COP5556 Programming Language Principles

Parsing 2

- Recap of parsing so far:
  - Specify phrase structure of programming language with context free grammar (CFG) using EBNF notation
  - CFGs generate sentences
  - Parsers recognize sentences
    - Several approaches to parsing
      - bottom up
      - top down
  - Can classify grammars according to the type of parsing algorithm that can be used to parse them.

- We are primarily interested in (mostly) LL(1) grammars which admit top-down parsing with one-step lookahead.
- We defined
  - FIRST set
  - FOLLOW set
  - PREDICT set
  - Grammar is LL(1) if the PREDICT sets of all productions with the same left hand side are disjoint

Is this grammar LL(1)?

dec ::= modifier type ident ;

dec ::= procedure ident (formals) body ;

type ::= int | bool

modifier ::= public | ε

formals ::= ε | type ident (, type ident)*

dec ::= modifier type ident ;

dec ::= procedure ident (formals) body ;

type ::= int | bool

modifier ::= public | ε

formals ::= ε | type ident (, type ident)*

PREDICT(dec ::= modifier type ident ;) = ????

dec ::= modifier type ident ;

dec ::= procedure ident (formals) body ;

type ::= int | bool

modifier ::= public | ε

formals ::= ε | type ident (, type ident)*

PREDICT(dec ::= modifier type ident ;) =

FIRST(modifier type ident ;) =

{ public, int, bool }

dec ::= modifier type ident ;

dec ::= procedure ident (formals) body ;

type ::= int | bool

modifier ::= public | ε

formals ::= ε | type ident (, type ident)*

PREDICT(dec ::= modifier type ident ;) =

{ public, int, bool }

PREDICT(dec ::= procedure ident (formals) body ;)=

{ procedure }

dec ::= modifier type ident ,

dec ::= procedure ident (form

type ::= int | bool

modifier ::= public | ε

formals ::= ε | type ident (

Same left hand side, PREDICT sets are disjoint, so far so good

---

PREDICT(dec ::= modifier type ident ,) =

{ public, int, bool }

PREDICT(dec ::= procedure ident (formals) body ;)

={ procedure }

dec ::= modifier type ident ;

dec ::= procedure ident (formals) body ;

type ::= int | bool

modifier ::= public | ε

formals ::= ε | type ident (, type ident)*

PREDICT (type ::= int) = { int }

PREDICT (type ::= bool) = { bool }

PREDICT (modifier ::= public) = { public }

PREDICT (modifier ::= ε) =FOLLOW(modifier) = { int, bool }

dec ::= modifier type ident ;
dec ::= procedure ident (formals) ;
type ::= int | bool
modifier ::= public | ε
formals ::= ε | type ident (, ...)

Same left hand side, PREDICT sets are disjoint, still on track to be LL(1)

PREDICT (type ::= int) = { int }
PREDICT (type ::= bool) = { bool }

PREDICT (modifier ::= public) = { public }
PREDICT (modfier ::= ε) = FOLLOW(modifier) =
{ int, bool }

dec ::= modifier type ident ;

dec ::= procedure ident (formals) body ;

type ::= int | bool

modifier ::= public | ε

formals ::= ε | type ident (, type ident)*

PREDICT (formals ::= type ident (, type ident)*) =

{ int, bool }

PREDICT (formals ::= ε) = FOLLOW(formals) = {) }

Same left hand side, PREDICT sets are disjoint.

Grammar is LL(1)

dec ::= modifier type ident ;

dec ::= procedure ident (f

type ::= int | bool

modifier ::= public | ε

formals ::= ε | type ident (, type ident)

PREDICT (formals ::= type ident (, type ident)*) =

{ int, bool }

PREDICT (formals ::= ε) =FOLLOW(formals) ={) }

dec ::= modifier type ident ; { public, int, bool }

dec ::= procedure ident (formals) body ;

{ procedure }

type ::= int {int } | bool { bool }

modifier ::= public { public }

| ε {int, bool }

formals ::= ε {) }

| type ident (, type ident)* {int, bool }

Top down parse of int ident ;

dec int ident ;

dec ::= modifier type ident ; { public, int, bool }

dec ::= procedure ident (formals) body ;

{ procedure }

type ::= int {int } | bool { bool }

modifier ::= public { public }

| ε {int, bool }

formals ::= ε {) }

| type ident (, type ident)* {int, bool }

Top down parse of int ident ;

dec

modifier type ident ;

int ident ;

dec ::= modifier type ident ; { public, int, bool }
dec ::= procedure ident (formals) body ;
{ procedure }
type ::= int {int } | bool { bool }
modifier ::= public { public }
| ε {int, bool }
formals ::= ε {) }
| type ident (, type ident)* {int, bool }

Top down parse of int ident ;

dec
modifier type ident ;
ε type ident ;

int ident ;

dec ::= modifier type ident ; { public, int, bool }
dec ::= procedure ident (formals) body ;
{ procedure }
type ::= int {int } | bool { bool }
modifier ::= public { public }
| ε {int, bool }
formals ::= ε {) }
| type ident (, type ident)* {int, bool }

Top down parse of int ident ;

dec int ident ;
modifier type ident ;
ε type ident ;
int ident ; int ident ;

- Summary
  - property of PREDICT sets tells us that grammar is LL(1)
  - PREDICT sets are also used during parsing to choose a production

Is this grammar LL(1)?

ident_list ::= ident

ident_list ::= *ident_list*, ident

Is this grammar LL(1)?

ident_list ::= ident | *ident_list* , ident

- FIRST(ident_list) =
  ident ∪ FIRST(ident_list)

PREDICT (*ident_list* ::= ident) = {ident}
PREDICT (*ident_list* ::= *ident_list* , ident)
= FIRST(ident_list) = {ident}

Is this grammar LL(1)?

ident_list ::= ident | *ident_list* , ident

- FIRST(ident_list) =
  ident U FIRST(ident_list)

PREDICT (*ident_list* ::= ident)
PREDICT (*ident_list* ::=
= FIRST(ident_lis

If you see left recursion,
the grammar is not LL

Left recursion

- We saw that left recursion makes a grammar non-LL(1)
- A grammar is left recursive if, for some nonterminal A,

  $A \rightarrow^{+} A\ \sigma$.
- If there is a production of the form

  $A ::= A\ \sigma \mid \beta$

  then we immediately see that it is left recursive.
- Left recursion may be indirect, involving several rules, but we will only worry about direct recursion

Left recursion removal

A production of the form

$A ::= A\,\sigma \mid \beta$

can be replaced in the grammar with

$A ::= \beta\ B$

$B ::= \sigma\ B \mid \varepsilon$

B is a new symbol

Left recursion removal (2)

More generally, productions of the form

$A ::= A\ \sigma 1 \mid \ldots \mid A\ \sigma n \mid \beta 1 \mid \ldots \mid \beta m$

can be replaced by

$A ::= \beta 1\ B \mid \ldots \mid \beta m\ B$

$B ::= \sigma 1\ B \mid \ldots\ \sigma n\ B \mid \varepsilon$

Original grammar

ident_list ::= ident | *ident_list* , ident

- Recall rule : replace A ::= A σ | β with
  A ::= β B
  B ::= σ B | ε
- Let
  - A := *ident_list*
  - σ := , ident,
  - β = ident,
  - introduce new symbol *ident_list_tail* instead of B

Equivalent grammar without left-recursion

ident_list ::= ident *ident_list_tail*
ident_list_tail ::= , ident *ident_list_tail* | ε

Transforming to EBNF

A ::= β B
B ::= σ B | ε

can be replaced with

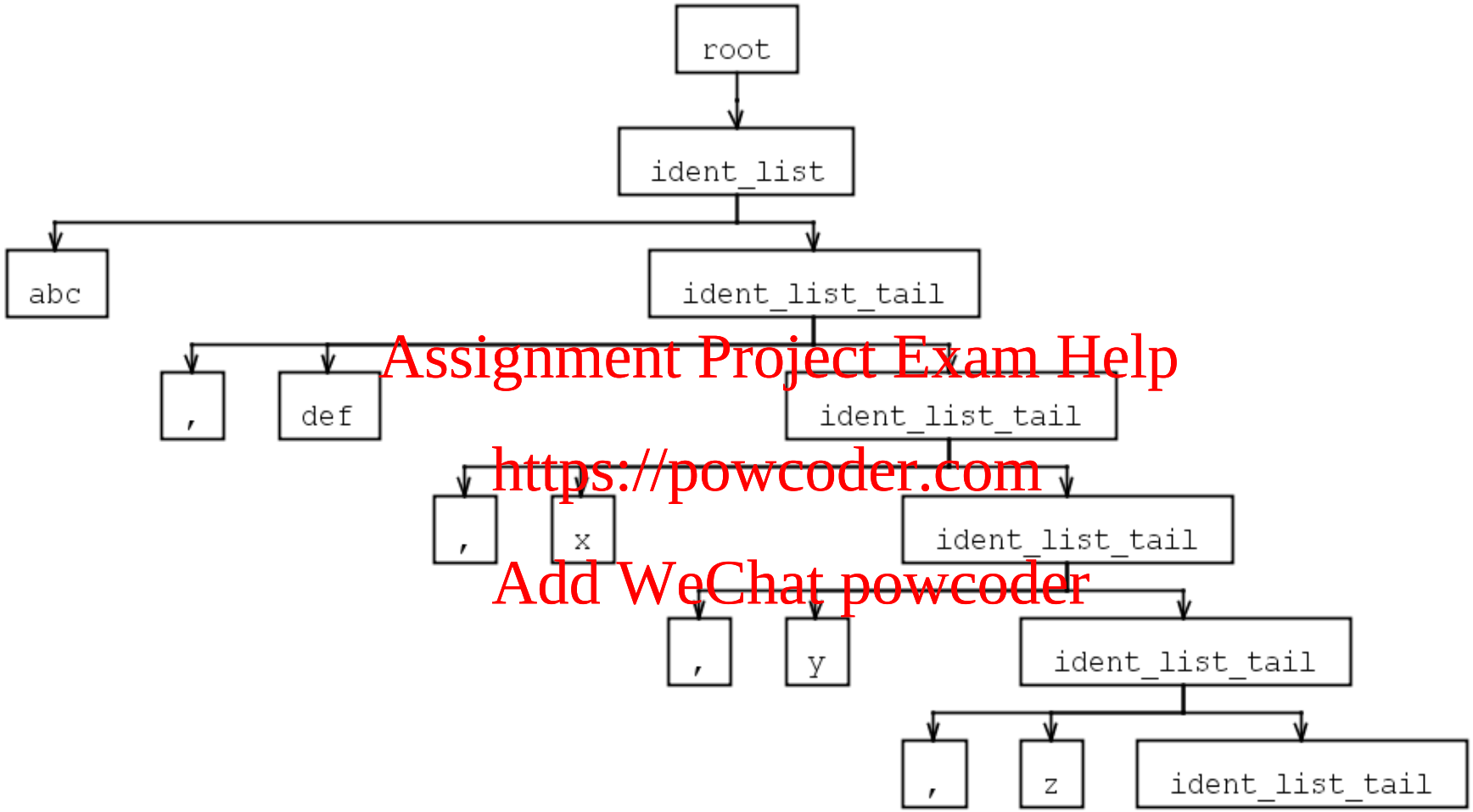
A ::= β σ*

ident_list ::= ident (, ident)*

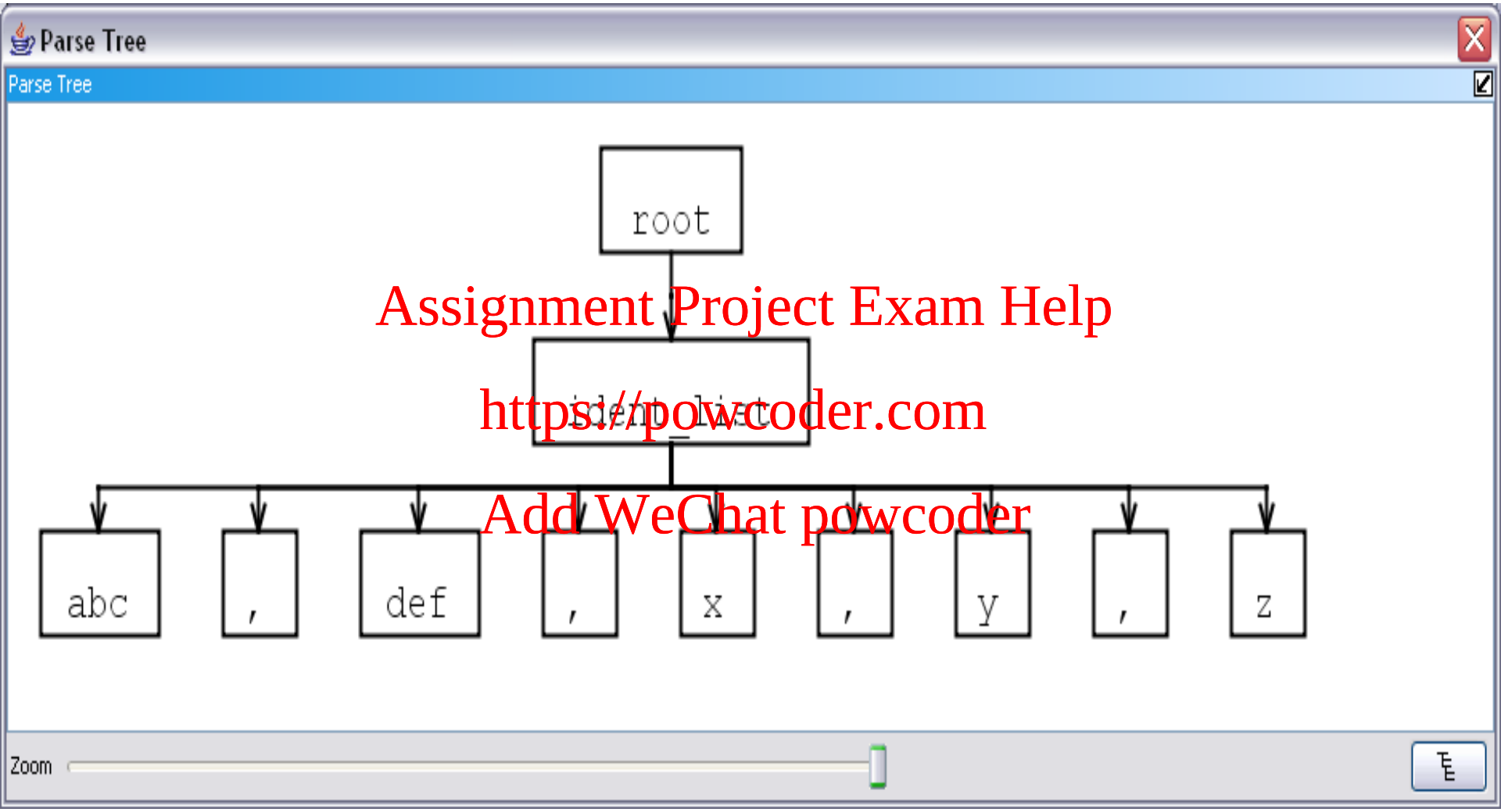
New grammar specifies the same language as the starting point, but the structure of the parse tree may change.

Parse Tree

Parse Tree



Zoom



Example: expression grammar with precedence and associativity

expr ::= *term* | *expr add_op term*

term ::= *factor* | *term mult_op factor*

factor ::= ident | number | - *factor* | (*expr*)

add_op ::= + | -

mult_op ::= * | /

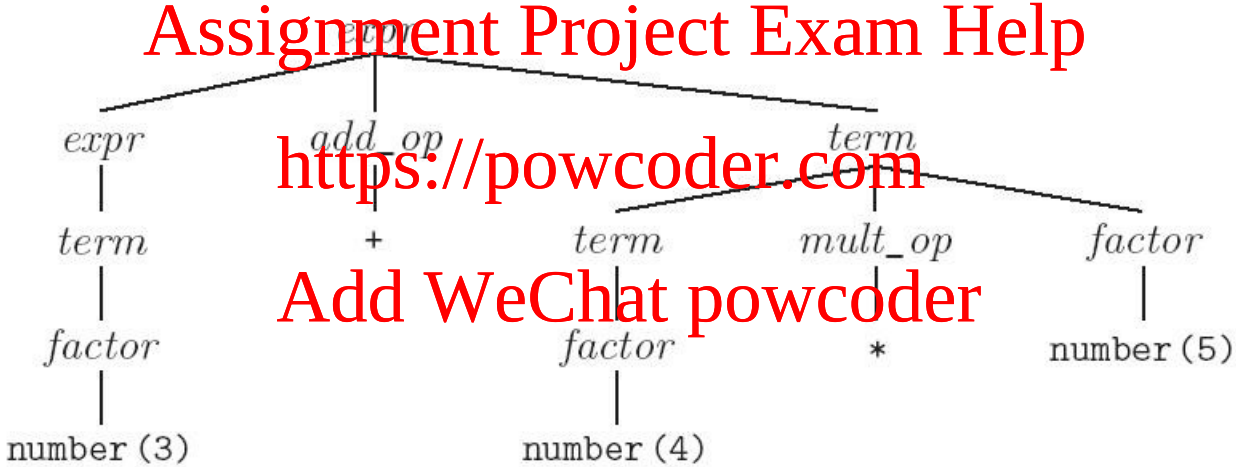
Operators are left associative:

10-4-3 = (10-4)-3

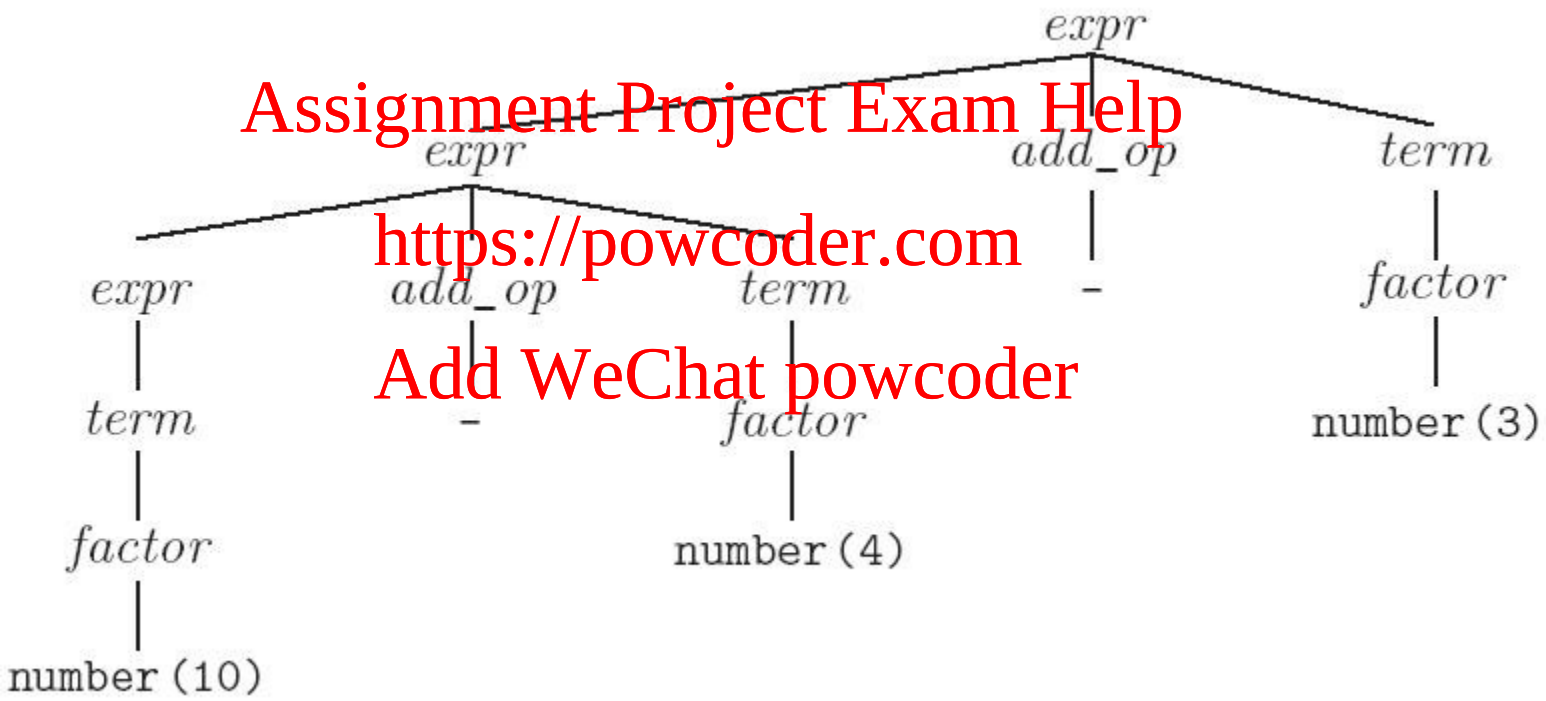
A mult_op has higher precedence than an add_op:

3+4*5 = 3+(4*5)

Parse tree for 3+4*5



Parse tree for 10-4-3



Is this expression grammar LL(1)?

expr ::= *term* | *expr add_op term*

term ::= *factor* | *term mult_op factor*

factor ::= id | number | - *factor* | (*expr*)

add_op ::= + | -

mult_op ::= * | /

Left recursive

expr ::= *term*
| *expr add_op term*
term ::= *factor* | *term mult_op factor*
factor ::= ident | number | - *factor* | (*expr*)
add_op ::= + | -
mult_op ::= * | /

FIRST(*expr*) = FIRST(*term*) ∪ FIRST(*expr*)

Grammar not LL(1)

Example

- Start with left recursive grammar for expressions

expr ::= number | *expr add_op* number

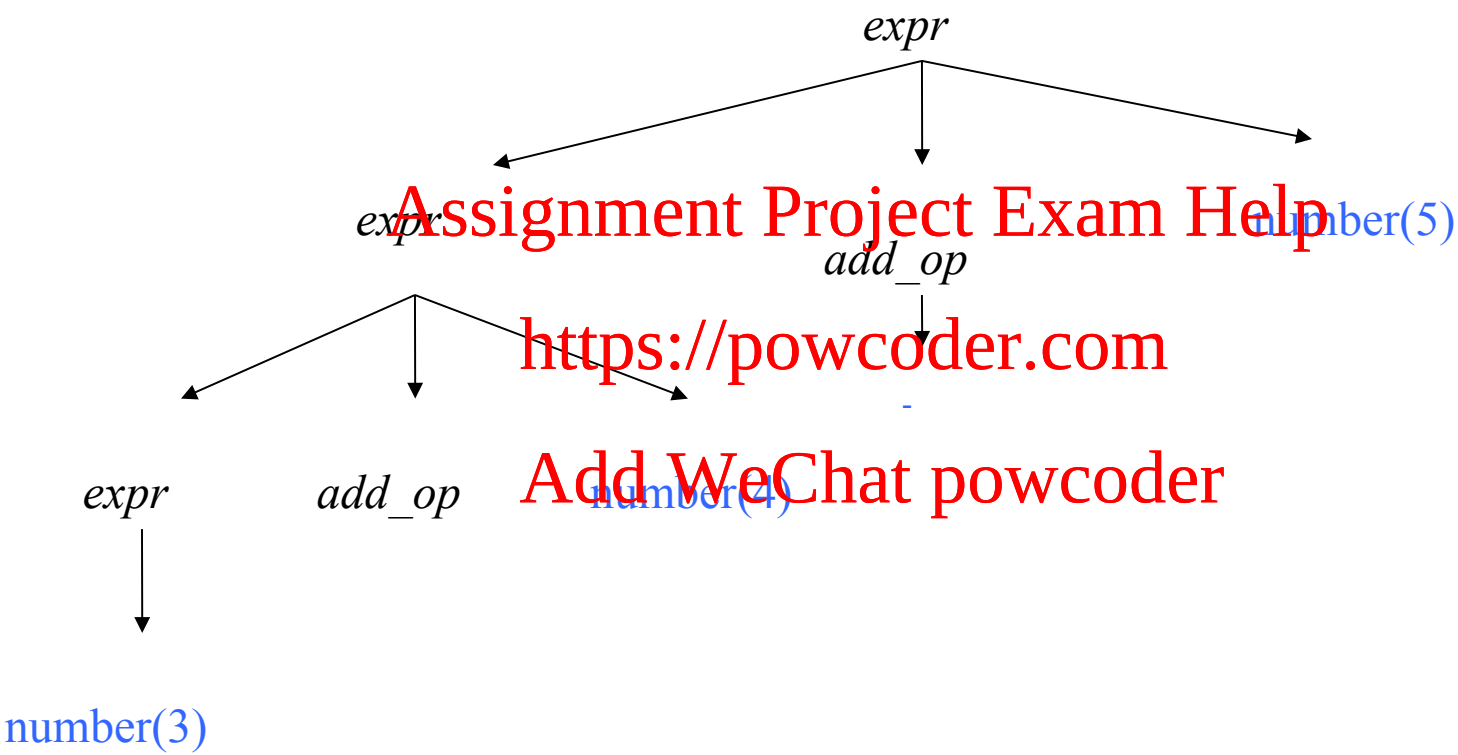
add_op ::= + | -

- Left recursion gives left-associative operators.
- Right recursion gives right-associative operators

expr ::= number

| number *right_assoc_op expr*

Parse tree for 4-3-5



Transformed example

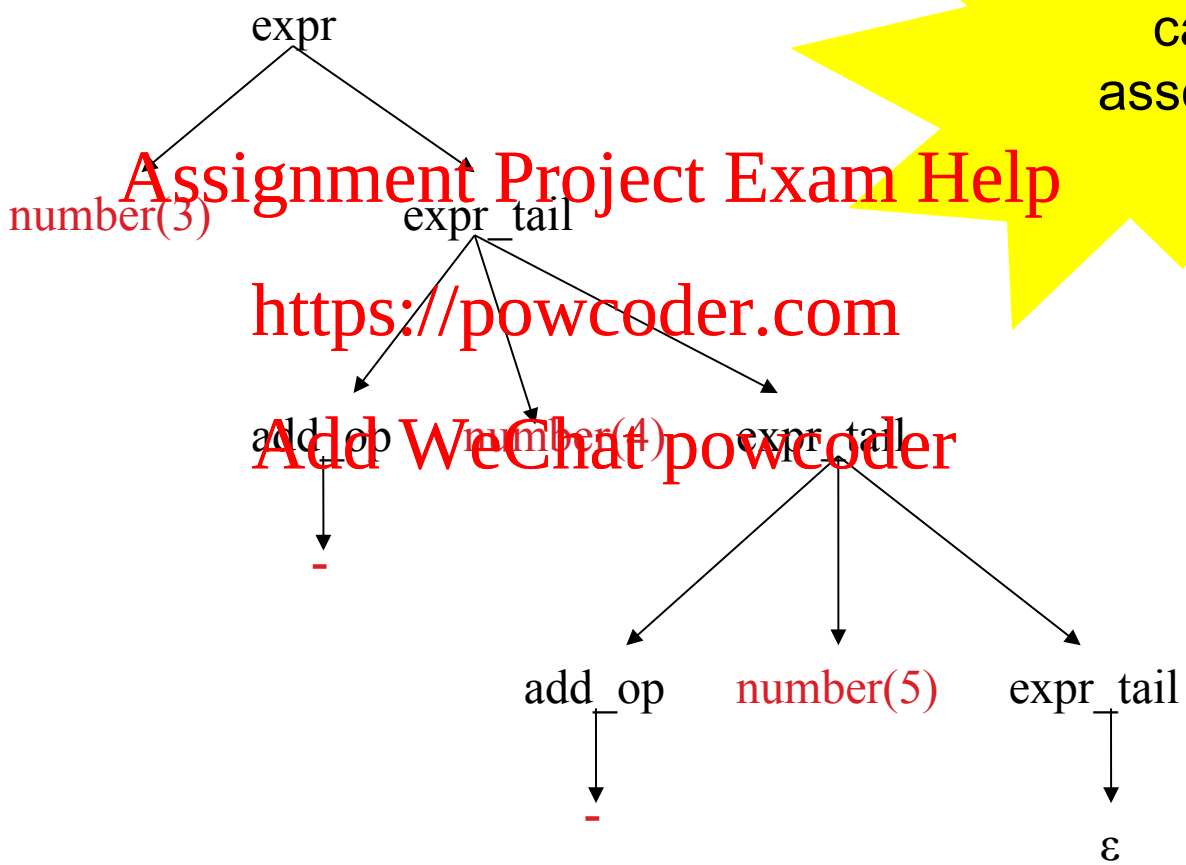
expr ::= number *expr_tail*

expr_tail ::= *add_op* number *expr_tail* | ε

add_op ::= + | -

Parse tree for 3-4-5

Fails to correctly capture associativity



Transformed again-- to EBNF

expr ::= number | *expr add_op* number

add_op := + | -

expr ::= number *expr_tail*

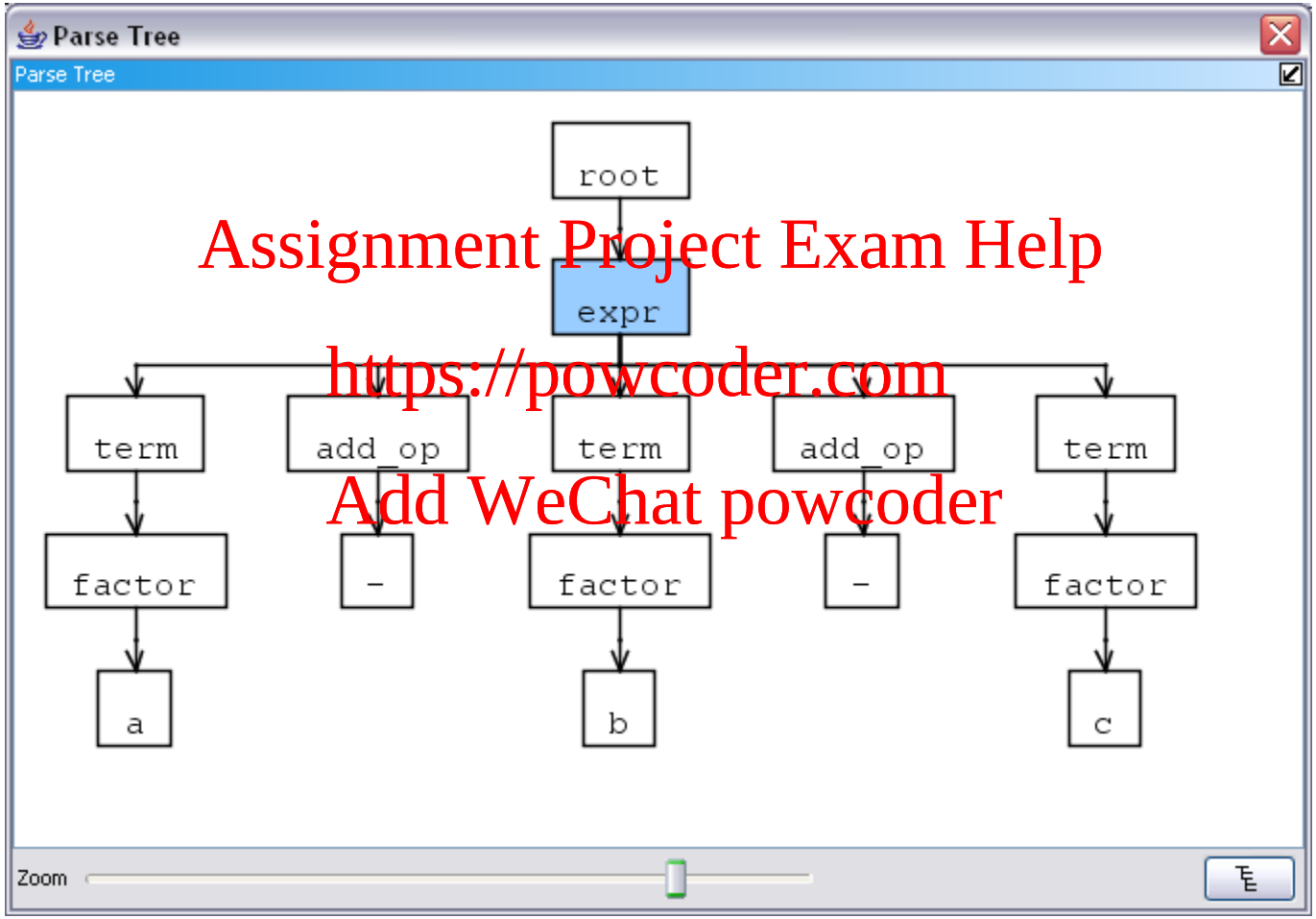
expr_tail ::= *add_op* number *expr_tail* | ε

add_op ::= + | -

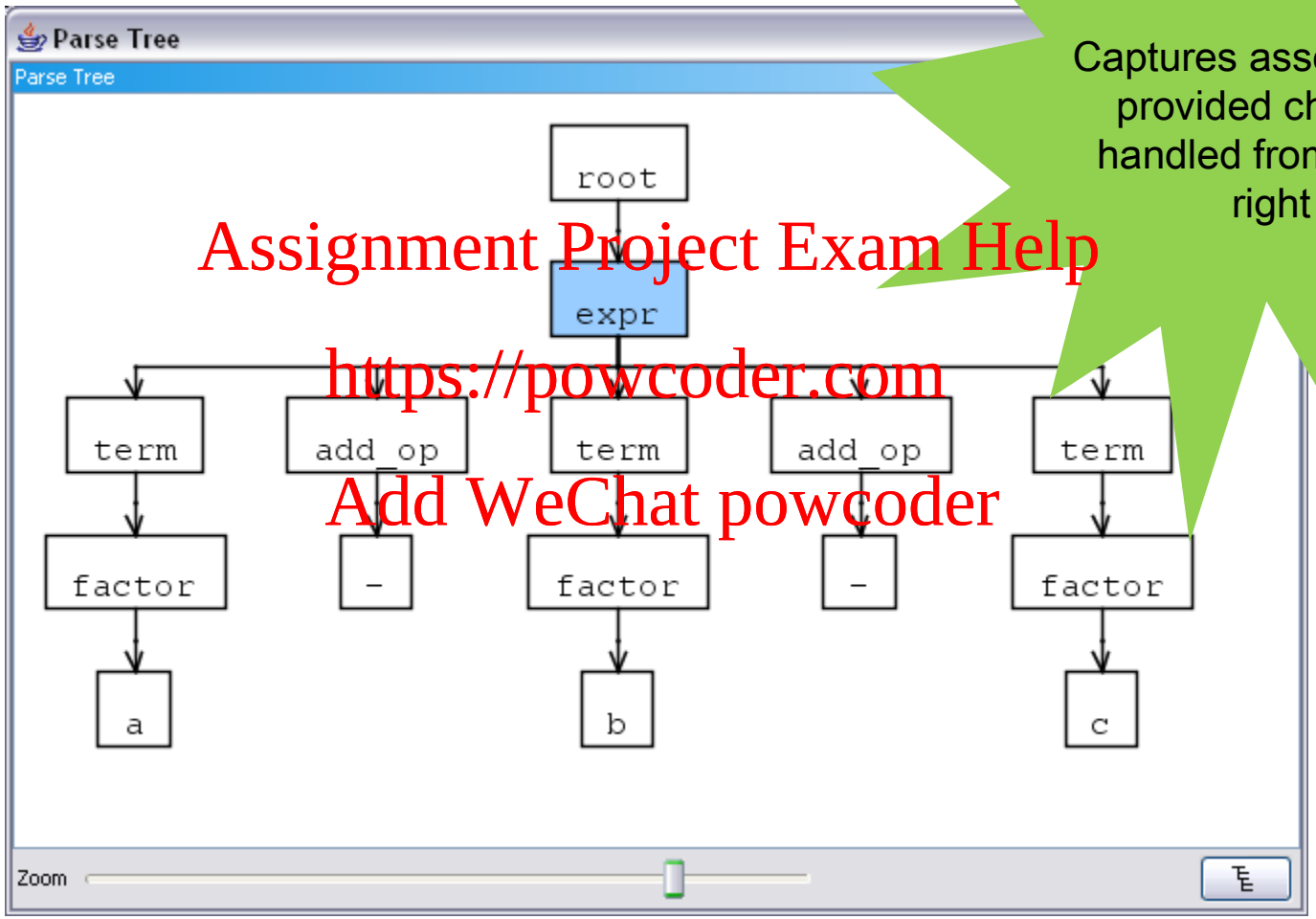
expr ::= number (*add_op* number)*

***add_op* ::= + | -**

Parse tree for 3-4-5



Parse tree for 3-4-5



Captures associativity provided children handled from left to right

- Thus the left-recursion in the grammar provided structural information beyond just the set of legal strings.

- We need to be somewhat careful when applying transformations.

- Note that bottom up parsers can handle left recursion just fine.

Another cause of non-LL(1) ness

Grammar with production of shape

$$A ::= \alpha\ \beta \mid \alpha\ \gamma$$

is not LL(1)

Common initial part can be factored out

$A ::= \alpha\beta \mid \alpha\gamma$

can be transformed using left factorization

$A ::= \alpha\ B$

$B ::= \beta \mid \gamma$

where B is a new non-terminal

We still need PREDICT(B ::= β) and PREDICT B ::= γ) to be disjoint

It is often convenient to replace

$$A ::= \alpha B$$
$$B ::= \beta \mid \gamma$$

with

$$A ::= \alpha (\beta \mid \gamma)$$

to avoid introducing a new symbol.

Example

stmt ::= ident := *expr* | ident (*arg_list*)

can be transformed to

stmt ::= ident *ident_stmt_tail*

ident_stmt_tail ::= := *expr* | (*arg_list*)

Or more compactly

stmt ::= ident (:= *expr* | (*arg_list*))

Example

stmt ::= ident *ident_stmt_tail*

ident_stmt_tail ::= := *expr* | (*arg_list*)

OK!

ε

Another example

expr ::= @*expr* . *field_name*

| @*expr* . *field_name* = *expr*

| @*expr* . *method_name* (ε | *expr* (, *expr*)*)

| … .

field_name ::= ident

method_name ::= ident

ε

Another example (2)

expr ::= @*expr* . *field_name*

| @*expr* . *field_name* = *expr*

| @*expr* . *method_name* (ε | *expr* (, *expr*)*)

| … .

field_name ::= ident

method_name ::= ident

ε

Another example (3)

expr ::= @*expr* . *field_name*
| @*expr* . *field_name* = *expr*
| @*expr* . *method_name* (ε | *expr* (, *expr*)*)
| … .

Problem!

field_name ::= ident
method_name ::= ident

ε

Another example (4)

substitute ident for field_name and method_name

expr ::= @*expr* . *ident*

| @*expr* . *ident* = *expr*

| @*expr* . *ident* (ε | *expr* (, *expr*)*)

| … .

ε

Another example (5)

expr ::= @*expr* . *ident*

| @*expr* . *ident* = *expr*

| @*expr* . *ident* (| *expr* (, *expr*)*)

| … .

ε

Another example (5)

expr ::= @*expr* . *ident*

| @*expr* . *ident* = *expr*

| @*expr* . *ident* (ε | *expr* (, *expr*)*)

| … .

or

expr ::= @*expr* . *ident*

(ε | = *expr* / (ε | *expr* (, *expr*)*))

ε

Another example (6)

expr ::= @*expr* . *ident*

(ε | *expr* / *expr* | *expr* (*expr*)*)

Because of the ε we can't be sure this will work without knowing FOLLOW(*expr*) in entire grammar

Transformations do not always work

- Note that eliminating left recursion and common prefixes does NOT necessarily make a grammar LL
- There are infinitely many non-LL LANGUAGES, and the mechanical transformations work on most of them just fine
- The few that arise in practice, however, can generally be handled with kludges

Systematic construction of parsers

- Given an LL(1) grammar, we can systematically construct a **recursive descent parser.**
- Tools exist that will do this automatically, we'll do it by hand.
- Our first parser will only recognize whether or not a sentence is legal.
- Later we'll extend the approach to construct an AST (abstract syntax tree)

Parser construction

- Compute the PREDICT sets for each production
- Rewrite the grammar so that each non-terminal is on the left side of exactly one production.
- For example, change

  A::= B

  A::= C

  to

  A::= B | C

- For each non-terminal A, we will have a method void A().
- For each possible shape of the rhs of a production

A ::= σ

we have a rule for constructing the code for the method A();

$\sigma_1 \mid \sigma_2 \mid \ldots \sigma_n$ (where none are ε)

becomes code fragment

if (token $\in$ PREDICT(lhs ::= σ_1)) parse σ_1 ;
else if (token $\in$ PREDICT(lhs ::= σ_2))
parse σ_2;

...
else if (token $\in$ PREDICT(lhs ::= σ_n))
parse σ_n
else error()

Notation: lhs = left hand side

$\sigma_1 \mid \sigma_2 \mid \ldots \sigma_n \mid \varepsilon$

becomes

if (token $\in$ PREDICT(lhs ::= σ_1)) parse σ_1 ;

else if (token $\in$ PREDICT(lhs ::= σ_2))

parse σ_2;

...

else if (token $\in$ PREDICT(lhs ::= σ_n))

parse σ_n ;

else just return //this branch matches ε

$\sigma_1 \sigma_2 \ldots \sigma_n$

becomes

parse σ_1;

parse σ_2;

…

parse σ_n

σ^*

becomes

while(token $\in$ FIRST(σ)) { parse σ; }

A (where A is a non-terminal)

becomes

A();

(i.e. just invoke the method for A)

c (where a is a terminal symbol)

becomes

```
match(c);
```

where

```
match(c)
{   if ( current token is c )
    {  get next token from scanner;}
    else error();
}
```

It may be worthwhile to simplify the grammar before starting.

For example,

A ::= B| xC
C ::= D

could be simplified to

A ::= B | xD

Also, code can be simplified after the fact to eliminate redundant tests, etc.

Implementing a Parser in Java

expr ::= *term* ((+ | -) *term*)*

term ::= *factor* ((* | /) *factor*)*

factor ::= *intlit* | (*expr*)

We are given a Scanner and Token class

Class SimpleParser

```
public class SimpleParser
{
  Scanner scanner; //Token producer
  Token t;  //nextToken

  Parser(Scanner scanner) {
     this.scanner = scanner;
     t = scanner.nextToken();
  }
```

```
Token consume()
{
   t = scanner.getNext();
}
```



```
 void match(Kind kind){
    if( t.isKind(kind)){
            consume();
    } else  handle error
}
```

```
//      expr ::= term ( ( + | - )  term )*

public void expr()
  {
        term();
        while (t.isKind(PLUS) || t.isKind(MINUS))
        {
              if (t.isKind(PLUS))
              {       match(PLUS);
              }
              else if (t.isKind(MINUS))
              {
                      match(MINUS);
              }
              term();
        }
  }
```

term ::= *factor* ((* | /) *factor*)*

same shape as expr()—here we use switch statement

```
void term()
{
        factor();
        while (t.isKind(TIMES) || t.isKind(DIV))
         {
                switch (t.kind)
                {
                        case TIMES:
                                match(TIMES);
                                break;
                        case DIV:
                                match(DIV);
                                break;
                }
                factor();
        }
}
```

term ::= *factor* ((* | /) *factor*)*

same shape as expr()

```
void term()
{
        factor();
        while (t.isKind(TIMES) || t.isKind(DIV))
         {
                switch (t.kind)
                {
                        case TIMES:
                                consume(); // match(TIMES);
                                break;
                        case DIV:
                                consume (); //match(DIV);
                                break;
                }
                factor();
        }
}
```


Slightly simplified

term ::= *factor* ((* | /) *factor*)*

```
void term()
{
        factor();
        while (t.isKind(TIMES) || t.isKind(DIV))
         {
                switch (t.kind)
                {
                        case TIMES: case DIV:
                                consume();
                                break;
//                      case DIV:
//                              consume ();
//                              break;
                }
                factor();
        }
}
```


Simplified again

```
//  term ::= factor ( ( * | / )  factor )*

void term()
{
        factor();
        while (t.isKind(TIMES) || t.isKind(DIV))
         {
//              switch (t.kind)
//              {
//                  case TIMES: case DIV
                consume();
//                      break;
//              }
                factor();

        }
}
```


And again

factor ::= *int_lit* | **(** *expr* **)**

```
void factor() {
        if (t.isKind(INT_LIT)) {
                match(INT_LIT);
        } else if (t.isKind(LPAREN)) {
                match(LPAREN);
                expr();
                match(RPAREN);
        }
      else handle error
  }
}
```

- We showed how to systematically construct a recursive descent parser for a language specified by an LL(1) grammar.

- Next
  - Example to show that the parser really does implement the top down parsing algorithm
  - Error handling in recursive descent parsers

Recap of example

expr ::= *term* ((+ | -) *term*)*

term ::= *factor* ((* | /) *factor*)*

factor ::= *int_lit* | (*expr*)

```
public void expr()  // expr ::= term ( ( + | - )  term )*
   {
            term();
            while (t.isKind(PLUS) || t.isKind(MINUS)) { consume(); term(); }
            return;
   }

void term()  // term ::= factor ( ( * | / )  factor )*
   {
            factor();
            while (t.isKind(TIMES) || t.isKind(DIV)){ consume(); factor(); }
            return;
   }

void factor()  //factor ::= int_lit | ( expr )
   {
      if (t.isKind(INT_LIT)) { consume(); }
      else if (t.isKind(LPAREN))
      { consume(); expr(); match(RPAREN); }
      else error();
      return
   }
```

Sentence to parse:

3 - (4 * 5)

Tokens:

num_lit(3), minus, lparen,

num_lit(4), times, numlit(5),

rparen

expr	3 – (4 * 5)	expr
term ((+ | -) *term*)*	3 – (4 * 5)	term
factor ((* | /) *factor*)* ((+ | -) *term*)*	3 – (4 * 5)	factor
int_lit ((* | /) *factor*)* ((+ | -) *term*)*	3 – (4 * 5)	if (intlit) consume() return from factor
((* | /) *factor*)* ((+ | -) *term*)*	– (4 * 5)	return from term
((+ | -) *term*)*	– (4 * 5)	if + or - consume
(+ | -) *term* ((+ | -) *term*)*	– (4 * 5)	
term ((+ | -) *term*)*	(4 * 5)	term
factor ((* | /) *factor*)* ((+ | -) *term*)*	(4 * 5)	factor
(*expr*) ((* | /) *factor*)* ((+ | -) *term*)*	(4 * 5)	if (consume

expr) ((* | /) *factor*)* ((+ | -) *term*)*	4 * 5)	expr
term ((+ | -) *term*)*) ((* | /) *factor*)* ((+ | -) *term*)*	4 * 5)	term
factor ((* | /) *factor*)* ((+ | -) *term*)*) ((* | /) *factor*)* ((+ | -) *term*)*	4 * 5)	factor
int_lit ((* | /) *factor*)* ((+ | -) *term*)*) ((* | /) *factor*)* ((+ | -) *term*)*	4 * 5)	if intlit consume return from factor
((* | /) *factor*)* ((+ | -) *term*)*) ((* | /) *factor*)* ((+ | -) *term*)*	* 5)	while * or /
* *factor* ((* | /) *factor*)* ((+ | -) *term*)*) ((* | /) *factor*)* ((+ | -) *term*)*	* 5)	consume

factor ((* | /) *factor*)* ((+ | -) *term*)*) ((* | /) *factor*)*((+ | -) *term*)**factor* ((* | /) *factor*)* ((+ | -) *term*)*) ((* | /) *factor*)* ((+ | -) *term*)*	5)	factor
int_lit ((* | /) *factor*)* ((+ | -) *term*)*) ((* | /) *factor*)* ((+ | -) *term*)*	5)	if (intlit) consume return from factor
((* | /) *factor*)* ((+ | -) *term*)*) ((* | /) *factor*)* ((+ | -) *term*)*	)	not * or / , terminate while loop and return from term
((+ | -) *term*)*) ((* | /) *factor*)* ((+ | -) *term*)*	)	not + or – terminate whileloop and return from expr
) ((* | /) *factor*)* ((+ | -) *term*)*	)	match) return from factor
((* | /) *factor*)* ((+ | -) *term*)*	eof	not * or / , end while return from term
((+ | -) *term*)*		not + or – , terminate while loop and return from expr

- Syntax Errors
  - Read Scott section 2.3.5 (in the online supplement) through Exception based errors
  - We will discuss several approaches
    - Halting
    - Syntax error recovery
      - Panic mode recovery
      - Phrase level recovery
      - Context specific
      - Exception based

  - http://Scott 4e Online Supplement

- Halting
  - When an error is detected, just stop and print message
  - This is easy for the compiler implementer, but inconvenient for the programmer using the compiler
- Syntax error recovery
  - compiler continues looking for errors
  - high quality error recovery essential for production compilers
  - **Problem: how to avoid cascading errors**

- Panic mode recovery
  - Define small set of safe symbols that delimit “clean” points in the input
  - When an error occurs, delete input tokens until a safe symbol is encountered
  - Return from subroutines until the parser is in a context where the new symbol might occur
  - Tends to generate lots of spurious errors in most languages

- Phrase level recovery
  - Use different sets of safe symbols in different contexts
  - When a parsing routine for non-terminal A discovers an error at its beginning, it deletes tokens until it find one that is in FIRST(A) and proceeds, or a member of FOLLOW(A) and returns

```
void factor()  //factor ::= int_lit | ( expr )
  {   if (! (t.isKind(INT_LIT) || t.isKind(LPAREN))
      {  report error
         do { t = s.next();}
         while ( t not in FIRST(factor) && t not in
                 FOLLOW(factor)
      }
         if (t.isKind(INT_LIT)) { consume(); }
    else if (t.isKind(LPAREN))
    { consume(); expr(); match(RPAREN); }
    //else error();
    return();
  }
```

```
void factor()  //factor ::= int_
  {   if (! (t.isKind(INT_LIT) ||
      {  report error
          do { t = s.next();}
          while ( t not in FIRST
                  FOLLOW(
        }
          if (t.isKind(INT_LIT)) { co ume(); }
    else if (t.isKind(LPAREN))
    { consume(); expr(); match(RPAREN); }
    //else error();
    return();
  }
```

if not the expected token, match just reports the error and returns, effectively inserting the expected token

```
void factor()  //factor ::= int_lit | ( expr )
  {   if (! (t.isKind(INT_LIT) || t.isKind(LPAREN))
      {  report error
         do { t = s.next();}
         while ( t not in FIRST(factor) && t not in
                 FOLLOW(factor)
      }
         if (t.isKind(INT_LIT)) { consume(); }
    else if (t.isKind(LPAREN))
    { consume(); expr(); match(RPAREN);
    //else error();
    else return();
  }
```


This example isn't very interesting—all tokens except (are in FOLLOW(factor)

More general description of phrase level recovery from Scott

```
procedure foo
    if (input_token ∉ FIRST(foo)) and (ε ∉ FIRST(foo))
        report_error                    -- print message for the user
        repeat
            delete_token
        until input_token ∈ FIRST(foo) ∪ FOLLOW(foo) ∪ {$}
    case input_token of
        . . . : . . .
        . . . : . . .                   -- valid starting tokens
        . . . : . . .
        otherwise return                -- error or foo ⟶ ε
```


If foo →ε, this approach tends to PREDICT ε and return when it should detect an error—

More general description of phrase level recovery from Scott

```
procedure foo
    if (input_token ∉ FIRST(foo)) and (ε ∉ FIRST(foo))
        report_error                    -- print message for the user
        repeat
            delete_token
        until input_token ∈ FIRST(foo) ∪ FOLLOW(foo) ∪ {EOF}
    case input_token of
        ...: ...
        ...: ...                        -- valid starting tokens
        ...: ...
        otherwise return                -- error or foo ⟶ ε
```


A symbol may be in FOLLOW set because it can follow foo somewhere, but not in the particular context

- Context-specific look-ahead (Wirth ’76)
  - Tokens may follow A somewhere in a valid program (and thus be in FOLLOW(A)) but are not necessarily allowed at a particular place.
  - Let the follow set be context-dependent
    - Pass in FOLLOW set in recursive calls
    - Otherwise, the same as phrase-level recovery
  - Example
    - stmt ::= ident := expr ;
      - would pass SEMI when calling expr here,
    - factor ::= (expr)
      - would pass RPAREN when calling expr here,
  - Additional heuristic—don’t delete tokens that start major constructs that require matching tokens later (BEGIN, WHILE, etc)

- Exception-based error recovery
  - Choose small set of contexts to back-out to when an error occurs (for example the beginning of code to handle a declaration, or a statement)
  - Attach exception handler to blocks of code that should implement recovery
  - When error is detected—raise an exception (throw in Java)
  - The system unwinds the stack to find most recent block of code with a handler.
  - The handler can continue, or re-throw the exception.

Aside: Exceptions in Java

class SyntaxException extends Exception{}

- Can add fields for additional information, constructors, etc.

- When an exception occurs
  - the block is exited.

  - If there is no handler, the exception is propagated to enclosing block.
  - Propagation continues until the exception is caught.
  - If the exception reaches the outermost block without being caught, the program terminates.

```
void factor()  throws SyntaxException
 {
        if (t.isKind(INT_LIT)) {
                consume();
        }
    else if (t.isKind(LPAREN)) {
                consume();
            expr();
           match(RPAREN);
   }
   //else error();
   else throw new SyntaxException(…);
   }
```

```
public void expr()
  {  try{
            term();
            while (t.isKind(PLUS) || t.isKind(MINUS))
        {   consume();
             term();
        }
   }
  catch (SyntaxException e)
  {  while (t not EOF)
       { if (t in FIRST(expr) expr(); return;}
         else if (t in FOLLOW(expr); return;}
         else t = s.next();
       }
   }

  }
```

Tools for parser generation

- Lex and Yacc
- ANTLR
  - LL(*)
- LPG
  - LR(k), backtracking possible
- Xtext
  - based on ANTRL
  - creates simple eclipse IDE
- many, many more

ANTLR

- ANTLR (ANother Tool for Language Recognition)
- Reads a grammar file and generates a
  - lexer
  - parser
- Can generate the lexer and parser in several languages (maybe)
  - java, c/c++, c#, python, ruby, …
- Can include target language fragments in the grammar file that will be included in the generated parser.
- More info at www.antlr.org
- Accepts LL grammars, including LL(*) (unbounded lookahead)
  - Holds the entire input string in memory.
  - Approach only feasible recently, but necessary for IDEs such as eclipse.