Compilers (CS31003)

Syntax Analysis or Parsing

Lecture 7-8

LL(1) parser

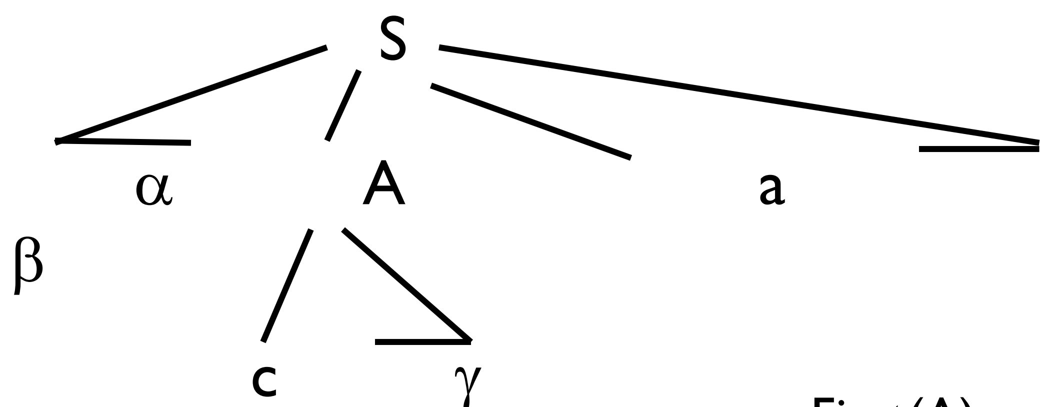
$$\begin{array}{llll} 0: & S & \rightarrow & E \\ 1: & E & \rightarrow & TE' \\ 2: & E' & \rightarrow & \varepsilon \end{array} \qquad \begin{array}{llll} 3: & E' & \rightarrow & +E \\ 4: & T & \rightarrow & FT' \\ 5: & T' & \rightarrow & \varepsilon \end{array} \qquad \begin{array}{llll} 6: & T' & \rightarrow & *T \\ 7: & F & \rightarrow & (E) \\ 8: & F & \rightarrow & \mathsf{Id} \end{array}$$

	(	)	+	*	Id	#
S	E	error	error	error	E	error
E	$(E)\ \langle E, F\rangle$	error	error	error	Id $\langle E, F\rangle$	error
T	$(E)\ \langle T, F\rangle$	error	error	error	Id $\langle T, F\rangle$	error
F	$(E)\ \langle F, F\rangle$	error	error	error	Id $\langle F, F\rangle$	error
$\langle E, F\rangle$	error	$\langle E, T\rangle$	$\langle E, T\rangle$	$\langle E, T\rangle$	error	$\langle E, T\rangle$
$\langle E, T\rangle$	error	$\langle E, E\rangle$	$\langle E, E\rangle$	$*F\ \langle E, T\rangle$	error	$\langle E, E\rangle$
$\langle E, E\rangle$	error	ε	$+T\ \langle E, E\rangle$	error	error	ε
$\langle T, F\rangle$	error	$\langle T, T\rangle$	$\langle T, T\rangle$	$\langle T, T\rangle$	error	$\langle T, T\rangle$
$\langle T, T\rangle$	error	ε	ε	$*F\ \langle T, T\rangle$	error	ε
$\langle F, F\rangle$	error	ε	ε	ε	error	ε

First and Follow

- First(α) is the set of terminals that begin strings derived from α, where α is any string of grammar symbols. If $\alpha \Rightarrow^* \varepsilon$ then ε is also in First(α).

- Follow(A), for a non-terminal A, is the set of terminals a that can appear immediately to the right of A in some sentential; that is, the set of terminals a such that there exists a derivation of the form $S \Rightarrow^* \alpha A a \beta$ for some α and β.



First(A)=c
Follow(A)=a

First and Follow

First(α):

1. First(X) = {X} if X is a terminal.
2. Add ε to First(X) if there exists X → ε.
3. If there is a production X → $Y_1Y_2Y_3..Y_k$, k≥1, then place a in First(X) if a is in First(Y_i) and $Y_1Y_2..Y_{i-1} \Rightarrow^* \varepsilon$.

Follow(A):

1. Follow(S)=$, where S is the start symbol and $ is the input right end marker.
2. If there is a production A → $\alpha B \beta$, then everything in First(β) except ε is in Follow(B).
3. If there is a production A → $\alpha B \mid \alpha B \beta$, where First($\beta$) contains ε then everything in Follow(A) is in Follow(B).

An example

stmt → if *expr* then *stmt*

| if *expr* then *stmt* else *stmt*

| other

Ambiguous

Make it unambiguous.

Item Pushdown Automata (IPDA)

$$
\begin{array}{lll}
(E) & \Delta([X \to \beta.Y\gamma], \varepsilon) & = \{[X \to \beta.Y\gamma][Y \to .\alpha] \mid Y \to \alpha \in P\} \\
(S) & \Delta([X \to \beta.a\gamma], a) & = \{[X \to \beta a.\gamma]\} \\
(R) & \Delta([X \to \beta.Y\gamma][Y \to \alpha.], \varepsilon) & = \{[X \to \beta Y.\gamma]\}.
\end{array}
$$

E/S/R ← Expanding/Shifting/Reducing Transition

Accepting the word *Id+Id*Id*

- $G_0' = (\{S,E,T,F\}, \{+,*,(,),Id\}, P_0', E)$
- P_0' $\quad S \rightarrow E \qquad E \rightarrow E + T \mid T$

  $T \rightarrow T * F \mid F$

  $F \rightarrow (E) \mid Id$

Pushdown	Remaining input
$[S \to .E]$	$\mathrm{Id} + \mathrm{Id} * \mathrm{Id}$
$[S \to .E][E \to .E+T]$	$\mathrm{Id} + \mathrm{Id} * \mathrm{Id}$
$[S \to .E][E \to .E+T][E \to .T]$	$\mathrm{Id} + \mathrm{Id} * \mathrm{Id}$
$[S \to .E][E \to .E+T][E \to .T][T \to .F]$	$\mathrm{Id} + \mathrm{Id} * \mathrm{Id}$
$[S \to .E][E \to .E+T][E \to .T][T \to .F][F \to .\mathrm{Id}]$	$\mathrm{Id} + \mathrm{Id} * \mathrm{Id}$
$[S \to .E][E \to .E+T][E \to .T][T \to .F][F \to \mathrm{Id}.]$	$+\mathrm{Id} * \mathrm{Id}$

Accepting the word *Id+Id*Id*

$[S \to .E][E \to .E + T][E \to .T][T \to F.]$	$+\text{Id} * \text{Id}$
$[S \to .E][E \to .E + T][E \to T.]$	$+\text{Id} * \text{Id}$
$[S \to .E][E \to E. + T]$	$+\text{Id} * \text{Id}$
$[S \to .E][E \to E + .T]$	$\text{Id} * \text{Id}$
$[S \to .E][E \to E + .T][T \to .T * F]$	$\text{Id} * \text{Id}$
$[S \to .E][E \to E + .T][T \to .T * F][T \to .F]$	$\text{Id} * \text{Id}$
$[S \to .E][E \to E + .T][T \to .T * F][T \to .F][F \to .\text{Id}]$	$\text{Id} * \text{Id}$
$[S \to .E][E \to E + .T][T \to .T * F][T \to .F][F \to \text{Id}.]$	$*\text{Id}$
$[S \to .E][E \to E + .T][T \to .T * F][T \to F.]$	$*\text{Id}$
$[S \to .E][E \to E + .T][T \to T. * F]$	$*\text{Id}$
$[S \to .E][E \to E + .T][T \to T * .F]$	Id
$[S \to .E][E \to E + .T][T \to T * .F][F \to .\text{Id}]$	Id
$[S \to .E][E \to E + .T][T \to T * .F][F \to \text{Id}.]$	
$[S \to .E][E \to E + .T][T \to T * F.]$	
$[S \to .E][E \to E + T.]$	
$[S \to E.]$	

LL(1)

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | Id

	id	+	*	(	)	$
E	E → T E'			E → T E'		
E'		E' → + T E'			E' → ε	E' → ε
T	T → F T'			T → F T'		
T'		T' → ε	T' → * F T'		T' → ε	T' → ε
F	F → Id			F → (E)		

LL(1)

	id	+	*
E	E → T E’		
E’		E’ → + T E’	
T	T → F T’		
T’		T’ → ε	T’ → * F T’
F	F → Id		

	(	)	$
E	E → T E’		
E’		E’ → ε	E’ → ε
T	T → F T’		
T’		T’ → ε	T’ → ε
F	F → (E)		

Matched	Stack	Input	Action
	E$	Id+Id*Id$	
	TE’$	Id+Id*Id$	Output E → TE’
	FT’E’$	Id+Id*Id$	Output T → FT’
	Id T’E’$	Id+Id*Id$	Output F → Id
Id	T’E’$	+Id*Id$	Match Id
Id	E’$	+Id*Id$	Output T’ → ε
Id	+TE’$	+Id*Id$	Output E’ →+TE’
Id+	TE’$	Id*Id$	Match +
Id+	FT’E’$	Id*Id$	Output T → FT’
Id+	Id T’E’$	Id*Id$	Output F → Id
Id+Id	T’E’$	*Id$	Match Id
Id+Id	* FT’E’$	*Id$	Output T’ → *FT’
Id+Id*	FT’E’$	Id$	Match *
Id+Id*	Id T’E’$	Id$	Output F → Id
Id+Id*Id	T’E’$	$	Match Id
Id+Id*Id	E’$	$	Output T’ → ε
Id+Id*Id	$	$	Output E’ → ε

Syntax error

- 1. Error is localized and reported.
- 2. Error is diagnosed.
- 3. Error is corrected.
- 4. Parser gets back to a state for further error detection.

A = B + (C + D * E ;

Should not go into endless loop while correcting errors.

Whenever the prefix ***u*** of a word has been analyzed without announcing an error, then there exists a word ***w*** such that ***uw*** is a word of the language.

Bottom-up Parser

- Read the next input symbol (*shift*)
- Reduce the right side of a production $X \rightarrow \alpha$ at the top of the pushdown by the left side X of the production (*reduce*).

Bottom-up parsing

- Bottom-up parsing:
  - The non-confirmed part of the prediction starts with a nonterminal.
  - It either reduce or shift to next input symbol.
  - $\gamma_1 A \gamma_2$ is reduced from $\gamma_1 \beta \gamma_2$ when $A \rightarrow \beta$ is a production rule.

Bottom-up parsing

Handle: A substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.

Right Sentential Form	Handle	Reducing Production
Id * Id	Id	F → Id
F * Id	F	T → F
T * Id	Id	F → Id
T * F	T * F	T → T * F
T	T	E → T

Shift-Reduce parsing

Shift / Reduce / Accept / Error

Stack	Input	Action
$	Id*Id$	Shift
$Id	*Id$	Reduce F → Id
$F	*Id$	Reduce T → F
$T	*Id$	Shift
$T*	Id$	Shift
$T*Id	$	Reduce F →Id
$T*F	$	Reduce T → T*F
$T	$	Reduce E → T
$E	$	Accept

Shift/Reduce conflict
Reduce/Reduce conflict

LR(k) parser

We call a CFG G an $LR(k)$-grammar, if in each of its rightmost derivations $S' = \alpha_0 \underset{rm}{\Longrightarrow} \alpha_1 \underset{rm}{\Longrightarrow} \alpha_2 \cdots \underset{rm}{\Longrightarrow} \alpha_m = v$ and each right sentential-forms α_i occurring in the derivation

- the handle can be localized, and
- the production to be applied can be determined

$$S' \overset{*}{\underset{rm}{\Longrightarrow}} \alpha X w \underset{rm}{\Longrightarrow} \alpha \beta w \quad \text{and}$$

$$S' \overset{*}{\underset{rm}{\Longrightarrow}} \gamma Y x \underset{rm}{\Longrightarrow} \alpha \beta y \quad \text{and}$$

$$w|_k = y|_k \qquad \text{implies} \qquad \alpha = \gamma \wedge X = Y \wedge x = y.$$

LR parsing

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

si ← shift and stack state i,
rj ← reduce by the production j,
Acc ← accept
Blank ← error

State	Action						Goto		
	Id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

LR parsing

(1) E → E + T

(2) E → T

(3) T → T * F

(4) T → F

(5) F → (E)

(6) F → Id

	Stack	**Symbol**	**Input**	**Action**
1	0		Id*Id+Id$	shift
2	0 5	Id	*Id+Id$	reduce F → Id
3	0 3	F	*Id+Id$	reduce T → F
4	0 2	T	*Id+Id$	shift
5	0 2 7	T*	Id+Id$	shift
6	0 2 7 5	T*Id	+Id$	reduce F → Id
7	0 2 7 10	T*F	+Id$	reduce T → T*F
8	0 2	T	+Id$	reduce E → T
9	0 1	E	+Id$	shift
10	0 1 6	E+	Id$	shift
11	0 1 6 5	E+Id	$	reduce F → Id
12	0 1 6 3	E+F	$	reduce T → F
13	0 1 6 9	E+T	$	reduce E → E+T
14	0 1	E	$	accept

Definitions

- Item, Kernel Items, Non-kernel Items
- Closure

  If I is a set of items for a grammar G, then CLOSURE(I) is the set of items constructed from I by:

  (i) Add every item in I to CLOSURE(I)

  (ii) $\forall\ A \rightarrow \alpha.B\beta \in$ CLOSURE(I) $\wedge\ B \rightarrow \gamma$

  Add $B \rightarrow .\gamma$ to CLOSURE(I) if it is not there.

- Action
- Goto

  GOTO(I,X) is defined to be the closure of the set of all items [$A \rightarrow \alpha X.\beta$] such that [$A \rightarrow \alpha.X\beta$] is in I.

LR(0) Automaton

I0: E'→.E
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id

I1: E'→E.
E→E.+T

I6: E→E+.T
T→.T*F
T→.F
F→.(E)
F→.id

I9: E→E+T.
T→T.*F

GOTO(*I*,+)

I2: E→T.
T→T.*F

I7: T→T*.F
F→.(E)
F→.id

I10: T→T*F.

I5: F→id.

I11: F→(E).

I4: F→(.E)
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id

I8: E→E.+T
F→(E.)

E' → E
E → E + T | T
T → T * F | F
F → (E) | id

I3: T→F.

Canonical LR (1) parsing table

Input: An augmented grammar G'.

Output: Canonical LR parsing table with <u>*Action*</u> and <u>*Goto*</u> for G'

Method:

1. Construct $C'=\{I_0,I_1, ..\}$, the collection of sets of LR(1) items for G'.

2. State i of the parser is constructed from I_i. The parsing action for state i is:

   (a) If $[A \rightarrow \alpha.a\beta, b]$ is in I_i and <u>*Goto*</u>$(I_i,a)=I_j$, then <u>*Action*</u>$[i,a]$ is "*shift j*". Here a must be a terminal.

   (b) If $[A \rightarrow \alpha., a]$ is in I_i, $A \neq S'$, then <u>*Action*</u>$[i,a]$ is "*reduce* $A \rightarrow \alpha$.".

   (c) If $[S' \rightarrow S., \$]$ is in I_i, then <u>*Action*</u>$[i,\$]$ is "*accept*".

3. The <u>*Goto*</u> transition for state i are constructed for all non-terminals A using the rule: If *Goto*$(I_i,A)=I_j$, then *Goto*$[i,A]=j$.

4. All blank entries are error.

5. Initial state is $[S' \rightarrow .S,\$]$.

LR(0)????
LR(2)?????

An example

S' → S
S → C C
C → c C |d

I0: S'→.S,$
S→.CC,$
C→.cC,c/d
C→.d,c/d

I1: S'→S.,$

I2: S→C.C,$
C→.cC,c/d
C→.d,$

I3: C→c.C,c/d
C→.cC,c/d
C→.d,c/d

I4: C→d.,c/d

I5: S→CC.,$

I6: C→c.C,$
C→.cC,$
C→.d,$

I7: C→d.,$

I8: C→cC.,c/d

I9: C→cC.,$

State	Action			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Error recovery in LR parsing

State	Action						Goto
	Id	+	*	(	)	$	E
0	s3	e1	e1	s2	e2	e1	1
1	e3	s4	s5	e3	e2	acc	
2	s3	e1	e1	s2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	s3	e1	e1	s2	e2	e1	7
5	s3	e1	e1	s2	e2	e1	8
6	e3	s4	s5	e3	s9	e4	
7	r1	r1	s5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

e1: Missing Operand
e2: Unbalanced right parenthesis
e3: Missing operator
e4: Missing right parenthesis.

Resolving conflicts

- Precedence
- Associativity