, one per set of items $\{ 0, 1, 2, 3, \ldots N \}$

for each $CC_i$ do
  for each item in $CC_i$ do
    if the item has the form $A \rightarrow \beta \cdot c \gamma, a$
      and $\text{MOVE}(CC_i, c) = CC_j$ then
      set $\text{ACTION}[i, c]$ to “Shift j”
    elseif the item has the form $A \rightarrow \beta \cdot a$ then
      set $\text{ACTION}[i, a]$ to “Reduce $A \rightarrow \beta$”
    elseif the item has the form $S' \rightarrow S \cdot, \$ then
      set $\text{ACTION}[i, \$]$ to “Accept”
  endFor
  for each nonterminal $A$ do
    if $\text{MOVE}(CC_i, A) = CC_j$ then
      set $\text{GOTO}[i, A]$ to $j$
    endIf
  endFor
endFor

The SLR Table Construction Algorithm

With SLR, we do not have the lookahead symbol.

<table>
<thead>
<tr>
<th>LR(1) items:</th>
<th>LR(0) items:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F \rightarrow (\cdot E),$</td>
<td>$F \rightarrow (\cdot E)$</td>
</tr>
<tr>
<td>$F \rightarrow (\cdot E),$ $+$</td>
<td></td>
</tr>
<tr>
<td>$F \rightarrow (\cdot E),$ $*$</td>
<td></td>
</tr>
</tbody>
</table>

Some information is lost.
Some states in $\text{CC}$ collapse into one state.
There are fewer states in $\text{CC}'$
  $\Rightarrow$ Fewer rows in the resulting tables.
With SLR, these combine into one state.

The SLR Table Construction Algorithm

The CLOSURE function is basically the same, but simpler.
The GOTO function is basically the same, but simpler.
The Construction of the Canonical Collection is the same.
The Construction of the ACTION and GOTO tables is a little different.

\[
\text{CC}_4 = \{ F \rightarrow ( \cdot E ), $ \\
F \rightarrow ( \cdot E ), + \\
F \rightarrow ( \cdot E ), * \\
E \rightarrow ( \cdot E + T ) \\
E \rightarrow ( \cdot E + T ), + \\
E \rightarrow ( \cdot E + T ), * \\
E \rightarrow ( \cdot E + T ) \\
E \rightarrow ( \cdot T ) \\
E \rightarrow ( \cdot T ), + \\
T \rightarrow ( \cdot T * F ) \\
T \rightarrow ( \cdot T * F ), + \\
T \rightarrow ( \cdot T * F ), * \\
T \rightarrow ( \cdot T * F ) \\
T \rightarrow ( \cdot F ) \\
T \rightarrow ( \cdot F ), + \\
T \rightarrow ( \cdot F ) \\
F \rightarrow ( \cdot E ) \\
F \rightarrow ( \cdot E ) \\
F \rightarrow ( \cdot E ) \\
F \rightarrow ( \cdot id ) \\
F \rightarrow ( \cdot id ) \\
F \rightarrow ( \cdot id ), * \\
\}
\]

\[
\text{CC}_{11} = \{ F \rightarrow ( \cdot E ) \\
F \rightarrow ( \cdot E ) \\
F \rightarrow ( \cdot E ) \\
F \rightarrow ( \cdot E ) \\
E \rightarrow ( \cdot E + T ) \\
E \rightarrow ( \cdot E + T ) \\
E \rightarrow ( \cdot E + T ), + \\
E \rightarrow ( \cdot E + T ), * \\
T \rightarrow ( \cdot T ) \\
T \rightarrow ( \cdot T ), + \\
T \rightarrow ( \cdot T * F ) \\
T \rightarrow ( \cdot T * F ) \\
T \rightarrow ( \cdot T * F ), + \\
T \rightarrow ( \cdot T * F ), * \\
T \rightarrow ( \cdot T * F ) \\
T \rightarrow ( \cdot F ) \\
T \rightarrow ( \cdot F ), + \\
T \rightarrow ( \cdot F ) \\
F \rightarrow ( \cdot E ) \\
F \rightarrow ( \cdot E ) \\
F \rightarrow ( \cdot E ) \\
F \rightarrow ( \cdot id ) \\
F \rightarrow ( \cdot id ) \\
F \rightarrow ( \cdot id ), * \\
\}
\]

\[
\text{With SLR, these combine into one state.}
\]
SLR: The CLOSURE Function

*Given:*  
I = a set of LR(0) items

*Output:*  
CLOSURE(I) = a new set of items

*Algorithm:*  
result = {}  
add all items in I to result  
repeat  
  for every item A → β · C δ in result do  
    for each rule C → γ in the grammar do  
      add C → γ to result  
    endFor  
  endFor  
until we can’t add anything more to result

SLR: The GOTO Function

Let I be a set of items...  
Let X be a grammar symbol (terminal or non-terminal)...  

function GOTO(I, X) returns a set of items  
result = {}  
look at all items in I...  
  if A → α · X δ is in I  
  then add A → α X · δ to result  
result = CLOSURE(result)

In other words, move the dot past the X in any items where it is in front of an X

...and take the CLOSURE of whatever items you get
SLR: Constructing the Canonical Collection

Each $CC_i$ will be a set of items.
We will build up a collection of these.
$CC_i = \text{"The Canonical Collection of LR(0) items"}$
= a set of sets of items
= \{ CC_0, CC_1, CC_2, CC_3, \ldots CC_N \}

Algorithm to construct $CC_i$, the Canonical Collection of LR(1) Items:

let $CC_0 = \text{CLOSURE} (\{ S' \rightarrow S \cdot \), $ \}$
add $CC_0$ to $CC$
repeat
  let $CC_i$ be some set of items already in $CC$
  for each $X$ (that follows a $\cdot$ in some item in $CC_i$)...
    compute $CC_k = \text{GOTO} (CC_i, X)$
    if $CC_k$ is not already in $CC$ then
      add it
      set $\text{MOVE}(CC_i, X) = CC_k$
    endif
  endFor
  until nothing more can be added to $CC$

SLR: Algorithm to Construct ACTION and GOTO Tables

Input: Grammar $G$, augmented with $S' \rightarrow S$
Output: ACTION and GOTO Tables

Construct \text{"Canonical Collection of LR(0) items"} along with $\text{MOVE}$ function.
$CC = \{ CC_0, CC_1, CC_2, CC_3, \ldots CC_N \}$
for each $CC_i$ do
  for each item in $CC_i$ do
    if the item has the form $A \rightarrow \beta \cdot \gamma$
      and $\text{MOVE}(CC_i, c) = CC_j$ then
        set $\text{ACTION}[i, c]$ to \"Shift j\"
    elseif the item has the form $A \rightarrow \beta \cdot$ then
      for all $b$ in $\text{FOLLOW}(A)$ do
        set $\text{ACTION}[i, b]$ to \"Reduce $A \rightarrow \beta$\"
      endFor
    elseif the item has the form $S' \rightarrow S \cdot$ then
      set $\text{ACTION}[i, S]$ to \"Accept\"
  endFor
  for each nonterminal $A$ do
    if $\text{MOVE}(CC_i, A) = CC_j$ then
      set $\text{GOTO}[i, A]$ to $j$
    endif
  endFor
endFor
YAPP

Yet Another PCAT Parser

An SLR Parser Generator

**INPUT:**
- A Grammar
- A String to Parse

**ACTION:**
- Build the parsing tables using the SLR algorithm
- Parse the string

*YAPP is written in PCAT!*

cs.pdx.edu/~harry/compilers/yapp

≈ 2100 lines of PCAT code.
Can be compiled by your compiler!!!

*Example Input:*
- A Grammar for PCAT (109 rules)
- String = the YAPP program itself

---

**Attributes in a Shift-Reduce Parser**

An attribute can be associated with each grammar symbol.

\[ \text{Expr}_0 \rightarrow \text{Expr}_1 + \text{Term} \quad \text{Expr}_0.t = \text{Expr}_1.t + \text{Term}.t; \]
Attributes in a Shift-Reduce Parser

An attribute can be associated with each grammar symbol.

\[ \text{Expr}_0 \rightarrow \text{Expr}_1 + \text{Term} \quad \text{Expr}_0.t = \text{Expr}_1.t + \text{Term}.t; \]

**Synthesized Attributes:**

The attributes are computed bottom-up in the parse tree.

```
Expr
  Expr  t = 100  +  Term  t = 23
```

© Harry H. Porter, 2005
Attributes in a Shift-Reduce Parser

An attribute can be associated with each grammar symbol.

\[ \text{Expr}_0 \rightarrow \text{Expr}_1 + \text{Term} \quad \text{Expr}_0.t = \text{Expr}_1.t + \text{Term}.t; \]

Synthesized Attributes:
The attributes are computed bottom-up in the parse tree.

Put the attributes on the stack, along with grammar symbols.
Attributes in a Shift-Reduce Parser

An attribute can be associated with each grammar symbol.

\[ \text{Expr}_0 \rightarrow \text{Expr}_1 + \text{Term} \quad \text{Expr}_0.t = \text{Expr}_1.t + \text{Term}.t; \]

**Synthesized Attributes:**
The attributes are computed bottom-up in the parse tree.

```
19
T 23
15
+ --
8
E 100
4
( --
0
```

To reduce by \( \text{Expr} \rightarrow \text{Expr} + \text{Term} \):
- Perform the attribute computation
- Pop the stack
- Push the new non-terminal with its attribute.
**YACC**

*Yet Another Compiler Compiler*
Unix tool to create an LALR parser.
Works with “Lex” tool: Calls `yylex()` to get next token.

- `grammar.y`
- `lex.l`
- `y.tab.c`
- `lex.yy.c`
- `lex.l`

**An Example YACC Grammar**

```c
#include <stdio.h>
define YYSTYPE double

{%
token NUMBER

S : S E `/n' { printf("%g\n", $2); }  // printf call
| S `/n
| |
E : E `+` T { $$ = $1 + $3; }        // arithmetic expression
| T { $$ = $1; }                    // term
| ;
T : T `*` F { $$ = $1 * $3; }       // factor
| F { $$ = $1; }                    // number
| ;
F : NUMBER { $$ = $1; }             // literal number
| `(` E `)` { $$ = $2; }            // parenthesized expression
| ;
%
#include lex.yy.c
%
```

This material copied as is to `yy.tab.c`

An attribute is associated with each symbol
This tells what type the attribute has.

The token types (from lexer)

Epsilon
End of rule

The grammar rules:

- `S -> S E /n`
- `S -> /n`
- `E -> E + T`
- `T -> T * F`
- `T -> NUMBER`
- `F -> NUMBER`
- `F -> ( E )`

© Harry H. Porter, 2005
An Example YACC Grammar

%{
#include <stdio.h>
#define YYSTYPE double
}%
token NUMBER

S : S E '/n' { printf("%g\n", $2); }
| S '/n'
| ;
E : E ' + ' T { $$ = $1 + $3; }
| T { $$ = $1; }
| ;
T : T ' * ' F { $$ = $1 * $3; }
| F { $$ = $1; }
| ;
F : NUMBER { $$ = $1; }
| '(' E ')' { $$ = $2; }
| ;
%
#include lex.yy.c
%

Actions are executed when reductions are done!
The attribute of each symbol is referred to by position:

E → E + T
$$ $1 $2 $3

The attributes (such as $$ and $1) are replaced by the appropriate code.

How the $ Notation in YACC Works

Expr<sub>0</sub> → Expr<sub>1</sub> + Term

Expr<sub>0</sub>.t = Expr<sub>1</sub>.t + Term.t;

$$ = $1 + $3;

Expr<sub>0</sub> = 123

19
15
+ 8
E 100
4

15
8
4

8

Expr<sub>1</sub> = 100

19

Term t = 23

23
8

23
8

Expr<sub>1</sub> = 23

23

0

0

Term t = 23

15

...