Parsing IV
LR(1) Parsers

LR(1) Parsers

- LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition
- LR(1) parsers recognize languages that have an LR(1) grammar

Informal definition:

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

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow sentence$$

We can

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

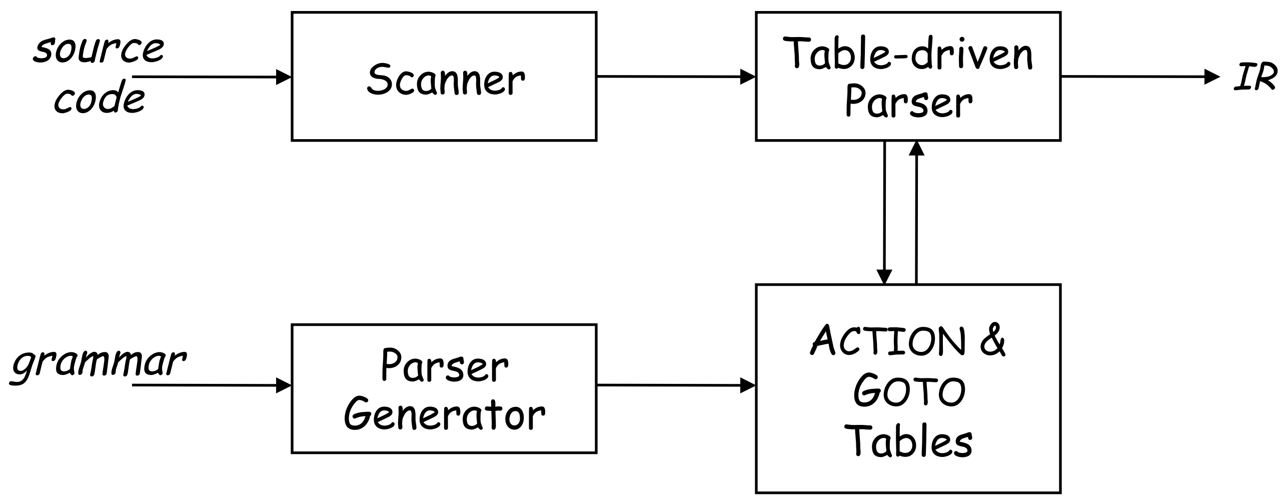
by scanning γ_i from left-to-right, going at most 1 symbol beyond the right end of the handle of γ_i

LR(1) Parsers

A table-driven LR(1) parser looks like

Tables _can_ be built by hand

However, this is a perfect task to automate



LR(1) Skeleton Parser

```
stack.push(INVALID);
stack.push(s0);
token = scanner.next_token();
do while (TRUE) {
        s = stack.top();
        if ( ACTION[s,token] == "shift si" ) then {
                stack.push(token);
                stack.push(si);
                token ← scanner.next_token();
        }
        else if ( ACTION[s,token] == "reduce A→β" ) then {
                stack.popnum(2*|β|); // pop 2*|β| symbols
                s = stack.top();
                stack.push(A);
                stack.push(GOTO[s,A]);
        }
        else if ( ACTION[s,token] == "accept"
                                & token == EOF ) then
                return;
        else report a syntax error and recover;
}
```

The skeleton parser

- uses ACTION & GOTO tables
- does |*words*| shifts
- does |derivation| reductions
- does 1 accept
- detects errors by failure of 3 other cases

LR(1) Parsers (parse tables)

To make a parser for *L(G)*, need a set of tables

The grammar

1	*Goal*	→	SheepNoise
2	*SheepNoise*	→	baa SheepNoise
3		|	baa

The tables

ACTION		
State	EOF	baa
0	—	shift 2
1	accept	
2	reduce 3	shift 2
3	reduce 2	

GOTO	
State	*SheepNoise*
0	1
1	
2	3
3	

Example Parse 1: The string "baa"

```
stack.push(INVALID);
stack.push(s0);
token = scanner.next_token();
do while (TRUE) {
        s = stack.top();
        if ( ACTION[s,token] == "shift si" ) then {
              stack.push(token);
            stack.push(si);
              token ← scanner.next_token();
        }
      else if ( ACTION[s,token] == "reduce A→β" ) then {
              stack.popnum(2*|β|); // pop 2*|β| symbols
            s = stack.top();
            stack.push(A);
            stack.push(GOTO[s,A]);
        }
        else if ( ACTION[s,token] == "accept"
                              & token == EOF ) then
            return;
        else report a syntax error and recover;
}
```

The grammar

1	Goal	→	SheepNoise
2	SheepNoise	→	baa SheepNoise
3		|	baa

The tables

ACTION		
State	EOF	baa
0	—	shift 2
1	accept	
2	reduce 3	shift 2
3	reduce 2	

GOTO	
State	SheepNoise
0	1
1	
2	3
3	

Example Parse 1

The string "baa"

Stack	Input	Action
$ s_0	baa EOF	shift 2

1	*Goal*	→	SheepNoise
2	*SheepNoise*	→	baa SheepNoise
3		|	baa

ACTION		
State	EOF	baa
0	—	shift 2
1	accept	
2	reduce 3	shift 2
3	reduce 2	

GOTO	
State	*SheepNoise*
0	1
1	
2	3
3	

Example Parse 1

The string "baa"

Stack	Input	Action
\$ s_0	baa EOF	shift 2
\$ s_0 baa s_2	EOF	reduce 3

1	*Goal*	→	SheepNoise
2	*SheepNoise*	→	baa SheepNoise
3		|	baa

ACTION		
State	EOF	baa
0	—	shift 2
1	accept	
2	reduce 3	shift 2
3	reduce 2	

GOTO	
State	*SheepNoise*
0	1
1	
2	3
3	

Example Parse 1

The string "baa"

Stack	Input	Action
$ s_0	baa EOF	shift 2
$ s_0 baa s_2	EOF	reduce 3
$ s_0 *SN* s_1	EOF	

1	*Goal*	→	SheepNoise
2	*SheepNoise*	→	baa SheepNoise
3		|	baa

ACTION		
State	EOF	baa
0	—	shift 2
1	accept	
2	reduce 3	shift 2
3	reduce 2	

GOTO	
State	*SheepNoise*
0	1
1	
2	3
3	

Example Parse 1

The string "baa"

Stack	Input	Action
$ s_0	baa EOF	shift 2
$ s_0 baa s_2	EOF	reduce 3
$ s_0 *SN* s_1	EOF	accept

1	*Goal*	→	SheepNoise
2	*SheepNoise*	→	baa SheepNoise
3		|	baa

ACTION		
State	EOF	baa
0	—	shift 2
1	accept	
2	reduce 3	shift 2
3	reduce 2	

GOTO	
State	*SheepNoise*
0	1
1	
2	3
3	

Example Parse 2: The string "baa baa"

```
stack.push(INVALID);
stack.push(s0);
token = scanner.next_token();
do while (TRUE) {
    s = stack.top();
    if ( ACTION[s,token] == "shift si" ) then {
        stack.push(token);
        stack.push(si);
        token ← scanner.next_token();
    }
    else if ( ACTION[s,token] == "reduce A→β" ) then {
        stack.popnum(2*|β|); // pop 2*|β| symbols
        s = stack.top();
        stack.push(A);
        stack.push(GOTO[s,A]);
    }
    else if ( ACTION[s,token] == "accept"
                  & token == EOF ) then
        return;
    else report a syntax error and recover;
}
```

The grammar

1	Goal	→	SheepNoise
2	SheepNoise	→	baa SheepNoise
3		|	baa

The tables

ACTION		
State	EOF	baa
0	—	shift 2
1	accept	
2	reduce 3	shift 2
3	reduce 2	

GOTO	
State	SheepNoise
0	1
1	
2	3
3	

Example Parse 2

The string "baa baa"

Stack	Input	Action
$ s_0	baa baa EOF	shift 2
$ s_0 baa s_2	baa EOF	

1	*Goal*	→	SheepNoise
2	*SheepNoise*	→	baa SheepNoise
3		|	baa

ACTION		
State	EOF	baa
0	—	shift 2
1	accept	
2	reduce 3	shift 2
3	reduce 2	

GOTO	
State	*SheepNoise*
0	1
1	
2	3
3	

Example Parse 2

The string “baa baa”

Stack	Input	Action
$ s_0	baa baa EOF	shift 2
$ s_0 baa s_2	baa EOF	shift 2
$ s_0 baa s_2 baa s_2	EOF	

1	*Goal*	→	SheepNoise
2	*SheepNoise*	→	baa SheepNoise
3		|	baa

ACTION		
State	EOF	baa
0	—	shift 2
1	accept	
2	reduce 3	shift 2
3	reduce 2	

GOTO	
State	*SheepNoise*
0	1
1	
2	3
3	

Example Parse 2

The string "baa baa"

Stack	Input	Action
\$ s_0	baa baa EOF	shift 2
\$ s_0 baa s_2	baa EOF	shift 2
\$ s_0 baa s_2 baa s_2	EOF	reduce 3
\$ s_0 *baa* s_2 SN s_3	EOF	

1	*Goal*	→	SheepNoise
2	*SheepNoise*	→	baa SheepNoise
3		|	baa

ACTION		
State	EOF	baa
0	—	shift 2
1	accept	
2	reduce 3	shift 2
3	reduce 2	

GOTO	
State	*SheepNoise*
0	1
1	
2	3
3	

Example Parse 2

The string "baa baa"

Stack	Input	Action
$ s_0	baa baa EOF	shift 2
$ s_0 baa s_2	baa EOF	shift 2
$ s_0 *baa* s_2 *baa* s_2	EOF	reduce 3
$ s_0 baa s_2 SN s_3	EOF	reduce 2
$ s_0 *SN* s_1	EOF	accept

1	*Goal*	→	SheepNoise
2	*SheepNoise*	→	baa SheepNoise
3		|	baa

ACTION		
State	EOF	baa
0	—	shift 2
1	accept	
2	reduce 3	shift 2
3	reduce 2	

GOTO	
State	*SheepNoise*
0	1
1	
2	3
3	

LR(1) Parsers

How does this LR(1) stuff work?

- Unambiguous grammar ⇒ unique rightmost derivation
- Keep upper fringe on a stack
  - → All active handles include top of stack (TOS)
  - → Shift inputs until TOS is right end of a handle
- Language of handles is regular (finite)
  - → Build a handle-recognizing DFA
  - → ACTION & GOTO tables encode the DFA
- Final state in DFA ⇒ a *reduce* action
  - → New state is GOTO[state at TOS (after pop), *lhs*]
  - → For *SN*, this takes the DFA to s_1

Building LR(1) Parsers

How do we generate the *ACTION* and *GOTO* tables?

- Use the grammar to build a model of the DFA
- Use the model to build *ACTION* & *GOTO* tables
- If construction succeeds, the grammar is LR(1)

The Big Picture

- Model the state of the parser
- Use two functions *goto(s, X)* and *closure(s)*
  - → *goto()* is analogous to Delta*()* in the subset construction
  - → *closure()* adds information to round out a state
- Build up the states and transition functions of the DFA
- Use this information to fill in the *ACTION* and *GOTO* tables

LR(*1*) items

The LR(1) table construction algorithm uses LR(1) items to represent valid configurations of an LR(1) parser

An LR(*1*) item is a pair [*P, a*], where

P is a production $A \rightarrow \beta$ with a • at some position in the *rhs and* ***a*** is a lookahead word (or EOF)

The • in an item indicates the position of the top of the stack

[*A*→•βγ,a] means that the input seen so far is consistent with the use of $A \rightarrow \beta\gamma$ immediately after the symbol on top of the stack

[*A* →β•γ,a] means that the input seen so far is consistent with $A \rightarrow \beta\gamma$ at this point in the parse, *and* that the parser has already recognized β

[*A* →βγ•,a] means that the parser has seen βγ, *and* that a lookahead symbol of a is consistent with reducing to *A*

LR(1) Items

The production $A \rightarrow \beta$, where $\beta = B_1 B_1 B_1$ with lookahead $\underline{a}$,
can give rise to 4 items

$[A \rightarrow \cdot B_1 B_2 B_3, \underline{a}]$, $[A \rightarrow B_1 \cdot B_2 B_3, \underline{a}]$, $[A \rightarrow B_1 B_2 \cdot B_3, \underline{a}]$, & $[A \rightarrow B_1 B_2 B_3 \cdot, \underline{a}]$

The set of LR(1) items for a grammar is finite

What's the point of all these lookahead symbols?

- Carry them along to choose the correct reduction, *if there is a choice*
- Lookaheads are bookkeeping, unless item has • at right end
  - Has no direct use in $[A \rightarrow \beta \cdot \gamma, \underline{a}]$
  - In $[A \rightarrow \beta \cdot, \underline{a}]$, a lookahead of $\underline{a}$ implies a reduction by $A \rightarrow \beta$
  - For $\{ [A \rightarrow \beta \cdot, \underline{a}], [B \rightarrow \gamma \cdot \delta, \underline{b}] \}$, $\underline{a} \Rightarrow$ ***reduce*** to A; FIRST(δ) $\Rightarrow$ ***shift***

$\Rightarrow$ Limited right context is enough to pick the actions

LR(1) Table Construction

High-level overview

1 Build the canonical collection of sets of LR(1) Items
   a Begin in an appropriate state, CC_0
      - [$S' \rightarrow \cdot S$,EOF], along with any equivalent items
      - Derive equivalent items as *closure*(CC_0)
   b Repeatedly compute, for each CC_k, and each X, *goto*(CC_k,X)
      - If the set is not already in the collection, add it
      - Record all the transitions created by *goto*( )

   This eventually reaches a fixed point

2 Fill in the tables from the collection of sets of LR(1) items

The canonical collection completely encodes the transition diagram for the handle-finding ***DFA***

Computing FIRST Sets

Define FIRST as

- If $\alpha \Rightarrow^* \underline{a}\beta$, $\underline{a} \in T$, $\beta \in (T \cup NT)^*$, then $\underline{a} \in$ FIRST(α)
- If $\alpha \Rightarrow^* \varepsilon$, then $\varepsilon \in$ FIRST(α)

Note: if $\alpha = X\beta$, FIRST(α) = FIRST(X)

To compute FIRST

- Use a fixed-point method
- FIRST(A) $\in 2^{(T \cup \varepsilon)}$
- Loop is monotonic

$\Rightarrow$ Algorithm halts

Computing Closures

Closure(s) adds all the items implied by items already in *s*

- Any item $[A \rightarrow \beta \bullet B\delta, \underline{a}]$ implies $[B \rightarrow \bullet \tau, x]$ for each production with *B* on the *lhs*, and each $x \in$ FIRST($\delta\underline{a}$)
- Since $\beta B\delta$ is valid, any way to derive $\beta B\delta$ is valid, too

The algorithm

```
Closure( s )
  while ( s is still changing )
    ∀ items [A → β • Bδ,a] ∈ s
      ∀ productions B → τ ∈ P
        ∀ b ∈ FIRST(δa) // δ might be ε
          if [B→ • τ,b] ∉ s
            then add [B→ • τ,b] to s
```

- Another fixed-point algorithm
- Halts because $s \subset$ ITEMS
- *Closure "fills out" a state*

Example From SheepNoise

Initial step builds the item [*Goal*→•*SheepNoise*,EOF] and takes its *closure( )*

Closure([*Goal*→•*SheepNoise*,EOF] *)*

Item	***From***
[*Goal*→ •*SheepNoise*,EOF]	**Original item**
[*SheepNoise*→ • baa SheepNoise,EOF]	**1, δa is EOF**
[*SheepNoise*→ • baa,EOF]	**1, δa is EOF**

CC_0 is

{ [*Goal*→ • *SheepNoise*,EOF], [*SheepNoise*→ • baa SheepNoise,EOF], [*SheepNoise*→ • baa,EOF]}

Computing Gotos

Goto(s,x) computes the state that the parser would reach if it recognized an *x* while in state *s*

- *Goto(* { [$A \rightarrow \beta \bullet X\delta$,$\underline{a}$] }, *X)* produces [$A \rightarrow \beta X \bullet \delta$,$\underline{a}$]
- It also includes *closure(* [$A \rightarrow \beta X \bullet \delta$,$\underline{a}$] *)* to fill out the state

The algorithm

Goto(s, X)
  new ← ∅
  ∀ *items* [$A \rightarrow \beta \cdot X\delta$,$\underline{a}$] ∈ *s*
    new ← *new* ∪ [$A \rightarrow \beta X \cdot \delta$,$\underline{a}$]
  return closure(new)

- **Straightforward computation**
- **Uses *closure*( )**

Goto() moves forward

Example from SheepNoise

CC_0 is { [*Goal*→ • *SheepNoise*,EOF], [*SheepNoise*→ •baa SheepNoise,EOF], [*SheepNoise*→ • baa,EOF]}

Goto(CC_0 , baa)

- Loop produces

Item	*From*
[*SheepNoise*→baa•SheepNoise, EOF]	Item 2 in CC_0
[*SheepNoise*→baa•, EOF]	Item 3 in CC_0

- Closure adds

Item	*From*
[*SheepNoise*→•baa SheepNoise, EOF]	Item 1 in CC_1
[*SheepNoise*→•baa, EOF]	Item 1 in CC_1

Example from SheepNoise

CC_0 : { [*Goal*→ • *SheepNoise*, EOF], [*SheepNoise*→ • baa SheepNoise, EOF],
[*SheepNoise*→• baa, EOF]}

CC_1 = *Goto*(CC_0 , SheepNoise) = {[*Goal*→ SheepNoise •, EOF]}

CC_2 = *Goto*(CC_0 , *baa*) =
{[*SheepNoise*→ *baa* • SheepNoise, EOF] , [*SheepNoise*→ *baa* •, EOF],
[*SheepNoise*→ • baa, EOF], [*SheepNoise*→ • baa SheepNoise, EOF]}

CC_3 = *Goto*(CC_2 , SheepNoise) = {[*SheepNoise*→ baa SheepNoise •, EOF] }

Building the Canonical Collection

Start from CC_0 = *closure*([$S' \rightarrow S$,<u>EOF</u>])

Repeatedly construct new states, until all are found

The algorithm

$s_0 \leftarrow$ *closure*([$S' \rightarrow S$,<u>EOF</u>])
$S \leftarrow \{ s_0 \}$
$k \leftarrow 1$

while (*S is still changing*)
  $\forall s_j \in S$ *and* $\forall x \in (T \cup NT)$
    $s_k \leftarrow goto(s_j, x)$
    record $s_j \rightarrow s_k$ *on* x
  if $s_k \notin S$ *then*
    $S \leftarrow S \cup s_k$
    $k \leftarrow k + 1$

- Fixed-point computation
- Loop adds to S
- $S \subseteq 2^{ITEMS}$, so S is finite

Example from SheepNoise

Starts with CC_0

CC_0 : { [***Goal***→ • ***SheepNoise***, EOF], [***SheepNoise***→ • **baa SheepNoise**, EOF], [***SheepNoise***→ • **baa**, EOF]}

Example from SheepNoise

Starts with CC_0

CC_0 : { [*Goal*→ • *SheepNoise*, EOF], [*SheepNoise*→ • baa SheepNoise, EOF],
[*SheepNoise*→• baa, EOF]}

Iteration 1 computes

CC_1 = *Goto*(CC_0 , SheepNoise) = { [*Goal*→ SheepNoise •, EOF]}

CC_2 = *Goto*(CC_0 , *baa*) =
{[*SheepNoise*→ *baa* • SheepNoise, EOF] , [*SheepNoise*→ *baa* •, EOF],
[*SheepNoise*→ • baa, EOF], [*SheepNoise*→ • baa SheepNoise, EOF]}

Example from SheepNoise

Starts with CC_0

CC_0 : { [*Goal*→ • *SheepNoise*, EOF], [*SheepNoise*→ • baa SheepNoise, EOF],
[*SheepNoise*→ • baa, EOF]}

Iteration 1 computes

CC_1 = *Goto*(CC_0 , SheepNoise) = { [*Goal*→ SheepNoise •, EOF]}

CC_2 = *Goto*(CC_0 , *baa*) =
{[*SheepNoise*→ *baa* • SheepNoise, EOF], [*SheepNoise*→ *baa* •, EOF],
[*SheepNoise*→ • baa, EOF], [*SheepNoise*→ • baa SheepNoise, EOF]}

Iteration 2 computes

CC_3 = *Goto*(CC_2 , SheepNoise) = { [*SheepNoise*→ baa SheepNoise •, EOF] }

Example from SheepNoise

Starts with CC_0

CC_0 : { [*Goal*→ • *SheepNoise*, EOF], [*SheepNoise*→ • baa SheepNoise, EOF],
[*SheepNoise*→ • baa, EOF]}

Iteration 1 computes

CC_1 = *Goto*(CC_0 , SheepNoise) = { [*Goal*→ SheepNoise •, EOF]}

CC_2 = *Goto*(CC_0 , *baa*) =
{[*SheepNoise*→ *baa* • SheepNoise, EOF], [*SheepNoise*→ *baa* •, EOF],
[*SheepNoise*→ • baa, EOF], [*SheepNoise*→ • baa SheepNoise, EOF]}

Iteration 2 computes

CC_3 = *Goto*(CC_2 , SheepNoise) = { [*SheepNoise*→ baa SheepNoise •, EOF] }

Nothing more to compute, since • is at the end of item in CC_3.