Chapter 4. Syntax Analysis II
(Bottom-Up Parsers)

Param Venkat Vivek Kesireddy
pkesireddy@fullerton.edu

Inputs from Prof Doina Bein & Prof James Choi

4.1 Introduction

Disadvantages of Top-Down parser:

- Remove left recursion and thus changing the grammar.
- Hard to generate the intermediate code

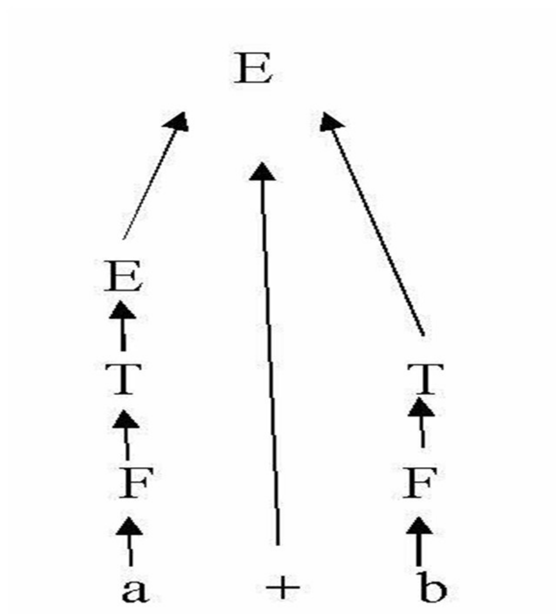
Q: is there a better way of parsing without changing the grammar?

A: Yes, parse the string bottom up
(Start from the bottom of the tree and build up)

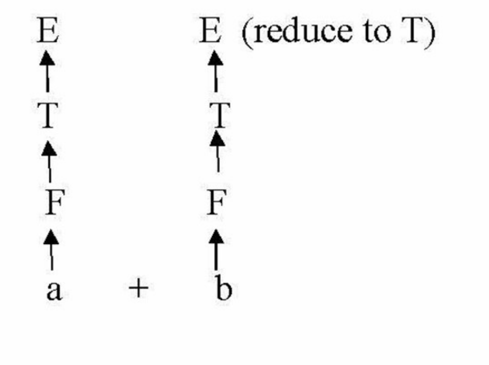
Ex: Let's consider the following grammar:

R1) E -> E + T
R2) E -> T
R3) T -> T * F
R4) T -> F
R5) F -> id

And parse the string a + b



But we also could have done this way!!



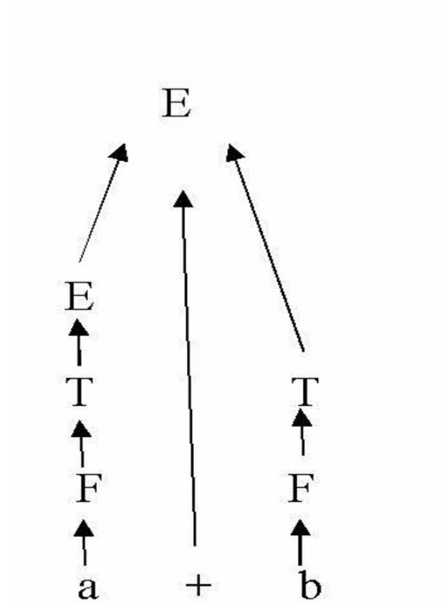
But it leads to nowhere

How do we know which Way???

Now let' consider a derivation
for a + b from right most
nonterminal (RMD)

R1) E -> E + T
R2) E -> T
R3) T -> T * F
R4) T -> F
R5) F -> id

E
1) => E + T
2) => E + F
3) => E + id (b)
4) => T + b
5) => F + b
6) => a + b



<u>RMD shows how to reduce in reverse</u>
It tells us that in (1) that we need to reduce according
to E -> E + T but NOT E -> T

<u>**Conclusion: We have to look at the RMD to see which reduction we need to apply In order to parse to bottom-up**</u>

R1) E -> E + T
R2) E -> T
R3) T -> T * F
R4) T -> F
R5) F -> id

Def: Handle

A handle is the **right hand side** of a production which reduced to get to the preceding step of the RMD

Ex.

E	
E + T	(handle: T or E+T?)
E + F	(F)
E + id (b)	(id)
T + b	(T)
F + b	(F)
a + b	(id)

	E	(RMD)
1) =>	E + T	
2) =>	E + F	
3) =>	E + id (b)	
4) =>	T + b	
5) =>	F + b	
6) =>	a + b	

So, bottom-up parser is about finding the handles and reduce until the top of the tree is found

Formal Def: Handle

For β to be a handle of the sentential form $\alpha\beta\omega$ (ω is reserved for all terminal symbols), we must have

$$\alpha B\omega \Rightarrow \alpha\beta\omega$$

$$S \Rightarrow^* \alpha B\omega$$

This implies that :

$B \rightarrow \beta$ exists and reduction leads to the preceding step of derivation RMD

4.2 Operator Precedence Parser

Simplest Bottom-up (BU) parser, however recognizes the smallest set of CFL. It uses Precedence table, stack and driver to recognize a sentence. The table consists of precedence relations such as

<. .= .>

<u>Ex. For the following grammar</u>

1) E -> E + E
2) E -> E * E
3) E -> id

stack \ token	+	*	Id	$
+	.>	<.	<.	.>
*	.>	.>	<.	.>
id	.>	.>		.>
$	<.	<.	<.	

Driver:

- Push $ onto the stack
- Append $ at the end of the input string
- Repeat

```
    Let   t = top most TERMINAL symbol in the stack and
          i = the incoming token
    Find Table [t, i]
    If NO entry then error
    else if   i .> t then
          push   <.   and  i  onto the stack  ( skip all NTs for   <.)
    else {    /* t .> i */
          found handle delimited by   <.   and  .>
           if no RHS match to the handle, then error
    else push LHS of the handle onto the stack
     }
Until  (t = $ and  i = $) or error found
```

stack \ token	+	*	Id	$
+	.>	<.	<.	.>
*	.>	.>	<.	.>
id	.>	.>		.>
$	<.	<.	<.	

1) E -> E + E
2) E -> E * E
3) E -> id

Ex. Parse a+b*c using OPP

Stack	Compare	Input	Production used
$	<.	a+b*c$	
$<.a	.>	+b*c$	E -> id
$E	<.	+b*c$	
$<.E+	<.	b*c$	
$<.E+<.b	.>	*c$	E -> id
$<.E+ E	<.	*c$	
$<.E+<E*	<.	c$	
$<.E+<.E*<.c	.>	$	E -> id
$<.E+<.E*E	.>	$	E -> E * E
$<.E + E	.>	$	E -> E + E
$E		$	Finished

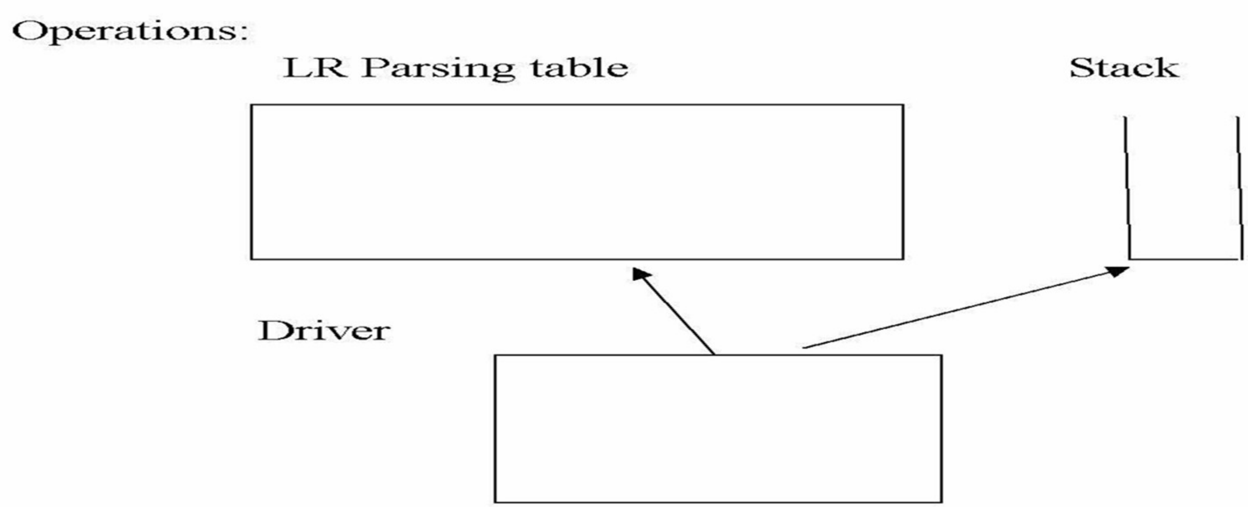
OPP Evaluation:

Advantage: fast and simple to use

Main Disadvantages:

- Cannot have two consecutive NT on the RHS of the production
- No ε production is allowed

4.3 LR parsers
(L = Left-to-right token scan, R= Right most derivation)



LR paring Table: (consists of 3 parts)

States	Terminals…………(Action Part)	NTs………(Goto Part)

Productions:

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

Action Part | GOTO Part

State	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				ACCT			
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			

4 kind of entries:

1. Sn (Shift)
2. Rn (Reduce)
3. N (goto)
4. Acct (Accept)

Driver:

Place $ at the end of the input string
Push state 0 on to the stack
Repeat
    Let qm be the current state (TOS state) and i the token
    Find x = Table [Qm, i];
    Case x of
        S(Qn) : Push (i) and enter qn, i.e., push (Qn);
        R(n): Reduce by production #n by popping 2x # of RHS symbols
            Let Qj be the TOS state
            Push the LHS L onto the stack
            Push Qk = Table [Qj, L] onto the stack
        ACCT: Parsing is complete
        Empty: error condition
Until ACCT or Error

Productions:

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

State	id	+	*	(	)	$	E	T	F
	Action Part						GOTO Part		
0	S5			S4			1	2	3
1		S6				ACCT			
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			

Example of Parsing a+b

Stack	Input	Table Entry	Action
0	a+b$	S5	Push(a);push(5)
0a5	+b$	R6	F-> id; Table[0,F]=3
0F3	+b$	R4	T -> F; Table[0,T] = 2
0T2	+b$	R2	E -> T; Table[0,E] = 1
0E1	+b$	S6	Push(+),push(6)
0E1+6	b$	S5	Push(b);push(5)
0E1+6b5	$	R6	F->id; Table[6,F]= 3
0E1+6F3	$	R4	T->F; Table[6,T] = 9
0E1+6T9	$	R1	E -> E+T;Table[0,E]=1
0E1	$	ACCT	

Question: How do we construct such as Parsing Table?

Need 3 Concepts:

Concept 1: Item

Item is a production with a marker . inserted somewhere on its RHS enclosed in [………].

E.g.
Consider the production E -> E + T then
[E -> .E + T], [E -> E. + T], [E -> E + .T], [E -> E + T.] are items.

Especially:
[E -> .E + T] is called the "Initial Item"
[E -> E + T.] is called the "Completed Item

Concept 2: Closure (I)

If I is a set of items, then Closure(I) is given by

1) I ϵ Closure (I)
2) If [A -> α.Bγ] is in the Closure(I) so far and B->β then [B -> .β] is in the Closure(I)

E.g. Consider the following "Expression Grammar"
E -> E +T, E -> T, T-> T*F, T -> F, F -> (E), F -> id

Find

- Closure (I = { [T -> .T *F]}) = { [T -> .T *F], [T -> .T * F], [T -> .F], [F -> .id], [F -> .(E)] } = = { [T -> .T * F], [T -> .F], [F -> .id], [F -> .(E)] }

- Closure ([E -> E .+T]) = { [E -> E .+ T] }

- Closure ([T -> T * .F], [F-> id.]) = { [T -> T * .F], [F -> id.], [F-> . (E)], [F -> .id] }

Concepts 3: Transition N(I, X)

If I is a set of items and X is ANY grammar symbol (T or NT), then N(I, X) is defined to be Closure set of all items [A -> α X.γ] such that [A -> α .Xγ] is in I.

E.g.,

N (I = {[E -> .E + T]}, E) = { [E -> E . + T]}

N (I= {[T-> T .* F]} , *) = { [T -> T *.F], [F -> .id], [F -> .(E)] }

N (I = { [T -> T * . F]} , T) = { }

4.3.1 Simple LR

Now we are ready to construct the Parsing Table
Step1) Augment the Grammar
Step2) Construct the Transition sets
Step3) Construct the Transition Table
Step4) write out the table

E.g., Expression Grammar:
R1. E -> E + T
R2. E -> T
R3. T -> T * F
R4. T -> F
R5. F -> (E)
R6. F -> id

R1. E -> E + T
R2. E -> T
R3. T -> T * F
R4. T -> F
R5. F -> (E)
R6. F -> id

Step1) Augment the Grammar

R0: E' –> E (needed whenever more than one production for the Starting Symbol)

Step 2) Construct the Transition Sets

i0 = State 0 = Def. Closure of Initial item of R0 = ([E' → .E])
= { [E' → .E], [E → .E+T], [E → .T], [T → .T*f], [T → .F], [F → .(E)], [F → .id] }

Transitions from State 0:

i1 = N (i0, E) = { [E' → E.], [E → E.+T] }
i2 = N (i0, T) = { [E → T.], [T → T.*F] }
i3 = N (i0, F) = { [T → F.] }
i4 = N (i0, ( ) = { [F → (.E)], [E → .E+T], [E → .T], [T → .T*F], [T → .F],
[F → .(E)], [F → .id] }
i5 = N (i0, id) = { [F → id.] }

R0. E’ -> E
R1. E -> E + T
R2. E -> T
R3. T -> T * F
R4. T -> F
R5. F -> (E)
R6. F -> id

<u>Transitions from State 1</u> = { [E’ → E.], [E → E.+T] }
i6 = N (i1, +) = { [E → E+.T], [T → .T*F], [T → .F], [F → .(E)], [F → .id] }

<u>Transitions from State 2</u> = { [E → T.], [T → T.*F] }
i7 = N (i2, *) = { [T → T*.F], [F → .(E)], [F → .id] }

<u>Transition from State 3</u> = { [T → F.] }
= { }

<u>Transitions from State 4</u> = { [F → (.E)], [E → .E+T], [E → .T], [T → .T*F],
[T → .F], [F → .(E)], [F → .id] }
i8 = N (i4, E) = { [F → (E.)], [E → E.+T] }
N (i4, T) = { [E → T.], [T → T.*F] } = i2
N (i4, F) = { [T → F.] } = i3
N (i4, ( ) = i4
N (i4, id) = { [F → id.] } = i5

R0. E' -> E
R1. E -> E + T
R2. E -> T
R3. T -> T * F
R4. T -> F
R5. F -> (E)
R6. F -> id

Transitions from State 6 = { [E → E+.T], [T → .T*F], [T → .F], [F → .(E)], [F → .id] }
i9 = N (i6, T) = { [E → E+T.], [T → T.*F] }
N (i6, F) = { [T → F.] } = i3
N (i6, ( ) = i4
N (i6, id) = i5

Transitions from State 7 = { [T → T*.F], [F → .(E)], [F → .id] }
i10 = N (i7, F) = { [T → T*F.] }
N (i7, ( ) = i4
N (i7, id) = i5

Transitions from State 8 = { [F → (E.)], [E → E.+T] }
i11 = N (i8,)) = { [F → (E).] }
N (i8, +) = i6

Transitions from State 9 = { [E → E+T.], [T → T.*F] }
N (i9, *) = { [T → T*.F], [F → .(E)], [F → .id] } = i7

No transitions from i10 & i11 => finished

Step3) Construct Transition Table

i1 = N (i0, E) = { [E’ → E.], [E’ → E.+T] }
i2 = N (i0, T) = { [E → T.], [T → T.*F] }
i3 = N (i0, F) = { [T → F.] }
i4 = N (i0, ( ) = { [F → (.E)], [E → .E+T], [E → .T], [T → .T*F], [T → .F], [F → .(E)], [F → .id] }
i5 = N (i0, id) = { [F → id.] }
i6 = N (i1, +) = { [E → E+.T], [T → .T*F], [T → .F], [F → .(E)], [F → .id] }
i7 = N (i2, *) = { [T → T*.F], [F → .(E)], [F → .id] }
i8 = N (i4, E) = { [F → (E.)], [E → E.+T] }
N (i4, T) = { [E → T.], [T → T.*F] } = i2, N (i4, F) = i3, N (i4, ( ) = i4, N (i4, id) = i5
i9 = N (i6, T) = { [E → E+T.], [T → T.*F] }, N (i6, F) = i3, N (i6, ( ) = i4, N (i6, id) = i5
i10 = N (i7, F) = { [T → T*F.] }, N (i7, ( ) = i4, N (i7, id) = i5
i11 = N (i8,)) = { [F → (E).] }, N (i8, +) = i6, N (i9, *) = i7

State	Id	+	*	(	)	$	E	T	F
0	5			4			1	2	3
1		6							
2			7						
3									
4	5			4			8	2	3
5									
6	5			4				9	3
7	5			4					10
8		6			11				
9			7						
10									
11									

Step 4) Write out the table

1) Change the entry m in the Action Part to "Sm"

2) If a state contains a completed item for Rn then

For each symbol in the follow set of LHS

Put the Reduction rule Rn

(*) Special Case: If a state Q contains the item [S' -> S.]

completed item for R0, then the entry for [Q, $] is "ACCT"

1) Change m to Sm in Action Part

State	Id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6							
2			S7						
3									
4	S5			S4			8	2	3
5									
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9			S7						
10									
11									

i1 = N (i0, E) = { [E’ → E.], [E’ → E.+T] } (Special case)
i2 = N (i0, T) = { [E → T.], [T → T.*F] }
i3 = N (i0, F) = { [T → F.] }
i4 = N (i0, ( ) = { [F → (.E)], [E → .E+T], [E → .T], [T → .T*F], [T → .F], [F → .(E)], [F → .id] }
i5 = N (i0, id) = { [F → id.] }
i6 = N (i1, +) = { [E → E+.T], [T → .T*F], [T → .F], [F → .(E)], [F → .id] }
i7 = N (i2, *) = { [T → T*.F], [F → .(E)], [F → .id] }
i8 = N (i4, E) = { [F → (E.)], [E → E.+T] }
i9 = N (i6, T) = { [E → E+T.] , [T → T.*F] }
i10 = N (i7, F) = { [T → T*F.] }
i11 = N (i8,)) = { [F → (E).] }

R0. E’ -> E
R1. E -> E + T
R2. E -> T
R3. T -> T * F
R4. T -> F
R5. F -> (E)
R6. F -> id

2) Find for completed item in each state

- **State 2 has a completed item [E -> T.] for R2**
  Follow (E) = {+, $,)} => Table [2, {+,$,)}] = R2
- **State 3 has a completed item [T -> F.] for R4**
  Follow (T) = { *, +,), $} => Table[3, { *, +,), $}] = R4
- **State 5 has a completed item [F -> id.] for R6**
  Follow(F) = { *, +,), $} => Table[5, { *, +,), $}] = R6
- **State 9 has a completed item [E -> E+T.] for R1**
  Follow (E) = {+, $,)} => Table [9, {+,$,)}] = R1
- **State 10 has a completed item [T -> T*F.] for R3**
  Follow (T) = { *, +,), $} => Table[10, { *, +,), $}] = R3
- **State 11 has a completed item [F -> (E).] for R5**
  Follow(F) = { *, +,), $} => Table[11, { *, +,), $}] = R5

(*) State 1 has a completed item [E ‘ -> E.] => Table [1, $] = ACCT

Result:

State	Id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				ACCT			
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			

Ex 2)
Consider the following Grammar

R0) S' -> S
R1) S -> E = E
R2) S -> id
R3) E -> E + id
R4) E -> id

Step2) Transition Sets

I0 = Closure ([S' -> .S]) = { [S' ->.S], [S -> .E = E], [S -> .id], [E -> . E + id], [E -> .id]}
I1 = N (I0, S) = { [S' -> S.]}
I2 = N (I0, E) = { [S -> E . = E], [E -> E. + id]}
I3 = N (I0, id) = { [S -> id.], [E -> id.] }
I4 = N (I2, =) = { [S -> E = .E], [E ->. E + id], [E -> .id] }
I5 = N (I2, +) = { [E -> E + .id] }
I6 = N (I4, E) = { [S -> E = E.], [E -> E . + id]}
I7 = N (I4, id) = { [E -> id.]}
I8 = N (I5, id) = { [E -> E + id.] }
N (I6, +) = { [E -> E + .id] } = I5

I0 = Closure ([S' -> .S]) = { [S' ->.S], [S -> .E = E], [S -> .id],
[E -> . E + id], [E -> .id]}
I1 = N (I0, S) = { [S' -> S.]}
I2 = N (I0, E) = { [S -> E . = E], [E -> E. + id]}
I3 = N (I0, id) = { [S -> id.], [E -> id.] }
I4 = N (I2, =) = { [S -> E = .E], [E ->. E + id], [E -> .id] }
I5 = N (I2, +) = { [E -> E + .id] }
I6 = N (I4, E) = { [S -> E = E.], [E -> E . + id]}
I7 = N (I4, id) = { [E -> id.]}
I8 = N (I5, id) = { [E -> E + id.] }
I5 = N (I6, +) = { [E -> E + .id] }

Step 3) Transition Table

	Id	=	+	$	S	E
0	3				1	2
1						
2		4	5			
3						
4	7					6
5	8					
6			5			
7						
8						

Step4) Write out the Table
1) m => Sm in Action Part
2) Just want to consider tor State 3
•State 3 = { [S -> id.], [E -> id.] } has two completed items:
[S -> id.] for R2 and [E -> id.] for R4
Compute Follow (S) = {$} and Follow (E) = {=, +, $}
=> Table [3, $] = R2 and Table[3, {=, +, $}] = R4

R0) S' -> S
R1) S -> E = E
R2) S -> id
R3) E -> E + id
R4) E -> id

	Id	=	+	$	S	E
0	S3				1	2
1				ACCT		
2		S4	S5			
3		R4	R4	R2/R4		
4	S7					6
5	S8					
6			S5			
7						
8						

➢There is a conflict in Table[3, $] = {R2/R4} Called Reduce/Reduce Conflict.
That is, if the stack has State 3 and input is $, there are two possibilities R2 and R4

	Id	=	+	$	S	E
0	S3				1	2
1				ACCT		
2		S4	S5			
3		R4	R4	(R2/R4)		
4	S7					6
5	S8					
6			S5			
7						
8						

Let's examine further: Consider a string "a" and try to parse

Stack	Input	Action
0	a$	S3
0a3	$	R2 or R4 (Choose R2) S -> id
0S1	$	ACCT

Stack	Input	Action
0	a$	S3
0a3	$	R2 or R4 (Choose R4) E -> id
0E2	$	Error????

<u>What does it mean ? => R4 should NOT be allowed in the first place</u>
<u>The conflict occurred because the technique we utilized is not powerful enough. We need a more powerful method</u>
<u>Canonical LR</u>

4.3.2) Canonical LR

Item: $[A \rightarrow \beta.\gamma\ ;\ l]$ a.k.a <u>Canonical Item</u>,
with core $A \rightarrow \beta.\gamma$ and l = look ahead set (LAS).
The idea is : Reduce for the complete items in LAS, rather than the follow set.

<u>Modifications from SLR:</u>

1. Our Starting State I0 = [S' -> . S; {$}]
2. For each item $[A \rightarrow \alpha.B\gamma;\ l]$ in the closure set do
   For each production $B \rightarrow .\beta$ do
      Create the initial item $[B \rightarrow .\beta;\ t]$ with $t = \bigcup_{x \in l} \mathrm{First}(\gamma x)$
   If the cores are the same, merge items by merging the LASs
3. Reduce for completed items in each state for tokens in the LAS rather than Follow set

Example: Construct CLR from the following productions.

Step1) Augument the grammar with R0

R0) S' -> S R1) S -> E = E R2) S -> id R3) E -> E + id R4) E -> id

Step 2) Transition Sets for LR states

I0 = Closure { [S' -> .S; {$}]} = { [S' -> .S;{$}], [S -> .E=E; {$}], [S ->. id; {$}],[E -> .E+id; {=}], [E -> .id; {=}], [E ->.E+id;{+}], [E->.id;{+}]}

= { [S'-> .S;{$}], [S->.E=E;{$}],[S->.id;{$}],[E->.E+id;{=,+}], [E->.id;{+,=}]}

Transitions from I0
I1 = N(I0, S) = { [S' -> S.];{$}] }
I2 = N(I0, E) = { [S -> E.=E; {$}], [E -> E.+id;{+, =}]}
I3 = N(I0, id) = { [S -> id.;{$}], [E -> id.; { =,+}] }

Transitions from I2
I4 = N(I2, =) = { [S -> E=.E; {$}], [E -> .E+id; {$}], [E -> .id; {$}], [E ->.E+id;{+}], [E->.id;{+}]}
= { [S -> E=.E; {$}], [E -> .E+id; {+, $}], [E -> .id; {+, $}]}

I5 = N(I2,+) = { [E -> E+.id; {=, +}]

<u>Transitions from I4</u>

I6 = N(I4, E) = {[S -> E=E.; {$}], [E -> E.+id; {+, $}]}

I7 = N(I4, id) = { [E -> id.; {+, $}]}

<u>Transitions from I5</u>

I8 = N(I5, id) = { [E -> E+id.; {=, +}]

<u>Transitions from I6</u>

I9 = N(I6, +) = {[E -> E+.id; {+, $}]}

<u>Transitions from I9</u>

I10 = N(I9, id) = {[E -> E+id.; {+, $}]}

Step 3) Transition Table

	Id	=	+	$	S	E
0	3				1	2
1						
2		4	5			
3						
4	7					6
5	8					
6			9			
7						
8						
9	10					
10						

Step 4) Write out the table

1) m -> Sm in the action part

2) Let's just consider:

• State 3: { [S -> id.;{$}], [E -> id.; { =,+}] } has two completed items.

[S -> id.] for R2 with l = {$} => Table [3, $] = R2

[E -> id.] for R4 with l = {+, =} => Table [3, {+, =}] = R4

	Id	=	+	$	S	E
0	S3				1	2
1				ACCT		
2		S4	S5			
3		R4	R4	R2		
4	S7					6
5	S8					
6			S9			
7			R4	R4		
8		R2	R2			
9	S10					
10			R3	R3		

•**The result: The conflict in state 3 is GONE!!**

Observations:

1. More states than SLR (machine got bigger) : from 9 to 11
2. However, some LR state are similar (i.e., same cores) with different LAS

I5 = { [E -> E + .id; {+, =}]
I9 = { [E -> E + .id; {+, $}]

Q: Can we make machine smaller? (same number of states as SLR)
Yes, LALR (Look Ahead LR)

4.3.3) LALR Parser

The simplest way to construct the LALR states is:

1) Find CLR states
2) Merge all states (S1, S2… Sn) having same core sets by taking the unions of all LASs and say that the resulting state is Q

Example: of our grammar

I5 = { [E −> E + .id; {=, +}] } and I9 = { [E −> E + .id; {$, +}] } can be merged

I5' = { [E −> E + .id; {=, +, $}] }

I8 = { [E -> E+id.; {=, +}] } and I10 = {[E -> E+id.; {+, $}]} can be merged

I8' = { [E −> E + id.; {=, +, $}] }

So it gives us 9 LALR states in total, same as SLR states

4.4 Error Handling

If there is a syntax error, your parser should do

a) Generate a “meaningful” error message

b) Recover from the error

a) Error message

Assume LR parser, current state is N and token = I, then
Table[N, i] = empty is an error.

In this case, we could look at the Table [N, x] = Not empty and announce all x tokens

State	Id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				ACCT			
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			

Ex. Current state = 5 and token (=> error => look for entry

b) Error recovery is more "tricky"

One way to do is to insert a fake token e.g., a(b+c) => will generate an error message of missing an operator => Insert any operator and move on

State	Id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				ACCT			
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			

Ex. Current state = 5 and token (=> fake token +, −, *

But what if the parser assumes wrong => But what if the parser assumes wrong => (e.g., state 4, multiple different entries =>
Will generate message which will be hard to understand

4.5 Symbol Table

A Database for symbol that occur during the compilation process.
It also provides set of procedures such as, lookup(), insert(), remove(), list() etc

<u>4.5.1 Organization of the table</u>

• Array :
Adv: Simplest and no overhead (pointers)
Disadv: fixed size, search O(N)

•Linked List
Adv: Expandable
DisAdv: Overhead (Pointers), search O(N)

- Binary Search Tree

Adv: Expandable, Search O(logN)
Disavd: Overhead (2 pointers), House-keeping for Binary Search Tree

- Hash Table (Most compiler use this) = Bucket + LL

Adv: Expandable, Search O(λ) = average bucket size
If hash function h(x) is good the bucket size will be good.
Some h(x) = mode function, middle square function etc

4.5.2 Scope issues

To make variables visible correctly

- No Scope => All variables are global (Basic)
- Totally Separate => All local = No nesting is allowed
- Nested scope : Nesting allowed

How Symbol table handles the scopes:

- No scope: All names (variables must be distinct)
- Totally Separate: Same names are allowed if scopes are different

- Nest Scope (many PLs)

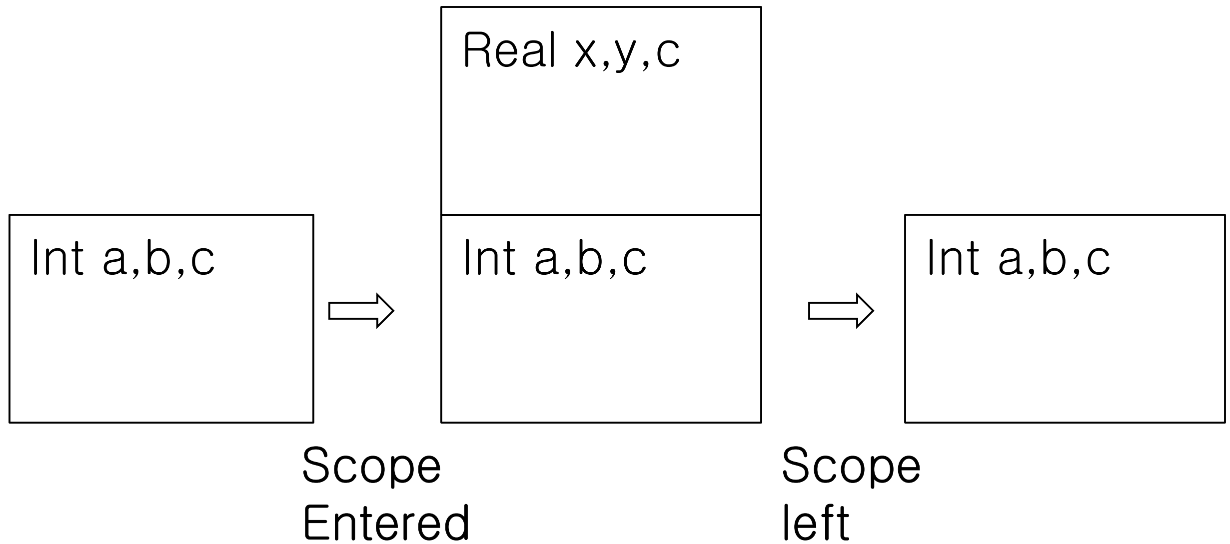
Use of dynamic (growing and shrinking) symbol table, i.e.,
 If a scope is entered the table grows by the symbols of the scope
 And if scope is left the table shrinks (STACK)

Example:

```
{
    int a,b, c;
    …….
        {
            real x,y,c;
            …..
            c= /* real */

        }

    …c = /* int */
}
```



END