CS335: Bottom-up Parsing

Swarnendu Biswas

Semester 2019-2020-II

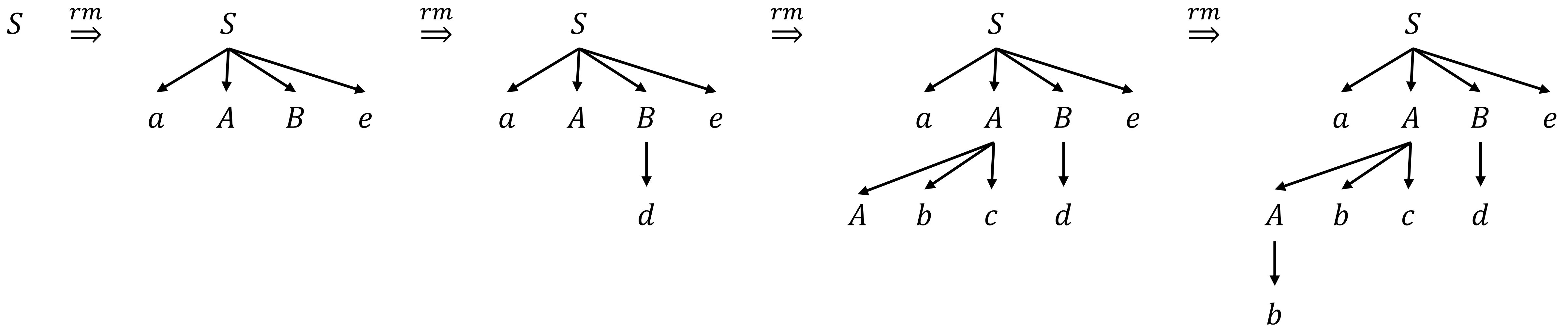
CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

Rightmost Derivation of $abbcde$

$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

Input string: $abbcde$
$S \rightarrow aABe$
$\rightarrow aAde$
$\rightarrow aAbcde$
$\rightarrow abbcde$



Bottom-up Parsing

Constructs the parse tree starting from the leaves and working up toward the root

$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

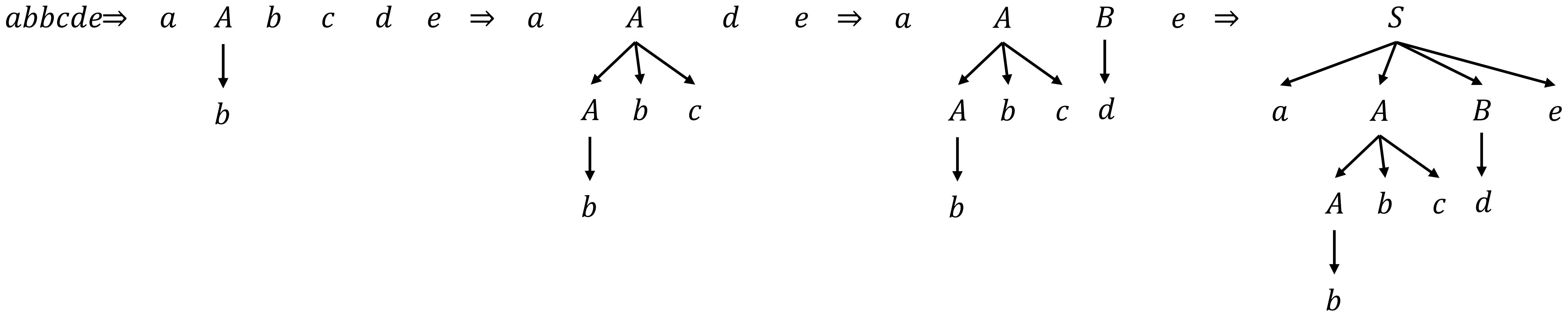
Input string: $abbcde$	
$S \rightarrow aABe$	$abbcde$
$\rightarrow aAde$	$\rightarrow aAbcde$
$\rightarrow aAbcde$	$\rightarrow aAde$
$\rightarrow abbcde$	$\rightarrow aABe$
	$\rightarrow S$

reverse of rightmost derivation

Bottom-up Parsing

$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

Input string: $abbcde$
$abbcde$
$\rightarrow aAbcde$
$\rightarrow aAde$
$\rightarrow aABe$
$\rightarrow S$



Reduction

- Bottom-up parsing **reduces** a string w to the start symbol S
  - At each reduction step, a chosen substring that is the rhs (or body) of a production is replaced by the lhs (or head) nonterminal

Derivation

$$\boxed{S \underset{rm}{\Rightarrow} \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \dots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w}$$

Bottom-up Parser

Handle

- Handle is a substring that matches the body of a production
  - Reducing the handle is one step in the reverse of the rightmost derivation

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

Right Sentential Form	Handle	Reducing Production
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \rightarrow \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \rightarrow F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	$T \rightarrow T * F$
T	T	$E \rightarrow T$

Handle

Although T is the body of the production $E \rightarrow T$, T is not a handle in the sentential form $T * \textbf{id}_2$

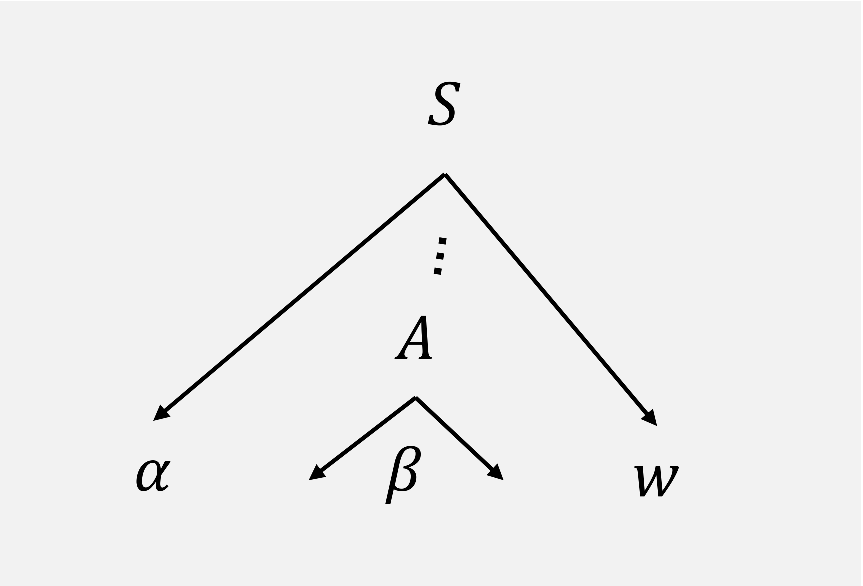
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \textbf{id}$$

Right Sentential Form	Handle	Reducing Production
$\textbf{id}_1 * \textbf{id}_2$	$\textbf{id}_1$	$F \rightarrow \textbf{id}$
$F * \textbf{id}_2$	F	$T \rightarrow F$
$T * \textbf{id}_2$	$\textbf{id}_2$	$F \rightarrow \textbf{id}$
$T * F$	$T * F$	$T \rightarrow T * F$
T	T	$E \rightarrow T$

Handle

- If $S \underset{rm}{\Rightarrow}^* \alpha A w \underset{rm}{\Rightarrow} \alpha\beta w$, then $A \rightarrow \beta$ is a handle of $\alpha\beta w$

- String w right of a handle must contain only terminals



A handle $A \rightarrow \beta$ in the parse tree for $\alpha\beta w$

Handle

If grammar G is unambiguous, then every right sentential form has only one handle

If G is ambiguous, then there can be more than one rightmost derivation of $\alpha\beta w$

Shift-Reduce Parsing

Shift-Reduce Parsing

- Type of bottom-up parsing with two primary actions, shift and reduce
  - Other obvious actions are accept and error

- The input string (i.e., being parsed) consists of two parts
  - Left part is a string of terminals and nonterminals, and is stored in stack
  - Right part is a string of terminals read from an input buffer
  - Bottom of the stack and end of input are represented by \$

Shift-Reduce Actions

- **Shift**: shift the next input symbol from the right string onto the top of the stack
- **Reduce**: identify a string on top of the stack that is the body of a production, and replace the body with the head

Shift-Reduce Parsing

- **Initial**

Stack	Input
$\$$	$w\$$

Shift $\Downarrow^*$ Reduce

- **Final goal**

Stack	Input
$\$S$	$\$$

Shift-Reduce Parsing

$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to (E) \mid \mathbf{id}$

Stack	Input	Action
$\$$	$\mathbf{id}_1 * \mathbf{id}_2\$$	Shift
$\$\mathbf{id}_1$	$* \mathbf{id}_2\$$	Reduce by $F \to$ id
$\$F$	$* \mathbf{id}_2\$$	Reduce by $T \to F$
$\$T$	$* \mathbf{id}_2\$$	Shift
$\$T *$	$\mathbf{id}_2\$$	Shift
$\$T * \mathbf{id}_2$	$\$$	Reduce by $F \to$ id
$\$T * F$	$\$$	Reduce by $T \to T * F$
$\$T$	$\$$	Reduce by $E \to T$
$\$E$	$\$$	Accept

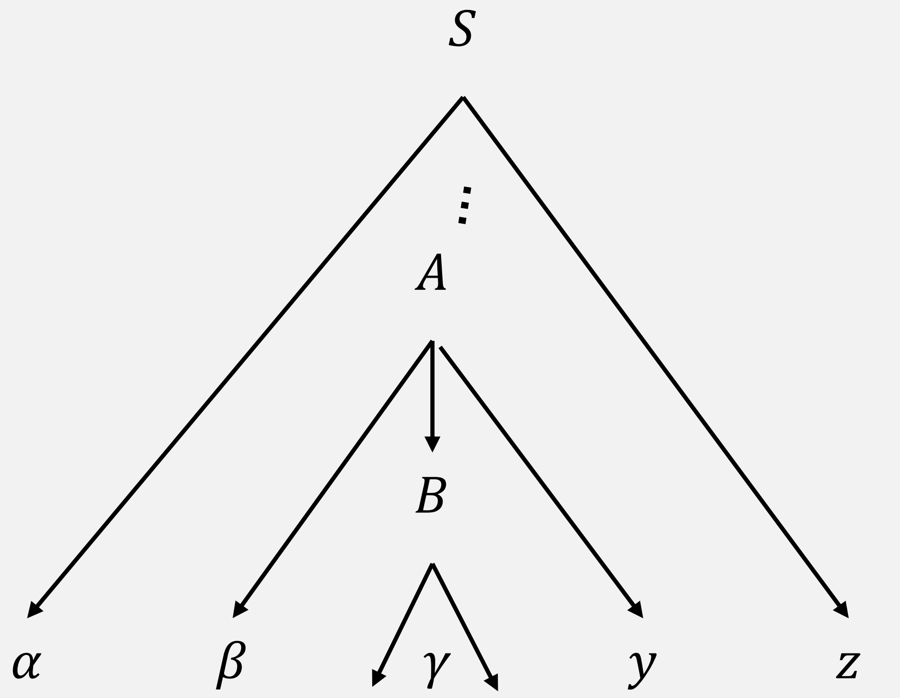
Handle on Top of the Stack

- Is the following scenario possible?

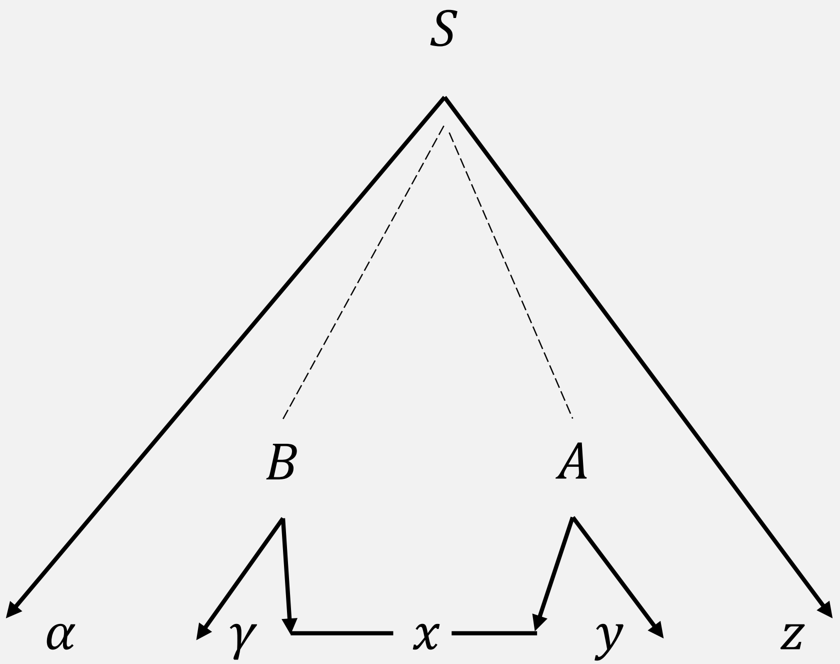
Stack	Input	Action
…		
$\$ \ \alpha\beta\gamma$	$w\$$	Reduce by $A \rightarrow \gamma$
$\$ \ \alpha\beta A$	$w\$$	Reduce by $B \rightarrow \beta$
$\$\alpha B A$	$w\$$	
…		

Possible Choices in Rightmost Derivation

$$1.\ S \underset{rm}{\Rightarrow} \alpha A z \underset{rm}{\Rightarrow} \alpha\beta B y z \underset{rm}{\Rightarrow} \alpha\beta\gamma y z$$



$$2.\ S \underset{rm}{\Rightarrow} \alpha B x A z \underset{rm}{\Rightarrow} \alpha B x y z \underset{rm}{\Rightarrow} \alpha\gamma x y z$$



Handle on Top of the Stack

- Is the following scenario possible?

Stack	Input	Action
…		

Handle always eventually appears on top of the stack, never inside

Shift-Reduce Actions

- Shift: shift the next input symbol from the right string onto the top of the stack
- Reduce: identify a string on top of the stack that is the body of a production, and replace the body with the head

How do you decide when to shift and when to reduce?

Steps in Shift-Reduce Parsers

- The general shift-reduce technique is
  - If there is no handle on the stack then shift
  - If there is a handle then reduce

- Bottom up parsing is essentially the process of detecting handles and reducing them
- Different bottom-up parsers differ in the way they detect handles

Challenges in Bottom-up Parsing

Which action do you pick when there is a choice?	• Both shift and reduce are valid, implies a shift-reduce conflict
Which rule to use if reduction is possible by more than one rule?	• Reduce-reduce conflict

Shift-Reduce Conflict

$E \rightarrow E + T \mid E * E \mid \textbf{id}$

$\textbf{id} + \textbf{id} * \textbf{id}$

Stack	Input	Action
$\$$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	Shift
...		
$\$E + E$	$* \textbf{id}\$$	Reduce by $E \rightarrow E + E$
$\$E$	$* \textbf{id}\$$	Shift
$\$E *$	$\textbf{id}\$$	Shift
$\$E * \textbf{id}$	$\$$	Reduce by $E \rightarrow \textbf{id}$
$\$E * E$	$\$$	Reduce by $E \rightarrow E * E$
$\$E$	$\$$	

$\textbf{id} + \textbf{id} * \textbf{id}$

Stack	Input	Action
$\$$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	Shift
...		
$\$E + E$	$* \textbf{id}\$$	Shift
$\$E + E *$	$\textbf{id}\$$	Shift
$\$E + E * \textbf{id}$	$\$$	Reduce by $E \rightarrow \textbf{id}$
$\$E + E * E$	$\$$	Reduce by $E \rightarrow E * E$
$\$E + E$	$\$$	Reduce by $E \rightarrow E + E$
$\$E$	$\$$	

Shift-Reduce Conflict

$$Stmt \rightarrow \textbf{if}\ Expr\ \textbf{then}\ Stmt$$
$$\quad |\ \textbf{if}\ Expr\ \textbf{then}\ Stmt\ \textbf{else}\ Stmt$$
$$\quad |\ other$$

Stack	Input	Action
… if $Expr$ then $Stmt$	else … $	

Shift-Reduce Conflict

$$\begin{aligned} Stmt \rightarrow & \textbf{ if } Expr \textbf{ then } Stmt \\ & \mid \textbf{if } Expr \textbf{ then } Stmt \textbf{ else } Stmt \\ & \mid other \end{aligned}$$

Stack	Input	Action
… **if** $Expr$ **then** $Stmt$	**else** … $\$$	

What is a correct thing to do for this grammar – shift or reduce?

Reduce-Reduce Conflict

$M \rightarrow R + R \mid R + c \mid R$
$R \rightarrow c$

$c + c$		
Stack	**Input**	**Action**
$\$$	$c + c\$$	Shift
$\$c$	$+c\$$	Reduce by $R \rightarrow c$
$\$R$	$+c\$$	Shift
$\$R +$	$c\$$	Shift
$\$R + c$	$\$$	Reduce by $R \rightarrow c$
$\$R + R$	$\$$	Reduce by $R \rightarrow R + R$
$\$M$	$\$$	

$c + c$		
Stack	**Input**	**Action**
$\$$	$c + c\$$	Shift
$\$c$	$+c\$$	Reduce by $R \rightarrow c$
$\$R$	$+c\$$	Shift
$\$R +$	$c\$$	Shift
$\$R + c$	$\$$	Reduce by $M \rightarrow R + c$
$\$M$	$\$$	

LR Parsing

LR(k) Parsing

- Popular bottom-up parsing scheme
  - L is for left-to-right scan of input
  - R is for reverse of rightmost derivation
  - k is the number of lookahead symbols
- LR parsers are table-driven, like the nonrecursive LL parser
- LR grammar is one for which we can construct an LR parsing table

Popularity of LR Parsing

- LR parsers can recognize all programming language constructs for which context-free grammars can be written
- LR-parsing is most general nonbacktracking shift-reduce parsing method
- LR parsing works for a superset of grammars that can be parsed with predictive or LL parsers
  - LL(k) parsing must predict which production to use having seen only the first k tokens of the right-hand side
  - LR(k) parsing postpones the decision until it has seen input tokens corresponding to the entire right-hand side of the production

Popularity of LR Parsing

Can recognize all language constructs with CFGs

Most general nonbacktracking shift-reduce parsing method

Works for a superset of grammars parsed with predictive or LL parsers

Why?

Popularity of LR Parsing

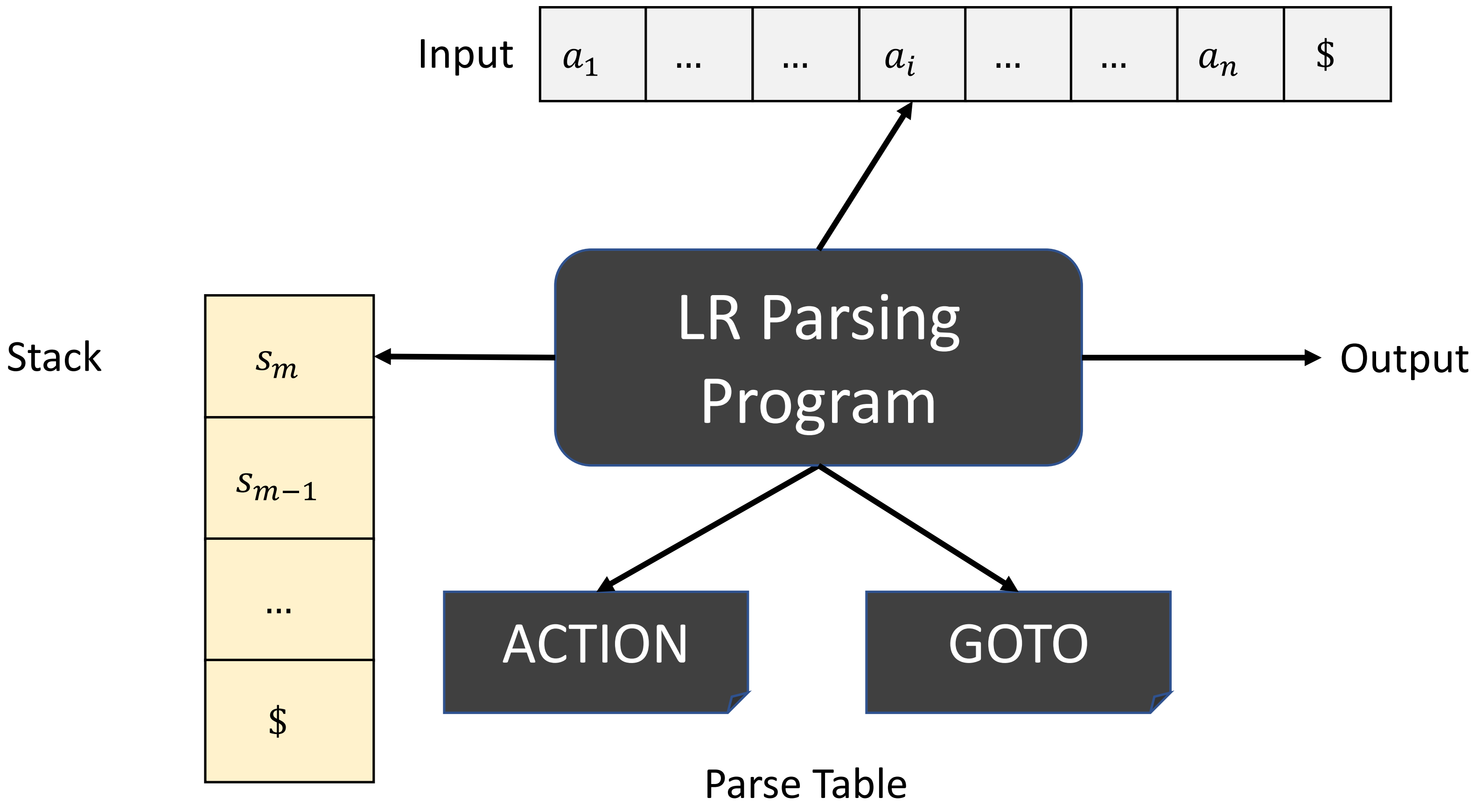
Can recognize all language constructs with CFGs

Most general nonbacktracking shift-reduce parsing method

Works for a superset of grammars parsed with predictive or LL parsers

- LL(k) parsing predicts which production to use having seen only the first k tokens of the right-hand side
- LR(k) parsing can decide after it has seen input tokens corresponding to the entire right-hand side of the production

Block Diagram of LR Parser



LR(0) Item

- An LR(0) item (also called item) of a grammar G is a production of G with a dot at some position in the body

- An item indicates how much of a production we have seen
  - Symbols on the left of "." are already on the stack
  - Symbols on the right of "." are expected in the input

Production	Items
$A \rightarrow XYZ$	$A \rightarrow .XYZ$
	$A \rightarrow X.YZ$
	$A \rightarrow XY.Z$
	$A \rightarrow XYZ.$

$A \rightarrow .XYZ$ indicates that we expect a string derivable from XYZ next on the input

LR(0) Automaton

- An LR parser makes shift-reduce decisions by maintaining states
- States represent sets of LR(0) items
- The canonical LR(0) collection is used for constructing a DFA for parsing

Closure Operation

- Let I be a set of items for a grammar G
- Closure(I) is constructed by
  1. Add every item in I to Closure(I)
  2. If $A \rightarrow \alpha.B\beta$ is in Closure(I) and $B \rightarrow \gamma$ is a rule, then add $B \rightarrow \gamma$ to Closure(I) if not already added
  3. Repeat until no more new items can be added to Closure(I)

Example of Closure

$E' \rightarrow E$
$E \rightarrow E + T | T$
$T \rightarrow T * F | F$
$F \rightarrow (E) | \mathbf{id}$

Suppose $I = \{E' \rightarrow .E\}$, compute Closure($I$)

Example of Closure

$$E' \rightarrow E$$
$$E \rightarrow E + T | T$$
$$T \rightarrow T * F | F$$
$$F \rightarrow (E) | \mathbf{id}$$

Suppose $I = \{E' \rightarrow .E\}$

Closure$(I) = \{$

$E' \rightarrow .E,$

$E \rightarrow .E + T,$

$E \rightarrow .T,$

$T \rightarrow .T * F,$

$T \rightarrow .F,$

$F \rightarrow .(E),$

$F \rightarrow .\mathbf{id}$

$\}$

Goto Operation

- Suppose I is a set of items and X is a grammar symbol
- Goto(I,X) is the closure of set all items $[A \rightarrow \alpha X.\beta]$ such that $[A \rightarrow \alpha.X\beta]$ is in I
  - If I is a set of items for some valid prefix α then Goto(I,X) is set of valid items for prefix αX

- Intuitively, Goto() defines the transitions in the LR(0) automaton

Example of Goto

- Suppose $I = \{E' \rightarrow E., E \rightarrow E.+T\}$
- Compute Goto(I, $+$)

Example of Goto

$$E' \rightarrow E$$
$$E \rightarrow E + T | T$$
$$T \rightarrow T * F | F$$
$$F \rightarrow (E) | \mathbf{id}$$

- Suppose $I = \{E' \rightarrow E., E \rightarrow E.+T\}$

Goto(I, +) = {

$E \rightarrow E+.T,$

$E \rightarrow E+.T,$

$T \rightarrow .T * F,$

$T \rightarrow .F,$

$F \rightarrow .(E),$

$F \rightarrow .\mathbf{id}$

}

Canonical Collection of Sets of LR(0) Items

C = Closure($\{S' \rightarrow S\}$)

repeat

    for each set of items I in C

        for each grammar symbol X

            if (Goto(I, X) is not empty and not in C)

                add GOTO(I, X) to C

until no new sets of items are added to C

Canonical Collection of Sets of LR(0) Items

$$
\begin{aligned}
&E' \rightarrow E \\
&E \rightarrow E + T | T \\
&T \rightarrow T * F | F \\
&F \rightarrow (E) | \mathbf{id}
\end{aligned}
$$

- Compute the canonical collection for the expression grammar

Canonical Collection of Sets of LR(0) Items

I_0 =Closure($E' \rightarrow E$)= {
$E' \rightarrow .E,$
$E \rightarrow .E + T,$
$E \rightarrow .T,$
$T \rightarrow .T * F,$
$T \rightarrow .F,$
$F \rightarrow .(E),$
$F \rightarrow .\mathbf{id},$
}

I_1 =Goto(I_0, E) = {
$E' \rightarrow E.,$
$E \rightarrow E.+T$
}

I_2 =Goto(I_0, T)= {
$E \rightarrow T.,$
$T \rightarrow T.* F$
}

I_3 =Goto(I_0, F)= {
$T \rightarrow F.$
}

I_5 =Goto($I_0, \mathbf{id}$)= {
$F \rightarrow \mathbf{id}.$
}

I_4 =Goto(I_0, "(")= {
$F \rightarrow (.E),$
$E \rightarrow .E + T,$
$E \rightarrow .T,$
$T \rightarrow .T * F,$
$T \rightarrow .F,$
$F \rightarrow .(E),$
$F \rightarrow .\mathbf{id},$
}

I_7 =Goto(I_2,$*$) = {
$T \rightarrow T *.F,$
$F \rightarrow .(E),$
$F \rightarrow .\mathbf{id}$
}

Canonical Collection of Sets of LR(0) Items

I_6 =Goto($I_1, +$)= {
$E \rightarrow E+.T,$
$T \rightarrow .T * F,$
$T \rightarrow .F,$
$F \rightarrow .(E),$
$F \rightarrow .\mathbf{id},$
}

I_8 =Goto(I_4, E)= {
$E \rightarrow E.+T,$
$F \rightarrow (E.)$
}

I_9 =Goto(I_6, T)= {
$E \rightarrow E + T.,$
$T \rightarrow T.* F$
}

I_{10} =Goto(I_7, F)= {
$T \rightarrow T * F.,$
}

I_{11} =Goto($I_8,)$)= {
$F \rightarrow (E).$
}

I_2 =Goto(I_4, T)
I_3 =Goto(I_4, F)
I_4 =Goto(I_4, "(")
I_5 =Goto($I_4, \mathbf{id}$)
I_3 =Goto(I_6, F)
I_4 =Goto(I_6, "(")
I_5 =Goto($I_6, \mathbf{id}$)
I_4 =Goto(I_7, "(")
I_5 =Goto($I_7, \mathbf{id}$)
I_6 =Goto($I_8, +$)
I_7 =Goto($I_9, *$)