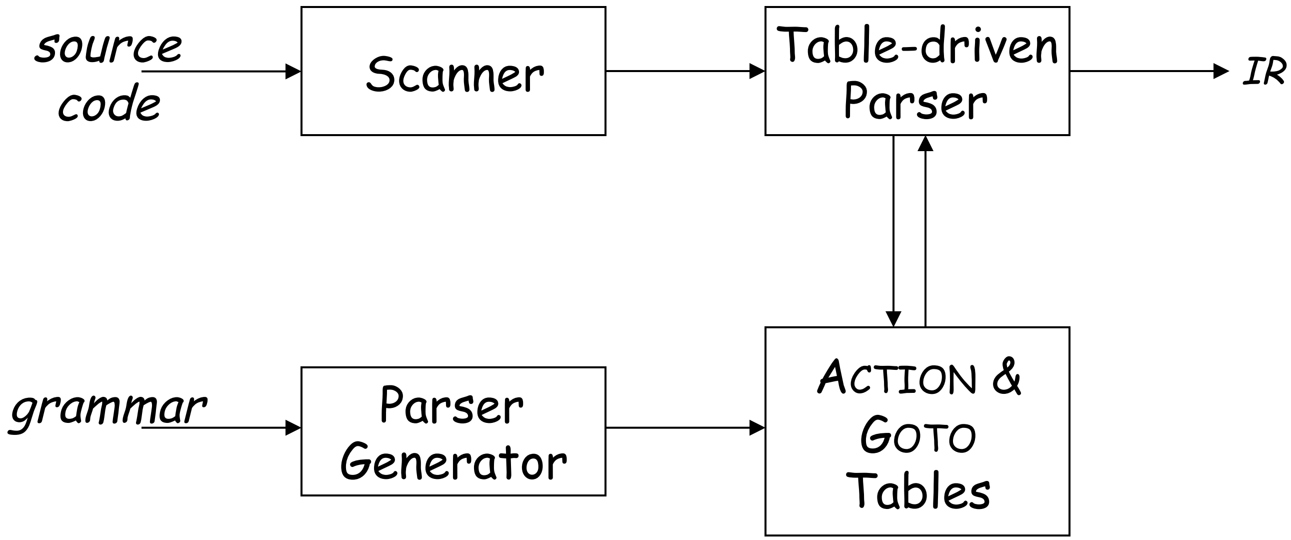
LR(1) Parsers

LR(1) Parsers

A table-driven LR(1) parser looks like

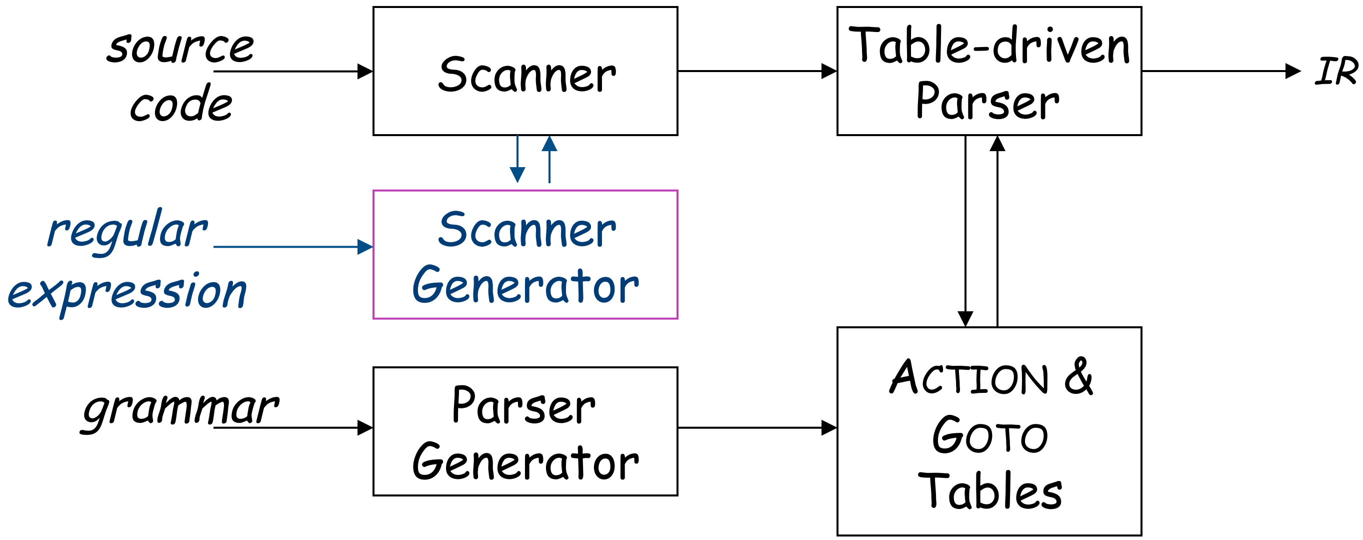


Tables *can* be built by hand

However, this is a perfect task to automate

LR(1) Parsers

A table-driven LR(1) parser looks like



Tables *can* be built by hand

However, this is a perfect task to automate

Just like automating construction of scanners …

Except that compiler writers use parser generators …

LR(1) Skeleton Parser

```
stack.push(INVALID);
stack.push(s0);                       // initial state
token = scanner.next_token();
loop forever {
    s = stack.top();
    if ( ACTION[s,token] == "reduce A→β" ) then {
        stack.popnum(2*|β|);          // pop 2*|β| symbols
        s = stack.top();
        stack.push(A);                // push A
        stack.push(GOTO[s,A]);        // push next state
    }
    else if ( ACTION[s,token] == "shift si" ) then {
        stack.push(token); stack.push(si);
        token ← scanner.next_token();
    }
    else if ( ACTION[s,token] == "accept"
                        & token == EOF )
        then break;
    else throw a syntax error;
}
report success;
```

The skeleton parser

- Relies on a stack

LR(1) Skeleton Parser

```
stack.push(INVALID);
stack.push(s0);                          // initial state
token = scanner.next_token();
loop forever {
    s = stack.top();
    if ( ACTION[s,token] == "reduce A→β" ) then {
        stack.popnum(2*|β|);         // pop 2*|β| symbols
        s = stack.top();
        stack.push(A);                 // push A
        stack.push(GOTO[s,A]);  // push next state
    }
    else if ( ACTION[s,token] == "shift si" ) then {
        stack.push(token); stack.push(si);
        token ← scanner.next_token();
    }
    else if ( ACTION[s,token] == "accept"
                    & token == EOF )
        then break;
    else throw a syntax error;
}
report success;
```

The skeleton parser

- Relies on a scanner

LR(1) Skeleton Parser

```
stack.push(INVALID);
stack.push(s0);                          // initial state
token = scanner.next_token();
loop forever {
    s = stack.top();
    if ( ACTION[s,token] == "reduce A→β" ) then {
        stack.popnum(2*|β|);          // pop 2*|β| symbols
        s = stack.top();
        stack.push(A);                // push A
        stack.push(GOTO[s,A]);        // push next state
    }
    else if ( ACTION[s,token] == "shift si" ) then {
        stack.push(token); stack.push(si);
        token ← scanner.next_token();
    }
    else if ( ACTION[s,token] == "accept"
                        & token == EOF )
        then break;
    else throw a syntax error;
}
report success;
```

The skeleton parser

- Relies on two tables, called ACTION and GOTO

LR(1) Skeleton Parser

```
stack.push(INVALID);
stack.push(s0);                    // initial state
token = scanner.next_token();
loop forever {
    s = stack.top();
    if ( ACTION[s,token] == "reduce A→β" ) then {
        stack.popnum(2*|β|);       // pop 2*|β| symbols
        s = stack.top();
        stack.push(A);             // push A
        stack.push(GOTO[s,A]);     // push next state
    }
    else if ( ACTION[s,token] == "shift si" ) then {
        stack.push(token); stack.push(si);
        token ← scanner.next_token();
    }
    else if ( ACTION[s,token] == "accept"
                  & token == EOF )
        then break;
    else throw a syntax error;
}
report success;
```

The skeleton parser

- Accepts at most once
- Detects errors by failure of all three cases
- Follows the intuitive shift-reduce parser from last lecture

LR(1) Parsers (*parse tables*)

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

The grammar

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

The tables

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

GOTO Table	
State	*SheepNoise*
0	1
1	0
2	0
3	0

Example Parse 1

The string baa

Stack	Input	Action
$ s_0	baa EOF	

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

s_0 is on top of stack
baa is lookahead

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

GOTO Table	
State	*SheepNoise*
0	1
1	0
2	0
3	0

LR(1) Skeleton Parser

```
stack.push(INVALID);
stack.push(s0);                          // initial state
token = scanner.next_token();
loop forever {
    s = stack.top();
    if ( ACTION[s,token] == "reduce A→β" ) then {
      stack.popnum(2*|β|);        // pop 2*|β| symbols
       s = stack.top();
       stack.push(A);                    // push A
       stack.push(GOTO[s,A]);  // push next state
    }
    else if ( ACTION[s,token] == "shift si" ) then {
        stack.push(token); stack.push(si);
        token ← scanner.next_token();
    }
    else if ( ACTION[s,token] == "accept"
                      & token == EOF )
        then break;
    else throw a syntax error;
}
report success;
```

s_0 is on top of stack
baa is lookahead token

ACTION[0, baa]

ACTION Table

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

LR(1) Skeleton Parser

```
stack.push(INVALID);
stack.push(s0);                        // initial state
token = scanner.next_token();
loop forever {
    s = stack.top();
    if ( ACTION[s,token] == "reduce A→β" ) then {
        stack.popnum(2*|β|);       // pop 2*|β| symbols
        s = stack.top();
        stack.push(A);             // push A
        stack.push(GOTO[s,A]);     // push next state
    }
    else if ( ACTION[s,token] == "shift si" ) then {
        stack.push(token); stack.push(si);
        token ← scanner.next_token();
    }
    else if ( ACTION[s,token] == "accept"
                    & token == EOF )
        then break;
    else throw a syntax error;
}
report success;
```

s_0 is on top of stack
baa is lookahead token
ACTION[0, baa]

ACTION Table

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

Example Parse 1

The string baa

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

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

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

GOTO Table	
State	*SheepNoise*
0	1
1	0
2	0
3	0

LR(1) Skeleton Parser

```
stack.push(INVALID);
stack.push(s0);                       // initial state
token = scanner.next_token();
loop forever {
    s = stack.top();
    if ( ACTION[s,token] == "reduce A→β" ) then {
        stack.popnum(2*|β|);          // pop 2*|β| symbols
        s = stack.top();
        stack.push(A);                // push A
        stack.push(GOTO[s,A]);        // push next state
    }
    else if ( ACTION[s,token] == "shift si" ) then {
        stack.push(token); stack.push(si);
        token ← scanner.next_token();
    }
    else if ( ACTION[s,token] == "accept"
                        & token == EOF )
        then break;
    else throw a syntax error;
}
report success;
```

s_2 is on top of stack
EOF is lookahead token

ACTION[2, EOF]

ACTION Table

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

LR(1) Skeleton Parser

```
stack.push(INVALID);
stack.push(s0);                      // initial state
token = scanner.next_token();
loop forever {
    s = stack.top();
    if ( ACTION[s,token] == "reduce A→β" ) then {
        stack.popnum(2*|β|);         // pop 2*|β| symbols
        s = stack.top();
        stack.push(A);               // push A
        stack.push(GOTO[s,A]);       // push next state
    }
    else if ( ACTION[s,token] == "shift si" ) then {
        stack.push(token); stack.push(si);
        token ← scanner.next_token();
    }
    else if ( ACTION[s,token] == "accept"
                    & token == EOF )
        then break;
    else throw a syntax error;
}
report success;
```

3	SheepNoise	→	<u>baa</u>

s_2 is on top of stack
EOF is lookahead token

ACTION[2, EOF]

ACTION Table

State	EOF	<u>baa</u>
0	—	*shift 2*
1	*accept*	*shift 3*
2	*reduce 3*	*reduce 3*
3	*reduce 2*	*reduce 2*

LR(1) Skeleton Parser

```
stack.push(INVALID);
stack.push(s0);                    // initial state
token = scanner.next_token();
loop forever {
    s = stack.top();
    if ( ACTION[s,token] == "reduce A→β" ) then {
        stack.popnum(2*|β|);       // pop 2*|β| symbols
        s = stack.top();
        stack.push(A);             // push A
        stack.push(GOTO[s,A]);     // push next state
    }
    else if ( ACTION[s,token] == "shift si" ) then {
        stack.push(token); stack.push(si);
        token ← scanner.next_token();
    }
    else if ( ACTION[s,token] == "accept"
                  & token == EOF )
        then break;
    else throw a syntax error;
}
report success;
```

3	SheepNoise	→	<u>baa</u>

After 2 pops,
s_0 is on top of stack

Stack
\$ s_0 <u>baa</u> s_2
\$ s_0

GOTO[0, SN]

GOTO Table	
State	*SheepNoise*
0	1
1	0
2	0
3	0

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*	→	*SheepNoise* baa
3		|	baa

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

GOTO Table	
State	*SheepNoise*
0	1
1	0
2	0
3	0

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*	→	*SheepNoise* baa
3		|	baa

Notice that we never cleared the stack — the table construction moved *accept* earlier by one action

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

GOTO Table	
State	*SheepNoise*
0	1
1	0
2	0
3	0

Example Parse 2

The string baa baa

Stack	Input	Action
$\$ \; s_0$	baa baa EOF	

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

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

GOTO Table	
State	*SheepNoise*
0	1
1	0
2	0
3	0

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*	→	*SheepNoise* baa
3		|	baa

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

GOTO Table	
State	*SheepNoise*
0	1
1	0
2	0
3	0

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*	→	*SheepNoise* baa
3		|	baa

```
   ...
if ( ACTION[s,token] == "reduce A→β" ) then {
       stack.popnum(2*|β|);      // pop 2*|β| symbols
         s = stack.top();
         stack.push(A);             // push A
         stack.push(GOTO[s,A]);  // push next state
   }
   ...
```

ACTION Tabl		
State	EOF	
0	—	
1	*accept*	
2	*reduce 3*	*reduce 3*
3	*reduce 2*	*reduce 2*

2	0
3	0

Example Parse 2

The string baa baa

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

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

Last example, we faced EOF and we accepted. With baa, we shift …

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

GOTO Table	
State	*SheepNoise*
0	1
1	0
2	0
3	0

Example Parse 2

The string baa baa

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

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

...

```
else if ( ACTION[s,token] == "shift si" ) then {
      stack.push(token); stack.push(si);
      token ← scanner.next_token();
}
```

...

ACTION Ta

State	EOF	
0	—	
1	*accept*	*shift 3*
2	*reduce 3*	*reduce 3*
3	*reduce 2*	*reduce 2*

1	0
2	0
3	0

Example Parse 2

The string baa baa

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

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

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

GOTO Table	
State	*SheepNoise*
0	1
1	0
2	0
3	0

Example Parse 2

The string baa baa

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

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

	ACTION Ta[bl]e	
State	EOF	
0	—	
1	*accept*	
2	*reduce 3*	
3	*reduce 2*	*reduce 2*

3	0

```
    ...
if ( ACTION[s,token] == "reduce A→β" ) then {
        stack.popnum(2*|β|);        // pop 2*|β| symbols
          s = stack.top();
          stack.push(A);                // push A
          stack.push(GOTO[s,A]);  // push next state
    }
    ...
```

Example Parse 2

The string <u>baa</u> <u>baa</u>

Stack	Input	Action
$ s_0	<u>baa</u> <u>baa</u> EOF	*shift 2*
$ s_0 <u>baa</u> s_2	<u>baa</u> EOF	*reduce 3*
$ s_0 *SN* s_1	<u>baa</u> EOF	*shift 3*
$ s_0 *SN* s_1 <u>baa</u> s_3	EOF	*reduce 2*
$ s_0 *SN* s_1	EOF	

1	*Goal*	→	*SheepNoise*
2	*SheepNoise*	→	*SheepNoise* <u>baa</u>
3		|	<u>baa</u>

Now, we accept

ACTION Table		
State	EOF	<u>baa</u>
0	—	*shift 2*
1	*accept*	*shift 3*
2	*reduce 3*	*reduce 3*
3	*reduce 2*	*reduce 2*

GOTO Table	
State	*SheepNoise*
0	1
1	0
2	0
3	0

Example Parse 2

The string baa baa

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

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

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

GOTO Table	
State	*SheepNoise*
0	1
1	0
2	0
3	0

Building LR(1) Tables : ACTION and GOTO

How do we build the parse tables for an LR(1) grammar?

- Use the grammar to build a model of the Control DFA
- Encode actions & transitions into the ACTION & GOTO tables
- If construction succeeds, the grammar is LR(1)
  - "Succeeds" means defines each table entry uniquely

Building LR(1) Tables: The Big Picture

- Model the state of the parser with "LR(1) items"
- Use two functions *goto(s, X)* and *closure(s)*
  - *goto()* is analogous to *move()* in the subset construction
  - *closure()* adds information to round out a state
- Build up the states and transition functions of the DFA ← This is a fixed-point algorithm
- Use this information to fill in the ACTION and GOTO tables

LR(1) Items

We represent valid configuration of LR(1) parser with a data structure called an LR(1) item

An LR(1) item is a pair [P, δ], where

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

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