ΕΠΛ323 - Θεωρία και Πρακτική Μεταγλωττιστών

Lecture 7a

Syntax Analysis

Elias Athanasopoulos
eliasathan@cs.ucy.ac.cy

Operator-precedence Parsing

- A class of shift-reduce parsers that can be written by hand
- No ε-productions, no two adjacent non-terminals on the right side

$$E \rightarrow EAE \mid (E) \mid -E \mid \mathbf{id}$$
$$E \rightarrow + \mid - \mid * \mid / \mid \hat{}$$

X

Operator Grammar

$$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E\hat{}E \mid (E) \mid -E \mid \mathbf{id}$$

✓

Operation Relation Table

RELATION	MEANING
α <· β	α “yields precedence to” β
α $\doteq$ β	α “has the same precedence” β
α ·> β	α “takes precedence over” β

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

E → *E*+*E* | *E**E* | **id**

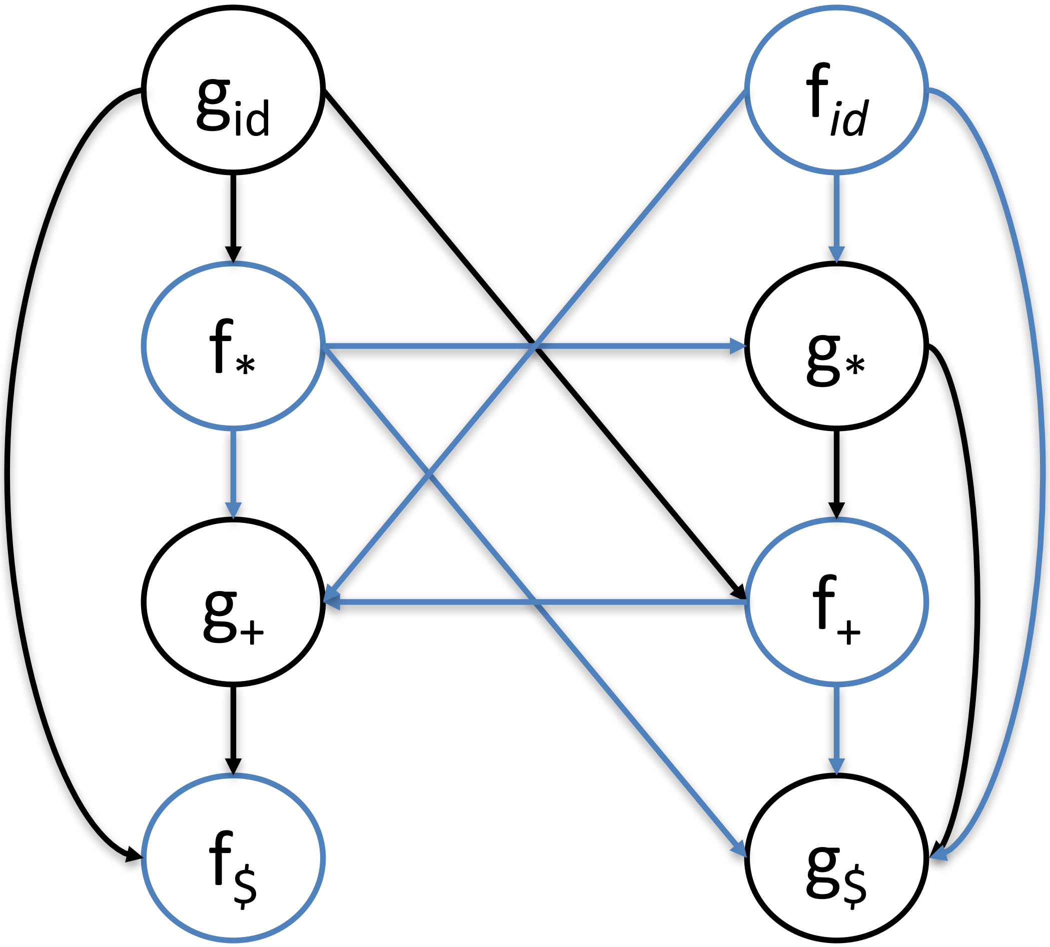
Example

STACK	INPUT	ACTION
$	**id**+**id***id**$	shift (push)
$**id**	+**id***id**$	reduce (pop)
$	+**id***id**$	shift (push)
$+	**id***id**$	shift (push)
$+**id**	***id**$	reduce (pop)
$+*	**id**$	shift (push)
$+***id**	$	shift (push)
$+*	$	reduce (pop)
$+	$	reduce (pop)
$	$	reduce (pop)

stack <· input: shift (push)
stack ·> input: reduce (pop)

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

Compression of Parsing Table



	+	*****	**id**	**$**
f	2	4	4	0
g	1	3	5	0

f \ g	**id**	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

Left-to-right scanning of the input.

Number of symbols for taking a decision (lookahead).

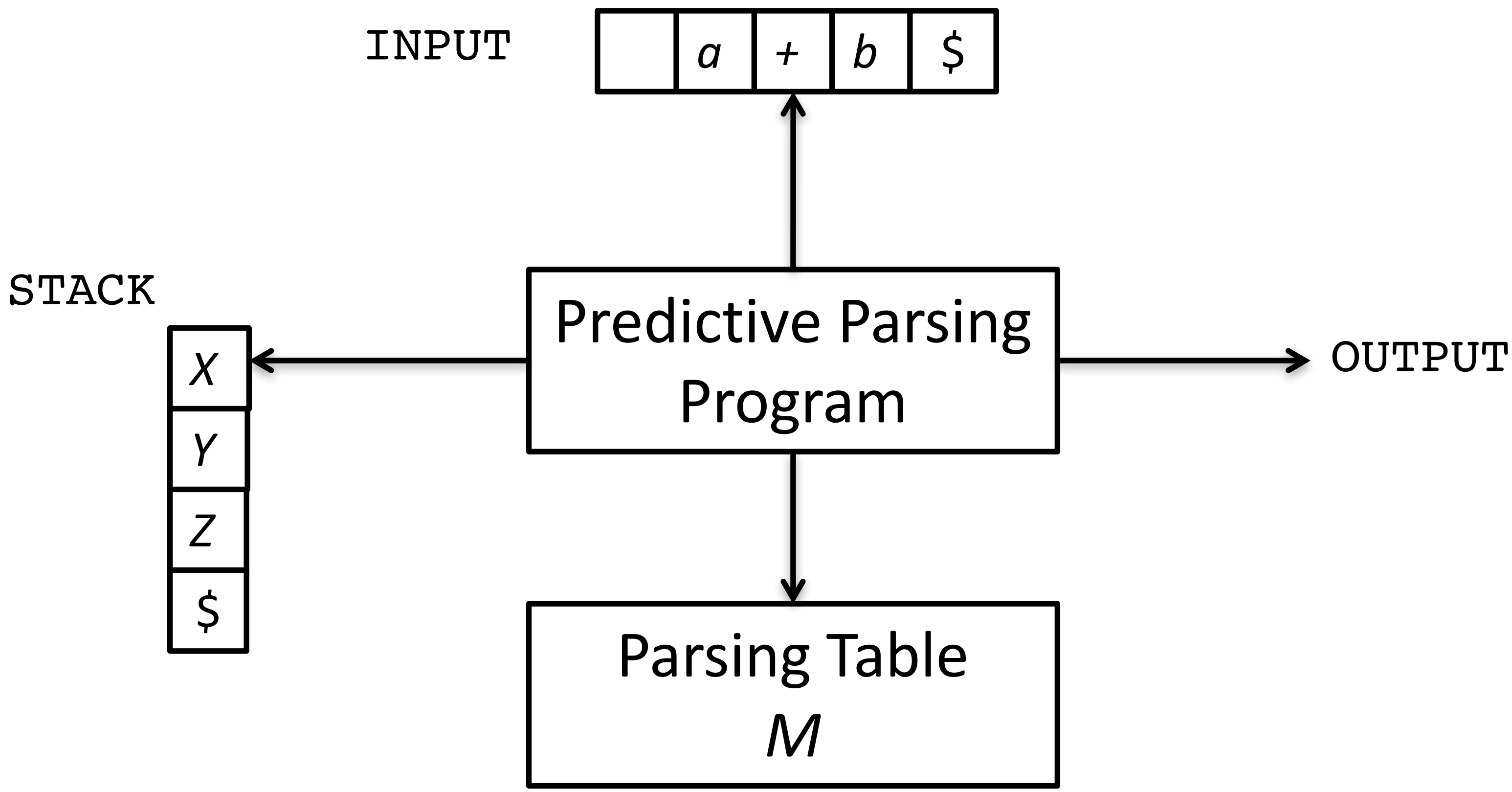
LR(*k*) parsers

Construction of a rightmost derivation in reverse

LR parsers

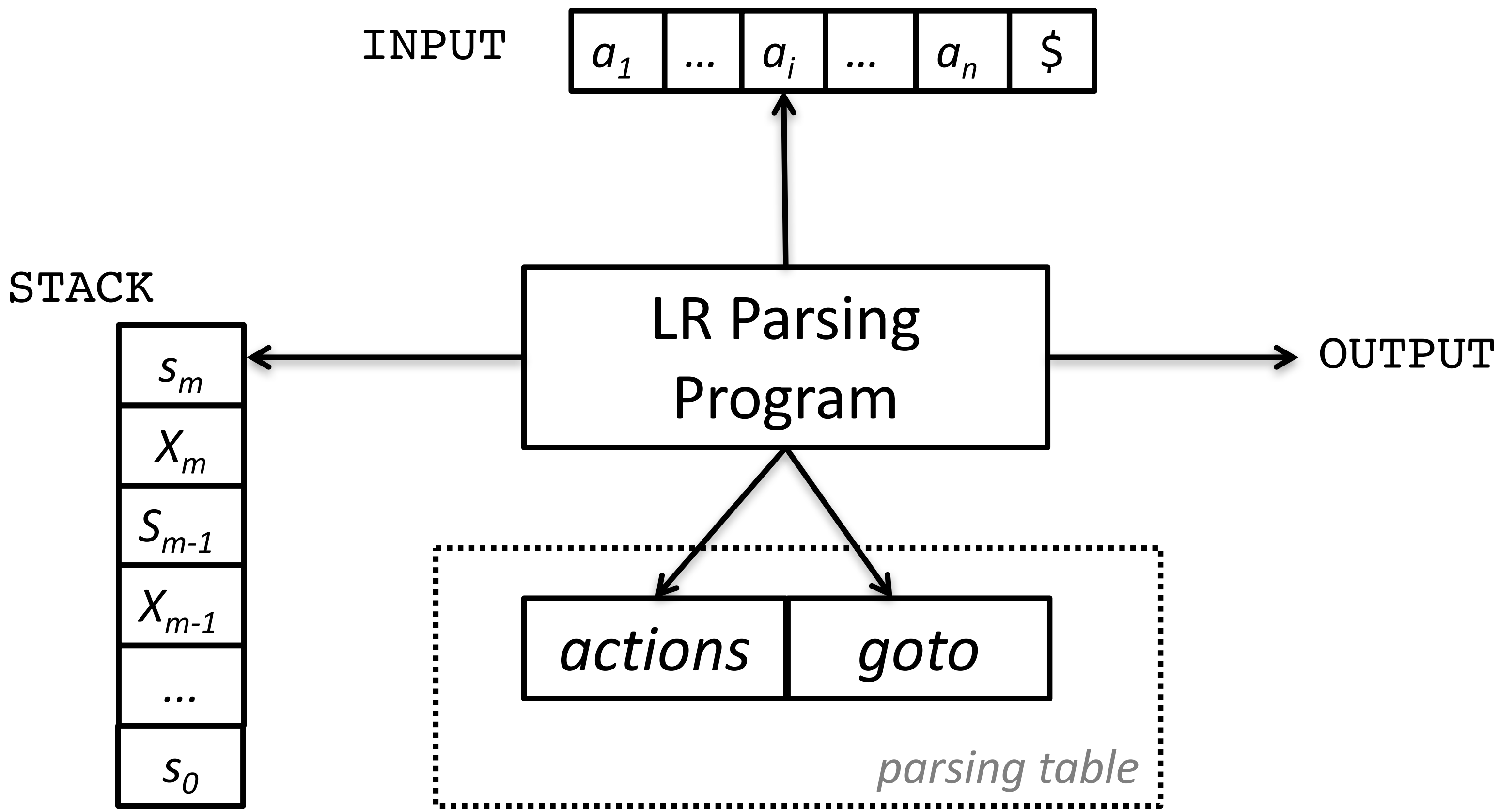
- LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written.
- The LR parsing method is the most general nonbacktracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.
- The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers (e.g., LL(1)).
- An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

Recall LL(1)



$: end symbol
X, Y, Z: non-terminals or terminals

LR parser



Algorithm (for *action*[s_m, a_i])
1. shift *s*, where *s* is state
2. reduce by a grammar production
3. accept, and
4. error.

LR Parsers

LR(0)

SLR(1)
(Simple LR)

LALR(1)
(Look Ahead LR)

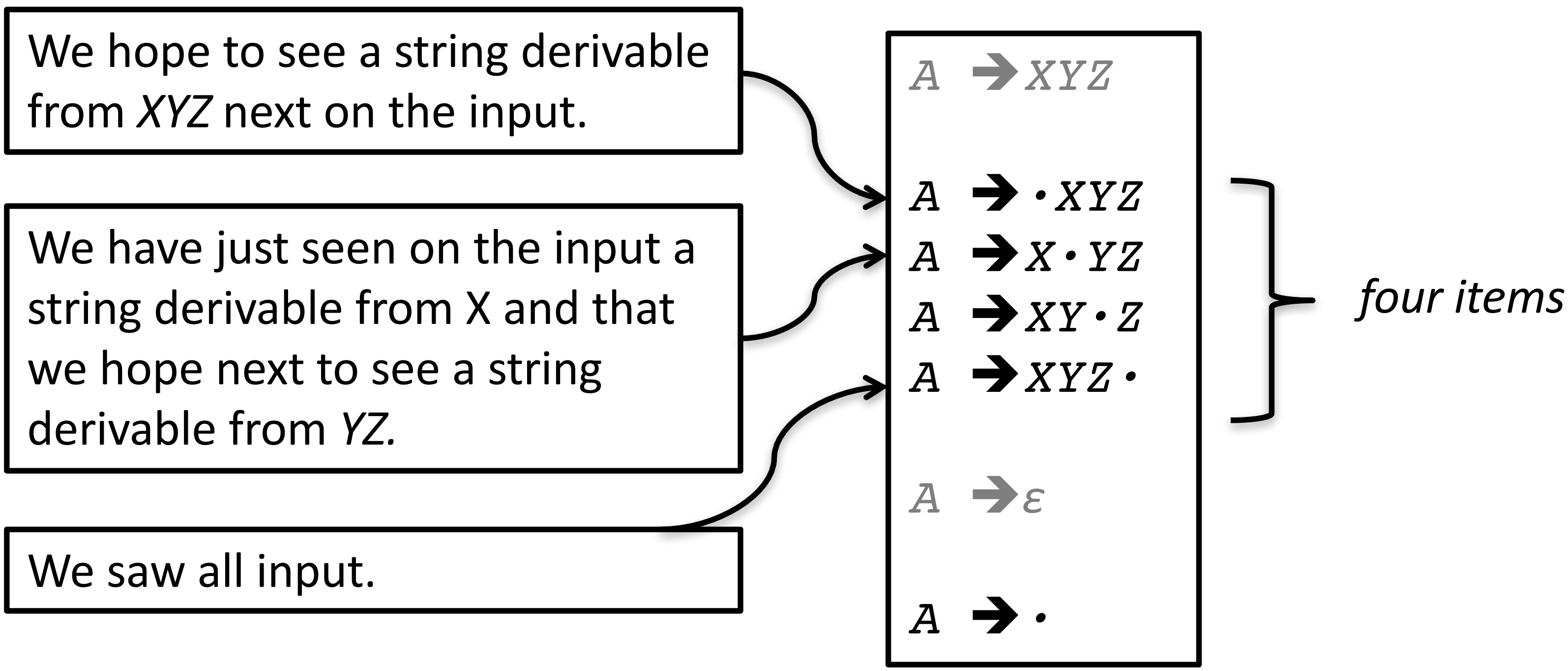
CLR(1)
(Canonical LR)

Power →

← **Simplicity**

Constructing an SLR parsing table

LR(0) item



We hope to see a string derivable from *XYZ* next on the input.
We have just seen on the input a string derivable from X and that we hope next to see a string derivable from *YZ.*
We saw all input.
$A \rightarrow XYZ$
$A \rightarrow \cdot XYZ$
$A \rightarrow X \cdot YZ$
$A \rightarrow XY \cdot Z$
$A \rightarrow XYZ \cdot$
four items
$A \rightarrow \varepsilon$
$A \rightarrow \cdot$

Closure

If *I* is a set of items for a grammar *G*, then *closure*(*I*) is the set of items constructed from *I* by the two rules:

- Initially, every item in *I* is added to the *closure*(*I*).
- If $A \rightarrow \alpha \cdot B\beta$ is in closure(I) and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \cdot \gamma$ to *I*, if it is not already there. We apply this rule until no more new items can be added to *closure*(*I*).

Example

Grammar

$E' \rightarrow E$

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid \textbf{id}$

$I = \{[E' \rightarrow \cdot E]\}$

$closure(I)$

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E+T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T*F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot \textbf{id}$

Goto

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

```
Grammar
E′→E
E →E+T|T
T →T*F|F
F →(E)|id
```

```
I = {[E′→E·], [E→E·+T]}
goto(I,+)
E →E+·T
T →·T*F
T →·F
F →·(E)
F →·id
```

How *goto*(*I*,+) is computed?
We computed *goto*(*I*, +) by examining *I* for items with + immediately to the right of the dot. $E' \rightarrow E\cdot$ is not such an item, but $E \rightarrow E\cdot+T$ is. We moved the dot over the + to get $\{E \rightarrow E+\cdot T\}$ and then took the closure of this set.

Canonical collection of LR(0) items

- Augment the grammar with a new symbol that produces the starting symbol of the grammar: $S' \rightarrow S$
- Compute the closure of the new production, C := *closure*($\{[S' \rightarrow \cdot S]\}$)
- For each set of items I in C, and each grammar symbol X, add *goto*(I, X) to C

Canonical collection of LR(0) items

I_0
$E' \rightarrow \cdot E$
$E \rightarrow \cdot E+T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T*F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (\mathrm{E})$
$F \rightarrow \cdot \mathbf{id}$

I_1
$E' \rightarrow E\cdot$
$E \rightarrow E\cdot +T$

I_2
$E \rightarrow T\cdot$
$T \rightarrow T\cdot *F$

I_3
$T \rightarrow F\cdot$

I_4
$F \rightarrow (\cdot E)$
$E \rightarrow \cdot E+T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T*F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (\mathrm{E})$
$F \rightarrow \cdot \mathbf{id}$

I_5
$F \rightarrow \mathbf{id}\cdot$

I_6
$E \rightarrow E+\cdot T$
$T \rightarrow \cdot T*F$
$T \rightarrow \cdot F$
$T \rightarrow \cdot (E)$
$T \rightarrow \cdot \mathbf{id}$

I_7
$T \rightarrow T*\cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot \mathbf{id}$

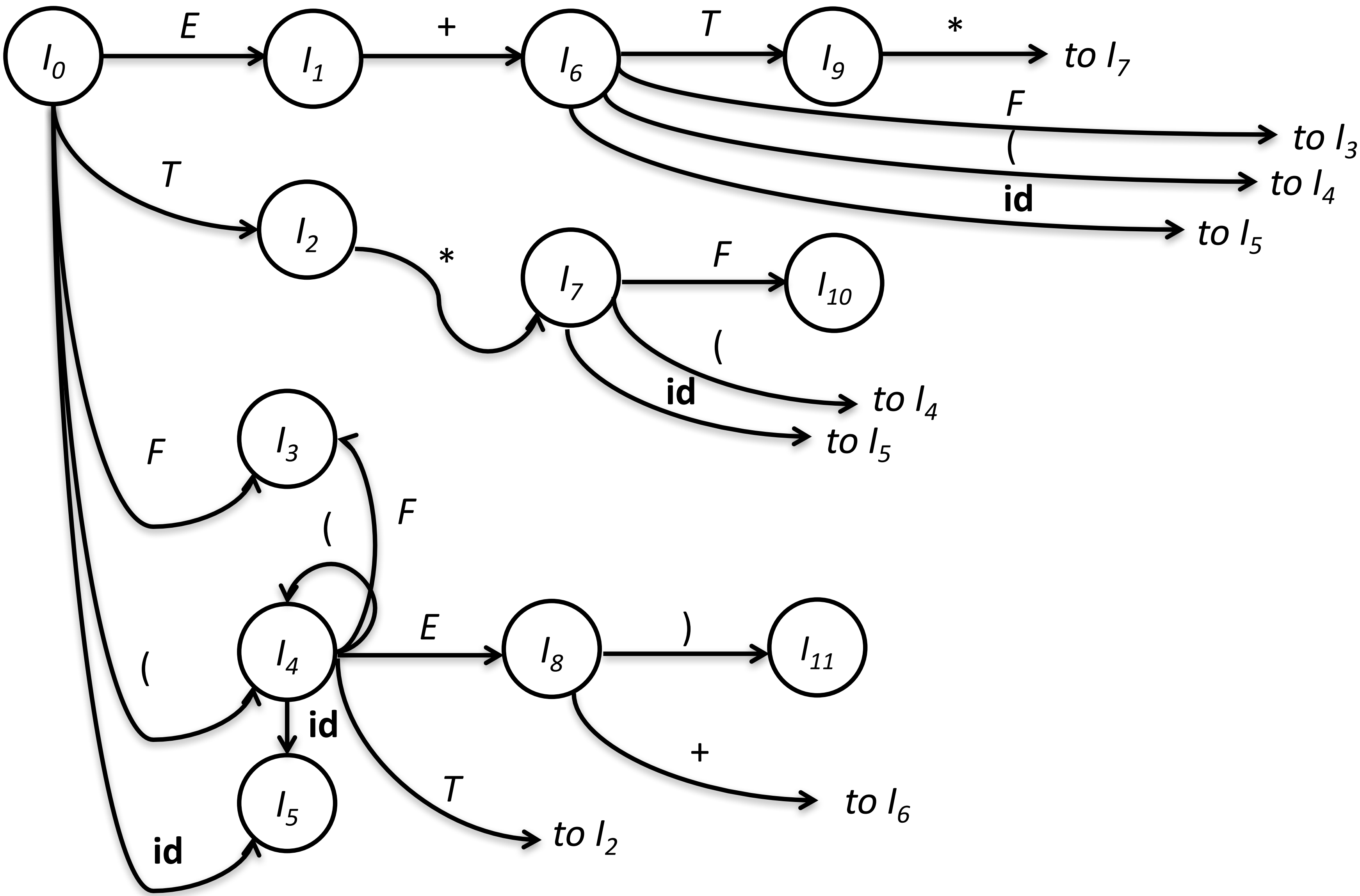
I_8
$F \rightarrow (E\cdot)$
$E \rightarrow E\cdot +T$

I_9
$E \rightarrow E+T\cdot$
$T \rightarrow T\cdot +F$

I_{10}
$T \rightarrow T*F\cdot$

I_{11}
$F \rightarrow (E)\cdot$

Transition Diagram



SLR Parsing Table

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			

Parsing Algorithm

```
set ip to point the first symbol of w$;
repeat forever begin
    let s be the state on top of the stack and
    a the symbol pointed to by ip;
    if (action[s, a] = shift s′ then begin
        push a then s′ on top of the stack;
        advance ip to the next input symbol
    end
    else if action[s, a] = reduce A→b then begin
        pop 2x|b| symbols off the stack;
        let s′ be the state now on top of the stack;
        push A then goto[s′, A] on top of the stack;
        output the production A→b
    end
    else if action[s, a] = accept then
        return
    else error()
end
```

`id*id+id`

STACK	INPUT	ACTION
(1) 0	**id*id+id$**	shift
(2) 0 **id** 5	***id+id$**	reduce by $F \rightarrow$ **id**
(3) 0 *F* 3	***id+id$**	reduce by $T \rightarrow F$
(4) 0 *T* 2	***id+id$**	shift
(5) 0 *T* 2 * 7	**id+id$**	shift
(6) 0 *T* 2 * 7 **id** 5	**+id$**	reduce by $F \rightarrow$ **id**
(7) 0 *T* 2 * 7 *F* 10	**+id$**	reduce by $T \rightarrow T*F$
(8) 0 *T* 2	**+id$**	reduce by $E \rightarrow T$
(9) 0 *E* 1	**+id$**	shift
(10) 0 *E* 1 + 6	**id$**	shift
(11) 0 *E* 1 + 6 **id** 5	$	reduce by $F \rightarrow$ **id**
(12) 0 *E* 1 + 6 *F* 3	$	reduce by $T \rightarrow F$
(13) 0 *E* 1 + 6 *T* 9	$	$E \rightarrow E+T$
(14) 0 *E* 1	$	accept

Productions
(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**