

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

11/5/15

1

Using Ocamllyacc

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

11/5/15

2

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

11/5/15

3

Ocamllyacc Input

- File format:

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

11/5/15

4

Ocamllyacc `<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

11/5/15

5

Ocamllyacc `<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`

11/5/15

6

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)

11/5/15

7

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 ...

11/5/15

8

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
```

11/5/15

9

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}
```

11/5/15

10

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
%%
```

11/5/15

11

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) }
```

11/5/15

12

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) }
```

11/5/15

13

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 }
```

11/5/15

14

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;;
```

11/5/15

15

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")))
```

11/5/15

16

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

11/5/15

17

Example: $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

$\langle \text{Sum} \rangle \Rightarrow$

$= (0 + 1) + 0$ shift

11/5/15

18

Example: $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

$\langle \text{Sum} \rangle \Rightarrow$

$= (0 + 1) + 0$ shift
 $= (0 + 1) + 0$ shift

11/5/15

19

Example: $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

$\langle \text{Sum} \rangle \Rightarrow$

$\Rightarrow (0 + 1) + 0$ reduce
 $= (0 + 1) + 0$ shift
 $= (0 + 1) + 0$ shift

11/5/15

20

Example: $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

$\langle \text{Sum} \rangle \Rightarrow$

$= (\langle \text{Sum} \rangle + 1) + 0$ shift
 $\Rightarrow (0 + 1) + 0$ reduce
 $= (0 + 1) + 0$ shift
 $= (0 + 1) + 0$ shift

11/5/15

21

Example: $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

$\langle \text{Sum} \rangle \Rightarrow$

$= (\langle \text{Sum} \rangle + 1) + 0$ shift
 $= (\langle \text{Sum} \rangle + 1) + 0$ shift
 $\Rightarrow (0 + 1) + 0$ reduce
 $= (0 + 1) + 0$ shift
 $= (0 + 1) + 0$ shift

11/5/15

22

Example: $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

$\langle \text{Sum} \rangle \Rightarrow$

$\Rightarrow (\langle \text{Sum} \rangle + 1) + 0$ reduce
 $= (\langle \text{Sum} \rangle + 1) + 0$ shift
 $= (\langle \text{Sum} \rangle + 1) + 0$ shift
 $\Rightarrow (0 + 1) + 0$ reduce
 $= (0 + 1) + 0$ shift
 $= (0 + 1) + 0$ shift

11/5/15

23

Example: $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

$\langle \text{Sum} \rangle \Rightarrow$

$\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + 0$ reduce
 $\Rightarrow (\langle \text{Sum} \rangle + 1) + 0$ reduce
 $= (\langle \text{Sum} \rangle + 1) + 0$ shift
 $= (\langle \text{Sum} \rangle + 1) + 0$ shift
 $\Rightarrow (0 + 1) + 0$ reduce
 $= (0 + 1) + 0$ shift
 $= (0 + 1) + 0$ shift

11/5/15

24

Example: $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

$\langle \text{Sum} \rangle \bullet \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \bullet$	reduce
$\Rightarrow \langle \text{Sum} \rangle + 0 \bullet$	reduce
$= \langle \text{Sum} \rangle + \bullet 0$	shift
$= \langle \text{Sum} \rangle \bullet + 0$	shift
$\Rightarrow (\langle \text{Sum} \rangle) \bullet + 0$	reduce
$= (\langle \text{Sum} \rangle \bullet) + 0$	shift
$\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle \bullet) + 0$	reduce
$\Rightarrow (\langle \text{Sum} \rangle + 1 \bullet) + 0$	reduce
$= (\langle \text{Sum} \rangle + \bullet 1) + 0$	shift
$= (\langle \text{Sum} \rangle \bullet + 1) + 0$	shift
$\Rightarrow (0 \bullet + 1) + 0$	reduce
$= (\bullet 0 + 1) + 0$	shift
$= \bullet (0 + 1) + 0$	shift

11/5/15

31

Example

(0 + 1) + 0



11/5/15

32

Example

(0 + 1) + 0



11/5/15

33

Example

(0 + 1) + 0



11/5/15

34

Example

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



11/5/15

35

Example

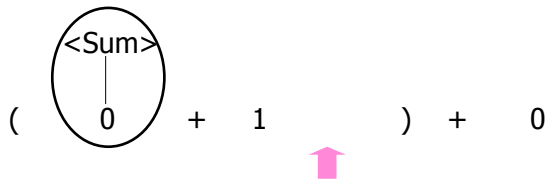
($\langle \text{Sum} \rangle$
0 + 1) + 0



11/5/15

36

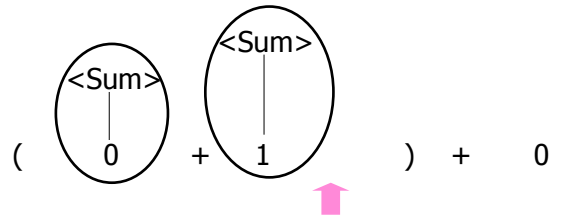
 Example



11/5/15

37

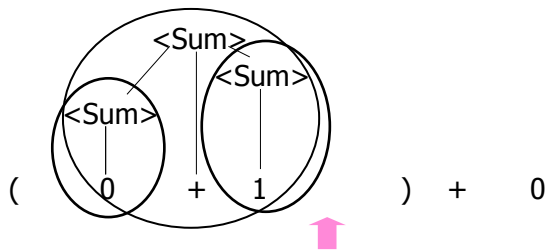
 Example



11/5/15

38

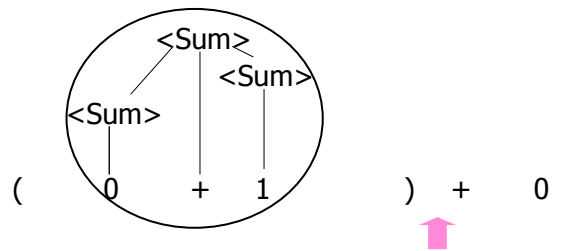
 Example



11/5/15

39

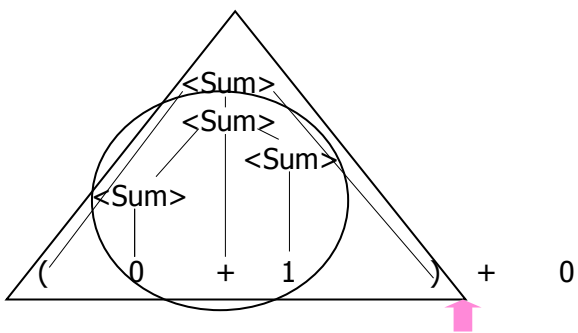
 Example



11/5/15

40

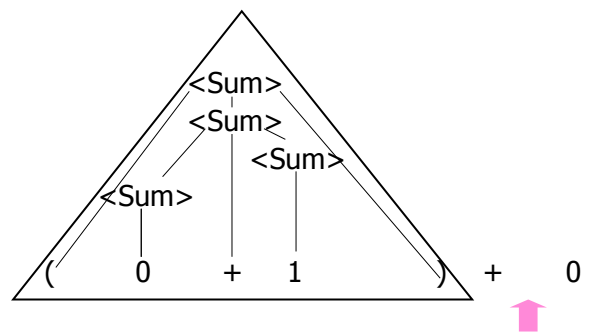
 Example



11/5/15

41

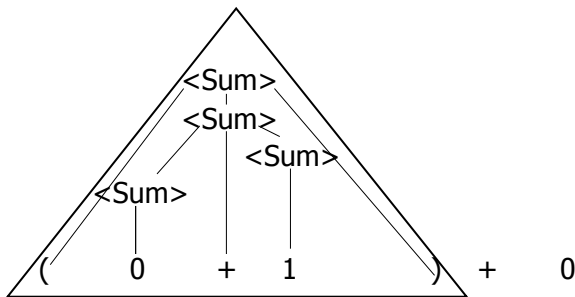
 Example



11/5/15

42

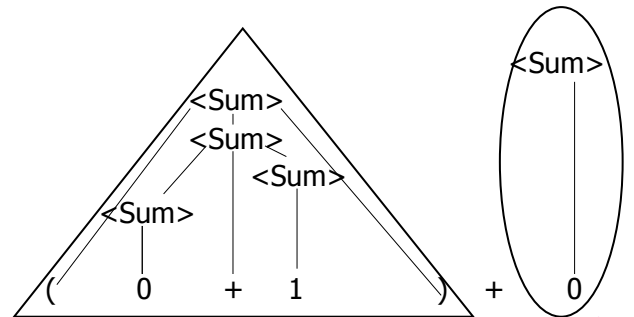
Example



11/5/15

43

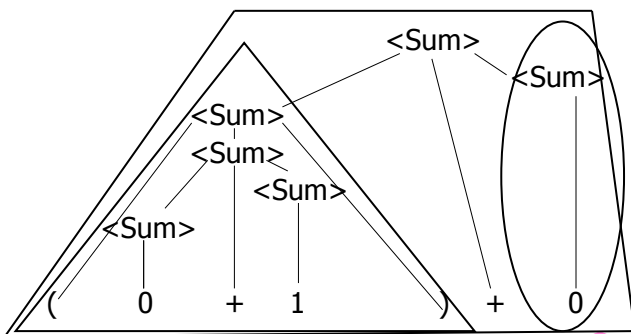
Example



11/5/15

44

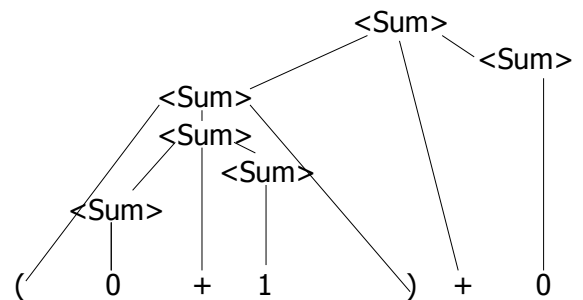
Example



11/5/15

45

Example



11/5/15

46

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

11/5/15

47

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

11/5/15

48

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

11/5/15

49

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 ($n, toks$)

11/5/15

50

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

11/5/15

51

LR(i) Parsing Algorithm

6. If action = **reduce** k where production k is $E ::= u$
 - a) Remove $2 * \text{length}(u)$ 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

11/5/15

52

LR(i) Parsing Algorithm

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

11/5/15

53

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

11/5/15

54

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

11/5/15

55

Example: $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

$\bullet 0 + 1 + 0$ shift
 $\rightarrow 0 \bullet + 1 + 0$ reduce
 $\rightarrow \langle \text{Sum} \rangle \bullet + 1 + 0$ shift
 $\rightarrow \langle \text{Sum} \rangle + \bullet 1 + 0$ shift
 $\rightarrow \langle \text{Sum} \rangle + 1 \bullet + 0$ reduce
 $\rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \bullet + 0$

11/5/15

56

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

11/5/15

57

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

11/5/15

58

Example

■ $S ::= A \mid aB$ $A ::= abc$ $B ::= bc$

$\bullet abc$ shift
 $a \bullet bc$ shift
 $ab \bullet c$ shift
 $abc \bullet$

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

11/5/15

59

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)

11/5/15

60

Recursive Descent Parsing

- 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

11/5/15

61

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

11/5/15

62

Sample Grammar

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

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

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

11/5/15

63

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
```

11/5/15

64

## 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)
```

11/5/15

65

## 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)
```

11/5/15

66

## Parse Trees as Datatypes

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

and factor =

Id\_as\_Factor of string

| Parenthesized\_Expr\_as\_Factor of expr

11/5/15

67

## 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

11/5/15

68

## 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) ->

11/5/15

69

## 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) ->

11/5/15

70

## 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) ->

11/5/15

71

## 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) ->

11/5/15

72

## 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) ->
```

11/5/15

73

## 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))
```

11/5/15

74

## 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))
```

11/5/15

75

## 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))
```

11/5/15

76

## 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))
```

11/5/15

77

## 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))
```

11/5/15

78

## Parsing an Expression as a Term

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle$

| \_ -> (Term\_as\_Expr **term\_parse** ,  
tokens\_after\_term))

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

11/5/15

79

## Parsing Factor as Id

$\langle \text{factor} \rangle ::= \langle \text{id} \rangle$

and factor tokens =  
(match tokens  
with (Id\_token id\_name :: tokens\_after\_id) =  
( **Id\_as\_Factor** id\_name, tokens\_after\_id)

11/5/15

80

## Parsing Factor as Parenthesized Expression

$\langle \text{factor} \rangle ::= ( \langle \text{expr} \rangle )$

| factor ( **Left\_parenthesis** :: tokens) =  
(match expr tokens  
with ( **expr\_parse** , tokens\_after\_expr) ->

11/5/15

81

## Parsing Factor as Parenthesized Expression

$\langle \text{factor} \rangle ::= ( \langle \text{expr} \rangle )$

(match tokens\_after\_expr  
with **Right\_parenthesis** :: tokens\_after\_rparen ->  
( **Parenthesized\_Expr\_as\_Factor** **expr\_parse** ,  
tokens\_after\_rparen)

11/5/15

## 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" );;

11/5/15

83

$( a + b ) * c - d$

expr [Left\_parenthesis; Id\_token "a";  
Plus\_token; Id\_token "b";  
Right\_parenthesis; Times\_token;  
Id\_token "c"; Minus\_token;  
Id\_token "d"];;

11/5/15

84

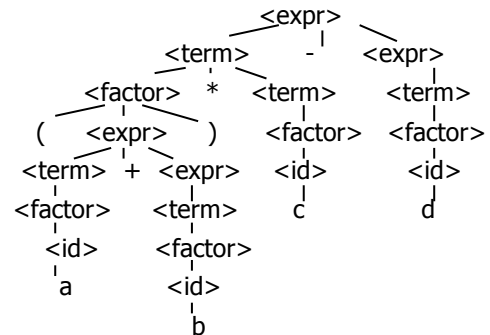
( a + b ) \* c - d

```
- : 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")))),
 [])
```

11/5/15

85

( a + b ) \* c - d



11/5/15

86

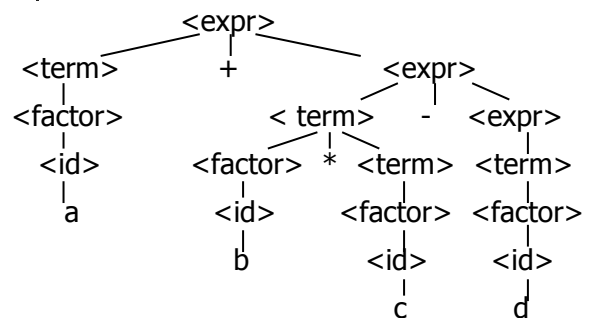
a + b \* c - d

```
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")))),
 [])
```

11/5/15

87

a + b \* c - d



11/5/15

88

( 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

11/5/15

89

( 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"])
```

11/5/15

90

## 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

11/5/15

91

## 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

11/5/15

92

## 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

11/5/15

93

## 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

11/5/15

94

## Pairwise Disjointedness Test

- For each rule  
 $A ::= y$   
Calculate  
 $\text{FIRST}(y) = \{a \mid y \Rightarrow^* aw\} \cup \{\epsilon \mid y \Rightarrow^* \epsilon\}$
- For each pair of rules  $A ::= y$  and  $A ::= z$ , require  $\text{FIRST}(y) \cap \text{FIRST}(z) = \{\}$

11/5/15

95

## Example

Grammar:  
 $\langle S \rangle ::= \langle A \rangle a \langle B \rangle b$   
 $\langle A \rangle ::= \langle A \rangle b \mid b$   
 $\langle B \rangle ::= a \langle B \rangle \mid a$

$\text{FIRST}(\langle A \rangle b) = \{b\}$   
 $\text{FIRST}(b) = \{b\}$   
Rules for  $\langle A \rangle$  not pairwise disjoint

11/5/15

96



## Eliminating Left Recursion

- Rewrite grammar to shift left recursion to right recursion
  - Changes associativity
- Given
$$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{term} \rangle$$
 and
$$\langle \text{expr} \rangle ::= \langle \text{term} \rangle$$
- Add new non-terminal  $\langle e \rangle$  and replace above rules with
$$\langle \text{expr} \rangle ::= \langle \text{term} \rangle \langle e \rangle$$
$$\langle e \rangle ::= + \langle \text{term} \rangle \langle e \rangle \mid \varepsilon$$

11/5/15

97

## Factoring Grammar

- Test too strong: Can't handle
$$\langle \text{expr} \rangle ::= \langle \text{term} \rangle [ ( + \mid - ) \langle \text{expr} \rangle ]$$
- Answer: Add new non-terminal and replace above rules by
$$\langle \text{expr} \rangle ::= \langle \text{term} \rangle \langle e \rangle$$
$$\langle e \rangle ::= + \langle \text{term} \rangle \langle e \rangle$$
$$\langle e \rangle ::= - \langle \text{term} \rangle \langle e \rangle$$
$$\langle e \rangle ::= \varepsilon$$
- You are delaying the decision point

11/5/15

98

## Example

Both  $\langle A \rangle$  and  $\langle B \rangle$  have problems: Transform grammar to:

$$\begin{aligned} \langle S \rangle &::= \langle A \rangle a \langle B \rangle b & \langle S \rangle &::= \langle A \rangle a \langle B \rangle b \\ \langle A \rangle &::= \langle A \rangle b \mid b & \langle A \rangle &::= b \langle A \rangle \\ \langle B \rangle &::= a \langle B \rangle \mid a & \langle A \rangle &::= b \langle A \rangle \mid \varepsilon \\ & & \langle B \rangle &::= a \langle B \rangle \\ & & \langle B \rangle &::= a \langle B \rangle \mid \varepsilon \end{aligned}$$

11/5/15

99

## 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

11/5/15

100

## Dynamic semantics

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

11/5/15

101

## Dynamic Semantics

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

11/5/15

102

## 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

11/5/15

103

## 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

11/5/15

104

## 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*

11/5/15

105

## 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

11/5/15

106

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

11/5/15

107

## Simple Imperative Programming Language

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

11/5/15

108

## Natural Semantics of Atomic Expressions

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

11/5/15

109

## Booleans:

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

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

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

11/5/15

110

## 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  $V$

11/5/15

111

## 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  $U \text{ op } V$

11/5/15

112

## Commands

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

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

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

11/5/15

113

## If Then Else Command

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

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

11/5/15

114

## While Command

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

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

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

11/5/15

115

## Example: If Then Else Rule

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

11/5/15

116

## Example: If Then Else Rule

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

11/5/15

117

## Example: Arith Relation

$$\frac{\begin{array}{l} ? > ? = ? \\ (x, \{x \rightarrow 7\}) \Downarrow ? \quad (5, \{x \rightarrow 7\}) \Downarrow ? \\ (x > 5, \{x \rightarrow 7\}) \Downarrow ? \end{array}}{(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) \Downarrow ?}$$

11/5/15

118

## Example: Identifier(s)

$$\frac{\begin{array}{l} 7 > 5 = \text{true} \\ (x, \{x \rightarrow 7\}) \Downarrow 7 \quad (5, \{x \rightarrow 7\}) \Downarrow 5 \\ (x > 5, \{x \rightarrow 7\}) \Downarrow ? \end{array}}{(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) \Downarrow ?}$$

11/5/15

119

## Example: Arith Relation

$$\frac{\begin{array}{l} 7 > 5 = \text{true} \\ (x, \{x \rightarrow 7\}) \Downarrow 7 \quad (5, \{x \rightarrow 7\}) \Downarrow 5 \\ (x > 5, \{x \rightarrow 7\}) \Downarrow \text{true} \end{array}}{(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) \Downarrow ?}$$

11/5/15

120

### Example: If Then Else Rule

$$\begin{array}{c}
 7 > 5 = \text{true} \\
 \frac{(x, \{x > 7\}) \Downarrow 7 \quad (5, \{x > 7\}) \Downarrow 5 \quad (y := 2 + 3, \{x > 7\}) \Downarrow ?}{(x > 5, \{x \rightarrow 7\}) \Downarrow \text{true} \quad \Downarrow ?} \\
 \frac{}{\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) \Downarrow ?}
 \end{array}$$

11/5/15

121

### Example: Assignment

$$\begin{array}{c}
 7 > 5 = \text{true} \\
 \frac{(x, \{x > 7\}) \Downarrow 7 \quad (5, \{x > 7\}) \Downarrow 5 \quad \frac{(2+3, \{x > 7\}) \Downarrow ?}{(y := 2 + 3, \{x > 7\}) \Downarrow ?}}{(x > 5, \{x \rightarrow 7\}) \Downarrow \text{true} \quad \Downarrow ?} \\
 \frac{}{\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) \Downarrow ?}
 \end{array}$$

11/5/15

122

### Example: Arith Op

$$\begin{array}{c}
 ? + ? = ? \\
 \frac{(2, \{x > 7\}) \Downarrow ? \quad (3, \{x > 7\}) \Downarrow ?}{(2+3, \{x > 7\}) \Downarrow ?} \\
 \frac{7 > 5 = \text{true} \quad (y := 2 + 3, \{x > 7\}) \Downarrow ?}{(x, \{x > 7\}) \Downarrow 7 \quad (5, \{x > 7\}) \Downarrow 5 \quad (x > 5, \{x \rightarrow 7\}) \Downarrow \text{true} \quad \Downarrow ?} \\
 \frac{}{\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) \Downarrow ?}
 \end{array}$$

11/5/15

123

### Example: Numerals

$$\begin{array}{c}
 2 + 3 = 5 \\
 \frac{(2, \{x > 7\}) \Downarrow 2 \quad (3, \{x > 7\}) \Downarrow 3}{(2+3, \{x > 7\}) \Downarrow ?} \\
 \frac{7 > 5 = \text{true} \quad (y := 2 + 3, \{x > 7\}) \Downarrow ?}{(x, \{x > 7\}) \Downarrow 7 \quad (5, \{x > 7\}) \Downarrow 5 \quad (x > 5, \{x \rightarrow 7\}) \Downarrow \text{true} \quad \Downarrow ?} \\
 \frac{}{\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) \Downarrow ?}
 \end{array}$$

11/5/15

124

### Example: Arith Op

$$\begin{array}{c}
 2 + 3 = 5 \\
 \frac{(2, \{x > 7\}) \Downarrow 2 \quad (3, \{x > 7\}) \Downarrow 3}{(2+3, \{x > 7\}) \Downarrow 5} \\
 \frac{7 > 5 = \text{true} \quad (y := 2 + 3, \{x > 7\}) \Downarrow ?}{(x, \{x > 7\}) \Downarrow 7 \quad (5, \{x > 7\}) \Downarrow 5 \quad (x > 5, \{x \rightarrow 7\}) \Downarrow \text{true} \quad \Downarrow ?} \\
 \frac{}{\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) \Downarrow ?}
 \end{array}$$

11/5/15

125

### Example: Assignment

$$\begin{array}{c}
 2 + 3 = 5 \\
 \frac{(2, \{x > 7\}) \Downarrow 2 \quad (3, \{x > 7\}) \Downarrow 3}{(2+3, \{x > 7\}) \Downarrow 5} \\
 \frac{7 > 5 = \text{true} \quad (y := 2 + 3, \{x > 7\}) \Downarrow ?}{(x, \{x > 7\}) \Downarrow 7 \quad (5, \{x > 7\}) \Downarrow 5 \quad (x > 5, \{x \rightarrow 7\}) \Downarrow \text{true} \quad \Downarrow \{x > 7, y > 5\}} \\
 \frac{}{\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) \Downarrow ?}
 \end{array}$$

11/5/15

126

## Example: If Then Else Rule

$$\begin{array}{c}
 2 + 3 = 5 \\
 \frac{(2, \{x \rightarrow 7\}) \Downarrow 2 \quad (3, \{x \rightarrow 7\}) \Downarrow 3}{(2+3, \{x \rightarrow 7\}) \Downarrow 5} \\
 7 > 5 = \text{true} \quad \frac{(y := 2 + 3, \{x \rightarrow 7\}) \Downarrow 5}{(y := 2 + 3, \{x \rightarrow 7\}) \Downarrow 5} \\
 \frac{(x > 5, \{x \rightarrow 7\}) \Downarrow \text{true} \quad \frac{(2+3, \{x \rightarrow 7\}) \Downarrow 5}{(y := 2 + 3, \{x \rightarrow 7\}) \Downarrow 5}}{(x > 5, \{x \rightarrow 7\}) \Downarrow \text{true} \quad \Downarrow \{x \rightarrow 7, y \rightarrow 5\}} \\
 \frac{\quad}{(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) \Downarrow \{x \rightarrow 7, y \rightarrow 5\}}
 \end{array}$$

11/5/15

127

## Let in Command

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

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

11/5/15

128

## Example

$$\begin{array}{c}
 \frac{(x, \{x \rightarrow 5\}) \Downarrow 5 \quad (3, \{x \rightarrow 5\}) \Downarrow 3}{(x+3, \{x \rightarrow 5\}) \Downarrow 8} \\
 \frac{(5, \{x \rightarrow 17\}) \Downarrow 5 \quad \frac{(x := x+3, \{x \rightarrow 5\}) \Downarrow \{x \rightarrow 8\}}{(x := x+3, \{x \rightarrow 5\}) \Downarrow \{x \rightarrow 8\}}}{(\text{let } x = 5 \text{ in } (x := x+3), \{x \rightarrow 17\}) \Downarrow ?}
 \end{array}$$

11/5/15

129

## Example

$$\begin{array}{c}
 \frac{(x, \{x \rightarrow 5\}) \Downarrow 5 \quad (3, \{x \rightarrow 5\}) \Downarrow 3}{(x+3, \{x \rightarrow 5\}) \Downarrow 8} \\
 \frac{(5, \{x \rightarrow 17\}) \Downarrow 5 \quad \frac{(x := x+3, \{x \rightarrow 5\}) \Downarrow \{x \rightarrow 8\}}{(x := x+3, \{x \rightarrow 5\}) \Downarrow \{x \rightarrow 8\}}}{(\text{let } x = 5 \text{ in } (x := x+3), \{x \rightarrow 17\}) \Downarrow \{x \rightarrow 17\}}
 \end{array}$$

11/5/15

130

## 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

11/5/15

131

## 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

11/5/15

132

## 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

11/5/15

133

## 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

11/5/15

134

## Natural Semantics Example

- $\text{compute\_exp}(\text{Var}(v), m) = \text{look\_up } v \text{ } m$
- $\text{compute\_exp}(\text{Int}(n), \_) = \text{Num}(n)$
- ...
- $\text{compute\_com}(\text{IfExp}(b, c1, c2), m) =$   
if  $\text{compute\_exp}(b, m) = \text{Bool}(\text{true})$   
then  $\text{compute\_com}(c1, m)$   
else  $\text{compute\_com}(c2, m)$

11/5/15

135

## Natural Semantics Example

- $\text{compute\_com}(\text{While}(b, c), m) =$   
if  $\text{compute\_exp}(b, m) = \text{Bool}(\text{false})$   
then  $m$   
else  $\text{compute\_com}(\text{While}(b, c), \text{compute\_com}(c, m))$
- May fail to terminate - exceed stack limits
- Returns no useful information then

11/5/15

136