Bottom-up parsing

Goal

Given a grammar G, construct a parse tree for string w by starting at the leaves and working to the root

Strategy

- construct a rightmost derivation, in reverse:

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$$

- For each *right-sentential form*, $\gamma_n \ldots \gamma_1$:
  - pick a production $A \rightarrow \alpha$
  - replace α with A

Table-driven, bottom-up parsing techniques

- general strategy: shift-reduce parsing (AS&U, §4.5)
- operator precedence parsers (*we will not cover these*) (AS&U, §4.6)
- LR parsers (AS&U, §4.7)

Finding reductions: An example

Consider the grammar:

1	⟨goal⟩	::=	a ⟨A⟩ ⟨B⟩ e
2	⟨A⟩	::=	⟨A⟩ b c
3		|	b
4	⟨B⟩	::=	d

Construct a rightmost derivation for input string abbcde:
⟨goal⟩ ⇒ a⟨A⟩⟨B⟩e ⇒ a⟨A⟩de ⇒ a⟨A⟩bcde ⇒ abbcde

	Next Reduction	
Sentential Form	*Production*	*Position*
abbcde	3	2
a⟨A⟩bcde	2	4
a⟨A⟩de	4	3
a⟨A⟩⟨B⟩e	1	4
⟨goal⟩	—	—

- Each pair ⟨production, position⟩ is called a **handle**
- The trick is scanning the input to find handles efficiently

Handle

Definition

A **handle** of a right-sentential form γ is a pair $\langle \alpha \rightarrow \beta, k \rangle$ where:

- $\alpha \rightarrow \beta \in P$
- k is the position in γ of β's rightmost symbol
- replacing β with α at position k produces the right-sentential form that preceded γ in the rightmost derivation

Properties

- Because γ is a right-sentential form, the substring to the right of a handle contains only terminal symbols.
  ⇒ we don't need to scan past the handle (*very far*)
- If G is unambiguous, then every right-sentential form has a unique handle.

Uniqueness of handles

Theorem

If G is unambiguous, then every right-sentential form has a unique handle.

Sketch of proof

Proof just follows from definitions:

G is unambiguous

⇒ rightmost derivation is unique.

⇒ a unique production $\alpha \rightarrow \beta$ applied to take γ_{i-1} to γ_i, *and*
a unique position k at which $\alpha \rightarrow \beta$ is applied

⇒ a unique handle $\langle \alpha \rightarrow \beta, k \rangle$

A Running Example Grammar

Grammar

This is a left-recursive expression grammar:

1	*goal*	→	*expr*
2	*expr*	→	*expr* + *term*
3		|	*expr* - *term*
4		|	*term*
5	*term*	→	*term* * *factor*
6		|	*term* / *factor*
7		|	*factor*
8	*factor*	→	`num`
9		|	`id`

An Example Parse

Example Expression

`x - 2 * y` `(id,x) - (num,2) * (id,y)`

Parsing Steps

Prod'n.	*Sentential Form*	*Handle*
—	*goal*	—
	expr	,
	expr - *term*	,
	expr - *term* * *factor*	
	expr - *term* * `(id,y)`	
	expr - *factor* * `(id,y)`	
8	*expr* - `(num,2) * (id,y)`	*8,3*
4	*term* - `(num,2) * (id,y)`	*4,1*
7	*factor* - `(num,2) * (id,y)`	*7,1*
9	`(id,x) - (num,2) * (id,y)`	*9,1*

Handle pruning

Handle Pruning

The process of finding a handle and reducing it to the appropriate left-hand side.

Informal overview

To construct a rightmost derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w,$$

apply the following algorithm:

do i = n to 1 by -1
  1) find the handle $(\alpha_i \to \beta_i, k_i)$ in γ_i
  2) replace β_i with α_i to generate γ_{i-1}

Key Challenge

Key is to find a handle efficiently. This has two parts:

- Find substring to be reduced: β_i
- Decide which production to use: $\alpha_i \to \beta_i$

Shift-Reduce Parsing

One implementation of a handle-pruning, bottom-up parser is the *shift-reduce* parser.
Shift-reduce parsers require a *stack* and an *input buffer*

The algorithm

```
push '$' onto the stack
token ← next_token()
repeat until (top of stack = goal & token = eof)
   if we have a handle α → β on top of the stack
     then reduce β to α
       pop |β| symbols off the stack
       push α onto the stack
     else shift
       shift token onto the stack
       token ← next_token()
```

The parser must also recognize syntax errors.

Back to “x-2*y”

Stack	Input	Handle	Action
$	`id - num * id`	*none*	shift
$ `id`	`- num * id`	9,1	reduce 9
$ *factor*	`- num * id`	7,1	reduce 7
$ *term*	`- num * id`	4,1	reduce 4
$ *expr*	`- num * id`	*none*	shift
$ *expr* `-`	`num * id`	*none*	shift
$ *expr* `- num`	`* id`	8,3	reduce 8
$ *expr* `-` *factor*	`* id`	7,3	reduce 7
$ *expr* `-` *term*	`* id`	*none*	shift
$ *expr* `-` *term* `*`	`id`	*none*	shift
$ *expr* `-` *term* `* id`		9,5	reduce 9
$ *expr* `-` *term* `*` *factor*		5,5	reduce 5
$ *expr* `-` *term*		3,3	reduce 3
$ *expr*		1,1	reduce 1
$ *goal*		*none*	accept

- *Shift until top of stack is the right end of a handle*
- *Find the left end of the handle and reduce*

5 shifts + 9 reduces + 1 accept

Why is a stack sufficient?

Claim:

Handle will always appear at the top of the stack.

Why?

Because we construct a rightmost derivation (in reverse).

Sketch of proof:

- *Base case: first handle to be reduced*:
  `shift` tokens until handle appears at top of stack; reduce
- *Inductive step*: *Assume that handle for k^{th} reduction is at top of stack.*
  - ⇒ After `reduce`, new non-terminal (say A) is on top of stack
  - ⇒ “Rightmost” derivation ⇒ next handle cannot end to the left of A (i.e. below top of stack)
  - ⇒ Shift zero or more input symbols to obtain next handle at top-of-stack

See AS&U, § 4.5 for more formal version of this argument

Actions of Shift-Reduce Parsing

- *Shift-reduce parsers are easily built and easily understood*
- *We make it a little more complicated to handle errors*

4 Actions of a S-R Parser

1. **shift** — next input symbol is shifted onto the top of the stack
2. **reduce** — right end of handle is on top of stack;
   locate left end of handle within the stack;
   pop handle off stack and push appropriate non-terminal *lhs*
3. **accept** — terminate parsing and signal success
4. **error** — call an error recovery routine

Cost

Actions 3 & 4 are simple
Action 1 is a push and a call to the scanner
Action 2 takes | *rhs* | pops and 1 push

What can go wrong?

Conflicts

Failure during parser construction. 2 possible reasons:

1.
2.

Shift/Reduce Conflicts

- Usually due to ambiguous grammar
- *Option 1*: modify the grammar to eliminate the conflict
- *Option 2*: resolve in favor of shifting
- *classic examples:* “dangling else” ambiguity, insufficient associativity or precedence rules

Conflicts (continued)

Reduce/Reduce Conflicts

- Often, no simple resolution
- Option 1: try to redesign grammar, perhaps with changes to language
- Option 2: use context information during parse (e.g., symbol table)
- *Classic real example:* **PL/1 call and subscript:** `id(id, id)`

When *Stack* = . . . `id ( id ,`   *input* = `id )` . . .

- Reduce by $expr \rightarrow$ `id`, *or*
- Reduce by $param \rightarrow$ `id`

Shift/reduce conflict

Example

The dangling-else ambiguity:

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S \\ & | & \texttt{if}\ E\ \texttt{then}\ S \\ & | & \texttt{other} \end{array}$$

Abbreviate as:

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & iSeS \\ & | & iS \\ & | & a \end{array}$$

The conflict

Consider the input: $i\ i\ a\ e\ a$.
After shifting i, i, a and reducing by $S \rightarrow a$, we get:

stack = $[\$iiS]$, next token = e.

Q. On token e, what action should we take?

- Shift e : `if (E)` $\{$ `if (E)` a `else` a $\}$
- Reduce by $S \rightarrow iS$: `if (E)` $\{$ `if (E)` a $\}$ `else` a

Shift/reduce conflict

Solution for the Example

Assume: Prefer to associate `else` with innermost `if`
$\Rightarrow$ *disambiguating rule*: prefer shift over reduce
$\Rightarrow$ `if (E)` $\{$ `if (E)` a `else` a $\}$

The role of precedence and associativity

Conflict-resolution rules

- Precedence and associativity rules can be used to resolve shift/reduce conflicts in ambiguous grammars:
  - lookahead with higher precedence $\Rightarrow$ *shift*
  - same precedence, left associative $\Rightarrow$ *reduce*
- alternative to encoding them in the grammar

Advantages

- more concise, albeit ambiguous, grammars
- shallower parse trees $\Rightarrow$ fewer reductions

$\Longrightarrow$ **A simpler expression grammar**

<expr>	::=	<expr> * <expr>
	|	<expr> / <expr>
	|	<expr> + <expr>
	|	<expr> - <expr>
	|	(<expr>)
	|	- <expr>
	|	`id`
	|	`num`

LR(1) grammars

Informal definition

A grammar G is LR(1) if, given a rightmost derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_n = w\ ,$$

we can, for each right-sentential form in the derivation,

- *isolate the handle of each right-sentential form,* and
- *determine the production by which to reduce*

by scanning γ_i from left to right, going at most 1 symbol beyond the right end of the handle of γ_i.

Complexity

- one reduction per step in derivation
- one handle discovery per reduction

Key goal: Recognizing handles efficiently.

Why study LR(1) grammars?

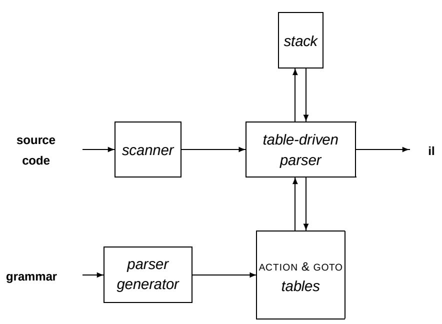
LR(1) grammars are widely used to construct parsers
These parsers are flexible & efficient

- Tools to build LR(1) parsers are widely available
- Virtually all context-free programming language constructs can be expressed in an LR(1) form
- LR grammars are the most general grammars parsable by a non-backtracking, shift-reduce parser — deterministic CFGs
- Efficient parsers can be implemented for LR(1) grammars — time proportional to *tokens + reductions*
- LR parsers detect an error as soon as possible in left-to-right scan of input
- LR grammars describe a proper superset of the languages recognized by predictive parsers

LR(1) *is a beautiful example of applying sophisticated theory to develop easy-to-use tools for a complex problem*

Table-driven *LR*(1) parsing

A table-driven LR(1) parser looks like:



The *LR* parser stack

Differences from Shift-Reduce stack

Stack two items per symbol: **symbol** and **state**. If shift-reduce stack contains:

$$X_1\ X_2\ \ldots\ X_{n-1}\ X_n$$

then LR parser stack contains:

$$X_1\ S_1\ X_2\ S_2\ \ldots\ X_{n-1}\ S_{n-1}\ X_n\ S_n$$

Stack operations

Let: *Stack* = $X_1\ S_1\ X_2\ S_2\ \ldots\ X_{n-1}\ S_{n-1}\ X_n\ S_n$,
Input = $a_1\ a_2\ \ldots\ a_k\ \ldots$

Shift S_{new} : The stack becomes

$$X_1\ S_1\ X_2\ S_2\ \ldots\ X_{n-1}\ S_{n-1}\ X_n\ S_n\ a_1\ S_{new}$$

Reduce $A \rightarrow \beta$: Let $r = \mid \beta \mid$. The stack becomes

$$X_1\ S_1\ X_2\ S_2\ \ldots\ X_{n-r}\ S_{n-r}\ A\ S_{new}$$

where S_{new} = GOTO$[S_{n-r}, A]$.

A fundamental theorem of LR parsing

- **Theorem:** If a handle can be recognized by reading the symbols on stack, then a *finite-state machine* is sufficient to do so!
- Why?
  - → each handle contains the *rhs* of some production
  - → set of handles is finite
  - → handle position is made stack-relative
- State S_i on LR parser stack is the state the FSM would be in if it read symbols $X_0 \ldots X_i$.
- GOTO$[S_i, X_i]$ is the state transition function for the FSM

Example tables

The Grammar

1	goal	→	expr
2	expr	→	term - expr
3		|	term
4	term	→	factor * term
5		|	factor
6	factor	→	id

Note: This is a simple little right-recursive grammar. It is not the same grammar as in previous lectures.

	ACTION				*GOTO*		
	id	-	*	eof	*expr*	*term*	*factor*
S_0	s4	—	—	—	1	2	3
S_1	—	—	—	acc	—	—	—
S_2	—	s5	—	r3	—	—	—
S_3	—	r5	s6	r5	—	—	—
S_4	—	r6	r6	r6	—	—	—
S_5	s4	—	—	—	7	2	3
S_6	s4	—	—	—	—	8	3
S_7	—	—	—	r2	—	—	—
S_8	—	r4	—	r4	—	—	—

Note: $S_{i+1} = $ GOTO$[S_i, X_i]$ is specified in:

- ACTION table if X_i is a token
- GOTO table if X_i is a non-terminal

LR(1) parsing: A skeleton LR(1) parsing algorithm

```
push '$'
push s0
token ← next_token()
repeat forever
   s ← top of stack
   if ACTION[s,token] = "reduce A → β" then
      pop 2 ∗ | β | symbols
      s ← top of stack     /* not a pop() */
      push A
      push GOTO[s,A]
   else if ACTION[s,token] = "shift si" then
      push token
      push si
      token ← next_token()
   else if ACTION[s, token] = "accept" and
         token = eof then
      report success
   else  report a syntax error
```


3 Common LR(1) Parsing Algorithms

LR(1) or "Canonical LR(1)"

- can recognize full set of LR(1) grammars
- largest tables *e.g., several thousand states for Pascal*
- slow, large construction

SLR(1) or "Simple LR(1)"

- can recognize smallest class of grammars
- smallest tables *e.g., several hundred states for Pascal*
- simple, fast construction

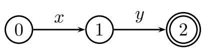
LALR(1) or "LookAhead LR(1)"

- can recognize intermediate class of grammars
- same number of states as SLR(1)
- efficient construction techniques exist, but are complex

The Goal

- We want to use a *state machine* to handle the LR parse for us.

Consider the simple grammar $A \to x\ y$. The resulting state machine is:



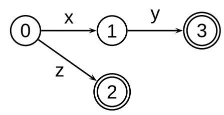
These states correspond to different stages of the production.

0. $A \to \bullet\ x\ y$
1. $A \to x\ \bullet\ y$
2. $A \to x\ y\ \bullet$

- The "$\bullet$" is called "dot" or "the cursor".
- Each entry $A \to \alpha$ with a $\bullet$ somewhere in α is called an LR(0) item.
- A state is a set of LR(0) items.

Multiple Productions

Consider the grammar $\begin{array}{l} A \to x\ y \\ A \to z \end{array}$

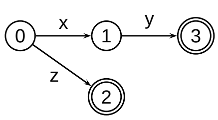


0. $A \to \bullet\ x\ y$
   $A \to \bullet\ z$

To start, place the cursor at the beginning of the A productions. This represents the beginning when we've received no input. You need to include both productions here, since we don't know which of the two productions we will use.

Multiple Productions

Consider the grammar $\begin{array}{l} A \to x\ y \\ A \to z \end{array}$

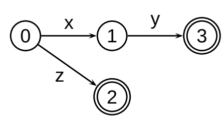


0. $A \to \bullet\ x\ y$ $\Leftarrow$
   $A \to \bullet\ z$
1. $A \to x\ \bullet\ y$

If you are in state 0 and input an x, you will advance the cursor past the x to get state 1.

Multiple Productions

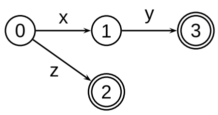
Consider the grammar $\begin{array}{l} A \to x\ y \\ A \to z \end{array}$



0. $A \to \bullet\ x\ y$
   $A \to \bullet\ z$ $\Leftarrow$
1. $A \to x\ \bullet\ y$
2. $A \to z\ \bullet$

If you are in state 0 and input a z, you will advance the cursor past the z to get state 2.

Multiple Productions

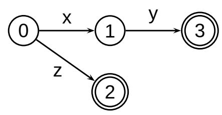


Consider the grammar $\begin{array}{l} A \rightarrow x\ y \\ A \rightarrow z \end{array}$

0. $A \rightarrow \bullet\ x\ y$
   $A \rightarrow \bullet\ z$
1. $A \rightarrow x\ \bullet\ y$ $\Leftarrow$
2. $A \rightarrow z\ \bullet$
3. $A \rightarrow x\ y\ \bullet$

If you are in state 1 and input a y, you will advance the cursor past the y to get state 3.

Multiple Productions

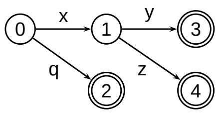


Consider the grammar $\begin{array}{l} A \rightarrow x\ y \\ A \rightarrow z \end{array}$

0. $A \rightarrow \bullet\ x\ y$
   $A \rightarrow \bullet\ z$
1. $A \rightarrow x\ \bullet\ y$
2. $A \rightarrow z\ \bullet$ $\Leftarrow$
3. $A \rightarrow x\ y\ \bullet$ $\Leftarrow$

These last two states are already complete, so no new states are formed.

Simultaneous Movement

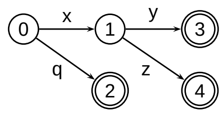


Consider the grammar $\begin{array}{l} A \rightarrow x\ y \\ A \rightarrow q \\ A \rightarrow x\ z \end{array}$

0. $A \rightarrow \bullet\ x\ y$
   $A \rightarrow \bullet\ q$
   $A \rightarrow \bullet\ x\ z$

To start, copy all the A productions and place the cursor in front.

Simultaneous Movement

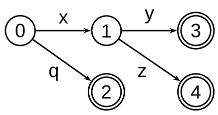


Consider the grammar $\begin{array}{l} A \rightarrow x\ y \\ A \rightarrow q \\ A \rightarrow x\ z \end{array}$

0. $A \rightarrow \bullet\ x\ y$ $\Leftarrow$
   $A \rightarrow \bullet\ q$
   $A \rightarrow \bullet\ x\ z$ $\Leftarrow$
1. $A \rightarrow x\ \bullet\ y$
   $A \rightarrow x\ \bullet\ z$

The transition from state 0 to 1 causes two productions to move: when we read x we could be parsing either xy or xz.

Simultaneous Movement



Consider the grammar
$A \to x\ y$
$A \to q$
$A \to x\ z$

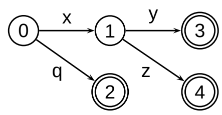
0. $A \to \bullet\ x\ y$
   $A \to \bullet\ q \Leftarrow$
   $A \to \bullet\ x\ z$

1. $A \to x\ \bullet\ y$
   $A \to x\ \bullet\ z$

2. $A \to q\ \bullet$

If we are in state 0, reading a q brings us to state 2.

Simultaneous Movement



Consider the grammar
$A \to x\ y$
$A \to q$
$A \to x\ z$

0. $A \to \bullet\ x\ y$
   $A \to \bullet\ q$
   $A \to \bullet\ x\ z$

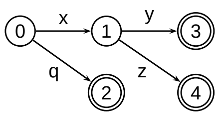
1. $A \to x\ \bullet\ y \Leftarrow$
   $A \to x\ \bullet\ z$

2. $A \to q\ \bullet$

3. $A \to x\ y\ \bullet$

If we are in state 1, reading a y brings us to state 3.

Simultaneous Movement



Consider the grammar
$A \to x\ y$
$A \to q$
$A \to x\ z$

0. $A \to \bullet\ x\ y$
   $A \to \bullet\ q$
   $A \to \bullet\ x\ z$

1. $A \to x\ \bullet\ y$
   $A \to x\ \bullet\ z \Leftarrow$

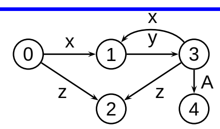
2. $A \to q\ \bullet$

3. $A \to x\ y\ \bullet$

4. $A \to x\ z\ \bullet$

If we are in state 1, reading a z brings us to state 4. None of the remaining states are expecting input.

Being Several Places at Once

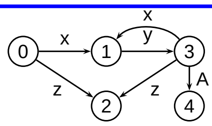


Consider the grammar
$A \to x\ y\ A$
$A \to z$

0. $A \to \bullet\ x\ y\ A$
   $A \to \bullet z$

Normal Start Sequence: Add the initial productions.

Being Several Places at Once



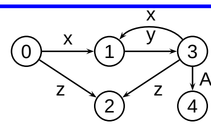
Consider the grammar $\begin{array}{l} A \to x\ y\ A \\ A \to z \end{array}$

0. $A \to \bullet\ x\ y\ A \Leftarrow$
   $A \to \bullet z$
1. $A \to x\ \bullet\ y\ A$

State 0: Shift the x to make state 1.

CS 426 Lecture 1: Course Overview

Being Several Places at Once



Consider the grammar $\begin{array}{l} A \to x\ y\ A \\ A \to z \end{array}$

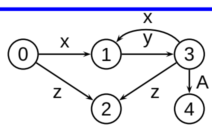
0. $A \to \bullet\ x\ y\ A$
   $A \to \bullet z \Leftarrow$
1. $A \to x\ \bullet\ y\ A$
2. $A \to z\ \bullet$

State 0: Shift the z to make state 2.

University of Illinois at Urbana-Champaign
Lecture 1: Course Overview – p.28/47

CS 426 Lecture 1: Course Overview

Being Several Places at Once



Consider the grammar $\begin{array}{l} A \to x\ y\ A \\ A \to z \end{array}$

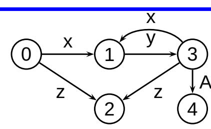
0. $A \to \bullet\ x\ y\ A$
   $A \to \bullet z$
1. $A \to x\ \bullet\ y\ A \Leftarrow$
2. $A \to z\ \bullet$
3. $A \to x\ y\ \bullet\ A$

State 1: Shift the y to make state 3. Note: the cursor is in front of an A now.

University of Illinois at Urbana-Champaign
Lecture 1: Course Overview – p.28/47

CS 426 Lecture 1: Course Overview

Being Several Places at Once



Consider the grammar $\begin{array}{l} A \to x\ y\ A \\ A \to z \end{array}$

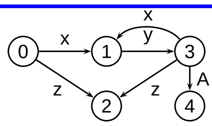
0. $A \to \bullet\ x\ y\ A$
   $A \to \bullet z$
1. $A \to x\ \bullet\ y\ A$
2. $A \to z\ \bullet$
3. $A \to x\ y\ \bullet\ A$
   $A \to \bullet\ x\ y\ A$
   $A \to \bullet\ z$

Because the cursor is in front of an A in state 3, we have to add the initial items for A again. This operation is known as taking the closure of the state.

University of Illinois at Urbana-Champaign
Lecture 1: Course Overview – p.28/47

Being Several Places at Once

Consider the grammar $\begin{array}{l} A \to x\ y\ A \\ A \to z \end{array}$



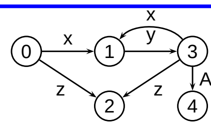
0. $A \to \bullet\, x\ y\ A$
   $A \to \bullet z$
1. $A \to x\ \bullet\ y\ A$
2. $A \to z\ \bullet \Leftarrow$
3. $A \to x\ y\ \bullet\ A$
   $A \to \bullet\, x\ y\ A$
   $A \to \bullet\ z$

State 2: No input expected.

CS 426 Lecture 1: Course Overview

Being Several Places at Once

Consider the grammar $\begin{array}{l} A \to x\ y\ A \\ A \to z \end{array}$



0. $A \to \bullet\, x\ y\ A$
   $A \to \bullet z$
1. $A \to x\ \bullet\ y\ A$
2. $A \to z\, \bullet$
3. $A \to x\ y\ \bullet\ A \Leftarrow$
   $A \to \bullet\, x\ y\ A$
   $A \to \bullet\ z$
4. $A \to x\ y\ A\, \bullet$

State 3: Shift the A to make state 4.

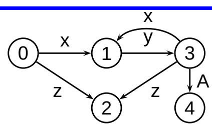
University of Illinois at Urbana-Champaign

Lecture 1: Course Overview – p.28/47

CS 426 Lecture 1: Course Overview

Being Several Places at Once

Consider the grammar $\begin{array}{l} A \to x\ y\ A \\ A \to z \end{array}$



0. $A \to \bullet\, x\ y\ A$
   $A \to \bullet z$
1. $A \to x\ \bullet\ y\ A$
2. $A \to z\, \bullet$
3. $A \to x\ y\ \bullet\ A$
   $A \to \bullet\, x\ y\ A \Leftarrow$
   $A \to \bullet\ z$
4. $A \to x\ y\ A\, \bullet$

State 3: Shifting the x will create a state just like state 1. So we recycle it. Note the “back arrow” in the state diagram.

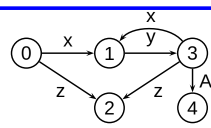
University of Illinois at Urbana-Champaign

Lecture 1: Course Overview – p.28/47

CS 426 Lecture 1: Course Overview

Being Several Places at Once

Consider the grammar $\begin{array}{l} A \to x\ y\ A \\ A \to z \end{array}$



0. $A \to \bullet\, x\ y\ A$
   $A \to \bullet z$
1. $A \to x\ \bullet\ y\ A$
2. $A \to z\, \bullet$
3. $A \to x\ y\ \bullet\ A$
   $A \to \bullet\, x\ y\ A$
   $A \to \bullet\ z \Leftarrow$
4. $A \to x\ y\ A\, \bullet$

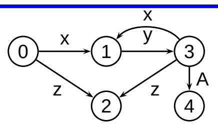
State 3: Same situation with shifting z, only we recycle state 2.

University of Illinois at Urbana-Champaign

Lecture 1: Course Overview – p.28/47

Being Several Places at Once

Consider the grammar $A \to x\ y\ A$
$A \to z$

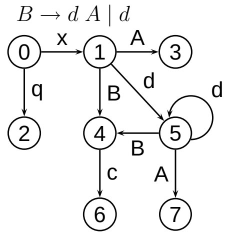


0. $A \to \bullet\ x\ y\ A$
   $A \to \bullet z$
1. $A \to x\ \bullet\ y\ A$
2. $A \to z\ \bullet$
3. $A \to x\ y\ \bullet\ A$
   $A \to \bullet\ x\ y\ A$
   $A \to \bullet\ z$
4. $A \to x\ y\ A\ \bullet \Leftarrow$

State 4: No input expected. The automaton is complete.

Transitive Closures

$S \to x\ A \mid q$
$A \to B\ c$
$B \to d\ A \mid d$

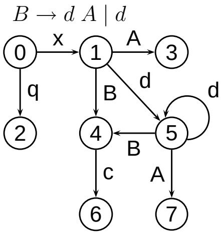


0. $S \to \bullet\ x\ A$
   $S \to \bullet\ q$

We have multiple productions this time. We only use the “start” symbol S.

Transitive Closures

$S \to x\ A \mid q$
$A \to B\ c$
$B \to d\ A \mid d$

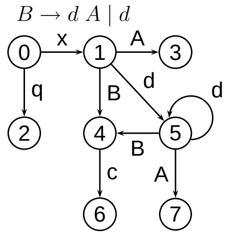


0. $S \to \bullet\ x\ A \Leftarrow$
   $S \to \bullet\ q$
1. $S \to x\ \bullet\ A$
   $A \to \bullet\ B\ c$
   $B \to \bullet\ d\ A$
   $B \to \bullet\ d$

State 0: shift x and take transitive closure to make state 1.

Transitive Closures

$S \to x\ A \mid q$
$A \to B\ c$
$B \to d\ A \mid d$



0. $S \to \bullet\ x\ A$
   $S \to \bullet\ q \Leftarrow$
1. $S \to x\ \bullet\ A$
   $A \to \bullet\ B\ c$
   $B \to \bullet\ d\ A$
   $B \to \bullet\ d$
2. $S \to q\ \bullet$

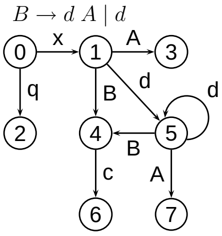
State 0: shift q to make state 2.

Transitive Closures

$S \to x\ A \mid q$

$A \to B\ c$

$B \to d\ A \mid d$



0. $S \to \bullet\ x\ A$
   $S \to \bullet\ q$
1. $S \to x\ \bullet\ A \Leftarrow$
   $A \to \bullet\ B\ c$
   $B \to \bullet\ d\ A$
   $B \to \bullet\ d$
2. $S \to q\ \bullet$
3. $S \to x\ A\ \bullet$

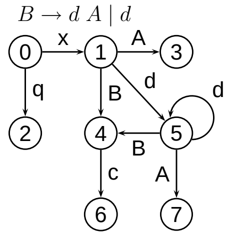
State 1: shift A (actually, match A) to make state 3.

Transitive Closures

$S \to x\ A \mid q$

$A \to B\ c$

$B \to d\ A \mid d$



0. $S \to \bullet\ x\ A$
   $S \to \bullet\ q$
1. $S \to x\ \bullet\ A$
   $A \to \bullet\ B\ c \Leftarrow$
   $B \to \bullet\ d\ A$
   $B \to \bullet\ d$
2. $S \to q\ \bullet$
3. $S \to x\ A\ \bullet$
4. $A \to B\ \bullet\ c$

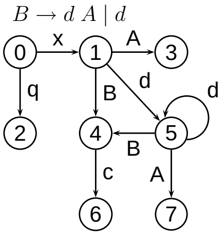
State 1: shift B (actually, match B) to make state 4.

Transitive Closures

$S \to x\ A \mid q$

$A \to B\ c$

$B \to d\ A \mid d$



0. $S \to \bullet\ x\ A$
   $S \to \bullet\ q$
1. $S \to x\ \bullet\ A$
   $A \to \bullet\ B\ c$
   $B \to \bullet\ d\ A \Leftarrow$
   $B \to \bullet\ d \Leftarrow$
2. $S \to q\ \bullet$
3. $S \to x\ A\ \bullet$
4. $A \to B\ \bullet\ c$
5. $B \to d\ \bullet\ A$
   $B \to d\ \bullet$
   $A \to \bullet\ B\ c$
   $B \to \bullet\ d\ A$
   $B \to \bullet\ d$

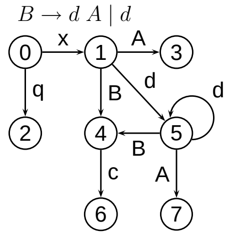
State 1: shift d to make state 5. A lot happens here!

Transitive Closures

$S \to x\ A \mid q$

$A \to B\ c$

$B \to d\ A \mid d$



0. $S \to \bullet\ x\ A$
   $S \to \bullet\ q$
1. $S \to x\ \bullet\ A$
   $A \to \bullet\ B\ c$
   $B \to \bullet\ d\ A$
   $B \to \bullet\ d$
2. $S \to q\ \bullet$
3. $S \to x\ A\ \bullet$
4. $A \to B\ \bullet\ c \Leftarrow$
5. $B \to d\ \bullet\ A$
   $B \to d\ \bullet$
   $A \to \bullet\ B\ c$
   $B \to \bullet\ d\ A$
   $B \to \bullet\ d$
6. $A \to B\ c\ \bullet$

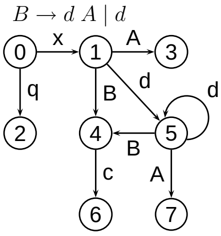
State 4: shift c to make state 6.

Transitive Closures

$S \rightarrow x\ A \mid q$

$A \rightarrow B\ c$

$B \rightarrow d\ A \mid d$



0. $S \rightarrow \bullet\ x\ A$
   $S \rightarrow \bullet\ q$
1. $S \rightarrow x\ \bullet\ A$
   $A \rightarrow \bullet\ B\ c$
   $B \rightarrow \bullet\ d\ A$
   $B \rightarrow \bullet\ d$
2. $S \rightarrow q\ \bullet$
3. $S \rightarrow x\ A\ \bullet$
4. $A \rightarrow B\ \bullet\ c$
5. $B \rightarrow d\ \bullet\ A \Leftarrow$
   $B \rightarrow d\ \bullet$
   $A \rightarrow \bullet\ B\ c$
   $B \rightarrow \bullet\ d\ A$
   $B \rightarrow \bullet\ d$
6. $A \rightarrow B\ c\ \bullet$
7. $B \rightarrow d\ A\ \bullet$

State 5: shift A to make state 7. The other shifts recycle. We are done.

LR(k) items: Definitions

Definition

An LR(k) *item* is a pair $[A, B]$, where
A is a production $\alpha \rightarrow \beta\gamma\delta$ with $\bullet$ at some position in the *rhs*
B is a lookahead string of length $\leq k$ (tokens or `eof`)

Finiteness

P is finite, T is finite
$\Rightarrow$ there are only $\mid T \mid \cdot (\max_{p \in P} \mid \text{rhs}(p) \mid +1)$ possible LR(1) items

Parser States

A parser state is a set of LR(K) items. These items describe valid productions we might use next or the tokens we might shift, given current contents of the stack.

LR(0) items are used in the SLR(1) table construction algorithm

LR(1) items are used in the LR(1) and LALR(1) algorithms

Example

LR(1) ***Items***

The production $\alpha \rightarrow \beta\gamma\delta$, with lookahead a generates 4 LR(1) items

1. $[\alpha \rightarrow \bullet\beta\gamma\delta, \text{a}]$
2. $[\alpha \rightarrow \beta \bullet \gamma\delta, \text{a}]$
3. $[\alpha \rightarrow \beta\gamma \bullet \delta, \text{a}]$
4. $[\alpha \rightarrow \beta\gamma\delta\bullet, \text{a}]$

The $\bullet$ indicates the position of the top of stack

$[\alpha \rightarrow \bullet\beta\gamma\delta, \mathbf{a}]$ means that the input seen so far is consistent with the use of $\alpha \rightarrow \beta\gamma\delta$ at this point in the parse

$[\alpha \rightarrow \beta\gamma \bullet \delta, \mathbf{a}]$ means that the input seen so far is consistent with the use of $\alpha \rightarrow \beta\gamma\delta$, and the parser has already recognized $\beta\gamma$

$[\alpha \rightarrow \beta\gamma\delta\bullet, \mathbf{a}]$ means that the parser has seen $\beta\gamma\delta$, and if next input token matches lookahead symbol a, then parser can reduce to α

Lookahead component of LR(1) state

What does the lookahead component of state mean?

- lookahead string is used to choose action when item has $\bullet$ at right end
- Let stack = $\delta\gamma$ and let next token be $a \neq EOF$
  - $\Rightarrow$ $A \rightarrow \gamma$ is a handle only if there is a right-sentential form containing $\delta A a \ldots$
  - $\Rightarrow$ State item $[A \rightarrow \gamma\bullet, \text{a}]$ indicates that a is acceptable when stack contains $\delta\gamma$

How is the lookahead component used?

1. For $[\alpha \rightarrow \gamma\bullet, \text{a}]$ and $[\beta \rightarrow \gamma\bullet, \text{b}]$,
   - on a reduce to α
   - on b reduce to β
2. For $[\alpha \rightarrow \gamma\bullet, \text{a}]$ and $[\beta \rightarrow \gamma \bullet \delta, \text{b}]$,
   - on a, reduce to α
   - else, for any $b \in \text{FIRST}(\delta)$, shift

$\Rightarrow$ *Next symbol from input is enough to pick actions more precisely*

FIRST Sets for a Grammar

Definition
For a string of grammar symbols α, define FIRST(α) as

- the set of terminal symbols that begin strings derived from α
- If $\alpha \Rightarrow^* \epsilon$, then $\epsilon \in$ FIRST(α)

FIRST(α) contains the set of tokens valid in the first position of α

Algorithm
To build FIRST(X):

1. if X is a terminal, FIRST(X) is $\{X\}$
2. if $X \rightarrow \epsilon$, then $\epsilon \in$ FIRST(X)
3. if $X \rightarrow Y_1 Y_2 \cdots Y_k$ then put FIRST(Y_1) in FIRST(X)
4. if X is a non-terminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$, then $a \in$ FIRST(X) if $a \in$ FIRST(Y_i) and $\epsilon \in$ FIRST(Y_j) for all $1 \leq j < i$
(If $\epsilon \notin$ FIRST(Y_1), then FIRST(Y_i) is irrelevant, for $1 < i$)

Example: Grammar & FIRST sets

Grammar

1. *goal* $\rightarrow$ *expr*
2. *expr* $\rightarrow$ *term* $-$ *expr*
3. *expr* $\rightarrow$ *term*
4. *term* $\rightarrow$ *factor* $*$ *term*
5. *term* $\rightarrow$ *factor*
6. *factor* $\rightarrow$ `id`

FIRST sets

Symbol	FIRST
goal	{`id`}
expr	
term	
factor	
$-$	
$*$	
`id`	

Possible State Transitions

Consider a state containing an item $[A \rightarrow \alpha \bullet X\beta,\ a]$

- Push X (token or NT) on the stack

$$[A \rightarrow \alpha \bullet X\beta,\ a] \ \stackrel{X}{\rightarrow}\ [A \rightarrow \alpha X \bullet \beta,\ a]$$

- But if X is a non-terminal, we can push X on the stack only via some production $X \rightarrow \gamma$. So we need to look for strings that can be derived from γ

$$[A \rightarrow \alpha \bullet X\beta,\ a] \ \stackrel{\epsilon}{\rightarrow}\ [X \rightarrow \bullet\gamma,\ b],\ \ \forall\, b \in \textsc{First}(\beta \text{a}).$$

  This says: X generates γ and then βa generates a string starting with b.

- Group above items into a single state, i.e., if a state contains item $[A \rightarrow \alpha \bullet X\beta,\ a]$, add items $[X \rightarrow \bullet\gamma,\ b]$ for all X productions, and $\forall\, b\ \in\ \textsc{First}(\beta \text{a})$

Computing the Closure

Algorithm to find "equivalent" item for a given set of items

The Closure Algorithm

```
Closure(s_i:  set of items)
  do
    changing ← false
    ∀ item [A → α • Xβ, a]  ∈  I               O(| s_i |)
      ∀ production  X → γ  ∈  P                O( alternatives for X )
        ∀  b  ∈  FIRST(βa)                     O(| FIRST(δa) |)
          if [X → •γ, b]  ∉  s_i  then
             add [X → •γ, b] to s_i
             changing ← true
  while (changing)
```

Example
Q. What is s_0 = `Closure( { [g→ •e, eof] } )` in the example grammar?

Computing the GOTO Function

Algorithm for GOTO

```
Goto(s_i, x)
  new ← ∅
  ∀ items i ∈ s_i           /* move the • */
    if i is [A→α • xβ, a] then
       new ← new ∪ [A→αx • β, a]
  new ← closure(new)    /* make it a DFSM state */
  return new
```

Complete LR(1) Table Construction Algorithm

1. Build C, the *canonical collection of sets of LR(1) items*
2. Iterate through C, filling in ACTION and GOTO tables
   (Coming up)

Example: Building the collection

Initial State

$I_0 \leftarrow$ closure({[g → • e,eof]})
= { [g→ • e,eof], [e→ • t - e,eof], [e → • t,eof],
[t→ • f * t,-], [t→ • f * t,eof], [t→ • f,-], [t→ • f,eof],
[f→ • id,-], [f→ • id,*] [f→ • id,eof] }

Iteration 1

$I_1 \leftarrow$ goto(I_0, e) = { [g→ e •,eof] }
$I_2 \leftarrow$ goto(I_0, t) = { [e → t •,eof],
[e → t • - e,eof] }
$I_3 \leftarrow$ goto(I_0, f) =
$I_4 \leftarrow$ goto(I_0, id) =

Iteration 2

$I_5 \leftarrow$ goto(I_2, -)
$I_6 \leftarrow$ goto(I_3, *)

Iteration 3

$I_7 \leftarrow$ goto(I_5, e)
$I_8 \leftarrow$ goto(I_6, t)

Example: Summary

I_0: [g → • e,eof], [e → • t - e,eof], [e → • t,eof],
[t → • f * t,{-,eof}], [t → • f,{-,eof}], [f → • id,{-,*,eof}]

I_1: [g→ e •,eof]

I_2: [e → t •,eof], [e → t • - e,eof]

I_3: [t → f •,{-,eof}], [t → f • * t, {-,eof}]

I_4: [f → id •, {-,*,eof}]

I_5: [e → t - • e,eof], [e → • t - e,eof], [e → • t,eof],
[t → • f * t,{-,eof}], [t → • f,{-,eof}],
[f → • id,{-,*,eof}]

I_6: [t → f * • t,{-,eof}], [t → • f * t,{-,eof}],
[t → • f,{-,eof}], [f → •id, {-,*,eof}]

I_7: [e → t - e •, eof]

I_8: [t → f * t •, {-,eof}]

LR(1) Table Construction

To build the table, we simply interpret the sets

1. G' = Augment grammar G by adding production $S'\to S$
   (e.g., goal→expr in our example)
2. Construct the canonical collection of sets of LR*(1)* items for G'.
3. State i of the parser is constructed from set I_i.
   (a) if $[A \to \alpha \bullet \mathrm{a}\beta, \mathrm{b}] \in I_i$ and goto$(I_i, \mathrm{a}) = I_j$, then set `action[i,a]` to "*shift j*". (a must be a terminal)
   (b) if $[A \to \alpha\bullet, \mathrm{a}] \in I_i$, then set `action[i,a]` to "*reduce* $A \to \alpha$".
   (c) if $[S' \to S\bullet, \mathtt{eof}] \in I_i$, then set `action[i,eof]` to "*accept*".
4. If goto$(I_i, A) = I_j$, then set `goto[i,A]` to j.
5. All other entries in `action` and `goto` are set to "*error*"
6. The initial state of the parser is the state constructed from the set containing the item $[S' \to \bullet S, \mathtt{eof}]$.

Example: Final Tables

Fill in rows S_3 and S_5 :

The Grammar

1.	*goal*	$\rightarrow$	*expr*
2.	*expr*	$\rightarrow$	*term* - *expr*
3.		|	*term*
4.	*term*	$\rightarrow$	*factor* * *term*
5.		|	*factor*
6.	*factor*	$\rightarrow$	`id`

	ACTION				GOTO		
	`id`	`-`	`*`	`eof`	*expr*	*term*	*factor*
S_0	s4	—	—	—	1	2	3
S_1	—	—	—	acc	—	—	—
S_2	—	s5	—	r3	—	—	—
S_3							
S_4	—	r6	r6	r6	—	—	—
S_5							
S_6	s4	—	—	—	—	8	3
S_7	—	—	—	r2	—	—	—
S_8	—	r4	—	r4	—	—	—

Conflicts During LR(1) Construction

Rules 3a, 3b, & 3c can construct two different actions for an entry in ACTION. If this happens, the grammar is not *LR*(1). Usually indicates an ambiguous construct in the grammar.

Example : dangling-else, again:

$S' \rightarrow S$

$S \rightarrow$ `if` E `then` S `else` S

$\quad\mid$ `if` E `then` S

$\quad\mid$ `other`

Abbreviate as:

$S' \rightarrow S$

$S \rightarrow iSeS$

$\quad\mid iS$

$\quad\mid a$

The conflict

State I_4 of LR(1) parser is:

$I_4 \;:\; [S{\rightarrow}iS\bullet eS,\ eof],\ [S{\rightarrow}iS\bullet eS,\ e]$
$\qquad [S{\rightarrow}iS\bullet,\ eof],\quad [S{\rightarrow}iS\bullet,\ e]$

Q. What action do we take on e?

- item $[S{\rightarrow}iS\bullet eS,\ e]$ says:
- item $[S{\rightarrow}iS\bullet,\ e]$ says:

Solution

???

LALR(1) parsing

Definition (Core) :

The *core* of a set of *LR*(1) items is the set of *LR*(0) items derived by ignoring the lookahead symbols.
Example: the two sets

- $\{[A \Rightarrow \alpha \bullet \beta, \mathrm{a}], [A \Rightarrow \alpha \bullet \beta, \mathrm{b}]\}$, and
- $\{[A \Rightarrow \alpha \bullet \beta, \mathrm{c}], [A \Rightarrow \alpha \bullet \beta, \mathrm{d}]\}$

have the same core.

Key Idea of LALR:

If two states, I_i and I_j, have the same core, we can merge those states in the `action` and `goto` tables.

Comparing LALR with LR

What new conflicts are possible?

- GOTO[I, X] depends only on core I, not on X, so just merge the GOTO functions for merged states
- `shift` action also depends only on core (e.g., $[A \rightarrow \alpha \bullet a\beta, b]$)
  `reduce` action depends on both (e.g., $[A \rightarrow \alpha\bullet, a]$),
  $\Rightarrow$ merging states as above does not introduce shift-reduce conflicts *unless there was one before*
- new reduce-reduce conflicts are possible

LALR(1) table construction

The simple algorithm

To construct LALR(1) parsing tables, we insert one step into the LR(1) table construction algorithm.

(1.5) For each core present among the set of LR(1) items, find all sets having that core and replace these sets by their union
Update the goto function to reflect the replacement sets

The resulting algorithm has large space requirements

A better algorithm

A more space efficient algorithm can be derived by observing that:

- we can represent I_i by its *kernel*, those items that are either the initial item $[S' \rightarrow \bullet S, \texttt{eof}]$ or do not have the $\bullet$ at the left end of the *rhs*.
- we can compute *shift*, *reduce*, and *goto* actions for the state derived from I_i directly from *kernel*(I_i).

This avoids building the complete canonical collection

LR(1) versus *LL*(1) grammars

Finding reductions in LR*(k) and* LL*(k)*

LR(k) $\Rightarrow$ Parser must select a reduction based on

1. everything to the left of the reducible phrase
2. everything derived from the reducible phrase itself
3. the next k terminal symbols

LL(k) $\Rightarrow$ Parser must select the reduction based on

1. everything to the left of the reducible phrase
2. the first k terminals derived from the reducible phrase

Thus, LR*(k) has more information to choose reductions*
$\Rightarrow$ LR*(k) parsers can parse more grammars than* LL*(k)*

"... in practice, programming languages do not actually seem to fall in the gap between LL*(1) languages and deterministic (aka LR) languages"*

J.J. Horning, " LR Grammars and Analysers", in *Compiler Construction, An Advanced Course*, Springer-Verlag, 1976.

The hierarchy of context-free grammars

Inclusion hierarchy for context-free grammars: