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

Using Ocamlyacc

- Input attribute grammar is put in file <*grammar*>.mly
- Execute
  ocamlyacc <*grammar*>.mly
- Produces code for parser in
  <*grammar*>.ml
  and interface (including type declaration for tokens) in
  <*grammar*>.mli

Parser Code

- <*grammar*>.ml defines one parsing function per entry point
- Parsing function takes a lexing function (lexer buffer to token) and a lexer buffer as arguments
- Returns semantic attribute of corresponding entry point

Ocamlyacc Input

- File format:

%{
  <*header*>
%}
  <*declarations*>
%%
  <*rules*>
%%
  <*trailer*>

Ocamlyacc <*header*>

- Contains arbitrary Ocaml code
- Typically used to give types and functions needed for the semantic actions of rules and to give specialized error recovery
- May be omitted
- <*footer*> similar. Possibly used to call parser

Ocamlyacc <declarations>

- %token *symbol … symbol*
- Declare given symbols as tokens
- %token <*type*> *symbol … symbol*
- Declare given symbols as token constructors, taking an argument of type <*type*>
- %start *symbol … symbol*
- Declare given symbols as entry points; functions of same names in <*grammar*>.ml

Ocamlyacc <*declarations*>

- %type <*type*> *symbol … symbol*

  Specify type of attributes for given symbols. Mandatory for start symbols
- %left *symbol … symbol*
- %right *symbol … symbol*
- %nonassoc *symbol … symbol*

  Associate precedence and associativity to given symbols. Same line,same precedence; earlier line, lower precedence (broadest scope)

Ocamlyacc <*rules*>

- *nonterminal* :

  symbol ... symbol { *semantic_action* }

  | ...

  | *symbol ... symbol* { *semantic_action* }

  ;
- Semantic actions are arbitrary Ocaml expressions
- Must be of same type as declared (or inferred) for *nonterminal*
- Access semantic attributes (values) of symbols by position: $1 for first symbol, $2 to second …

Example - Base types

```
(* File: expr.ml *)
type expr =
    Term_as_Expr of term
  | Plus_Expr of (term * expr)
  | Minus_Expr of (term * expr)
and term =
    Factor_as_Term of factor
  | Mult_Term of (factor * term)
  | Div_Term of (factor * term)
and factor =
    Id_as_Factor of string
  | Parenthesized_Expr_as_Factor of expr
```


Example - Lexer (exprlex.mll)

```
{ (*open Exprparse*) }
let numeric = ['0' - '9']
let letter =['a' - 'z' 'A' - 'Z']
rule token = parse
  | "+" {Plus_token}
  | "-"  {Minus_token}
  | "*"  {Times_token}
  | "/"  {Divide_token}
  | "("  {Left_parenthesis}
  | ")"  {Right_parenthesis}
  | letter (letter|numeric|"_")* as id  {Id_token id}
  | [' ' '\t' '\n'] {token lexbuf}
  | eof {EOL}
```


Example - Parser (exprparse.mly)

```
%{ open Expr
%}
%token <string> Id_token
%token Left_parenthesis Right_parenthesis
%token Times_token Divide_token
%token Plus_token Minus_token
%token EOL
%start main
%type <expr> main
%%
```


Example - Parser (exprparse.mly)

```
expr:
   term
      { Term_as_Expr $1 }
 | term Plus_token expr
      { Plus_Expr ($1, $3) }
 | term Minus_token expr
      { Minus_Expr ($1, $3) }
```

Example - Parser (exprparse.mly)

```
term:
   factor
        { Factor_as_Term $1 }
 | factor Times_token term
        { Mult_Term ($1, $3) }
 | factor Divide_token term
        { Div_Term ($1, $3) }
```


Example - Parser (exprparse.mly)

```
factor:
   Id_token
        { Id_as_Factor $1 }
 | Left_parenthesis expr Right_parenthesis
        {Parenthesized_Expr_as_Factor $2 }
main:
 | expr EOL
        { $1 }
```


Example - Using Parser

```
# #use "expr.ml";;
…
# #use "exprparse.ml";;
…
# #use "exprlex.ml";;
…
# let test s =
  let lexbuf = Lexing.from_string (s^"\n") in
      main token lexbuf;;
```


Example - Using Parser

```
# test "a + b";;
- : expr =
Plus_Expr
 (Factor_as_Term (Id_as_Factor "a"),
  Term_as_Expr (Factor_as_Term
   (Id_as_Factor "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> =>

= ● (0 + 1) + 0 shift

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> =>

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

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

<Sum> =>

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

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

<Sum> =>

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

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

<Sum> =>

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

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

<Sum> =>

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

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

<Sum> =>

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

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

<Sum> =>

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

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

<Sum> =>

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

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

<Sum> =>

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

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

<Sum>

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

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

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

Example

(0 + 1) + 0

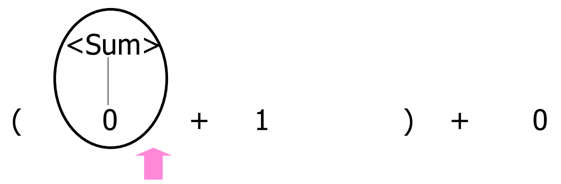
Example

(0 + 1) + 0

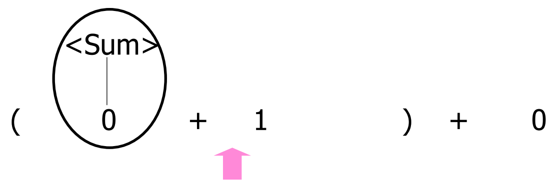
Example

(0 + 1) + 0

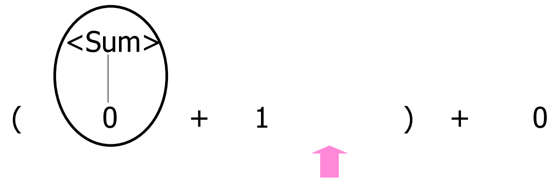
Example



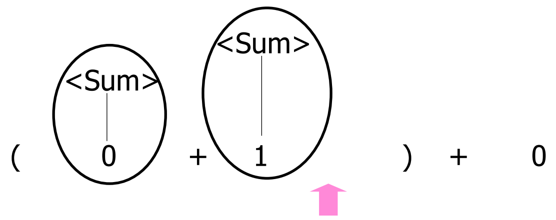
Example



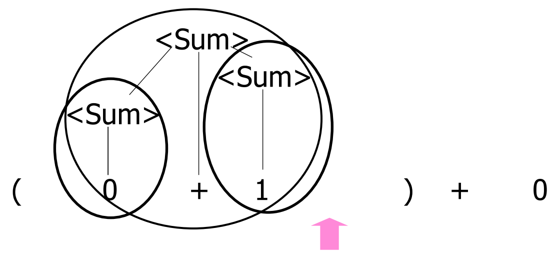
Example



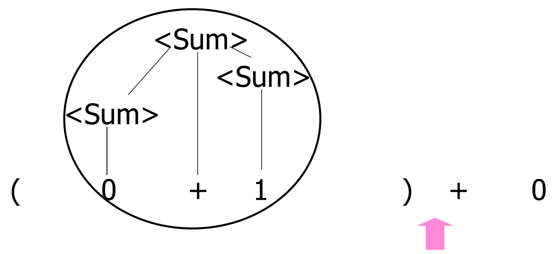
Example



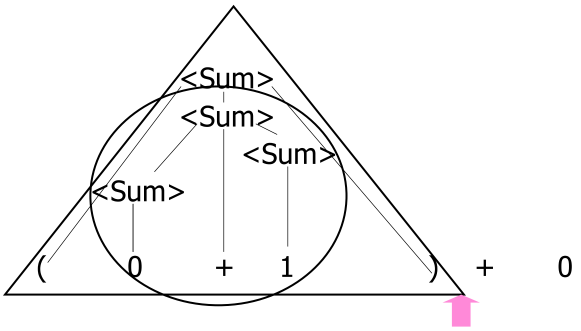
Example



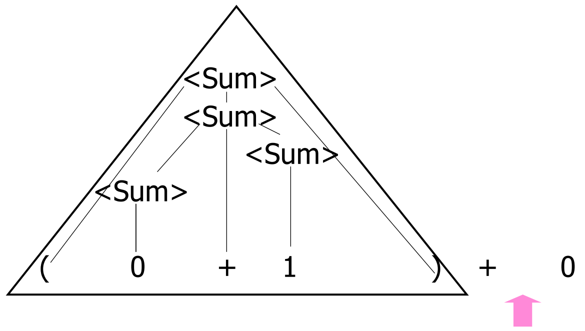
Example



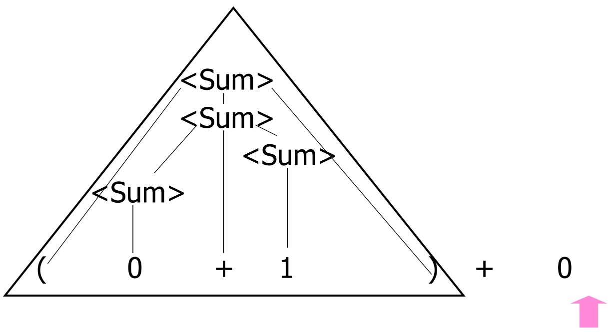
Example



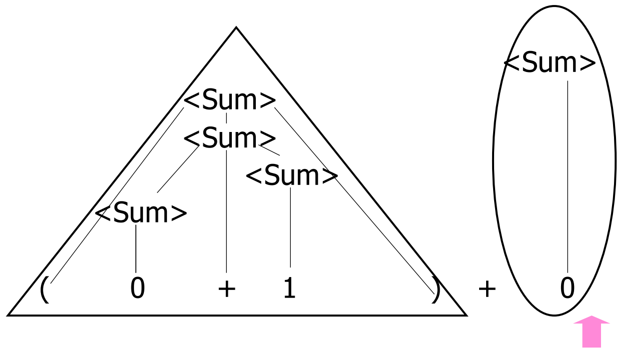
Example



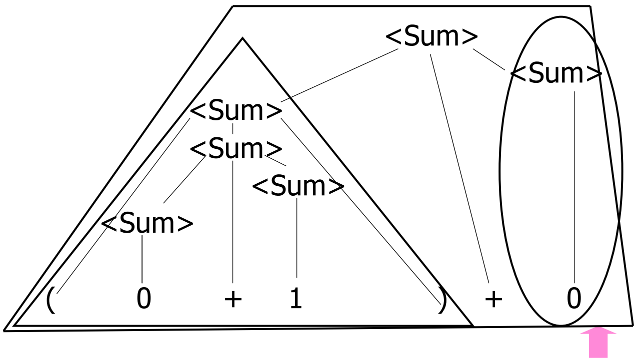
Example



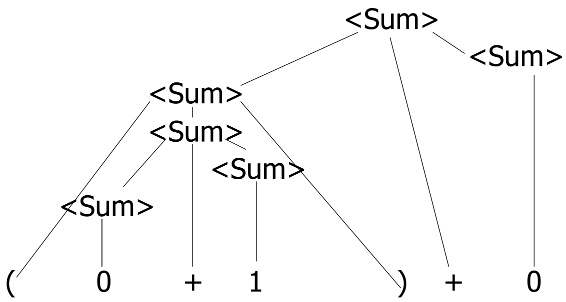
Example



Example



Example



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*)

LR(i) Parsing Algorithm

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

LR(i) Parsing Algorithm

6. If action = **reduce** *k* where production *k* is E ::= u
   a) Remove 2 * 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

LR(i) Parsing Algorithm

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

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

- S ::= A | aB   A ::= abc   B ::= bc

● abc	shift
a ● bc	shift
ab ● c	shift
abc ●	

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

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 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 ::= 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: $(\text{true},m) \Downarrow \text{true}$
  $(\text{false},m) \Downarrow \text{false}$

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 ~ hold on the meaning of U and V
- May be specified by a mathematical expression/equation or rules matching U and V

Arithmetic Expressions

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

where N is the specified value for U *op* V

Commands

Skip: (skip, m) ⇓ 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

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

Example: If Then Else Rule

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

Example: Arith Relation

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

Example: Identifier(s)

```
      7 > 5 = true
(x,{x->7})⇓7   (5,{x->7})⇓5
   (x > 5, {x -> 7})⇓?
  ----------------------------------------------
     (if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
                 {x -> 7}) ⇓ ?
```


Example: Arith Relation

```
      7 > 5 = true
(x,{x->7})⇓7   (5,{x->7})⇓5
   (x > 5, {x -> 7})⇓true
  ----------------------------------------------
     (if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
                 {x -> 7}) ⇓ ?
```


Example: If Then Else Rule

```
      7 > 5 = true                 ------------------
(x,{x->7})⇓7   (5,{x->7})⇓5        (y:= 2 + 3, {x-> 7}
  (x > 5, {x -> 7})⇓true            ⇓ ?             .
  ----------------------------------------------
     (if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
                 {x -> 7}) ⇓ ?
```


Example: Assignment

```
                              ---------------------------
      7 > 5 = true                   (2+3, {x->7})⇓?
(x,{x->7})⇓7   (5,{x->7})⇓5        (y:= 2 + 3, {x-> 7}
  (x > 5, {x -> 7})⇓true            ⇓ ?
  ----------------------------------------------
     (if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
                 {x -> 7}) ⇓ ?
```


Example: Arith Op

```
                                          ? + ? = ?
                              (2,{x->7})⇓?   (3,{x->7}) ⇓?
      7 > 5 = true                  (2+3, {x->7})⇓?
(x,{x->7})⇓7   (5,{x->7})⇓5        (y:= 2 + 3, {x-> 7}
  (x > 5, {x -> 7})⇓true            ⇓?              .
  ----------------------------------------------
     (if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
                 {x -> 7}) ⇓ ?
```


Example: Numerals

```
                                          2 + 3 = 5
                              (2,{x->7})⇓2   (3,{x->7}) ⇓3
      7 > 5 = true                  (2+3, {x->7})⇓?
(x,{x->7})⇓7   (5,{x->7})⇓5        (y:= 2 + 3, {x-> 7}
  (x > 5, {x -> 7})⇓true              ⇓ ?
  ----------------------------------------------
     (if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
                 {x -> 7}) ⇓ ?
```

Example: Arith Op

2 + 3 = 5
(2,{x->7})⇓2   (3,{x->7}) ⇓3
7 > 5 = true   (2+3, {x->7})⇓5
(x,{x->7})⇓7   (5,{x->7})⇓5   (y:= 2 + 3, {x-> 7}
(x > 5, {x -> 7})⇓true   ⇓?
(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7}) ⇓ ?

Example: Assignment

2 + 3 = 5
(2,{x->7})⇓2   (3,{x->7}) ⇓3
7 > 5 = true   (2+3, {x->7})⇓5
(x,{x->7})⇓7   (5,{x->7})⇓5   (y:= 2 + 3, {x-> 7}
(x > 5, {x -> 7})⇓true   ⇓ {x->7, y->5}
(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7}) ⇓?

Example: If Then Else Rule

2 + 3 = 5
(2,{x->7})⇓2   (3,{x->7}) ⇓3
7 > 5 = true   (2+3, {x->7})⇓5
(x,{x->7})⇓7   (5,{x->7})⇓5   (y:= 2 + 3, {x-> 7}
(x > 5, {x -> 7})⇓true   ⇓ {x->7, y->5}
(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7}) ⇓ {x->7, y->5}

Let in Command

(*E*,*m*) ⇓v   (*C*,*m*[*I*<-*v*]) ⇓ *m'*
(let *I* = *E* in *C*, *m*) ⇓ *m''*

Where *m''* (*y*) = *m'* (*y*) for *y*≠ *I* and
m'' (*I*) = *m* (*I*) if *m*(*I*) is defined,
and *m''* (*I*) is undefined otherwise

Example

(x,{x->5}) ⇓ 5   (3,{x->5}) ⇓ 3
(x+3,{x->5}) ⇓ 8
(5,{x->17}) ⇓ 5   (x:=x+3,{x->5}) ⇓ {x->8}
(let x = 5 in (x:=x+3), {x -> 17}) ⇓ ?

Example

(x,{x->5}) ⇓ 5   (3,{x->5}) ⇓ 3
(x+3,{x->5}) ⇓ 8
(5,{x->17}) ⇓ 5   (x:=x+3,{x->5}) ⇓ {x->8}
(let x = 5 in (x:=x+3), {x -> 17}) ⇓ {x->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