Building LR Tables

How to construct the ACTION and GOTO tables?

- Define "items"
- Define "viable prefix"
- Define the "closure function"

  Set-of-items $\rightarrow$ Set-of-items
- Define the GOTO function
- Work with a set of sets of items

  A collection of sets of items

  CC = Cannonical 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:

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

$E \rightarrow \bullet T$
$E \rightarrow T \bullet$

$T \rightarrow \bullet T * F$
$T \rightarrow T \bullet * F$
$T \rightarrow T * \bullet F$
$T \rightarrow T * F \bullet$

$T \rightarrow \bullet F$
$T \rightarrow F \bullet$

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

$F \rightarrow \bullet$ id
$F \rightarrow$ id $\bullet$

Special Case:

Rule:

$A \rightarrow \varepsilon$

Yields one item:

$A \rightarrow \bullet$

LR(1) Items

Just like before, except...

- Look-ahead symbol
- Terminal symbol from grammar

The look-ahead symbol

Grammar:

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

Examples:

E → • E + T ,)
E → • E + T , $
E → E • + T ,)
E → E • + T , $
. . .

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

Intuition behind LR(1) Items

F → (• E) ,)

We were hoping / expecting to see an F next, followed by a)
and we have already seen a (.
We are on the path to finding an F, followed by a) .
Using rule 5, one way to find an F is to find (E) next.
So now we are looking for E) , followed by a) .

F → (E) • ,)

We were looking for an F, followed by a)
and we have found (E)
If a) comes next then the parse is going great!
... Now reduce, using rule F → (E)

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

Intuition behind LR(1) Items

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

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).

$E \rightarrow \bullet T ,\ )$

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) ?

$T \rightarrow \bullet T * F ,\ )$

$T \rightarrow \bullet F ,\ )$

Step 1

Augment the grammar by adding...

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

"Goal"

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

0. S' → E
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!

0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

The CLOSURE Function

Let's say we have this item:

E → • T,)

What are the ways to find a T?

T → F

T → T * F

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

T → • F,)

T → • T * F,)

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

How can we find that?

F → • (E),)

F → • id ,)

We can also find a T followed by a) if we find an T * F following by a) .

To find that, we need to first find another T, but followed by *.

T → • F, *

T → • T * F, *

So we should also look for a F followed by a *.

F → • (E), *

F → • id , *

The CLOSURE Function

Given:

I = a set of items

Output:

CLOSURE(I) = a new set of items

Algorithm:

```
result = {}
add all items in I to result
repeat
  for every item A→β•Cδ,a in result do
    for each rule C→γ in the grammar do
      for each b in FIRST(δa) do
        add C →•γ,b to result
      endFor
    endFor
  endFor
until we can't add anything more to result
```

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. $F \rightarrow$ `id`

CLOSURE Function Example

Example: Let I_1 = { E → E • + T,)
T → T • * F ,)
F → id •,)
F → (E) •,)
}

Compute: CLOSURE (I_1) = {

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. $F \rightarrow$ id

CLOSURE Function Example

Example: Let I_1 = { $E \rightarrow E \bullet$ + T,)
$T \rightarrow T \bullet$ * F,)
$F \rightarrow$ id $\bullet$,)
$F \rightarrow$ (E) $\bullet$,)
}

Compute: CLOSURE (I_1) = {

Start by adding all items in I...

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

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. $F \rightarrow$ id

CLOSURE Function Example

Example: Let I_1 = {
$E \rightarrow E \bullet + T$,)
$T \rightarrow T \bullet * F$,)
$F \rightarrow$ id $\bullet$,)
$F \rightarrow$ (E) $\bullet$,)
}

Compute: CLOSURE (I_1) = {

Start by adding all items in I...

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

Is the dot in front of a non-terminal?

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. $F \rightarrow$ `id`

CLOSURE Function Example

Example: Let $I_1 = \{$
$E \rightarrow E \bullet + T,)$
$T \rightarrow T \bullet * F,)$
$F \rightarrow \mathtt{id} \bullet,)$
$F \rightarrow (E) \bullet,)$
$\}$

Compute: CLOSURE (I_1) = {

Start by adding all items in I...

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

Is the dot in front of a non-terminal?

No... no more items are added.

}

0.	S'	→	E		
1.	E	→	E	+	T
2.	E	→	T		
3.	T	→	T	*	F
4.	T	→	F		
5.	F	→	(	E	)
6.	F	→	id		

CLOSURE Function Example

Example: Let I_2 = { T → • F,)
T → • T * F,) }

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

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. $F \rightarrow$ <u>id</u>

CLOSURE Function Example

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

Compute: CLOSURE (I_2) = {

Start by adding all items in I...

(1) $T \rightarrow \bullet F$,)

(2) $T \rightarrow \bullet T$ * F,)

Look at (1) first...

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. $F \rightarrow$ id

CLOSURE Function Example

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

Compute: CLOSURE (I_2) = {

Start by adding all items in I...

(1) $T \rightarrow \bullet F$,)

(2) $T \rightarrow \bullet T * F$,)

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

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. $F \rightarrow$ id

CLOSURE Function Example

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

Compute: CLOSURE (I_2) = {

Start by adding all items in I...

(1) $T \rightarrow \bullet F$,)

(2) $T \rightarrow \bullet T * F$,)

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

(3) $F \rightarrow \bullet (E)$,)

(4) $F \rightarrow \bullet$ id,)

Look at (2) next...

0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

CLOSURE Function Example

Example: Let I_2 = { T → • F,)
T → • T * F,) }

Compute: CLOSURE (I_2) = {

Start by adding all items in I...

(1) T → • F,)

(2) T → • T * F,)

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

(3) F → • (E),)

(4) F → • id ,)

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

(5) T → • F, *

(6) T → • T * F, *

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

0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

CLOSURE Function Example

Example: Let I_2 = { T → • F,)
T → • T * F,) }

Compute: CLOSURE (I_2) = {

Start by adding all items in I...

(1) T → • F,)
(2) T → • T * F,)

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

(3) F → • (E),)
(4) F → • id,)

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

(5) T → • F, *
(6) T → • T * F, *

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

Look at (5) next...

0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

CLOSURE Function Example

Example: Let I_2 = { T → •F,)
T → •T * F,) }

Compute: CLOSURE (I_2) = {

Start by adding all items in I...

(1) T → •F,)
(2) T → •T * F,)

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

(3) F → • (E),)
(4) F → •id,)

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

(5) T → •F, *
(6) T → •T * 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 FIRST (ε*) = { * }...

(7) F → • (E), *
(8) F → •id, *

Look at (6) next...

0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

CLOSURE Function Example

Example: Let I_2 = { T → • F,)
T → • T * F,) }

Compute: CLOSURE (I_2) = {

Start by adding all items in I...

(1) T → • F,)
(2) T → • T * F,)

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

(3) F → • (E),)
(4) F → • id,)

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

(5) T → • F, *
(6) T → • T * 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 FIRST (ε*) = {*}...

(7) F → • (E), *
(8) F → • id, *

Look at (6) next. Look at each T rule. For every b in FIRST (*F*) = {*}...
We already added (5) and (6)

Look at (7) and (8) next...

0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

CLOSURE Function Example

Example: Let I_2 = { T → • F,)
T → • T * F,) }

Compute: CLOSURE (I_2) = {

Start by adding all items in I...

(1) T → • F,)
(2) T → • T * F,)

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

(3) F → • (E),)
(4) F → • id ,)

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

(5) T → • F, *
(6) T → • T * 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 FIRST (ε*) = {*}...

(7) F → • (E), *
(8) F → • id , *

Look at (6) next. Look at each T rule. For every b in FIRST (*F*) = {*}...

We already added (5) and (6)

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

}

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. $F \rightarrow$ id

CLOSURE Function Example

Example: Let $I_3 = \{ E \rightarrow \bullet E + T,) \}$

Compute: CLOSURE (I_3) = {

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

Look at E rules. For every b in FIRST (+T)) = {+}...

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

$E \rightarrow \bullet T, +$

Look at E rules. For every b in FIRST (+T+)... (Nothing new added)

Look at T rules. For every b in FIRST (ε+) = {+}

$T \rightarrow \bullet T * F, +$

$T \rightarrow \bullet F, +$

Look at T rules. For every b in FIRST (*F+) = {*}

$T \rightarrow \bullet T * F, *$

$T \rightarrow \bullet F, *$

Look at F rules. For every b in FIRST (ε+) = {+}

$F \rightarrow \bullet (E), +$

$F \rightarrow \bullet$ id, +

Look at F rules. For every b in FIRST (ε*) = {*}

$F \rightarrow \bullet (E), *$

$F \rightarrow \bullet$ id, *

}

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 , a is in I
        then add A → α X • δ , a 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

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 , a is in I
        then add A → α X • δ , a 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

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.

0.	S' → E
1.	E → E + T
2.	E → T
3.	T → T * F
4.	T → F
5.	F → (E)
6.	F → id

GOTO Function Example

Example: Let I_4 = { E → T •,)
T → T • * F,) }

Compute: GOTO (I_4, *) = {

Is the • in front of * in any of the items?

T → T * • F,)

Now take the closure...

F → • (E) ,)
F → • id,)
}

Intuition:

We found a T and then we found a *. What are we looking for next?

T → T * • F,)

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

F → • (E) ,)

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

F → • id,)

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

0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

GOTO Function Example

Example: Let I_5 = { E → • T ,)
T → • T * F ,) }

Compute: GOTO (I_5, T) = {

Is the • in front of T in any of the items?

E → T • ,)
T → T • * F ,)

Now take the closure...

Is the • in from of any non-terminal? Nothing more added...

}

Intuition:

We were looking for a T. Then we found it. What are we looking for next?

E → T • ,)

Means: We are now looking for)

T → T • * F ,)

Means: We are now looking for * F followed by)

Constructing the Canonical Collection

Each CC_i will be a set of items.

We will build up a collection of these.

$\mathbb{CC}$ = "*The Canonical Collection of LR(1) items*"

= a set of sets of items

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

Constructing the Canonical Collection

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

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

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

```
let CC₀ = CLOSURE ( {S' → •S , $} )
add CC₀ to ℂℂ
repeat
  let CCᵢ be some set of items already in ℂℂ
  for each X (that follows a • in some item in CCᵢ) . . .
    compute CCₖ = GOTO (CCᵢ,X)
    if CCₖ is not already in ℂℂ then
      add it
      set MOVE(CCᵢ,X) = CCₖ
    endIf
  endFor
until nothing more can be added to ℂℂ
```

0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

Example:

CC_0 = CLOSURE({S' → • E , $})
= { S' → • E , $
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 , * }

Example:

0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

CC_0 = CLOSURE({S' → • E , $})
= { S' → • E , $
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, id, and (...

Next, we'll compute...

GOTO (CC_0, E) ⇒ CC_1
GOTO (CC_0, T) ⇒ CC_2
GOTO (CC_0, F) ⇒ CC_3
GOTO (CC_0, id) ⇒ CC_5
GOTO (CC_0, ( ) ⇒ CC_4

0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

GOTO (CC_0, E) = CC_1

Advance • past **E** in the items containing • **E**

CC_1 = {

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 we will look for a **+** next.

The • is in front of **+**. We'll come back to CC_1 later.

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. $F \rightarrow$ id

GOTO (CC_0, T) = CC_2

Advance • past T

CC_2 = {

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 CC_2 later.

0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

GOTO (CC_0, F) = CC_3

Advance • past **F**

CC_3 = {

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.

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. $F \rightarrow$ **id**

GOTO (CC_0, **id**) = CC_5

Advance • past **id**

CC_5= {

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.

Advance • past `(`

GOTO (CC_0, `(`) = CC_4

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

And take the closure...

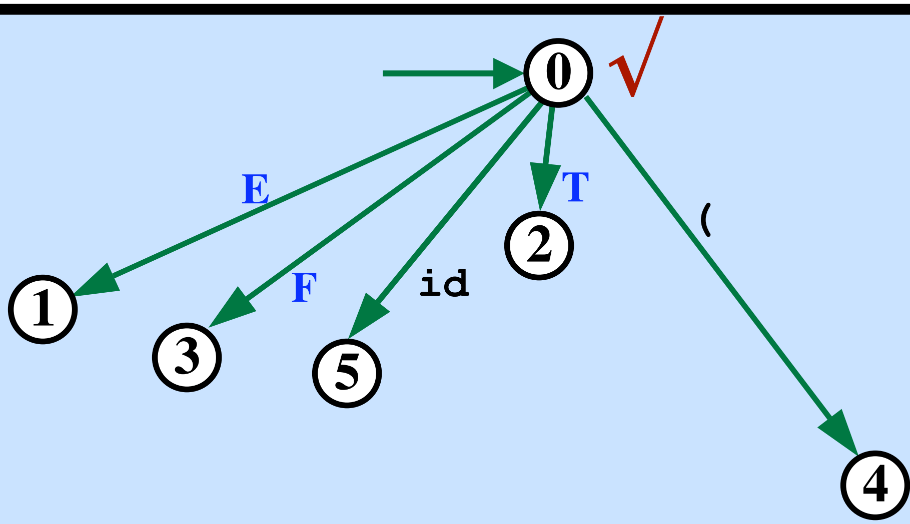
E → • E + T ,)
E → • E + T , +
E → • T ,)
E → • T , +
T → • T * F ,)
T → • T * F , +
T → • T * F , *
T → • F ,)
T → • F , +
T → • F , *
F → • (E),)
F → • (E), +
F → • (E), *
F → • id ,)
F → • id , +
F → • id , * }

0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

The • is before E, T, F, (, and id...

Next, we'll compute...

GOTO (CC_4, E) ⇒ CC_8
GOTO (CC_4, T) ⇒ CC_9
GOTO (CC_4, F) ⇒ CC_{10}
GOTO (CC_4, ( ) ⇒ CC_{11}
GOTO (CC_4, id) ⇒ CC_{12}


0
√
E
T
(
1
2
F
id
3
5
4

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. $F \rightarrow$ `id`

GOTO (CC_1, +) = CC_6

CC_1 = { $S' \rightarrow E \bullet$, `$`
$E \rightarrow E \bullet$ `+` T, `$`
$E \rightarrow E \bullet$ `+` T, `+` }

Advance • past **+** in the items containing • **+**

CC_6 = { $E \rightarrow E$ `+` $\bullet T$, `$`
$E \rightarrow E$ `+` $\bullet T$, `+`

And take the closure...

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

0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

GOTO (CC_2, *) = CC_7

CC_2 = {
E → T •, $
E → T •, +
T → T • * F, $
T → T • * F, +
T → T • * F, *

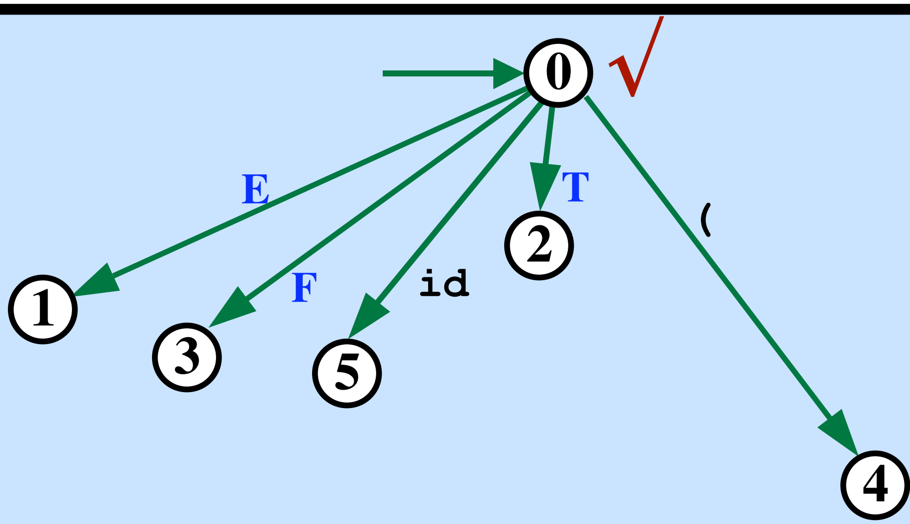
Advance • past *

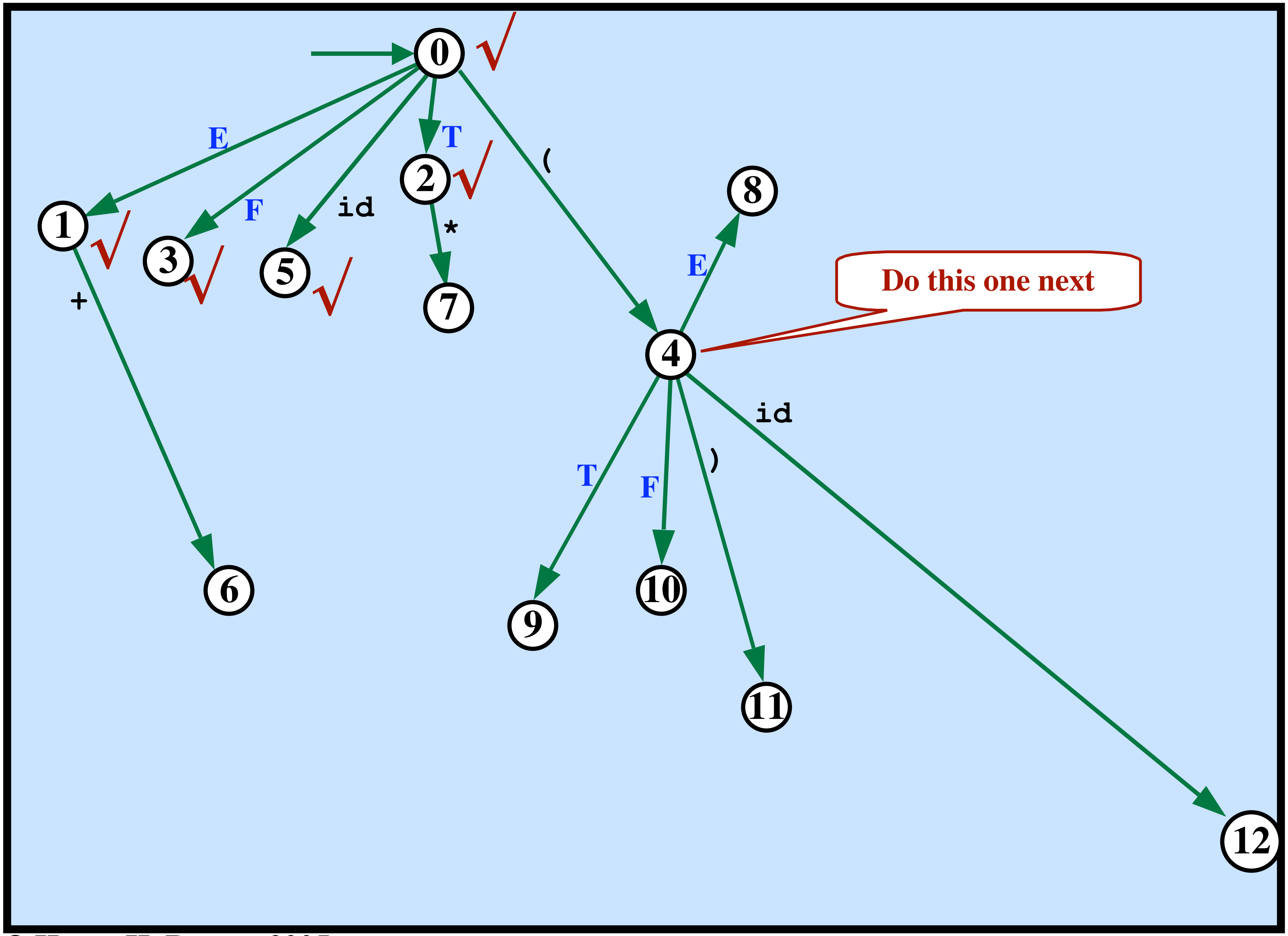
CC_7 = {
T → T * • F, $
T → T * • F, +
T → T * • F, *

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

Intuition:

We have found T *.
Next, look for a F followed by $, +, or *.


0
E
T
(
F
id
1
2
3
5
4



0
E
T
F
id
(
1
2
3
5
7
+
*
8
E
4
Do this one next
id
T
F
)
6
9
10
11
12

0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

GOTO (CC_4, E) = CC_8

CC_8 = {
F → (E •), $
F → (E •), +
F → (E •), *
E → E • + T ,)
E → E • + T , +
And take the closure...
}

0.	S'	→	E		
1.	E	→	E	+	T
2.	E	→	T		
3.	T	→	T	*	F
4.	T	→	F		
5.	F	→	(	E	)
6.	F	→	id		

GOTO (CC_4, T) = CC_9

CC_9 = {

E → T •,)

E → T •, +

T → T • * F,)

T → T • * F, +

T → T • * F, *

And take the closure...

}

0.	S' →	E
1.	E →	E + T
2.	E →	T
3.	T →	T * F
4.	T →	F
5.	F →	(E)
6.	F →	id

GOTO (CC_4, F) = CC_{10}

CC_{10} = {

T → F •,)

T → F •, +

T → F •, *

And take the closure...

}

0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

GOTO (CC_4, id) = CC_{12}

CC_{12} = {
F → id•,)
F → id•, +
F → id•, *
And take the closure...
}

GOTO (CC_4, ( ) = CC_{11}

CC_{11} = {

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 , * }

0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

CC_{11} is similar to, but not quite the same as CC_0

The • is before E, T, F, id, and (...

Next, we'll compute...

GOTO (CC_{11}, E) ⇒ CC_{18}
GOTO (CC_{11}, T) ⇒ CC_9
GOTO (CC_{11}, F) ⇒ CC_{10}
GOTO (CC_{11}, id) ⇒ CC_{12}
GOTO (CC_{11}, ( ) ⇒ CC_{11}

GOTO (CC_{11}, E) = CC_{18}

CC_{18} = {

F → (E •),)
F → (E •), +
F → (E •), *
E → E • + T ,)
E → E • + T , +

And take the closure...

}

0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

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. $F \rightarrow$ `id`

GOTO (CC_{11}, T) = CC_9

CC_9 = {
$E \rightarrow T \bullet$, `)`
$E \rightarrow T \bullet$, `+`
$T \rightarrow T \bullet$ `*` F, `)`
$T \rightarrow T \bullet$ `*` F, `+`
$T \rightarrow T \bullet$ `*` F, `*`
And take the closure...
}

We have seen CC_9 before!

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.	$F \rightarrow$ id	

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

CC_{10} = {

$T \rightarrow F \bullet$,)

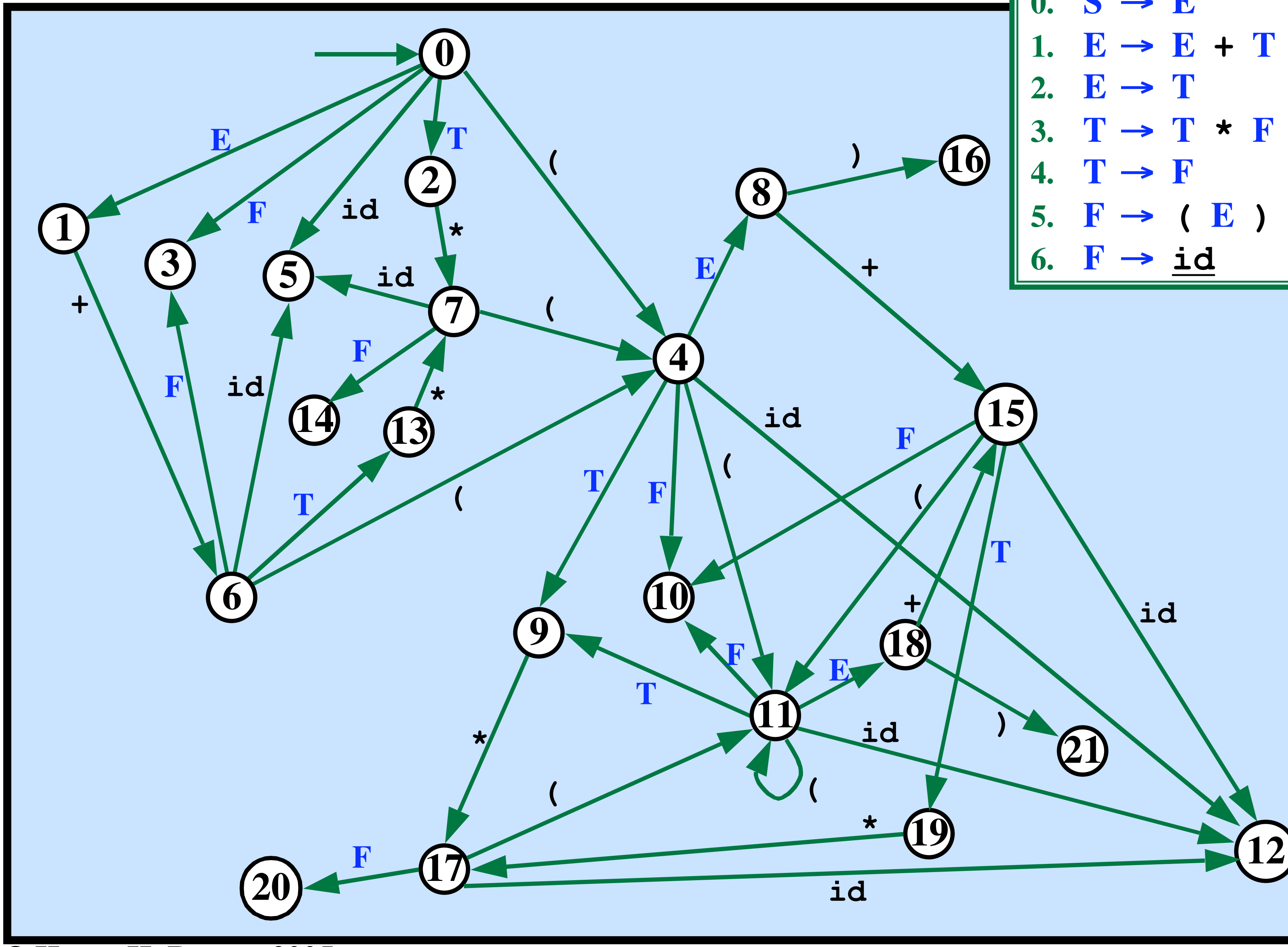
$T \rightarrow F \bullet$, +

$T \rightarrow F \bullet$, *

And take the closure...

}

We have seen CC_{10} before!



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. $F \rightarrow \underline{id}$

Viable Prefixes

Consider a right-sentential form

$$S \Rightarrow_{RM}^{*} XXA\mathtt{fff} \Rightarrow_{RM} XX\underbrace{BCDE}_{\textit{Handle}}\mathtt{fff} \Rightarrow_{RM} \cdots$$

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.

$$\underbrace{XXBC}_{\textit{A Viable prefix}}DE\mathtt{fff}$$

Viable Prefixes

Consider a right-sentential form

$S \Rightarrow_{RM}^{*} XXA\mathtt{fff} \Rightarrow_{RM} XX\underbrace{BCDE}_{\textit{Handle}}\mathtt{fff} \Rightarrow_{RM} \ldots$

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.

$\underbrace{XXBC}_{\textit{A Viable prefix}}DE\mathtt{fff}$

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

$D \Rightarrow^{*} \mathtt{ddd}$ and $E \Rightarrow^{*} \mathtt{eee}$

$S \ldots \Rightarrow_{RM}^{*} XXBCDE\mathtt{fff} \Rightarrow_{RM} XXBCD\mathtt{eeefff} \Rightarrow_{RM} XXBC\mathtt{dddeeefff}$

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 || REMAINING-INPUT

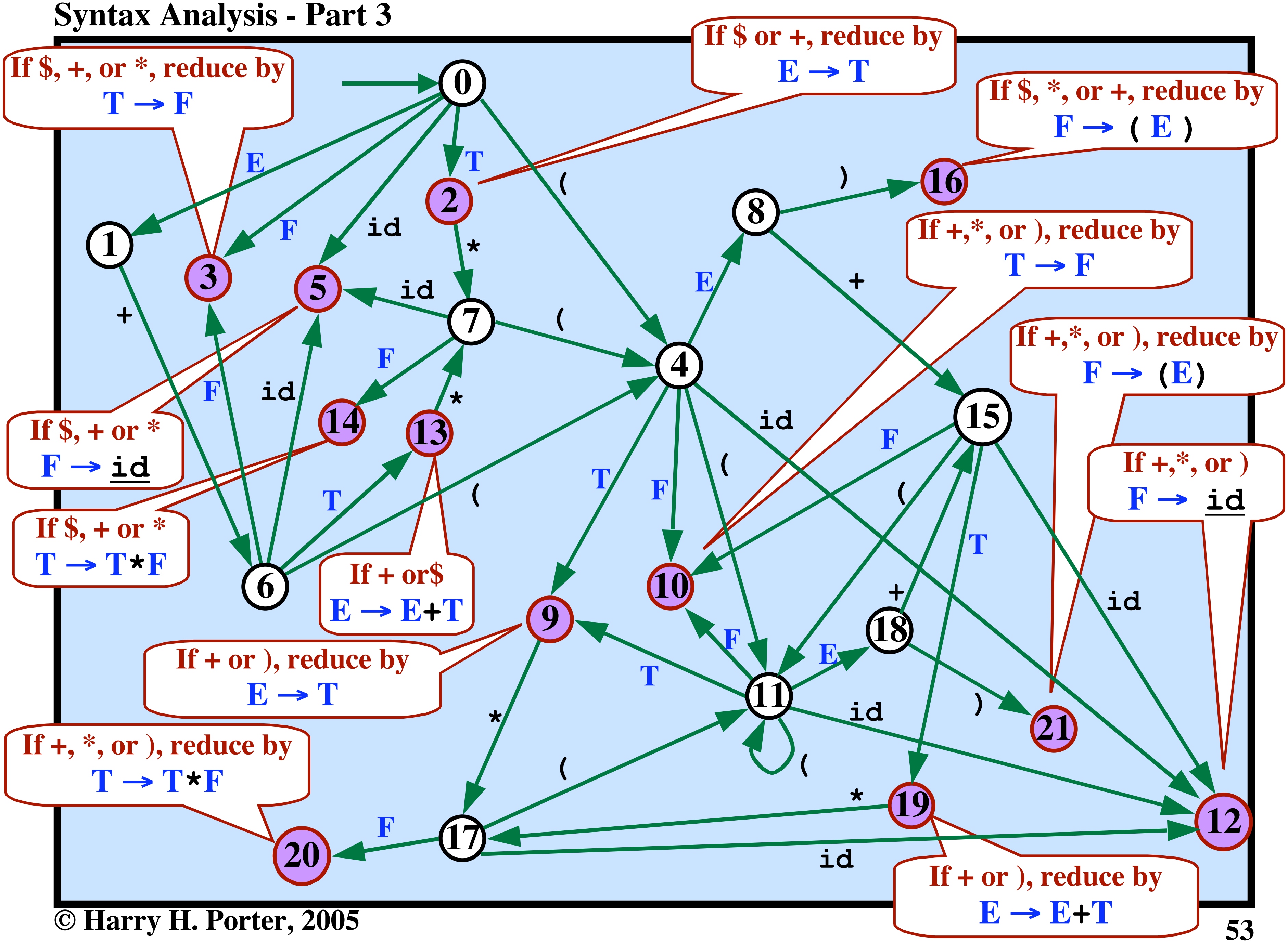
  into a right-sentential form.
- We are on course to finding a rightmost derivation.

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. $A \rightarrow XYZ$


If $, +, or *, reduce by
T → F
If $ or +, reduce by
E → T
If $, *, or +, reduce by
F → (E)
If +,*, or), reduce by
T → F
If +,*, or), reduce by
F → (E)
If +,*, or)
F → id
If $, + or *
F → id
If $, + or *
T → T*F
If + or$
E → E+T
If + or), reduce by
E → T
If +, *, or), reduce by
T → T*F
If + or), reduce by
E → E+T

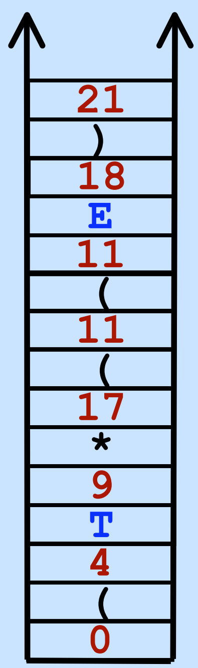
Example

Consider this viable prefix. Trace throught the DFA.

`( T * ( ( E )` Ends up in state 21, a final state.

If next token is `+`, `*`, or) then reduce by F→`(`E`)`

0. S' → E
1. E → E `+` T
2. E → T
3. T → T `*` F
4. T → F
5. F → `(` E `)`
6. F → <u>`id`</u>



The stack shows our path through the DFA.
"How we got to state 21"

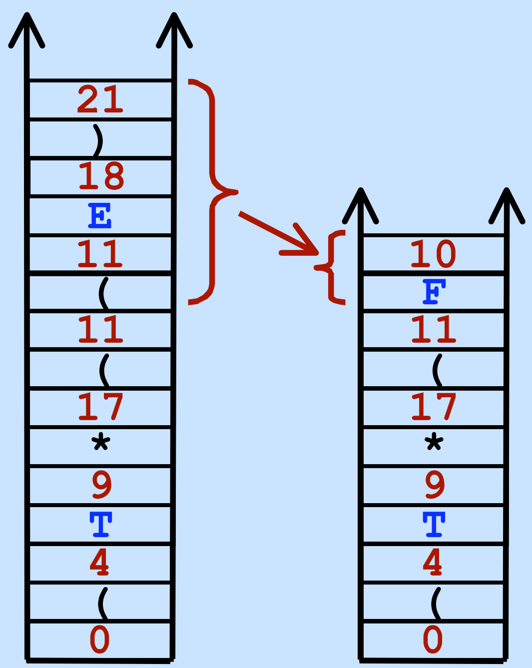
0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

Example

Consider this viable prefix. Trace throught the DFA.

`( T * ( ( E )` Ends up in state 21, a final state.

If next token is `+`, `*`, or) then reduce by F→ `(E )`



Back up along our path...
to where we were
before we saw `(E)`.
Then take the `F` edge
to get to state 10.

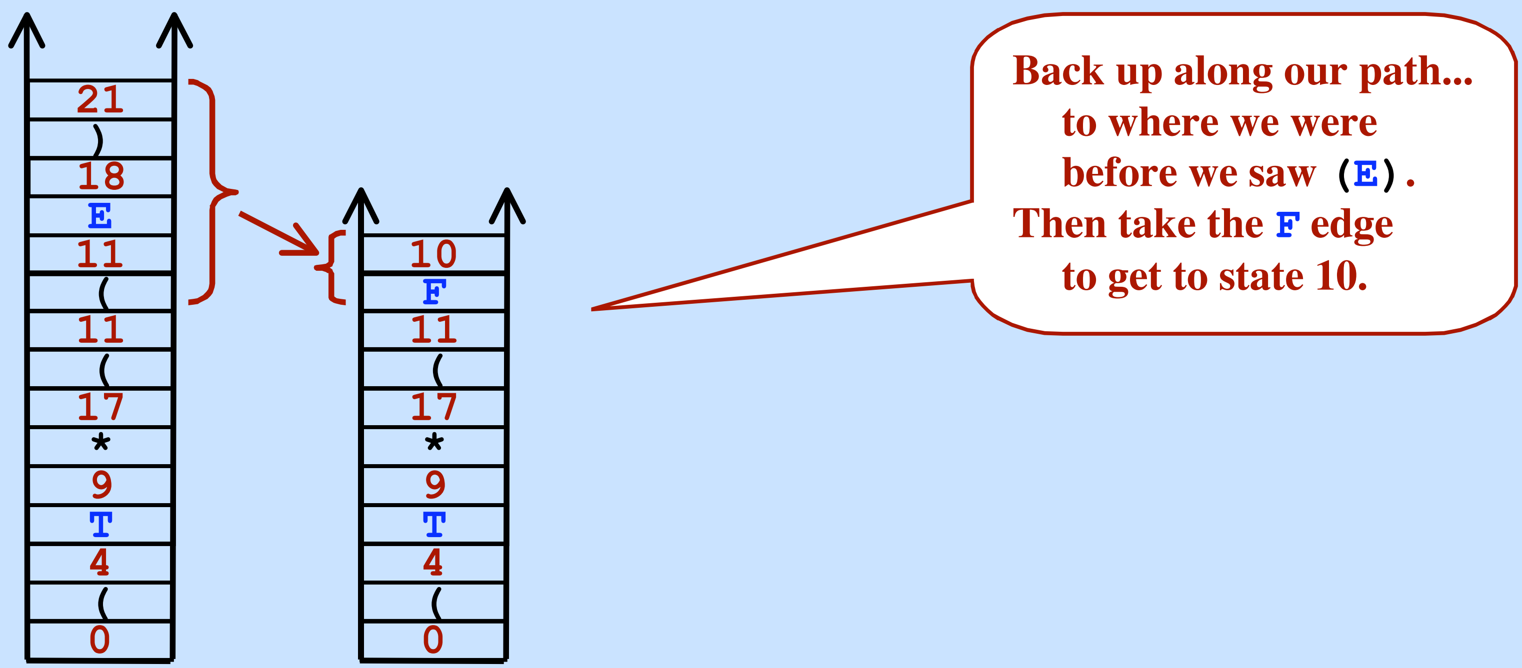
0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

Example

Consider this viable prefix. Trace throught 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



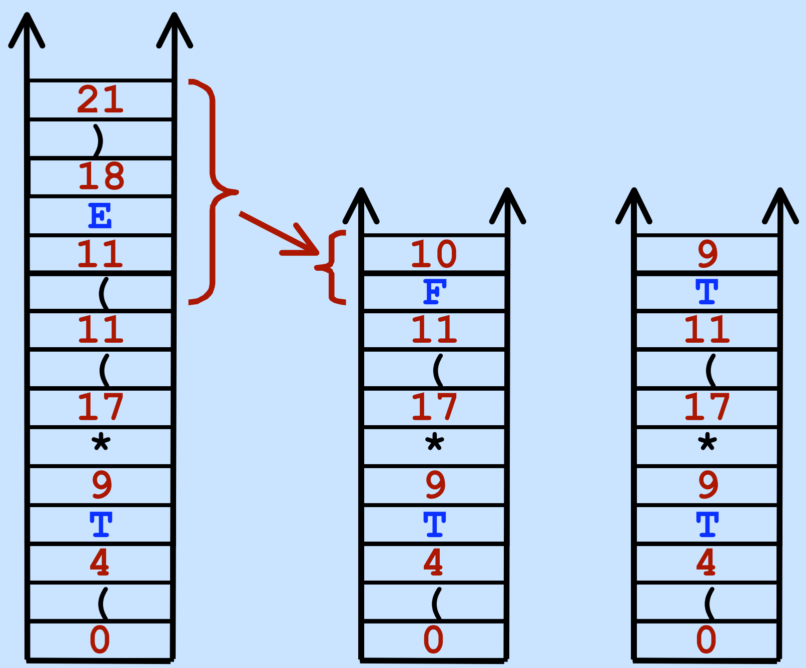
0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

Example

Consider this viable prefix. Trace throught 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.

0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

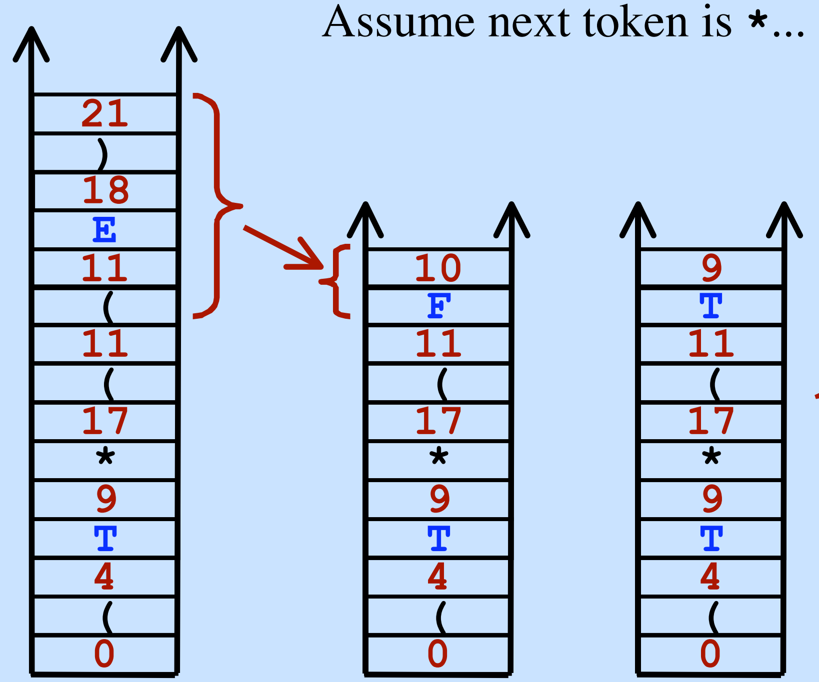
Example

Consider this viable prefix. Trace throught 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



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

0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

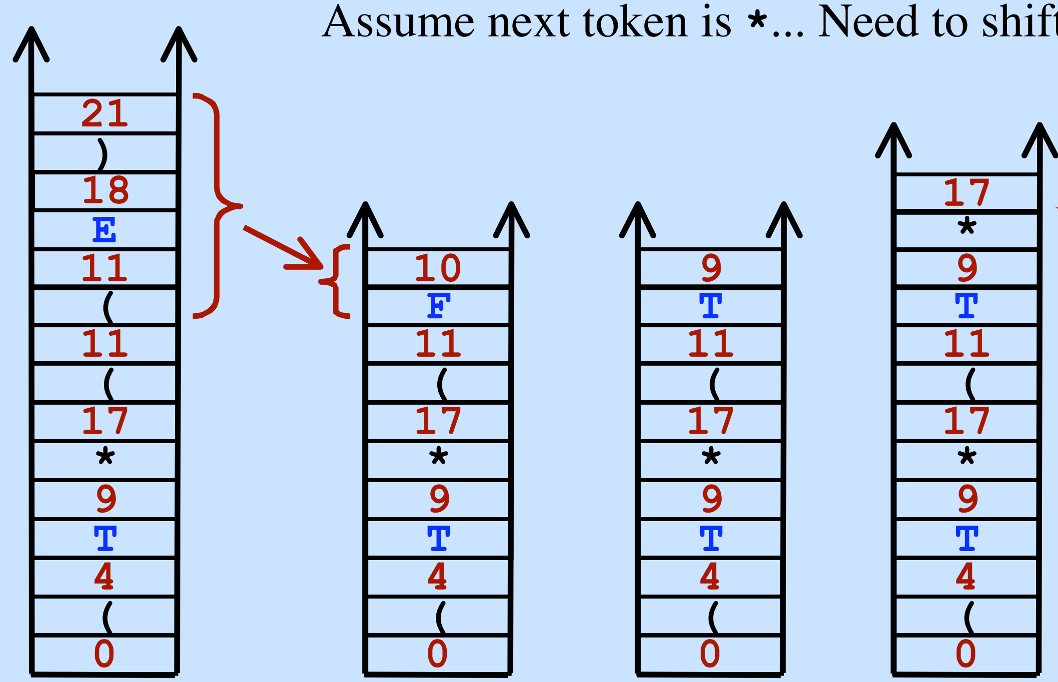
Example

Consider this viable prefix. Trace throught 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



There is an edge from state 9 labeled *.
Take the * edge to get to state 17.

Now we've got this viable prefix:
`( T * ( T *`

0.	S'	→	E		
1.	E	→	E	+	T
2.	E	→	T		
3.	T	→	T	*	F
4.	T	→	F		
5.	F	→	(	E	)
6.	F	→	id		

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.

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

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

```
for each CCi do
  for each item in CCi do
    if the item has the form A → β • c γ, a
                   and MOVE(CCi,c) = CCj then
      set ACTION[i,c] to "Shift j"
    elseIf the item has the form A → β •, a then
      set ACTION[i,a] to "Reduce A → β"
    elseIf the item has the form S' → S •, $ then
      set ACTION[i,$] to "Accept"
  endFor
  for each nonterminal A do
    if MOVE(CCi,A) = CCj then
      set GOTO[i,A] to j
    endIf
  endFor
endFor
```

The SLR Table Construction Algorithm

With SLR, we do not have the lookahead symbol.

LR(1) items:

$F \to (\bullet E),\ )$
$F \to (\bullet E),\ +$
$F \to (\bullet E),\ *$

LR(0) items:

$F \to (\bullet E)$

Some information is lost.
Some states in $\mathbb{CC}$ collapse into one state.
There are fewer states in $\mathbb{CC}$
$\Rightarrow$ Fewer rows in the resulting tables.

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

CC_{11} = { F → (• E) ,)
F → (• E) , +
F → (• E) , *
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 , * }

With SLR, these combine into one state

{ F → (• E)
E → • E + T
E → • T
T → • T * F
T → • F
F → • (E)
F → • id }

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.

```
...
elseIf the item has the form A → β •, a then
  set ACTION[i,a] to "Reduce A → β"
...
```

↓

```
...
elseIf the item has the form A → β • then
  for all b in FOLLOW(A) do
    set ACTION[i,b] to "Reduce A → β"
  endFor
...
```

Sometimes SLR may try to put two actions in one table entry

...while the LR(1) tables would have more states, more rows, and no conflicts.

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.

$\mathbb{CC}$ = "*The Canonical Collection of LR(0) items*"

= a set of sets of items

= { CC_0, CC_1, CC_2, CC_3,... CC_N}

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

```
let CC0 = CLOSURE ( {S' → •S , $} )
add CC0 to ℂℂ
repeat
  let CCi be some set of items already in ℂℂ
  for each X (that follows a • in some item in CCi) . . .
    compute CCk = GOTO (CCi,X)
    if CCk is not already in ℂℂ then
      add it
      set MOVE(CCi,X) = CCk
    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 "Canonical Collection of LR(0) items" along with MOVE function.
    ℂℂ = { CC0, CC1, CC2, CC3, ... CCN}
for each CCi do
  for each item in CCi do
    if the item has the form A → β • c γ
                     and MOVE(CCi,c) = CCj then
      set ACTION[i,c] to "Shift j"
    elseIf the item has the form A → β • then
      for all b in FOLLOW(A) do
        set ACTION[i,b] to "Reduce A → β"
      endFor
    elseIf the item has the form S' → S • then
      set ACTION[i,$] to "Accept"
  endFor
  for each nonterminal A do
    if MOVE(CCi,A) = CCj then
      set 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.

$\text{Expr}_{\mathbf{0}} \rightarrow \text{Expr}_{\mathbf{1}}$ **+** Term   `Expr`$_0$`.t = Expr`$_1$`.t + Term.t;`

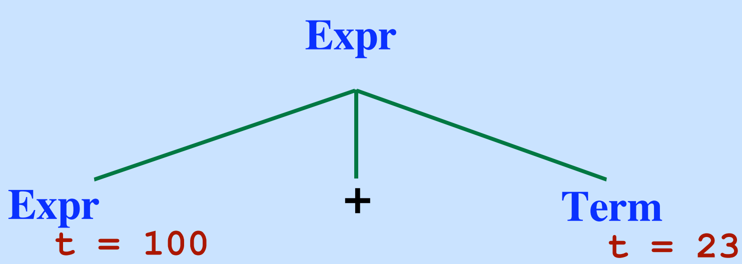
Attributes in a Shift-Reduce Parser

An attribute can be associated with each grammar symbol.

$\text{Expr}_0 \rightarrow \text{Expr}_1 + \text{Term}$ `Expr0.t = Expr1.t + Term.t;`

Synthesized Attributes:

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



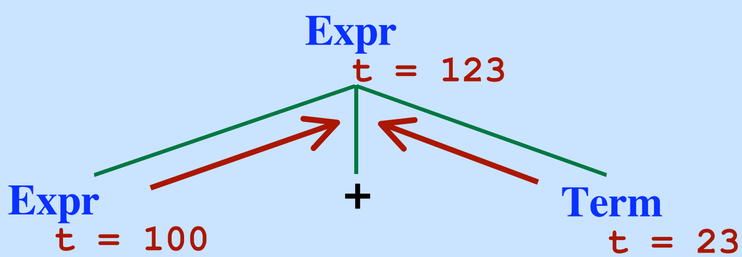
Attributes in a Shift-Reduce Parser

An attribute can be associated with each grammar symbol.

$Expr_0 \rightarrow Expr_1 + Term$ `Expr0.t = Expr1.t + Term.t;`

Synthesized Attributes:

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



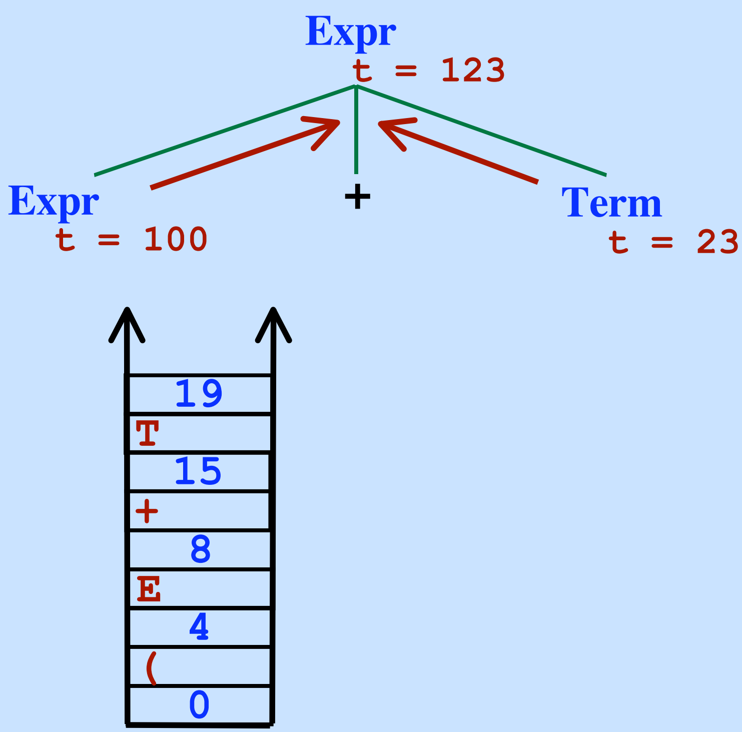
Attributes in a Shift-Reduce Parser

An attribute can be associated with each grammar symbol.

$\text{Expr}_0 \rightarrow \text{Expr}_1 + \text{Term}$ `Expr0.t = Expr1.t + Term.t;`

Synthesized Attributes:

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



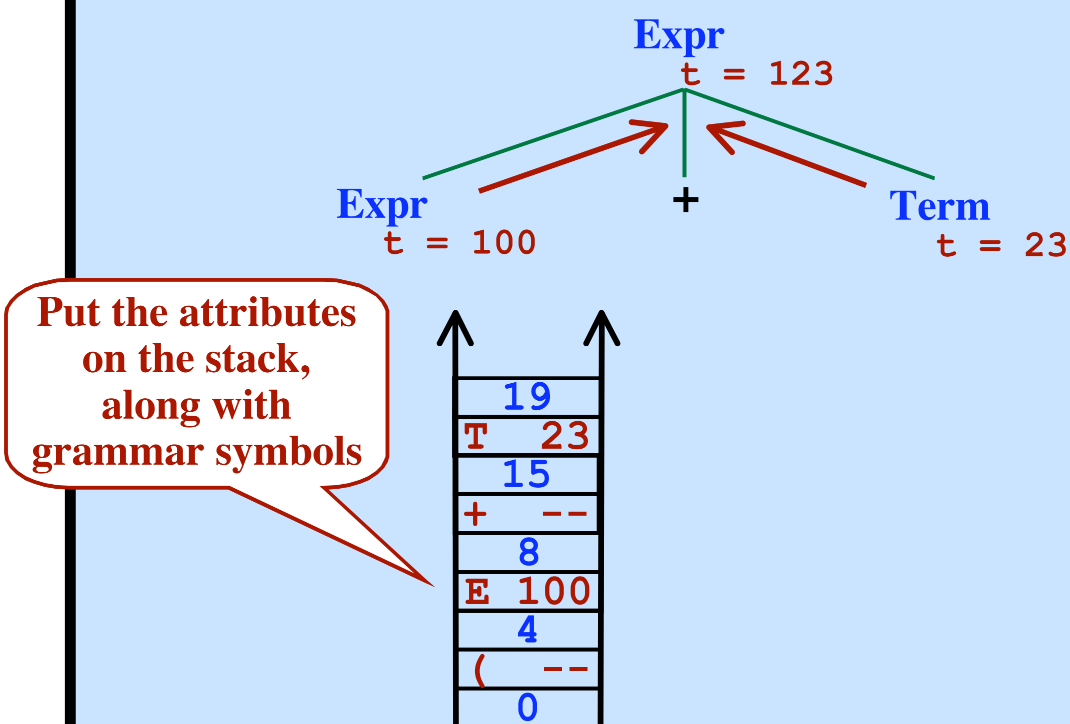
Attributes in a Shift-Reduce Parser

An attribute can be associated with each grammar symbol.

$\text{Expr}_0 \rightarrow \text{Expr}_1 + \text{Term}$ `Expr0.t = Expr1.t + Term.t;`

Synthesized Attributes:

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



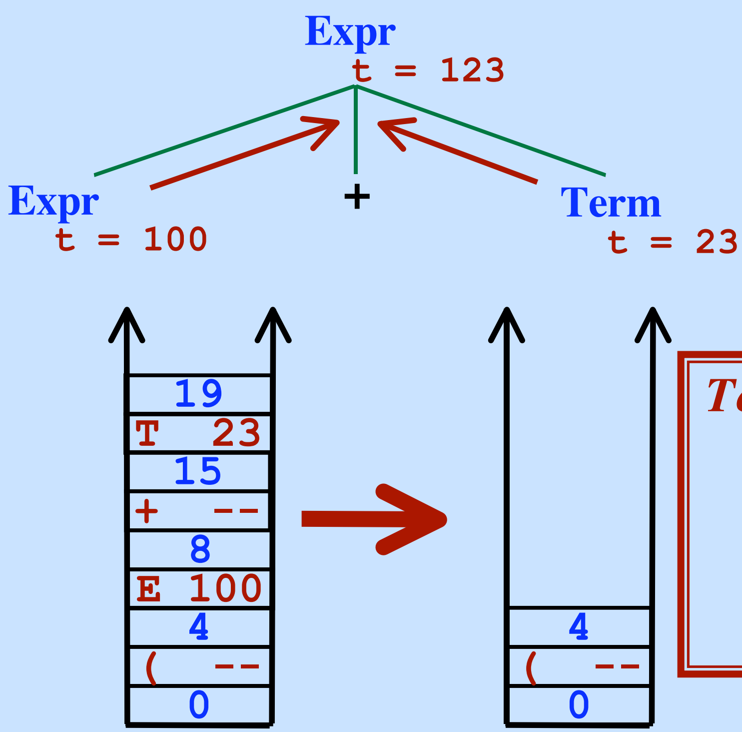
Attributes in a Shift-Reduce Parser

An attribute can be associated with each grammar symbol.

$Expr_0 \rightarrow Expr_1 + Term$ `Expr0.t = Expr1.t + Term.t;`

Synthesized Attributes:

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



To reduce by $Expr \rightarrow Expr + Term$
- *Perform the attibute computation*
- *Pop the stack*
- *Push the new non-terminal with its attribute.*

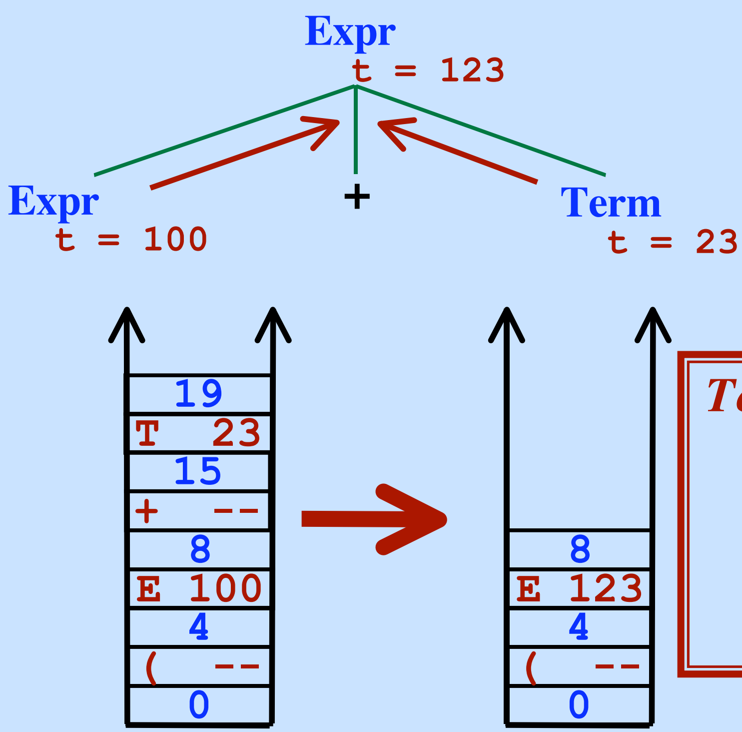
Attributes in a Shift-Reduce Parser

An attribute can be associated with each grammar symbol.

$\text{Expr}_0 \rightarrow \text{Expr}_1 + \text{Term}$ `Expr₀.t = Expr₁.t + Term.t;`

Synthesized Attributes:

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



To reduce by $\text{Expr} \rightarrow \text{Expr} + \text{Term}$

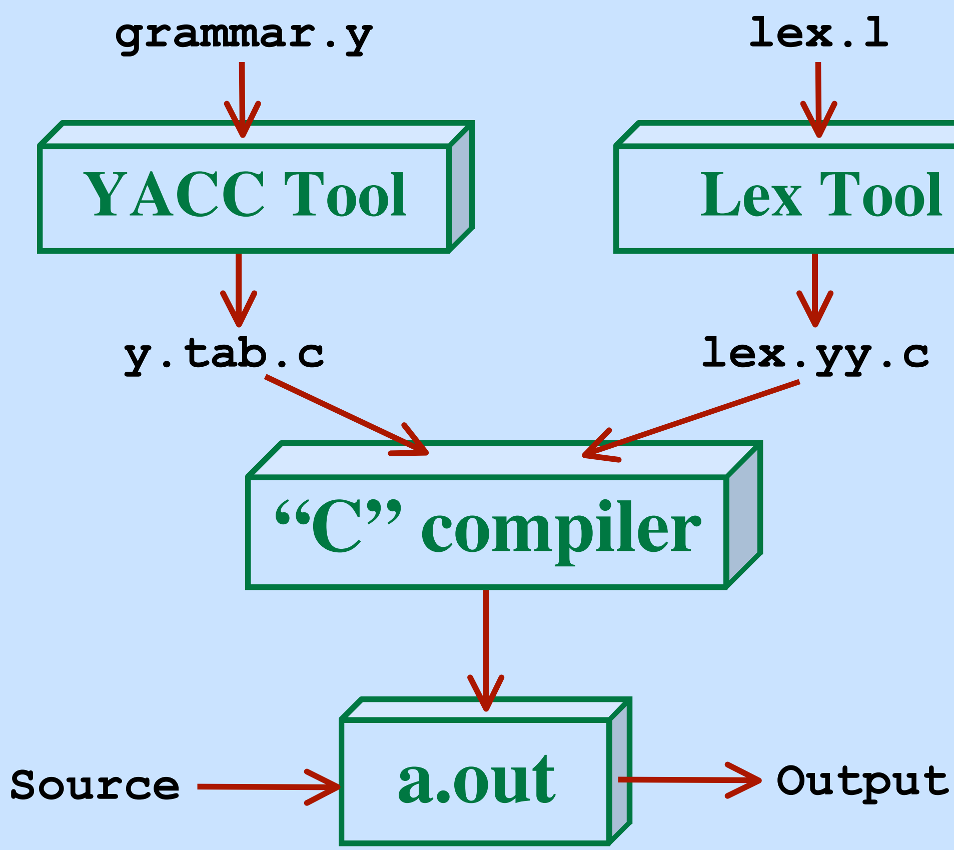
- *Perform the attibute computation*
- *Pop the stack*
- *Push the new non-terminal with its attribute.*

YACC

Yet Another Compiler Compiler

Unix tool to create an LALR parser.

Works with “Lex” tool: Calls `yylex()` to get next token.



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
%%
```

This material copied as is to yy.tab.c

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

The token types (from lexer)

Epsilon

End of rule

The grammar rules:

$S \rightarrow S\ E$ /n
$\rightarrow S$ /n
$\rightarrow \varepsilon$
$E \rightarrow E + T$
$\rightarrow T$
$T \rightarrow T * F$
$\rightarrow F$
$F \rightarrow$ NUMBER
$\rightarrow (E)$

This material copied as is to yy.tab.c

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:

$$\underset{\$\$}{E} \rightarrow \underset{\$1}{E} \; \underset{\$2}{+} \; \underset{\$3}{T}$$

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

How the $ Notation in YACC Works

$Expr_0 \rightarrow Expr_1 + Term$

`$$=$1` `$1` `$2` `$3`

```
Expr0.t = Expr1.t + Term.t;
$$ = $1 + $3;
```