COMP 520 - Compilers

Lecture 9 (Tue Feb 15, 2022)

AST wrap-up and Bottom-up Parsing

- Written assignment 3
  - Exercise to help you transform your PA1 grammar to construct a precedence parser for PA2
  - Due Mon Feb 21

Topics

- PA1 tester for use in Eclipse
  - view of projects and packages

- AST construction and visitor classes
  - AST construction and traversal example
    - simpleAST example walkthrough – code on web site
  - miniJava ASTs
    - miniJava AbstractSyntaxClasses walkthrough
    - ASTDisplay

- Precedence parsing using a bottom-up parser
  - how does a bottom up parser work?
  - yacc & lex
    - precedence parsing

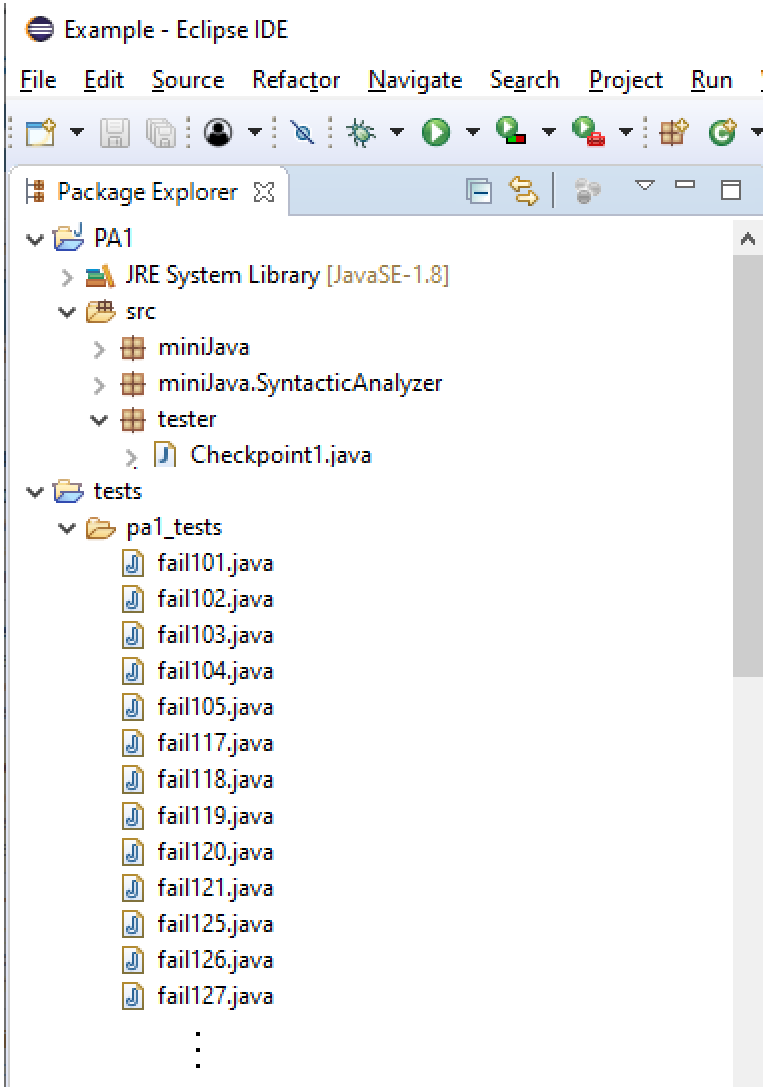
PA1 testing and grading

- Tricky parts in PA1 scanning and parsing
  - Left factoring the miniJava grammar
  - Comments as whitespace
  - Scanning tokens with valid prefixes
    - e.g. < vs <= or / vs // vs /*
  - Unclosed comments

- Testing
  - Special accommodation for small errors with large consequences
    - oblivious parsers

- Tests
  - `pa1_tests` and the `Checkpoint1.java` tester
  - run all tests or debug individual tests

Eclipse view of Tester

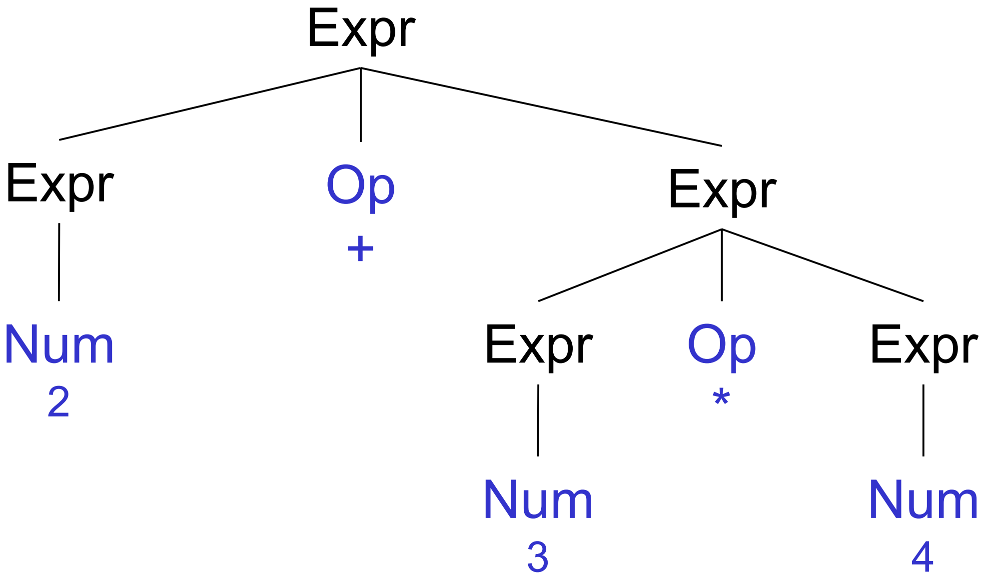


Simple AST example

- Check simpleAST in Examples section online
  - ex: simple arithmetic expressions

Expr ::= Expr Op Expr (BinExpr)
| Num (NumExpr)

- Abstract syntax tree for 2 + (3 * 4)


Expr
Expr
Op
+
Expr
Num
2
Expr
Op
*
Expr
Num
3
Num
4

AST representation

- Example

Expr ::= Expr Oper Expr (BinExpr)
| Num (NumExpr)

```
abstract public class AST {}

abstract public class Expr extends AST {}

public class BinExpr extends Expr {
   public Token op;
   public Expr left, right;
   public BinExpr(Expr left, Terminal oper, Expr right) { … }
}

public class NumExpr extends Expr {
   public Token num;
   public NumExpr(Terminal num) { … }
}
```

Building an AST during a concrete syntax parse

- concrete syntax for arithmetic expression grammar

  E ::= T | E *op* T
  T ::= (E) | *num*

- transformed and augmented

  S ::= E $
  E ::= T (*op* T)*
  T ::= (E) | *num*

- abstract syntax

  Expr ::= Expr Op Expr (BinExpr)
  | Num (NumExpr)

- how to build AST?
  - modify parse procedures to return pieces of AST
    - assume curToken has type Terminal

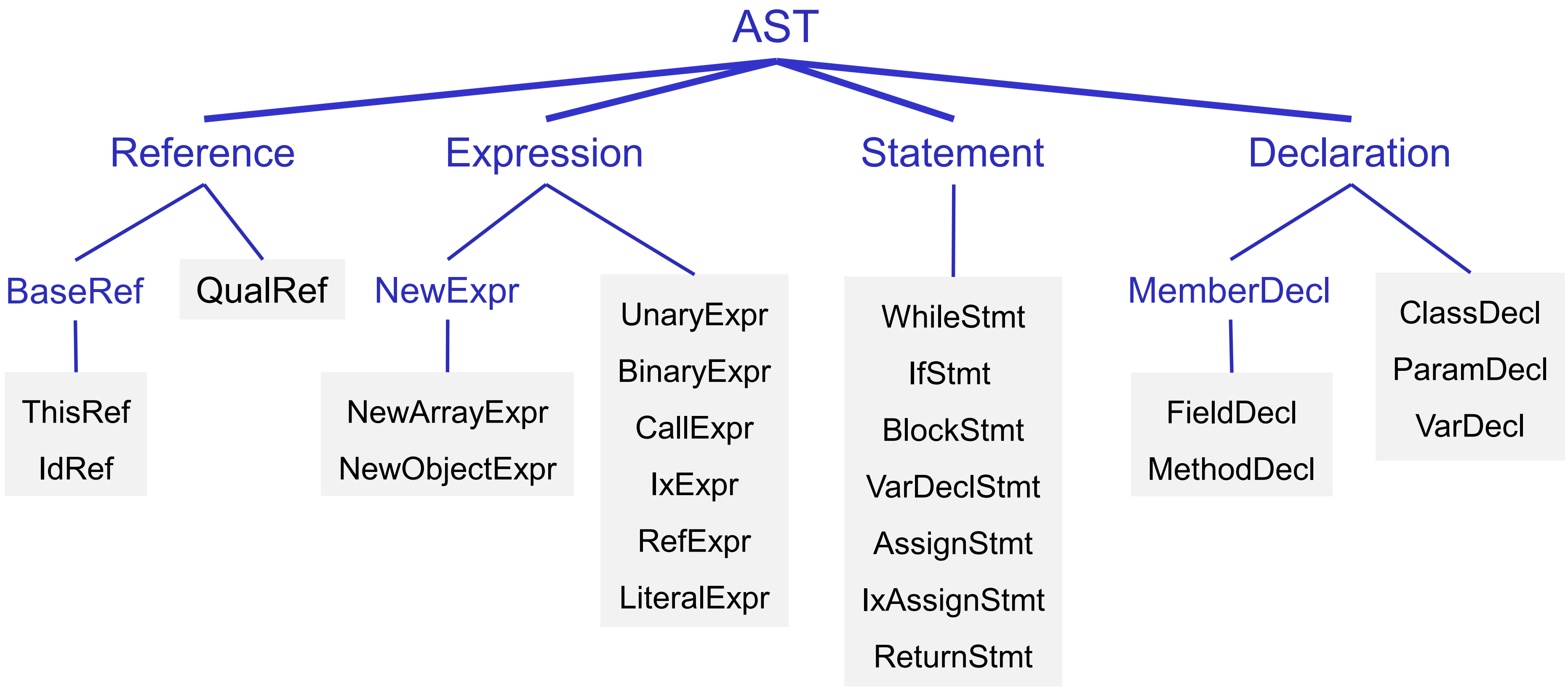
```
Expr parseS() {
   Expr e = parseE();
   accept(Token.eot);
   return e
}

Expr parseE() {
   Expr e1 = parseT();
   while (curToken.kind == Token.op) {
       Terminal op = curToken;
       acceptIt();
       Expr e2 = parseT();
       e1 = new BinExpr(e1,op,e2);
   }
   return e1;
}

Expr parseT() {
   case  (curToken.kind) {
   Token.LPAREN:
      acceptIt();
      Expr e1 = parseE();
      accept(Token.RPAREN);
      return e1;

   Token.num:
      NumExpr e2 = new NumExpr(curToken);
      acceptIt();
      return e2;
}}
```

The miniJava AST classes



Additional constructors for lists of class instances, with an iterator for traversal of the list:
ClassDeclList, FieldDeclList, MethodDeclList, ParamDeclList, StatementList, ExprList

Sample parseProgram()

```
public Package parseProgram() {
        // start scanner
        currentToken = lexicalAnalyser.scan();
        previousToken = currentToken;

        SourcePosition start =  currentToken.posn;
        try {
                ClassDeclList cl = new ClassDeclList();
                while (currentToken.kind == TokenKind.CLASS) {
                        cl.add(parseClass());
                }
                SourcePosition end = previousToken.posn;
                if (currentToken.kind != TokenKind.EOT)
                        syntaxError("Unexpected text \"%\" after end of program",
                                        currentToken.spelling);
                return new Package(cl, new SourcePosition(start, end));
        }
        catch (SyntaxError s) { return null; }
}
```

Bottom-up parsing

- Example

  CFG G has N = {S, A, B, D}, T = {a,b,d,$}

  S ::= A $
  A ::= B | D
  B ::= a B | b
  D ::= a D | d

  Why not LL(1)? Can we left-factor?

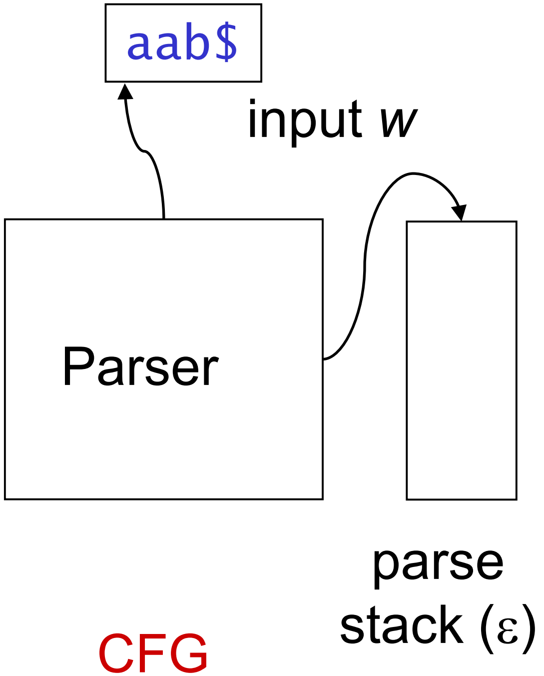
- CFG can be parsed "as is" using a bottom up parser
  - BU parser works by procrastination!
  - To parse w
    - parser delays its decision which rule to use as long as possible
  - It maintains the *viable prefix* property
    - If v is the prefix of w that has been read so far, then there must exist some u ∈ NT* such that S =>* vu. This means vu ∈ L(G)
    - How to ensure the viable prefix property?

Bottom-up parsing

- How do we recognize sentences using a BU parser?
  - Simulate a derivation
    - input is read *left to right*
    - BU parser simulates a *rightmost* derivation in *reverse!*
    - LR parser

- Bottom-up parser operation
  - parse stack initialized to ε
  - repeat until no choice available
    - SHIFT terminal `t` onto parse stack

    OR
    - REDUCE α at top of parse stack to A
      - “predicting” correct rule A ::= α
  - $w \in L(G)$ iff parse stack = S when input is consumed



CFG

S ::= A $
A ::= B | D
B ::= a B | b
D ::= a D | d

Bottom-up parser

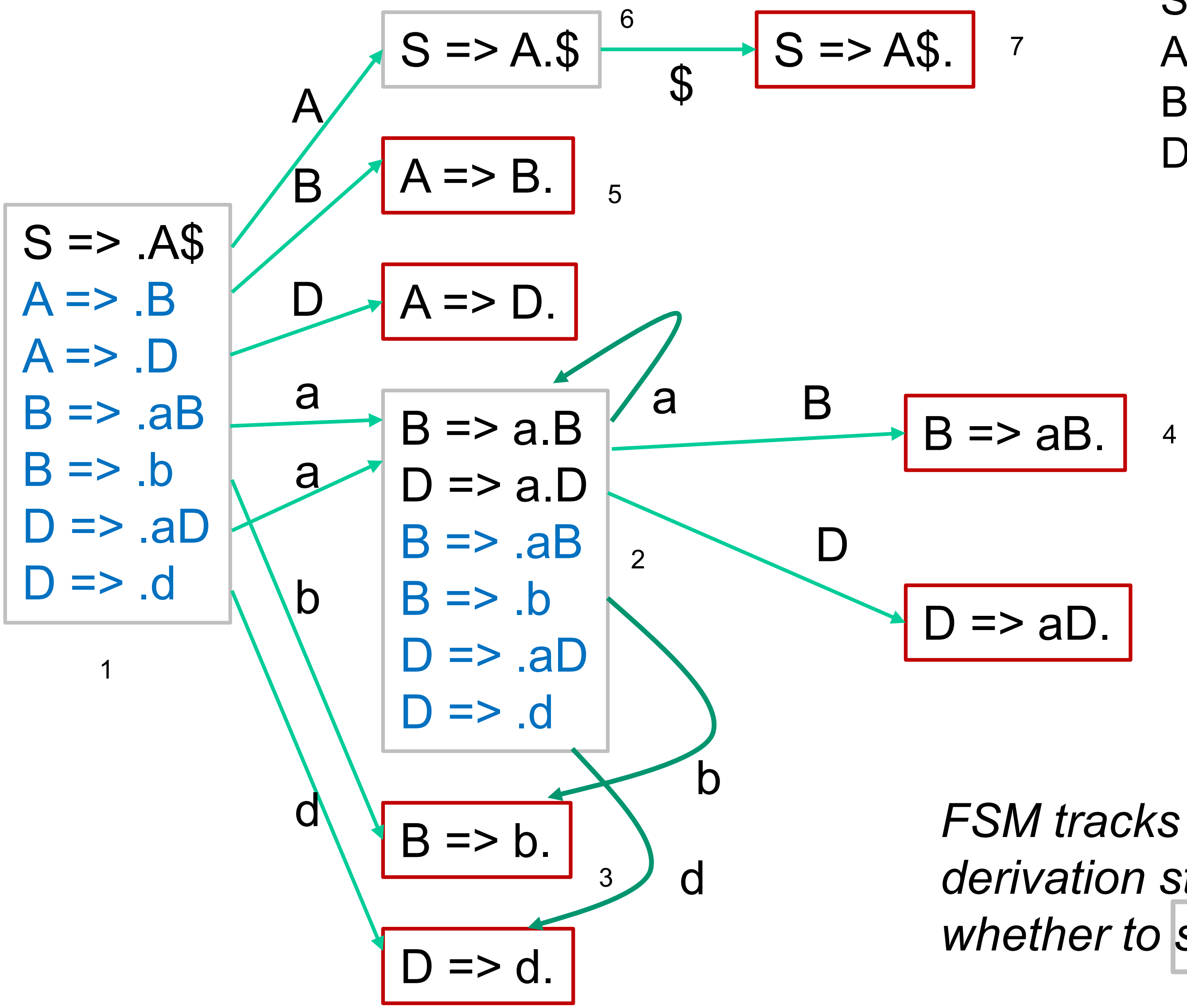
- Most powerful linear-time parser
  - parses largest class of grammars with given lookahead
    - LR(0), LR(1), …
  - but can not parse all CFGs

- Uses a pair of stacks and a single table State x terminal -> State
  - symbol stack
    - symbol stack concatenated with remaining input is a sentential form in a rightmost derivation
  - state stack
    - determines which right hand side of a rule appears at stack top

- Works “backwards”
  - shifts input onto empty stack and replaces right hand sides of rules at stack top with left-hand nonterminal

- not amenable to direct implementation using recursive procedures
  - grammar analysis can generate a table-driven parser
  - typically use simplified lookahead LALR(1) to keep table size small

Canonical FSM for grammar



S => .A\$
A => .B
A => .D
B => .aB
B => .b
D => .aD
D => .d

S => A.\$

S => A\$.

A => B.

A => D.

B => a.B
D => a.D
B => .aB
B => .b
D => .aD
D => .d

B => aB.

D => aD.

B => b.

D => d.

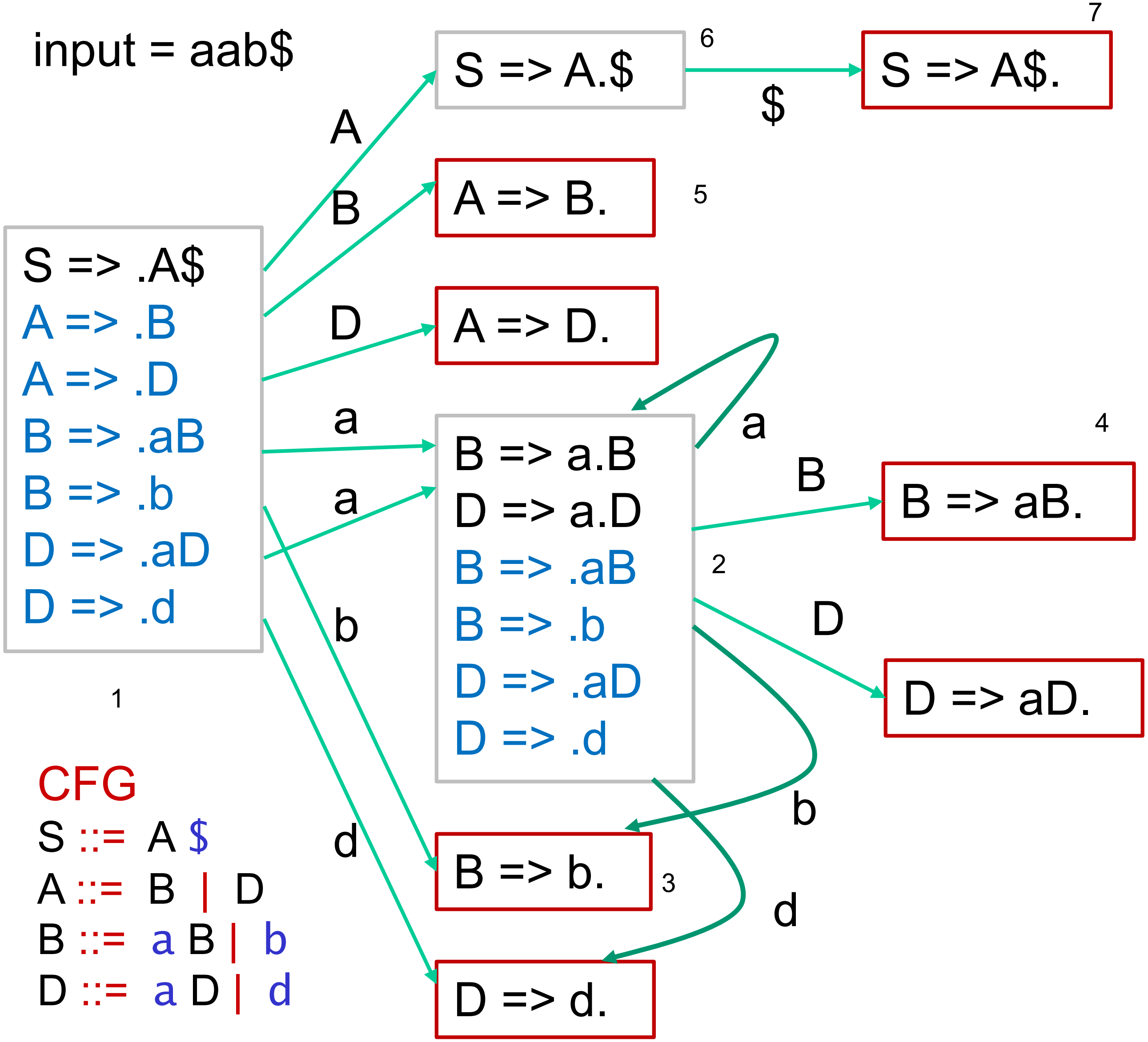
CFG

S ::= A \$
A ::= B | D
B ::= a B | b
D ::= a D | d

FSM tracks possible derivation states to decide whether to shift or reduce

Tracing BU recognition of aab$

input = aab$



1
S => .A$
A => .B
A => .D
B => .aB
B => .b
D => .aD
D => .d

2
B => a.B
D => a.D
B => .aB
B => .b
D => .aD
D => .d

3
B => b.

4
B => aB.

5
A => B.

6
S => A.$

7
S => A$.

D => aD.

A => D.

D => d.

CFG
S ::= A $
A ::= B | D
B ::= a B | b
D ::= a D | d

symb stack		state stack
ϵ		1
	sh a	
a		1 2
	sh a	
aa		1 2 2
	sh b	
aab		1 2 2 3
	red	
aaB		1 2 4
	red	
aB		1 4
	red	
B		1 5
	red	
A		1 6
	sh $	
A$		1 7
	red	
S		ϵ

So why don’t we use a bottom-up parser?

- Canonical FSM doesn’t always yield conflict-free decisions in each state!
  - shift-reduce conflict
  - reduce-reduce conflict

- How does this get presented to the parser programmer?
  - shift-reduce conflict in state 285
  - reduce-reduce conflict in state 317, 424, 111
  - requires considerable expertise to resolve

- Bottom-up parsing
  - left recursion is very easy and generally doesn’t require any lookahead!
  - right recursion requires stacking input and lookahead
  - But …/ there exist simple grammars that are not LR(k) for any k>0

BU parser easily incorporates precedence and evaluation

- Scanner (flex)

```
/* scanner for integer expression
 * evaluator
 */

%{
#include "y.tab.h"
%}

Num [0-9]+
WS  [ \t\n]*

%%

"+"       return(T_PLUS);
"-"       return(T_MINUS);
"*"       return(T_TIMES);
"/"       return(T_DIV);
"("       return(T_LPAREN);
")"       return(T_RPAREN);
{Num}     return(T_NUM);
{WS}      ;
.         printf("lexical error");
          exit(4);
```

Associativity and precedence

- Parser (yacc)

```
/* parser for integer expression evaluation
 * with precedence
 */

%{
extern char yytext[];   /* token spelling */
%}

%token T_NUM T_PLUS T_MINUS T_TIMES T_DIV
       T_LPAREN T_RPAREN
%left T_PLUS T_MINUS
%left T_TIMES T_DIV
%right NEG

%%

S     : Expr      {printf("Ans = %d\n",$1);}
      ;

Expr  : Expr T_PLUS Expr    {$$ = $1 + $3;}
      | Expr T_MINUS Expr   {$$ = $1 - $3;}
      | Expr T_TIMES Expr   {$$ = $1 * $3;}
      | Expr T_DIV Expr     {$$ = $1 / $3;}
      | T_LPAREN Expr T_RPAREN   {$$ = $2;}
      | T_MINUS Expr  %prec NEG  {$$ = -$2;}
      | T_NUM             {$$ = atoi(yytext);}
      ;
```