RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers

LR Parsing & Attribute Grammar

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Review – LR(1) Table Construction

High-level overview

1. Build the canonical collection of sets of LR(1) Items, *I*
   a. Begin in an appropriate state, s_0
      - Assume: $S' \rightarrow S$, and S' is unique start symbol that does not occur on any RHS of a production (extended CFG - ECFG)
      - $[S' \rightarrow \bullet S, \underline{EOF}]$, along with any equivalent items
      - Derive equivalent items as *closure(s_0)*
   b. Repeatedly compute, for each s_k, and each *X*, *goto*(s_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 table from the collection of sets of LR(1) items

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

Review: LR(1) items

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

An LR(*k*) 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 of length ≤ *k* (words or EOF)

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

LR(1):

[$A \rightarrow \bullet \beta\gamma$, 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

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

[$A \rightarrow \beta\gamma \bullet$, a] means that the parser has seen βγ, *and* that a lookahead symbol of a is consistent with reducing to *A*.

Review - Computing Closures

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

- Any item $[A \rightarrow \beta \bullet B\delta, a]$ implies $[B \rightarrow \bullet\tau, x]$ for each production with *B* on the *lhs*, and each $x \in \text{FIRST}(\delta a)$ – for LR(1) item

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
```

- Classic fixed-point method
- Halts because $s \subset$ ITEMS

  Closure "fills out" a state

Review - 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 [$A \rightarrow \beta X \bullet \delta, \underline{a}$] *(easy part)*
- Should also includes *closure(* [$A \rightarrow \beta X \bullet \delta, \underline{a}$] *)* *(fill out the state)*

The algorithm

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

- Not a fixed-point method!
- Straightforward computation
- Uses *closure( )*

Goto() moves forward

Review - Building the Canonical Collection

Start from s_0 = *closure*([S' → S, EOF])

Repeatedly construct new states, until all are found

The algorithm

```
cc0 ← closure( [S' → •S, EOF])
CC ← { cc0 }
while ( new sets are still being added to CC)
   for each unmarked set ccj ∈ CC
      mark ccj as processed
      for each x following a • in an item in ccj
         temp ← goto(ccj, x)
         if temp ∉ CC
            then CC ← CC ∪ { temp }
         record transitions from ccj to temp on x
```

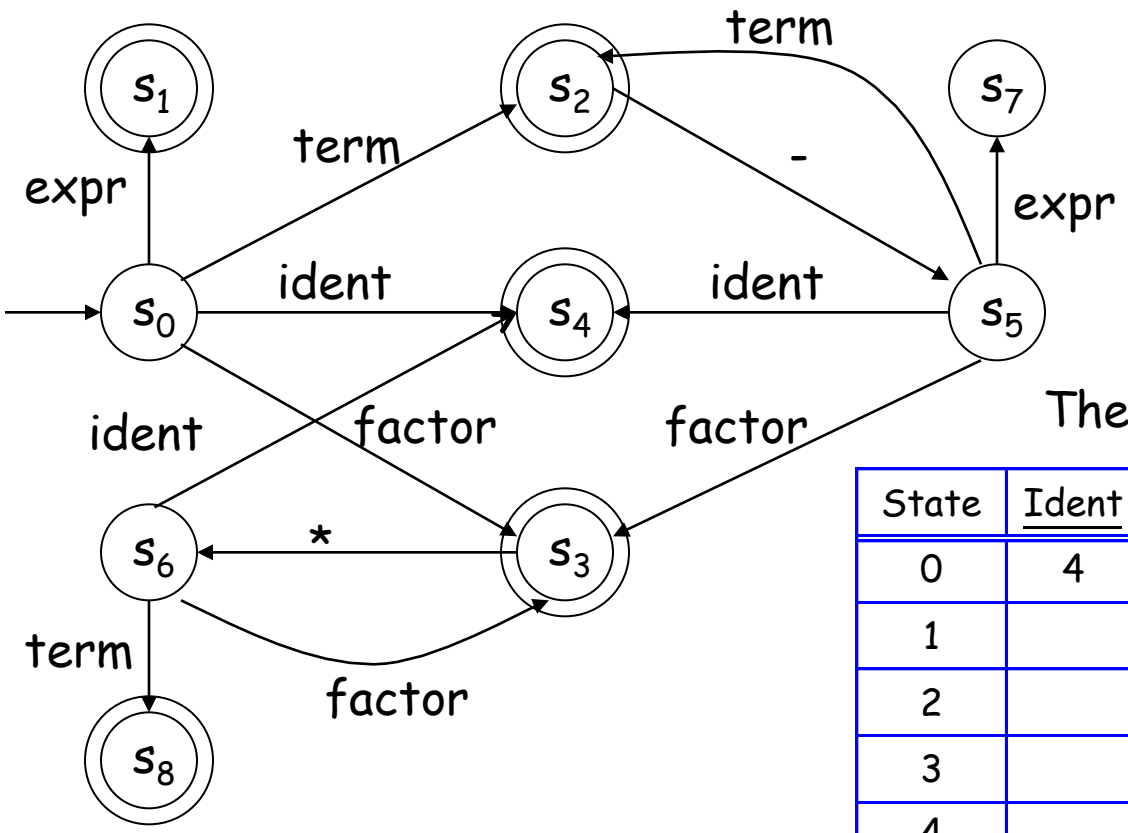
- Fixed-point computation (worklist version)
- Loop adds to *CC*
- $CC \subseteq 2^{ITEMS}$, so *CC* is finite

Review: Expression Grammar Example

1: *Goal* → *Expr*
2: *Expr* → *Term* - *Expr*
3: *Expr* → *Term*
4: *Term* → *Factor* * *Term*
5: *Term* → *Factor*
6: *Factor* → *ident*

Symbol	FIRST
Goal	{ ident }
Expr	{ ident }
Term	{ ident }
Factor	{ ident }
-	{ - }
*	{ * }
ident	{ ident }

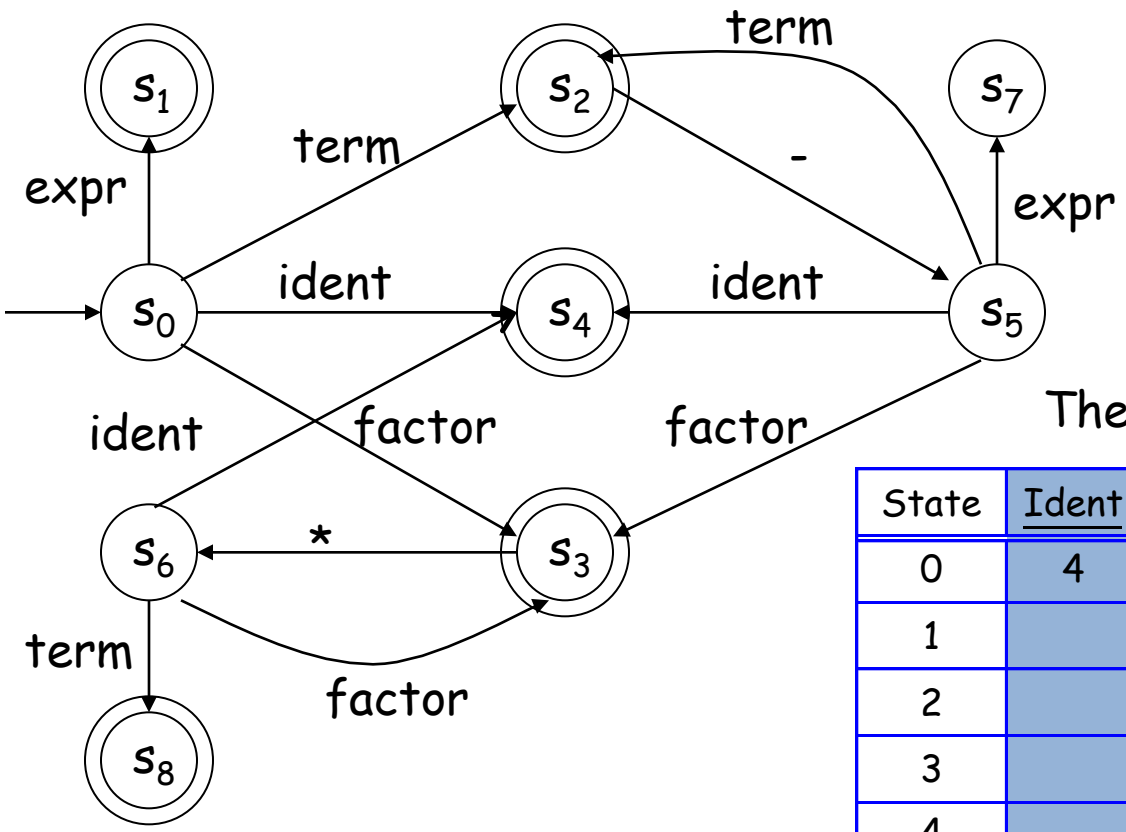
Example (DFA)



The State Transition Table

State	Ident	-	*	Expr	Term	Factor
0	4			1	2	3
1						
2		5				
3			6			
4						
5	4			7	2	3
6	4				8	3
7						
8						

Example (DFA)



The State Transition Table

State	Ident	-	*	Expr	Term	Factor
0	4			1	2	3
1						
2		5				
3			6			
4						
5	4			7	2	3
6	4				8	3
7						
8						

Filling in the ACTION and GOTO Tables

The algorithm

∀ *set* $s_x \in S$
  ∀ *item* $i \in s_x$
    if *i is* $[A \rightarrow \beta \cdot \underline{a}d, \underline{b}]$ *and* $goto(s_x, \underline{a}) = s_k$, $\underline{a} \in T$
      then ACTION$[x, \underline{a}] \leftarrow$ "*shift k*"
    else if *i is* $[S' \rightarrow S \cdot, EOF]$
      then ACTION$[x, EOF] \leftarrow$ "*accept*"
    else if *i is* $[A \rightarrow \beta \cdot, \underline{a}]$
      then ACTION$[x, \underline{a}] \leftarrow$ "*reduce* $A \rightarrow \beta$"
  ∀ $n \in NT$
    if $goto(s_x, n) = s_k$
      then GOTO$[x, n] \leftarrow k$

Many items generate no table entry

Example (*Filling in the tables*)

The algorithm produces LR(1) parse table

	ACTION				GOTO		
	Ident	-	*	EOF	*Expr*	*Term*	*Factor*
0	s 4				1	2	3
1				acc			
2		s 5		r 3			
3		r 5	s 6	r 5			
4		r 6	r 6	r 6			
5	s 4				7	2	3
6	s 4					8	3
7				r 2			
8		r 4		r 4			

Plugs into the skeleton LR(1) parser

Remember the state transition table?

State	Ident	-	*	Expr	Term	Factor
0	4			1	2	3
1						
2		5				
3			6			
4						
5	4			7	2	3
6	4				8	3
7						
8						

What can go wrong?

What if set *s* contains [$A \rightarrow \beta \cdot \underline{a}\gamma, \underline{b}$] and [$B \rightarrow \beta \cdot, \underline{a}$] ?

- First item generates "shift", second generates "reduce"
- Both define ACTION[s,$\underline{a}$] — cannot do both actions
- This is a fundamental ambiguity, called a *shift/reduce error*
- Modify the grammar to eliminate it *(if-then-else)*

EaC includes a worked example

What if set *s* contains [$A \rightarrow \gamma \cdot, \underline{a}$] and [$B \rightarrow \gamma \cdot, \underline{a}$] ?

- Each generates "reduce", but with a different production
- Both define ACTION[s,$\underline{a}$] — cannot do both reductions
- This fundamental ambiguity is called a *reduce/reduce error*
- Modify the grammar to eliminate it

In either case, the grammar is not LR(1)

Shrinking the Tables

Three options:

- Combine terminals such as number & identifier, + & -, * & /
  - → Directly removes a column, may remove a row
  - → For expression grammar, 198 (vs. 384) table entries
- Combine rows or columns (*table compression*)
  - → Implement identical rows once & remap states
  - → Requires extra indirection on each lookup
  - → Use separate mapping for ACTION & for GOTO
- Use another construction algorithm
  - → Both LALR(1) and SLR(1) produce smaller tables
  - → Implementations are readily available

LR(0) versus SLR(1) versus LR(1)

LR(0) ? -- set of LR(0) items as states

LR(1) ? -- set of LR(1) items as states, may have different states compared with LR(0)

SLR(1) ? -- LR(0) items and canonical sets, same as LR(0)

SLR(1): add FOLLOW(A) to each LR(0) item $[A \rightarrow \gamma \bullet]$ as its second component: $[A \rightarrow \gamma \bullet, \underline{a}]$, $\forall a \in FOLLOW(A)$

LR(0) versus SLR(1) versus LR(1)

Example:

$$S' \rightarrow S$$
$$S \rightarrow S\ ;\ a \mid a$$

LR(0) ?

LR(1) ?

SLR(1) ?

LR(0) versus LR(1) versus SLR(1)

S' → S
S → S ; a | a

LR(0) States

s0 = Closure({[S' → •S]}) = {[S' -> •S], [S -> •S; a], [S -> •a] }

s1 = GoTo (s0, S) =
{[S' → S.],
[S → S•; a] }

s2 = GoTo (s0, a) =
{[S → a•]}

s3 = GoTo (s1, ;) =
{[S → S; • a]}

s4 = GoTo (s3, a) =
{[S → S;a •] }

LR(1) States

s0 = Closure({[S' → •S,eof]}) = {[S' -> •S,eof], [S -> •S; a,eof], [S -> •a,;] }

s1 = GoTo (s0, S) =
{[S' → S •, eof],
[S → S•; a, eof] }

s2 = GoTo (s0, a) =
{[S → a•, ;]}

s3 = GoTo (s1, ;) =
{[S → S; • a, eof]}

s4 = GoTo (s3, a) =
{[S → S;a •, eof] }

Grammar is not LR(0), but LR(1) and SLR(1)

LALR(1) versus LR(1)

LALR(1) ? LR(1) items, State -> Grouped LR(1) states

LALR(1): Merge two sets of LR(1) items (states), if they have the same core.

core of set of LR(1) items: set of LR(0) items derived by ignoring the lookahead symbols

FACT: collapsing LR(1) states into LALR(1) states cannot introduce shift/reduce conflicts

LALR(1) versus LR(1)

S' → S
S → S ; a | a

s0 = Closure({[S' → •S, eof]})

s1 = Closure(GoTo (s0, a)) =
{[S → a • Ad, eof],
[S → a • Be, eof],
[A → •c, d], [B → •c, e]}

s2 = Closure(GoTo (s0, b)) =
{[S → b • Ae, eof],
[S → b • Bd, eof],
[A → •c, e], [B → •c, d]}

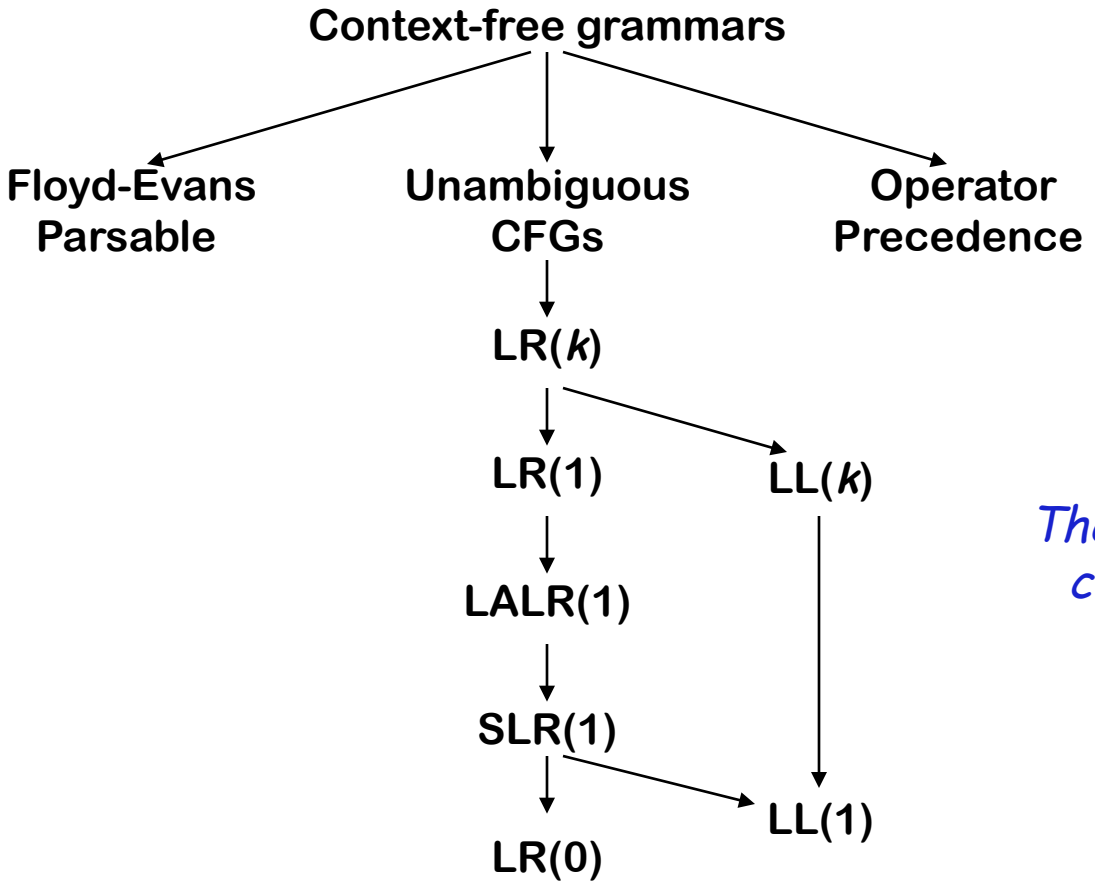
s3 = Closure(GoTo (s1, c)) =
{[A → c• , d],
[B → c• , e]}

s4 = Closure(GoTo (s2, c)) =
{[A → c• , e],
[B → c• , d]}

There are other states that are not listed here!

Grammar is LR(1), but not LALR(1), since collapsing s3 and s4 (same core) will introduce reduce-reduce conflict.

Hierarchy of Context-Free Grammars



- Operator precedence includes some ambiguous grammars
- LL(1) is a subset of SLR(1)

The inclusion hierarchy for context-free grammars

Ref Book: Michael Sipser, "Introduction to the Theory of Computation", 3rd Edition