COMPILER (CSE 4120)

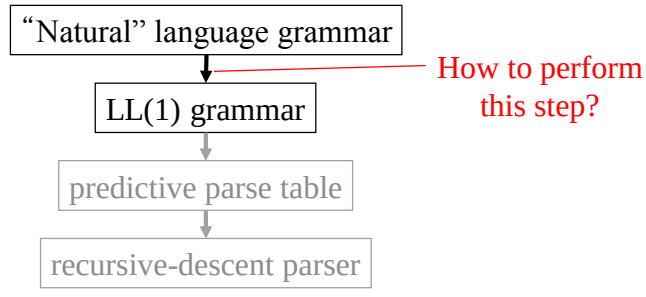
(Lecture 5: Parsing 3 – Bottom-up Parsing)

Sungwon Jung

Mobile Computing & Data Engineering Lab
Dept. of Computer Science and Engineering
Sogang University
Seoul, Korea
Tel: +82-2-705-8930
Email : jungsung@sogang.ac.kr

Review

- Can make recursive descent parsers for LL(1) grammars



Bottom-up Parsing

- A more powerful parsing technology
- LR grammars -- more expressive than LL
  - can handle left-recursive grammars, virtually all programming languages
  - More natural expression of programming language syntax
- Shift-reduce parsers
  - automatic parser generators (*e.g.* yacc)
  - detect errors as soon as possible
  - allows better error recovery

Bottom-up Parsing

- Right-most derivation -- backward
  - Start with the tokens
  - End with the start symbol

(1+2+(3+4))+5 ← (***E***+2+(3+4))+5 ← (***S***+2+(3+4))+5 ←
(S+***E***+(3+4))+5 ← (***S***+(3+4))+5 ← (S+(***E***+4))+5 ←
(S+(***S***+4))+5 ← (S+(S+***E***))+5 ← (S+(***S***))+5 ← (S+***E***)+5
← (***S***)+5 ← ***E***+5 ← ***S***+5 ← *S*+***E*** ← ***S***

Progress of bottom-up parsing

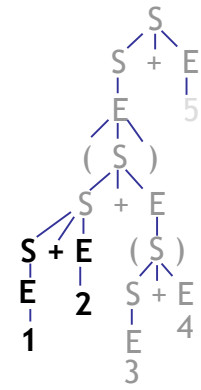
right-most derivation

(1+2+(3+4))+5 ←	(1+2+(3+4))+5
(*E*+2+(3+4))+5 ←	(1 +2+(3+4))+5
(*S*+2+(3+4))+5 ←	(1 +2+(3+4))+5
(S+*E*+(3+4))+5 ←	(1+2 +(3+4))+5
(*S*+(3+4))+5 ←	(1+2+(3 +4))+5
(S+(*E*+4))+5 ←	(1+2+(3 +4))+5
(S+(*S*+4))+5 ←	(1+2+(3 +4))+5
(S+(S+*E*))+5 ←	(1+2+(3+4))+5
(S+(*S*))+5 ←	(1+2+(3+4))+5
(S+*E*)+5 ←	(1+2+(3+4))+5
(*S*)+5 ←	(1+2+(3+4))+5
E+5 ←	(1+2+(3+4)) +5
S+*E* ←	(1+2+(3+4))+5
S	(1+2+(3+4))+5

Bottom-up Parsing

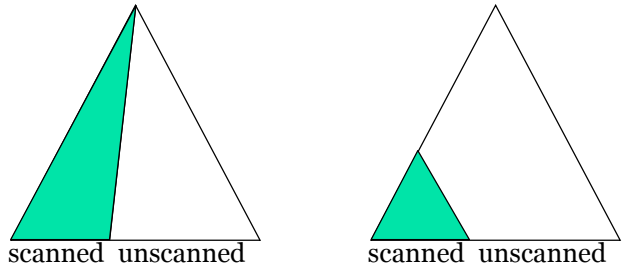
$S \rightarrow S + E \mid E$
$E \rightarrow \text{number} \mid (S)$

- (1+2+(3+4))+5 ←(*E*+2+(3+4))+5 ←(*S*+2+(3+4))+5 ←(S+*E*+(3+4))+5 …
- **Advantage of bottom-up parsing: can select productions based on more information**



Top-down vs. Bottom-up

- Bottom-up: Don't need to figure out as much of the parse tree for a given amount of input



Top-down Bottom-up

Shift-reduce parsing

- Parsing is a sequence of *shift* and *reduce* operations
- Parser state is a stack of terminals and non-terminals (grows to the right)
- Unconsumed input is a string of terminals
- Current derivation step is always stack+input

Derivation step	stack	unconsumed input
(1+2+(3+4))+5 ←		(1+2+(3+4))+5
(*E*+2+(3+4))+5 ←	(E	+2+(3+4))+5
(*S*+2+(3+4))+5 ←	(S	+2+(3+4))+5
(S+*E*+(3+4))+5 ←	(S+E	+(3+4))+5

Shift-reduce parsing

- Parsing is a sequence of *shifts* and *reduces*
- **Shift** -- move look-ahead token to stack

stack	input	action
(	1+2+(3+4))+5	*shift* 1
(1	+2+(3+4))+5	

- **Reduce** -- Replace symbols γ in top of stack with non-terminal symbol X, corresponding to production X → γ (pop γ, push X)

stack	input	action
(S+E	+(3+4))+5	*reduce* S→ S+E
(S	+(3+4))+5	

Dept. of Computer Science and Engineering at Sogang University

Shift-reduce Parsing

$S \rightarrow S + E \mid E$
$E \rightarrow \text{number} \mid (S)$

derivation	stack	input stream	action
(1+2+(3+4))+5 ←		(1+2+(3+4))+5	*shift*
(1+2+(3+4))+5 ←	(	1+2+(3+4))+5	*shift*
(1+2+(3+4))+5 ←	(1	+2+(3+4))+5	*reduce* E→num
(E+2+(3+4))+5 ←	(E	+2+(3+4))+5	*reduce* S → E
(S+2+(3+4))+5 ←	(S	+2+(3+4))+5	*shift*
(S+2+(3+4))+5 ←	(S+	2+(3+4))+5	*shift*
(S+2+(3+4))+5 ←	(S+2	+(3+4))+5	*reduce* E→num
(S+E+(3+4))+5 ←	(S+E	+(3+4))+5	*reduce* S→S+E
(S+(3+4))+5 ←	(S	+(3+4))+5	*shift*
(S+(3+4))+5 ←	(S+	(3+4))+5	*shift*
(S+(3+4))+5 ←	(S+(	3+4))+5	*shift*
(S+(3+4))+5 ←	(S+(3	+4))+5	*reduce* E→num

CSE 4120 Fundamentals of Compiler Construction -- Sungwon Jung

Problem

- How do we know which action to take --
  - whether to shift or reduce, and which production?
- Sometimes can reduce but shouldn't
  - e.g., $X \rightarrow \varepsilon$ can *always* be reduced
- Sometimes can reduce in different ways

Dept. of Computer Science and Engineering at Sogang University

Action Selection Problem

- Given stack σ and look-ahead symbol b, should we
  - **shift** b onto the stack (making it σb)
  - **reduce** some production $X \rightarrow \gamma$ assuming that stack has the form $\alpha\gamma$ (making it αX)
- If stack has form $\alpha\gamma$, should apply reduction $X \rightarrow \gamma$ depending on what stack prefix α is -- but α is different for different possible reductions, since γ's have different length.
  - How to keep track?

CSE 4120 Fundamentals of Compiler Construction -- Sungwon Jung

Parser States

- Goal: know what reductions are legal at any given point
- Idea: summarize all possible stack prefixes α as a parser *state*
- Parser state is defined by a DFA that reads in the stack α
- Accept states of DFA: unique reduction!
- Summarizing discards information
  - affects what grammars parser handles
  - affects size of DFA (number of states)

Dept. of Computer Science and Engineering at Sogang University

LR(0) parser

- **L**eft-to-right scanning, **R**ight-most derivation, "**zero**" look-ahead characters
- Too weak to handle most language grammars
- But will help us understand how to build better parsers

CSE 4120 Fundamentals of Compiler Construction -- Sungwon Jung

LR(0) items

- An LR(0) item is a string [α], where
  - α is a production form G with a **.** at some position in the rhs
  - The **.** indicates how much of an item we have seen at a given state in the parse
  - [A→ **.**XYZ] indicates that the parser is looking for a string that can be derived from XYZ
  - [A→ XY**.**Z] indicates that the parser has seen a string derivable from XY and is looking for one derivable from Z
  - A→ XYZ generates 4 LR(0) items

    1. [A→ **.**XYZ] 2. [A→ X**.**YZ]
    3. [A→ XY**.**Z] 4. [A→ XYZ**.**]

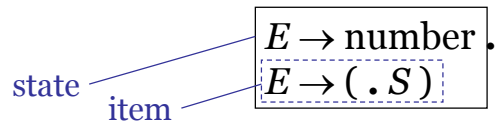
An LR(0) grammar: non-empty lists

- $S \rightarrow (L)$
  $S \rightarrow id$
  $L \rightarrow S$
  $L \rightarrow L , S$

- Examples:
  - x (x,y) (x, (y,z), w)
  - ((((x)))) (x, (y, (z, w)))

LR(0) states

- A state is a set of *items*
- An *LR(0) item* is a production from the language with a separator "**.**" somewhere in the RHS of the production



- Stuff before "**.**" already on stack (beginnings of possible γ's to be reduced)
- Stuff after "**.**" : what we might see next
- The prefixes α represented by state itself

$S \rightarrow (L) \mid id$
$L \rightarrow S \mid L, S$

Start State & Closure

DFA start state $S' \rightarrow . S \$$ closure ⇒ $S' \rightarrow . S \$$, $S \rightarrow . (L)$, $S \rightarrow . id$

- First step: augment grammar with production $S' \rightarrow S \$$
- Start state of DFA: empty stack = $S' \rightarrow . S \$$
- ***Closure*** of a state adds items for all productions whose LHS occurs in an item in the state, just after "**.**"
  - set of possible productions to be reduced next
  - Added items have the "**.**" located at the beginning: no symbols for these items on the stack yet

Closure (I)

- Given an item $[A \rightarrow \alpha.B\beta]$, its closure contains the item and any other items that can generate legal substrings to follow α.
- Thus, if the parser has α on its stack, the input should reduce to $B\beta$ (or γ for some other item $[B \rightarrow .\gamma]$ in the closure).

Dept. of Computer Science and Engineering at Sogang University

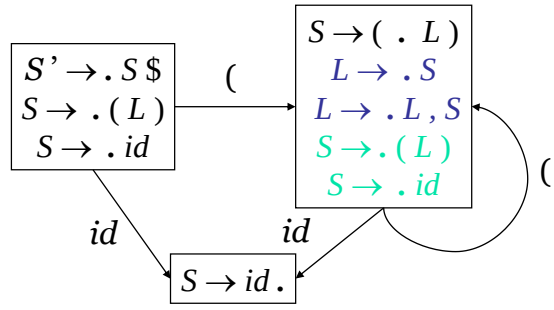
GOTO (I, X)

- Goto(I, X)
  - is the closure of the set of all items $[A \rightarrow \alpha X.\beta]$ such that $[A \rightarrow \alpha.X\beta] \in I$
  - goto(I, X) represents state after recognizing X in state I

CSE 4120 Fundamentals of Compiler Construction -- Sungwon Jung

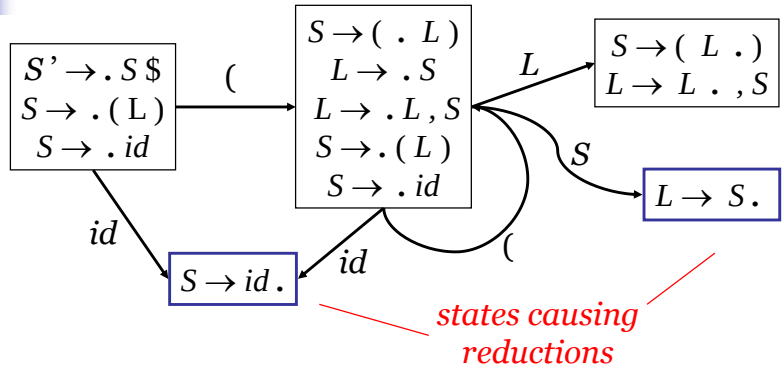
Applying symbols

$S \rightarrow (L) \mid id$
$L \rightarrow S \mid L , S$



- In new state, include all items that have appropriate input symbol just after dot, and advance dot in those items (*and take closure.*)

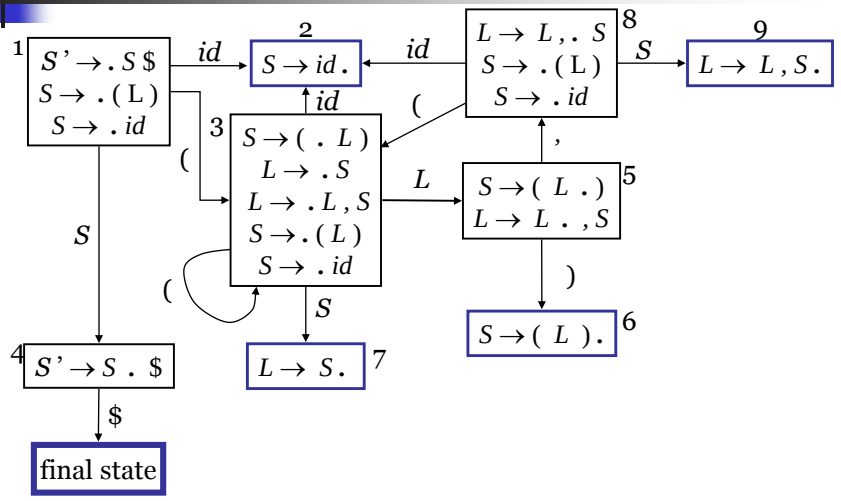
Applying reduce actions



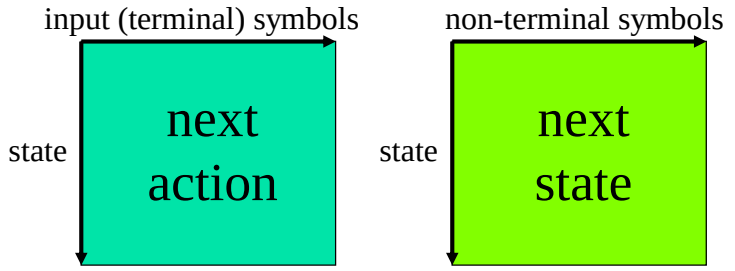
- Pop RHS off stack, replace with LHS X (X→γ), rerun DFA (e.g. (x))

$S \rightarrow (L) \mid id$
$L \rightarrow S \mid L, S$

Full DFA



Implementation: LR parsing table



Action table
Used at every step to decide whether to shift or reduce

Goto table
Used only when reducing, to determine next state

Construction of a parsing table

- For each edge $I \rightarrow J$ where I and J are LR(0) states,
  - If X is a terminal,
    - Put the action *shift* J at position (I,X) of the table
  - If X is a nonterminal,
    - Put *goto* J at position (I,X) of the table
- For each state I containing $S' \rightarrow S.\$$,
  - Put an *accept* action at $(I,\$)$ of the table
- For a state I containing an item $A \rightarrow \gamma .$ (production n with the dot at the end),
  - Put a *reduce n* action at (I,Y) of the table for every terminal token Y

Terminologies for parser table

- Elements in LR parsing table are labeled with the following four kinds of actions:
  - s*n*: Shift into state n;
  - g*n*: Goto state n;
  - r*k*: Reduce by rule k;
  - a: Accept;
  - blank: Error;
- If action is:
  - Shift(*n*): Advance input one token; push n on stack;
  - Reduce(*k*): if rule *k* is $A \rightarrow \beta$ then
    - Pop $2 \cdot |\beta|$ symbols off the stack ($|\beta|$ state and $|\beta|$ grammar symbols)
    - Let *s* be the state on the top of the stack
      - Push *A* symbol on top of the stack
      - Push GOTO(*s*,*A*) on top of the stack
  - Accept: Stop parsing and report success
  - Error: Stop parsing and report failure

r1: $S \rightarrow (L)$
r2: $S \rightarrow id$
r3: $L \rightarrow S$
r4: $L \rightarrow L, S$

List Grammar Parser Table

	(	)	*id*	,	$	*S*	*L*
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					accept		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

$S \rightarrow (L) \mid id$
$L \rightarrow S \mid L, S$

Let's try parsing ((x),y)

derivation	***stack***	***input***	***action***
((x),y) ←	$_1$	((x),y)$	shift, goto 3
((x),y) ←	$_1 (_3$	(x),y)$	shift, goto 3
((x),y) ←	$_1 (_3 (_3$	x),y)$	shift, goto 2
((x),y) ←	$_1 (_3 (_3 x_2$	),y)$	reduce $S \rightarrow id$
((***S***),y) ←	$_1 (_3 (_3 S_7$	),y)$	reduce $L \rightarrow S$
((***L***),y) ←	$_1 (_3 (_3 L_5$	),y)$	shift, goto 6
((*L*),y) ←	$_1 (_3 (_3 L_5)_6$	,y)$	reduce $S \rightarrow (L)$
(***S***,y) ←	$_1 (_3 S_7$	,y)$	reduce $L \rightarrow S$
(***L***,y) ←	$_1 (_3 L_5$	,y)$	shift, goto 8
(*L*,y) ←	$_1 (_3 L_5 ,_8$	y)$	shift, goto 9
(*L*,y) ←	$_1 (_3 L_5 ,_8 y_2$	)$	reduce $S \rightarrow id$
(*L*,***S***) ←	$_1 (_3 L_5 ,_8 S_9$	)$	reduce $L \rightarrow L , S$
(***L***) ←	$_1 (_3 L_5$	)$	shift, goto 6
(***L***) ←	$_1 (_3 L_5)_6$	$	reduce $S \rightarrow (L)$
S	$_1 S_4$	$	*done*

LR(0) Limitations

- An LR(0) machine only works if states with reduce actions have a *single* reduce action -- in those states, *always* reduce ignoring input
- With more complex grammar, construction gives states with shift/reduce or reduce/reduce conflicts
- Need to use look-ahead to choose

ok

$L \rightarrow L\,,S\,.$

shift /reduce

$L \rightarrow L\,,S\,.$
$S \rightarrow S\,.\,,L$

reduce / reduce

$L \rightarrow S\,,L\,.$
$L \rightarrow L\,.$

List grammar parse table

	(	)	id	,	$	S	L
1	s3		s2			4	
2	*S→id*	*S→id*	*S→id*	*S→id*	*S→id*		
3	s3		s2			7	5
4					accept		
5		s6		s8			
6	*S→(L)*	*S→(L)*	*S→(L)*	*S→(L)*	*S→(L)*		
7	*L→S*	*L→S*	*L→S*	*L→S*	*L→S*		
8	s3		s2			9	
9	*L→L,S*	*L→L,S*	*L→L,S*	*L→L,S*	*L→L,S*		

How about sum grammar?

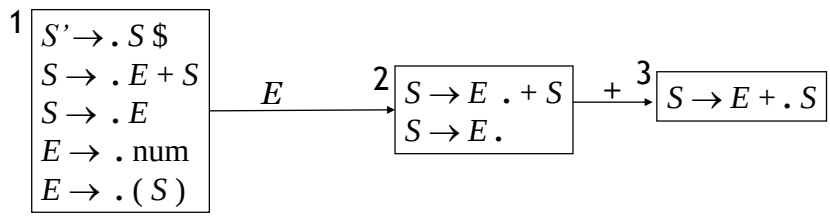
$$S \rightarrow S + E \mid E$$
$$E \rightarrow \text{num} \mid (\ S\ )$$

- This is LR(0)
- Right-associative version isn't:

$$S \rightarrow E + S \mid E$$
$$E \rightarrow \text{num} \mid (\ S\ )$$

LR(0) construction

$$S \rightarrow E + S \mid E$$
$$E \rightarrow \text{num} \mid (\ S\ )$$



What to do on +?

	+	$	E
1			2
2	s3/S→E	S→E	

SLR grammars

$S \rightarrow E + S \mid E$
$E \rightarrow \text{num} \mid (\ S\ )$

- Idea: Only add reduce action to table if look-ahead symbol is in the *FOLLOW* set of the non-terminal being reduced
- Eliminates some conflicts
- *FOLLOW*(S) = { $,) }
- Many language grammars are SLR

	+	$	E
1			2
2		s3	S→E

SLR(1) table construction

- Construct the collection of sets of LR(0) item for *G'*
- State *i* of the parser is constructed from I_i
  - (a) if $[A := \alpha.a\beta] \in I_i$ and goto(I_i,*a*) = I_j, then set ACTION [*i*,*a*] to *shift j*. (*a* must be a terminal)
  - (b) if $[A := \alpha.] \in I_i$ and then set ACTION [*i*,*a*] to *reduce* $A := \alpha$ for all *a* in FOLLOW(*A*)
  - (c) if $[S' := S.] \in I_i$ and then set ACTION [*i*,$] to accept
- If goto(I_i,*A*) = I_j for nonterminal *A*, then set GOTO[*i*, *A*] to *j*
- All other entries in ACTION and GOTO are set to "*error*"
- The initial state of the parser is the one constructed from the set containing the item $[S' := .S]$.

SLR(1) Parser: An Example

$E := T + E \mid T$
$T :=$ id

- The Augmented grammar
  0 S' := E
  1 E := T + E
  2 E := T
  3 T := id

FIRST(S') = {id} FOLLOW(S') = {$}
FIRST(E) = {id} FOLLOW(E) = {$}
FIRST(T) = {id} FOLLOW(T) = {+, $}

SLR(1) Parser: An Example (Cont'd)

- **LR(0) states**
  S_0: [S' := .E], [E := .T+E], [E := .T], [T := .id]
  S_1: [S' := E.]
  S_2: [E := T.+E], [E := T.]
  S_3: [T := id.]
  S_4: [E := T+.E], [E := .T+E], [E := .T], [T := .id]
  S_5: [E := T+E.]
- **GOTO function**
  - Iteration 1: goto(S_0, E) = S_1, goto(S_0, T) = S_2, goto(S_0, id) = S_3
  - Iteration 2: goto(S_2, +) = S_4
  - Iteration 3: goto(S_4, id) = S_3, goto(S_4, E) = S_5, goto(S_4, T) = S_2

LR(K) items

- An *LR*(*k*) item is a pair [α, β], where
  - α is a production from *G* with a **.** at some position in the *rhs*
  - β is a lookahead string containing *k* symbols (terminals or **$**)
- What about LR(1) items ?
  - Example LR(1) item: [A := X**.**YZ, **a**]
  - LR(1) items have lookahead strings of length 1
  - several LR(1) items may have the same core

    [A := X**.**YZ, **a**], [A := X**.**YZ, **b**]

    We represent this as [A := X**.**YZ, **a/b**]

LR(1) lookahead

- What's the point of all these lookahead symbols?
  - carry them along to allow us to choose correct reduction when there is any choice
  - lookaheads are bookkeeping, unless item has **.** at right end.
    - In [A := X**.**YZ, **a**], **a** has no direct use
    - In [A := XYZ**.**, **a**], **a** is useful
  - allows use of grammars that are not *uniquely invertible*
    - G is ***uniquely invertible*** if no two productions have the same *rhs*
    - For [A := α**.**, **a**] and [B := α**.**, **b**], we can decide between reducing to A and to B by looking at limited right context

Canonical LR(1) items

- The canonical collection of LR(1) items:
  - set of items derivable from [S' := **.**S, $]
  - set of all items that can derive the final configuration
- Each set in canonical collection of sets of LR(1) items represents a state in NFA that recognizes viable prefixes
- To construct the canonical collection we need two functions:
  - closure (I)
  - goto(I, X)

LR(1) closure

- Given an item [A := α**.**Bβ, **a**], its closure contains the item and any other items that can generate legal substrings to follow α.
- Thus, if the parser has viable prefix α on its stack, the input should reduce to Bβ (or γ for some other item [B := **.**γ, b] in the closure).
- To compute closure (I)

```
function closure (I)
repeat
   new_item = false
   for each item [A := α.Bβ, a] ∈ I,
      each production B := .γ ∈ G'  and each terminal b ∈ FIRST(βa),
      if [B := .γ, b] ∉ I then
         add [B := .γ, b] to I
         new_item = true
      endif
until (new_item = false)
return I
```

LR(1) goto

- Let I be a set of LR(1) items and X be a grammar symbol
- Then, goto(I, X) is the closure of the set of all items
  [A := αX.β, **a**] such that [A := α.**X**β, **a**] ∈ I
- goto(I, X) represents state after recognizing X in state I
- To compute goto(I, X)

function goto (I, X)
J = set of items [A := αX.β, **a**] such that [A := α.**X**β, **a**] ∈ I
J' = closure (J)
return J'

Collection of sets of LR(1) items

- We start the construction of the collection of sets of LR(1) items with the item [S' := .S, $], where
  - S' is the start symbol of the augmented grammar G'
  - S is the start symbol of G, and
  - $ is the right end of string marker
- To compute the collection of sets of LR(1) items

```
procedure item (G')
C = {closure({[S' := .S, $]})}
repeat
  new_item = false
  for each set of items I in C and each grammar symbol X such that
  goto(I, X) ≠ ∅ and goto(I, X) ∉ C
       add goto(I, X) to  C
       new_item = true
  endfor
until (new_item = false)
```

LR(1) table construction

- Construct the collection of sets of LR(1) item *G'*
- State *i* of the parser is the constructed from I_i
  - (a) if [$A := \alpha.\mathbf{a}\beta, \mathbf{b}$] $\in I_i$ and **goto**(I_i,**a**) = I_j ,
    then set **action**[*i*, **a**] to *shift j*. (**a** must be a terminal)
  - (b) if [$A := \alpha., \mathbf{a}$] $\in I_i$ and then set **action**[*i*,**a**] to *reduce* $A := \alpha$
  - (c) if [$S' := S., \mathbf{\$}$] $\in I_i$ and then set **action**[*i*,$] to accept
- If **goto**(I_i,*A*) = I_j , then set **goto**[*i*, *A*] to *j*
- All other entries in **action** and **goto** are set to "*error*"
- The initial state of the parser is the state constructed from the set containing the item [*S' :=* **.***S*, $].

An Example for LR(1) Grammar

- The grammar

1	G := E
2	E := T + E
3	E := T
4	T := F * T
5	T := F
6	F := id

	ACTION				GOTO		
	id	+	*	$	E	T	F
S0	s4				1	2	3
S1				acc			
S2		s5		r3			
S3		r5	s6	r5			
S4		r6	r6	r6			
S5	s4				7	2	3
S6	s4					8	3
S7				r2			
S8		r4		r4			

An Example for LR(1) Grammar (Cont'd)

- The grammar

  1 G := E
  2 E := T + E
  3 E := T
  4 T := F * T
  5 T := F
  6 F := id

- Step 1

  I0 = {[g := .e, $]}
  I0 = closure(I0) = { [g := .e,$], [e := .t+e, $], [e := .t, $], [t := .f*t, +], [t := .f*t, $], [t := .f, +], [t := .f, $], [f := .id, +], [f := .id, *], [f := .id, $]}
  Iteration 1: I1=goto(I0,e), I2=goto(I0,t), I3=goto(I0,f), I4=goto(I0,id)
  Iteration 2: I5=goto(I2,+), I6=goto(I3,*)
  Iteration 3: I7=goto(I5,e), I8=goto(I6,t)

Dept. of Computer Science and Engineering at Sogang University

An Example for LR(1) Grammar (Cont'd)

- I0: [g := .e, $], [e := .t+e, $], [e := .t, $], [t := .f*t, +/$], [t := .f, +/$], [f := .id, +/*/$]
- I1: [g := e., $]
- I2: [e := t., $], [e := t.+e, $]
- I3: [t := f., +/$], [t := f.*t, +/$]
- I4: [f := id., +/*/$]
- I5: [e := t+.e, $], [e := .t+e, $], [e := .t, $], [t := .f*t, +/$], [t:= .f, +/$], [f := .id, +/*/$]
- I6: [t := f*.t, +/$], [t := .f*t, +/$], [t := .f, +/$], [f := .id, +/*/$]
- I7: [e := t+e., $]
- I8: [t := f*t., +/$]

CSE 4120 Fundamentals of Compiler Construction -- Sungwon Jung