Disambiguating a Grammar

- `<exp>::= 0|1|b<exp> | <exp>a
  | <exp>m<exp>`
- Want a has higher precedence than b, which in turn has higher precedence than m, and such that m associates to the left.

```
<exp> ::= <no m> ::= <no b> ::= <no b>a|0|1
<exp>m<no m> | <no m> b<no m> | <no b>
```

LR Parsing

- Read tokens left to right (L)
- Create a rightmost derivation (R)
- How is this possible?
- Start at the bottom (left) and work your way up
- Last step has only one non-terminal to be replaced so is right-most
- Working backwards, replace mixed strings by non-terminals
- Always proceed so that there are no non-terminals to the right of the string to be replaced

```
Example: <Sum> = 0 | 1 | (<Sum>)
            | <Sum> + <Sum>
<Sum> =>

= <Sum>= shift
(0 + 1) + 0
```

```
Example: <Sum> = 0 | 1 | (<Sum>)
            | <Sum> + <Sum>
<Sum> =>

= (0 + 1) + 0 shift
= (0 + 1) + 0 shift
```
Example: <Sum> = 0 | 1 | (<Sum>)
| <Sum> + <Sum>

<Sum> = >

=> (0 ● 1) + 0
    reduce
    = (0 + 1) + 0
    shift
    => (0 + 1) + 0
    shift

Example: <Sum> = 0 | 1 | (<Sum>)
| <Sum> + <Sum>

<Sum> = >

=> (0 ● 1) + 0
    reduce
    = (0 + 1) + 0
    shift
    => (0 + 1) + 0
    shift

Example: <Sum> = 0 | 1 | (<Sum>)
| <Sum> + <Sum>

<Sum> = >

=> (0 ● 1) + 0
    reduce
    = (0 + 1) + 0
    shift
    => (0 + 1) + 0
    shift

Example: <Sum> = 0 | 1 | (<Sum>)
| <Sum> + <Sum>

<Sum> = >

=> (0 ● 1) + 0
    reduce
    = (0 + 1) + 0
    shift
    => (0 + 1) + 0
    shift

Example: <Sum> = 0 | 1 | (<Sum>)
| <Sum> + <Sum>

<Sum> = >

=> (0 ● 1) + 0
    reduce
    = (0 + 1) + 0
    shift
    => (0 + 1) + 0
    shift

Example: <Sum> = 0 | 1 | (<Sum>)
| <Sum> + <Sum>

<Sum> = >

=> (0 ● 1) + 0
    reduce
    = (0 + 1) + 0
    shift
    => (0 + 1) + 0
    shift
Example

(0 + 1) + 0

Example

(0 + 1) + 0

Example

<Sum>

(0 + 1) + 0

Example

<Sum>

(0 + 1) + 0

Example

<Sum>

(0 + 1) + 0

Example

<Sum>

(0 + 1) + 0
Example

(0 + 1) + 0

Example

(0 + 1) + 0

Example

(0 + 1) + 0

Example

(0 + 1) + 0

Example

(0 + 1) + 0
LR Parsing Tables
- Build a pair of tables, Action and Goto, from the grammar
  - This is the hardest part, we omit here
  - Rows labeled by states
  - For Action, columns labeled by terminals and “end-of-tokens” marker
    - (more generally strings of terminals of fixed length)
  - For Goto, columns labeled by non-terminals

Action and Goto Tables
- Given a state and the next input, Action table says either
  - **shift** and go to state \( n \), or
  - **reduce** by production \( k \) (explained in a bit)
  - **accept** or **error**
- Given a state and a non-terminal, Goto table says
  - go to state \( m \)

LR(i) Parsing Algorithm
- Based on push-down automata
- Uses states and transitions (as recorded in Action and Goto tables)
- Uses a stack containing states, terminals and non-terminals
LR(i) Parsing Algorithm

0. Insure token stream ends in special "end-of-tokens" symbol
1. Start in state 1 with an empty stack
2. Push state(1) onto stack
3. Look at next \(i\) tokens from token stream (\(toks\)) (don't remove yet)
4. If top symbol on stack is state(\(n\)), look up action in Action table at (\(n, toks\))

5. If action = shift \(m\),
   a) Remove the top token from token stream and push it onto the stack
   b) Push state(\(m\)) onto stack
   c) Go to step 3

6. If action = reduce \(k\) where production \(k\) is \(E ::= u\)
   a) Remove \(2 \times \text{length}(u)\) symbols from stack (\(u\) and all the interleaved states)
   b) If new top symbol on stack is state(\(m\)), look up new state \(p\) in Goto(\(m, E\))
   c) Push E onto the stack, then push state(\(p\)) onto the stack
   d) Go to step 3

7. If action = accept
   Stop parsing, return success
8. If action = error,
   Stop parsing, return failure

Adding Synthesized Attributes

- Add to each reduce a rule for calculating the new synthesized attribute from the component attributes
- Add to each non-terminal pushed onto the stack, the attribute calculated for it
- When performing a reduce,
  - gather the recorded attributes from each non-terminal popped from stack
  - Compute new attribute for non-terminal pushed onto stack

Shift-Reduce Conflicts

- Problem: can't decide whether the action for a state and input character should be shift or reduce
- Caused by ambiguity in grammar
- Usually caused by lack of associativity or precedence information in grammar
Example: \( <\text{Sum}> = 0 \mid 1 \mid ( <\text{Sum}> ) \) 
\( <\text{Sum}> + <\text{Sum}> \)

\[
\begin{align*}
0 + 1 + 0 & \quad \text{shift} \\
\rightarrow 0 & \quad \text{reduce} \\
\rightarrow <\text{Sum}> + 1 + 0 & \quad \text{shift} \\
\rightarrow <\text{Sum}> + 1 & \quad \text{reduce} \\
\rightarrow <\text{Sum}> + <\text{Sum}> & + 0 
\end{align*}
\]

Example - cont

Problem: shift or reduce?

You can shift-shift-reduce-reduce or reduce-shift-shift-reduce

Shift first - right associative
Reduce first- left associative

Reduce - Reduce Conflicts

Problem: can’t decide between two different rules to reduce by
Again caused by ambiguity in grammar
Symptom: RHS of one production suffix of another
Requires examining grammar and rewriting it
Harder to solve than shift-reduce errors

Example

\[
\begin{align*}
S ::= A \mid aB \\
A ::= abc \\
B ::= bc
\end{align*}
\]

Problem: reduce by \( B ::= bc \) then by \( S ::= aB \), or by \( A ::= abc \) then \( S ::= A \)?
Recursive Descent Parsing

- Each subprogram must be able to decide how to begin parsing by looking at the left-most character in the string to be parsed.
- May do so directly, or indirectly by calling another parsing subprogram.
- Recursive descent parsers, like other top-down parsers, cannot be built from left-recursive grammars.
- Sometimes can modify grammar to suit.

Sample Grammar

\[
\begin{align*}
<expr> & ::= <term> | <term> + <expr> \\
& \quad | <term> - <expr> \\
<term> & ::= <factor> | <factor> * <term> \\
& \quad | <factor> / <term> \\
<factor> & ::= <id> | ( <expr> )
\end{align*}
\]

Tokens as OCaml Types

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

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

Parse Trees as Datatypes

\[
\begin{align*}
<expr> & ::= <term> | <term> + <expr> \\
& \quad | <term> - <expr> \\
<term> & ::= <factor> | <factor> * <term> \\
& \quad | <factor> / <term> \\
<factor> & ::= <id> | ( <expr> )
\end{align*}
\]

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

```ocaml
type term = 
  Factor_as_Term of factor  
| Mult_Term of (factor * term) 
| Div_Term of (factor * term)
```

```ocaml
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

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

```latex
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

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

```latex
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

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

```latex
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

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

```latex
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))

Building Plus Expression Parse Tree

<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 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 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**

\[
\text{<factor> ::= <id>} \]

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**

\[
\text{<factor> ::= ( <expr> )} \]

| factor (Left_parenthesis :: tokens) = (match expr tokens with (expr_parse, tokens_after_expr) ->

---

**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"];;

---

- : 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 [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"),
    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"))))

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 on 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 =>* aw\} \cup \{\epsilon | if y =>* \epsilon\}.
- For each pair of rules A ::= y and A ::= z, require FIRST(y) \cap FIRST(z) = \{\}.

Example

Grammar:

\(<S> ::= <A> a <B> b\\n<A> ::= <A> b | b\\n<B> ::= a <B> | a\\n\)

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> | \epsilon.
Factoring Grammar
- Test too strong: Can’t handle 
  \( <\text{expr}> ::= <\text{term}> [ ( + | - ) <\text{expr}> ] \)
- Answer: Add new non-terminal and replace above rules by 
  \( <\text{expr}> ::= <\text{term}><\text{e}> \)
  \( <\text{e}> ::= + <\text{term}><\text{e}> \)
  \( <\text{e}> ::= - <\text{term}><\text{e}> \)
  \( <\text{e}> ::= \varepsilon \)
- You are delaying the decision point

Example
Both \(<A>\) and \(<B>\)
have problems: 
Transform grammar to:

\[
\begin{align*}
<\text{S}> & ::= <\text{A}>a<\text{B}>b \\
<\text{A}> & ::= <\text{A}>b | b \\
<\text{B}> & ::= a<\text{B}> | a \\
<\text{A}1> & ::= b<\text{A}1> | \varepsilon \\
<\text{B}1> & ::= a<\text{B}1> | \varepsilon
\end{align*}
\]

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:
  \{\text{Precondition}\} \text{ Program } \{\text{Postcondition}\}
- Source of idea of loop invariant

Denotational Semantics

- Construct a function $M$ 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 Structural Operational 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
- $B ::= \text{true} \mid \text{false} \mid B \& B \mid B \lor B \mid \text{not } B$
  $\mid E < E \mid E = E$
- $E ::= N \mid I \mid E + E \mid E \* E \mid E - E \mid - E$
- $C ::= \text{skip} \mid C \text{;} C \mid I ::= E$
  $\mid \text{if } B \text{ then } C \text{ else } C \text{ fi} \mid \text{while } B \text{ do } C \text{ od}$

Natural Semantics of Atomic Expressions

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

\[
\begin{align*}
(B, m) & \Downarrow \text{false} \\
(B \& B', m) & \Downarrow \text{false} \\
B & \Downarrow \text{true} \\
(B \& B', m) & \Downarrow \text{b} \\
(B, m) & \Downarrow \text{true} \\
(B, m) & \Downarrow \text{false} \\
(B, m) & \Downarrow \text{true} \\
(B, m) & \Downarrow \text{false}
\end{align*}
\]

**Relations**

\[
\begin{align*}
(E, m) & \Downarrow U \\
(E', m) & \Downarrow V \\
(U \sim V = b) & \\
(E \sim E', m) & \Downarrow \text{b}
\end{align*}
\]

- 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**

\[
\begin{align*}
(E, m) & \Downarrow U \\
(E', m) & \Downarrow V \\
(U \circ V = N) & \\
(E \circ E', m) & \Downarrow N
\end{align*}
\]

where \( N \) is the specified value for \( U \circ V \).

**Commands**

- **Skip:**
  \[
  (\text{skip}, m) \Downarrow m
  \]

- **Assignment:**
  \[
  (E, m) \Downarrow V \\
  (I::=E,m) \Downarrow m[I <-- V]
  \]

- **Sequencing:**
  \[
  (C, m) \Downarrow m' \quad (C', m') \Downarrow m'' \\
  (C;C', m) \Downarrow m''
  \]

**If Then Else Command**

\[
\begin{align*}
(B, m) & \Downarrow \text{true} \quad (C, m) \Downarrow m' \\
& \quad (\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m' \\
(B, m) & \Downarrow \text{false} \quad (C', m) \Downarrow m' \\
& \quad (\text{if } B \text{ then } C' \text{ fi}, m) \Downarrow m'
\end{align*}
\]

**While Command**

\[
\begin{align*}
(B, m) & \Downarrow \text{false} \\
& \quad (\text{while } B \text{ do } C \text{ od}, m) \Downarrow m \\
(B', m) & \Downarrow \text{true} \quad (C, m) \Downarrow m' \\
& \quad (\text{while } B \text{ do } C \text{ od}, m') \Downarrow m''
\end{align*}
\]
Example: If Then Else Rule

\[
\begin{align*}
\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} \\
\{x \rightarrow 7\} \Downarrow ?
\end{align*}
\]

Example: Arith Relation

\[
\begin{align*}
? > ? &= \text{true} \\
(x, \{x \rightarrow 7\}) \Downarrow 7 (5, \{x \rightarrow 7\}) \Downarrow 5 \\
(x > 5, \{x \rightarrow 7\}) \Downarrow \\
\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} \\
\{x \rightarrow 7\} \Downarrow ?
\end{align*}
\]

Example: Identifier(s)

\[
\begin{align*}
7 > 5 &= \text{true} \\
(x, \{x \rightarrow 7\}) \Downarrow 7 (5, \{x \rightarrow 7\}) \Downarrow 5 \\
(x > 5, \{x \rightarrow 7\}) \Downarrow \\
\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} \\
\{x \rightarrow 7\} \Downarrow ?
\end{align*}
\]
Example: Assignment

\[
\begin{align*}
7 > 5 &= \text{true} \\
(x, \{x -> 7\}) &\Downarrow 7 \\
(2+3, \{x -> 7\}) &\Downarrow 5 \\
(\text{x > 5, } \{x -> 7\}) &\Downarrow \text{if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,} \\
&\Downarrow ? \\
\end{align*}
\]

Example: Arith Op

\[
\begin{align*}
? + ? &= ? \\
7 > 5 &= \text{true} \\
(2+3, \{x -> 7\}) &\Downarrow 5 \\
(\text{x > 5, } \{x -> 7\}) &\Downarrow \text{if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,} \\
&\Downarrow ? \\
\end{align*}
\]

Example: Numerals

\[
\begin{align*}
2 + 3 &= 5 \\
7 > 5 &= \text{true} \\
(2+3, \{x -> 7\}) &\Downarrow 5 \\
(\text{x > 5, } \{x -> 7\}) &\Downarrow \text{if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,} \\
&\Downarrow ? \\
\end{align*}
\]

Example: If Then Else Rule

\[
\begin{align*}
2 + 3 &= 5 \\
7 > 5 &= \text{true} \\
(2+3, \{x -> 7\}) &\Downarrow 5 \\
(\text{x > 5, } \{x -> 7\}) &\Downarrow \text{if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,} \\
&\Downarrow ? \\
\end{align*}
\]
Let in Command

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

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

\[(x,\{x->5\}) \downarrow 5 \ (3,\{x->5\}) \downarrow 3\]
\[(x+3,\{x->5\}) \downarrow 8\]
\[(5,\{x->17\}) \downarrow 5 \ (x:=x+3,\{x->5\}) \downarrow \{x->8\}\]
\[(\text{let } x = 5 \text{ in } (x:=x+3), \{x -> 17\}) \downarrow ?\]

Comment

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

Interpreter 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)

- 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