Lecture 08

Syntax Analyzer (Parser)

Part 5: Implementation of SLR parsing

Hyosu Kim

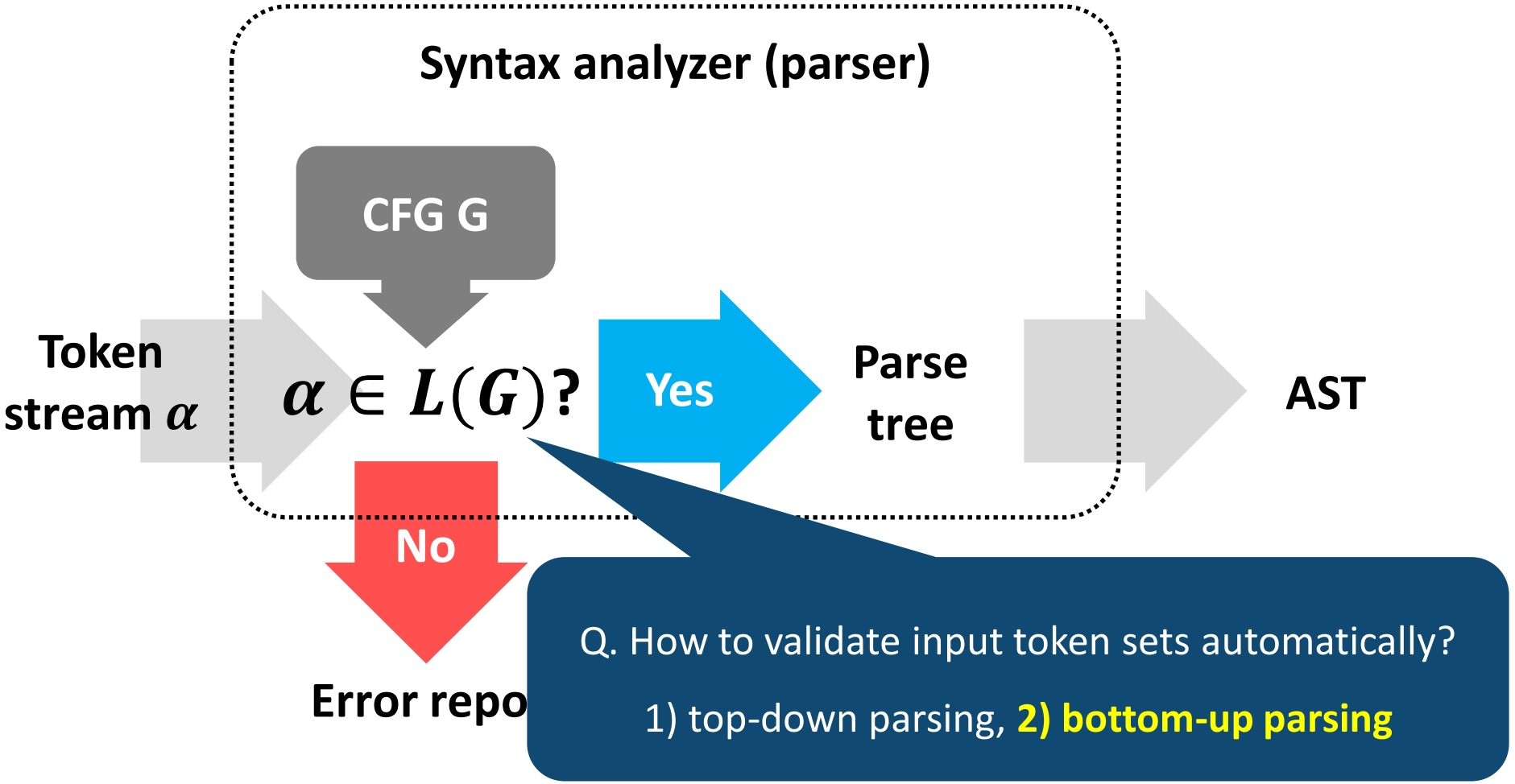
School of Computer Science and Engineering

Chung-Ang University, Seoul, Korea

https://hcslab.cau.ac.kr

hskimhello@cau.ac.kr, hskim.hello@gmail.com

Syntax analyzer



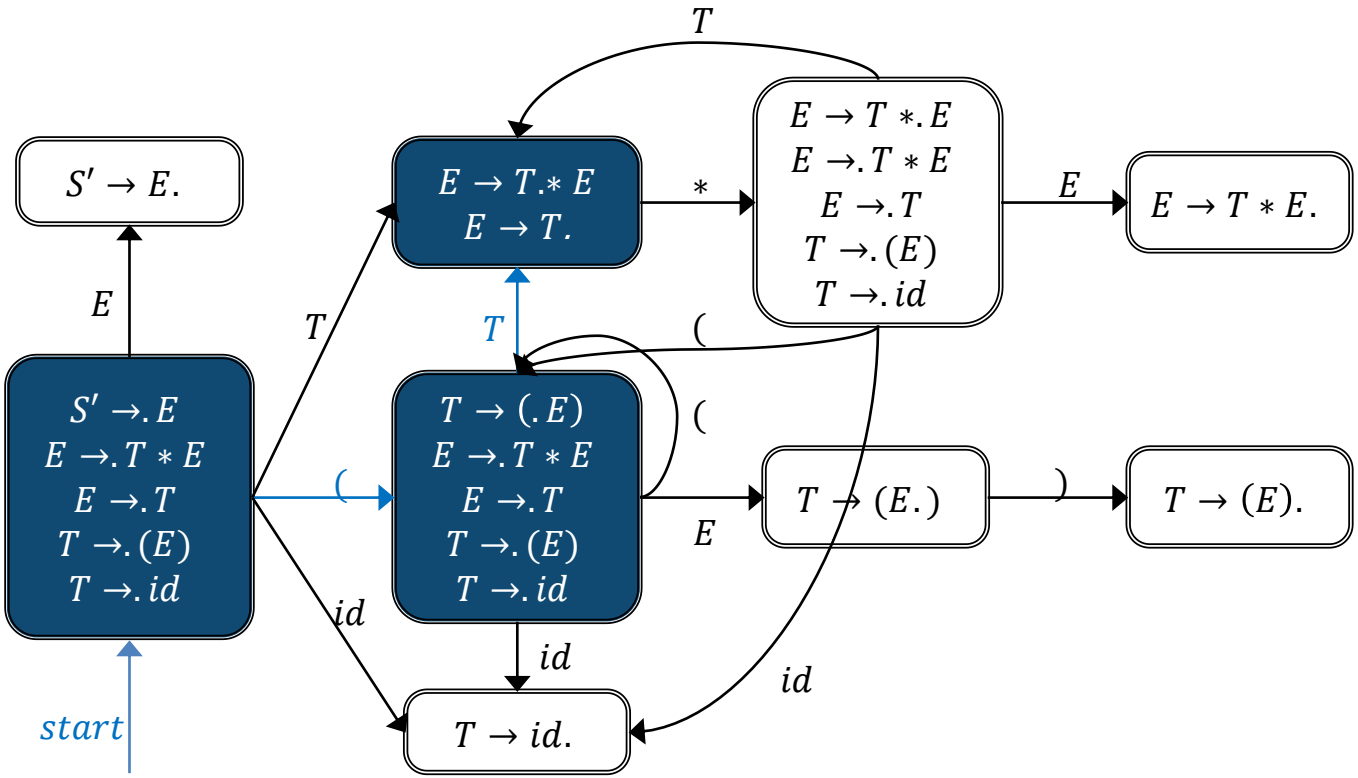
SLR parsing

When $\alpha|b\omega$,

1. Check whether α is a viable prefix or not by using DFA

   Travel DFA with α and check whether it terminates in any state q_i of the DFA

   - e.g., when the current parsing state is $(T| * id)$,



SLR parsing

When $\alpha|b\omega$,

1. Check whether α is a viable prefix or not

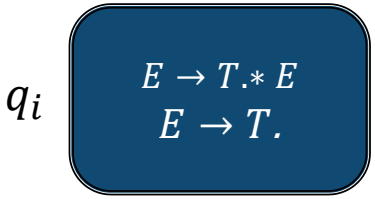
   Travel DFA with α and check whether it terminates in any state q_i of the DFA

2. When it terminates in a state q_i,

   - **Reduce by $X \rightarrow \beta$** if q_i contains item $X \rightarrow \beta.$ and $\boldsymbol{b \in Follow(X)}$

     where β is a suffix of α
   - **Shift** if q_i has a transition on an input symbol b, **reject** otherwise

   e.g., when the current parsing state is $(T| * id)$,



$$\beta = T \qquad X = E \qquad * \notin Follow(E)$$

We do shift!!

SLR parsing

When $\alpha|b\omega$,

1. Check whether α is a viable prefix or not

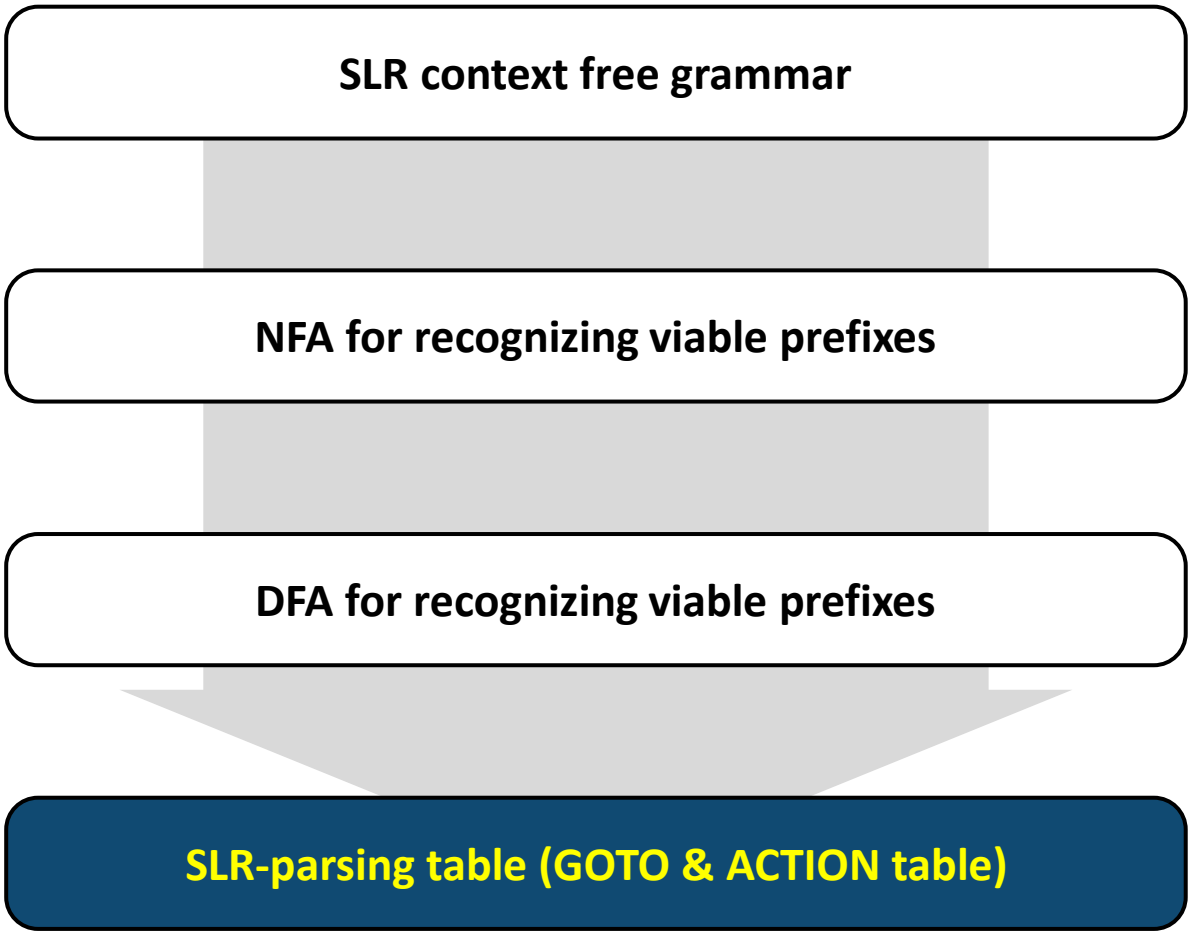
   Travel DFA with α and check whether it terminates in any state q_i of the DFA

2. When it terminates in a state q_i,
   - **Reduce by $X \rightarrow \beta$** if q_i contains item $X \rightarrow \beta.$ and $\boldsymbol{b \in Follow(X)}$

     where β is a suffix of α
   - **Shift** if q_i has a transition on an input symbol b, **reject** otherwise

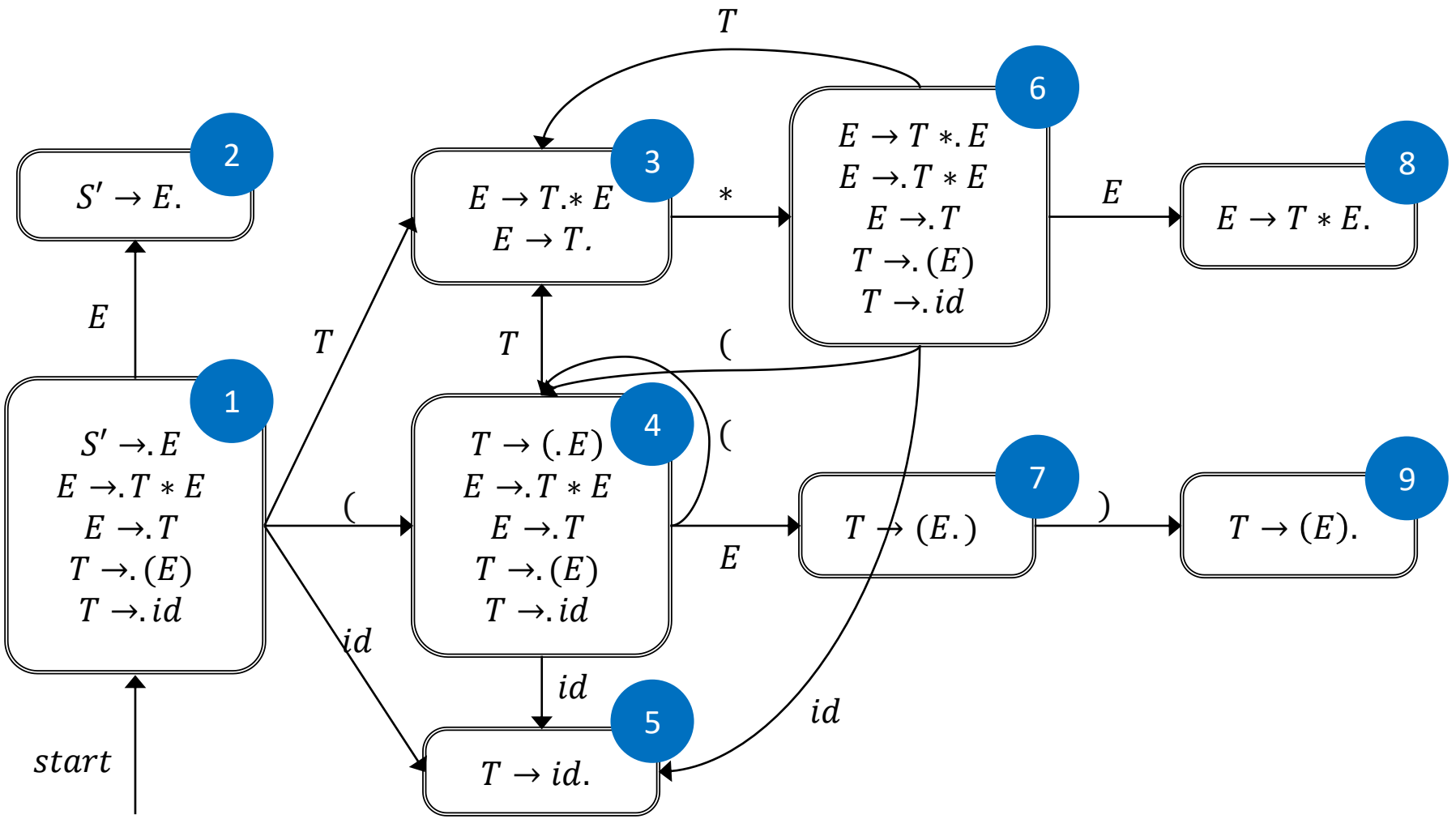
Q. How to implement a program which does do this SLR parsing process efficiently and automatically?

For the implementation of SLR parser



SLR-parsing table construction

(1) $S' \rightarrow E,$ (2) $E \rightarrow T * E,$ (3) $E \rightarrow T,$ (4) $T \rightarrow (E)$ (5) $T \rightarrow id$



SLR-parsing table construction

Step 1. Construct a goto table for each state q_i and each non-terminal A

- $GOTO(q_i, A) = q_j,\ if\ there\ is\ a\ transition\ from\ q_i\ to\ q_j\ with\ A\colon\ \delta(q_i, A) = q_j$

	ACTION					GOTO	
	*	(	)	id	$	E	T
1						2	3
2							
3							
4						7	3
5							
6						8	3
7							
8							
9							

SLR-parsing table construction

Step 2. Construct an action table for each state q_i and each terminal a

1) If q_i has item $X \rightarrow \alpha.a\beta$ and $\delta(q_i, a) = q_j$, then $ACTION(q_i, a) = shift\ \&\ goto\ q_j$

	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2							
3	S6						
4		S4		S5		7	3
5							
6		S4		S5		8	3
7			S9				
8							
9							

SLR-parsing table construction

Step 2. Construct an action table for each state q_i and each terminal a

2) If q_i has item $X \rightarrow \alpha.$ and $a \in Follow(X)$, then $ACTION(q_i, a) = reduce\ by\ X \rightarrow \alpha$

	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

SLR-parsing table construction

Step 2. Construct an action table for each state q_i and each terminal a

3) Otherwise, error

	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

Implementation of SLR parsing

Let's use a SLR-parsing table

For $|id * id\$$

Start state: 1

Next input symbol: id

Decision: Shift & goto 5

	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

Implementation of SLR parsing

Let's use a SLR-parsing table

For $id|*id\$$

Step 1: check whether a left substring id is a viable prefix or not

Start state: 1

Next input symbol: id

Decision: Goto 5

(we can ignore shift

because an indicator is already shifted)

Step 2: make a decision

Current state: 5

Next input symbol: *

Decision: Reduce by (5) $T \rightarrow id$

	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

Implementation of SLR parsing

Let's use a SLR-parsing table

For $T| * id\$$

Step 1: check whether a left substring T is a viable prefix or not

Start state: 1

Next input symbol: T

Decision: Goto 3

Step 2: make a decision

Current state: 3

Next input symbol: *

Decision: Shift and goto 6

	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

Implementation of SLR parsing

Let's use a SLR-parsing table

For $T * |id\$$

Step 1: check whether a left substring $T *$ is a viable prefix or not

Start state: 1 Next input symbol: T Decision: Goto 3

Current state: 3 Next input symbol: * Decision: Goto 6

Step 2: make a decision

Current state: 6

Next input symbol: id

Decision: Shift and goto 5

	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

Implementation of SLR parsing

Let's use a SLR-parsing table

For $T * id|\$$

Step 1: check whether a left substring $T * id$ is a viable prefix or not

Start state: 1 Next input symbol: T Decision: Goto 3

Current state: 3 Next input symbol: * Decision: Goto 6

Current state: 6 Next input symbol: id Decision: Goto 5

Step 2: make a decision

Current state: 5

Next input symbol: $

Decision: Reduce by (5) $T \rightarrow id$

	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

Implementation of SLR parsing

Let's use a SLR-parsing table

For $T * T|\$$

Step 1: check whether a left substring $T * T$ is a viable prefix or not

Start state: 1 Next input symbol: T Decision: Goto 3

Current state: 3 Next input symbol: * Decision: Goto 6

Current state: 6 Next input symbol: T Decision: Goto 3

Step 2: make a decision

Current state: 3

Next input symbol: $

Decision: Reduce by (3) $E \rightarrow T$

	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

Implementation of SLR parsing

Let's use a SLR-parsing table

For $T * E|\$$

Step 1: check whether a left substring $T * E$ is a viable prefix or not

Start state: 1 Next input symbol: T Decision: Goto 3

Current state: 3 Next input symbol: * Decision: Goto 6

Current state: 6 Next input symbol: E Decision: Goto 8

Step 2: make a decision

Current state: 8

Next input symbol: $

Decision: Reduce by (2) $E \rightarrow T * E$

	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

Implementation of SLR parsing

Let's use a SLR-parsing table

For $E|\$$

Step 1: check whether a left substring E is a viable prefix or not

Start state: 1 Next input symbol: E Decision: Goto 2

Step 2: make a decision

Current state: 2

Next input symbol: $

Decision: Reduce by (1) $S' \rightarrow E$

	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

Implementation of SLR parsing

Let's use a SLR-parsing table

For $S'|\$$

An input string is reduced to the dummy start symbol S', then we accept the string

Implementation of SLR parsing

Let's use a SLR-parsing table

For $T * E|\$$

Step 1: check whether a left substring $T * E$ is a viable prefix or not

Start state: 1 Next input symbol: T Decision: Goto 3

Current state: 3 Next input symbol: * Decision: Goto 6

Current state: 6 Next input symbol: E Decision: Goto 8

Step 2: make a decision

Current state: 8

Next input symbol: $

Decision: Reduce by (2) E

	ACTION					GOTO	
	*	(	)	id	$	E	T
		S4		S5		2	3
					R(1)		
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

Every time we make a new decision,

we should repeat to travel DFA… It's too inefficient…

Implementation of SLR parsing

Let's use a SLR-parsing table with a stack

For $|id * id\$$

Initialization

- Push the start state (e.g., 1) into the stack
  - Current state = the state stored in the top of the stack
  - Next input symbol = the leftmost terminal of a right substring

Stack
1

Current state: 1

Next input symbol: id

	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

Implementation of SLR parsing

Let's use a SLR-parsing table with a stack

For $\left|id * id\$ \Rightarrow_{shift} id\right| * id\$$

Make a decision: Shift and goto 5

- Push the next state (e.g., 5) into the stack
- Move the splitter to the right

Stack
5
1

Current state: $1 \Rightarrow 5$

Next input symbol: $id \Rightarrow *$

	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

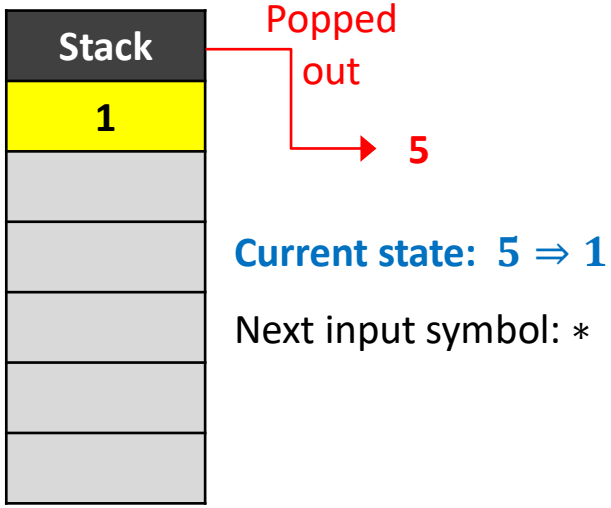
Implementation of SLR parsing

Let's use a SLR-parsing table with a stack

For $id|* id\$ \Rightarrow_{reduce} T|*id\$$

Make a decision: Reduce by (5) $T \rightarrow id$

- Step 1: For $A \rightarrow \alpha$, Pop $|\alpha|$ contents from the stack (e.g., |id| = 1)



	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

Implementation of SLR parsing

Let's use a SLR-parsing table with a stack

For $id|* id\$ \Rightarrow_{reduce} T| * id\$$

Make a decision: Reduce by (5) $T \rightarrow id$

- Step 1: For $A \rightarrow \alpha$, Pop $|\alpha|$ contents from the stack (e.g., |id| = 1)
- **Step 2: For $A \rightarrow \alpha$, Push GOTO (current state, A) into the stack (e.g., GOTO(1, T) = 3)**

Stack
3
1

Current state: $1 \Rightarrow 3$

Next input symbol: $*$

	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

Implementation of SLR parsing

Let's use a SLR-parsing table with a stack

For $T|*id\$ \Rightarrow_{shift} T*|id\$$

Make a decision: Shift and goto 6

- Push the next state (e.g., 6) into the stack
- Move the splitter to the right

Stack
6
3
1

Current state: $3 \Rightarrow 6$

Next input symbol: $*\Rightarrow id$

	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

Implementation of SLR parsing

Let's use a SLR-parsing table with a stack

For $T * |id\$ \Rightarrow_{shift} T * id|\$$

Make a decision: Shift and goto 5

- Push the next state (e.g., 5) into the stack
- Move the splitter to the right

Stack
5
6
3
1

Current state: $6 \Rightarrow 5$

Next input symbol: $id \Rightarrow \$$

	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

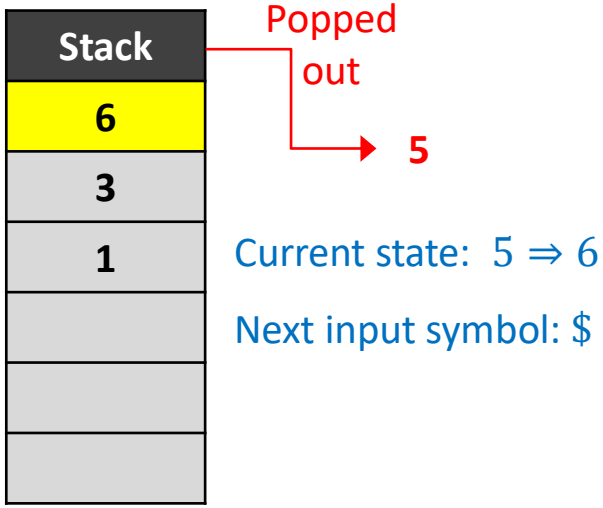
Implementation of SLR parsing

Let's use a SLR-parsing table with a stack

For $T * id|\$ \Rightarrow_{reduce} T * T|\$$

Make a decision: Reduce by (5) $T \rightarrow id$

- Step 1: For $A \rightarrow \alpha$, Pop $|\alpha|$ contents from the stack (e.g., |id| = 1)



	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

Implementation of SLR parsing

Let's use a SLR-parsing table with a stack

For $T * id|\$ \Rightarrow_{reduce} T * T|\$$

Make a decision: Reduce by (5) $T \rightarrow id$

- Step 1: For $A \rightarrow \alpha$, Pop $|\alpha|$ contents from the stack (e.g., |id| = 1)
- Step 2: For $A \rightarrow \alpha$, Push GOTO (current state, A) into the stack (e.g., GOTO(6, T) = 3)

Stack
3
6
3
1

Current state: $6 \Rightarrow 3$

Next input symbol: $

	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

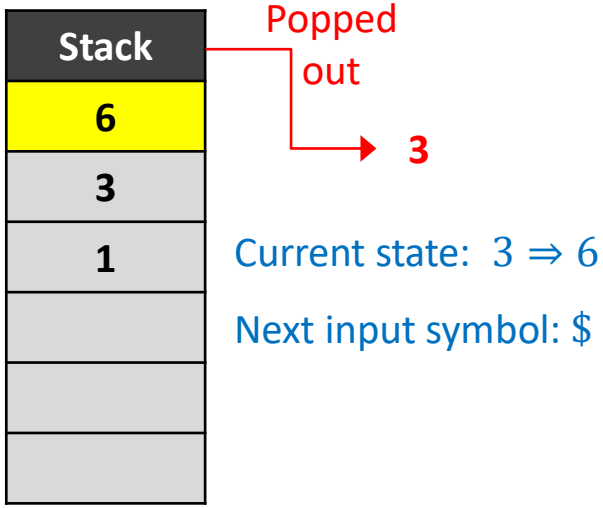
Implementation of SLR parsing

Let's use a SLR-parsing table with a stack

For $T * T|\$ \Rightarrow_{reduce} T * E|\$$

Make a decision: Reduce by (3) $E \rightarrow T$

- Step 1: For $A \rightarrow \alpha$, Pop $|\alpha|$ contents from the stack (e.g., |T| = 1)



	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

Implementation of SLR parsing

Let's use a SLR-parsing table with a stack

For $T * T|\$ \Rightarrow_{reduce} T * E|\$$

Make a decision: Reduce by (3) $E \rightarrow T$

- Step 1: For $A \rightarrow \alpha$, Pop $|\alpha|$ contents from the stack (e.g., |T| = 1)
- Step 2: For $A \rightarrow \alpha$, Push GOTO (current state, A) into the stack (e.g., GOTO(6, E) = 8)

Stack
8
6
3
1

Current state: $6 \Rightarrow 8$

Next input symbol: $

	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

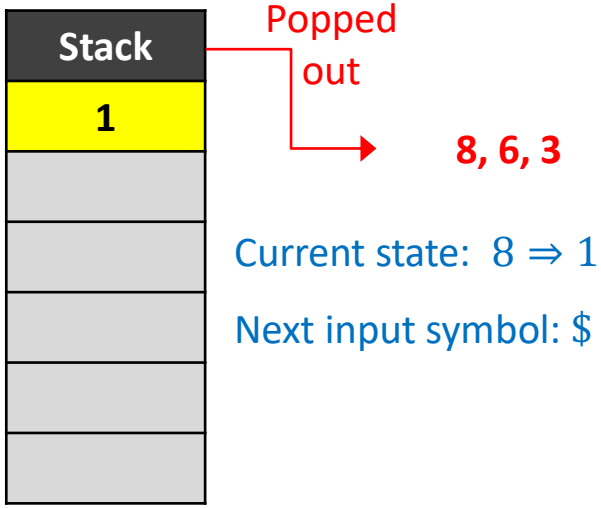
Implementation of SLR parsing

Let's use a SLR-parsing table with a stack

For $T * E|\$ \Rightarrow_{reduce} E|\$$

Make a decision: Reduce by (2) $E \rightarrow T * E$

- Step 1: For $A \rightarrow \alpha$, Pop $|\alpha|$ contents from the stack (e.g., |T*E| = 3)



	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

Implementation of SLR parsing

Let's use a SLR-parsing table with a stack

For $T * E|\$ \Rightarrow_{reduce} E|\$$

Make a decision: Reduce by (2) $E \rightarrow T * E$

- Step 1: For $A \rightarrow \alpha$, Pop $|\alpha|$ contents from the stack (e.g., |T*E| = 3)
- Step 2: For $A \rightarrow \alpha$, Push GOTO (current state, A) into the stack (e.g., GOTO(1, E) = 2)

Stack
2
1

Current state: 1 ⇒ 2

Next input symbol: $

	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

Implementation of SLR parsing

Let's use a SLR-parsing table with a stack

For $E|\$ \Rightarrow_{reduce} S|\$$

Make a decision: Reduce by (1) $S' \rightarrow E$

- The input string id*id is reduced to the dummy start symbol S'!!
- Accept!!

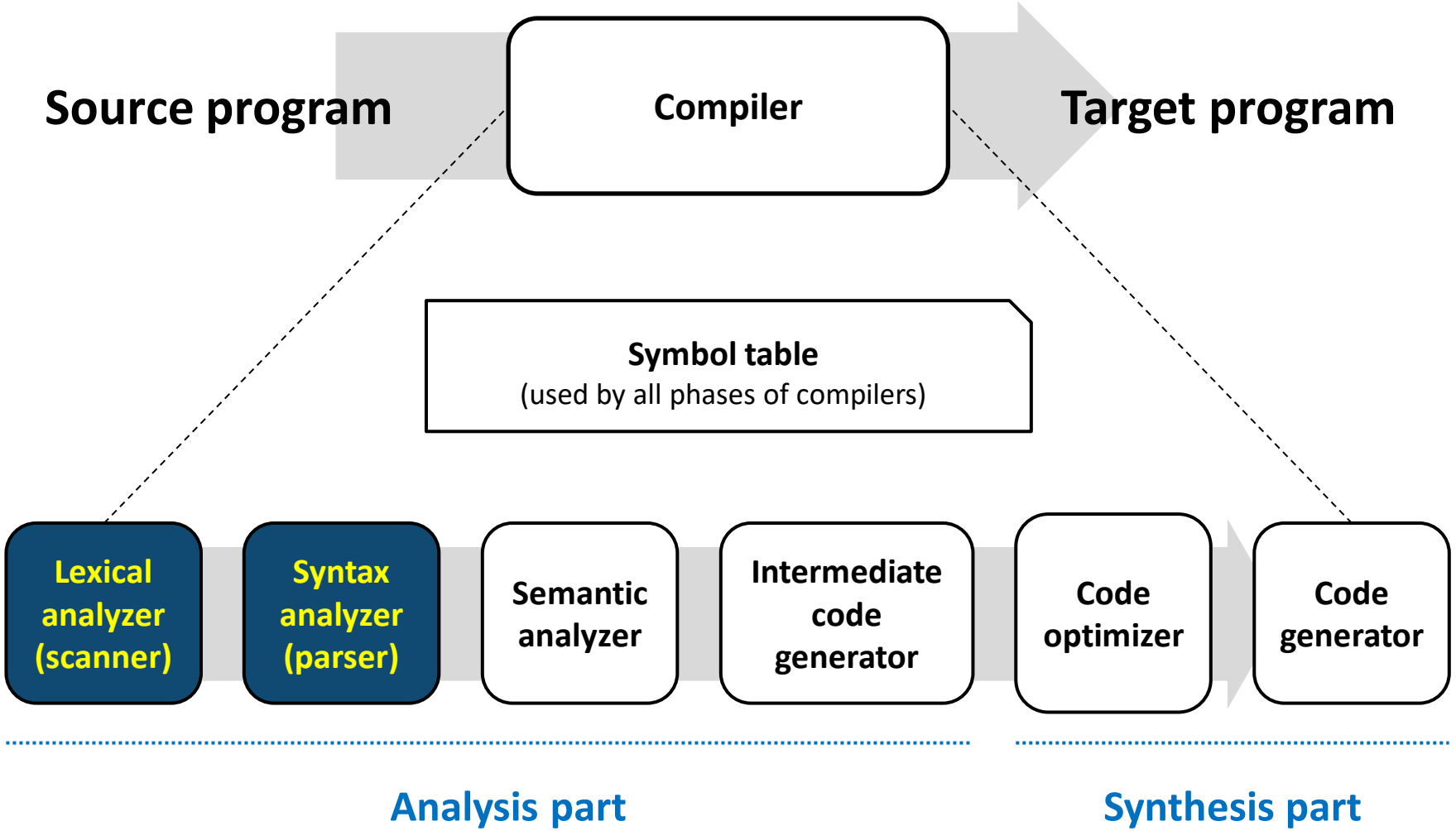
Stack
2
1

Current state: 2

Next input symbol: $

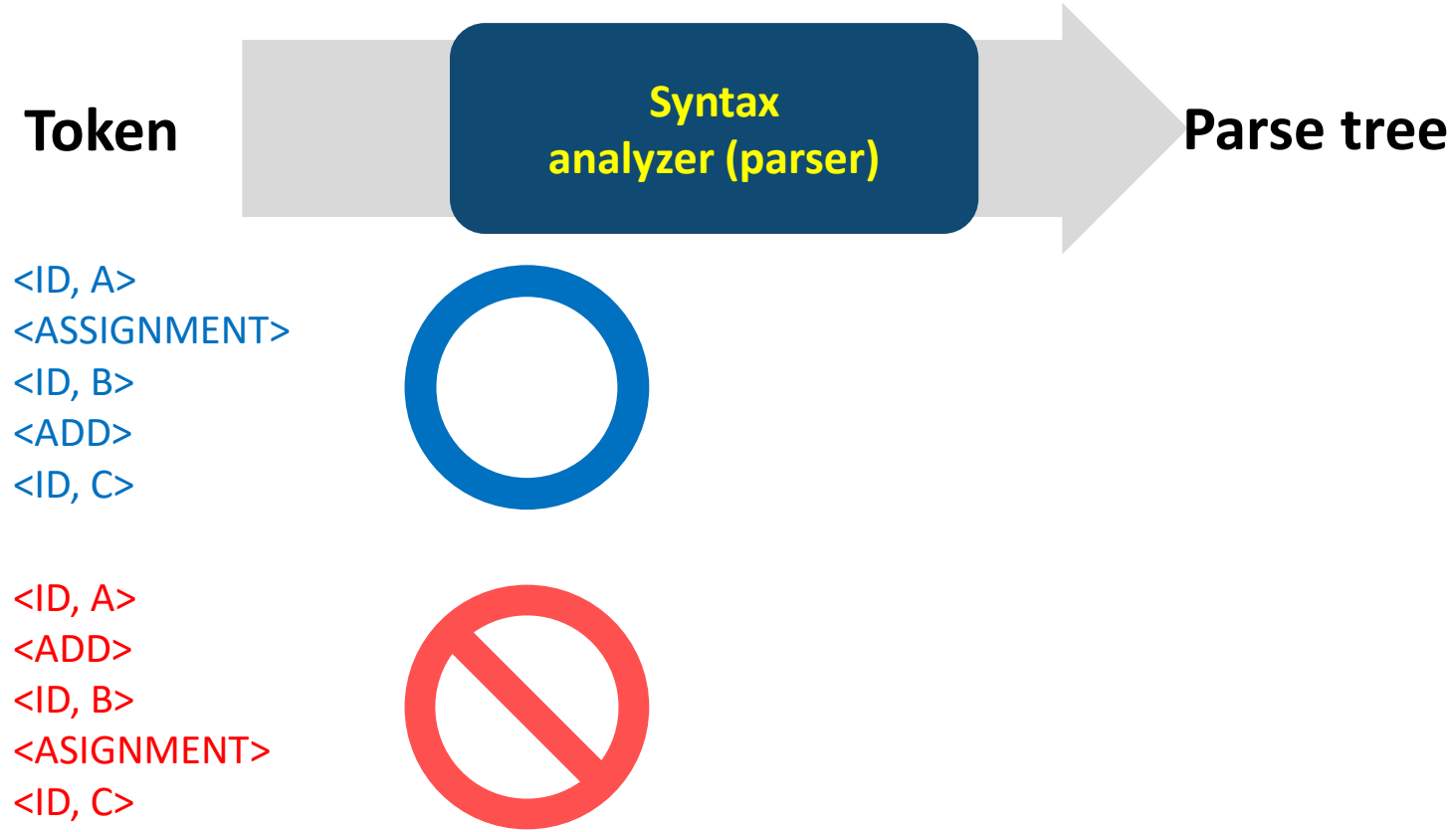
	ACTION					GOTO	
	*	(	)	id	$	E	T
1		S4		S5		2	3
2					R(1)		
3	S6		R(3)		R(3)		
4		S4		S5		7	3
5	R(5)		R(5)		R(5)		
6		S4		S5		8	3
7			S9				
8			R(2)		R(2)		
9	R(4)		R(4)		R(4)		

Summary


Source program
Compiler
Target program
Symbol table
(used by all phases of compilers)
Lexical analyzer (scanner)
Syntax analyzer (parser)
Semantic analyzer
Intermediate code generator
Code optimizer
Code generator
Analysis part
Synthesis part

Syntax analyzer

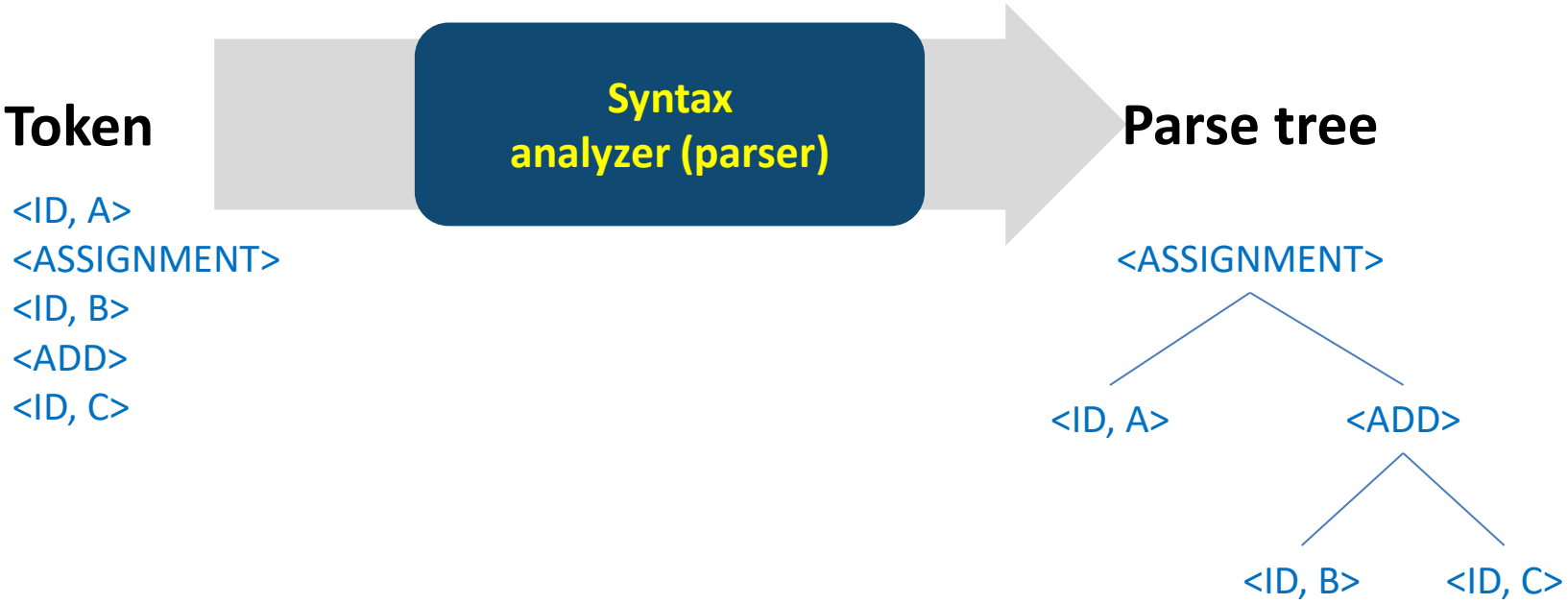
1. Decides whether a given set of tokens is valid or not



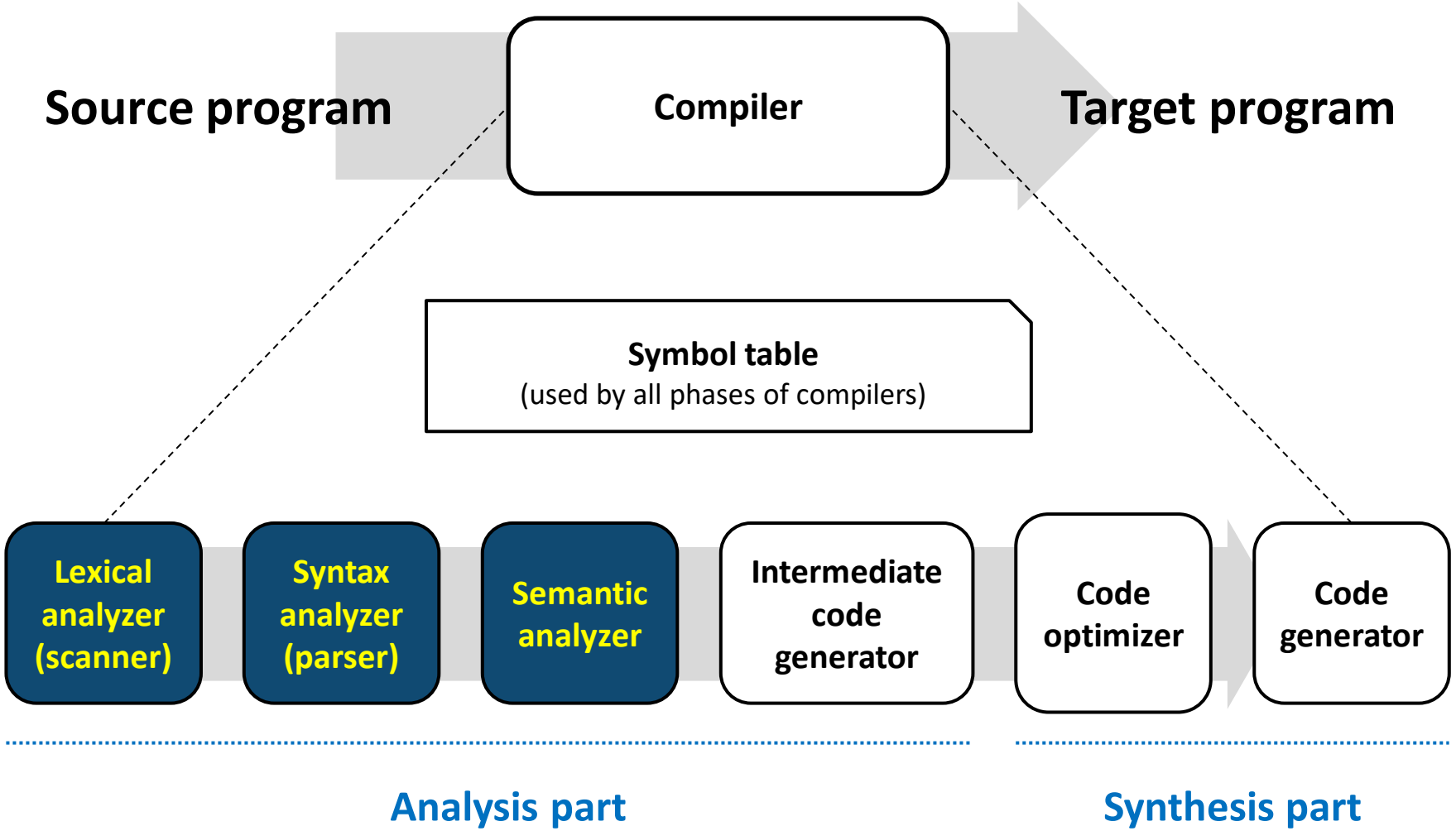
Syntax analyzer

2. Creates a tree-like intermediate representation (e.g., parse tree)

that depicts the grammatical structure of the token stream



Summary


Source program
Compiler
Target program
Symbol table
(used by all phases of compilers)
Lexical analyzer (scanner)
Syntax analyzer (parser)
Semantic analyzer
Intermediate code generator
Code optimizer
Code generator
Analysis part
Synthesis part