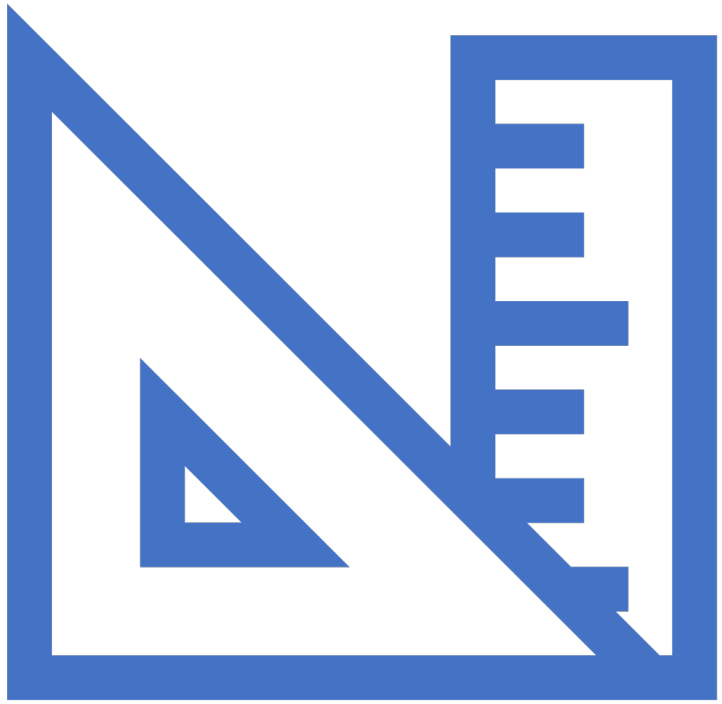


CSC 416/565: DESIGN AND CONSTRUCTION OF COMPILERS

West Chester University
Dr. Richard Burns
Fall 2023

PREVIOUS CLASSES

- Top-down parsing
  - Recursive descent, requires backtracking
  - Predictive parsing: LL(1), deterministic

TODAY

- Bottom-up Parsing

Weakness of LL(k) parsing techniques is that they must *predict* which production to use, after having seen only k tokens.

Bottom-up Parsing Overview

- Builds on ideas of top-down parsing
- More powerful than top-down
- **Postpones "reduction" decision until it has seen input tokens corresponding to the entire RHS**

Benefits of Bottom-up Parsing

- *(not as restrictive as top-down)*
- Left-factoring not necessary
- Elimination of left-recursion unnecessary (because parse tree is built from the leaves up)
  - Right-recursion is not a problem, but may lead to some inefficiencies
- Ambiguous grammars remain problematic
  - (as they should!)

Bottom-up Parsing

- Most basic bottom-up parsing technique is LR(k) parsing
  - Reading input left-to-right
  - Using a rightmost derivation
  - With k tokens of lookahead

- *Idea:* bottom-up parsing reduces a string to the start symbol by *inverting productions*

- *How?* Similar to LL(1) predictive parsing:
  - a parse table is first build based on the grammar
  - At compile time, the driver reads the input tokens, consults the parse table, and creates a parse from the bottom up

Example

Grammar:

```
E → T + E | T
T → int * T | int | ( E )
```

Observe what a bottom-up parse looks like:

- How to know which reductions to use, and when to use them?
- *Notice:* going down-to-up, a rightmost derivation is present
- Eventually reduce, and build complete tree, all the way to the start symbol
- Each step combines subtrees into larger trees

Shift / Reduce

- Shift / Reduce: the main strategy used by all bottom-up parsers
  - There are a number of LR-parsing variants: (SLR, LALR, …)
  - All use the same *driver*
  - Only differ in the generated table

Parsing Driver for “LR”-family

- Parser has a stack and input.
  - Driver reads the input and consults the parsing table.
- Parsing table contains shift and reduce entries.
- *Initially:*
  - Stack is empty
  - Parser is at the beginning of input
- Shift: move the next input token onto the top of the stack; bump the input pointer
- Reduce: use a grammar rule “reduction”
- *Example:* given rule X → ABC, and top of the stack is C B A, pop C B A (the RHS) and push X (the LHS) onto stack
- *Note:* only ever pushing nonterminals onto the stack in reductions

Example

Grammar:

```
E → T + E | T
T → int * T | int | ( E )
```

Input String:

```
int * int + int
```

LR(1) Parse Table Driver Examples

Example 1

Stack	*Input*	*Action*
1	a := 7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 id$_4$	:= 7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 id$_4$:=$_6$	7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 id$_4$:=$_6$ num$_{10}$	; b := c + (d := 5 + 6 , d) $	*reduce* $E \rightarrow$ num
1 id$_4$:=$_6$ E_{11}	; b := c + (d := 5 + 6 , d) $	*reduce* $S \rightarrow$ id:=E
1 S_2	; b := c + (d := 5 + 6 , d) $	*shift*
1 S_2 ;$_3$	b := c + (d := 5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id$_4$	:= c + (d := 5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$	c + (d := 5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$ id$_{20}$	+ (d := 5 + 6 , d) $	*reduce* $E \rightarrow$ id
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11}	+ (d := 5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$	(d := 5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8	d := 5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$	:= 5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$:=$_6$	5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$:=$_6$ num$_{10}$	+ 6 , d) $	*reduce* $E \rightarrow$ num
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$:=$_6$ E_{11}	+ 6 , d) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$:=$_6$ E_{11} +$_{16}$	6 , d) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$:=$_6$ E_{11} +$_{16}$ num$_{10}$	, d) $	*reduce* $E \rightarrow$ num
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$:=$_6$ E_{11} +$_{16}$ E_{17}	, d) $	*reduce* $E \rightarrow E + E$
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$:=$_6$ E_{11}	, d) $	*reduce* $S \rightarrow$ id:=E
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 S_{12}	, d) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 S_{12} ,$_{18}$	d) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 S_{12} ,$_{18}$ id$_{20}$	) $	*reduce* $E \rightarrow$ id
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 S_{12} ,$_{18}$ E_{21}	) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 S_{12} ,$_{18}$ E_{21})$_{22}$	$	*reduce* $E \rightarrow (S, E)$
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ E_{17}	$	*reduce* $E \rightarrow E + E$
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11}	$	*reduce* $S \rightarrow$ id:=E
1 S_2 ;$_3$ S_5	$	*reduce* $S \rightarrow S; S$
1 S_2	$	*accept*

	id	num	print	;	,	+	:=	(	)	$	S	E	L
1	s4		s7								g2		
2				s3						a			
3	s4		s7								g5		
4							s6						
5				r1	r1					r1			
6	s20	s10						s8				g11	
7								s9					
8	s4		s7								g12		
9	s20	s10						s8				g15	g14
10				r5	r5	r5			r5	r5			
11				r2	r2	s16				r2			
12				s3	s18								
13				r3	r3					r3			
14					s19				s13				
15					r8				r8				
16	s20	s10						s8				g17	
17				r6	r6	s16			r6	r6			
18	s20	s10						s8				g21	
19	s20	s10						s8				g23	
20				r4	r4	r4			r4	r4			
21									s22				
22				r7	r7	r7			r7	r7			
23					r9	s16			r9				

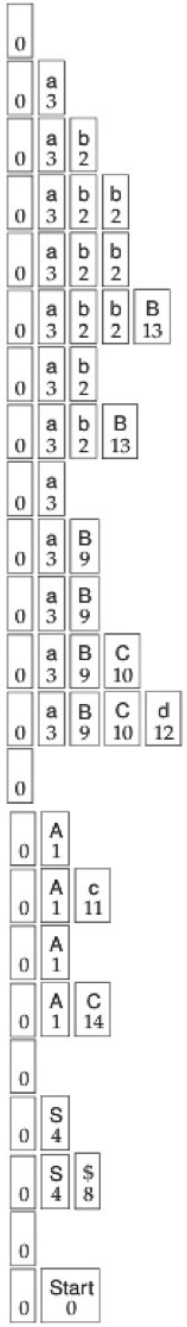
1. $S \rightarrow S\ ;\ S$
2. $S \rightarrow \text{id} := E$
3. $S \rightarrow \text{print}\ (\ L\ )$
4. $E \rightarrow \text{id}$
5. $E \rightarrow \text{num}$
6. $E \rightarrow E + E$
7. $E \rightarrow (S\ ,\ E\ )$
8. $L \rightarrow E$
9. $L \rightarrow L\ ,\ E$

Example 2

State	a	b	c	d	q	$	Start	S	A	B	C	Q
0	3	2	8		8	8	accept	4	1	5		
1			11			4					14	
2		2	8	8	8	8				13		
3		2	8	8						9		
4						8						
5			10		7	10						6
6			6			6						
7			9			9						
8						1						
9			11	4							10	
10				12								
11				3		3						
12			5			5						
13			7	7	7	7						
14						2						

1 Start → S $
2 S → A C
3 C → c
4 | λ
5 A → a B C d
6 | B Q
7 B → b B
8 | λ
9 Q → q
10 | λ

Rule	Derivation
1	Start $\Rightarrow_{rm}$ S $
2	$\Rightarrow_{rm}$ A C $
3	$\Rightarrow_{rm}$ A c $
5	$\Rightarrow_{rm}$ a B C d c $
4	$\Rightarrow_{rm}$ a B d c $
7	$\Rightarrow_{rm}$ a b B d c $
7	$\Rightarrow_{rm}$ a b b B d c $
8	$\Rightarrow_{rm}$ a b b d c $



Initial Configuration	a b b d c $
shift a	b b d c $
shift b	b d c $
shift b	d c $
Reduce λ to B	B d c $
shift B	d c $
Reduce b B to B	B d c $
shift B	d c $
Reduce b B to B	B d c $
shift B	d c $
Reduce λ to C	C d c $
shift C	d c $
shift d	c $
Reduce a B C d to A	A c $
shift A	c $
shift c	$
Reduce c to C	C $
shift C	$
Reduce A C to S	S $
shift S	$
shift $	$
Reduce S $ to Start	Start $
shift Start	$
Accept	

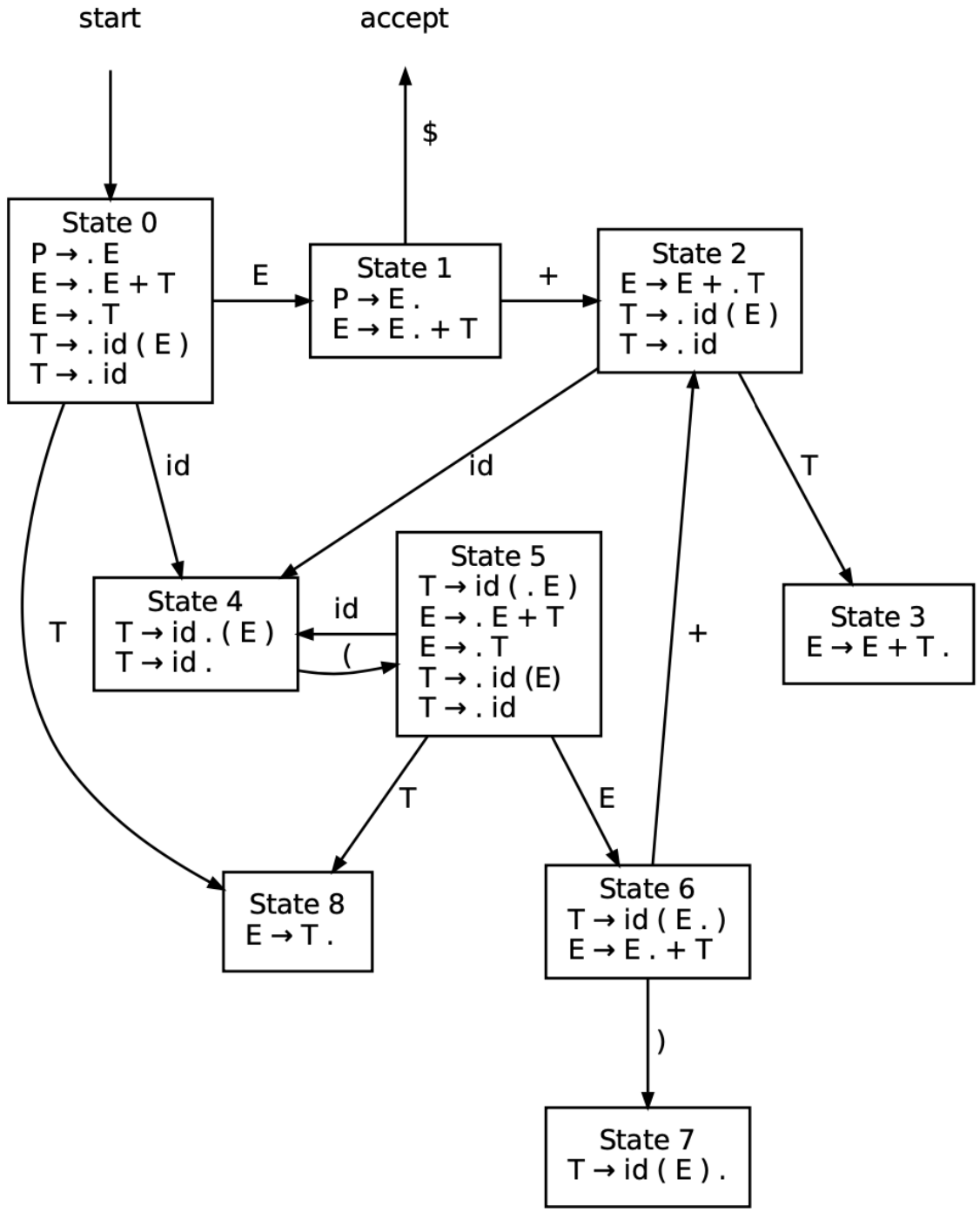
Example 3

State	GOTO		ACTION				
	E	T	id	(	)	+	$
0	G1	G8	S4				
1						S2	R1
2		G3	S4				
3					R2	R2	R2
4				S5	R5	R5	R5
5	G6	G8	S4				
6					S7	S2	
7					R4	R4	R4
8					R3	R3	R3

1. P → E
2. E → E + T
3. E → T
4. T → id (E)
5. T → id

Stack	**Symbols**	**Input**	**Action**
0		id (id + id) $	shift 4
0 4	id	(id + id) $	shift 5
0 4 5	id (	id + id) $	shift 4
0 4 5 4	id (id	+ id) $	reduce T → id
0 4 5 8	id (T	+ id) $	reduce E → T
0 4 5 6	id (E	+ id) $	shift 2
0 4 5 6 2	id (E +	id) $	shift 4
0 4 5 6 2 4	id (E + id	) $	reduce T → id
0 4 5 6 2 3	id (E + T	) $	reduce E → E + T
0 4 5 6	id (E	) $	shift 7
0 4 5 6 7	id (E)	$	reduce T → id(E)
0 8	T	$	reduce E → T
0 1	E	$	accept

LR(0) Automaton for the Previous Slide



LR(0) Automaton

- LR(0) automaton represents all possible rules that are currently under consideration by a shift-reduce parser
- Each box represents a *state* in the machine, connected by transitions for both terminals and non-terminals in the grammar
- State contains multiple *items* augmented by a dot (.) to indicate parser's *current position*

Example:

E → E . + T

Indicates E is currently on the stack and + T is a possible next sequence of tokens

Construction of Automaton

Grammar:

1. P → E
2. E → E + T
3. E → T
4. T → id (E)
5. T → id

- Start production, with a dot (.) at the beginning of the RHS, becomes State 0 (first item is sometimes called the *kernel* of the state)
- Recursively compute the *closure* of this state and add as additional *items* in the same state until no more can be added
  - *Closure:* For each item in the state with a non-terminal X immediately to the right of the dot (.), add all rules in the grammar that have X as the LHS

Kernel of State 0

P → . E

Closure of State 0

P → . E
E → . E + T
E → . T
T → . id (E)
T → . id

Construction of Automaton (cont.)

Closure of State 0

$P \rightarrow . E$
$E \rightarrow . E + T$
$E \rightarrow . T$
$T \rightarrow . id (E)$
$T \rightarrow . id$

Grammar:

1. $P \rightarrow E$
2. $E \rightarrow E + T$
3. $E \rightarrow T$
4. $T \rightarrow id (E)$
5. $T \rightarrow id$

- All terminals and non-terminals to the right of the dot (.) are possible outgoing transitions
- If the automaton takes that transition it makes a *new state* containing the matching items, with the dot (.) moved one place to the right

Transition on E:

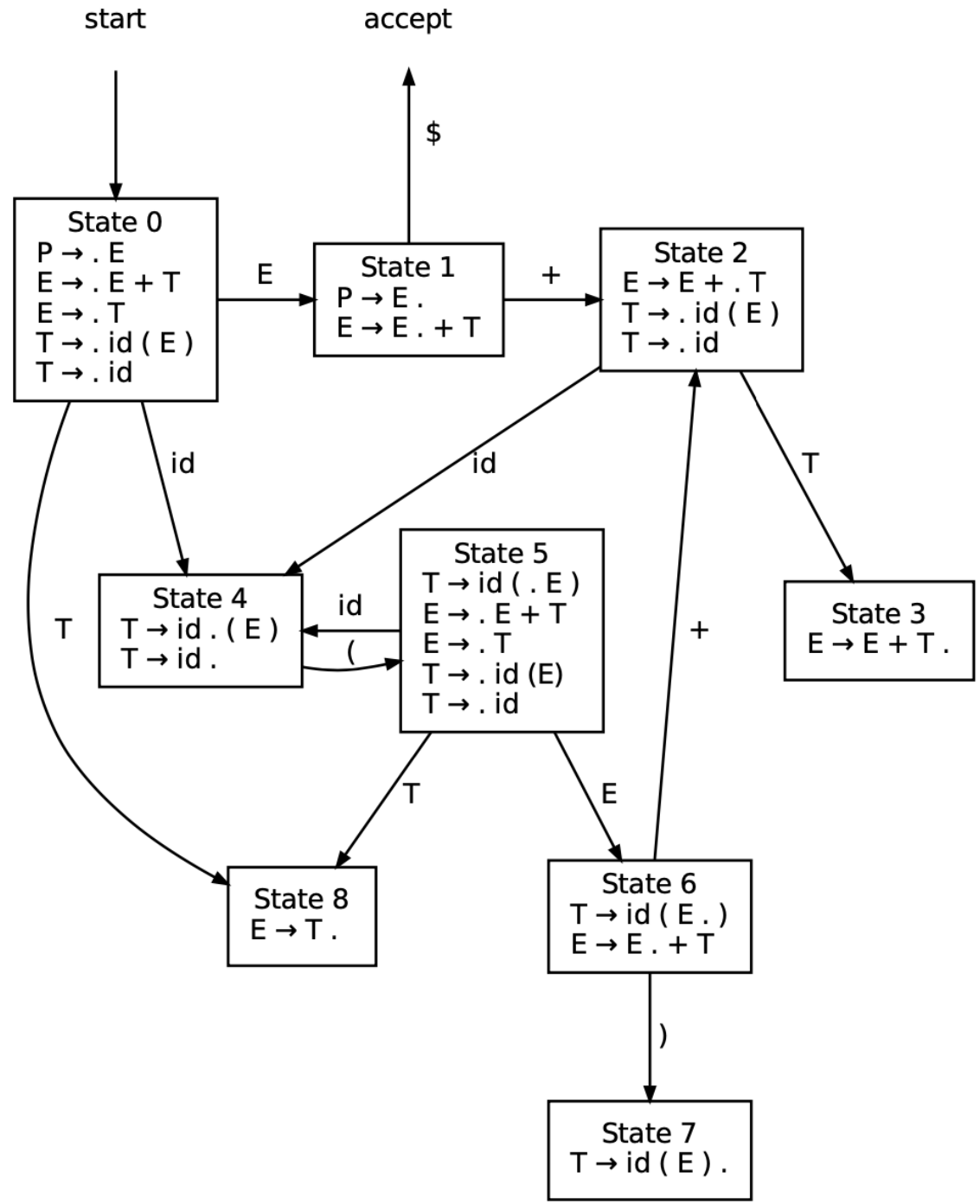
$P \rightarrow E .$
$E \rightarrow E . + T$

Transition on T:

$E \rightarrow T .$

Transition on id:

$T \rightarrow id . (E)$
$T \rightarrow id .$


start
accept
$
State 0
P → . E
E → . E + T
E → . T
T → . id (E)
T → . id
E
State 1
P → E .
E → E . + T
+
State 2
E → E + . T
T → . id (E)
T → . id
id
id
T
T
State 4
T → id . (E)
T → id .
id
(
State 5
T → id (. E)
E → . E + T
E → . T
T → . id (E)
T → . id
+
State 3
E → E + T .
T
E
State 8
E → T .
State 6
T → id (E .)
E → E . + T
)
State 7
T → id (E) .

Shift-Reduce Conflict:

$$T \rightarrow id\ .\ (\ E\ )$$
$$T \rightarrow id\ .$$

LR(0) Automaton

Reduce-Reduce Conflict:

$$S \rightarrow id\ (\ E\ )\ .$$
$$E \rightarrow id\ (\ E\ )\ .$$

- LR(0) automata enumerate choices available at any step of the parse
- A state containing an item with a dot (.) at the end indicates a possible reduction
- A transition on a terminal that moves the dot (.) one position to the right indicates a possible shift
- Two types of conflicts can appear in an LR grammar:
  1. **shift-reduce conflict** indicates a choice between a shift action and a reduce action
  2. **reduce-reduce conflict** indicates that two distinct rules have been completely matched

SLR Parse Table Creation.

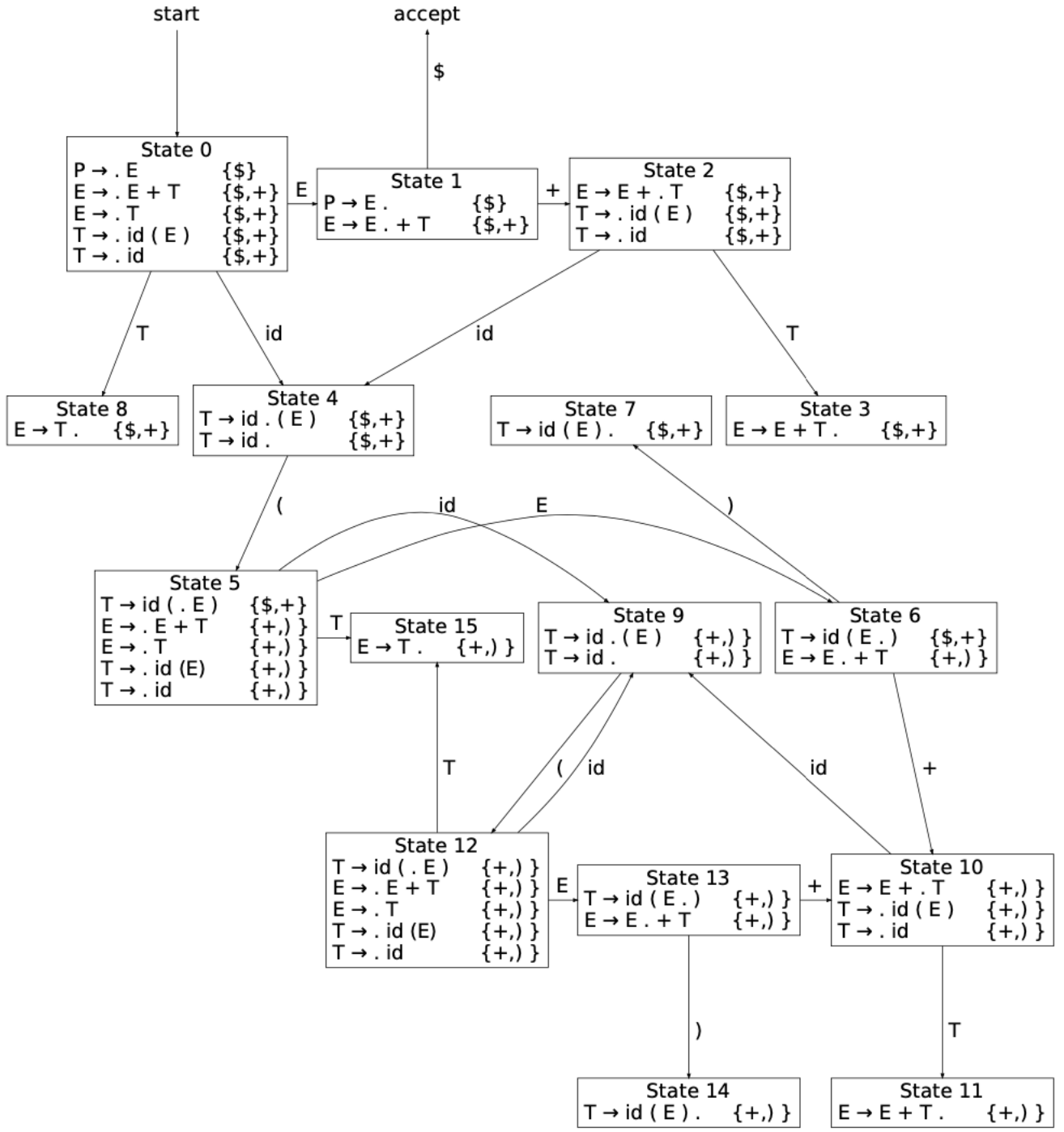
Given a grammar G and corresponding LR(0) automaton,
create tables ACTION$[s, a]$ and GOTO$[s, A]$ for all states s,
terminals a, and non-terminals A in G.

For each state s:
- For each item like A $\rightarrow \alpha$. a β
  - ACTION$[s, a]$ = **shift** to state t according to the LR(0) automaton.
- For each item like A $\rightarrow \alpha$. B β
  - GOTO$[s, B]$ = **goto** state t according to the LR(0) automaton.
- For each item like A $\rightarrow \alpha$.
  - For each terminal a in FOLLOW(A):
    - ACTION$[s, a]$ = **reduce** by rule A $\rightarrow \alpha$

All remaining states are considered error states.

Concluding Thoughts

- SLR is a subset of LR(1) and not all LR(1) grammars are SLR
- In practice, a LR(1) automaton is used with an algorithm known as Lookahead LR (LALR)
  - All items in the automaton are augmented with a lookahead of the set of tokens that could potentially follow it



After the Break

Parsing Tools and an introduction to Programming Assignment #3.

Parsing Tools

Task of constructing a parser is simple enough to be automated.

JavaCC – LL(k) Parser Generator

- JavaCC generates a top-down, recursive-descent parser.

Productions written in the form:

```
void Assignment() : { } { Identifier() "=" Expression() ";" }
```

to capture the rule:

```
Assignment → Identifier = Expression;
```

Perhaps better written, using tokens for terminals, as:

```
void Assignment() : { } { Identifier() <EQUALS> Expression() <SEMICOLON> }
```

JavaCC will detect left recursion, which must be eliminated.

Example:

```
StmList → Stm | StmList ; Stm
```

```
void StmList() :
{ }
{ Stm()
  | StmList() <SEMICOLON> Stm()
}
```

JavaCC syntax:

1. curly braces for `{Java Declarations}`
   - Will use these in next part of the course
2. `{Rule Definitions}` used to specify the RHS of productions

Would produce the error:

`Left Recursion detected:" "StmList ... → StmList ...`

Example

JavaCC fragment:

```
void S(): {}
{
    "a" "b" "c"
    |
    "a" "d" "c"
}
```

- Is this grammar LL(1)?
  - It could be after left-factoring.
- Is this grammar LL(2)?
  - Probably yes
  - But LL(1) tables are much smaller and desirable
  - There are no parsing tables in recursive descent anyway!

Lookahead

JavaCC Solution:

- `LOOKAHEAD` directive will allow JavaCC to use more than one token for lookahead
- JavaCC will look at the next two symbols before deciding which rule to use
- Lookahead directives are placed at "choice points"
  - Places in the grammar where there is more than one possible rule that can match

```
void S(): {}
{
    "a" "b" "c"
    |
    "a" "d" "c"
}
```

```
void S(): {}
{
    LOOKAHEAD(2) "a" "b" "c"
    |
    "a" "d" "c"
}
```

Example 2

Grammar:

S → a(bc|bd)

One solution using lookahead of only 2:

```
void S(): {}
{
    "a" (LOOKAHEAD(2)("b""c")|("b""d"))
}
```

NOTE: this wouldn't work

```
void S(): {}
{
    LOOKAHEAD(2) "a" (("b""c")|("b""d"))
}
```

Alternative: could have also factored `b` out and avoided the lookahead

Other Notes

- JavaCC can provide a parser for some grammars that are not LL(1)
- the parser that is produced is not guaranteed to correctly parse the language described by the grammar
- a warning will be issued when JavaCC is run on a non-LL(1) grammar
- for a non-LL(1) grammar, the rule that applies *first* will be used
- `LOOKAHEAD` suppresses warning messages
  - JavaCC assumes that you know what you are doing

Error Recovery

- What should happen when the parser encounters an error?
- Not only report the *first* error, but ideally recognize *all* of the errors in a program.
- *Idea:* provide additional grammar rules for error capturing and how far ahead to "skip" to get past error, and to try and resume parsing

Original Grammar:

```
exp → Id
exp → exp + exp
exp → (exps)
exps → exp
exps → exps ; exp
```

New Grammar Rule for errors:

```
exp → (error)
exps → error ; exp
```

Idea: skip to the next `;` or `)`

- keep discarding input symbols until a lookahead is reached.
  - neither JavaCC nor SableCC supports the error symbol (yacc does)

Shallow Error Recovery

Grammar Rule:

```
Stm → IfStm | WhileStm
```

```
void Stm() :
{}
{
    IfStm()
|
    WhileStm()
|
    error_skipto(SEMICOLON)
}
```

- will catch if next token is not <IF> or <WHILE> keyword
- Need to manually define a procedure that will keep skipping tokens until a ; and then try to resume

Deep Error Recovery

```
void Stm() :
{}
{
    try {
        (
            IfStm()
        |
            WhileStm()
        )
    } catch (ParserException e) {}
        error_skipto(SEMICOLON);
    }
}
```

- will catch deeper errors
- one advantage of JavaCC (and recursive descent parsers) is that they know which production they are in when the error happened
  - e.g. was in an `IfStm()`

Global Error Repair

- finds the smallest set of insertions and deletions that would turn the source string into a syntactically correct string
- *one algorithm:* "Burke-Fisher error repair”
  - tries *every possible* single-token insertion, deletion, or replacement, at every point no earlier than K tokens before the point where the parser reported the error
  - Example: if K=10, if parsing engine gets stuck at the 50th token, try *every* possible repair between 40th and 50th token.
  - correction that allows the parser to parse furthest *beyond* the original reported error is taken as best error repair.
- in general, repairs that carry parser 4 tokens beyond the original error are "good enough"
- benefits of technique:
  1. grammar is not modified with error productions
  2. parsing table is not modified
- Burke-Fisher is not implemented in JavaCC, but that would be an interesting project!