Disambiguating a Grammar

- \( \langle \text{exp}\rangle ::= 0|1| b<\text{exp}> | <\text{exp}>a \)
- \( | <\text{exp}>m<\text{exp}> \)
- Want a has higher precedence than b, which in turn has higher precedence than m, and such that m associates to the left.

Disambiguating a Grammar – Take 2

- \( \langle \text{exp}\rangle ::= 0|1| b<\text{exp}> | <\text{exp}>a \)
- \( | <\text{exp}>m<\text{exp}> \)
- Want b has higher precedence than m, which in turn has higher precedence than a, and such that m associates to the right.

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: \( \text{<Sum>} = 0 | 1 | (\text{<Sum>}) \)
| \( \text{<Sum>} + \text{<Sum>} \)

\[
\text{<Sum>} \Rightarrow
\]
\[
= \text{●}(0+1)+0 \quad \text{shift}
\]
Example: \(<\text{Sum}\> = 0 \mid 1 \mid (<\text{Sum}\>)

\begin{align*}
\text{Sum} & \Rightarrow \\
\Rightarrow (\text{Sum} + \text{Sum} \cdot 0) + 0 & \text{ reduce} \\
\Rightarrow (\text{Sum} + 1 \cdot 0) + 0 & \text{ reduce} \\
= (\text{Sum} + 1) + 0 & \text{ shift} \\
\Rightarrow (\text{Sum} + 1) + 0 & \text{ reduce} \\
= (0 + 1) + 0 & \text{ shift} \\
\Rightarrow (0 + 1) + 0 & \text{ shift} \\
\end{align*}
Example: \( \text{Sum} = 0 \mid 1 \mid (\text{Sum}) \)
\[\begin{align*}
\text{Sum} & \Rightarrow \text{Sum} + \text{Sum} \quad \text{reduce} \\
& \Rightarrow \text{Sum} + 0 \quad \text{reduce} \\
& = \text{Sum} + 0 \quad \text{shift} \\
& = (\text{Sum}) + 0 \quad \text{shift} \\
& = (\text{Sum} + \text{Sum}) + 0 \quad \text{reduce} \\
& \Rightarrow (\text{Sum} + 1) + 0 \quad \text{reduce} \\
& = (\text{Sum} + 1) + 0 \quad \text{shift} \\
& = (0 + 1) + 0 \quad \text{reduce} \\
& = (0 + 1) + 0 \quad \text{shift} \\
& = (0 + 1) + 0 \quad \text{shift}
\end{align*}\]
Example

\[ \langle \text{Sum} \rangle (0 + 1) + 0 \]
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

---

0. Insure token stream ends in special “end-of-tokens” symbol
1. Start in state 1 with an empty stack
2. Push \( \text{state}(1) \) onto stack
3. Look at next \( i \) tokens from token stream \( \text{toks} \) (don’t remove yet)
4. If top symbol on stack is \( \text{state}(n) \), look up action in Action table at \( (n, \text{toks}) \)
5. If action = **shift** \( m \),
   a) Remove the top token from token stream and push it onto the stack
   b) Push \( \text{state}(m) \) 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 \( \text{state}(m) \), look up new state \( p \) in Goto\((m,E)\)
   c) Push \( E \) onto the stack, then push \( \text{state}(p) \) 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: `<Sum> = 0 | 1 | (<Sum>)<Sum> + <Sum>`

- `0 + 1 + 0` shift
- `0 + 1 + 0` reduce
- `<Sum> + 1 + 0` shift
- `<Sum> + 1 + 0` reduce
- `<Sum> + <Sum> + 0` reduce

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

```plaintext
S ::= A | aB  A ::= abc  B ::= bc
```

- `abc` shift
- `a bc` shift
- `ab c` shift
- `abc` shift

- Problem: reduce by `B ::= bc` then by `S ::= aB`, or by `A ::= abc` then `S ::= A`?
Recursive Descent Parsing

- Recursive descent parsers are a class of parsers derived fairly directly from BNF grammars.
- A recursive descent parser traces out a parse tree in top-down order, corresponding to a left-most derivation (LL - left-to-right scanning, leftmost derivation).

Each nonterminal in the grammar has a subprogram associated with it; the subprogram parses all phrases that the nonterminal can generate.

Each nonterminal in right-hand side of a rule corresponds to a recursive call to the associated subprogram.

Recursive descent parsers, like other top-down parsers, cannot be built from left-recursive grammars. Sometimes can modify grammar to suit.

Sample Grammar

```latex
<expr> ::= <term> | <term> + <expr>
         | <term> - <expr>
<term> ::= <factor> | <factor> * <term>
         | <factor> / <term>
<factor> ::= <id> | ( <expr> )
```

Tokens as OCaml Types

```latex
+ - * / ( ) <id>
```

Becomes an OCaml datatype.

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

Parse Trees as Datatypes

```latex
type expr =
    Term_as_Expr of term
    | Plus_Expr of (term * expr)
    | Minus_Expr of (term * expr)
```
Parse Trees as Datatypes

\[
\text{<term> ::= <factor> | <factor> * <term> | <factor> / <term>}
\]

and term =
  \text{Factor\_as\_Term of factor}
  \text{Mult\_Term of (factor * term)}
  \text{Div\_Term of (factor * term)}

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

\[
\text{<expr> ::= <term> [(+ | -) <expr> ]}
\]

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

\[
\text{<expr> ::= <term> [(+ | -) <expr> ]}
\]

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

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

and factor =
  \text{Id\_as\_Factor of string}
  \text{Parenthesized\_Expr\_as\_Factor of expr}
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

<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 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

\[ \text{<expr>} ::= \text{<term>} - \text{<expr>} \]

\( | \text{ ( Minus_token} :: \text{tokens_after_minus) -> (match expr tokens_after_minus with ( expr_parse , tokens_after_expr) -> ( Minus_Expr ( term_parse , expr_parse ), tokens_after_expr))} \)

Parsing an Expression as a Term

\[ \text{<expr>} ::= \text{<term>} \]

\( | \_ \rightarrow (\text{Term_as_Expr} \text{ term_parse} , \text{tokens_after_term})) \)

- Code for \text{term} is same except for replacing addition with multiplication and subtraction with division

Parsing Factor as Id

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

\( \text{and factor tokens = (match tokens with (Id_token id_name :: tokens_after_id) = ( Id_as_Factor id_name, tokens_after_id))} \)

Parsing Factor as Parenthesized Expression

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

\( | \text{ factor ( Left_parenthesis} :: \text{tokens) = (match expr tokens with ( expr_parse , tokens_after_expr) -> ( Parenthesized_Expr_as_Factor expr_parse , tokens_after_expr))} \)

Error Cases

- What if no matching right parenthesis?
  \( | \_ \rightarrow \text{raise (Failure "No matching rparen")}) \)

- What if no leading id or left parenthesis?
  \( | \_ \rightarrow \text{raise (Failure "No id or lparen")}) \)
(a + b) * c - d

```plaintext
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

```plaintext
- : 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

```plaintext
# expr [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

```plaintext
+ : 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"))));
[])
```

(a + b) * c - d

```plaintext
# expr [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

```plaintext
+ : 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"))));
[])
```

(a + b) * c - d

```plaintext
# 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.
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. Can 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

Pairwise Disjointedness Test

- For each rule  
  \[ A ::= y \]
  Calculate \( \text{FIRST}(y) = \{ a \mid y =>^* aw \} \cup \{ \varepsilon \mid y =>^* \varepsilon \} \)
- For each pair of rules  
  \[ A ::= y \] and  
  \[ A ::= z \]
  require \( \text{FIRST}(y) \cap \text{FIRST}(z) = \{ \} \)
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:
Transform grammar to:
<S> ::= <A> a <B> b <S> ::= <A> a <B> b
<A> ::= <A> b | b <A> ::= b<A1>
<B> ::= a <B> | a <A1> ::= b<A1> | ε
<B> ::= a<B1> | ε
<B1> ::= a<B1> | ε

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 (i.e., 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

- 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 $M$ 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

- \( I \in \text{Identifiers} \)
- \( N \in \text{Numerals} \)
- \( B ::= \text{true} \mid \text{false} \mid B \& B \mid B \lor B \mid \text{not} B \)
- \( E ::= N \mid I \mid E + E \mid E * E \mid E - E \mid - E \)
- \( C ::= \text{skip} \mid C;C \mid I ::= E \mid \text{if} B \text{ then } C \text{ else } C \text{ fi} \mid \text{while} B \text{ do } C \text{ od} \)

Natural Semantics of Atomic Expressions

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

Relations

- \((E, m) \Downarrow U \quad (E', m) \Downarrow V \quad U \sim V = b\)
- 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

- \((E, m) \Downarrow U\)
- \((E', m) \Downarrow V\)
- \(U \text{ op } V = N\)
- \((E \text{ op } E', m) \Downarrow N\)
- where \( N \) is the specified value for \( U \text{ op } V \)

Commands

- Skip: \((\text{skip}, m) \Downarrow m\)
- Assignment: \((E,m) \Downarrow V\)
- \((I::=E,m) \Downarrow m[I <-- V]\)
- Sequencing: \((C,m) \Downarrow m'\)
- \((C',m') \Downarrow m''\)
- \((C;C', m) \Downarrow m''\)
If Then Else Command

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

While Command

\[(B,m) \downarrow \text{false} \quad (\text{while } B \text{ do } C \text{ od, } m) \downarrow m \]
\[
\begin{align*}
(B,m) & \downarrow \text{true} \quad (C,m) \downarrow m' \quad (\text{while } B \text{ do } C \text{ od, } m') \downarrow m'' \\
(B,m) & \downarrow \text{false} \quad (C',m) \downarrow m' \quad (\text{while } B \text{ do } C \text{ od, } m) \downarrow m''
\end{align*}
\]

Example: If Then Else Rule

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

Example: Identifier(s)

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

Example: Arith Relation

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

\[ 7 > 5 = true \]
\[
\frac{(x,\langle x \rightarrow 7 \rangle) \cup 2 \quad (5,\langle x \rightarrow 7 \rangle) \cup 5}{(x > 5, \langle x \rightarrow 7 \rangle) \cup true}
\]
\[
\frac{(if\ x > 5\ then\ y := 2 + 3\ else\ y := 3 + 4\ fi, \langle x \rightarrow 7 \rangle) \cup ?}{?}
\]

**Example: If Then Else Rule**

\[ 7 > 5 = true \]
\[
\frac{(x,\langle x \rightarrow 7 \rangle) \cup 2 \quad (5,\langle x \rightarrow 7 \rangle) \cup 5}{(y := 2 + 3, \langle x \rightarrow 7 \rangle) \cup \langle x > 7 \rangle \cup true \quad \cup ?}
\]
\[
\frac{(if\ x > 5\ then\ y := 2 + 3\ else\ y := 3 + 4\ fi, \langle x \rightarrow 7 \rangle) \cup ?}{?}
\]

**Example: Assignment**

\[ 7 > 5 = true \]
\[
\frac{(x,\langle x \rightarrow 7 \rangle) \cup 2 \quad (5,\langle x \rightarrow 7 \rangle) \cup 5}{(x > 5, \langle x \rightarrow 7 \rangle) \cup true}
\]
\[
\frac{(if\ x > 5\ then\ y := 2 + 3\ else\ y := 3 + 4\ fi, \langle x \rightarrow 7 \rangle) \cup ?}{?}
\]

**Example: Arith Op**

\[ 2 + 3 = 5 \]
\[
\frac{(2,\langle x \rightarrow 7 \rangle) \cup 2 \quad (3,\langle x \rightarrow 7 \rangle) \cup 3}{(x,\langle x \rightarrow 7 \rangle) \cup 7 \quad (5,\langle x \rightarrow 7 \rangle) \cup 5}
\]
\[
\frac{(if\ x > 5\ then\ y := 2 + 3\ else\ y := 3 + 4\ fi, \langle x \rightarrow 7 \rangle) \cup ?}{?}
\]

**Example: Numerals**

\[ 2 + 3 = 5 \]
\[
\frac{(2,\langle x \rightarrow 7 \rangle) \cup 2 \quad (3,\langle x \rightarrow 7 \rangle) \cup 3}{(x,\langle x \rightarrow 7 \rangle) \cup 7 \quad (5,\langle x \rightarrow 7 \rangle) \cup 5}
\]
\[
\frac{(if\ x > 5\ then\ y := 2 + 3\ else\ y := 3 + 4\ fi, \langle x \rightarrow 7 \rangle) \cup ?}{?}
\]

**Example: Arith Op**

\[ 2 + 3 = 5 \]
\[
\frac{(2,\langle x \rightarrow 7 \rangle) \cup 2 \quad (3,\langle x \rightarrow 7 \rangle) \cup 3}{(x,\langle x \rightarrow 7 \rangle) \cup 7 \quad (5,\langle x \rightarrow 7 \rangle) \cup 5}
\]
\[
\frac{(if\ x > 5\ then\ y := 2 + 3\ else\ y := 3 + 4\ fi, \langle x \rightarrow 7 \rangle) \cup ?}{?}
\]
**Example: Assignment**

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

**Example: If Then Else Rule**

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

**Let in Command**

\[
\begin{align*}
\langle E, m \rangle &\Downarrow v \quad \langle C, m[I\leftarrow v]\rangle \Downarrow m' \\
\text{(let } I = E \text{ in } C, m) &\Downarrow m''
\end{align*}
\]

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.

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