Parsing - 2

- **LR(0) parsing**
  - **Closures and goto sets**
- **SLR parsing**
  - **Using FOLLOW sets**
- **LR(k) parsing**
  - **Using lookaheads**

LR(0) parsing

- **LR(k) parsing**
  - **Left-to-right parse, Rightmost derivation, k-token lookahead**
  - **Recognize virtually all real programming languages**
  - **Detects a syntax error as soon as possible in a left to right scan of the input stream**
  - **Most powerful shift-reduce parsing method, yet efficient to implement**

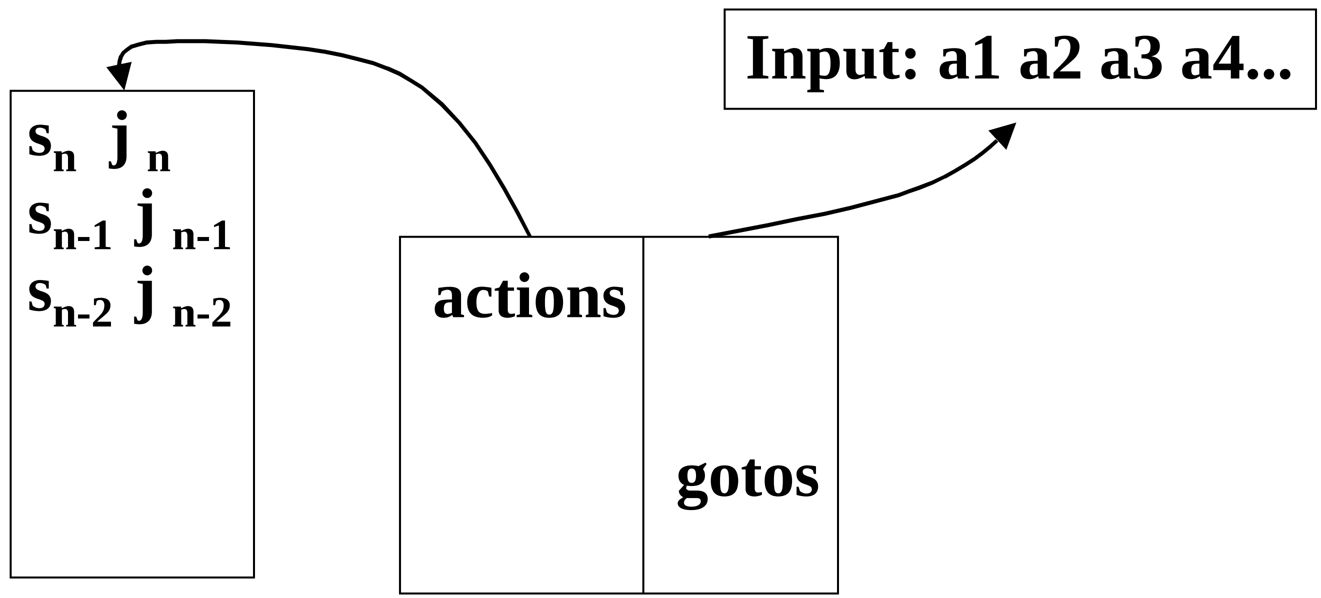
Building a Parser

- **How to build the DFA which is the decision maker for the stack parser in last lecture?**
  - **Need a stack which takes <state,symbol> pairs**
  - **Transition table contains four kinds of actions:**
    - **shift into state *n* *(s n)***
    - **reduce by rule *y* with lefthandside X and then goto state *m (r y + goto entry* when *X* on top of stack*) ;* this is where actions occur**
    - **accept**
    - **error**

LR(0) Parsing

- **Given parser in state *s* and token *j* is next, parser does action [*s*, *j*] in transition table**

stack with
state, symbol
pairs



LR parser

Example

- **Start with distinguished symbol rule and build start state**

  S' $\rightarrow$ S *grammar*

  S $\rightarrow$ a S b | ab

  I_0 : S' $\rightarrow$. S *start state*

- **Then add in closure items**

  I_0 : S' $\rightarrow$. S

  S $\rightarrow$. aSb

  S $\rightarrow$. ab

- **Now look for states to transition to on inputs or to goto if top of stack is a nonterminal**

Example

- **Transition from I_0 on an *a* to I_1**

I_1 : $S \rightarrow a\,.\,S\,b$
$S \rightarrow a\,.\,b$

- **Now add in closure items to complete I_1**

I_1 : $S \rightarrow a\,.\,S\,b$
$S \rightarrow a\,.\,b$
$S \rightarrow .\,aSb$
$S \rightarrow .\,ab$

- **Continue like this until have all the states and transitions**

Example

A very simple grammar:
S' → S
S → a S b | ab

First, build states from items.

I_0: S' → . S
S → . aSb
S → . ab

closure(S)

$I_1 = goto(I_0, a)$
$I_1 = goto(I_1, a)$

I_1: S → a . S b
S → a . b
S → . aSb
S → . ab

closure(S)

I_3: S' → S .

$I_3 = goto(I_0, S)$

$I_2 = goto(I_1, S)$

I_2: S → a S . b

$I_4 = goto(I_1, b)$

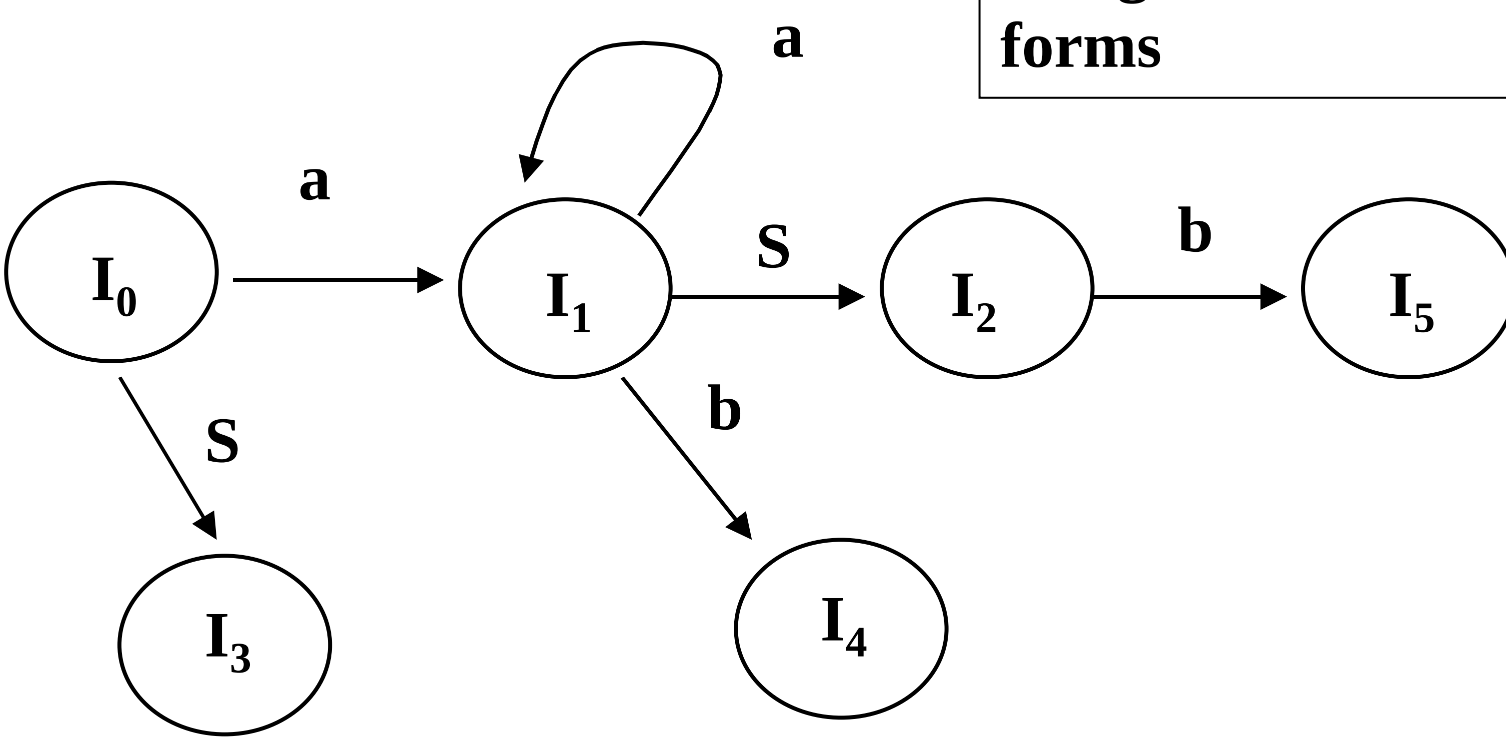
I_4: S → a b.

I_5: S → a S b .

$I_5 = goto(I_2, b)$

Example

Recognizes prefixes of right sentential forms



Encoding the parser

states\inputs	actions			gotos
	a	b	$	S
0	s1	_	_	3
1	s1	s4	_	2
2	_	s5	_	
3	_	_	accept	
4	r(iii)	r(iii)	r(iii)	
5	r(ii)	r(ii)	r(ii)	

If $A \rightarrow \alpha\ .\ a\ \beta$ in I_k and $goto(I_k, a) = I_j$ table entry for (k,*a*) is sj for *a* terminal symbol.

If $A \rightarrow \alpha\ .$ in I_k then table entry for (k,*b*) is r(rule#) where *b* is any input symbol.

If $S' \rightarrow S.$ in I_k then table entry for (k,*$*) is accept.

Example

states/inputs	actions			gotos
	a	b	$	S
0	s1	_	_	3
1	s1	s4	_	2
2	_	s5	_	
3	_	_	accept	
4	r(iii)	r(iii)	r(iii)	
5	r(ii)	r(ii)	r(ii)	

stack	input	action
$ 0	aabb$	s1
$ 0 a 1	abb$	s1
$ 0 a 1 a 1	bb$	s4
$ 0 a 1 a 1 b 4	b$	r(iii), goto(1,b) = 2
$ 0 a 1 S 2	b$	s5
$ 0 a 1 S 2 b 5	$	r(ii), goto(0,$)=3
$ 0 S 3	$	accept

SLR(1)

- **Previous parser is called LR(0) because we used no knowledge of the input**
- **SLR(1) is a somewhat stronger parser that adds knowledge about next input symbol**
  - **Sometimes needed to break shift-reduce conflicts**
  - **Need to precompute information about the grammar (from the rules) to use in parsing**

SLR(1)

- ***Follow set*: the set of terminals which can follow a specific nonterminal in a rightmost derivation**
  - **New rule for reduce: only reduce when next input symbol is an element of Follow set of the resulting nonterminal**
  - **In the previous example, we would eliminate reductions in states 4,5 on *a* because this can't be followed by *a***
  - **Follow sets are used also in top down parsing**

Shift/Reduce Conflict

$S' \rightarrow S$
$S \rightarrow A\ b \mid d\ c \mid b\ A\ c$
$A \rightarrow d$

A very simple language = {db, dc, bdc}

Follow(S) = {\$}, Follow(A) = {b,c}

Form part of the SLR(1) parser:

I_0 : $S' \rightarrow .\, S$
$S \rightarrow .A\ b$
$S \rightarrow .dc$
$S \rightarrow .b\ A\ c$
$A \rightarrow .d$

I_1 : $S \rightarrow d\ .\ c$
$A \rightarrow d.$

But since *c* is in Follow(A), we don't know whether to reduce or shift in state I_1 if *c* is next input symbol!

Deriv1: $S' \rightarrow S \rightarrow dc$; Deriv2; $S' \rightarrow S \rightarrow bAc \rightarrow bdc$

Reduce/Reduce Conflict

$S' \rightarrow S$
$S \rightarrow b\ A\ e \mid b\ B\ d \mid A\ c$
$A \rightarrow d$
$B \rightarrow E\ c$
$E \rightarrow d$

Deriv1: $S' \rightarrow S \rightarrow Ac \rightarrow dc$
Deriv2: $S' \rightarrow S \rightarrow bBd \rightarrow bEcd \rightarrow bdcd$
Deriv3: $S' \rightarrow S \rightarrow bAe \rightarrow bde$

$I_0: S' \rightarrow .\ S$
$S \rightarrow .\ b\ A\ e$
$S \rightarrow .\ b\ B\ d$
$S \rightarrow .\ A\ c$
$A \rightarrow .d$

$I_1: S \rightarrow b\ .\ A\ e$
$S \rightarrow b\ .\ B\ d$
$A \rightarrow .\ d$
$B \rightarrow .\ E\ c$
$E \rightarrow .\ d$

$I_2: A \rightarrow d.$
$E \rightarrow d.$

Which reduction to take?
Follow set too imprecise here to decide.

LR(k)

- **Solution: keep more information about what next input symbol can be on any parse**
  - **Idea: keep an input *lookahead* as part of each item**
  - **More precise than Follow sets which essentially union these lookaheads for nonterminal A over all sentential forms in which the A appears**
  - **Potentially gives rise to much bigger parsers than SLR(1) (more states)**

LR(k)

- **LR(k) looks k symbols ahead into the input**
- **There are some grammars which are not parsable with only k lookahead symbols**
- **Most computer programming languages are LR(k)**

LR(1) Idea

$S' \rightarrow S$
$S \rightarrow b\ A\ e \mid b\ B\ d \mid A\ c$
$A \rightarrow d$
$B \rightarrow E\ c$
$E \rightarrow d$

Deriv1: $S' \rightarrow S \rightarrow Ac \rightarrow dc$
Deriv2: $S' \rightarrow S \rightarrow bBd \rightarrow bEcd \rightarrow bdcd$
Deriv3: $S' \rightarrow S \rightarrow bAe \rightarrow bde$

I_0 : $S' \rightarrow .\ S,\ \$$
$S \rightarrow .\ b\ A\ e,\ \$$
$S \rightarrow .\ b\ B\ d,\ \$$
$S \rightarrow .\ A\ c,\ \$$
$A \rightarrow .d,c$

I_1 : $S \rightarrow b.\ A\ e,\$$
$S \rightarrow b\ .\ B\ d,\$$
$A \rightarrow .d,\ e$
$B \rightarrow .\ E\ c,\ d$
$E \rightarrow .d,\ c$

I_2 : $A \rightarrow d.,\ e$
$E \rightarrow d.,\ c$

Now can distinguish derivations by next expected input symbol.

However potential to generate more states.

LR(1) Example

$S' \to S$

$S \to a\ A\ d \mid b\ B\ d \mid a\ B\ e \mid b\ A\ e$

$A \to c$

$B \to c$

$I_0: S' \to .\ S\ ,\ \$$
$S \to .a\ A\ d,\ \$$
$S \to .b\ B\ d,\ \$$
$S \to .a\ B\ e,\ \$$
$S \to .b\ A\ e,\ \$$

a

$I_1: S \to a\ .\ B\ e, \$$
$S \to a.\ A\ d,\ \$$
$B \to .c,\ e$
$A \to .c,\ d$

b

$I_2: S \to b.\ B\ d,\ \$$
$S \to b.\ A\ e,\ \$$
$B \to .c,\ d$
$A \to .c,\ e$

c

$I_3: A \to c.,\ d$
$B \to c.,\ e$

c

$I_4: A \to c.,\ e$
$B \to c.,\ d$

Fill in the 8 missing states.

I_3 is for *ace* and *acd*;
I_4 is for *bcd* and *bce*