COMPILER DESIGN

Bottom-up parsing

Concepts

LR parsing table construction

Concepts of bottom-up parsing

Top-down vs. bottom-up parsing

- **Top-down** parsers construct a derivation from the root to the leaves of a parse tree
- **Bottom-up** parsers construct a reverse derivation from the leaves up to the root of the parse tree
  - can handle a larger class of grammars
  - most parser generators use bottom-up methods
  - requires less grammar transformations (generally no transformation at all)

Bottom-up parsing: concepts

- Often called "shift-reduce" parsers because they implement two main operations:
  - shift: pushes the lookahead symbol onto the stack and reads another token
  - reduce: matches a group of adjacent symbols β on the stack with the right-hand-side of a production and replaces β with the left-hand-side of the production (reverse application of a production). We reduce only if β is a handle.
  - handle: A handle of a sentential form is a portion of a sentential form that matches the right-hand-side of a production, and whose reduction to the non-terminal on the left-hand-side of the production represents one step along the reverse and rightmost derivation.

LL vs. LR parsing

```
    id + id * id
⇐   F  + id * id
⇐   T  + id * id
⇐   E  + id * id
⇐   E  + F  * id
⇐   E  + T  * id
⇐   E  + T  * F
⇐   E  + T
⇐   E
```

- LR(k) stands for
  - scan Left to right
  - compute Rightmost derivation
  - using k lookahead tokens

```
    E
⇒   E  + T
⇒   T  + T
⇒   F  + T
⇒   id + T
⇒   id + T  * F
⇒   id + F  * F
⇒   id + id * F
⇒   id + id * id
```

- LL(k) stands for
  - scan Left to right
  - compute Leftmost derivation
  - using k lookahead tokens

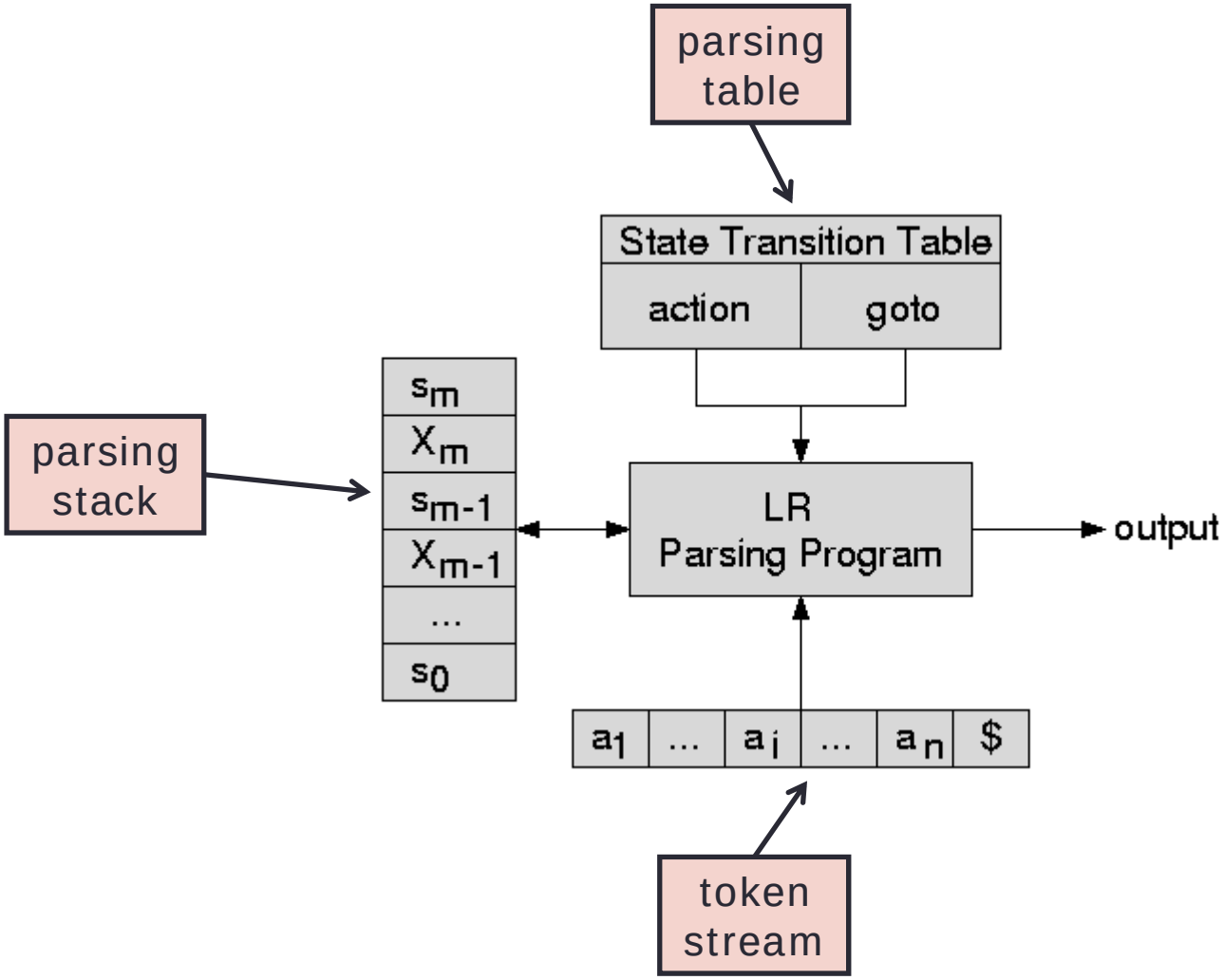
LR parsers

- LR parsers can be constructed to recognize virtually all programming language constructs for which a context-free grammar can be written
- The LR method is the most general non-backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods
- The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with LL predictive parsers
- An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input

LR parsers

- SLR: simple LR
  - easiest to implement, but the least powerful. May fail to produce a parsing table for some grammars.
- CLR: canonical LR
  - most powerful, general and expensive LR method.
- LALR: lookahead LR
  - intermediate in power and cost. Will work for most programming language constructs.

LR parsers


parsing table
State Transition Table
action
goto
parsing stack
s_m
X_m
s_{m-1}
X_{m-1}
...
s_0
LR Parsing Program
output
a_1 ... a_i ... a_n $
token stream

LR parsers

- All LR parsers are table-driven, which is derived from an FSA generated from the context-free grammar representing the language
- The same algorithm is used to parse, independently on the specific parsing method used
- Only the nature of the method used to generate the table distinguishes the parsing method used.
- All LR methods use a table that has two parts:
  - action: dictates whether a shift or reduce operation is needed
  - goto: dictates a state transition after a reduce operation

LR parsing algorithm

```
push(0)
x = top()
a = nextToken()
repeat forever
  if ( action[x,a] == shift s' )
    push(a)
    push(s')
    a = nextToken()
  else if ( action[x,a] == reduce A→β )
    multiPop(2*|β|)
    s' = top()
    push(A)
    push(goto[s',A])
    write(A→β)
  else if ( action[x,a] == accept )
    return true
  else
    return false
```

- 4 cases:
  - action is a shift
  - action is a reduce
  - action is accept
  - parsing error

LR parsing table: example

state	action						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			

1	E → E + T
2	E → T
3	T → T * F
4	T → F
5	F → (E)
6	F → id

	FOLLOW
E	{+,),$}
T	{+,*,),$}
F	{+,*,),$}

LR parsing: example

	stack	input	action	derivation
1	0	id+id*id$	shift 5	id+id*id
2	0id5	+id*id$	reduce (F→id)	⇐ F+id*id
3	0F3	+id*id$	reduce (T→F)	⇐ T+id*id
4	0T2	+id*id$	reduce (E→T)	⇐ E+id*id
5	0E1	+id*id$	shift 6	⇐ E+id*id
6	0E1+6	id*id$	shift 5	⇐ E+id*id
7	0E1+6id5	*id$	reduce (F→id)	⇐ E+F*id
8	0E1+6F3	*id$	reduce (T→F)	⇐ E+T*id
9	0E1+6T9	*id$	shift 7	⇐ E+T*id
10	0E1+6T9*7	id$	shift 5	⇐ E+T*id
11	0E1+6T9*7id5	$	reduce (F→id)	⇐ E+T*F
12	0E1+6T9*7F10	$	reduce (T→T*F)	⇐ E+T
13	0E1+6T9	$	reduce (E→E+T)	⇐ E
14	0E1	$	accept	

Construction of an LR parsing table

LR parsing tables

- Representation of the states of a DFA
- The states:
  - represent what has been parsed in the recent past on the way to recognize a handle
  - control how the DFA will respond to the next token in input
  - a stack is used to keep track of the yet unresolved part of the path taken by the parse
- The goal is to find and reduce handles.
- To keep track of how far we have gotten through the growth of a handle, we use items.

Constructing LR parsing tables: items

- An item is a production with a place marker inserted somewhere in its right-hand-side, indicating the current parsing state in the production, e.g. $E \rightarrow E+T$ has 4 items:

$$[E \rightarrow \bullet E+T]$$
$$[E \rightarrow E\bullet+T]$$
$$[E \rightarrow E+\bullet T]$$
$$[E \rightarrow E+T\bullet]$$

- Symbols to the left of the dot are already on the stack, those to the right are expected to be coming in input

Constructing LR parsing tables: items

- Initial item: an item beginning with a dot
- Completed item: an item ending with a dot. These are candidates for reduction. When we have a completed item, a handle has been recognized. We can then reduce this handle to its corresponding left hand side non-terminal.

- Moving the dot one symbol further in the right hand side corresponds to a state transition in the DFA.

- Starting from state 0 (starting symbol), and computing the closure (i.e. all different possible expansions using the grammar rules) we find all possible state transitions using items (i.e. the item set of a state).
- Then for each group of items with the same symbol to the right of the dot in the item set, we apply a transition to another state (i.e. moving the dot one step). If the state is new, we compute its closure and further transitions.
- This proceeds until no more new states can be generated and all handles have been processed.

Constructing the item sets

State 0 : V[ε]	: [S→•E]	state 1 : V[E]
closure(S→•E)	: [E→•E+T]	state 1
	[E→•T]	state 2 : V[T]
	[T→•T*F]	state 2
	[T→•F]	state 3 : V[F]
	[F→•(E)]	state 4 : V[(]
	[F→•id]	state 5 : V[id]
State 1 : V[E]	: [S→E•]	accept
	[E→E•+T]	state 6 : V[E+]
State 2 : V[T]	: [E →T•]	handle
	[T→T•*F]	state 7 : V[T*]
State 3 : V[F]	: [T→F•]	handle

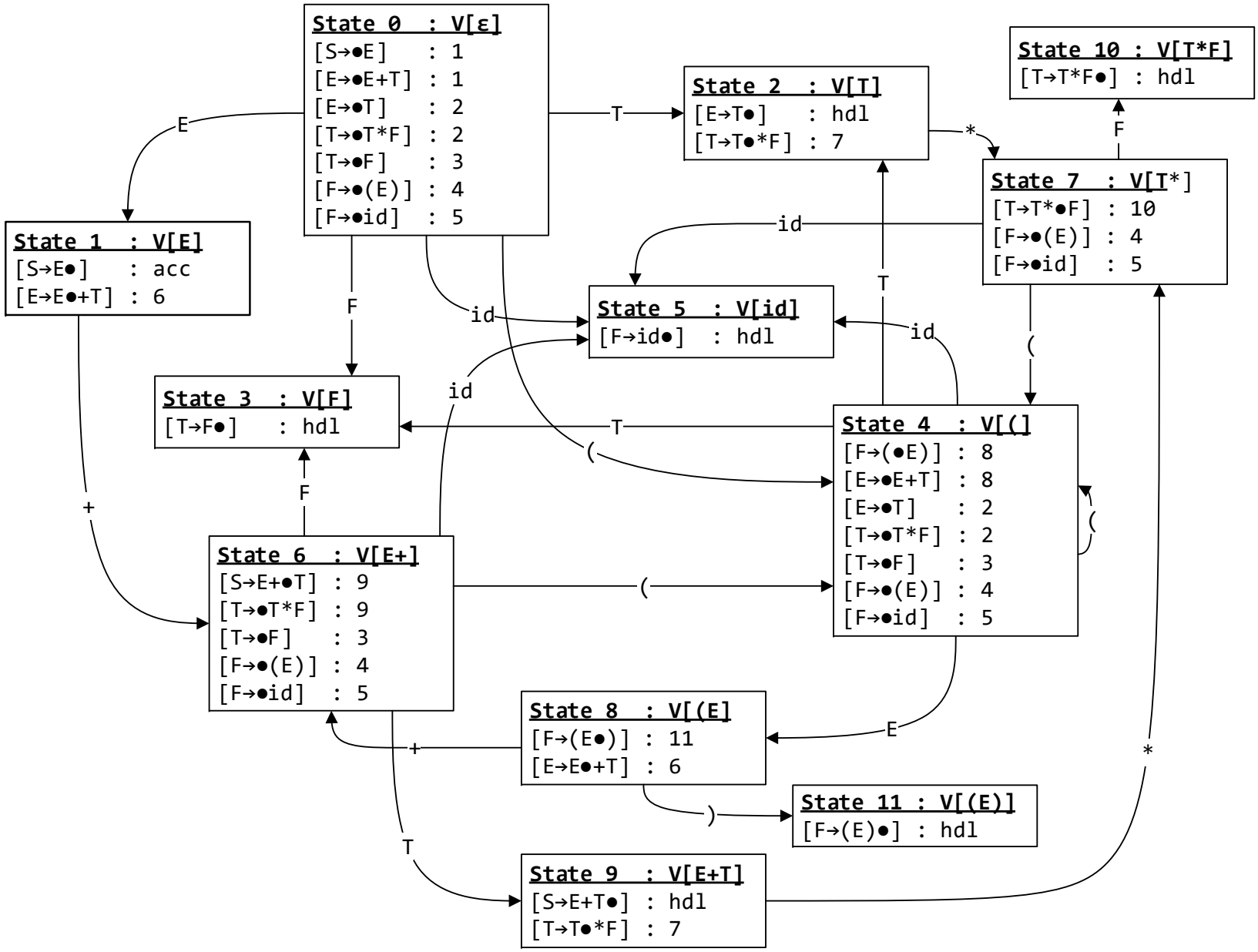
Constructing the item sets

State 4 : V[( ]	:	[F→(•E)]	state 8 : V[(E]
closure(F→(•E))	:	[E→•E+T]	state 8
		[E→•T]	state 2
		[T→•T*F]	state 2
		[T→•F]	state 3
		[F→•(E)]	state 4
		[F→•id]	state 5
State 5 : V[id]	:	[F→id•]	handle
State 6 : V[E+]	:	[E→ E+•T]	state 9 : V[E+T]
closure(E→ E+•T)	:	[T→•T*F]	state 9
		[T→•F]	state 3
		[F→•(E)]	state 4
		[F→•id]	state 5

Constructing the item sets

```
State 7 : V[T*]   : [F→T*•F]      state 10 : V[T*F]
closure(F→T*•F)   : [F→•(E)]      state 4
                    [F→•id]       state 5
State 8 : V[(E]   : [F→(E•)]      state 11 : V[(E)]
                    [E→ E•+T]     state 6
State 9 : V[E+T]  : [E→ E+T•]     handle
                    [T→T•*F]      state 7
State 10 : V[T*F]: [F→T*F•]       handle
State 11 : V[(E)]: [F→ (E)•]      handle
```

Resulting DFA



Filling the table

(1) goto table

```
for each state
   for each transition A→ α•β... that ( is not a completed item AND
                                       has a non-terminal symbol after the dot )
     put state(V[αβ]) in column β
```

(2) action table : shift actions

```
for each state
   for each transition A→ α•β... that ( is not a completed item AND
                                       has a terminal symbol after the dot )
      put a shift action state(V[αβ]) in column β
```

(3) action table : reduce actions

```
for each state
   for each transition that ( is a completed item A→ β•)
      for all elements f of FOLLOW(A)
         put a reduce n action in column f where n is the production number
```

(4) action table : accept action

```
find the completed start symbol item S→ α• and its corresponding state s
 put an accept action in TT[s,$]
```

Filling the table

state	action						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			

1	E → E + T
2	E → T
3	T → T * F
4	T → F
5	F → (E)
6	F → id

	FOLLOW
E	{+,),$}
T	{+,*,),$}
F	{+,*,),$}

References

- C.N. Fischer, R.K. Cytron, R.J. LeBlanc Jr., *Crafting a Compiler*, Adison-Wesley, 2009. Chapter 6.