

# Lecture 9: Bottom-up parsing; ocamlyacc

- Bottom-up parsing
- ocamlyacc

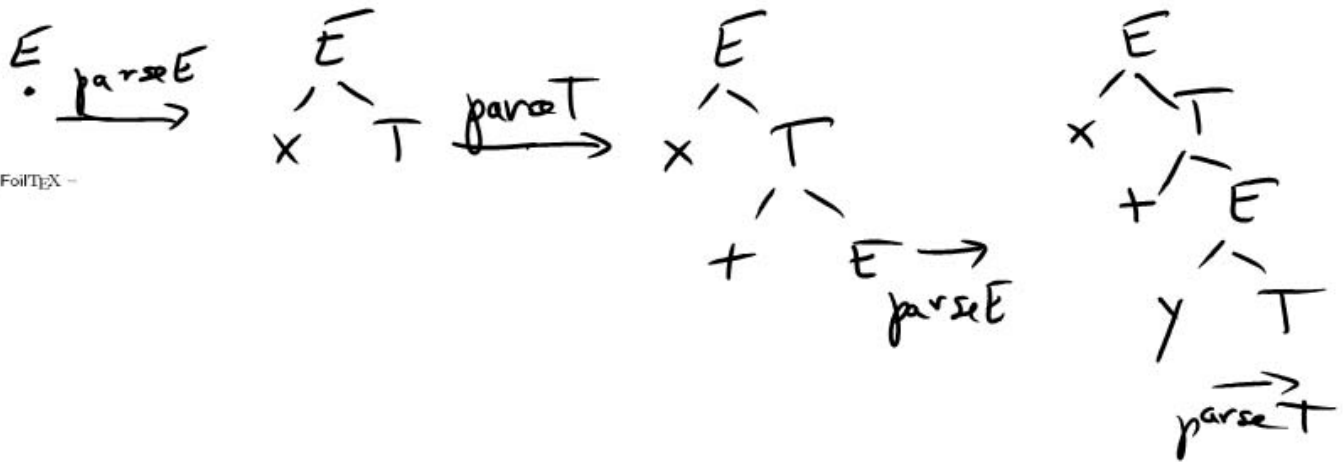
# Top-down vs. bottom-up parsing

- Why is top-down called "top-down"?

As we consume tokens, we build a parse tree. At any time, we are filling in the children of a particular non-terminal. As soon as we decide what production to use, we can fill in the tree. In this sense, we are building the tree from the top down.

- Example:  $E \rightarrow id T$   
 $T \rightarrow \epsilon \mid + E \mid * E$

Input:  $x + y * z$



# Bottom-up parsing

- Bottom-up parsing works by creating small parse trees and joining them together into larger ones.

- Example:  $E \rightarrow id \ T$   
 $T \rightarrow \epsilon \mid + \ E \mid * \ E$

Input:  $x + y * z$

1. Construct  $\cdot x$

2. Add  $+$  :  $\cdot x \quad \cdot +$

3. Add  $y$  :  $\cdot x \quad \cdot + \quad \cdot y$

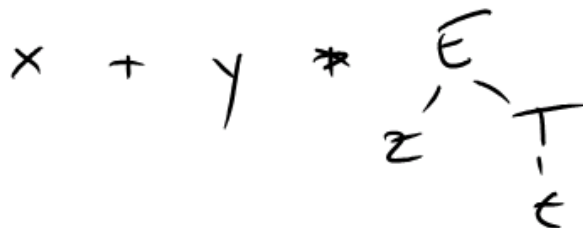
4. "  $*$  :  $\cdot x \quad \cdot + \quad \cdot y \quad \cdot *$

5. "  $z$  :  $x \quad + \quad y \quad * \quad z$

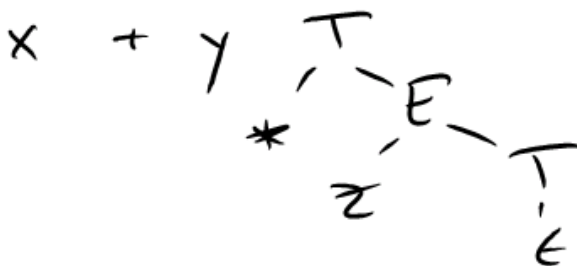
6. Use  $T \rightarrow \epsilon$   $x \quad + \quad y \quad * \quad z \quad T$   
- Typeset by FoilTeX -

7.  $E \rightarrow \epsilon \ T$   $x \quad + \quad y \quad * \quad z \quad E$   
 $T \rightarrow \epsilon$

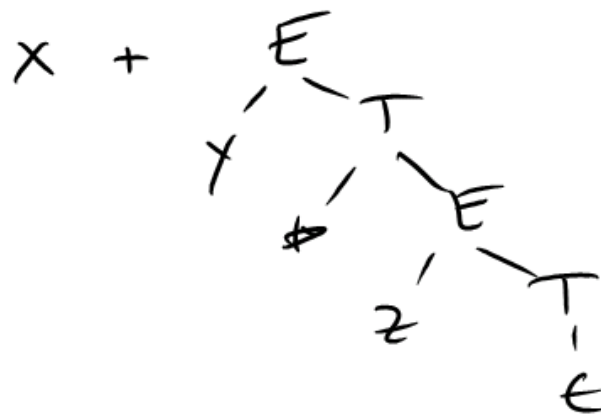
7.  $E \rightarrow \alpha T$



8. Use  $T \rightarrow *E$



9. Use  $E \rightarrow \alpha T$



10.

11.

# How bottom-up parsing works

- Keep a stack of small parse trees. Based on what's in this stack, and the next input token, take one of these actions:
  - Shift: Move lookahead token to stack
  - Reduce  $A \rightarrow \alpha$ : If roots of trees on stack match  $\alpha$ , replace those trees on stack by single tree with root  $A$ .
  - Accept: Reduce when non-terminal is goal, look-ahead is eof
  - Reject
- Bottom-up parsing is also called *shift-reduce parsing*.

# Shift-reduce example 1

- **Example:**  $L \rightarrow L ; E \mid E$   
 $E \rightarrow id$

Input: x; y; z

<u>Action</u>	<u>Stack</u>	<u>Input</u>
Sh		x; y; z
R $E \rightarrow id$	x	; y; z
R $L \rightarrow E$	E   x	y; z
Sh	L   E   x	; z

# Shift-reduce example 1

- **Example:**  $L \rightarrow L ; E \mid E$   
 $E \rightarrow id$

Input: x; y; z

Sh  $\begin{array}{c} L \\ | \\ E \\ | \\ X \end{array} ; \quad y ; z$

---

R  $E \rightarrow id$   $\begin{array}{c} L \\ | \\ E \\ | \\ X \end{array} ; \quad y \quad ; z$

---

R  $L \rightarrow L ; E$   $\begin{array}{c} L \\ | \\ E \\ | \\ X \end{array} ; \quad E \\ | \\ Y \quad ; z$

# Shift-reduce example 1

- Example:  $L \rightarrow L; E \mid E$   
 $E \rightarrow id$

Input: x; y; z

Sh



, z

Sh

" ; z

R  $E \rightarrow id$

" ; z

R  $L \rightarrow L; E$

" ; E z

- Typeset by Foil[1]2X -

Acc



eof





# Using ocaml yacc

- **Input attribute grammar is put in file `<grammar>.mly`**
- **Execute `ocaml yacc <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**

# Example - expression grammar

In this example, we will take a simple expression grammar and create a parser to parse inputs and produce abstract syntax.

*Grammar:*

$$M \rightarrow \text{Exp eof}$$

$$\text{Exp} \rightarrow \text{Term} \mid \text{Term} + \text{Exp} \mid \text{Term} - \text{Exp}$$

$$\text{Term} \rightarrow \text{Factor} \mid \text{Factor} * \text{Term} \mid \text{Factor} / \text{Term}$$

$$\text{Factor} \rightarrow \text{id} \mid ( \text{Exp} )$$

*Abstract syntax:*

```
(* File: expr.ml *)
```

```
type expr =  
  Plus of expr * expr  
| Minus of expr * expr  
| Mult of expr * expr  
| Div of expr * expr  
| Id of string
```

# Example - lexer

```
(* File: exprlex.mll *)
let numeric = ['0' - '9']
let letter = ['a' - 'z' 'A' - 'Z']
rule tokenize = 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

```
(* File: 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                {$1}
| term Plus_token  expr  {Plus($1,$3)}
| term Minus_token expr  {Minus($1,$3)}
```

term:

```
factor              {$1}
| factor Times_token term  {Mult($1,$3)}
| factor Divide_token term {Div($1,$3)}
```

factor:

```
Id_token {Id $1}
| Left_parenthesis expr Right_parenthesis {$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 tokenize lexbuf;;
# test "a + b";;
- : expr = Plus(Id "a",Id "b")
```



# 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 symbol**
- `%left symbol ... symbol`
- `%right symbol ... symbol`
- `%nonassoc symbol ... symbol`  
**Associate precedences and associativities 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 values semantic attributes of symbols by position: \$1 for first symbol, \$2 for second, etc.**

# Friday's class

- **Big question:** how to choose whether to shift or reduce.
- **ocamlyacc** uses a method — called *LALR(1)* — to construct tables which say what action to take.
- There are times when there is no good way to make this decision. (ocamlyacc will reject grammar and give an error message.) In bottom-up parsing, these are called *conflicts*. There are two types: shift/reduce and reduce/reduce.
- As with top-down parsing, these problems can sometimes be resolved by modifying the grammar.
- On Friday, will discuss these conflicts and give some advice on how to resolve them.

# MP 6

- MP 6 starts with a grammar embedded in an incomplete ocaml yacc specification. You will need to finish the spec:
  - Remove “extended BNF” productions - ocaml yacc cannot handle them
  - Resolve grammar conflicts
  - Fill in actions so as to produce ASTs.

Arg list  $\rightarrow ((\text{Type Ident})^+)?$

