Lecture 14 – Recap

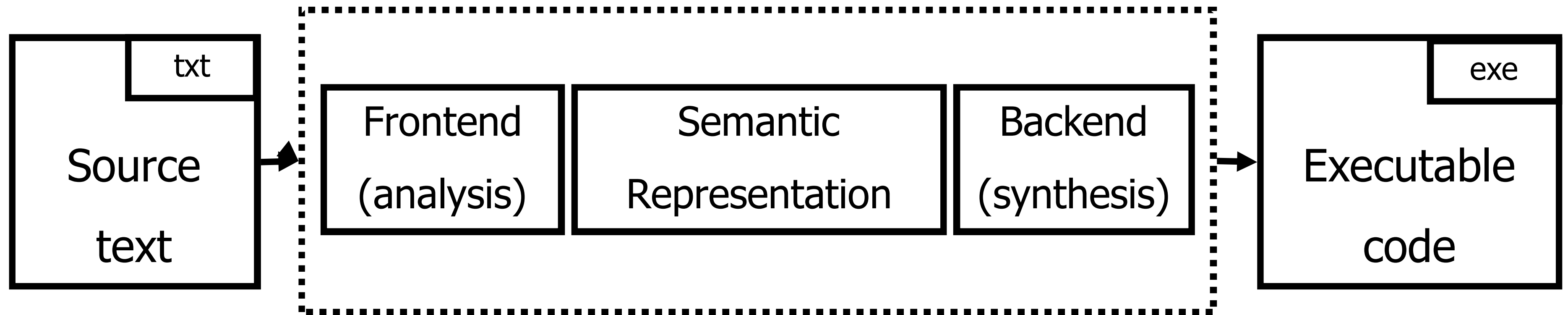
THEORY OF COMPILATION

Eran Yahav

Thanks to **Ohad Shacham** (TAU) for some of the slides in this lecture

www.cs.technion.ac.il/~yahave/tocs2011/compilers-lec14.pptx

Generic compiler structure



Lexical Analysis

- Input
  - program text (file)
- Output
  - sequence of tokens

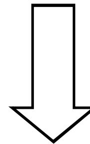
- Read input file
- Identify language keywords
- Count line numbers
- Remove whitespaces
- Report illegal symbols

Lexical Analysis

```
class Hello {
  boolean state;
  static void main(string[] args) {
    Hello h = new Hello();
    boolean s = h.rise();
    Library.printb(s);
    h.setState(false);
  }
  boolean rise() {
    boolean oldState = state;
    state = true;
    return oldState;
  }
  void setState(boolean newState) {
    state = newState;
  }
}
```

Issues in lexical analysis

- Language changes
  - New keywords
  - New operators
  - New meta-language features, e.g., annotations



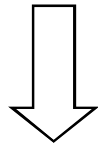
CLASS,CLASS_ID(Hello),LB,BOOLEAN,ID(state),SEMI …

Parsing

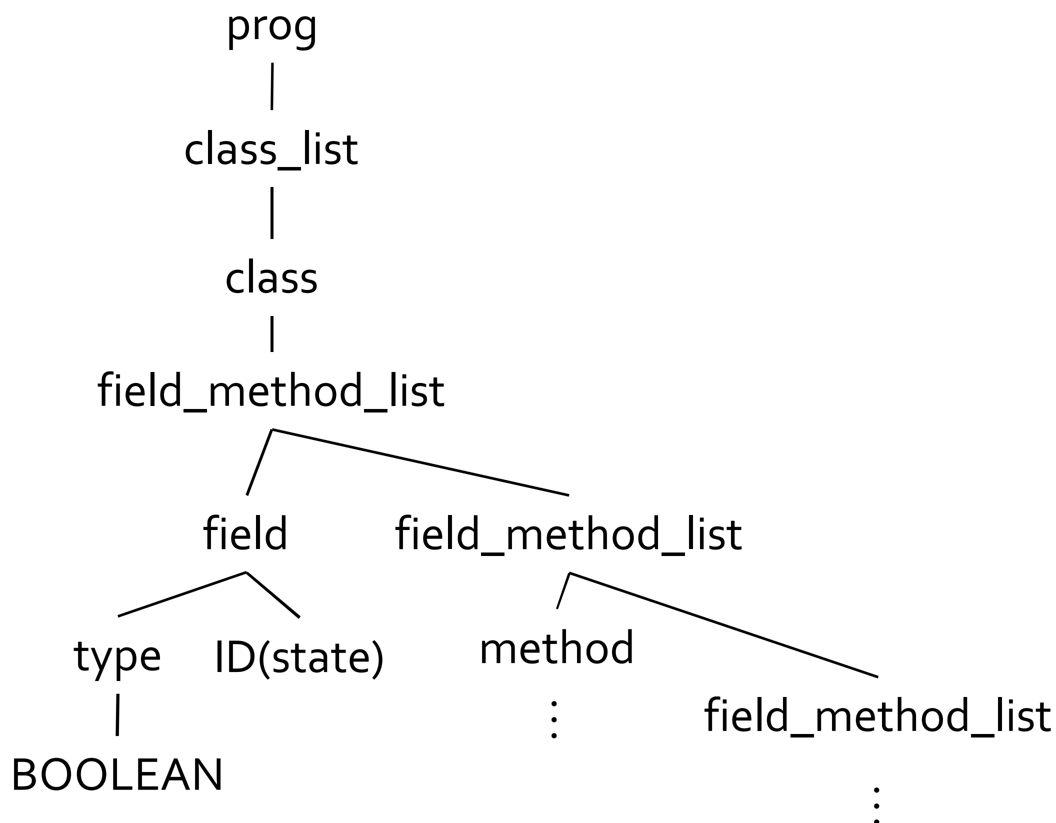
- Input
  - A context free grammar
  - A stream of tokens
- Output
  - An abstract syntax tree or error

Parsing and AST

`CLASS,CLASS_ID(Hello),LB,BOOLEAN,ID(state),SEMI …`



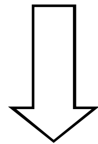
parser uses stream of tokens and generate derivation tree



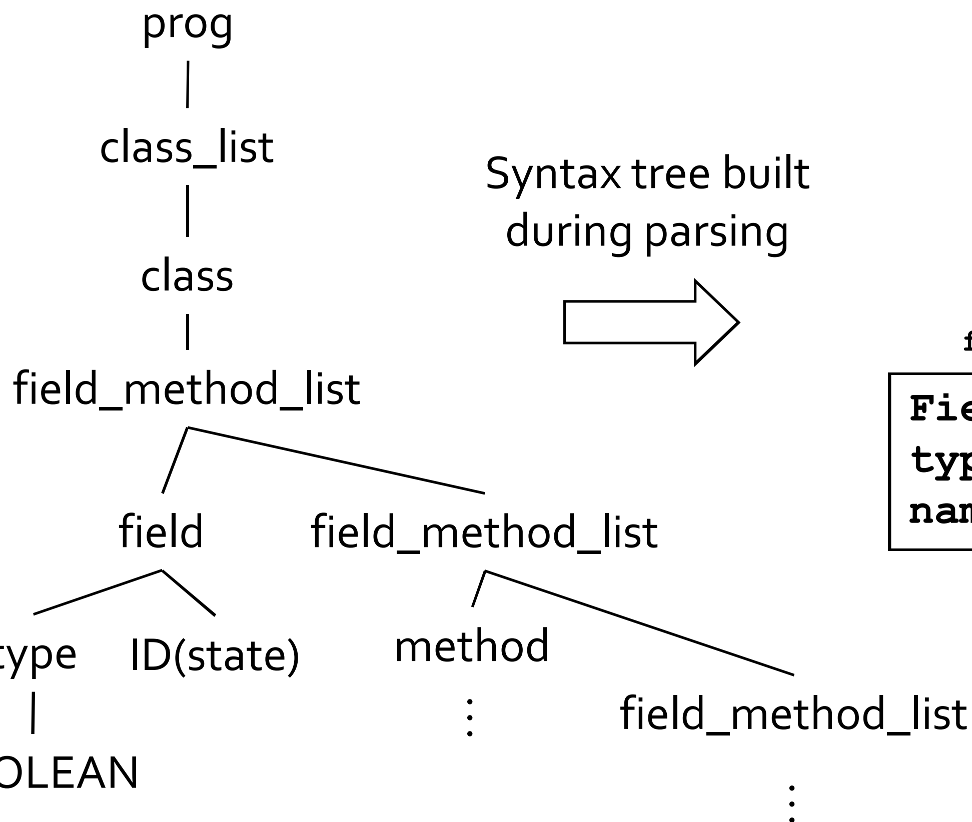
- Grammars: LL(1), LR(0), SLR(1), LALR(1), LR(1)
- Building parsers
  - Transition diagram
  - Parse table
  - Running automaton
- Conflict resolution
- Write LR grammar for a language
- Ambiguity

Parsing and AST

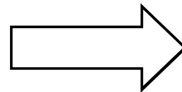
`CLASS,CLASS_ID(Hello),LB,BOOLEAN,ID(state),SEMI …`



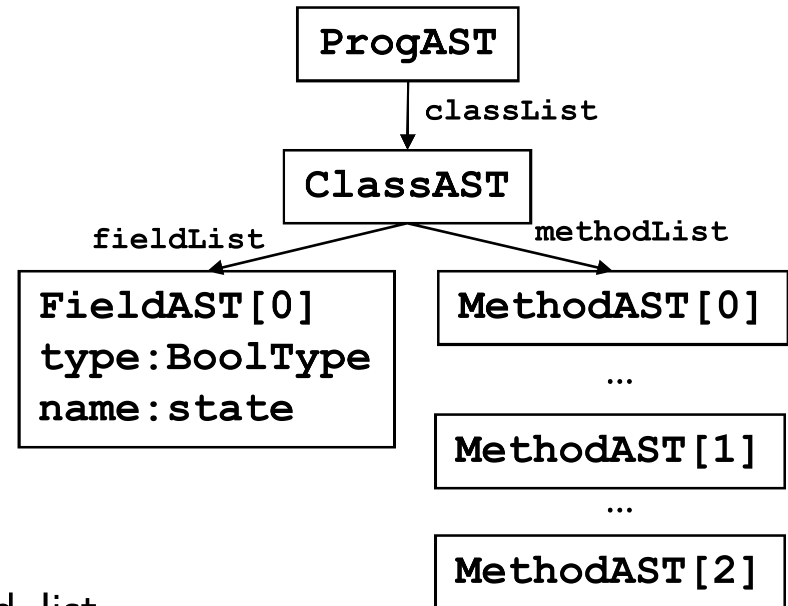
parser uses stream of token and generate derivation tree



Syntax tree built during parsing



- Should know difference between derivation tree and AST
- Know how to build AST from input



```
FieldsOrMethods  ::=

Field:field FieldsOrMethods:next
{: RESULT = next;
  RESULT. addField(field); :}
|
 Method:method FieldsOrMethods:next
 {: RESULT = next;
   RESULT.addMethod(method); :}
|
/* empty */
{: RESULT = new FieldsMethods(); :};
```

Typical Questions

- Build an LR grammar for the language
- Is the following grammar in LR(0), SLR(1), LALR(1), LR(1)
- Build a parser for the grammar
- Run an input string using your parser

Q1: Parsing

- Is the following grammar in LR(o)?

$E \rightarrow E + T$

$E \rightarrow T$

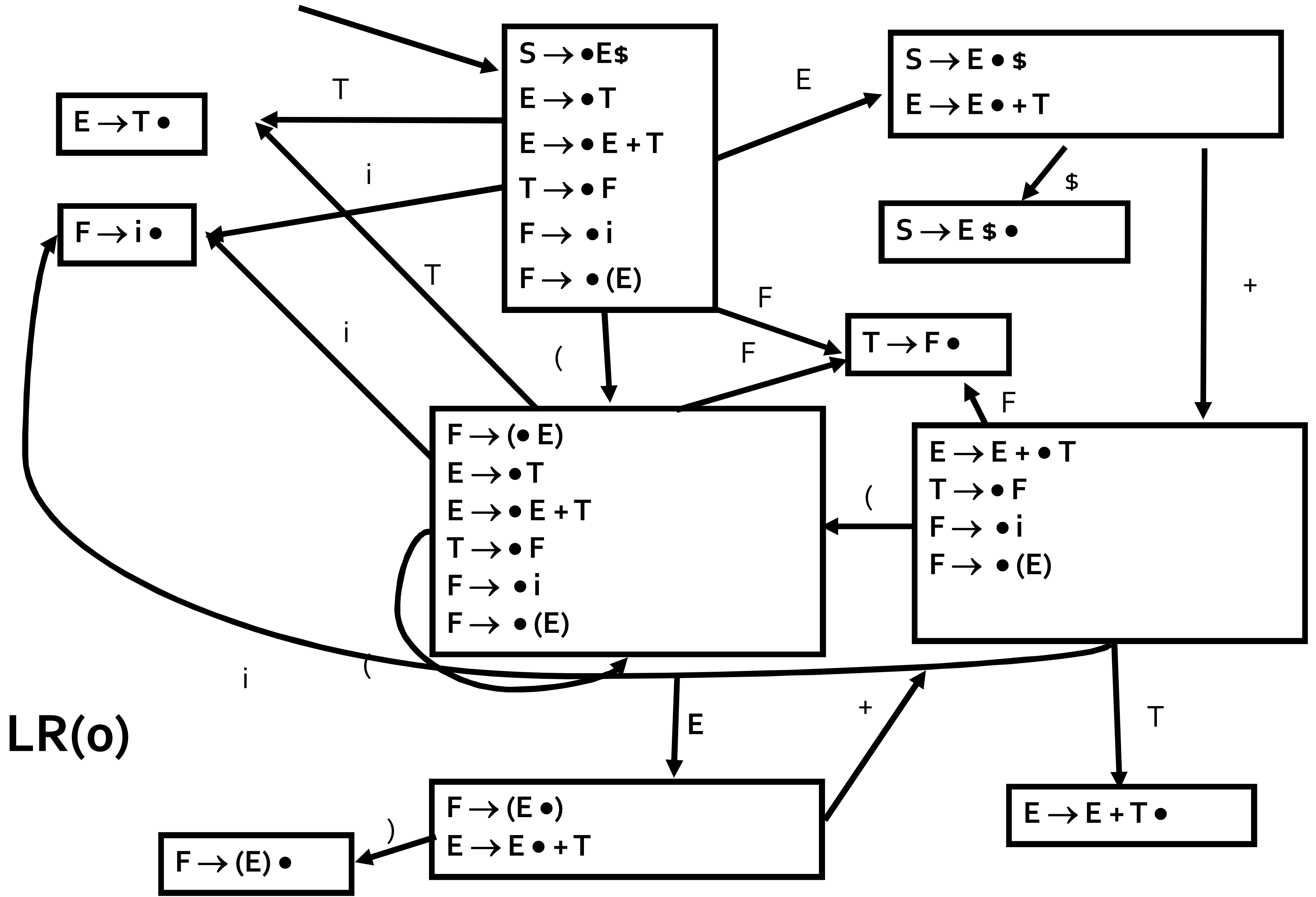
$T \rightarrow F$

$F \rightarrow id$

$F \rightarrow (E)$

Answer

- Add a production $S \rightarrow E\$$
- Construct a finite automaton
- States are set of items $A \rightarrow \alpha \bullet \beta$



LR(o)

Q2: Parsing

- Is the following grammar in LR(0)?

$E \rightarrow E + T$

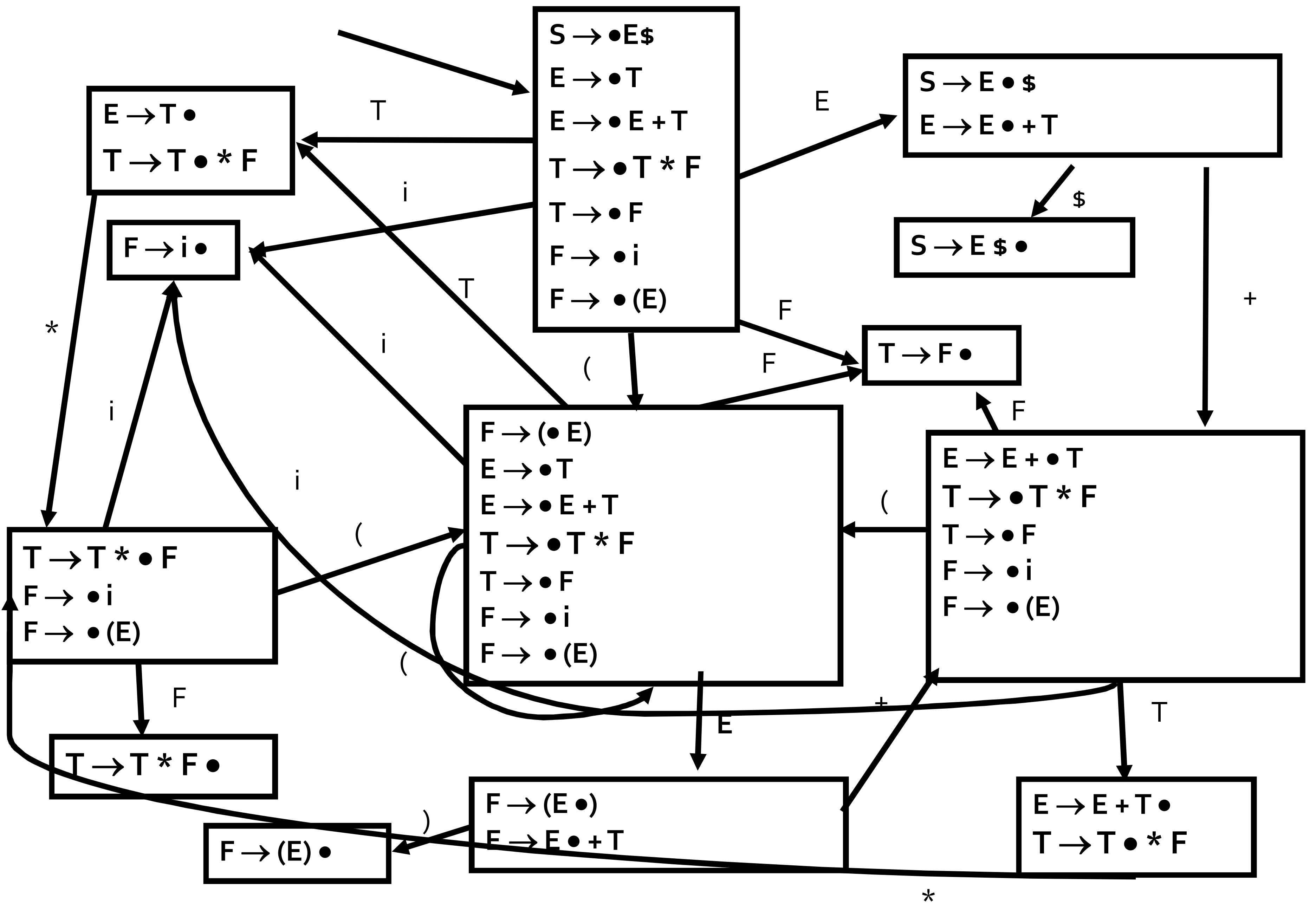
$T \rightarrow T * F$

$E \rightarrow T$

$T \rightarrow F$

$F \rightarrow id$

$F \rightarrow (E)$



S → •E$
E → •T
E → • E + T
T → • T * F
T → • F
F → •i
F → •(E)
E → T •
T → T • * F
F → i •
S → E • $
E → E • + T
S → E $ •
T → F •
F → (• E)
E → •T
E → • E + T
T → • T * F
T → • F
F → •i
F → •(E)
E → E + • T
T → • T * F
T → • F
F → •i
F → •(E)
T → T * • F
F → •i
F → •(E)
T → T * F •
F → (E) •
F → (E •)
F → E • + T
E → E + T •
T → T • * F
T
E
i
$
T
F
i
(
F
+
*
F
i
i
(
(
(
F
E
+
T
)
*

Q3: Parsing

- Is the following grammar in SLR(1)?

$E \rightarrow E + T$

$T \rightarrow T * F$

$E \rightarrow T$

$T \rightarrow F$

$F \rightarrow id$

$F \rightarrow (E)$

(We already know that its not LR(0))

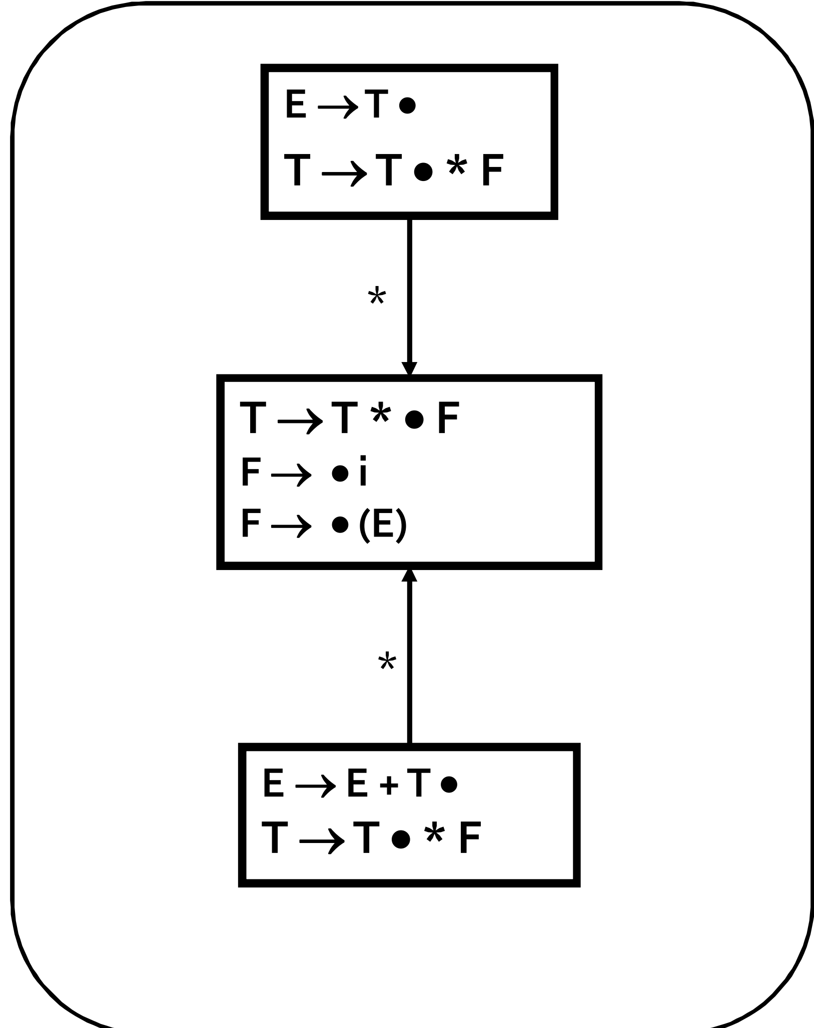
$S \rightarrow E\$$
$E \rightarrow E + T$
$T \rightarrow T * F$
$E \rightarrow T$
$T \rightarrow F$
$F \rightarrow id$
$F \rightarrow (E)$

FIRST(E) = ?
$E \rightarrow E + T$
$E \rightarrow T$
FIRST(T) = ?
$T \rightarrow T * F$
$T \rightarrow F$
FIRST(F) = { id,(}

$S \rightarrow E\$$
FOLLOW(E) = FOLLOW(E) U { $ }
$E \rightarrow E + T$
FOLLOW(E) = FOLLOW(E) U { + }
$F \rightarrow (E)$
FOLLOW(E) = FOLLOW(E) U {) }

FOLLOW(E) = {),+,$}

- compute FOLLOW sets for each non terminal
- Use the FOLLOW set to break conflicts



SLR(1)

Q4: Parsing

2.31 Can you create a top-down parser for the following grammars?

(a) $S \rightarrow$ ‘(‘ S ‘)’ | ‘)’

(b) $S \rightarrow$ ‘(‘ S ‘)’ | ε

(c) $S \rightarrow$ ‘(‘ S ‘)’ | ‘)’ | ε

Answer

2.31 Can you create a top-down parser for the following grammars?

(a) $S \rightarrow$ '(' S ')' | ')'

Yes – FIRST sets differ

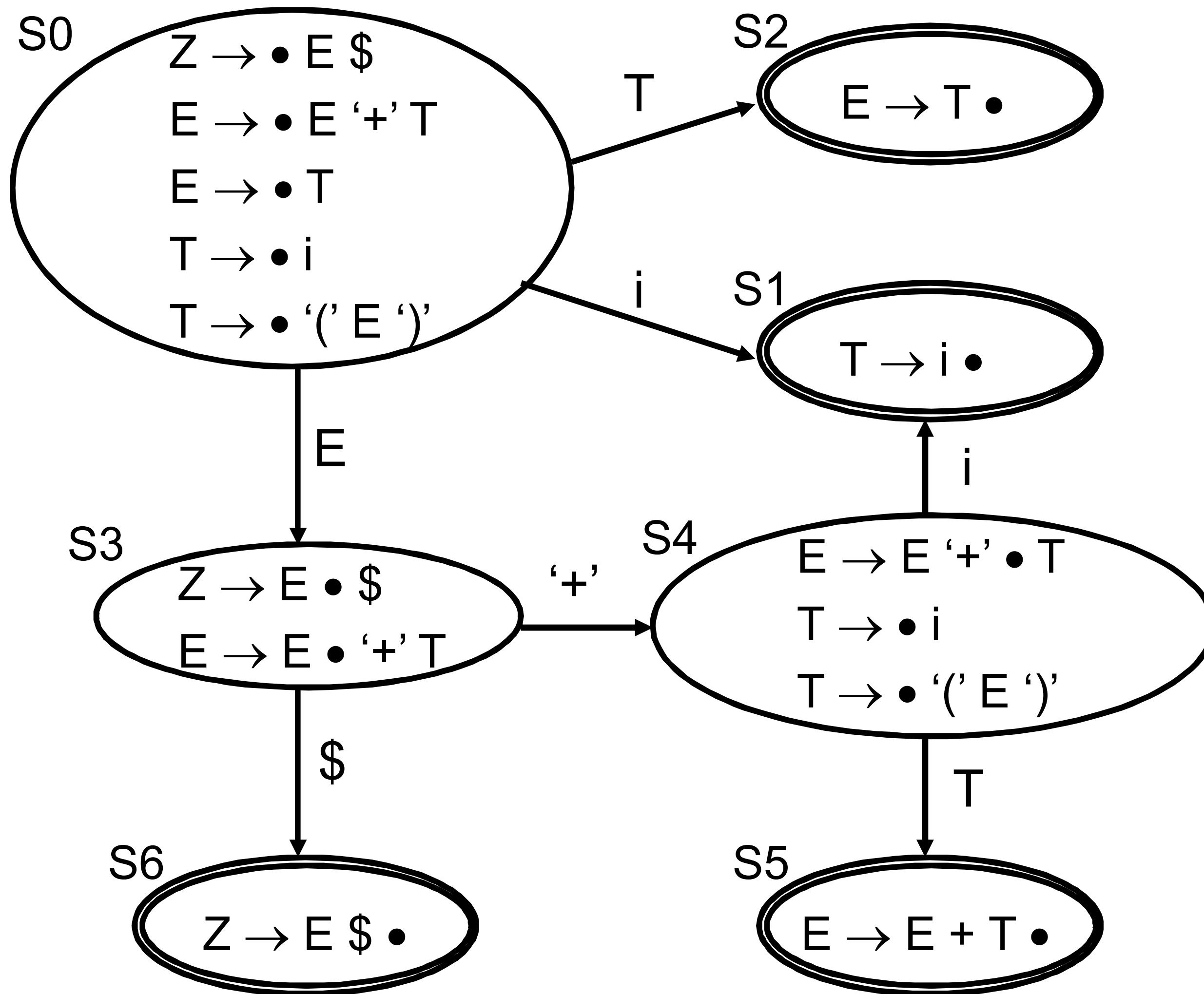
(b) $S \rightarrow$ '(' S ')' | ε

Yes – FIRST and FOLLOW set differ

(c) $S \rightarrow$ '(' S ')' | ')' | ε

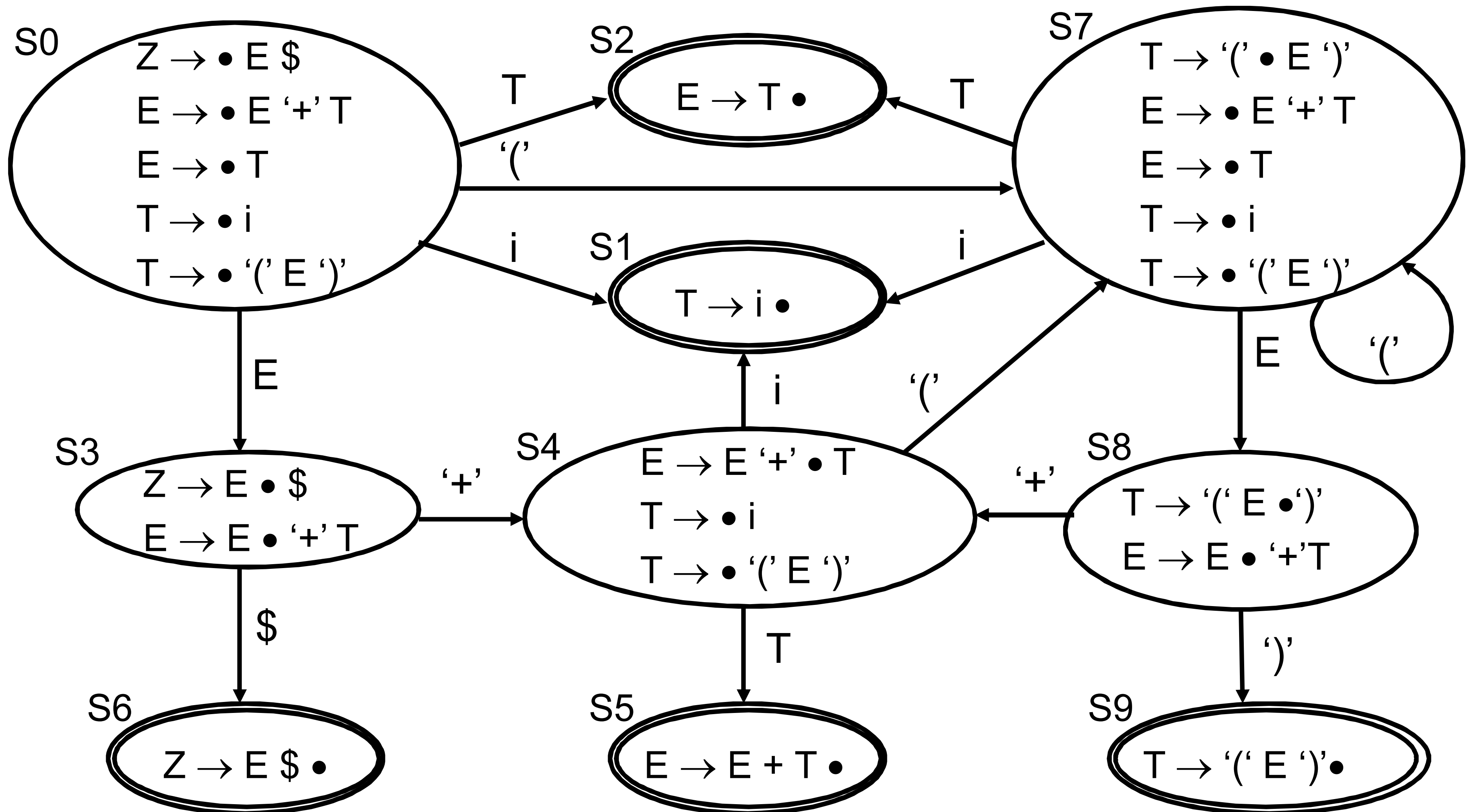
No – FIRST and FOLLOW set overlap

Transition diagram



Q5: Complete the diagram for the LR(0) automaton…

Answer Q5 (fig 2.89)



Q6: Parsing

Can you find an input the exercises all states of the automaton?

Answer Q6

The following expression exercises all states

(i) + i

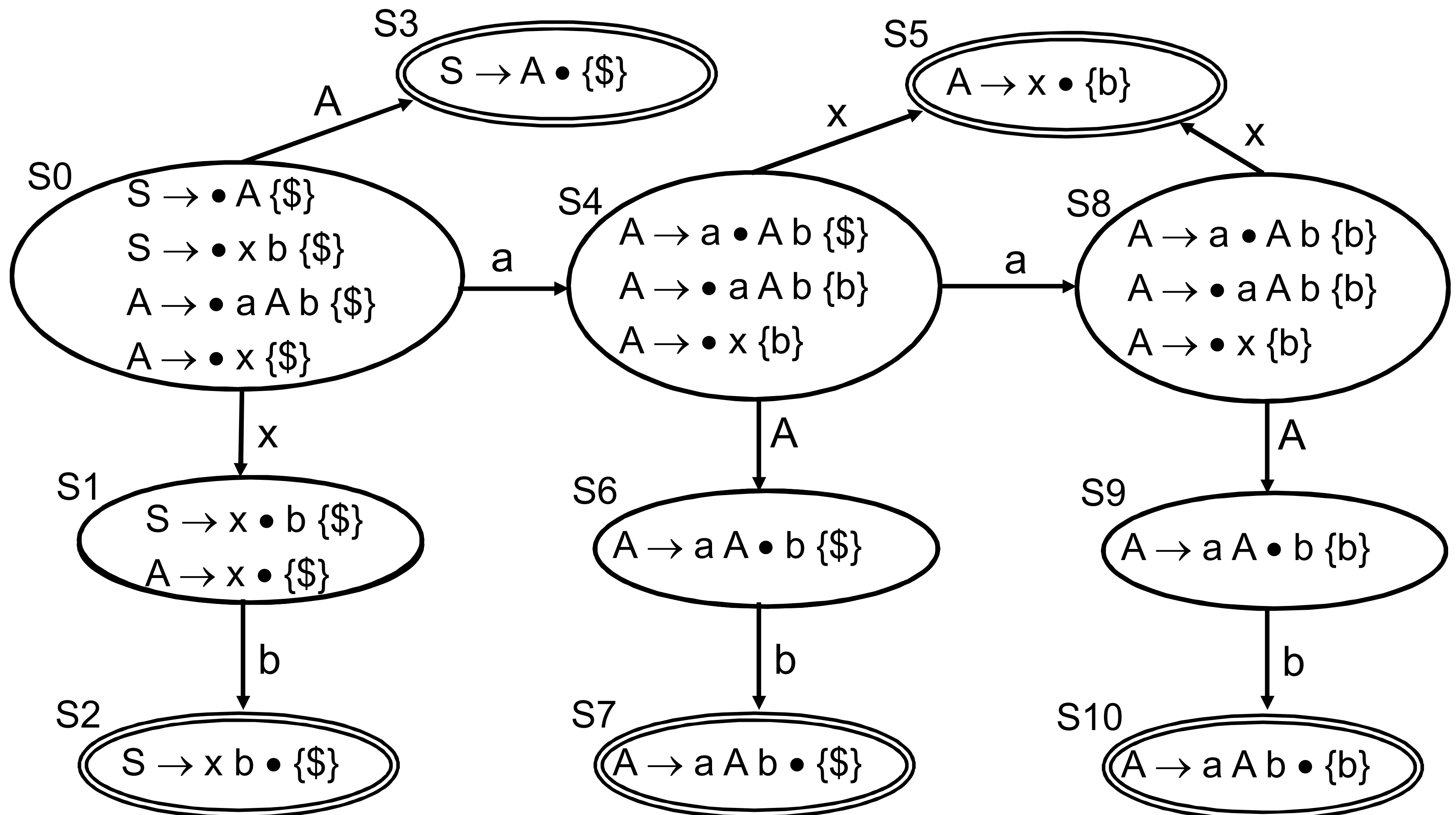
Q7

- derive the LR(1) ACTION/GOTO table for the following grammar:

$S \rightarrow A \mid x\ b$

$A \rightarrow a\ A\ b \mid x$

Answer Q7 - LR(1) automaton



Q8

2.50 Is the following grammar LR(0), LALR(1), or LR(1) ?

(a) $S \rightarrow x\ S\ x \mid y$

(b) $S \rightarrow x\ S\ x \mid x$

Answer Q8

2.50 Is the following grammar LR(0), LALR(1), or LR(1) ?

(a) $S \rightarrow x\, S\, x \mid y$

LR(0)

(b) $S \rightarrow x\, S\, x \mid x$

None! A shift-reduce conflict remains

Q9

$$S \rightarrow \varepsilon \mid a \mid \text{'('}\ S\ \text{')'} \mid \text{'('}\ S\ \text{';'}\ S\ \text{')'}$$

(a) Is this grammar LL(1) ?

(b) Is this language LL(1) ?

(c) Is this grammar LR(0) ?

(d) Is this grammar LR(1) ?

(e) Is this language regular ?

Answer Q9

$$S \rightarrow \varepsilon \mid a \mid \text{'('}\ S\ \text{')'} \mid \text{'('}\ S\ \text{';'}\ S\ \text{')'}$$

(a) Is this grammar LL(1) ?

No, last two rules have a common prefix

(b) Is this language LL(1) ?

Yes, can apply factoring to get

$$S \rightarrow \varepsilon \mid a \mid \text{'('}\ S\ T$$

$$T \rightarrow \text{')'} \mid \text{';'}\ S\ \text{')'}$$

Answer Q9

$$S \rightarrow \varepsilon \mid a \mid \text{`('}\ S\ \text{`)'} \mid \text{`('}\ S\ \text{`;'}\ S\ \text{`)'}$$

(c) Is this grammar LR(0) ?

No, there is a shift reduce conflict between the epsilon rule and the others

(d) Is this grammar LR(1) ?

Yes, the grammar is LR(1). FOLLOW(S) = { $, ; ,) } .

Any LR(1) lookahead set for an S rule must be a subset of FOLLOW(s), and *a* and *(* are not in FOLLOW(S). Thus, the LR(0) shift-reduce conflict is eliminated.

(e) Is this language regular ?

No, it requires bracketing (matching), and requires an unbounded number of states and so cannot be recognized by a finite automaton.

Q10

Prove that the following grammar is LL(1):

declaration $\rightarrow$ ID declaration_tail

declaration_tail $\rightarrow$, declaration

declaration_tail $\rightarrow$: ID ;

Answer Q10

Prove that the following grammar is LL(1):

declaration $\rightarrow$ ID declaration_tail

declaration_tail $\rightarrow$, declaration

declaration_tail $\rightarrow$: ID ;

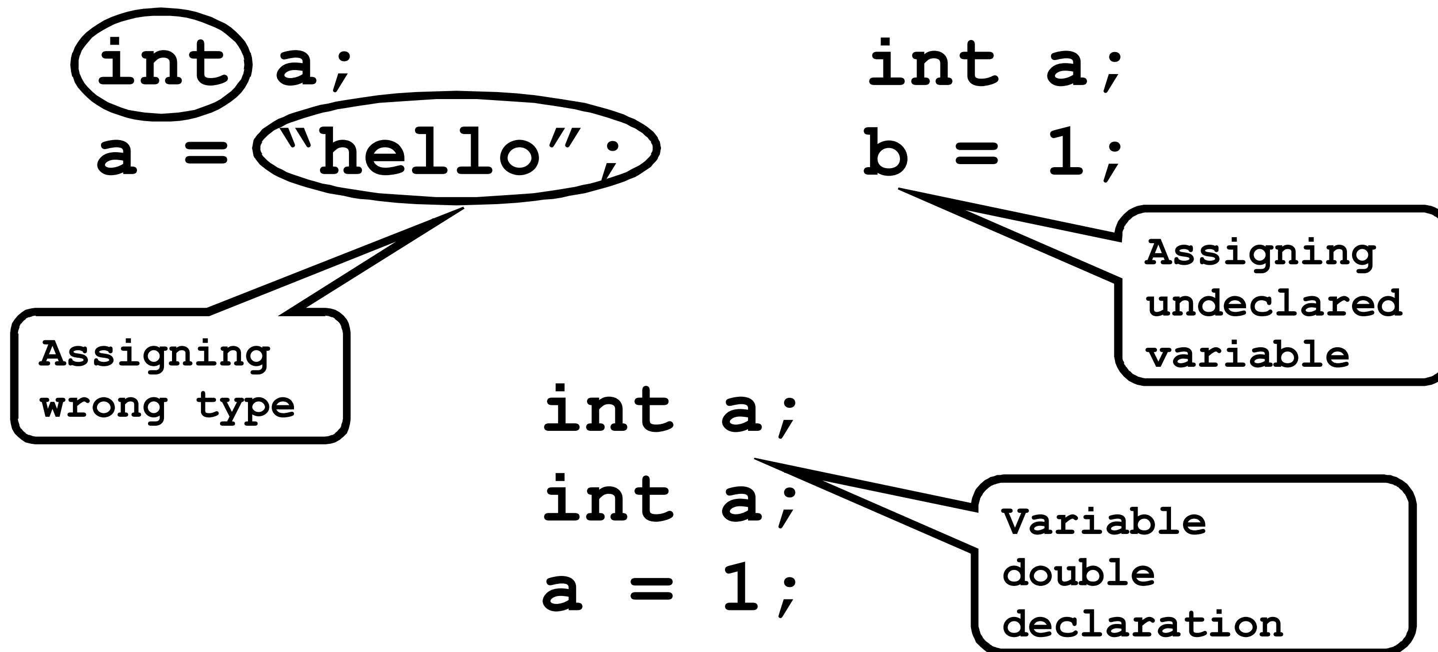
By definition, a grammar is LL(1) if it can be parsed by an LL(1) parser. It can be parsed by an LL(1) parser if no conflicts arise in the creation of the parse table. In this grammar, no symbols generate ε , so the table can be built entirely from FIRST sets; FOLLOW sets do not matter. There is only one symbol, declaration_tail, with more than one production, and the FIRST sets for the right-hand sides of those productions are distinct ({,} and {;}). Therefore no conflicts arise.

Semantic analysis

- Context analysis
  - Does break and continue appear only inside while statement?
- Scope analysis
  - Every variable is predefined
  - No double definitions
  - Bound variable use to its definition
- Type checking
  - Every expression is well typed
  - Every statement is well typed

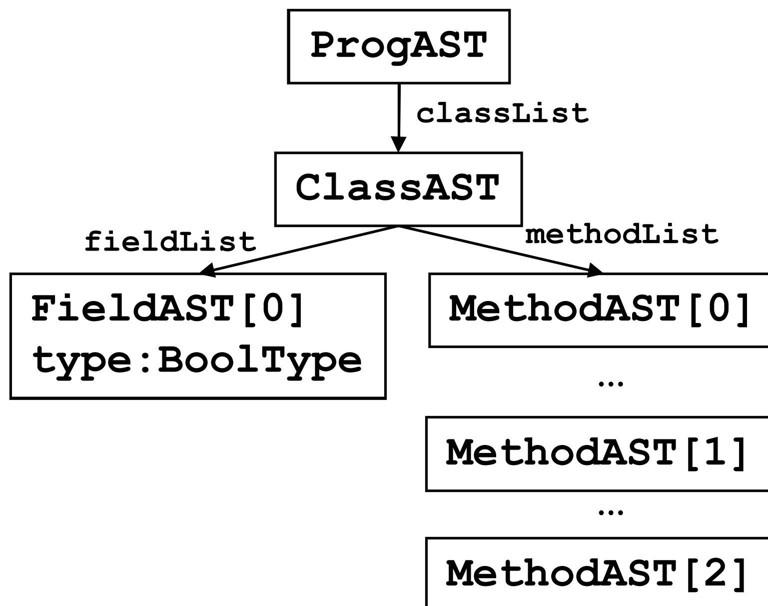
Semantic analysis

Syntax analysis is not enough



Semantic analysis

- Representing scopes
- Type-checking
- Semantic checks



(Program)

Symbol	Kind	Type
Hello	class	Hello

(Hello)

Symbol	Kind	Type	Properties
state	field	boolean	instance
main	method	string[]->void	static
rise	method	void->boolean	instance
setState	method	boolean->void	instance

(setState)

Symbol	Kind	Type
newState	param	int

...

Examples of type errors

assigned type doesn't match declared type

```
int a; a = true;
```

relational operator applied to non-int type

```
1 < true
```

```
void foo(int x) {
    int x;
    foo(5,7);
}
```

argument list doesn't match formal parameters

```
class A {…}
class B extends A {
    void foo() {
        A a;
        B b;
        b = a;
    }
}
```

a is not a subtype of **b**

Type rules

$$\frac{}{E \vdash \text{true} : \text{bool}} \qquad \frac{}{E \vdash \text{false} : \text{bool}}$$

$$\frac{}{E \vdash \textit{int-literal} : \text{int}} \qquad \frac{}{E \vdash \textit{string-literal} : \text{string}}$$

$$\frac{E \vdash \text{e1} : \text{int} \qquad E \vdash \text{e2} : \text{int}}{E \vdash \text{e1} \; \textit{op} \; \text{e2} : \text{int}} \qquad \textit{op} \in \{ +, -, /, *, \% \}$$

$$\frac{E \vdash \text{e1} : \text{int} \qquad E \vdash \text{e2} : \text{int}}{E \vdash \text{e1} \; \textit{rop} \; \text{e2} : \text{bool}} \qquad \textit{rop} \in \{ <=, <, >, >= \}$$

$$\frac{E \vdash \text{e1} : T \qquad E \vdash \text{e2} : T}{E \vdash \text{e1} \; \textit{rop} \; \text{e2} : \text{bool}} \qquad \textit{rop} \in \{ ==, != \}$$

Q11: Semantic conditions

- What is checked in compile-time and what is checked in runtime?

Event	C/R
Program execution halts	
Break/continue inside a while statement	
Array index within bound	
In Java the cast statement **`(A)f`** is legal	
In Java method o.m(…) is illegal since m is private	

Semantic conditions

- What is checked in compile-time and what is checked in runtime?

Event	C/R
Program execution halts	**R** (undecidable in general)
Break/continue inside a while statement	**C**
Array index within bound	**R** (undecidable in general)
In Java the cast statement **`(A)f`** is legal	Depends: if **`A`** is sub-type of **`f`** then checked during runtime (raising exception), otherwise flagged as an error during compilation
In Java method o.m(…) is illegal since m is private	**C**

Q12: language features

- Support Java override annotation *inside comments*
  - // @Override
  - Annotation is written above method to indicate it overrides a method in superclass
- Describe the phases in the compiler affected by the change and the changes themselves

Legal program

```
class A {
  void rise() {…}
}
class B extends A {
  // @Override
  void rise() {…}
}
```

Illegal program

```
class A {
  void rise() {…}
}
class B extends A {
  // @Override
  void ris() {…}
}
```

Answer

- The change affects the lexical analysis, syntax analysis and semantic analysis

- Does not affect later phases
  - User semantic condition

Changes to scanner

- Add pattern for @Override inside comment state patterns
- Change action for comments
  - instead of not returning any tokens, we now return a token for the annotation

```
boolean override=false;
%%
<INITIAL> // { override=false; yybegin(comment); }
<comment> @Override { override=true; }
<comment> \n { if (override)
                  return new Token(…,override,…)
             }
```

Changes to parser and AST

PARSER

method $\rightarrow$ static type name params '{' mbody '}'

| type name params '{' mbody '}'

| OVERRIDE type name params '{' mbody '}'

AST

Add a Boolean flag to the method AST node to indicate that the method is annotated

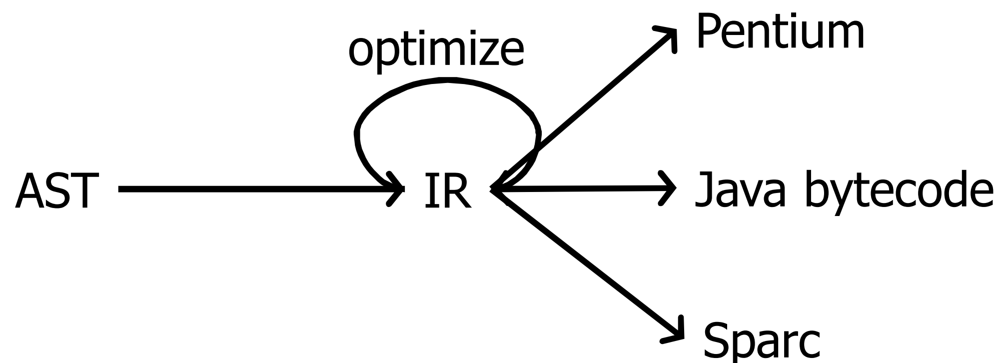
Changes to semantic analysis

- Suppose we have an override annotation for a method *m* in class *A*

- We check the following semantic conditions
    1. class *A* extends a superclass (otherwise it does not make sense to override a method)
    2. Traverse the superclasses of A by going up the class hierarchy until we find the first method *m* and check that it has the same signature as *A.m*
       If we fail to find such a method we report an error

Intermediate representation

- Allows language-independent, machine independent optimizations and transformations
- Easy to translate from AST
- Easy to translate to assembly



Translation to IR

- Accept AST and translate functions into lists of instructions
  - Compute offsets for fields and virtual functions
- Dispatch vectors
- Register allocation

Q13: Translation to IR

Question: write the method tables for **Rectangle** and **Square**

```
class Shape {
  boolean isShape() {return true;}
  boolean isRectangle() {return false;}
  boolean isSquare() {return false;}
  double surfaceArea() {…}
}
class Rectangle extends Shape {
  double surfaceArea() {…}
  boolean isRectangle() {return true;}
}
class Square extends Rectangle {
  boolean isSquare() {return true;}
}
```

Answer

Method table for rectangle
Shape_isShape
Rectangle_isRectangle
Shape_isSqaure
Rectangle_surfaceArea

Method table for square
Shape_isShape
Rectangle_isRectangle
Sqaure_isSqaure
Rectangle_surfaceArea

Q14: Semantic Analysis

6.3 The following declarations are given for a language that uses name equivalence.

```
A, B: array [1..10] of int;
C : array [1..10] of int;
D : array [1..10] of int;
```

Which of these four variables have the same type?

Answer Q14

6.3 The following declarations are given for a language that uses name equivalence.

```
A, B: array [1..10] of int;
C : array [1..10] of int;
D : array [1..10] of int;
```

A and B

Q15

```
class A {...};
class B extends A {...};
B[] bArray = new B[10];
A[] aArray = bArray;
A x =new A();
if (...)x =new B();
aArray[5]=x;
```

(a) Explain why line 4 of the Java code, A[] aArray =bArray; is considered well-typed in Java.
(b) Under what conditions could the assignment aArray[5]=x; lead to a run-time type error? Explain.
(c) What does Java do to manage this problem with the assignment aArray[5]=x?

Q16

- Add support of access qualifiers to a Java-like langauge
  - Allow methods to be defined as public or private
  - Public --- method accessible to all classes
  - Private --- method accessible only to its own class
  - Assume that subclasses cannot modify the accessibility defined by a superclass (e.g., a method defined private by a superclass cannot be made private by an overriding implementation)

- Why is it helpful to have the assumption that access qualifiers are not modified by subclasses?

Disclaimer

Questions provided here are just a sample of reasonable questions and do not cover all course material.

In particular, emphasis in this collection was on parsing, neglecting other topics.

EXTRAS

Q17

Calculate nullable, FIRST and FOLLOW sets for the following grammar

$S \rightarrow uBDz$

$B \rightarrow Bv$

$B \rightarrow w$

$D \rightarrow EF$

$E \rightarrow y$

$E \rightarrow \varepsilon$

$F \rightarrow x$

$F \rightarrow \varepsilon$

Answer Q17

	nullable	FIRST	FOLLOW
B	no	w	v,x,y,z
D	yes	x,y	z
E	yes	y	x,z
F	yes	x	z
S	no	u	

Q18 Problem 3.8 from [Appel]

A simple left-recursive grammar:

$S \rightarrow S + a$

$S \rightarrow a$

A simple right-recursive grammar that accepts the same language:

$S \rightarrow a + S$

$S \rightarrow a$

Which has better behavior for shift-reduce parsing?

Answer Q18

Consider the input string: a+a+a+a+a

For the left-recursive case, the stack looks like:
a (reduce)
S
S +
S + a (reduce)
S
S +
S + a (reduce)
S
S +
S + a (reduce)
S
S +
S + a (reduce)
S

The stack never has more than three items on it. In general, with LR-parsing of left-recursive grammars, an input string of length O(n) requires only O(1) space on the stack.

Answer Q18

For the right-recursive case, the stack looks like:

a
a +
a + a
a + a +
a + a + a
a + a + a
a + a + a + a
a + a + a + a +
a + a + a + a + a (reduce)
a + a + a + a + S (reduce)
a + a + a + S (reduce)
a + a + S (reduce)
a + S (reduce)
S

The stack grows as large as the input string. In general, with LR-parsing of right-recursive grammars, an input string of length O(n) requires O(n) space on the stack.

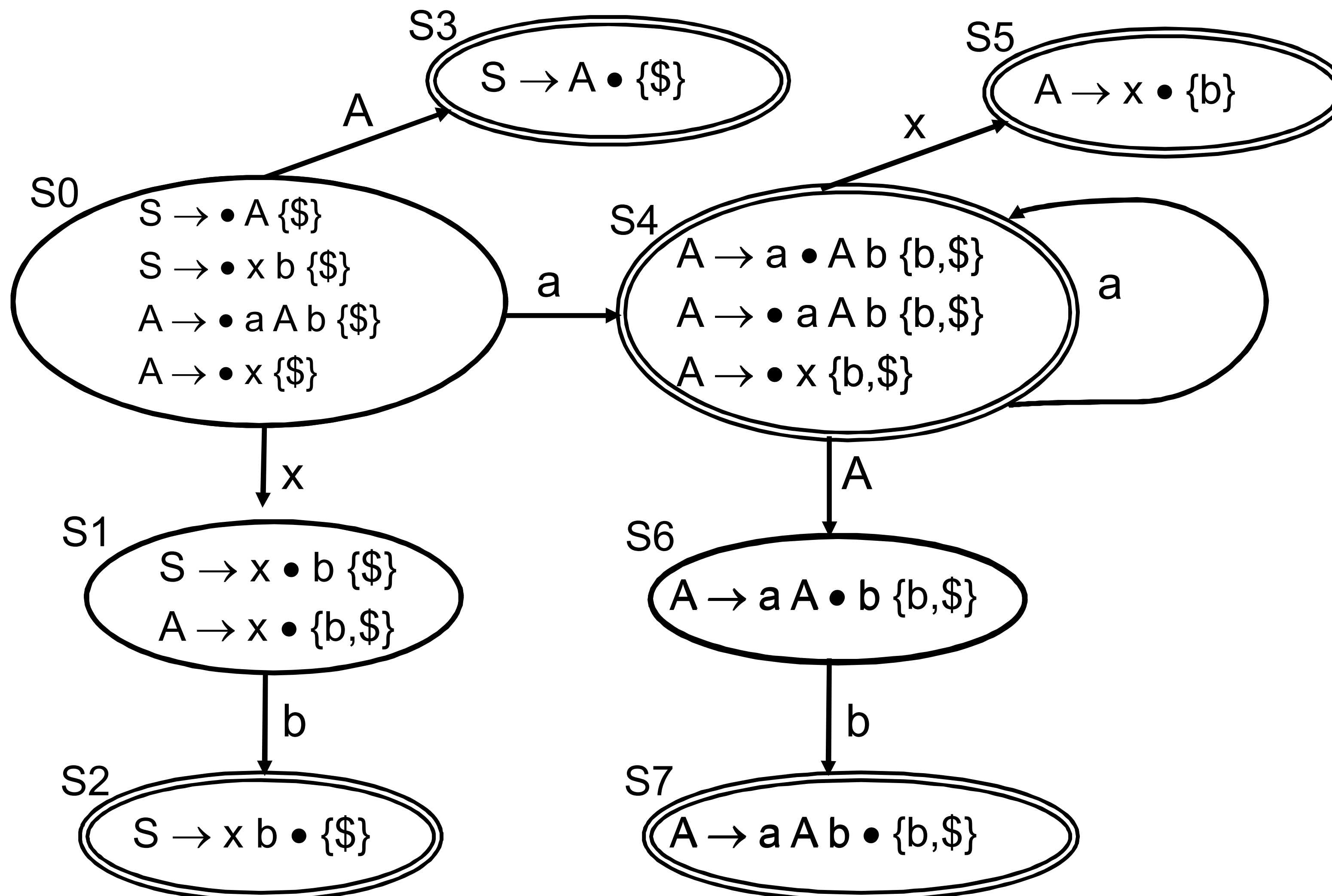
(taken from http://science.slc.edu/~msiff/old-courses/compilers/notes/parse.html)

Q19

- derive the LALR(1) automaton (and ACTION/GOTO table) for the following grammar:

$$S \rightarrow A \mid x\ b$$
$$A \rightarrow a\ A\ b \mid x$$

Answer Q19 - LALR(1) automaton



LALR(1) ACTION/GOTO table

state	stack symbol / look-ahead token				
	a	b	x	$	A
0	s4		s1		s3
1		s2		r4	
2		r2		r2	
3				r1	
4	s4		s5		s6
5		r4		r4	
6		s7			
7		r3		r3	

1: S $\rightarrow$ A
2: S $\rightarrow$ x b
3: A $\rightarrow$ a A b
4: A $\rightarrow$ x

Q20

- derive the SLR(1) ACTION/GOTO table (with shift-reduce conflict) for the following grammar:

$$S \rightarrow A \mid x\ b$$
$$A \rightarrow a\ A\ b \mid x$$

Answer Q20

state	stack symbol / look-ahead token				
	a	b	x	$	A
0	s4		s1		s3
1		s2/r4		r4	
2		r2		r2	
3				r1	
4	s4		s5		s6
5		r4		r4	
6		s7			
7		r3		r3	

1: $S \rightarrow A$
2: $S \rightarrow x\ b$
3: $A \rightarrow a\ A\ b$
4: $A \rightarrow x$

FOLLOW(S) = {$}
FOLLOW(A) = {$,b}

Q21

$P \rightarrow E$

$E \rightarrow int$

$E \rightarrow E + E$

$E \rightarrow E / E$

$E \rightarrow E - E$

$E \rightarrow E * E$

$E \rightarrow E \% E$

(a) Is this grammar ambiguous ? Why?
[Yes]

Q21

(a) Draw an AST for the expression below. Label each AST node clearly with the meaning of the node (for example, "addition," "identifier," etc). Invent new types of AST nodes as necessary.

x++ + ++x

(a) It is interesting to observe that while

x++ + ++x

is a legal Java expression, the same expression without white spaces, namely

x+++++x

is not a legal Java expression. That is, the latter expression will cause a compile-time error. Identify the phase of the compiler in which the error occurred. Depending on the compiler, the error can be flagged in different stages, and so there is more than one correct answer.