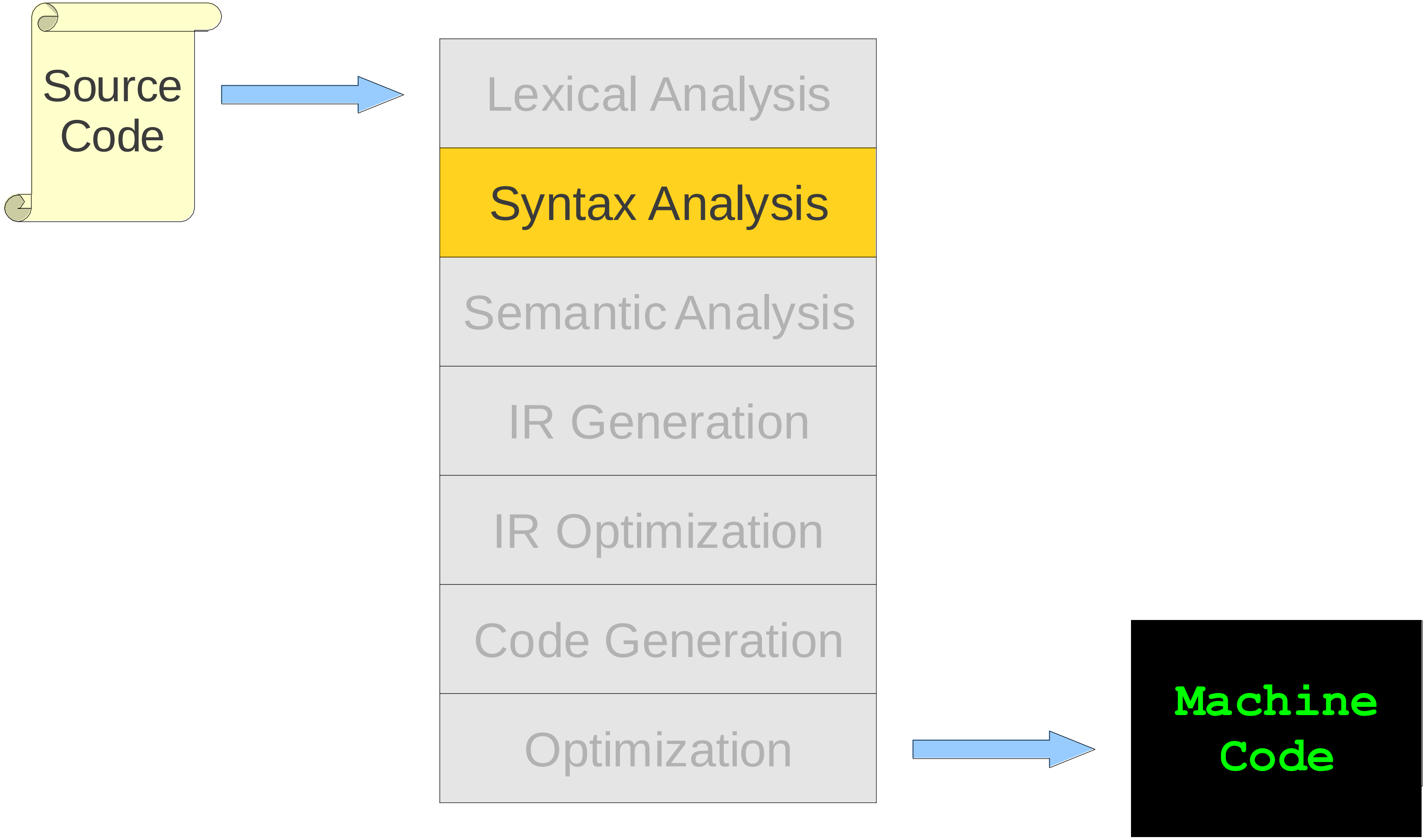
Compiler
– 3-5. Bottom-up Parsing –

JIEUNG KIM

jieungkim@yonsei.ac.kr

YONSEI UNIVERSITY

Where are we?

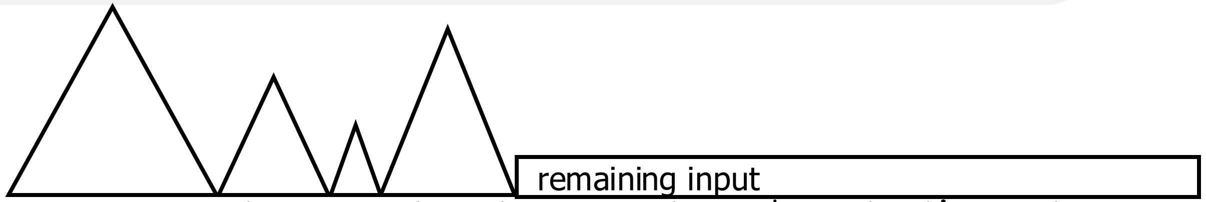

Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Outlines

- Role of the syntax analysis (parser)
- Context free grammar
- Push down automata
- Top-down parsing
- Bottom-up parsing
- Simple LR
- More powerful LR parsers and other issues in parsers
- Syntactic error handler
- Parser generator

Bottom-up parsing

Bottom-up parsing



- **Bottom-up parsing**
  - Definition
    - For a given input stream w, and given grammar, to construct parse tree by starting at leaf node working to the root
  - LR methods (**L**eft-to-right, **R**ightmost derivation)
    - Shift-reduce parsing, SLR, Canonical LR, LALR
  - Reductions
    - Process of "reducing" a string to the start symbol of grammar
      - Accept when all input read and reduced to start symbol of the grammar
    - Decision related to when to reduce and what production rule to apply
      - Whenever we've matched the right-hand side of a production, reduce it to the appropriate non-terminal and add that non-terminal to the parse tree
    - The upper edge of this partial parse tree is known as the *frontier*

YONSEI UNIVERSITY

Bottom-up parsing

- An eagle eye of a bottom-up parsing

$E \rightarrow T$
$E \rightarrow E + T$
$T \rightarrow int$
$T \rightarrow (E)$

```
   int + (int + int + int)
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E
```

YONSEI UNIVERSITY

Bottom-up parsing

- An eagle eye of a bottom-up parsing

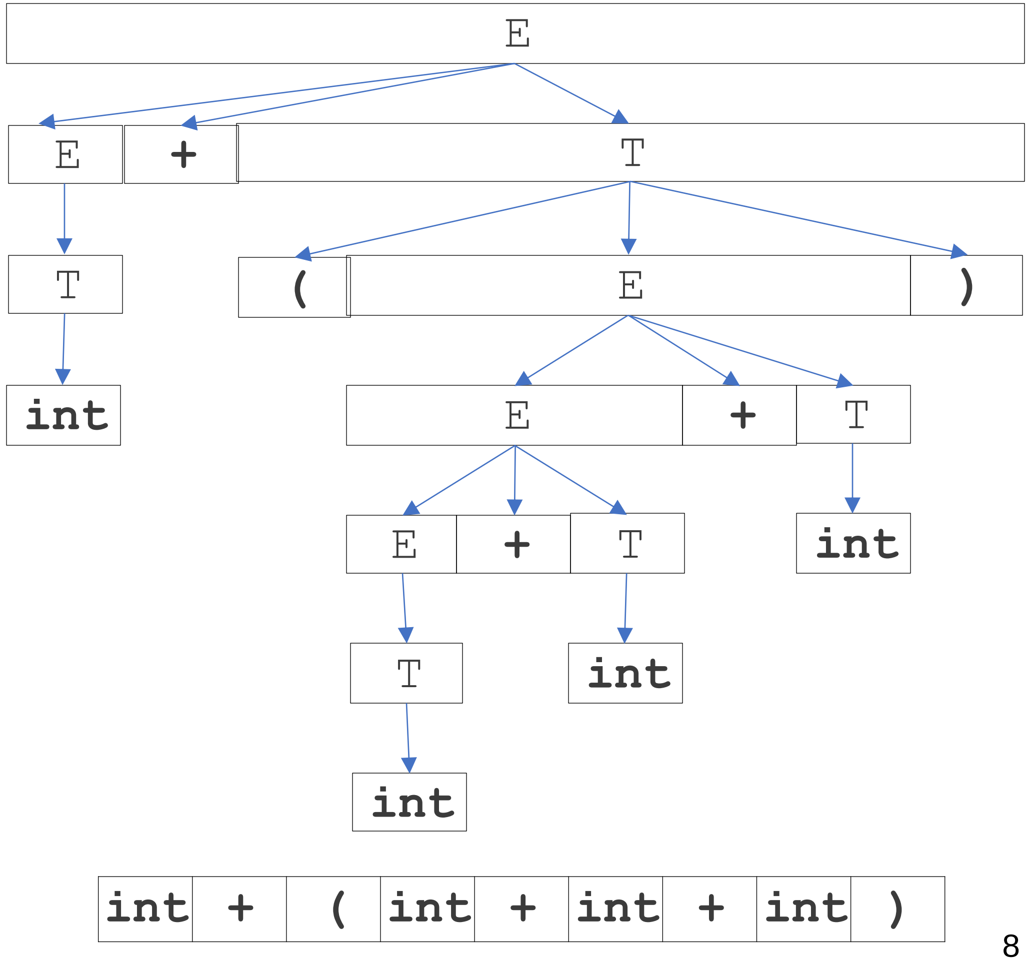
```
   int   +  (int   +  int   +  int)
⇒ T   +  (int   +  int   +  int)
⇒ E   +  (int   +  int   +  int)
⇒ E   +  (T   +  int   +  int)
⇒ E   +  (E   +  int   +  int)
⇒ E   +  (E   +  T   + int)
⇒ E   +  (E   +  int)
⇒ E   +  (E   +  T)
⇒ E   +  (E)
⇒ E   +  T
⇒ E
```

Each step in this bottom-up parse is called a reduction. We reduce a substring of the sentential form back to a nonterminal.

Bottom-up parsing

- An eagle eye of a bottom-up parsing

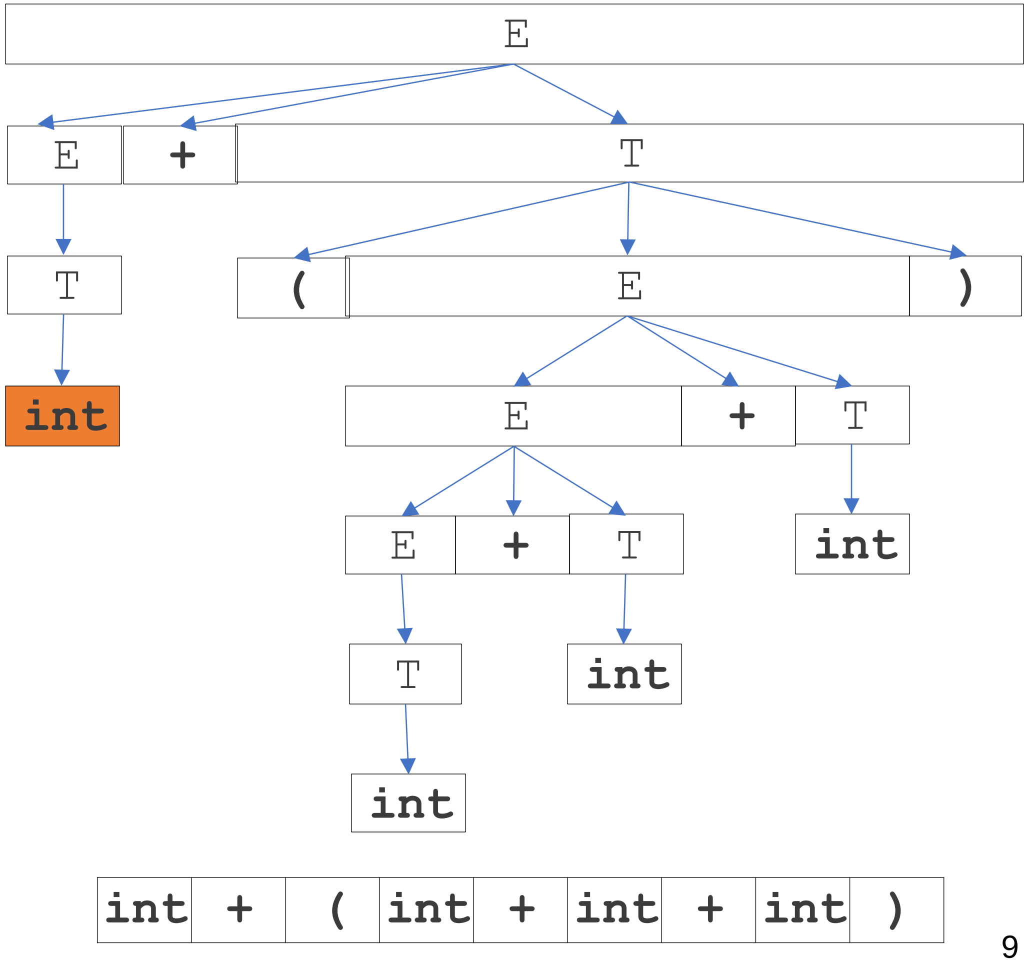
```
   int   +  (int   +  int   +  int)
⇒  T     +  (int   +  int   +  int)
⇒  E     +  (int   +  int   +  int)
⇒  E     +  (T     +  int   +  int)
⇒  E     +  (E     +  int   +  int)
⇒  E     +  (E     +  T     +  int)
⇒  E     +  (E     +  int)
⇒  E     +  (E     +  T)
⇒  E     +  (E)
⇒  E     +  T
⇒  E
```



Bottom-up parsing

- An eagle eye of a bottom-up parsing

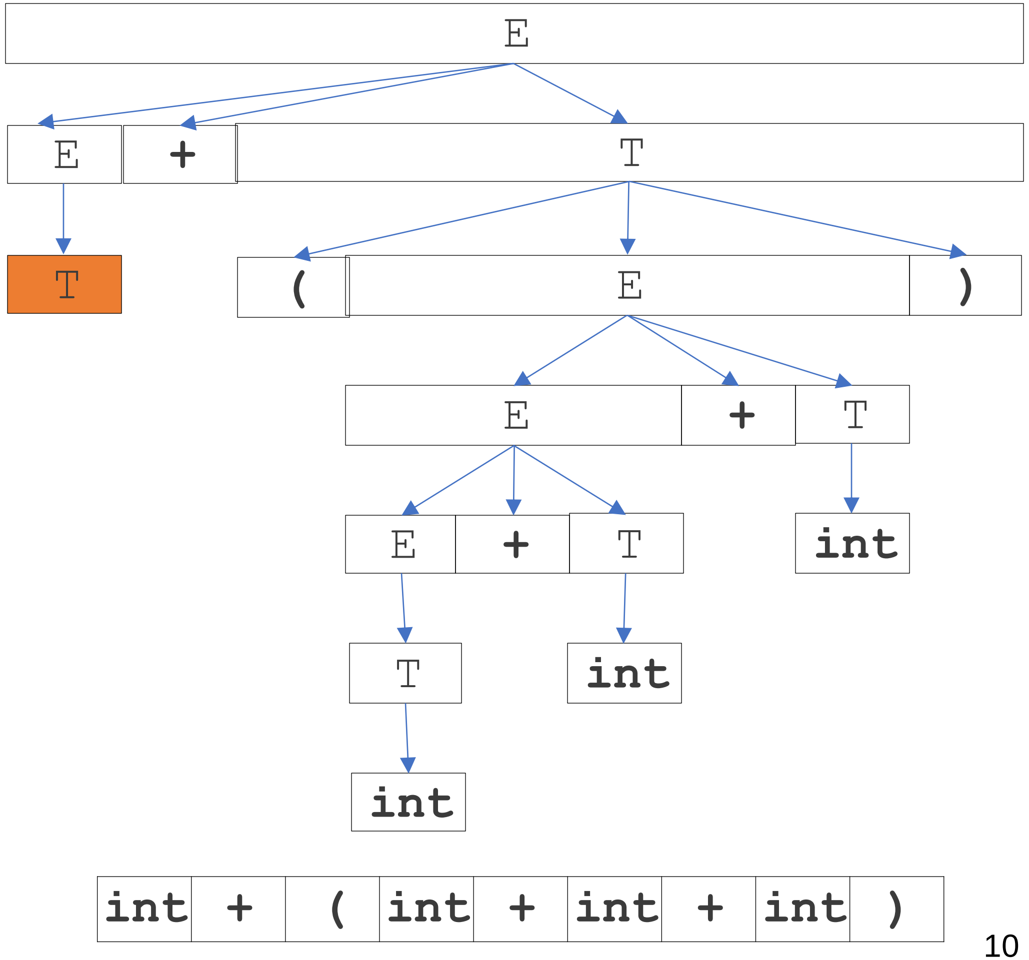
```
  int + (int + int + int)
⇒ T   + (int + int + int)
⇒ E   + (int + int + int)
⇒ E   + (T   + int + int)
⇒ E   + (E   + int + int)
⇒ E   + (E   + T   + int)
⇒ E   + (E   + int)
⇒ E   + (E   + T)
⇒ E   + (E)
⇒ E   + T
⇒ E
```



Bottom-up parsing

- An eagle eye of a bottom-up parsing

⇒ **T** + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E



Bottom-up parsing

- An eagle eye of a bottom-up parsing

⇒ E + (int + int + int)

⇒ E + (T + int + int)

⇒ E + (E + int + int)

⇒ E + (E + T + int)

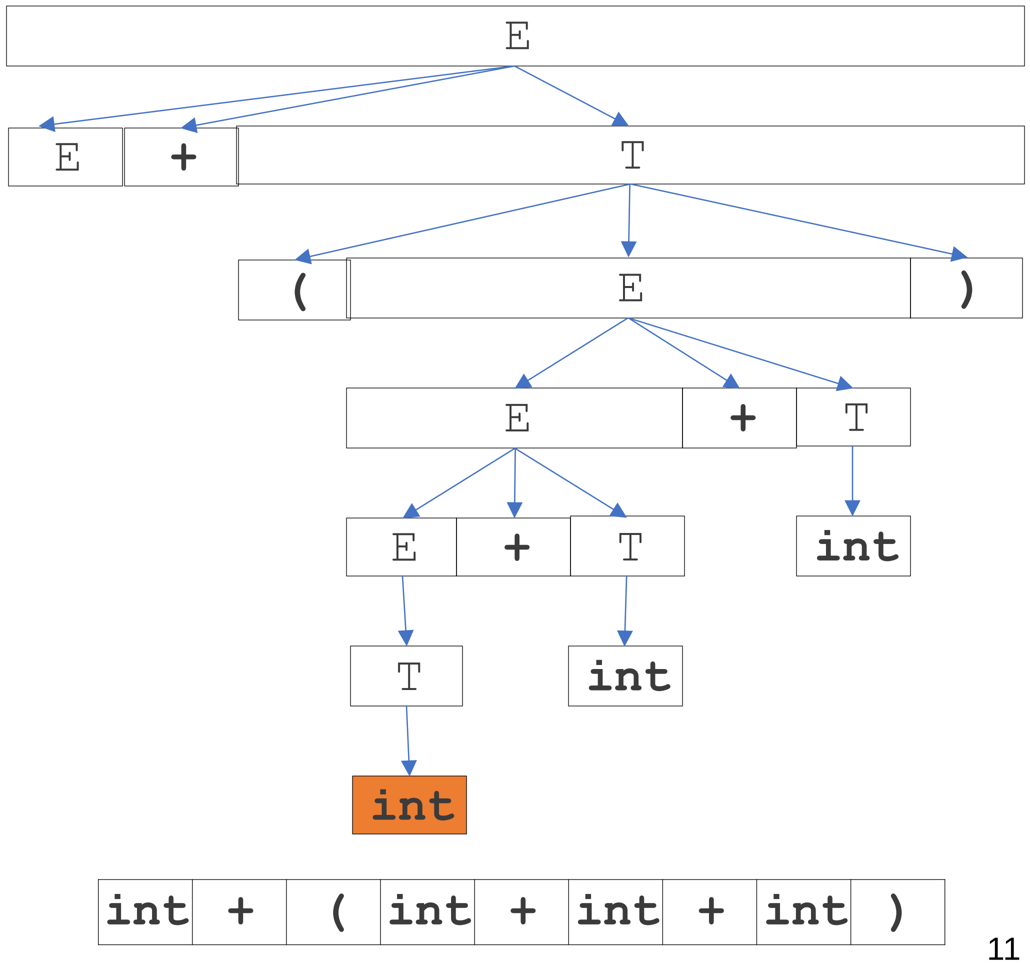
⇒ E + (E + int)

⇒ E + (E + T)

⇒ E + (E)

⇒ E + T

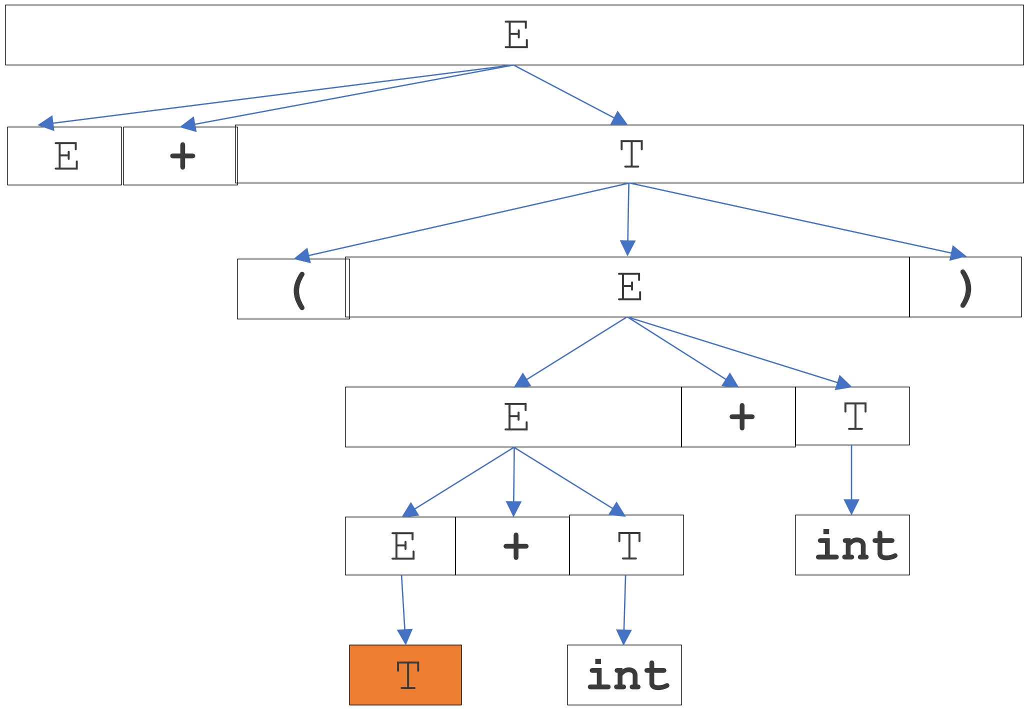
⇒ E



Bottom-up parsing

- An eagle eye of a bottom-up parsing

⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E

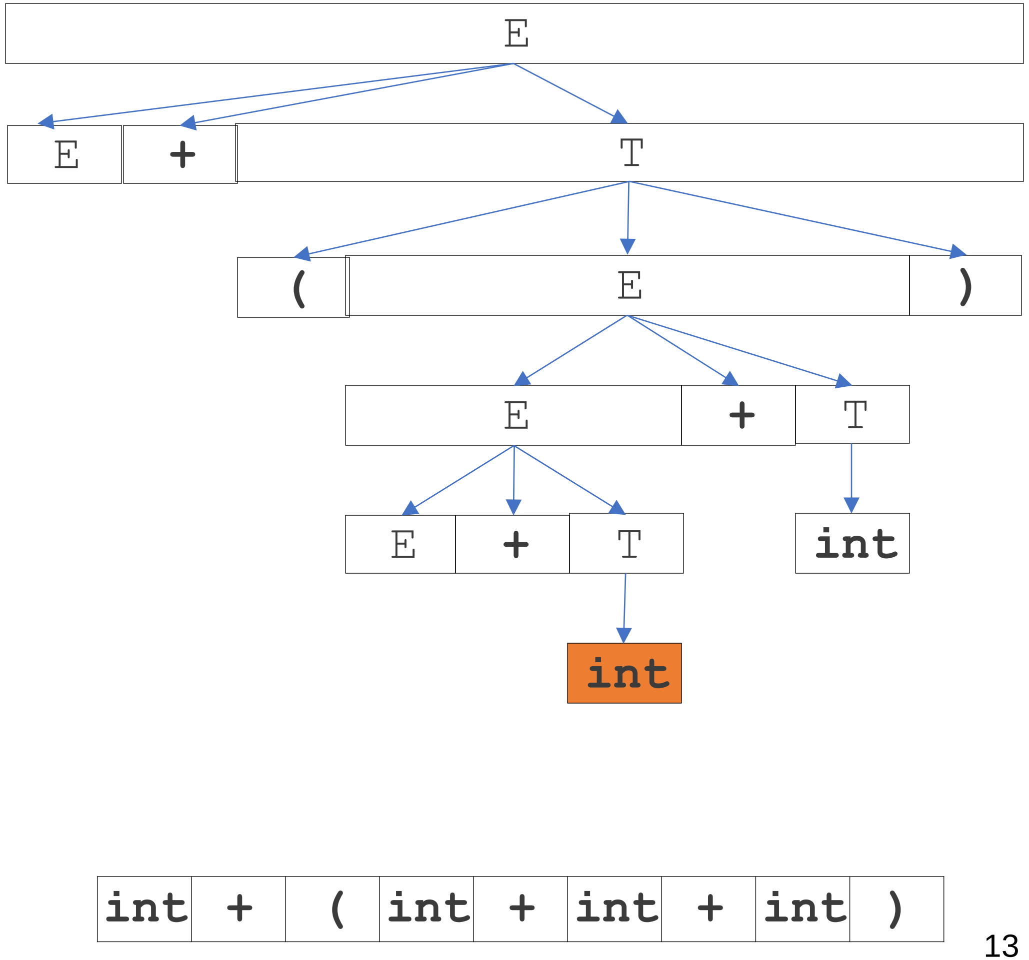


int	+	(	int	+	int	+	int	)

YONSEI UNIVERSITY

Bottom-up parsing

- An eagle eye of a bottom-up parsing



⇒ **E + (E + int + int)**

⇒ **E + (E + T + int)**

⇒ **E + (E + int)**

⇒ **E + (E + T)**

⇒ **E + (E)**

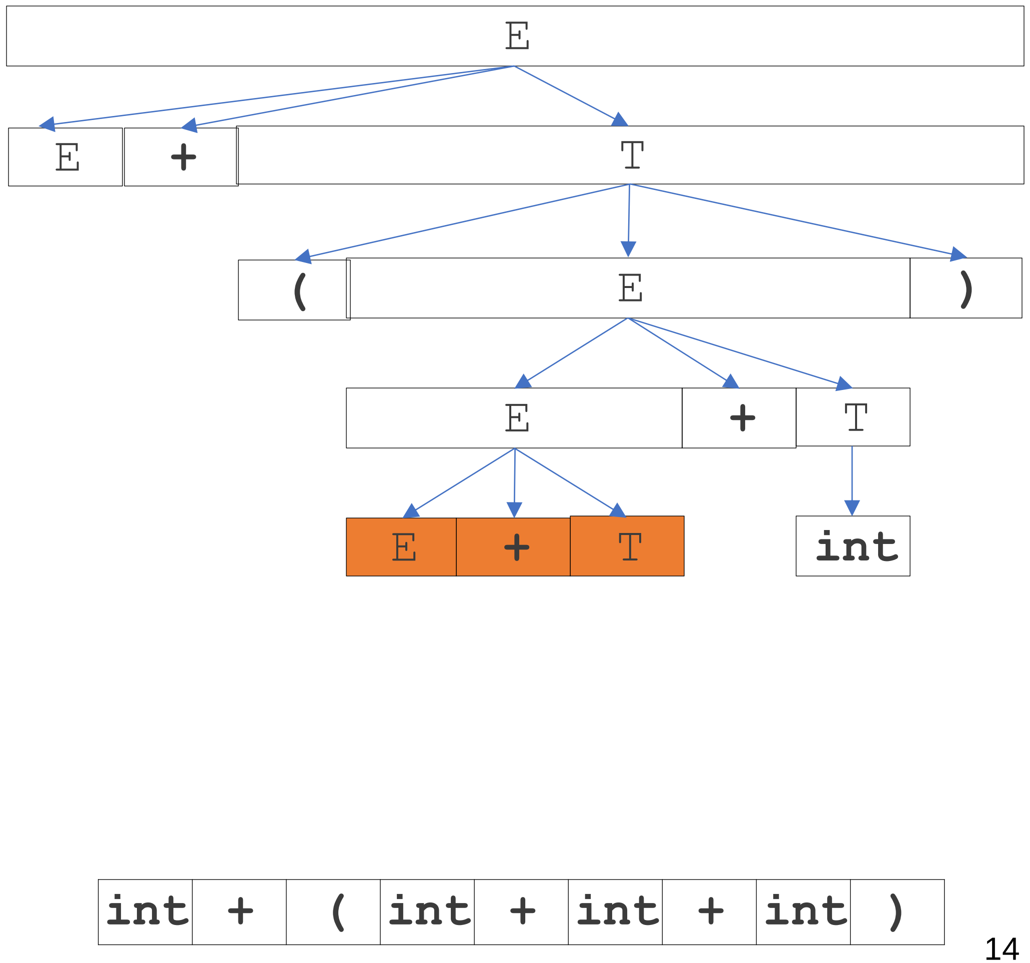
⇒ **E + T**

⇒ **E**

YONSEI UNIVERSITY

Bottom-up parsing

- An eagle eye of a bottom-up parsing



⇒ E + (E + T + int)

⇒ E + (E + int)

⇒ E + (E + T)

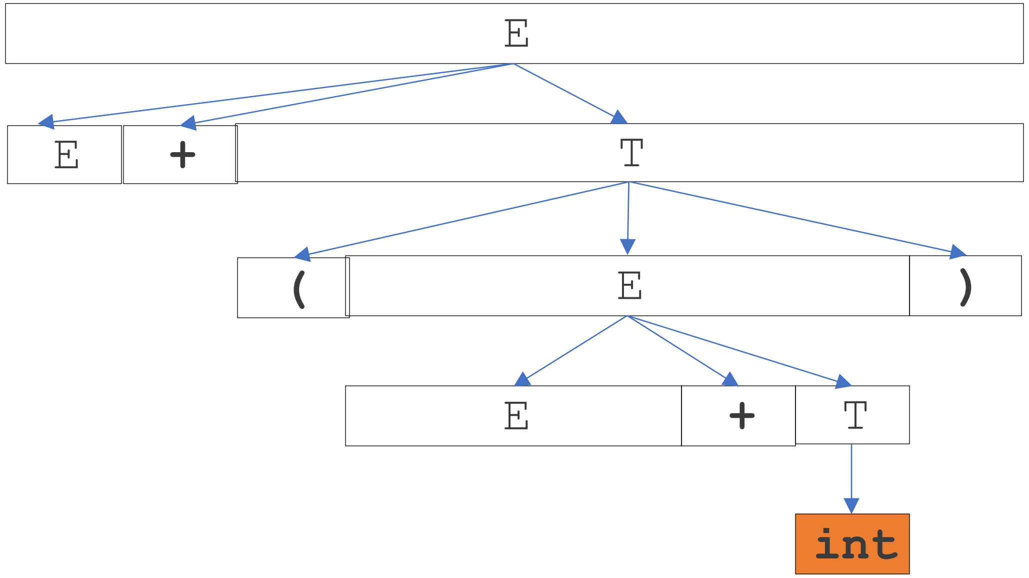
⇒ E + (E)

⇒ E + T

⇒ E

Bottom-up parsing

- An eagle eye of a bottom-up parsing



$\Rightarrow$ **E + (E + int)**

$\Rightarrow$ **E + (E + T)**

$\Rightarrow$ **E + (E)**

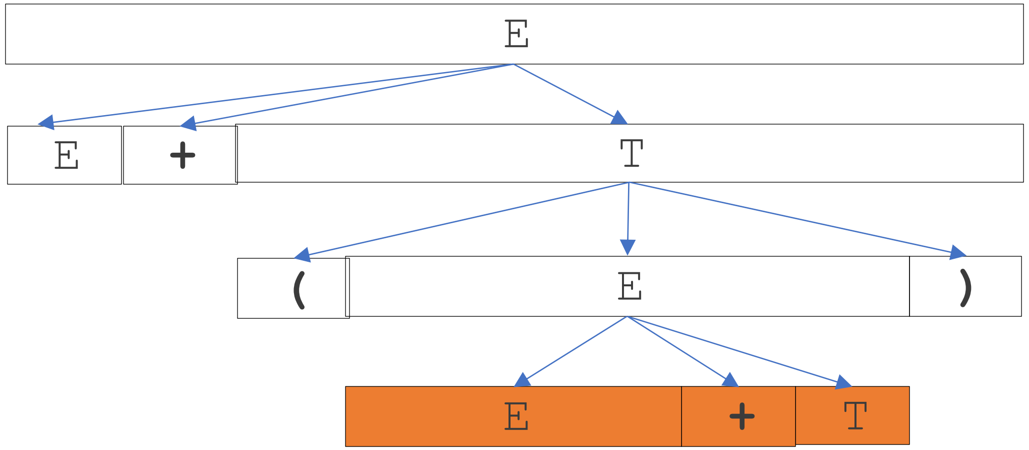
$\Rightarrow$ **E + T**

$\Rightarrow$ **E**

int	+	(	int	+	int	+	int	)

Bottom-up parsing

- An eagle eye of a bottom-up parsing

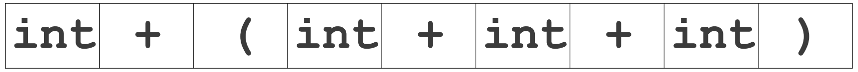


⇒ **E + (E + T)**

⇒ **E + (E)**

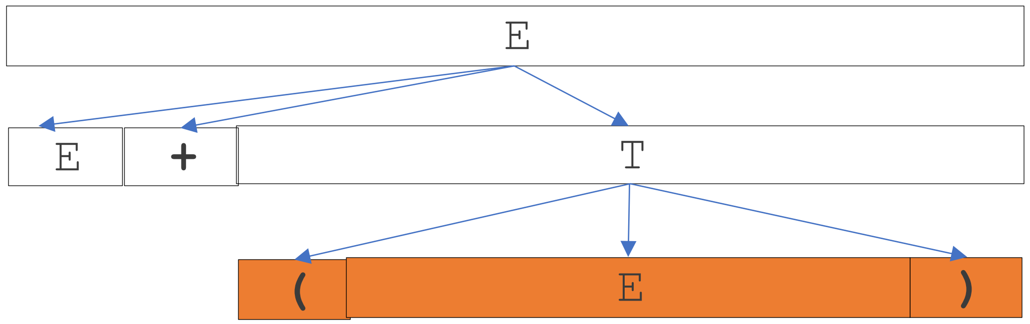
⇒ **E + T**

⇒ **E**



Bottom-up parsing

- An eagle eye of a bottom-up parsing



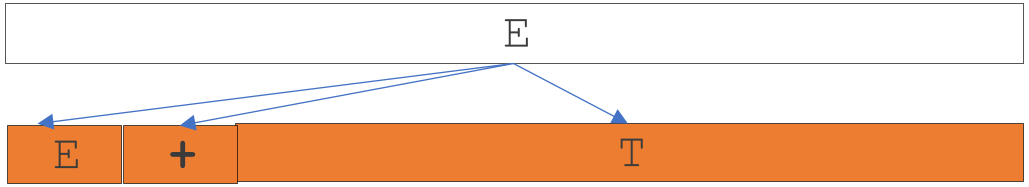
$\Rightarrow$ **E + (E)**
$\Rightarrow$ **E + T**
$\Rightarrow$ **E**

int	+	(	int	+	int	+	int	)

YONSEI UNIVERSITY

Bottom-up parsing

- An eagle eye of a bottom-up parsing



⇒ **E + T**

⇒ **E**

int	+	(	int	+	int	+	int	)

YONSEI UNIVERSITY

Bottom-up parsing

- An eagle eye of a bottom-up parsing

E

$\Rightarrow$ **E**

`int`	`+`	`(`	`int`	`+`	`int`	`+`	`int`	`)`

Bottom-up parsing

- Another example

Grammar:

S ::= **a** A B **e**
A ::= A **b c** | **b**
B ::= **d**

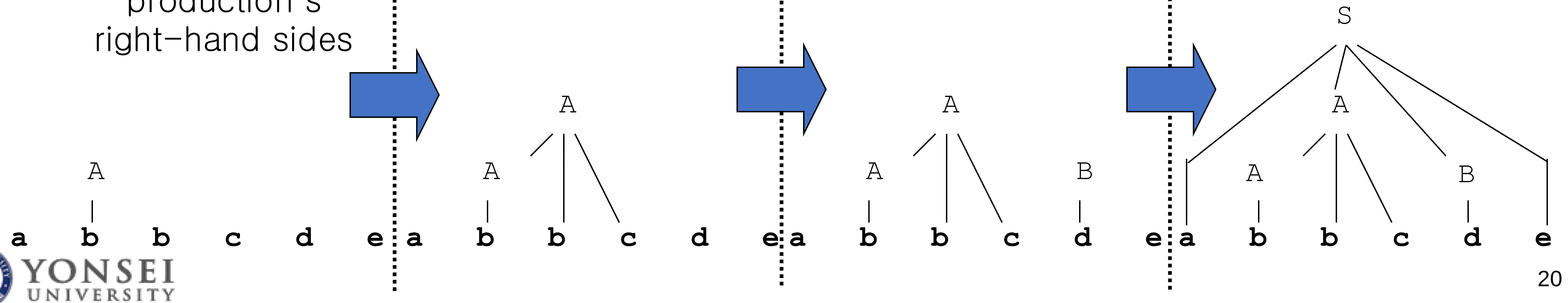
Reducing a sentence:

a b b c d e
a A **b c d e**
a A **d e**
a A B **e**
S

These match production's right-hand sides

Shift-reduce corresponds to a rightmost derivation:

$$
\begin{aligned}
S &\Rightarrow_{rm} \mathbf{a}\ A\ B\ \mathbf{e} \\
&\Rightarrow_{rm} \mathbf{a}\ A\ \mathbf{d}\ \mathbf{e} \\
&\Rightarrow_{rm} \mathbf{a}\ A\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e} \\
&\Rightarrow_{rm} \mathbf{a}\ \mathbf{b}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}
\end{aligned}
$$



Bottom-up parsing

- Handles

Grammar:

```
S ::= a A B e
A ::= A b c | b
B ::= d
```

Reducing a sentence:

```
a b b c d e
a A b c d e
a A d e
a A B e
S
```

Handle

- A *handle* is a substring that connects a right-hand side of the production rule in the grammar and whose reduction to the non-terminal on the left-hand side of that grammar rule is a step along with the reverse of a rightmost derivation

```
a b b c d e
a A b c d e
a A A e
… ?
```

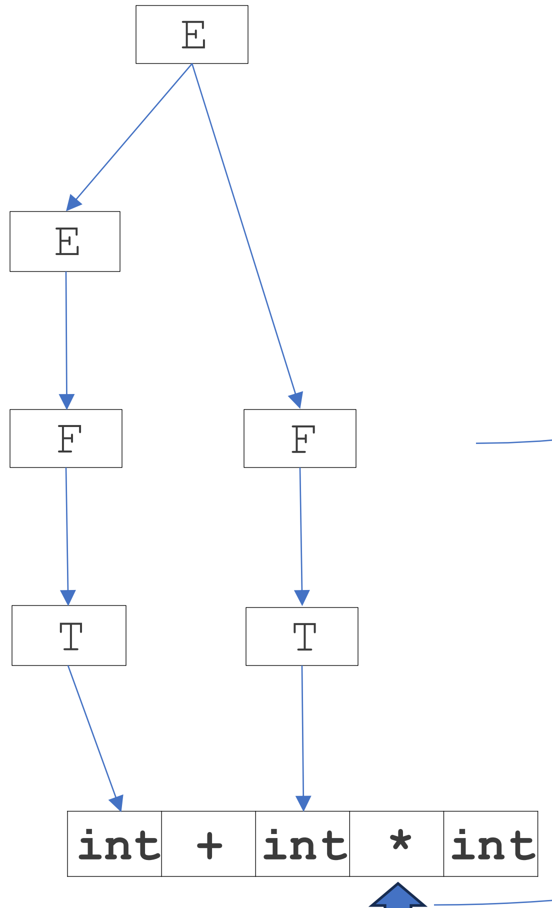
NOT a handle, because further reductions will fail

(result is not a sentential form)

Bottom-up parsing

- Handles

E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



This wasn't a handle

Not available reduction
When handling this

Bottom-up parsing

- Handles
  - Informal definition
    - A substring of the *tree frontier* that matches the right side of a production
    - Even if `A::=`β is a production, β is a handle only if it matches the **frontier at a point** where `A::=`β was used in the derivation
    - β may appear in many other places in the frontier without being a handle for that particular production
  - Formal definition
    - A *handle* of a right-sentential form γ is a production `A ::=` β and a position in γ where β may be replaced by `A` to produce the previous right-sentential form in the rightmost derivation of γ

YONSEI UNIVERSITY

Bottom-up parsing

- Handles
  - In the derivation: $S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abbcde$
    - a**b**bcde is a right sentential form whose handle is `A ::= b` at position 2
    - a**Abc**de is a right sentential form whose handle is `A ::= Abc` at position 4
      - Note: some books take the left of the match as the position

Bottom-up parsing

- **Details of bottom-up parsing**
  - The bottom-up parser reconstructs a **reverse rightmost derivation**
  - Given the rightmost derivation
    $$S \Rightarrow_{rm} \beta_1 \Rightarrow_{rm} \beta_2 \Rightarrow_{rm} \dots \Rightarrow_{rm} \beta_{n-2} \Rightarrow_{rm} \beta_{n-1} \Rightarrow_{rm} \beta_n = w$$
    the parser will first discover $\beta_{n-1} \Rightarrow_{rm} \beta_n$, then $\beta_{n-2} \Rightarrow_{rm} \beta_{n-1}$, etc
  - Parsing terminates when
    - β_1 reduced to S (start symbol, success), or
    - No match can be found (syntax error)

Bottom-up parsing

- **How do we parse?**
  - Key: given what we've already seen and the next input symbol, decide what to do
  - Choices
    - Perform a reduction (**reduce**)
    - Look ahead further (**shift**)
  - Can reduce $A \Rightarrow \beta$ if both followings hold:
    - $A \Rightarrow \beta$ is a valid production
    - $A \Rightarrow \beta$ is a step in *this* rightmost derivation
  - This is known as a *shift-reduce* parser

Bottom-up parsing

- Shift-reduce parser
  - Handle-pruning, bottom-up parser
  - Shift-reduce parsers use a stack and an input buffer
  - Key Data structures
    - A stack holding the frontier of the tree
      - Token, nonterminal symbol, states
    - A string with the remaining input
  - Key operations
    - *Reduce* – if the top of the stack is the right side of a handle `A::=`β, pop the right side β and push the left side `A`
    - *Shift* – push the next input symbol onto the stack
    - *Accept* – announce success
    - *Error* – syntax error discovered

YONSEI UNIVERSITY

Bottom-up parsing

- Shift-reduce parser process
  - 1. initialize stack with $ (empty stack)
  - 2. Repeat until the top of the stack is the goal symbol (start symbol) and the input token is $
    - a) find the handle: if we don't have a handle on top of the stack, shift an input symbol onto the stack
    - b) prune the handle: if we have a handle `A ::= β` on the stack, reduce
      - i) pop `β` symbols off the stack
      - ii) push `A` onto the stack

Shift-reduce parser components

Parsing stack	Input	Action
`$`	`InputString$`	
…	…	
…	…	
`$ StartSymbol`	`$`	`Accept`

YONSEI UNIVERSITY

Bottom-up parsing

- Examples – shift reduce parser

Grammar:

E′ ::= E

E ::= E + n | n

Parsing stack	Input	Action
$	n+n$	Shift
$n	+n$	Reduce E ::= n
$E	+n$	Shift
$E+	n$	Shift
$E+n	$	Reduce E ::= E + n
$E	$	Reduce E′::= E
$E′	$	Accept

$$E' \Rightarrow_{rm} E \Rightarrow_{rm} E+n \Rightarrow_{rm} n+n$$

YONSEI UNIVERSITY

Bottom-up parsing

- Examples – shift reduce parser

Grammar:

S′ ::= S

S ::= (S) S | ε

Parsing stack	Input	Action
$	()$	Shift
$(	)$	Reduce S ::= ε
$(S	)$	Shift
$(S)	$	Reduce S ::= ε
$(S)S	$	Reduce S ::= (S)S
$S	$	Reduce S′::= S
$S′	$	Accept

$$S' \Rightarrow_{rm} S \Rightarrow_{rm} (S)S \Rightarrow_{rm} (S) \Rightarrow_{rm} ()$$

YONSEI UNIVERSITY

Bottom-up parsing – self-study slide

- Examples – shift reduce parser

Grammar:

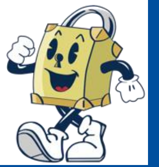
S ::= aABe

A ::= Abc | b

B ::= d

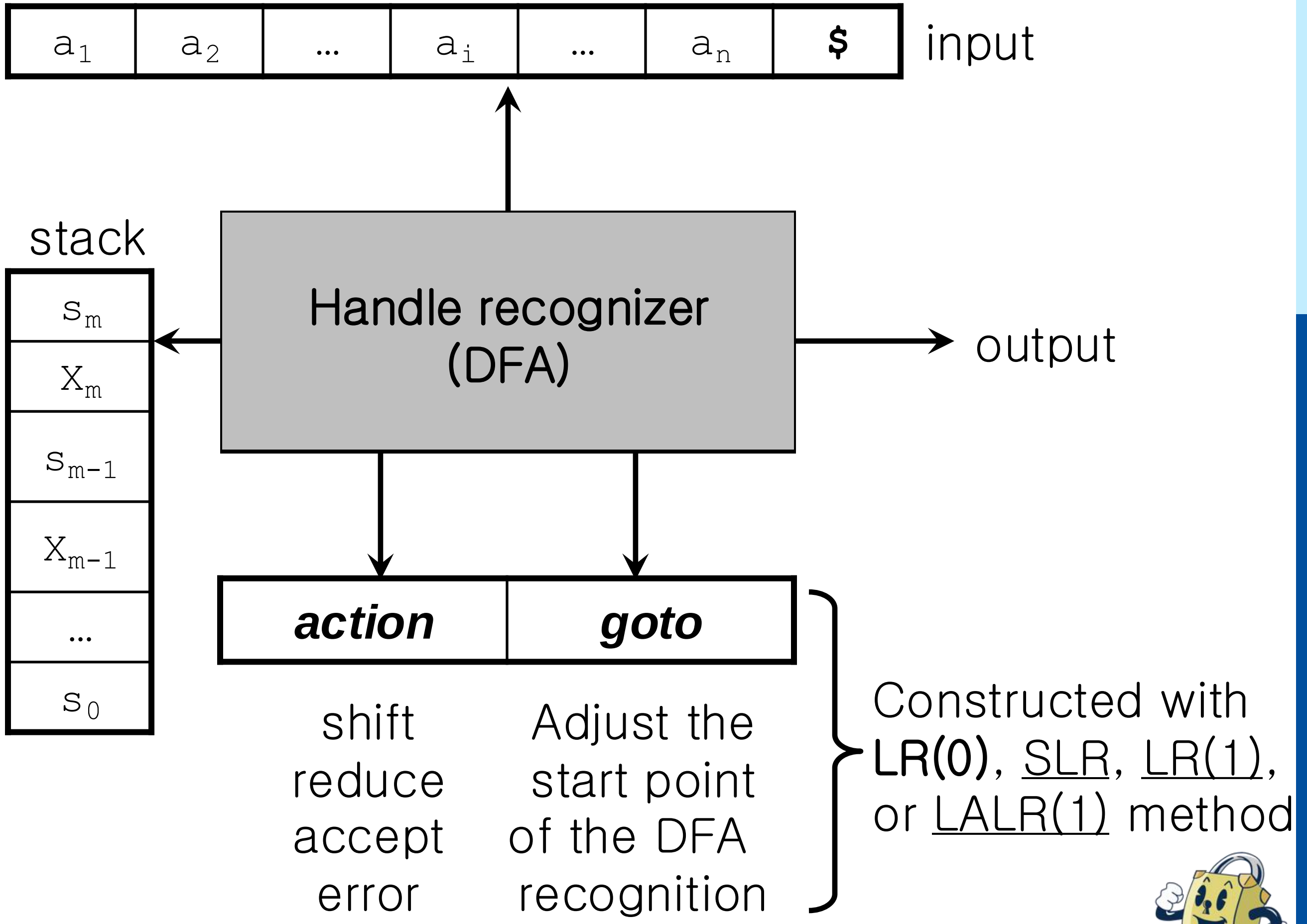
Parsing stack	Input	Action
$	abbcde$	Shift
$a	bbcde$	Shift
$ab	bcde$	Reduce A ::= b
$aA	bcde$	Shift
$aAb	cde$	Shift
$aAbc	de$	Reduce A ::= Abc
$aA	de$	Shift
$aAd	e$	Reduce B ::= d
$aAB	e$	Shift
$aABe	$	Reduce S ::= aABe
$S	$	Accept

YONSEI UNIVERSITY



Bottom-up parsing

- LR state machine
  - Idea
    - Build a DFA that recognizes handles
  - Is it possible?
    - Language generated by a CFG is generally not regular
    - However, language of handles for a CFG is regular
      - So, a DFA can be used to recognize handles
    - Parser reduces when DFA accepts

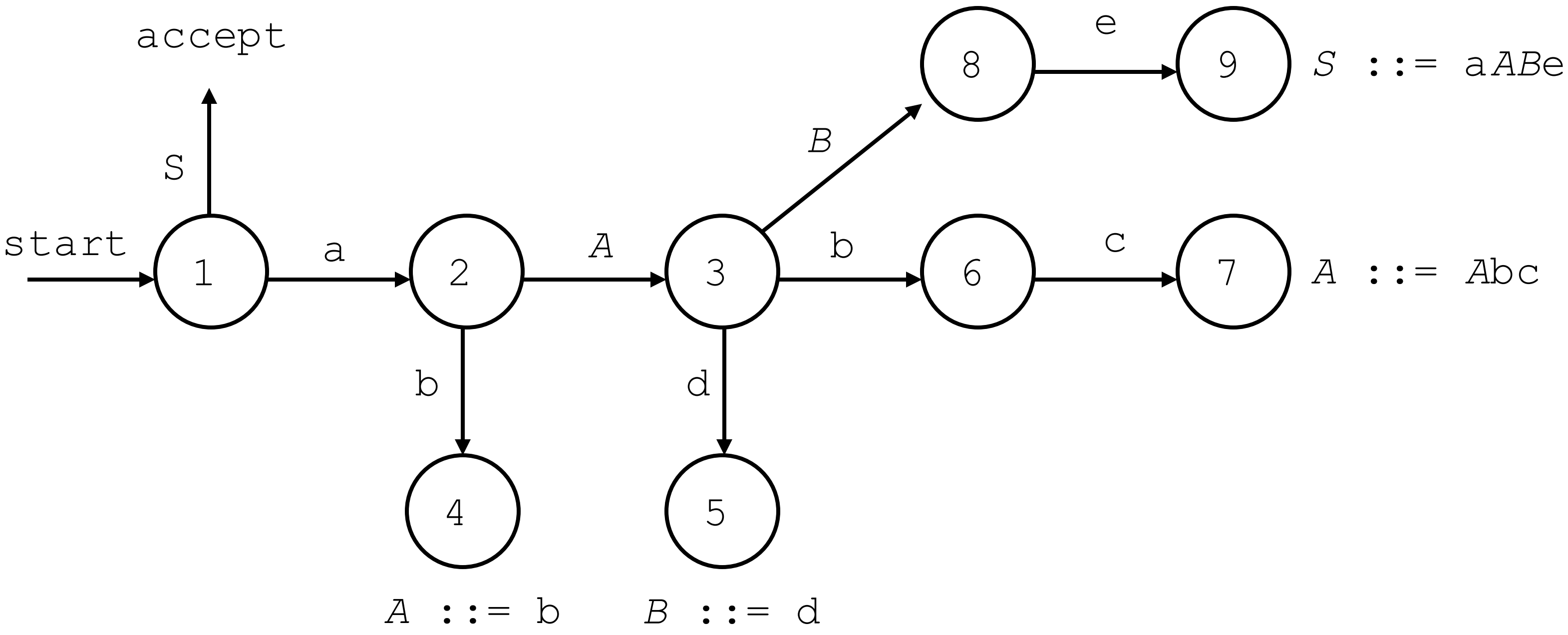


Bottom-up parsing

- Example – DFA to recognize handles

Grammar:

```
S ::= aABe
A ::= Abc | b
B ::= d
```



Bottom-up parsing

- Example – DFA to recognize handles

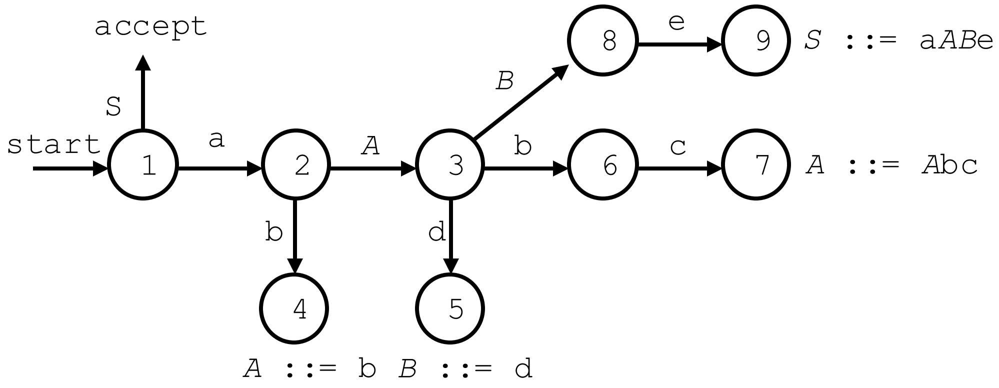
Grammar:

S ::= aABe

A ::= Abc | b

B ::= d

Stack	Input
$	abbcde$
$a	bbcde$
$ab	bcde$
$aA	bcde$
$aAb	cde$
$aAbc	de$
$aA	de$
$aAd	e$
$aAB	e$
$aABe	$
$S	$



Bottom-up parsing

- **DFA to recognize handles**
  - **Viable prefix**
    - A prefix of a right- sentential form that can appear on the stack of the shift-reduce parser
    - Equivalently, a prefix of a right sentential form that does not continue past the right end of the rightmost handle of that sentential form
  - **Idea**
    - Construct a **DFA** to **recognize viable prefixes** given the stack and remaining input
      - *Let's ignore how the DFA can be constructed at this moment (later slides will handle it)*
    - **Perform reductions** when we recognize them

YONSEI UNIVERSITY

Bottom-up parsing

- **Problems in the DFA using viable prefix**
  - Cannot perform linear time parsing
    - Way too much backtracking
    - We want the parser to run in time proportional to the length of the input

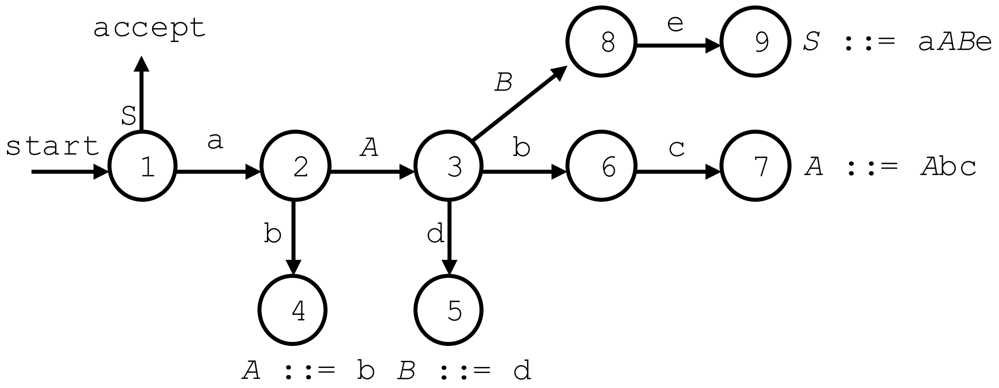
Grammar:

S ::= aABe

A ::= Abc | b

B ::= d

Stack	Input
$	abbcde$
$a	bbcde$
$ab	bcde$
$aA	bcde$
$aAb	cde$
$aAbc	de$
$aA	de$
$aAd	e$
$aAB	e$
$aABe	$
$S	$



Bottom-up parsing

- **Problems in the DFA using viable prefix**
  - What's the reason?
    - After a reduction, the contents of the stack are the same as before except for the new non-terminal on top
      - Scanning the stack will take us through the same transitions as before until the last one
      - If we record state numbers on the stack, we can go directly to the appropriate state when we pop the right-hand side of a production from the stack
  - Change the stack to contain pairs of states and symbols from the grammar

$$\$ s_0 X_1 S_1 X_2 S_2 \ \ldots \ X_n S_n$$

where state s_0 represents the accept state (s_0 will be added depending on presentations)

Observation: in an actual parser, only the state numbers need to be pushed, since they implicitly contain the symbol information, but for explanations, it's clearer to use both

YONSEI UNIVERSITY

Bottom-up parsing

- LR(0) parsing table
  - A shift-reduce parser's DFA can be encoded in two tables (*action* and *goto table*)
  - One row for each state
  - *Action table* encodes what to do given the current state and the next input symbol
  - *Goto table* encodes the transitions to take after a reduction

Bottom-up parsing

- Example – LR(0) parsing table

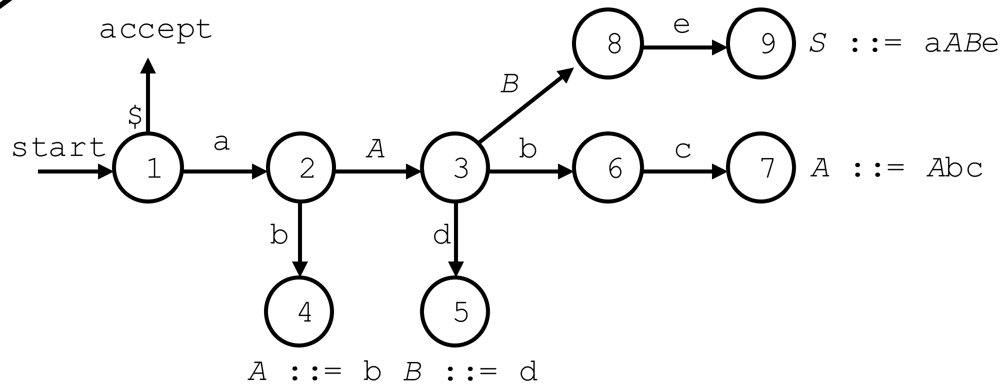
State	*action*						*goto*		
	a	b	c	d	e	$	A	B	S
1	s2					acc			g1
2		s4					g3		
3		s6		s5				g8	
4	r3	r3	r3	r3	r3	r3			
5	r4	r4	r4	r4	r4	r4			
6			s7						
7	r2	r2	r2	r2	r2	r2			
8					s9				
9	r1	r1	r1	r1	r1	r1			

Shift & goto 4

Reduce by production #1

Grammar:

1. S ::= aABe
2. A ::= Abc
3. A ::= b
4. B ::= d



Grammar:

1. S ::= aABe
2. A ::= Abc
3. A ::= b
4. B ::= d

State	action						goto		
	a	b	c	d	e	$	A	B	S
1	s2					acc			g1
2		s4					g3		
3		s6		s5				g8	
4	r3	r3	r3	r3	r3	r3			
5	r4	r4	r4	r4	r4	r4			
6			s7						
7	r2	r2	r2	r2	r2	r2			
8					s9				
9	r1	r1	r1	r1	r1	r1			

Stack	Input
$ 1	abbcde$
$ 1 a 2	bbcde$
$ 1 a 2 b 4	bcde$
$ 1 a 2 A 3	bcde$
$ 1 a 2 A 3 b 6	cde$
$ 1 a 2 A 3 b 6 c 7	de$
$ 1 a 2 A 3	de$
$ 1 a 2 A 3 d 5	e$
$ 1 a 2 A 3 B 8	e$
$ 1 a 2 A 3 B 8 e 9	$
$ 1 S 1	$

Bottom-up parsing

- Action table in LR(0)
  - Given the current state and input symbol, the main possible actions are
    - **Shift** (s_i): shift the input symbol and state `i` onto the stack (i.e., shift and move to state `i`)
    - **Reduce** (r_j): reduce using grammar production `j`
      - The production number tells us how many `<symbol,state>` pairs to pop off the stack
    - **Accept**: accept the string
    - **Noop** (no transition): syntax error during parsing
      - The parser will detect an error as soon as possible on a left-to-right scan
      - A real compiler needs to produce an error message, recover, and continue parsing when this happens

State	*action*						*goto*		
	a	b	c	d	e	$	A	B	S
1	s2					acc			g1
2		s4					g3		
3		s6		s5				g8	
4	r3	r3	r3	r3	r3	r3			
5	r4	r4	r4	r4	r4	r4			
6			s7						
7	r2	r2	r2	r2	r2	r2			
8					s9				
9	r1	r1	r1	r1	r1	r1			

YONSEI UNIVERSITY

Bottom-up parsing

Grammar:
1. S ::= aABe
2. A ::= Abc
3. A ::= b
4. B ::= d

Stack	Input
$ 1	abbcde$
$ 1 a 2	bbcde$
$ 1 a 2 b 4	bcde$
$ 1 a 2 A 3	bcde$
$ 1 a 2 A 3 b 6	cde$
$ 1 a 2 A 3 b 6 c 7	de$
$ 1 a 2 A 3	de$
$ 1 a 2 A 3 d 5	e$
$ 1 a 2 A 3 B 8	e$
$ 1 a 2 A 3 B 8 e 9	$
$ 1 S 1	$

- Action table in LR(0)
  - Configuration (= LR parser state): $(\underbrace{s_0X_1s_1X_2s_2 \ldots X_ms_m}_{\textit{stack}},\ \underbrace{a_ia_{i+1} \ldots a_n\$}_{\textit{input}})$
  - If action(s_m,a_i)=shift s then push a_i, push s, and advance input: $(s_0X_1s_1X_2s_2 \ldots X_ms_m\mathbf{a_i s},\ \mathbf{a_{i+1}} \ldots a_n\$)$
  - If action(s_m,a_i)=reduce A::=β and goto(s_{m-r},A)=s with r=|β| then pop 2r symbols, push A, and push s: $(s_0X_1s_1X_2s_2 \ldots \mathbf{X_{m-r}s_{m-r}As},\ \mathbf{a_i}a_{i+1} \ldots a_n\$)$
  - If action(s_m,a_i)=accept then stop
  - If action(s_m,a_i)=error (no actions are specified in the table), then attempt recovery

State	*action*						*goto*		
	a	b	c	d	e	$	A	B	S
1	s2					acc			g1
2		s4					g3		
3		s6		s5				g8	
4	r3	r3	r3	r3	r3	r3			
5	r4	r4	r4	r4	r4	r4			
6			s7						
7	r2	r2	r2	r2	r2	r2			
8					s9				
9	r1	r1	r1	r1	r1	r1			

Bottom-up parsing

- **LR(0) states and transitions**
  - Each state encodes
    - The set of all possible productions that we could be looking at, given the current state of the parse, and **where** we are in the right-hand side of each of those productions
    - We introduce "**item**"
  - Item
    - An item is a production with a dot in the right-hand side
    - Example: Items for production `A ::= XY`

      ```
      A ::= .XY
      A ::= X.Y
      A ::= XY.
      ```

    - Idea: The dot represents a position in the production

Bottom-up parsing

- LR(0) states and transitions
  - Closure operation
    - Suppose `I` is a set of items for `G`, then `closure(I)` is defined as
      - Every item in `I` is added to `closure(I)`
      - If `A ::= α·Bβ ∈ closure(I)` & `B ::= γ` exists, then add `B ::= ·γ` to `cloasure(I)`
    - Example
      - ```
        E'::= E
        E ::= E + T | T
        T ::= T * F | F
        F ::= (E) | id
        ```

Start with `I={ E' ::= ·E}`
then

```
closure(I)={ E'::= ·E
             E ::= ·E + T
             E ::= ·T
             T ::= ·T * F
             T ::= ·F
             F ::= ·(E)
             F ::= ·id     }
```
```

Bottom-up parsing

- LR(0) states and transitions
  - Goto operation (with closure)
    - Suppose `I` be a set of items and `X` be a grammar symbol, then `goto(I, X)` is the closure of the set of all items `[A ::= αX·β]` such that `A ::= α·Xβ ∈ I`
    - Example

```
I = {E'::= E·,
     E ::= E·+ T}
```

then

```
goto(I, +) = {    E ::= E + ·T,
                  T ::= ·T * F,
                  T ::= ·F,
                  F ::= ·(E),
                  F ::= ·id     }
```

Bottom-up parsing

- LR(0) states and transitions
  - Example grammar

```
S’::= S $
S ::= ( L )
S ::= x
L ::= S
L ::= L , S
```

  - We add a production `S’` with the original start symbol followed by end of file (`$`)

Bottom-up parsing

- LR(0) states and transitions
  - Initially
    - Stack is empty
    - Input is the righthand side of `S'`, i.e., `S$`
    - Initial configuration is `[S' ::= .S$]`
    - But, since position is just before `S`, we are also just before anything that can be derived from `S`
  - A state is just a set of items
    - Start: an initial set of items
    - Completion (or closure): additional productions whose lefthand side appears to the right of the dot in some item already in the state

```
S'::= .S$
S  ::= .(L)
S  ::= .x
```

start

completion

Grammar:

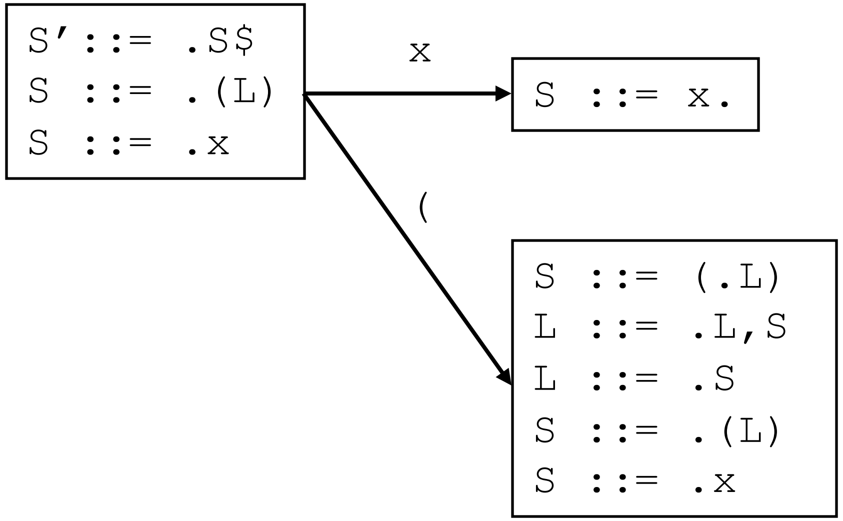
```
S'::= S $
S ::= ( L )
S ::= x
L ::= S
L ::= L , S
```

Bottom-up parsing

- LR(0) states and transitions – shift actions
  - To shift past the `x`, add a new state with the appropriate item(s)
    - In this case, a single item; the closure adds nothing
    - This state will lead to a reduction since no further shift is possible
  - If we shift past the `(`, we are at the beginning of `L`
  - The closure adds all productions that start with `L`, which requires adding all productions starting with `S`

Grammar:

```
S' ::= S $
S  ::= ( L )
S  ::= x
L  ::= S
L  ::= L , S
```

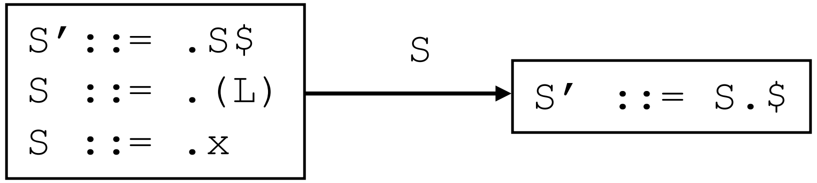


Bottom-up parsing

- LR(0) states and transitions – goto actions
  - Once we reduce `S`, we’ll pop the rhs from the stack exposing the first state
  - Add a *goto* transition on `S` for this

Grammar:

```
S’::= S $
S ::= ( L )
S ::= x
L ::= S
L ::= L , S
```



Bottom-up parsing

- LR(0) states and transitions – basic operations (closure and goto)

- `closure(S)`
  - Adds all items implied by items already in `S`

```
closure(S)
repeat
  for any item [A ::= α.Xβ] in S
    for all productions X ::= γ
      add [X ::= .γ] to S
until S  does not change
return S
```

- `goto(I,X)`
  - `I` is a set of items
  - `X` is a grammar symbol (terminal or non-terminal)
  - Goto moves the dot past the symbol `X` in all appropriate items in set `I`

```
goto(I,X)
set new to the empty set
for each item [A ::= α.Xβ] in I
  add [A ::= αX.β] to new
return closure (new)
```

It may create a new state, or may return an existing one

Bottom-up parsing

- Building the LR(0) parser – LR(0) construction
  - First, augment the grammar with an extra start production `S' ::= S$`
  - Let `T` be the set of states
  - Let `E` be the set of edges
  - Initialize `T` to Closure (`[S' ::= .S$]`)
  - Initialize `E` to empty
  - Note: for symbol `$`, we don't compute `goto(I, $)`; instead, we make this an *accept* action.

```
repeat
  for each state I in T
    for each item [A ::= α.Xβ] in I
      Let new be goto (I, X)
      Add new to T if not present
      Add I -X-> new  to E if not present
until E and T do not change in this iteration
```

Bottom-up parsing

- Building the LR(0) parser – LR(0) reduce actions

```
Initialize R to empty
for each state I in T
  for each item [A ::= α.] in I
    add (I,A ::= α) to R
```

Bottom-up parsing

- **Building the LR(0) parser**
  - For each edge $i \xrightarrow{X} j$
    - if X is a terminal, put s_j in column X, row i of the action table (shift to state j)
    - If X is a non-terminal, put g_j in column X, row i of the goto table
  - For each state i containing an item [S'::= S.$], put accept in column $ of row i
  - Finally, for any state containing [A ::= γ.] put action r_n in every column of row i in the table, where n is the production number

Bottom-up parsing

- Examples – DFA with items (LR parser)

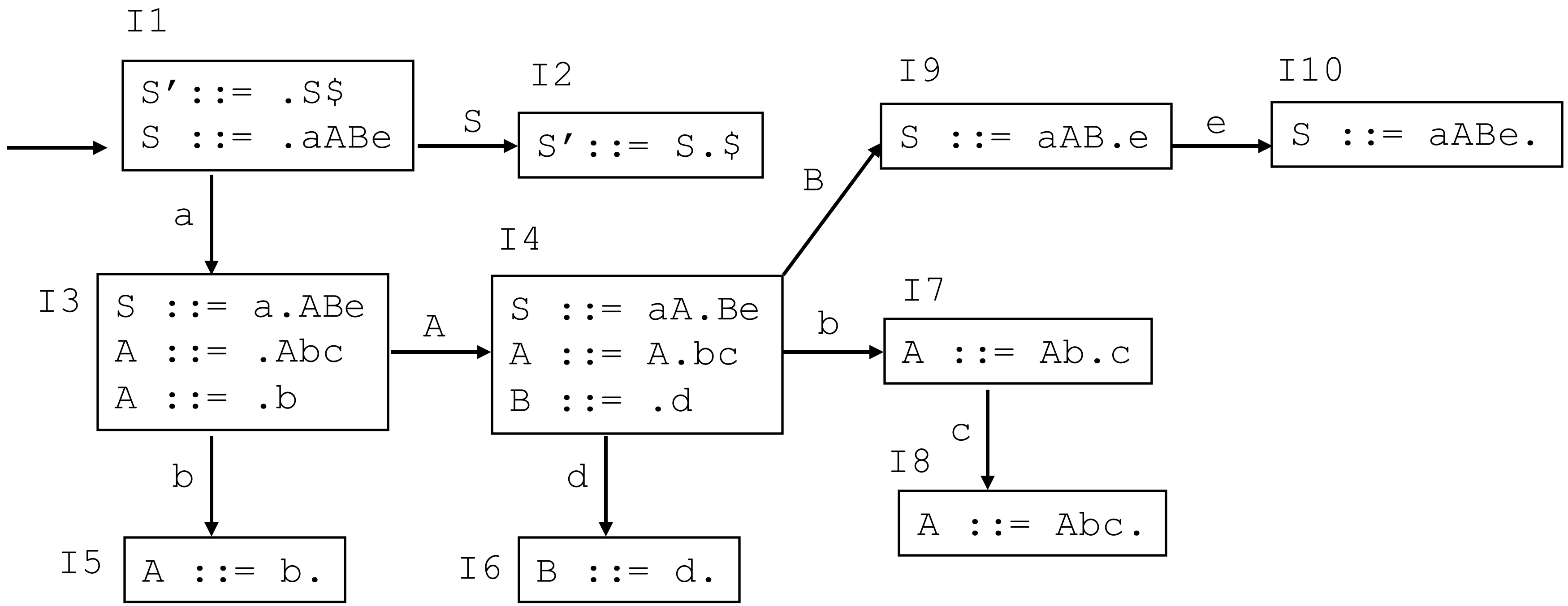
Grammar:

S’::= S$

S ::= aABe

A ::= Abc | b

B ::= d



Bottom-up parsing

- **Problems with grammars**
  - Grammars can cause to problems when constructing the parsing table
    - Shift-reduce conflicts
    - Reduce-reduce conflicts

Bottom-up parsing

- Shift-reduce conflicts
  - Situation: both a shift and a reduce are possible at a given point in the parse
  - Well known example: if-else statement

```
S ::= ifthen S  | ifthen S else S
```

```
I1  S ::= .ifthen S
    S ::= .ifthen S else S
         | ifthen
         v
I2  S ::= ifthen .S
    S ::= ifthen .S else S
         | S
         v
I3  S ::= ifthen S .
    S ::= ifthen S .else S
         | else
         v
I4  S ::= ifthen S else .S
```

Note:

`S ::= .ifthen`

items for `I2-I4` states are ignored to save space

State `I3` has a shift-reduce conflict

1. Can shift past else into state 4 (`s4`)
2. Can reduce (`r1`) `S ::= ifthen S`

Bottom-up parsing

- Fix shift-reduce conflicts
  - Change the grammar
    - In the Java reference grammar and others
  - Use a parse tool with a "*longest match*" rule
    - If there is a conflict, choose to shift instead of reduce
    - Does exactly what most parsers want for if-else case
    - Guideline
      - A few shift-reduce conflicts are fine
      - But be sure they do what you hope to provide

Bottom-up parsing

- Reduce-reduce conflicts
  - Situation: two different reductions are possible in the given state
  - Reduce-reduce conflicts indicate a serious problem with the grammar
  - Example:

```
S ::= A | B
A ::= x
B ::= x
```

```
I1  S ::= .A
    S ::= .B
    A ::= .x
    B ::= .x

      | x
      v

I2  A ::= x.
    B ::= x.
```

State `I2` has a reduce-reduce conflict (`r3, r4`)

Bottom-up parsing

- Fix reduce-reduce conflicts
  - Use a different kind of parser generator that takes *lookahead* information into account when constructing the states
    - LR(1) instead of SLR for example – We will discuss the difference later
    - Most practical tools use this information
  - Fix the grammar

Bottom-up parsing

- Example – fix reduce-reduce conflicts
  - Suppose the grammar separates arithmetic and bool expressions, which create reduce-reduce conflicts

```
expr ::= aexp | bexp
aexp ::= aexp * aident | aident
bexp ::= bexp && bident | bident
aident ::= id
bident ::= id
```

  - How to fix?
    - Possible solution 1: merge `aident` and `bident` into a single non-terminal
    - Possible solution 2: use `id` in place of `aident` and `bident` everywhere they appear
    - Remarkable issues related to two above solutions
      - Both are covering grammars that include some programs that are not generated by the original grammar
      - Use the type checker or other static semantic analysis to filter out illegal programs

Bottom-up parsing

- **Bottom-up parsing and LR parsers**
  - Bottom-up parsing (LR(k) parsing)
    - **L**eft-to-right scanning of **R**ight most derivation with k symbol lookahead (k = 1, if not specified)
  - Benefits of LR Parser
    - LR parser virtually covers all CFG programming languages
    - No backtracking
    - LR parser can detect syntactic errors ASAP
    - LR can recognize all grammars accepted by LL
  - Disadvantage
    - It requires a lot of efforts to construct the parser

YONSEI UNIVERSITY

Questions?