Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

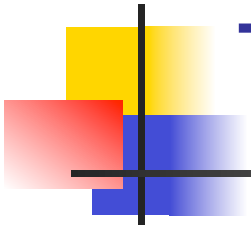
Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Sample Grammar

<expr> ::= <term> | <term> + <expr>
| <term> - <expr>

<term> ::= <factor> | <factor> * <term>
| <factor> / <term>

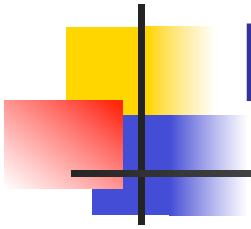
<factor> ::= <id> | (<expr>)



Tokens as OCaml Types

- + - * / ( ) <id>
- Becomes an OCaml datatype

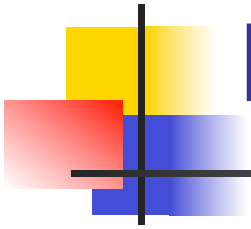
```
type token =
    Id_token of string
  | Left_parenthesis | Right_parenthesis
  | Times_token | Divide_token
  | Plus_token | Minus_token
```



Parse Trees as Datatypes

<expr> ::= <term> | <term> + <expr>
| <term> - <expr>

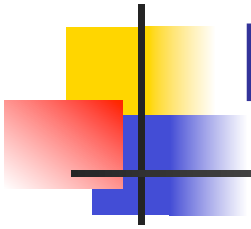
```
type expr =
    Term_as_Expr of term
  | Plus_Expr of (term * expr)
  | Minus_Expr of (term * expr)
```



Parse Trees as Datatypes

<term> ::= <factor> | <factor> * <term>
| <factor> / <term>

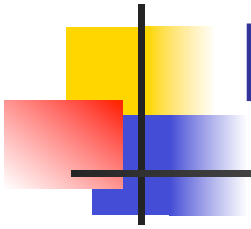
and term =
Factor_as_Term of factor
| Mult_Term of (factor * term)
| Div_Term of (factor * term)



Parse Trees as Datatypes

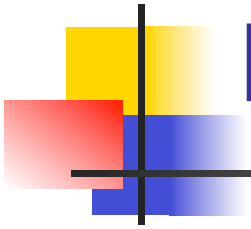
<factor> ::= <id> | (<expr>)

```
and factor =
    Id_as_Factor of string
  | Parenthesized_Expr_as_Factor of expr
```



Parsing Lists of Tokens

- Will create three mutually recursive functions:
  - expr : token list -> (expr * token list)
  - term : token list -> (term * token list)
  - factor : token list -> (factor * token list)
- Each parses what it can and gives back parse and remaining tokens



Parsing an Expression

<expr> ::= <term> [(+ | -) <expr>]

```
let rec expr tokens =
  (match term tokens
    with ( term_parse , tokens_after_term) ->
      (match tokens_after_term
        with( Plus_token  :: tokens_after_plus) ->
```

Parsing an Expression

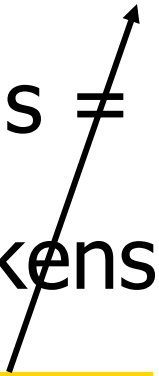
<expr> ::= <term> [(+ | -) <expr>]

```
let rec expr tokens =
  (match term tokens
    with ( term_parse , tokens_after_term) ->
      (match tokens_after_term
       with ( Plus_token  :: tokens_after_plus) ->
```

Parsing a Plus Expression

<expr> ::= <term> [(+ | -) <expr>]

```
let rec expr tokens =
  (match term tokens
    with ( term_parse , tokens_after_term) ->
      (match tokens_after_term
       with ( Plus_token  :: tokens_after_plus) ->
```



Parsing a Plus Expression

<expr> ::= <term> [(+ | -) <expr>]

```
let rec expr tokens =
  (match term tokens
    with ( term_parse , tokens_after_term) ->
      (match tokens_after_term
       with ( Plus_token  :: tokens_after_plus) ->
```

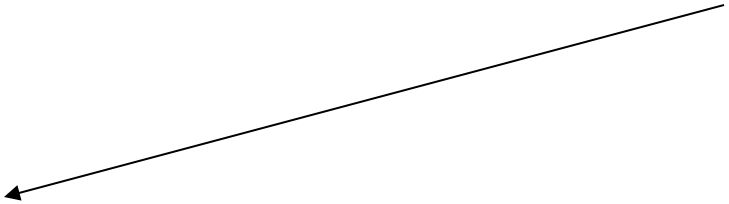
Parsing a Plus Expression

<expr> ::= <term> [(+ | -) <expr>]

```
let rec expr tokens =
  (match term tokens
    with ( term_parse , tokens_after_term) ->
      (match tokens_after_term
       with ( Plus_token  :: tokens_after_plus) ->
```

Parsing a Plus Expression

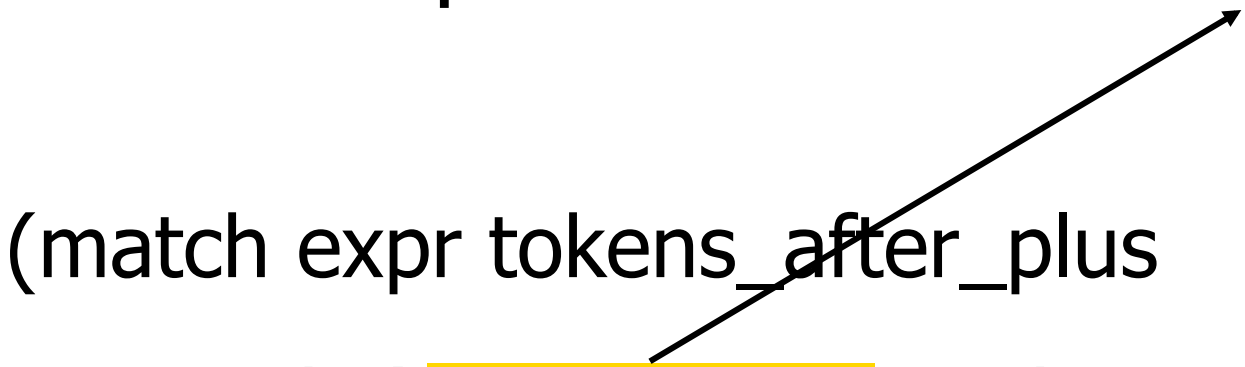
<expr> ::= <term> + <expr>



```
(match expr tokens_after_plus
 with ( expr_parse  , tokens_after_expr) ->
( Plus_Expr  ( term_parse  ,  expr_parse ),
 tokens_after_expr))
```

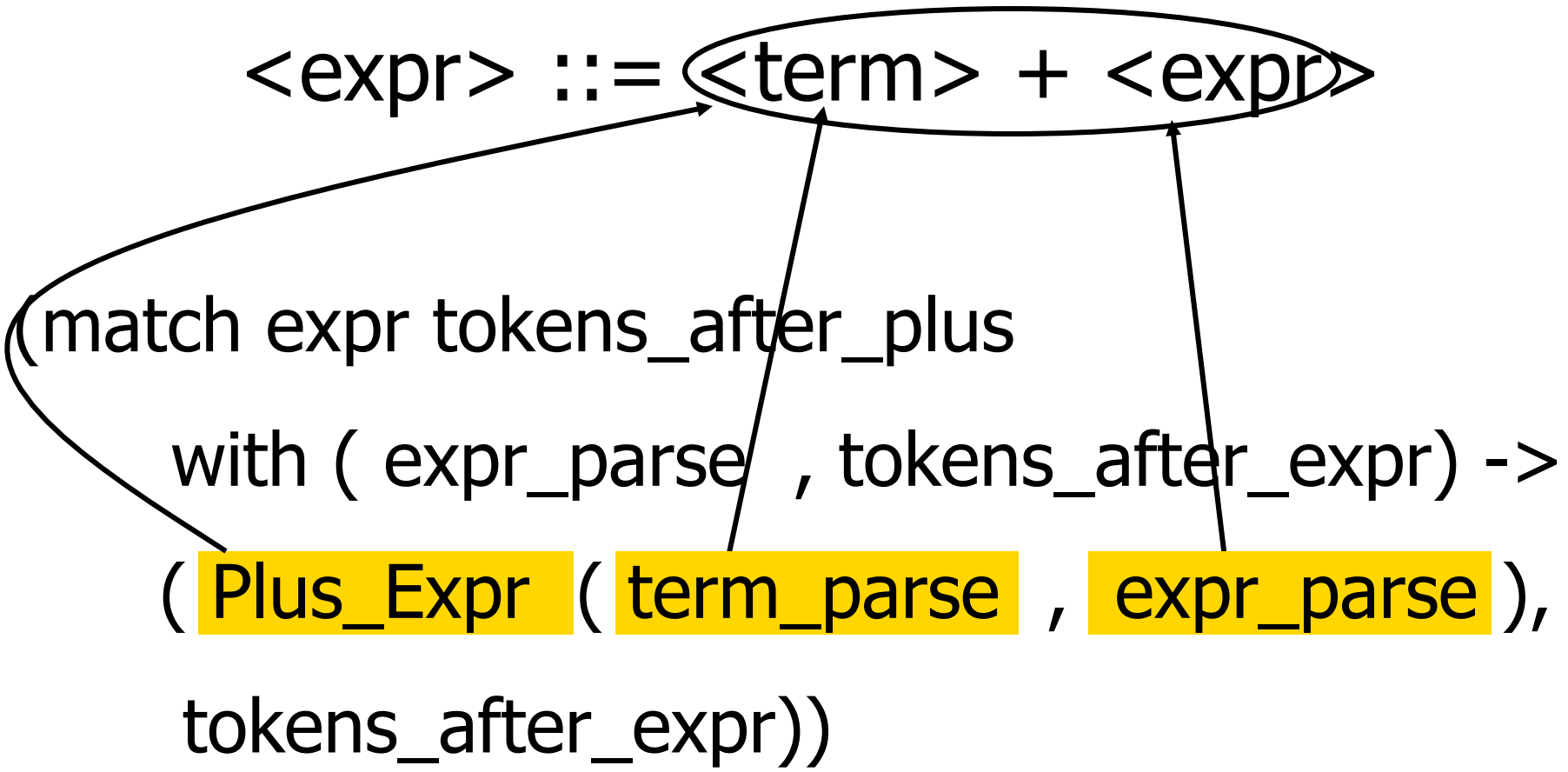
Parsing a Plus Expression

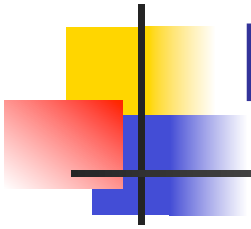
<expr> ::= <term> + <expr>



```
(match expr tokens_after_plus
      with ( expr_parse , tokens_after_expr) ->
     ( Plus_Expr  ( term_parse  ,  expr_parse ),
      tokens_after_expr))
```

Building Plus Expression Parse Tree



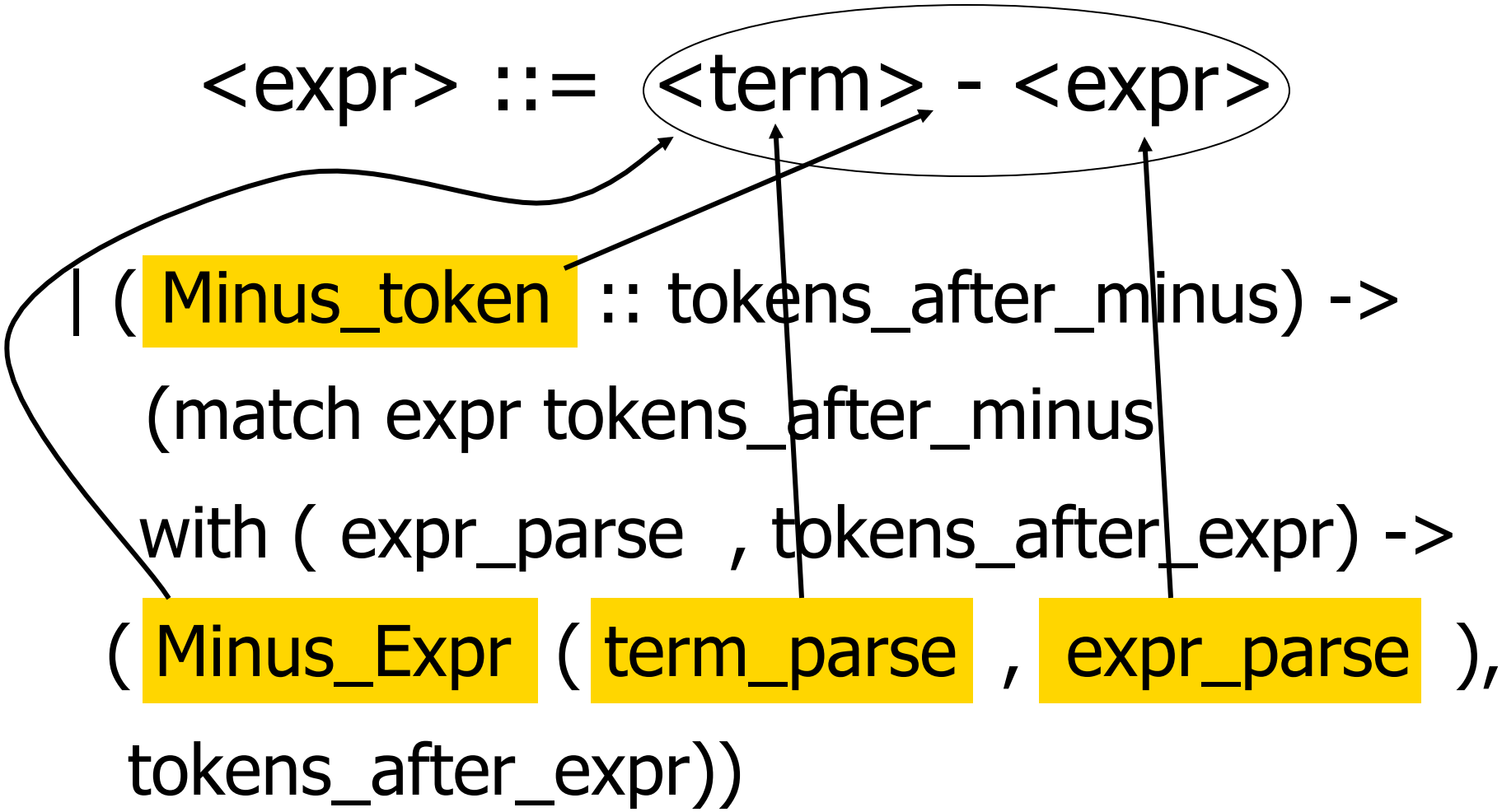


Parsing a Minus Expression

<expr> ::= <term> - <expr>

```
| ( Minus_token  :: tokens_after_minus) ->
   (match expr tokens_after_minus
   with ( expr_parse  , tokens_after_expr) ->
  ( Minus_Expr  ( term_parse  ,  expr_parse  ),
   tokens_after_expr))
```

Parsing a Minus Expression



Parsing an Expression as a Term

<expr> ::= <term>

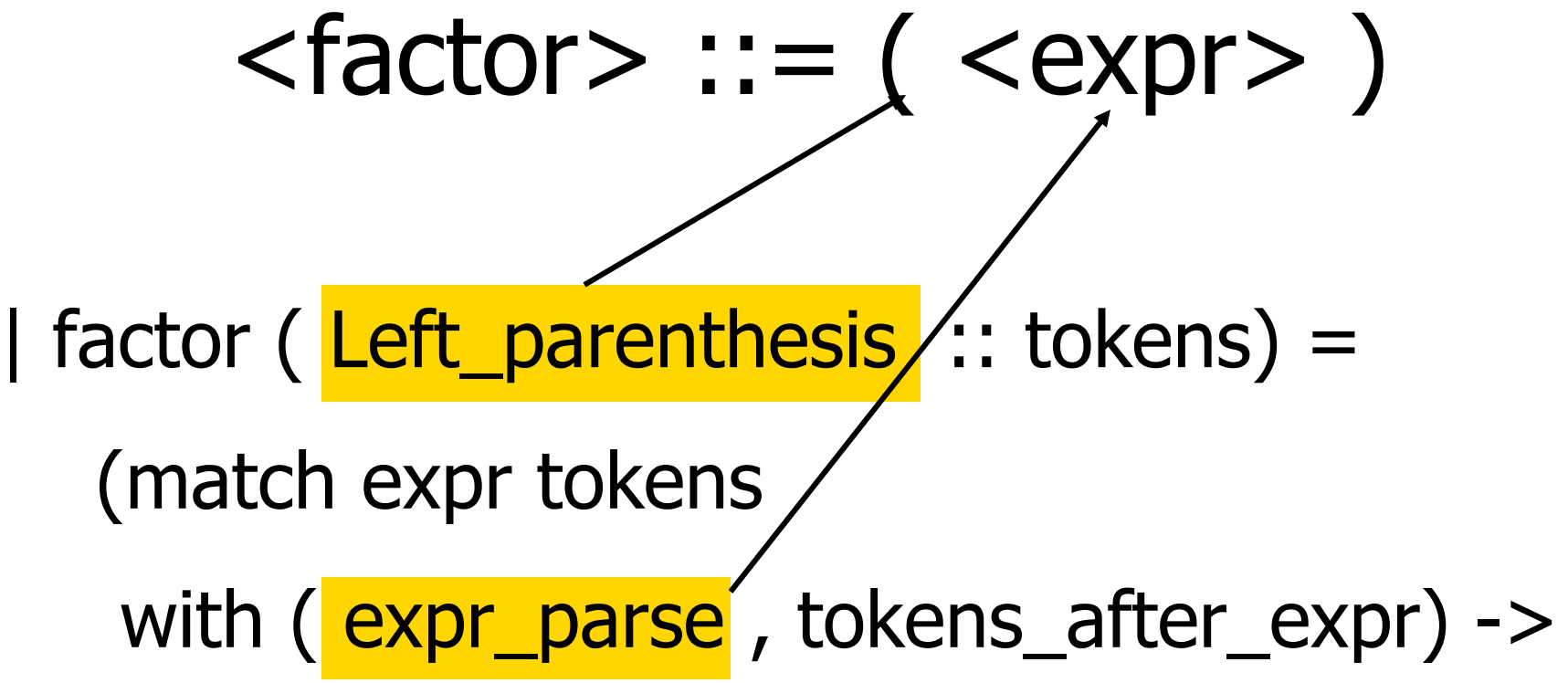
| _ -> (Term_as_Expr term_parse ,
tokens_after_term)))

- Code for **term** is same except for replacing addition with multiplication and subtraction with division

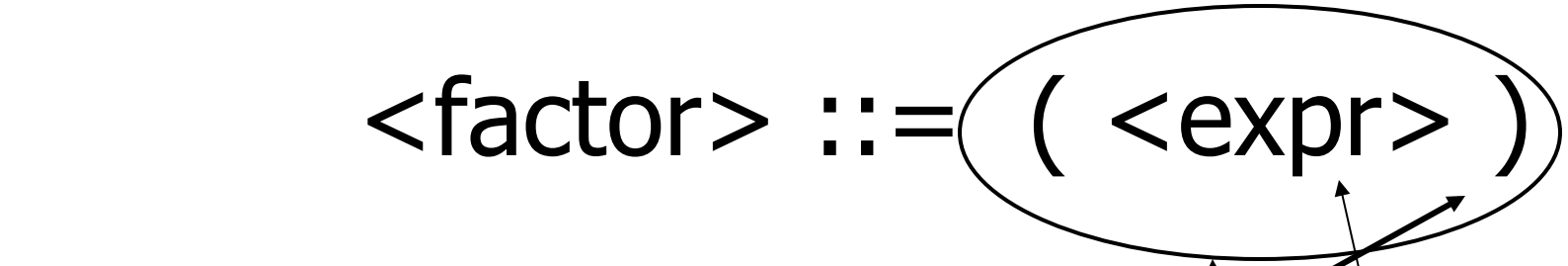
Parsing Factor as Id


```
and factor  tokens =
 (match tokens
  with (Id_token id_name :: tokens_after_id) =
   ( Id_as_Factor  id_name, tokens_after_id)
```

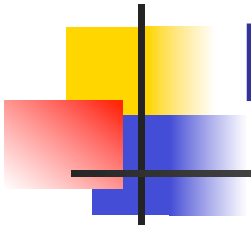
Parsing Factor as Parenthesized Expression



Parsing Factor as Parenthesized Expression




```
(match tokens_after_expr
with Right_parenthesis :: tokens_after_rparen ->
 ( Parenthesized_Expr_as_Factor  expr_parse ,
   tokens_after_rparen)
```



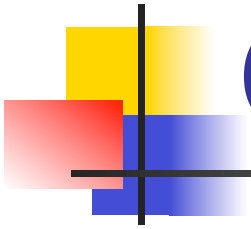
Error Cases

- What if no matching right parenthesis?

```
 | _ -> raise (Failure "No matching
rparen") ))
```

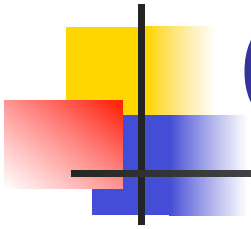
- What if no leading id or left parenthesis?

```
| _ -> raise (Failure "No id or lparen" ));;
```



(a + b) * c - d

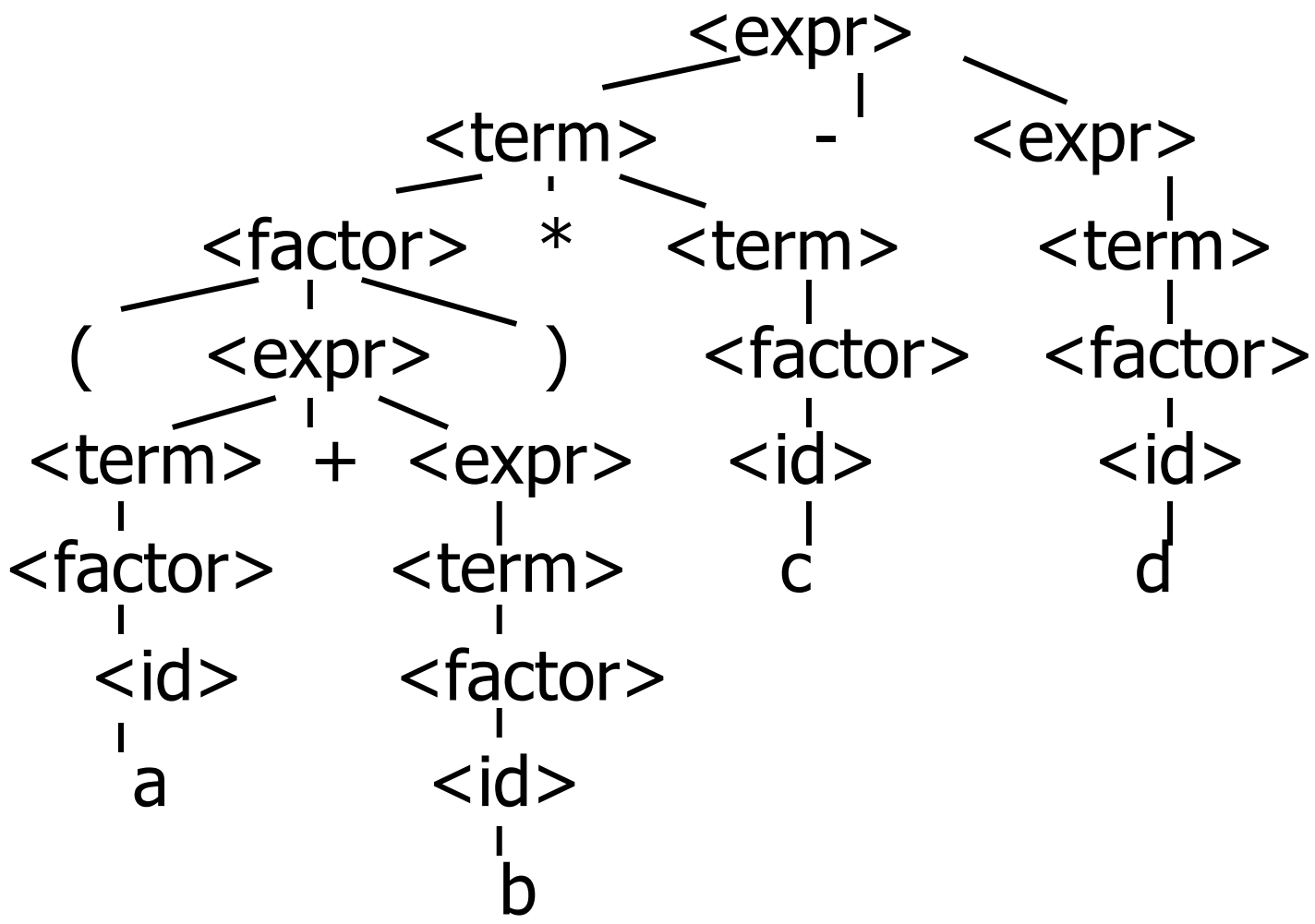
```
expr [Left_parenthesis; Id_token "a”;
  Plus_token; Id_token "b”;
  Right_parenthesis; Times_token;
  Id_token "c”; Minus_token;
  Id_token "d"];;
```

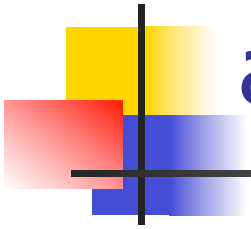


(a + b) * c - d

```
- : expr * token list =
(Minus_Expr
  (Mult_Term
    (Parenthesized_Expr_as_Factor
      (Plus_Expr
        (Factor_as_Term (Id_as_Factor "a"),
         Term_as_Expr (Factor_as_Term
   (Id_as_Factor "b")))),
     Factor_as_Term (Id_as_Factor "c")),
   Term_as_Expr (Factor_as_Term (Id_as_Factor
   "d"))),
 [])
```

(a + b) * c – d

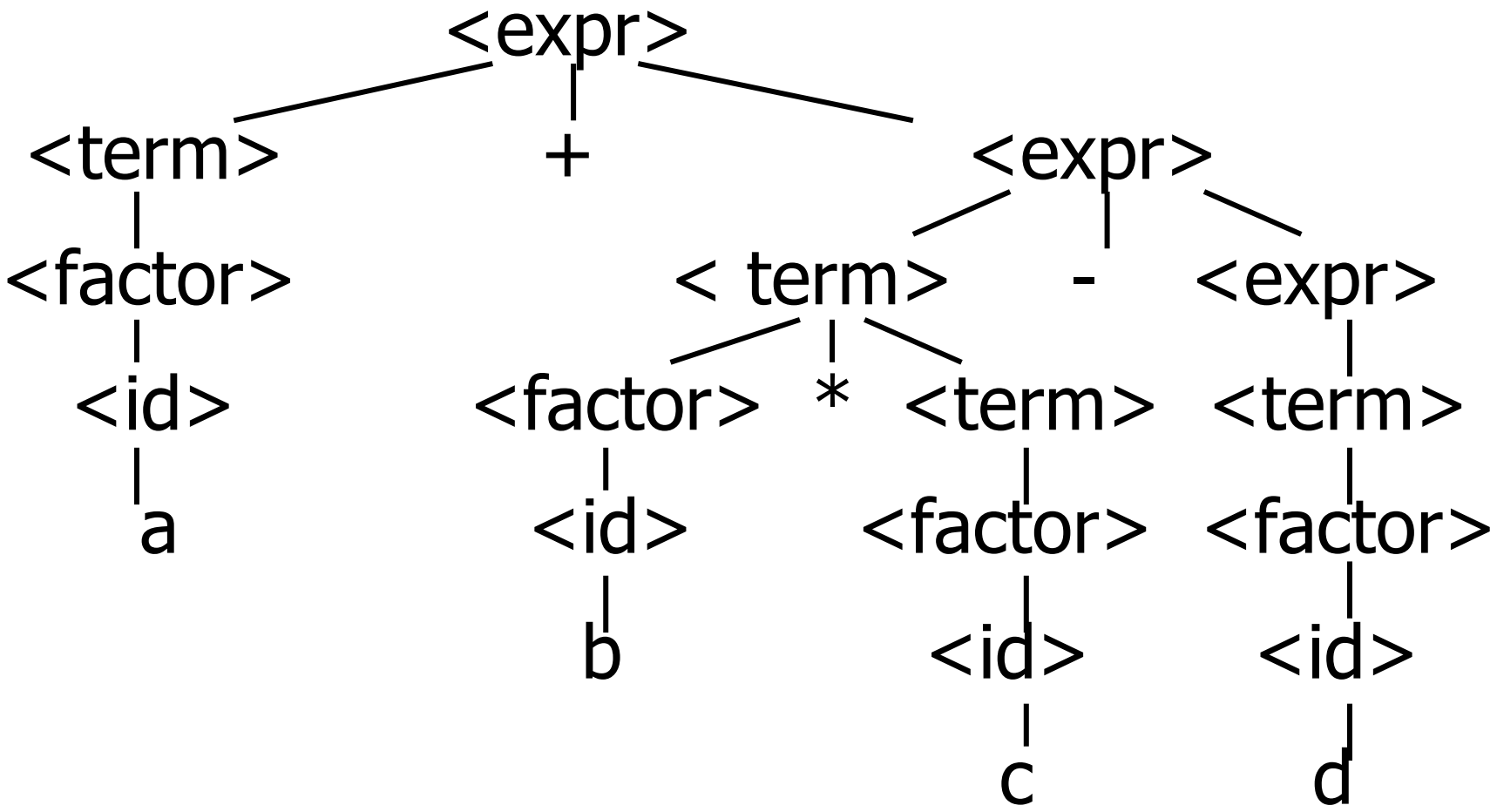

<expr>
<term>
-
<expr>
<factor>
*
<term>
<term>
(
<expr>
)
<factor>
<factor>
<term>
+
<expr>
<id>
<id>
<factor>
<term>
c
d
<id>
<factor>
a
<id>
b

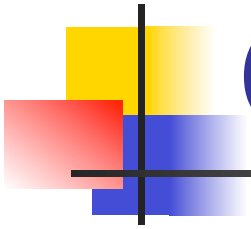


a + b * c – d

```
# expr [Id_token "a”; Plus_token; Id_token "b”;
   Times_token; Id_token "c”; Minus_token;
        Id_token "d"];;
- : expr * token list =
(Plus_Expr
  (Factor_as_Term (Id_as_Factor "a"),
   Minus_Expr
    (Mult_Term (Id_as_Factor "b", Factor_as_Term
   (Id_as_Factor "c")),
     Term_as_Expr (Factor_as_Term (Id_as_Factor
   "d")))),
 [])
```

a + b * c – d


<expr>
<term>
+
<expr>
<factor>
< term>
-
<expr>
<id>
<factor>
*
<term>
<term>
a
<id>
<factor>
<factor>
b
<id>
<id>
c
d

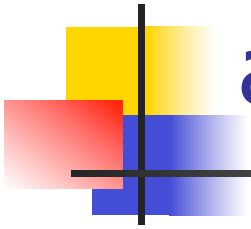


(a + b * c - d

```
# expr [Left_parenthesis; Id_token "a”;
Plus_token; Id_token "b”; Times_token;
Id_token "c”; Minus_token; Id_token "d"];;
```

Exception: Failure "No matching rparen".

Can’t parse because it was expecting a right parenthesis but it got to the end without finding one



a + b) * c - d *)

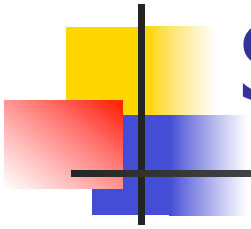
```
expr [Id_token "a"; Plus_token; Id_token "b";
   Right_parenthesis; Times_token; Id_token "c";
   Minus_token; Id_token "d"];;
- : expr * token list =
(Plus_Expr
  (Factor_as_Term (Id_as_Factor "a"),
   Term_as_Expr (Factor_as_Term (Id_as_Factor
   "b"))),
 [Right_parenthesis; Times_token; Id_token "c";
   Minus_token; Id_token "d"])
```

Parsing Whole String

- Q: How to guarantee whole string parses?
- A: Check returned tokens empty

```
let parse tokens  =
    match expr tokens
     with (expr_parse, []) -> expr_parse
     | _ -> raise (Failure “No parse");;
```

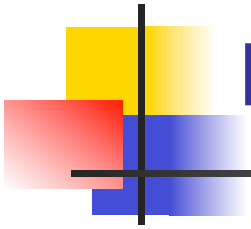
- Fixes <expr> as start symbol



Streams in Place of Lists

- More realistically, we don't want to create the entire list of tokens before we can start parsing
- We want to generate one token at a time and use it to make one step in parsing
- Will use (token * (unit -> token)) or (token * (unit -> token option))

  in place of token list



Problems for Recursive-Descent Parsing

- Left Recursion:

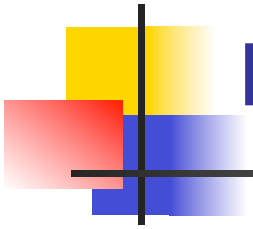
  A ::= Aw

  translates to a subroutine that loops forever
- Indirect Left Recursion:

  A ::= Bw

  B ::= Av

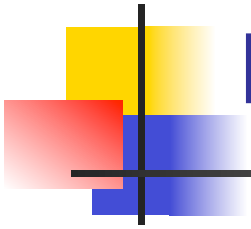
  causes the same problem



Problems for Recursive-Descent Parsing

- Parser must always be able to choose the next action based only only the very next token

- Pairwise Disjointedness Test: Can we always determine which rule (in the non-extended BNF) to choose based on just the first token



Pairwise Disjointedness Test

- For each rule

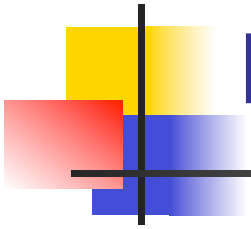
  A ::= *y*

  Calculate

  FIRST (*y*) =

  {a | *y* =>* a*w*} ∪ {ε | if y =>* ε}

- For each pair of rules A ::= *y* and A ::= *z*, require FIRST(*y*) ∩ FIRST(*z*) = { }



Example

Grammar:

<S> ::= <A> a <B> b

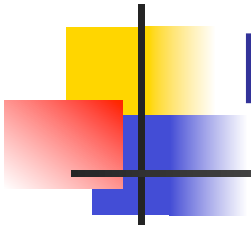
<A> ::= <A> b | b

<B> ::= a <B> | a

FIRST (<A> b) = {b}

FIRST (b) = {b}

Rules for <A> not pairwise disjoint



Eliminating Left Recursion

- Rewrite grammar to shift left recursion to right recursion
  - Changes associativity
- Given

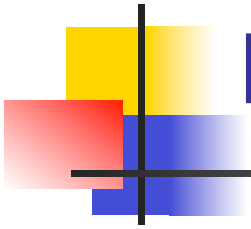
<expr> ::= <expr> + <term> and

<expr> ::= <term>

- Add new non-terminal <e> and replace above rules with

<expr> ::= <term><e>

<e> ::= + <term><e> | ε



Factoring Grammar

- Test too strong: Can't handle

  <expr> ::= <term> [(+ | -) <expr>]

- Answer: Add new non-terminal and replace above rules by

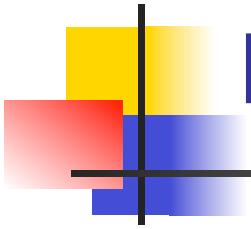
  <expr> ::= <term><e>

  <e> ::= + <term><e>

  <e> ::= - <term><e>

  <e> ::= ε

- You are delaying the decision point



Example

Both <A> and <B> have problems:

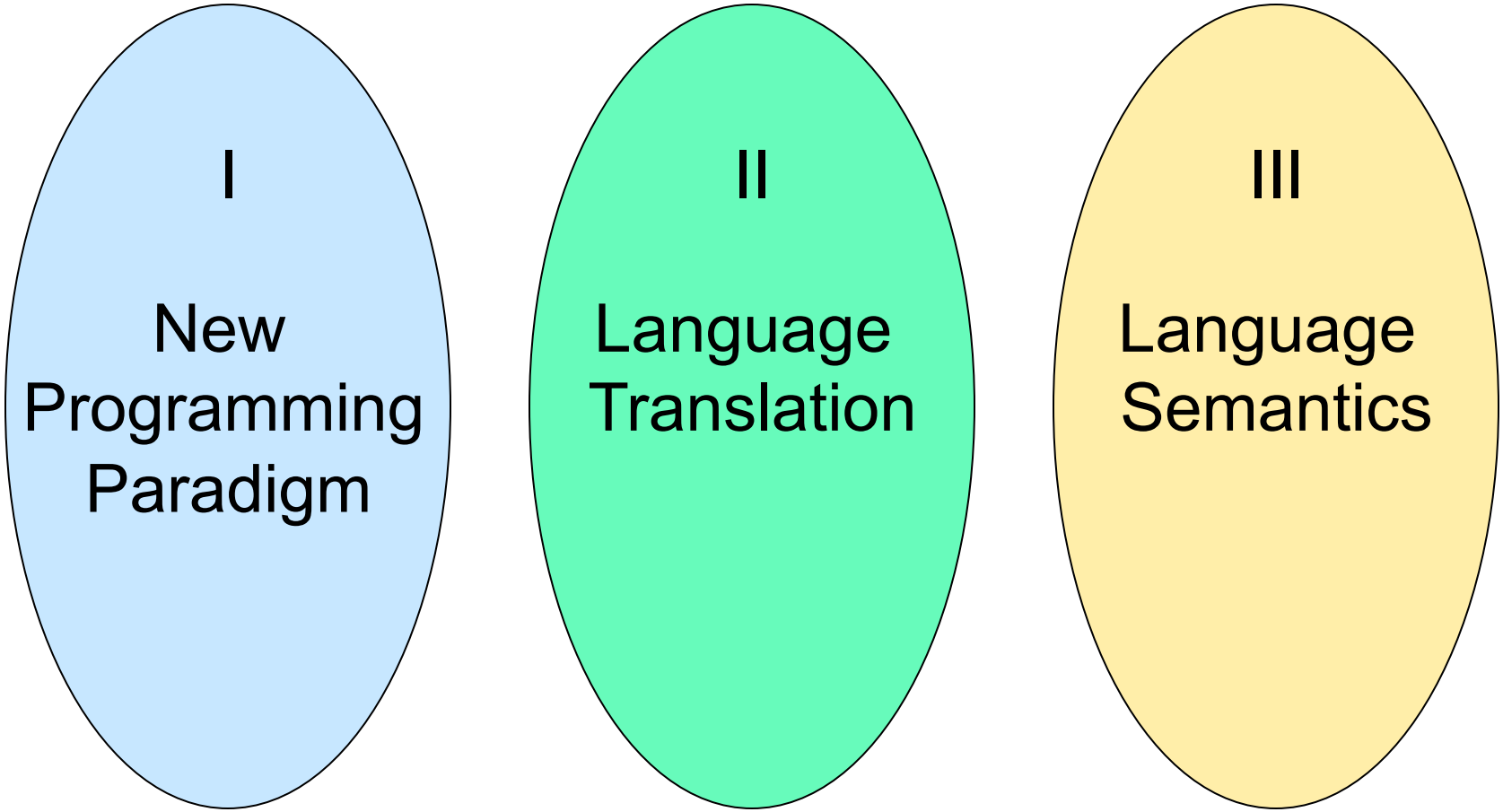
<S> ::= <A> a <B> b
<A> ::= <A> b | b
<B> ::= a <B> | a

Transform grammar to:

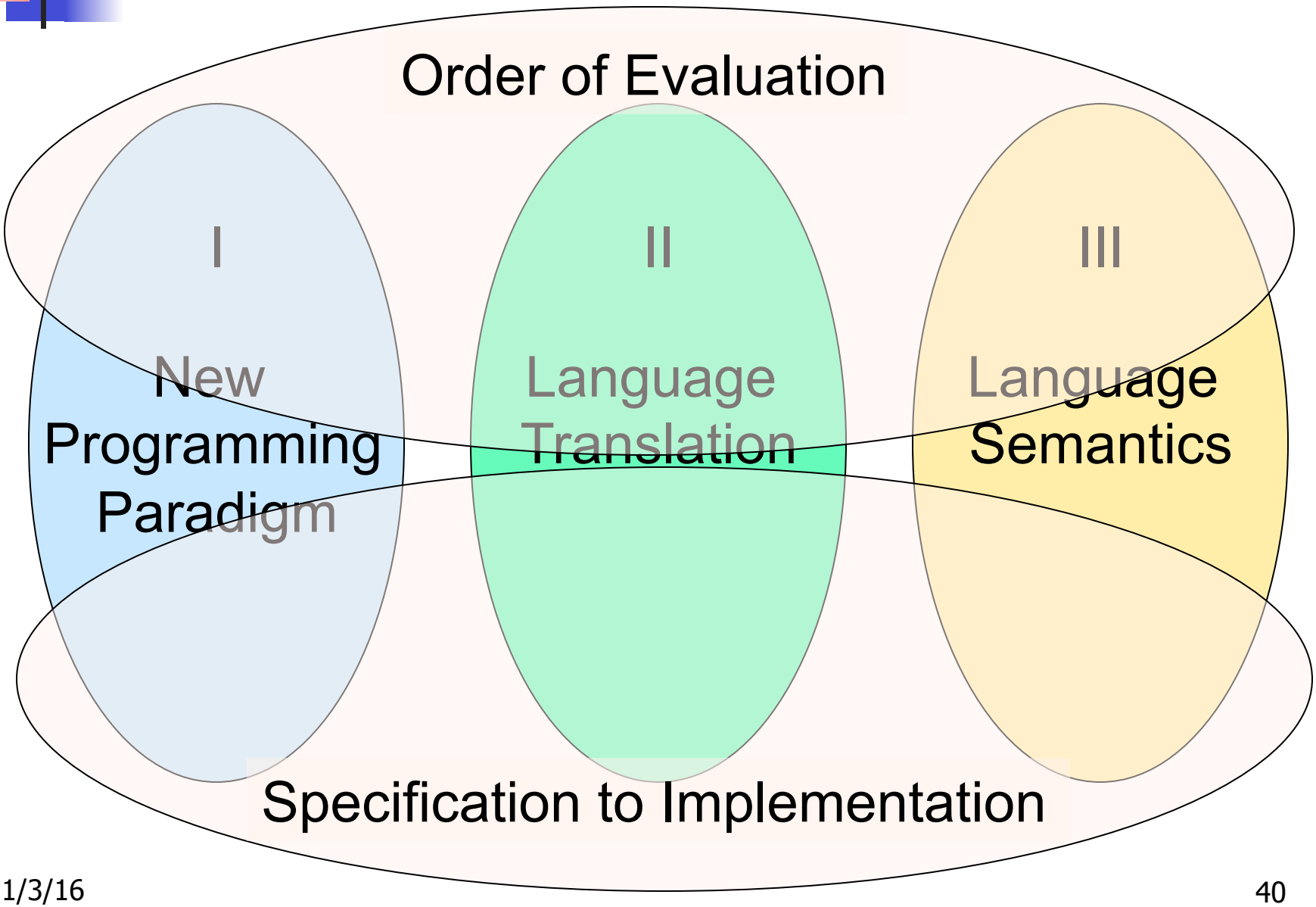
<S> ::= <A> a <B> b
<A> ::-= b<A1>
<A1> :: b<A1> | ε
<B> ::= a<B1>
<B1> ::= a<B1> | ε

Programming Languages & Compilers

Three Main Topics of the Course

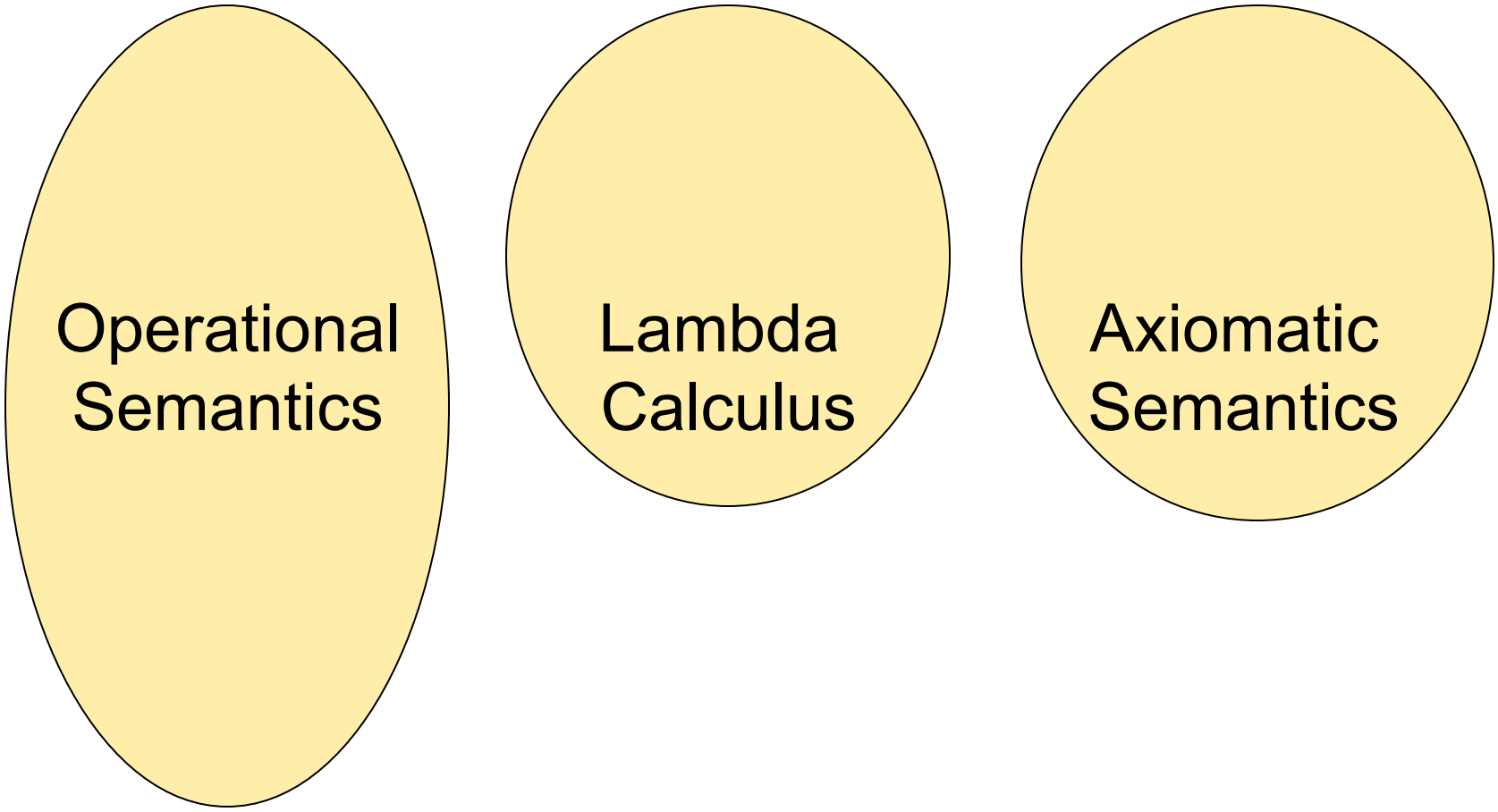


Programming Languages & Compilers

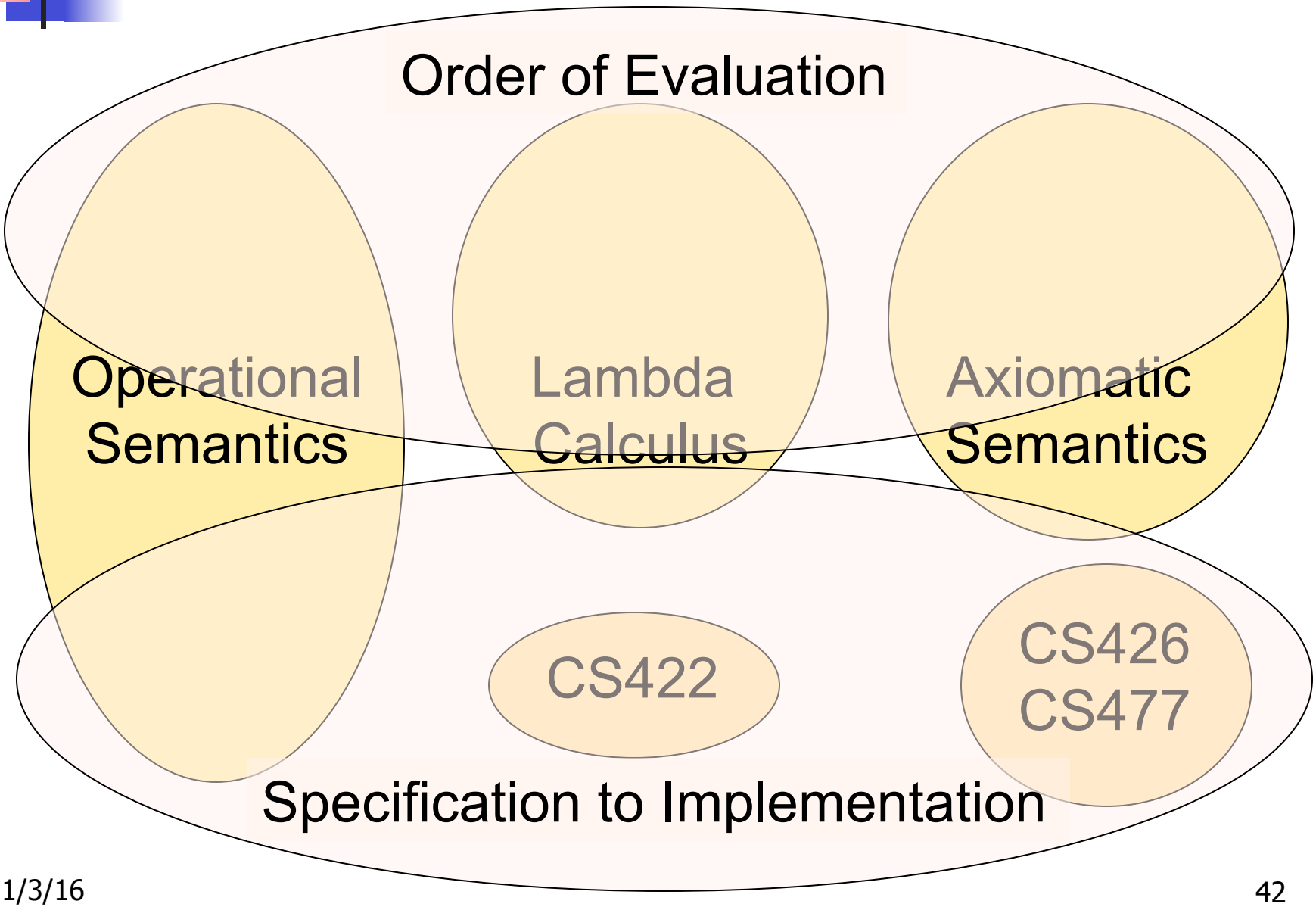


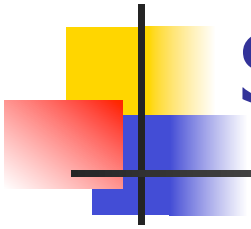
Programming Languages & Compilers

III : Language Semantics



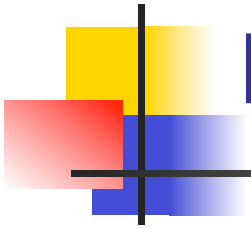
Programming Languages & Compilers





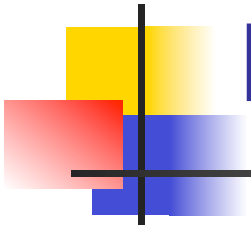
Semantics

- Expresses the meaning of syntax
- Static semantics
  - Meaning based only on the form of the expression without executing it
  - Usually restricted to type checking / type inference



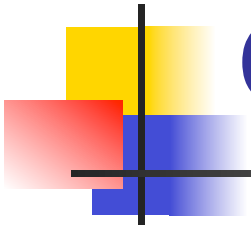
Dynamic semantics

- Method of describing meaning of executing a program
- Several different types:
  - Operational Semantics
  - Axiomatic Semantics
  - Denotational Semantics



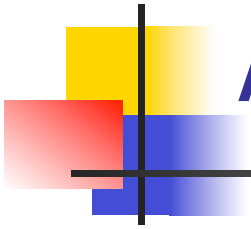
Dynamic Semantics

- Different languages better suited to different types of semantics
- Different types of semantics serve different purposes



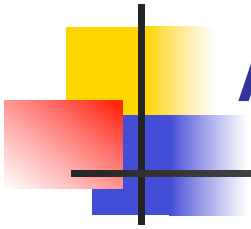
Operational Semantics

- Start with a simple notion of machine
- Describe how to execute (implement) programs of language on virtual machine, by describing how to execute each program statement (ie, following the *structure* of the program)
- Meaning of program is how its execution changes the state of the machine
- Useful as basis for implementations



Axiomatic Semantics

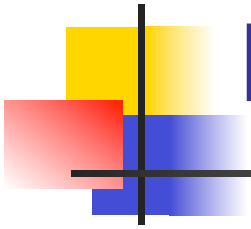
- Also called Floyd-Hoare Logic
- Based on formal logic (first order predicate calculus)
- Axiomatic Semantics is a logical system built from *axioms* and *inference rules*
- Mainly suited to simple imperative programming languages



Axiomatic Semantics

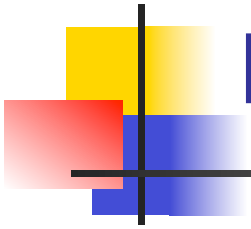
- Used to formally prove a property (*post-condition*) of the *state* (the values of the program variables) after the execution of program, assuming another property (*pre-condition*) of the state before execution
- Written :

  {Precondition} Program {Postcondition}
- Source of idea of *loop invariant*



Denotational Semantics

- Construct a function $\mathcal{M}$ assigning a mathematical meaning to each program construct
- Lambda calculus often used as the range of the meaning function
- Meaning function is compositional: meaning of construct built from meaning of parts
- Useful for proving properties of programs



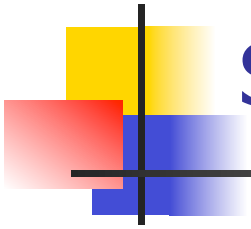
Natural Semantics

- Aka “Big Step Semantics”
- Provide value for a program by rules and derivations, similar to type derivations
- Rule conclusions look like

$$(C, m) \Downarrow m'$$

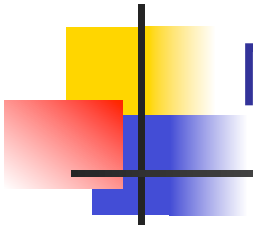
or

$$(E, m) \Downarrow v$$



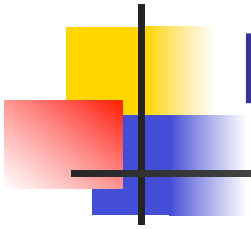
Simple Imperative Programming Language

- $I \in Identifiers$
- $N \in Numerals$
- B ::= true | false | B & B | B or B | not B | $E < E$ | $E = E$
- $E ::= N \mid I \mid E + E \mid E * E \mid E - E \mid - E$
- C::= skip | C;C | I ::= E | if B then C else C fi | while B do C od



Natural Semantics of Atomic Expressions

- Identifiers: $(I,m) \Downarrow m(I)$
- Numerals are values: $(N,m) \Downarrow N$
- Booleans: (true,*m*) ⇓ true
  (false ,*m*) ⇓ false

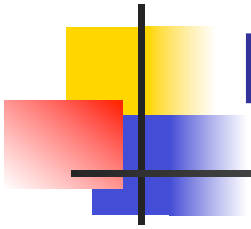


Booleans:

$$\frac{(B, m) \Downarrow \text{false}}{(B \,\&\, B', m) \Downarrow \text{false}} \qquad \frac{(B, m) \Downarrow \text{true} \quad (B', m) \Downarrow b}{(B \,\&\, B', m) \Downarrow b}$$

$$\frac{(B, m) \Downarrow \text{true}}{(B \text{ or } B', m) \Downarrow \text{true}} \qquad \frac{(B, m) \Downarrow \text{false} \quad (B', m) \Downarrow b}{(B \text{ or } B', m) \Downarrow b}$$

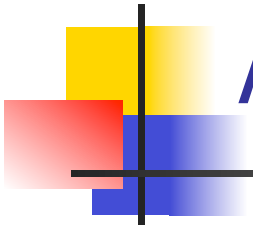
$$\frac{(B, m) \Downarrow \text{true}}{(\text{not } B, m) \Downarrow \text{false}} \qquad \frac{(B, m) \Downarrow \text{false}}{(\text{not } B, m) \Downarrow \text{true}}$$



Relations

$$\frac{(E, m) \Downarrow U \quad (E', m) \Downarrow V \quad U \sim V = b}{(E \sim E', m) \Downarrow b}$$

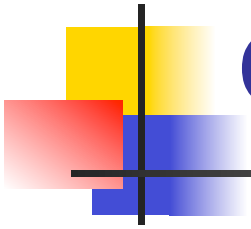
- By $U \sim V = b$, we mean does (the meaning of) the relation $\sim$ hold on the meaning of U and V
- May be specified by a mathematical expression/equation or rules matching U and V



Arithmetic Expressions

$$\frac{(E, m) \Downarrow U \quad (E', m) \Downarrow V \quad U \; op \; V = N}{(E \; op \; E', m) \Downarrow N}$$

where N is the specified value for $U \; op \; V$

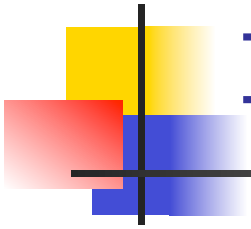


Commands

Skip: $(\text{skip}, m) \Downarrow m$

Assignment:
$$\frac{(E,m) \Downarrow V}{(I ::= E, m) \Downarrow m[I \text{ <-- } V]}$$

Sequencing:
$$\frac{(C,m) \Downarrow m' \quad (C',m') \Downarrow m''}{(C;C', m) \Downarrow m''}$$



If Then Else Command

$$\frac{(B,m) \Downarrow \text{true} \quad (C,m) \Downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m'}$$

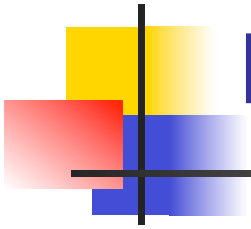
$$\frac{(B,m) \Downarrow \text{false} \quad (C',m) \Downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m'}$$

While Command

$$\frac{(B,m) \Downarrow \text{false}}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m}$$

$$\frac{(B,m) \Downarrow \text{true} \quad (C,m) \Downarrow m' \quad (\text{while } B \text{ do } C \text{ od}, m') \Downarrow m''}{(\text{while } B \text{ do C od}, m) \Downarrow m''}$$

Example: If Then Else Rule



$$\frac{}{(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi}, \{x \to 7\}) \Downarrow ?}$$

Example: If Then Else Rule

$$\frac{(x > 5, \{x \to 7\}) \Downarrow ?}{(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi}, \{x \to 7\}) \Downarrow \ ?}$$

Example: Arith Relation

$$\dfrac{\dfrac{\dfrac{}{? > ? = ?}}{(x,\{x\text{->}7\})\Downarrow?\quad(5,\{x\text{->}7\})\Downarrow?}}{(x > 5, \{x \text{->} 7\})\Downarrow?}$$

$$\overline{(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi}, \{x \text{->} 7\}) \Downarrow ?}$$

Example: Identifier(s)

$$\frac{7 > 5 = \text{true} \qquad \underline{(x,\{x\text{->}7\})\Downarrow 7 \quad (5,\{x\text{->}7\})\Downarrow 5}}{\dfrac{(x > 5, \{x \text{ -> } 7\})\Downarrow ?}{(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi}, \{x \text{ -> } 7\}) \Downarrow ?}}$$

Example: Arith Relation

$$\dfrac{\dfrac{7 > 5 = \text{true}}{(x,\{x\text{->}7\})\Downarrow 7 \quad (5,\{x\text{->}7\})\Downarrow 5}}{\dfrac{(x > 5, \{x \text{ -> } 7\})\Downarrow \text{true}}{(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi}, \{x \text{ -> } 7\}) \Downarrow ?}}$$

Example: If Then Else Rule

$$\frac{\dfrac{7 > 5 = \text{true}}{(x,\{x\text{->}7\}) \Downarrow 7 \quad (5,\{x\text{->}7\}) \Downarrow 5} \qquad \dfrac{}{(y{:=}\ 2 + 3,\{x\text{->}\ 7\}) \Downarrow ?}\ .}{(x > 5, \{x \text{->} 7\}) \Downarrow \text{true}}$$

$$\frac{}{(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi}, \{x \text{->} 7\}) \Downarrow ?}$$

Example: Assignment

$$\dfrac{\dfrac{7 > 5 = \text{true}}{(x,\{x\text{->}7\})\Downarrow 7 \quad (5,\{x\text{->}7\})\Downarrow 5} \quad \dfrac{\dfrac{}{(2+3,\ \{x\text{->}7\})\Downarrow ?}}{(y{:=}\ 2 + 3,\{x\text{->}\ 7\})}}{\dfrac{(x > 5,\ \{x \text{->}\ 7\})\Downarrow \text{true} \qquad \Downarrow\ ?}{(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi},\ \{x \text{->}\ 7\}) \Downarrow\ ?}}$$

Example: Arith Op

$$\dfrac{\dfrac{\dfrac{}{(x,\{x\text{->}7\})\Downarrow 7}\quad\dfrac{}{(5,\{x\text{->}7\})\Downarrow 5}}{(x > 5,\{x \text{->} 7\})\Downarrow \text{true}}\qquad \dfrac{\dfrac{\dfrac{}{(2,\{x\text{->}7\})\Downarrow ?}\quad\dfrac{}{(3,\{x\text{->}7\})\Downarrow ?}}{(2+3,\{x\text{->}7\})\Downarrow ?}}{(y:= 2 + 3,\{x\text{->} 7\})\Downarrow ?}\quad .}{(\text{if } x > 5 \text{ then } y:= 2 + 3 \text{ else } y:=3 + 4 \text{ fi},\{x \text{->} 7\}) \Downarrow ?}$$

? + ? = ?

7 > 5 = true

Example: Numerals

$$\dfrac{\dfrac{7 > 5 = \text{true}}{(x,\{x\text{->}7\})\Downarrow 7 \quad (5,\{x\text{->}7\})\Downarrow 5} \qquad \dfrac{\dfrac{2 + 3 = 5}{(2,\{x\text{->}7\})\Downarrow 2 \quad (3,\{x\text{->}7\}) \Downarrow 3}}{(2+3, \{x\text{->}7\})\Downarrow ?}}{\dfrac{(x > 5, \{x \text{->} 7\})\Downarrow \text{true} \qquad (y\text{:=}\ 2 + 3,\{x\text{->}\ 7\}) \Downarrow\ ?}{(\text{if } x > 5 \text{ then } y\text{:=}\ 2 + 3 \text{ else } y\text{:=}3 + 4 \text{ fi}, \{x \text{->} 7\}) \Downarrow\ ?}}$$

Example: Arith Op

$$\dfrac{\dfrac{7 > 5 = \text{true}}{(x,\{x\text{->}7\})\Downarrow 7 \quad (5,\{x\text{->}7\})\Downarrow 5} \qquad \dfrac{\dfrac{2 + 3 = 5}{(2,\{x\text{->}7\})\Downarrow 2 \quad (3,\{x\text{->}7\}) \Downarrow 3}}{(2+3, \{x\text{->}7\})\Downarrow 5}}{\dfrac{(x > 5, \{x \text{->} 7\})\Downarrow \text{true} \qquad (y\text{:=}\ 2 + 3,\{x\text{->}\ 7\}) \Downarrow ?}{(\text{if } x > 5 \text{ then } y\text{:=}\ 2 + 3 \text{ else } y\text{:=}3 + 4 \text{ fi}, \{x \text{->} 7\}) \Downarrow\ ?}}$$

Example: Assignment

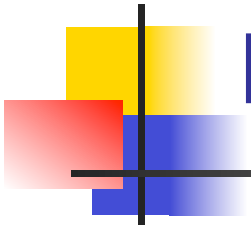
$$\dfrac{\dfrac{\dfrac{}{7 > 5 = \text{true}}\quad \dfrac{\dfrac{}{2 + 3 = 5}}{(2,\{x\text{->}7\})\Downarrow 2 \quad (3,\{x\text{->}7\})\Downarrow 3}}{(x,\{x\text{->}7\})\Downarrow 7 \quad (5,\{x\text{->}7\})\Downarrow 5} \quad \dfrac{(2+3,\{x\text{->}7\})\Downarrow 5}{(y{:=}\ 2 + 3,\{x\text{->}\ 7\})}}{(x > 5,\{x \text{->}\ 7\})\Downarrow \text{true} \qquad \Downarrow \{x\text{->}7,\ y\text{->}5\}}$$

$$(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi},\ \{x \text{->}\ 7\}) \Downarrow ?$$

Example: If Then Else Rule

$$\dfrac{\dfrac{\dfrac{}{7 > 5 = \text{true}}}{(x,\{x\text{->}7\})\Downarrow 7 \quad (5,\{x\text{->}7\})\Downarrow 5} \qquad \dfrac{\dfrac{}{2 + 3 = 5}}{(2,\{x\text{->}7\})\Downarrow 2 \quad (3,\{x\text{->}7\})\Downarrow 3}}{(x > 5, \{x \text{->} 7\})\Downarrow \text{true}} \qquad \dfrac{(2+3, \{x\text{->}7\})\Downarrow 5}{(y{:=}\ 2 + 3,\{x\text{->}\ 7\}) \Downarrow \{x\text{->}7, y\text{->}5\}}$$

$$(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi}, \{x \text{->} 7\}) \Downarrow \{x\text{->}7, y\text{->}5\}$$



Let in Command

$$\frac{(E,m) \Downarrow v \quad (C,m[I<\text{-}v]) \Downarrow m'}{(\text{let } I = E \text{ in } C,\ m) \Downarrow m''}$$

Where $m''(y) = m'(y)$ for $y \neq I$ and $m''(I) = m(I)$ if $m(I)$ is defined, and $m''(I)$ is undefined otherwise

Example

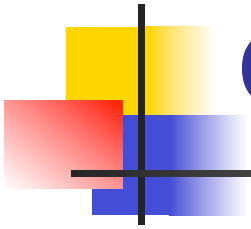
$$\dfrac{(5,\{x\text{->}17\}) \Downarrow 5 \qquad \dfrac{(x,\{x\text{->}5\}) \Downarrow 5 \quad (3,\{x\text{->}5\}) \Downarrow 3}{(x+3,\{x\text{->}5\}) \Downarrow 8}}{(x\text{:=}x+3,\{x\text{->}5\}) \Downarrow \{x\text{->}8\}}$$

$$(\text{let } x = 5 \text{ in } (x\text{:=}x+3), \{x \text{ -> } 17\}) \Downarrow ?$$

Example

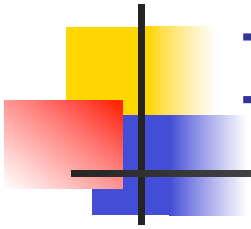
$$\dfrac{(5,\{x\text{->}17\}) \Downarrow 5 \qquad \dfrac{(x,\{x\text{->}5\}) \Downarrow 5 \quad (3,\{x\text{->}5\}) \Downarrow 3}{(x+3,\{x\text{->}5\}) \Downarrow 8}}{(x\text{:=}x+3,\{x\text{->}5\}) \Downarrow \{x\text{->}8\}}$$

$$(\text{let } x = 5 \text{ in } (x\text{:=}x+3), \{x \text{ -> } 17\}) \Downarrow \{x\text{->}17\}$$



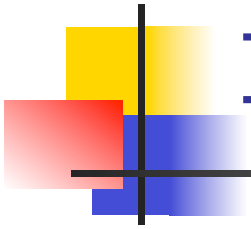
Comment

- Simple Imperative Programming Language introduces variables *implicitly* through assignment
- The let-in command introduces scoped variables *explictly*
- Clash of constructs apparent in awkward semantics



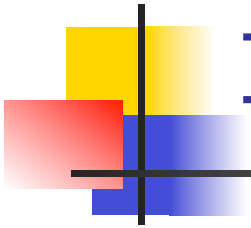
Interpretation Versus Compilation

- A **compiler** from language L1 to language L2 is a program that takes an L1 program and for each piece of code in L1 generates a piece of code in L2 of same meaning
- An **interpreter** of L1 in L2 is an L2 program that executes the meaning of a given L1 program
- Compiler would examine the body of a loop once; an interpreter would examine it every time the loop was executed



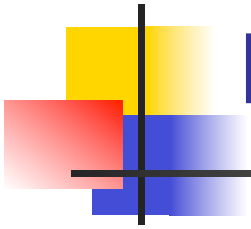
Interpreter

- An *Interpreter* represents the operational semantics of a language L1 (source language) in the language of implementation L2 (target language)
- Built incrementally
  - Start with literals
  - Variables
  - Primitive operations
  - Evaluation of expressions
  - Evaluation of commands/declarations



Interpreter

- Takes abstract syntax trees as input
  - In simple cases could be just strings
- One procedure for each syntactic category (nonterminal)
  - eg one for expressions, another for commands
- If Natural semantics used, tells how to compute final value from code
- If Transition semantics used, tells how to compute next “state”
  - To get final value, put in a loop



Natural Semantics Example

- compute_exp (Var(v), m) = look_up v m
- compute_exp (Int(n), _) = Num (n)
- …
- compute_com(IfExp(b,c1,c2),m) =

  if compute_exp (b,m) = Bool(true)

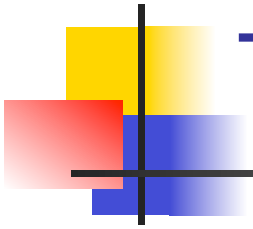
  then compute_com (c1,m)

  else compute_com (c2,m)

Natural Semantics Example

- compute_com(While(b,c), m) =
  if compute_exp (b,m) = Bool(false)
  then m
  else compute_com
  (While(b,c), compute_com(c,m))

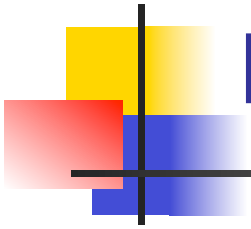
- May fail to terminate - exceed stack limits
- Returns no useful information then



Transition Semantics

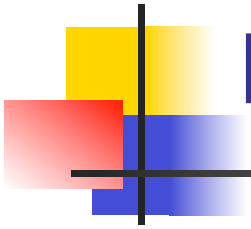
- Form of operational semantics
- Describes how each program construct transforms machine state by *transitions*
- Rules look like

  (C, *m*) --> (C', *m'*) or (C,*m*) --> *m'*
- *C, C'* is code remaining to be executed
- m, m' represent the state/store/memory/ environment
  - Partial mapping from identifiers to values
  - Sometimes *m* (or *C*) not needed
- Indicates exactly one step of computation



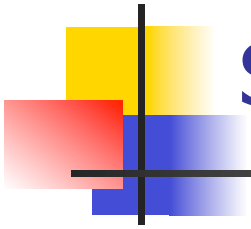
Expressions and Values

- *C, C'* used for commands; *E, E' for* expressions; *U,V* for values
- Special class of expressions designated as *values*
  - Eg 2, 3 are values, but 2+3 is only an expression
- Memory only holds values
  - Other possibilities exist



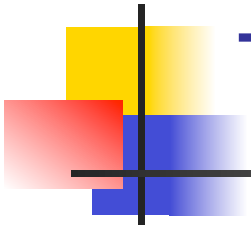
Evaluation Semantics

- Transitions successfully stops when E/*C* is a value/memory
- Evaluation fails if no transition possible, but not at value/memory
- Value/memory is the final *meaning* of original expression/command (in the given state)
- Coarse semantics: final value / memory
- More fine grained: whole transition sequence



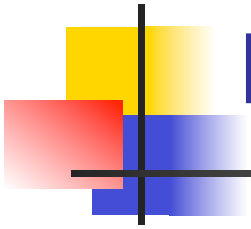
Simple Imperative Programming Language

- $I \in Identifiers$
- $N \in Numerals$
- B ::= true | false | B & B | B or B | not B | $E < E$ | $E = E$
- $E ::= N \mid I \mid E + E \mid E * E \mid E - E \mid - E$
- C::= skip | $C;C$ | I ::= E
  | if B then C else C fi | while B do C od



Transitions for Expressions

- Numerals are values
- Boolean values = {true, false}
- Identifiers: (*I*,*m*) --> (*m*(*I*), *m*)



Boolean Operations:

- Operators: (short-circuit)

(false & *B*, *m*) --> (false,*m*)

(true & *B*, *m*) --> (*B*,*m*)

$$\frac{(B, m) \rightarrow (B'', m)}{(B \,\&\, B', m) \rightarrow (B'' \,\&\, B', m)}$$

(true or *B*, *m*) --> (true,*m*)

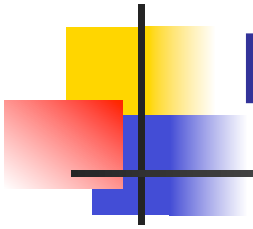
(false or *B*, *m*) --> (*B*,*m*)

$$\frac{(B, m) \rightarrow (B'', m)}{(B \text{ or } B', m) \rightarrow (B'' \text{ or } B', m)}$$

(not true, m) --> (false,*m*)

(not false, m) --> (true,*m*)

$$\frac{(B, m) \rightarrow (B', m)}{(\text{not } B, m) \rightarrow (\text{not } B', m)}$$



Relations

$$\frac{(E, m) \text{ --> } (E'', m)}{(E \sim E', m) \text{ --> } (E'' \sim E', m)}$$

$$\frac{(E, m) \text{ --> } (E', m)}{(V \sim E, m) \text{ --> } (V \sim E', m)}$$

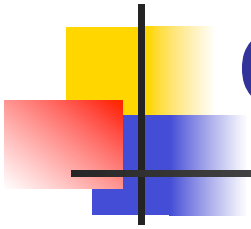
$(U \sim V, m)$ --> (true,m) or (false,m)
depending on whether $U \sim V$ holds or not

Arithmetic Expressions

$$\frac{(E, m) \text{ --> } (E'', m)}{(E \text{ op } E', m) \text{ --> } (E'' \text{ op } E', m)}$$

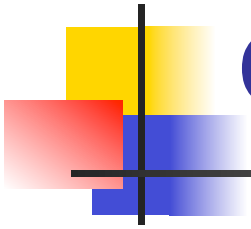
$$\frac{(E, m) \text{ --> } (E', m)}{(V \text{ op } E, m) \text{ --> } (V \text{ op } E', m)}$$

$(U \text{ op } V, m) \text{ --> } (N, m)$ where N is the specified value for $U \text{ op } V$



Commands - in English

- skip means done evaluating
- When evaluating an assignment, evaluate the expression first
- If the expression being assigned is already a value, update the memory with the new value for the identifier
- When evaluating a sequence, work on the first command in the sequence first
- If the first command evaluates to a new memory (ie completes), evaluate remainder with new memory



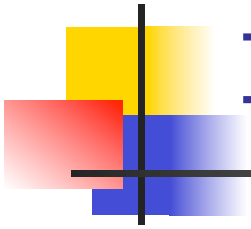
Commands

$$(\text{skip}, m) \text{ --> } m$$

$$\frac{(E,m) \text{ --> } (E',m)}{(I::=E,m) \text{ --> } (I::=E',m)}$$

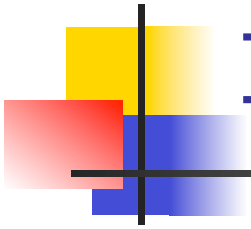
$$(I::=V,m) \text{ --> } m[I \text{ <-- } V]$$

$$\frac{(C,m) \text{ --> } (C'',m')}{(C;C', m) \text{ --> } (C'';C',m')} \qquad \frac{(C,m) \text{ --> } m'}{(C;C', m) \text{ --> } (C',m')}$$



If Then Else Command - in English

- If the boolean guard in an if_then_else is true, then evaluate the first branch
- If it is false, evaluate the second branch
- If the boolean guard is not a value, then start by evaluating it first.

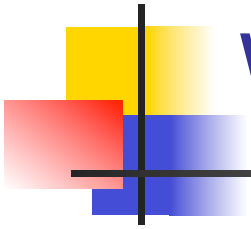


If Then Else Command

(if true then *C* else *C'* fi, *m*) --> (*C*, *m*)

(if false then *C* else *C'* fi, *m*) --> (*C'*, *m*)

$$\frac{(B,m) \text{ --> } (B',m)}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \text{ --> } (\text{if } B' \text{ then } C \text{ else } C' \text{ fi}, m)}$$



While Command

(while *B* do *C* od, *m*) -->

(if *B* then *C*; while *B* do *C* od else skip fi, m)

In English: Expand a While into a test of the boolean guard, with the true case being to do the body and then try the while loop again, and the false case being to stop.