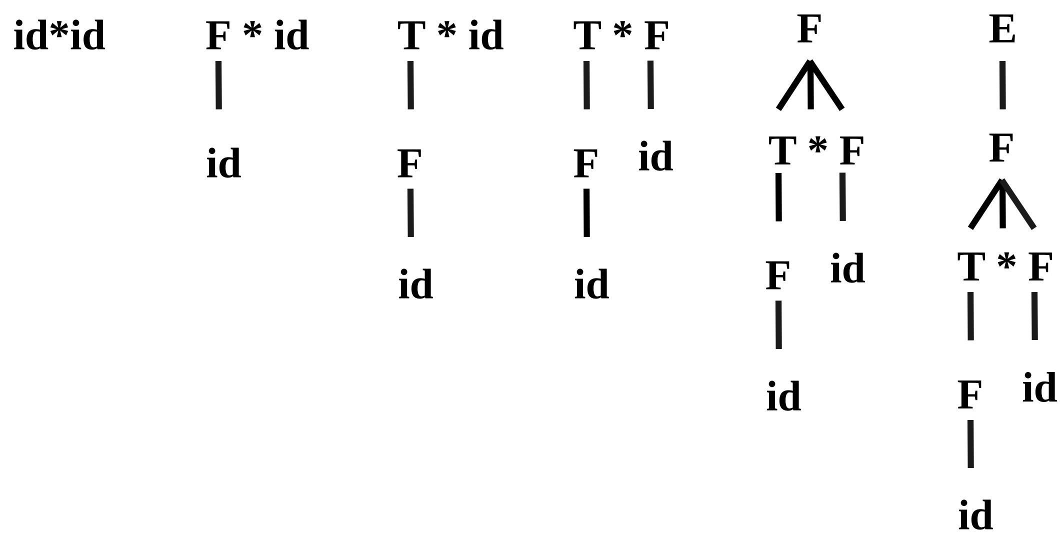
COMPILER DESIGN

Topic: Bottom-Up parsing

Introduction

- Constructs parse tree for an input string beginning at the leaves (the bottom) and working towards the root (the top)
- Example: id*id

E -> E + T | T
T -> T * F | F
F -> (E) | **id**



Shift-reduce parser

- The general idea is to shift some symbols of input to the stack until a reduction can be applied
- At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of the production
- The key decisions during bottom-up parsing are about when to reduce and about what production to apply
- A reduction is a reverse of a step in a derivation
- The goal of a bottom-up parser is to construct a derivation in reverse:
  - E=>T=>T*F=>T*id=>F*id=>id*id

Handle pruning

- A Handle is a substring that matches the body of a production and whose reduction represents one step along the reverse of a rightmost derivation

Right sentential form	Handle	Reducing production
id*id	id	F->id
F*id	F	T->F
T*id	id	F->id
T*F	T*F	E->T*F

Shift reduce parsing

- A stack is used to hold grammar symbols
- Handle always appear on top of the stack
- Initial configuration:

Stack	Input
$	w$

- Acceptance configuration

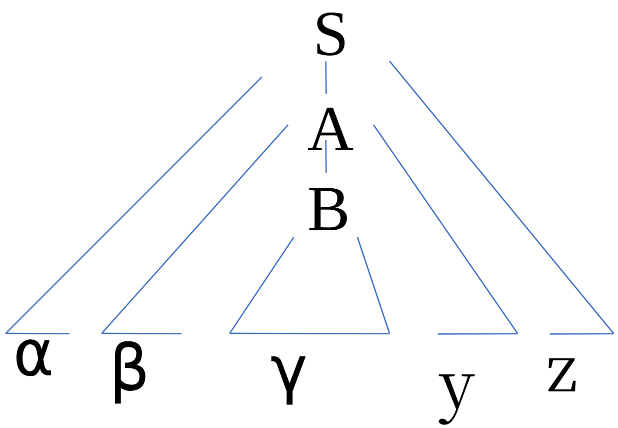
Stack	Input
$S	$

Shift reduce parsing (cont.)

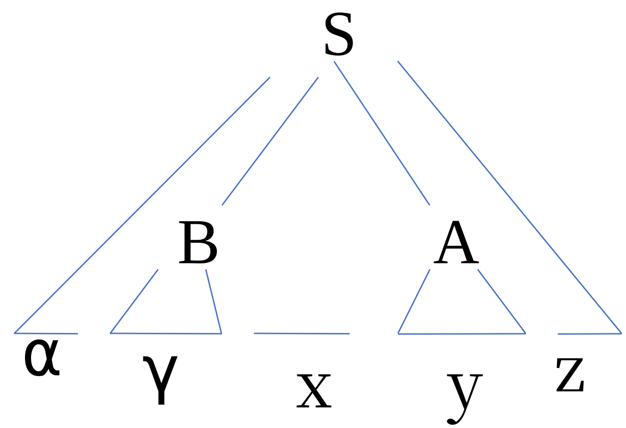
- Basic operations:
  - Shift
  - Reduce
  - Accept
  - Error
- Example: id*id

Stack	Input	Action
$	id*id$	shift
$id	*id$	reduce by F->id
$F	*id$	reduce by T->F
$T	*id$	shift
$T*	id$	shift
$T*id	$	reduce by F->id
$T*F	$	reduce by T->T*F
$T	$	reduce by E->T
$E	$	accept

Handle will appear on top of the stack



Stack	Input
$αβγ	yz$
$αβB	yz$
$αβBy	z$



Stack	Input
$αγ	xyz$
$αBxy	z$

Conflicts during shit reduce parsing

- Two kind of conflicts
  - Shift/reduce conflict
  - Reduce/reduce conflict
- Example:

stmt $\longrightarrow$ **if** expr **then** stmt
| **if** expr **then** stmt **else** stmt
| **other**

Stack	Input
… if expr then stmt	else …$

Reduce/reduce conflict

stmt -> id(parameter_list)
stmt -> expr:=expr
parameter_list->parameter_list, parameter
parameter_list->parameter
parameter->id
expr->id(expr_list)
expr->id
expr_list->expr_list, expr
expr_list->expr

Stack	Input
… id(id	,id) …$

LR Parsing

- The most prevalent type of bottom-up parsers
- LR(k), mostly interested on parsers with k<=1
- Why LR parsers?
  - Table driven
  - Can be constructed to recognize all programming language constructs
  - Most general non-backtracking shift-reduce parsing method
  - Can detect a syntactic error as soon as it is possible to do so
  - Class of grammars for which we can construct LR parsers are superset of those which we can construct LL parsers

States of an LR parser

- States represent set of items
- An LR(0) item of G is a production of G with the dot at some position of the body:
  - For A->XYZ we have following items
    - A->.XYZ
    - A->X.YZ
    - A->XY.Z
    - A->XYZ.
  - In a state having A->.XYZ we hope to see a string derivable from XYZ next on the input.
  - What about A->X.YZ?

Constructing canonical LR(0) item sets

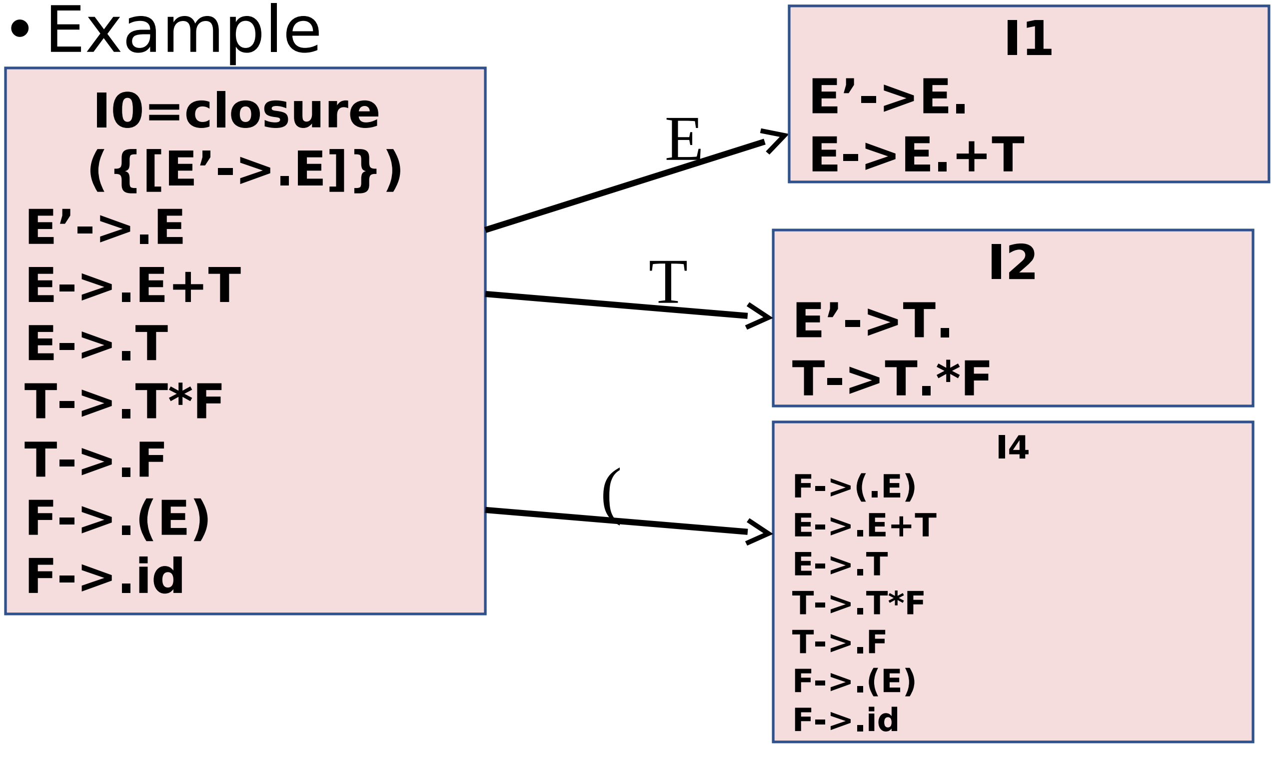
- Augmented grammar:
  - G with addition of a production: S'->S
- Closure of item sets:
  - If I is a set of items, closure(I) is a set of items constructed from I by the following rules:
    - Add every item in I to closure(I)
    - If A->α.Bβ is in closure(I) and B->γ is a production then add the item B->.γ to clsoure(I).
- Example:

E'->E
E -> E + T | T
T -> T * F | F
F -> (E) | **id**

I0=closure({[E'->.E]}
E'->.E
E->.E+T
E->.T
T->.T*F
T->.F
F->.(E)
F->.id

Constructing canonical LR(0) item sets (cont.)

- Goto (I,X) where I is an item set and X is a grammar symbol is closure of set of all items [A-> $\alpha X.\ \beta$] where [A-> $\alpha.X\ \beta$] is in I
- Example



Closure algorithm

```
SetOfItems CLOSURE(I) {
  J=I;
  repeat
        for (each item A-> α.Bβ in J)
              for (each production B->γ of G)
                    if (B->.γ is not in J)
                          add B->.γ to J;
  until no more items are added to J on one round;
  return J;
```

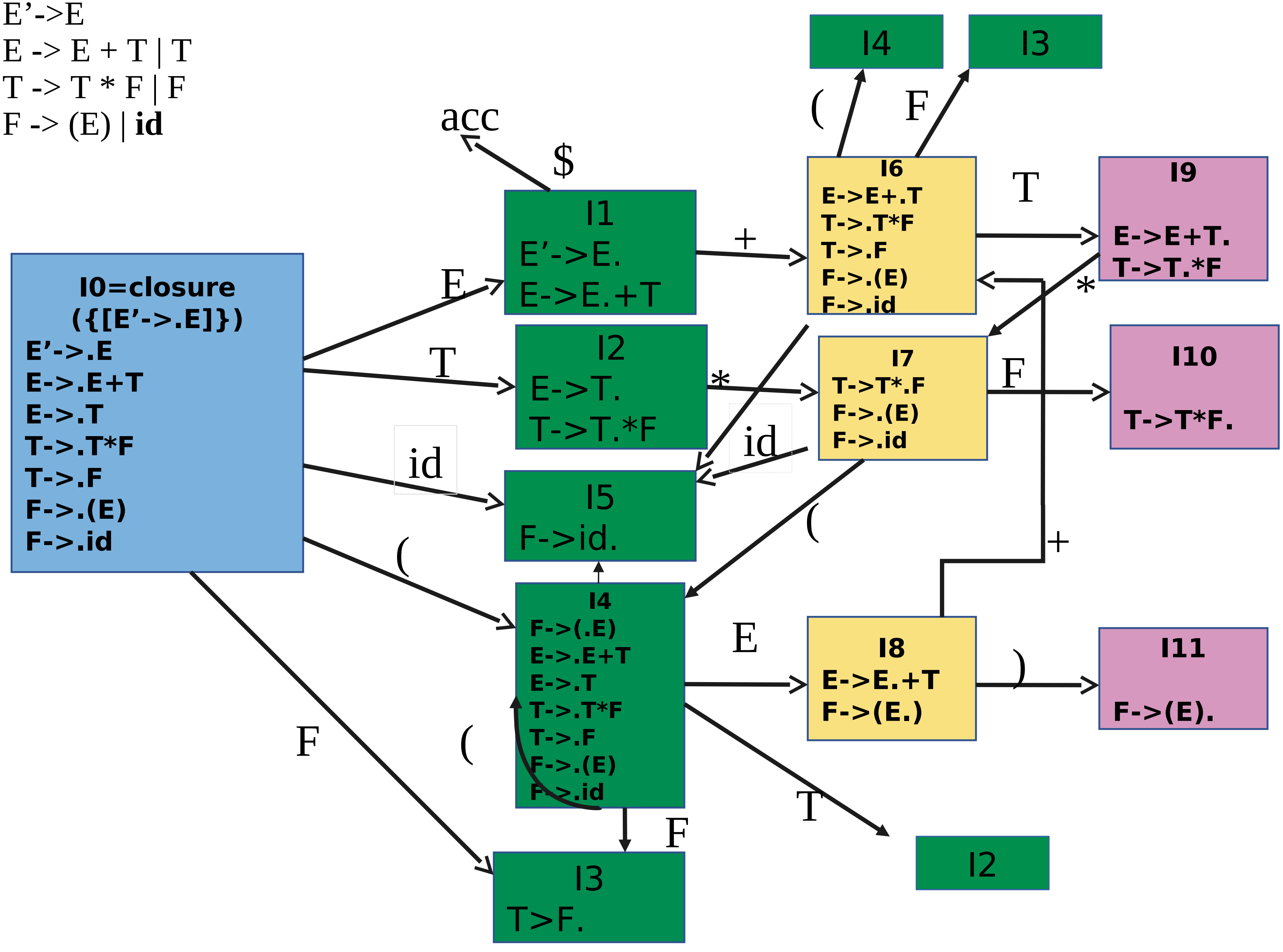
GOTO algorithm

```
SetOfItems  GOTO(I,X) {
   J=empty;
  if (A-> α.X β is in I)
        add CLOSURE(A-> αX. β ) to J;
  return J;
}
```

Canonical LR(0) items

```
Void items(G’) {
  C= CLOSURE({[S’->.S]});
  repeat
        for (each set of items I in C)
           for (each grammar symbol X)
              if (GOTO(I,X) is not empty and not in C)
                add GOTO(I,X) to C;
  until no new set of items are added to C on a round;
}
```

E’->E
E -> E + T | T
T -> T * F | F
F -> (E) | **id**



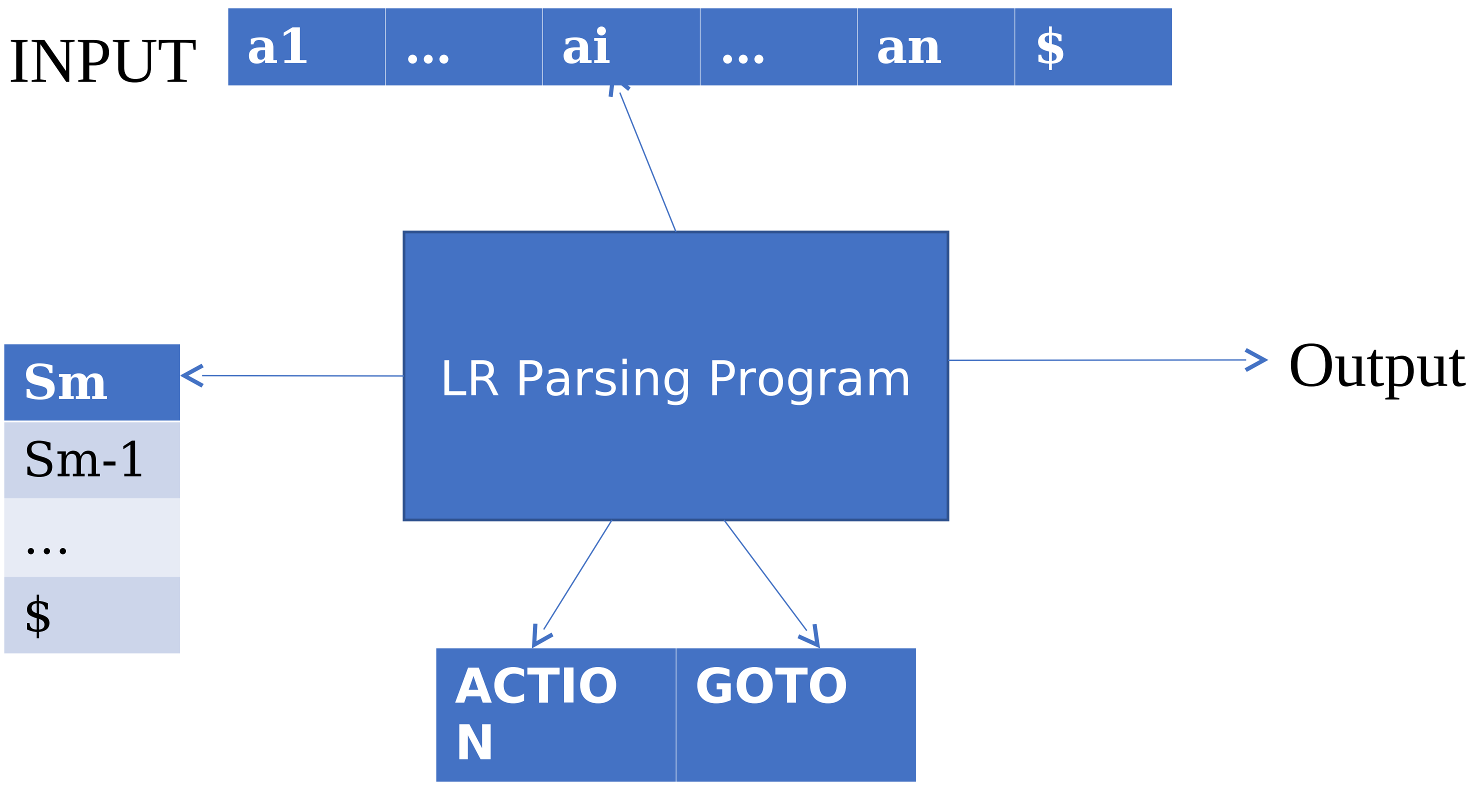
STATE	ACTON						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Acc			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Use of LR(0) automaton

- Example: id*id

Line	Stack	Symbols	Input	Action
(1)	0	$	id*id$	Shift to 5
(2)	05	$id	*id$	Reduce by F->id
(3)	03	$F	*id$	Reduce by T->F
(4)	02	$T	*id$	Shift to 7
(5)	027	$T*	id$	Shift to 5
(6)	0275	$T*id	$	Reduce by F->id
(7)	02710	$T*F	$	Reduce by T->T*F
(8)	02	$T	$	Reduce by E->T
(9)	01	$E	$	accept

LR-Parsing model



LR parsing algorithm

```
let a be the first symbol of w$;
while(1) { /*repeat forever */
  let s be the state on top of the stack;
  if (ACTION[s,a] = shift t) {
        push t onto the stack;
        let a be the next input symbol;
  } else if (ACTION[s,a] = reduce A->β) {
        pop |β| symbols of the stack;
        let state t now be on top of the stack;
        push GOTO[t,A] onto the stack;
        output the production A->β;
  } else if (ACTION[s,a]=accept) break; /* parsing is done */
  else call error-recovery routine;
}
```

Example

(0) E’->E
(1) E -> E + T
(2) E-> T
(3) T -> T * F
(4) T-> F
(5) F -> (E)
(6) F->**id**

STATE	ACTON						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Acc			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

id*id+id?

Line	Stack	Symbols	Input	Action
(1)	0		id*id+id$	Shift to 5
(2)	05	id	*id+id$	Reduce by F->id
(3)	03	F	*id+id$	Reduce by T->F
(4)	02	T	*id+id$	Shift to 7
(5)	027	T*	id+id$	Shift to 5
(6)	0275	T*id	+id$	Reduce by F->id
(7)	02710	T*F	+id$	Reduce by T->T*F
(8)	02	T	+id$	Reduce by E->T
(9)	01	E	+id$	Shift
(10)	016	E+	id$	Shift
(11)	0165	E+id	$	Reduce by F->id

Constructing SLR parsing table

- Method
  - Construct C={I0,I1, … , In}, the collection of LR(0) items for G’
  - State i is constructed from state Ii:
    - If [A->α.aβ] is in Ii and Goto(Ii,a)=Ij, then set ACTION[i,a] to “shift j”
    - If [A->α.] is in Ii, then set ACTION[i,a] to “reduce A->α” for all a in follow(A)
    - If {S’->.S] is in Ii, then set ACTION[I,$] to “Accept”
  - If any conflicts appears then we say that the grammar is not SLR(1).
  - If GOTO(Ii,A) = Ij then GOTO[i,A]=j
  - All entries not defined by above rules are made “error”
  - The initial state of the parser is the one constructed from the set of items containing [S’->.S]

Example grammar which is not SLR(1)

S -> L=R | R
L -> *R | id
R -> L

I0
S'->.S
S -> .L=R
S->.R
L -> .*R |
L->.id
R ->. L

I1
S'->S.

I2
S ->L.=R
R ->L.

I3
S ->R.

I4
L->*.R
R->.L
L->.*R
L->.id

I5
L -> id.

I6
S->L=.R
R->.L
L->.*R
L->.id

I7
L -> *R.

I8
R -> L.

I9
S -> L=R.

	Action =
2	Shift 6 Reduce R->L

More powerful LR parsers

- Canonical-LR or just LR method
  - Use lookahead symbols for items: LR(1) items
  - Results in a large collection of items
- LALR: lookaheads are introduced in LR(0) items

Canonical LR(1) items

- In LR(1) items each item is in the form: [A->α.β,a]
- An LR(1) item [A->α.β,a] is valid for a viable prefix γ if there is a derivation $S \overset{*}{\Rightarrow} \delta A w \underset{rm}{\Rightarrow} \delta\alpha\beta w$, where
  - Γ= δα
  - Either a is the first symbol of w, or w is ε and a is $
- Example:
  - S->BB
  - B->aB|b

$$S \overset{*}{\Rightarrow} aaBab \underset{rm}{\Rightarrow} aaaBab$$

Item [B->a.B,a] is valid for γ=aaa and w=ab

Constructing LR(1) sets of items

```
SetOfItems Closure(I) {
    repeat
            for (each item [A->α.Bβ,a] in I)
                    for (each production B->γ in G')
                            for (each terminal b in First(βa))
                                    add [B->.γ, b] to set I;
    until no more items are added to I;
    return I;
}

SetOfItems Goto(I,X) {
    initialize J to be the empty set;
    for (each item [A->α.Xβ,a] in I)
            add item [A->αX.β,a] to set J;
    return closure(J);
}

void items(G'){
    initialize C to Closure({[S'->.S,$]});
    repeat
            for (each set of items I in C)
                    for (each grammar symbol X)
                            if (Goto(I,X) is not empty and not in C)
                                    add Goto(I,X) to C;
    until no new sets of items are added to C;
}
```

Example

S’->S

S->CC

C->cC

C->d

Canonical LR(1) parsing table

- Method
  - Construct C={I0,I1, … , In}, the collection of LR(1) items for G’
  - State i is constructed from state Ii:
    - If [A->α.aβ, b] is in Ii and Goto(Ii,a)=Ij, then set ACTION[i,a] to “shift j”
    - If [A->α., a] is in Ii, then set ACTION[i,a] to “reduce A->α”
    - If {S’->.S,$] is in Ii, then set ACTION[I,$] to “Accept”
  - If any conflicts appears then we say that the grammar is not LR(1).
  - If GOTO(Ii,A) = Ij then GOTO[i,A]=j
  - All entries not defined by above rules are made “error”
  - The initial state of the parser is the one constructed from the set of items containing [S’->.S,$]

Example

S’->S

S->CC

C->cC

C->d

LALR Parsing Table

- For the previous example we had:

I4

C->d. , c/d

I47

C->d. , c/d/$

I7

C->d. , $

- State merges cant produce Shift-Reduce conflicts. Why?
- But it may produce reduce-reduce conflict

Example of RR conflict in state merging

S’->S

S -> aAd | bBd | aBe | bAe

A -> c

B -> c

An easy but space-consuming LALR table construction

- Method:
  1. Construct C={I0,I1,…,In} the collection of LR(1) items.
  2. For each core among the set of LR(1) items, find all sets having that core, and replace these sets by their union.
  3. Let C'={J0,J1,…,Jm} be the resulting sets. The parsing actions for state i, is constructed from Ji as before. If there is a conflict grammar is not LALR(1).
  4. If J is the union of one or more sets of LR(1) items, that is J = I1 UI2…IIk then the cores of Goto(I1,X), …, Goto(Ik,X) are the same and is a state like K, then we set Goto(J,X) =k.
- This method is not efficient, a more efficient one is discussed in the book

Compaction of LR parsing table

- Many rows of action tables are identical
  - Store those rows separately and have pointers to them from different states
  - Make lists of (terminal-symbol, action) for each state
  - Implement Goto table by having a link list for each nonterinal in the form (current state, next state)

Using ambiguous grammars

E->E+E

E->E*E

E->(E)

E->id

STATE	ACTON						GO TO
	id	+	*	(	)	$	E
0	S3			S2			1
1		S4	S5			Acc	
2	S3		S2				6
3		R4	R4		R4	R4	
4	S3			S2			7
5	S3			S2			8
6		S4	S5				
7		R1	S5		R1	R1	
8		R2	R2		R2	R2	
9		R3	R3		R3	R3	

I0: E’->.E
E->.E+E
E->.E*E
E->.(E)
E->.id

I1: E’->E.
E->E.+E
E->E.*E

I2: E->(.E)
E->.E+E
E->.E*E
E->.(E)
E->.id

I3: E->.id

I4: E->E+.E
E->.E+E
E->.E*E
E->.(E)
E->.id

I5: E->E*.E
E->(.E)
E->.E+E
E->.E*E
E->.(E)
E->.id

I6: E->(E.)
E->E.+E
E->E.*E

I7: E->E+E.
E->E.+E
E->E.*E

I8: E->E*E.
E->E.+E
E->E.*E

I9: E->(E).

Thank You