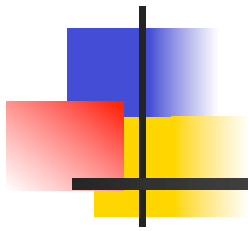


Programming Languages and Compilers (CS 421)



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



Example Regular Expressions

- **$(0 \vee 1)^* 1$**
 - The set of all strings of **0**'s and **1**'s ending in 1, **$\{1, 01, 11, \dots\}$**
- **$a^* b (a^*)$**
 - The set of all strings of a's and b's with exactly one b
- **$((01) \vee (10))^*$**
 - You tell me
- Regular expressions (equivalently, regular grammars) important for lexing, breaking strings into recognized words



Regular Grammars

- Subclass of BNF (covered in detail sool)
- Only rules of form
 $\langle \text{nonterminal} \rangle ::= \langle \text{terminal} \rangle \langle \text{nonterminal} \rangle$ or
 $\langle \text{nonterminal} \rangle ::= \langle \text{terminal} \rangle$ or
 $\langle \text{nonterminal} \rangle ::= \varepsilon$
- Defines same class of languages as regular expressions
- Important for writing lexers (programs that convert strings of characters into strings of tokens)
- Close connection to nondeterministic finite state automata – nonterminals \cong states; rule \cong edge



Example

- Regular grammar:

$\langle \text{Balanced} \rangle ::= \varepsilon$

$\langle \text{Balanced} \rangle ::= 0 \langle \text{OneAndMore} \rangle$

$\langle \text{Balanced} \rangle ::= 1 \langle \text{ZeroAndMore} \rangle$

$\langle \text{OneAndMore} \rangle ::= 1 \langle \text{Balanced} \rangle$

$\langle \text{ZeroAndMore} \rangle ::= 0 \langle \text{Balanced} \rangle$

- Generates even length strings where every initial substring of even length has same number of 0's as 1's



Example: Lexing

- Regular expressions good for describing lexemes (words) in a programming language
 - Identifier = $(a \vee b \vee \dots \vee z \vee A \vee B \vee \dots \vee Z) (a \vee b \vee \dots \vee z \vee A \vee B \vee \dots \vee Z \vee 0 \vee 1 \vee \dots \vee 9)^*$
 - Digit = $(0 \vee 1 \vee \dots \vee 9)$
 - Number = $0 \vee (1 \vee \dots \vee 9)(0 \vee \dots \vee 9)^* \vee \sim (1 \vee \dots \vee 9)(0 \vee \dots \vee 9)^*$
 - Keywords: if = if, while = while,...



Implementing Regular Expressions

- Regular expressions reasonable way to generate strings in language
- Not so good for recognizing when a string is in language
- Problems with Regular Expressions
 - which option to choose,
 - how many repetitions to make
- Answer: finite state automata
- Should have seen in CS374



Lexing

- Different syntactic categories of “words”:
tokens

Example:

- Convert sequence of characters into
sequence of strings, integers, and floating
point numbers.
- "asd 123 jkl 3.14" will become:
[String "asd"; Int 123; String "jkl"; Float
3.14]



Lex, ocamllex

- Could write the reg exp, then translate to DFA by hand
 - A lot of work
- Better: Write program to take reg exp as input and automatically generates automata
- Lex is such a program
- ocamllex version for ocaml



How to do it

- To use regular expressions to parse our input we need:
 - Some way to identify the input string — call it a lexing buffer
 - Set of regular expressions,
 - Corresponding set of actions to take when they are matched.



How to do it

- The lexer will take the regular expressions and generate a state machine.
- The state machine will take our lexing buffer and apply the transitions...
- If we reach an accepting state from which we can go no further, the machine will perform the appropriate action.



Mechanics

- Put table of reg exp and corresponding actions (written in ocaml) into a file *<filename>.ml*
- Call

```
ocamllex <filename>.ml
```
- Produces Ocaml code for a lexical analyzer in file *<filename>.ml*



Sample Input

```
rule main = parse
  ['0'-'9']+ { print_string "Int\n"}
  | ['0'-'9']+ '.' ['0'-'9']+ { print_string "Float\n"}
  | ['a'-'z']+ { print_string "String\n"}
  | _ { main lexbuf }
  {
  let newlexbuf = (Lexing.from_channel stdin) in
  print_string "Ready to lex.\n";
  main newlexbuf
  }
```



General Input

{ *header* }

let *ident* = *regex* ...

rule *entrypoint* [*arg1*... *argn*] = parse
 regex { *action* }

| ...

| *regex* { *action* }

and *entrypoint* [*arg1*... *argn*] =
 parse ...and ...

{ *trailer* }



Ocamlex Input

- *header* and *trailer* contain arbitrary ocaml code put at top and bottom of *<filename>.ml*
- `let ident = regexp ...` Introduces *ident* for use in later regular expressions



Ocamlex Input

- *<filename>.ml* contains one lexing function per *entrypoint*
 - Name of function is name given for *entrypoint*
 - Each entry point becomes an Ocaml function that takes $n+1$ arguments, the extra implicit last argument being of type `Lexing.lexbuf`
- *arg1... argn* are for use in *action*



Ocamlex Regular Expression

- Single quoted characters for letters:
‘a’
- `_`: (underscore) matches any letter
- `Eof`: special “end_of_file” marker
- Concatenation same as usual
- “*string*”: concatenation of sequence of characters
- e_1 / e_2 : choice - what was $e_1 \vee e_2$



Ocamlex Regular Expression

- $[c_1 - c_2]$: choice of any character between first and second inclusive, as determined by character codes
- $[\wedge c_1 - c_2]$: choice of any character NOT in set
- e^* : same as before
- $e+$: same as $e e^*$
- $e?$: option - was $e_1 \vee \varepsilon$



Ocamlex Regular Expression

- $e_1 \# e_2$: the characters in e_1 but not in e_2 ; e_1 and e_2 must describe just sets of characters
- *ident*: abbreviation for earlier reg exp in `let ident = regexp`
- e_1 *as* *id*: binds the result of e_1 to *id* to be used in the associated *action*



Ocamllex Manual

- More details can be found at

[http://caml.inria.fr/pub/docs/manual-ocaml/
manual026.html](http://caml.inria.fr/pub/docs/manual-ocaml/manual026.html)



Example : test.ml

```
{ type result = Int of int | Float of float |  
  String of string }
```

```
let digit = ['0'-'9']
```

```
let digits = digit +
```

```
let lower_case = ['a'-'z']
```

```
let upper_case = ['A'-'Z']
```

```
let letter = upper_case | lower_case
```

```
let letters = letter +
```



Example : test.ml

```
rule main = parse
  (digits)'.'digits as f { Float (float_of_string f) }
| digits as n           { Int (int_of_string n) }
| letters as s          { String s}
| _ { main lexbuf }
{ let newlexbuf = (Lexing.from_channel stdin) in
  print_string "Ready to lex.";
  print_newline ();
  main newlexbuf }
```



Example

```
# #use "test.ml";;
```

```
...
```

```
val main : Lexing.lexbuf -> result = <fun>
```

```
val __ocaml_lex_main_rec : Lexing.lexbuf -> int ->  
  result = <fun>
```

Ready to lex.

hi there 234 5.2

```
- : result = String "hi"
```

What happened to the rest?!?



Example

```
# let b = Lexing.from_channel stdin;;
```

```
# main b;;
```

```
hi 673 there
```

```
- : result = String "hi"
```

```
# main b;;
```

```
- : result = Int 673
```

```
# main b;;
```

```
- : result = String "there"
```



Your Turn

- Work on ML4
 - Add a few keywords
 - Implement booleans and unit
 - Implement Ints and Floats
 - Implement identifiers



Problem

- How to get lexer to look at more than the first token at one time?
- One Answer: *action* tells it to -- recursive calls
- Side Benefit: can add “state” into lexing
- Note: already used this with the _ case



Example

rule main = parse

(digits) '.' digits as f { Float

(float_of_string f) :: **main lexbuf**}

| digits as n { Int (int_of_string n) ::
main lexbuf }

| letters as s { String s :: **main lexbuf**}

| eof { [] }

| _ { **main lexbuf** }



Example Results

Ready to lex.

hi there 234 5.2

- : result list = [String "hi"; String "there"; Int 234; Float 5.2]

#

Used Ctrl-d to send the end-of-file signal



Dealing with comments

First Attempt

```
let open_comment = "("*
```

```
let close_comment = "*"
```

```
rule main = parse
```

```
  (digits) '.' digits as f { Float (float_of_string  
f) :: main lexbuf }
```

```
| digits as n          { Int (int_of_string n) ::  
main lexbuf }
```

```
| letters as s         { String s :: main lexbuf }
```



Dealing with comments

| **open_comment** { comment lexbuf }

| eof { [] }

| _ { main lexbuf }

and **comment** = parse

close_comment { main lexbuf }

| _ { **comment lexbuf** }



Dealing with nested comments

rule main = parse ...

| open_comment { comment 1 lexbuf }

| eof { [] }

| _ { main lexbuf }

and comment depth = parse

open_comment { comment (depth+1)
lexbuf }

| close_comment { if depth = 1
then main lexbuf
else comment (depth - 1) lexbuf }

| _ { comment depth lexbuf }



Dealing with nested comments

rule main = parse

(digits) '.' digits as f { Float (float_of_string f) ::
main lexbuf }

| digits as n { Int (int_of_string n) :: main
lexbuf }

| letters as s { String s :: main lexbuf }

| open_comment { (comment 1 lexbuf }

| eof { [] }

| _ { main lexbuf }



Dealing with nested comments

and comment depth = parse

open_comment { comment (depth+1) lexbuf }

| close_comment { if depth = 1

then main lexbuf

else comment (depth - 1) lexbuf }

| _ { comment depth lexbuf }



Types of Formal Language Descriptions

- Regular expressions, regular grammars
- Context-free grammars, BNF grammars, syntax diagrams
- Finite state automata

- Whole family more of grammars and automata – covered in automata theory



Sample Grammar

- Language: Parenthesized sums of 0's and 1's
- $\langle \text{Sum} \rangle ::= 0$
- $\langle \text{Sum} \rangle ::= 1$
- $\langle \text{Sum} \rangle ::= \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$
- $\langle \text{Sum} \rangle ::= (\langle \text{Sum} \rangle)$



BNF Grammars

- Start with a set of characters, **a,b,c,...**
 - We call these *terminals*
- Add a set of different characters, **X,Y,Z,**
...
 - We call these *nonterminals*
- One special nonterminal **S** called *start symbol*



BNF Grammars

- BNF rules (aka *productions*) have form

$$\mathbf{X} ::= y$$

where \mathbf{X} is any nonterminal and y is a string of terminals and nonterminals

- BNF *grammar* is a set of BNF rules such that every nonterminal appears on the left of some rule



Sample Grammar

- Terminals: 0 1 + ()
- Nonterminals: $\langle \text{Sum} \rangle$
- Start symbol = $\langle \text{Sum} \rangle$

- $\langle \text{Sum} \rangle ::= 0$
- $\langle \text{Sum} \rangle ::= 1$
- $\langle \text{Sum} \rangle ::= \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$
- $\langle \text{Sum} \rangle ::= (\langle \text{Sum} \rangle)$
- Can be abbreviated as
$$\langle \text{Sum} \rangle ::= 0 \mid 1 \mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \mid (\langle \text{Sum} \rangle)$$



BNF Derivations

- Given rules

$$\mathbf{X} ::= y\mathbf{Z}w \text{ and } \mathbf{Z} ::= v$$

we may replace \mathbf{Z} by v to say

$$\mathbf{X} \Rightarrow y\mathbf{Z}w \Rightarrow yvw$$

- Sequence of such replacements called *derivation*
- Derivation called *right-most* if always replace the right-most non-terminal



BNF Derivations

- Start with the start symbol:

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



BNF Derivations

- Pick a non-terminal

<Sum> =>



BNF Derivations

- Pick a rule and substitute:

- $\langle \text{Sum} \rangle ::= \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

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



BNF Derivations

- Pick a non-terminal:

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



BNF Derivations

- Pick a rule and substitute:

- $\langle \text{Sum} \rangle ::= (\langle \text{Sum} \rangle)$

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

$\Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$



BNF Derivations

- Pick a non-terminal:

$$\begin{aligned} \langle \text{Sum} \rangle & \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \\ & \Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle \end{aligned}$$



BNF Derivations

- Pick a rule and substitute:

- $\langle \text{Sum} \rangle ::= \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

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

$$\Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$$

$$\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$$



BNF Derivations

- Pick a non-terminal:

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

$\Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$

$\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$



BNF Derivations

- Pick a rule and substitute:

- $\langle \text{Sum} \rangle ::= 1$

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

$\Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$

$\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$

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



BNF Derivations

- Pick a non-terminal:

$$\begin{aligned}\langle \text{Sum} \rangle &\Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \\ &\Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle \\ &\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle \\ &\Rightarrow (\langle \text{Sum} \rangle + 1) + \langle \text{Sum} \rangle\end{aligned}$$



BNF Derivations

- Pick a rule and substitute:

- $\langle \text{Sum} \rangle ::= 0$

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

$\Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$

$\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$

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

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



BNF Derivations

- Pick a non-terminal:

$$\begin{aligned} \langle \text{Sum} \rangle & \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \\ & \Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle \\ & \Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle \\ & \Rightarrow (\langle \text{Sum} \rangle + 1) + \langle \text{Sum} \rangle \\ & \Rightarrow (\langle \text{Sum} \rangle + 1) + 0 \end{aligned}$$



BNF Derivations

- Pick a rule and substitute

- $\langle \text{Sum} \rangle ::= 0$

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

$\Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$

$\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$

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

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

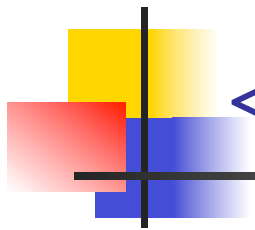
$\Rightarrow (0 + 1) + 0$



BNF Derivations

- $(0 + 1) + 0$ is generated by grammar

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



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

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



BNF Semantics

- The meaning of a BNF grammar is the set of all strings consisting only of terminals that can be derived from the Start symbol



Regular Grammars

- Subclass of BNF
- Only rules of form
 $\langle \text{nonterminal} \rangle ::= \langle \text{terminal} \rangle \langle \text{nonterminal} \rangle$ or
 $\langle \text{nonterminal} \rangle ::= \langle \text{terminal} \rangle$ or
 $\langle \text{nonterminal} \rangle ::= \varepsilon$
- Defines same class of languages as regular expressions
- Important for writing lexers (programs that convert strings of characters into strings of tokens)
- Close connection to nondeterministic finite state automata – nonterminals \cong states; rule \cong edge



Example

- Regular grammar:

$\langle \text{Balanced} \rangle ::= \varepsilon$

$\langle \text{Balanced} \rangle ::= 0 \langle \text{OneAndMore} \rangle$

$\langle \text{Balanced} \rangle ::= 1 \langle \text{ZeroAndMore} \rangle$

$\langle \text{OneAndMore} \rangle ::= 1 \langle \text{Balanced} \rangle$

$\langle \text{ZeroAndMore} \rangle ::= 0 \langle \text{Balanced} \rangle$

- Generates even length strings where every initial substring of even length has same number of 0's as 1's



Extended BNF Grammars

- Alternatives: allow rules of form $X ::= y/z$
 - Abbreviates $X ::= y, X ::= z$
- Options: $X ::= y[v]z$
 - Abbreviates $X ::= yvz, X ::= yz$
- Repetition: $X ::= y\{v\}^*z$
 - Can be eliminated by adding new nonterminal V and rules $X ::= yz, X ::= yVz, V ::= v, V ::= w$



Parse Trees

- Graphical representation of derivation
- Each node labeled with either non-terminal or terminal
- If node is labeled with a terminal, then it is a leaf (no sub-trees)
- If node is labeled with a non-terminal, then it has one branch for each character in the right-hand side of rule used to substitute for it



Example

- Consider grammar:

$$\langle \text{exp} \rangle ::= \langle \text{factor} \rangle$$
$$\quad \quad \quad | \langle \text{factor} \rangle + \langle \text{factor} \rangle$$
$$\langle \text{factor} \rangle ::= \langle \text{bin} \rangle$$
$$\quad \quad \quad | \langle \text{bin} \rangle * \langle \text{exp} \rangle$$
$$\langle \text{bin} \rangle ::= 0 \quad | \quad 1$$

- Problem: Build parse tree for $1 * 1 + 0$ as an $\langle \text{exp} \rangle$



Example cont.

- $1 * 1 + 0$: $\langle \text{exp} \rangle$

$\langle \text{exp} \rangle$ is the start symbol for this parse tree



Example cont.

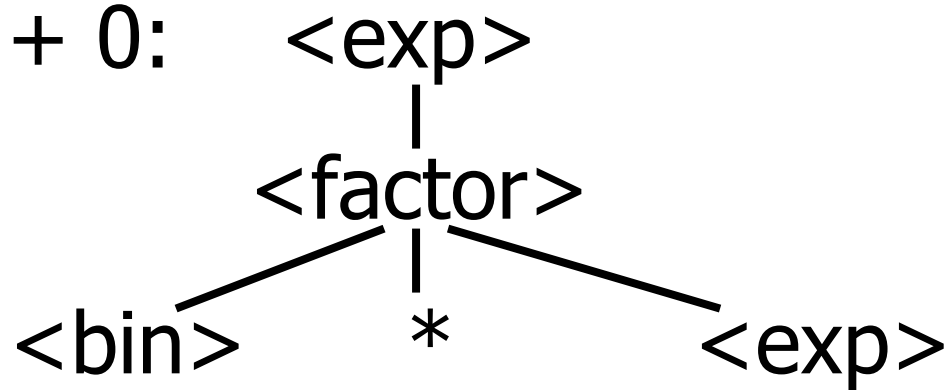
■ $1 * 1 + 0$: $\langle \text{exp} \rangle$
|
 $\langle \text{factor} \rangle$

Use rule: $\langle \text{exp} \rangle ::= \langle \text{factor} \rangle$



Example cont.

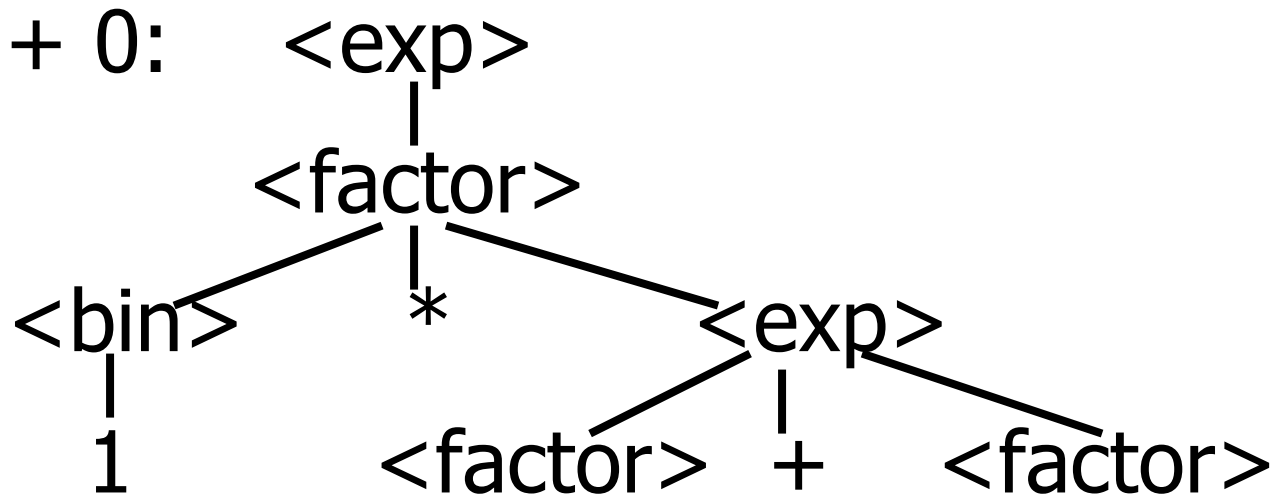
- 1 * 1 + 0:



Use rule: $\langle \text{factor} \rangle ::= \langle \text{bin} \rangle * \langle \text{exp} \rangle$

Example cont.

- $1 * 1 + 0$:

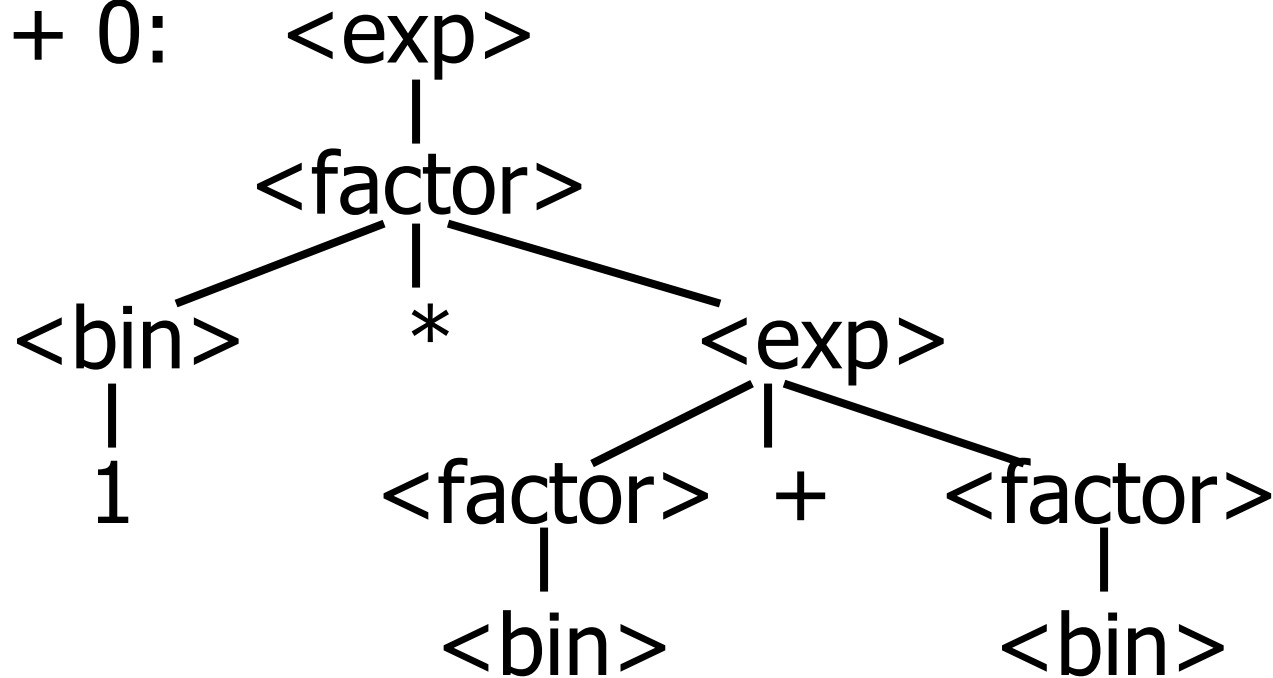


Use rules: $\langle \text{bin} \rangle ::= 1$ and
 $\langle \text{exp} \rangle ::= \langle \text{factor} \rangle + \langle \text{factor} \rangle$



Example cont.

- $1 * 1 + 0$:

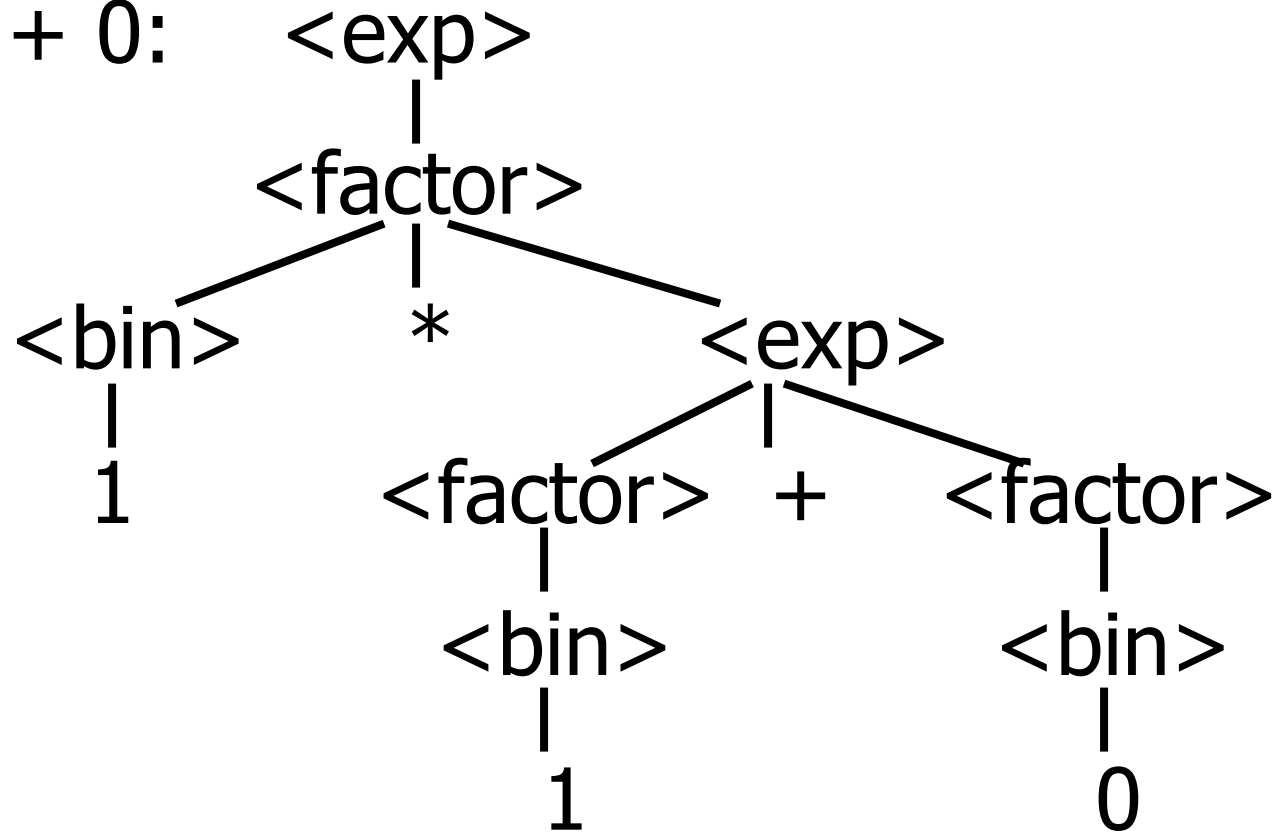


Use rule: $\langle \text{factor} \rangle ::= \langle \text{bin} \rangle$



Example cont.

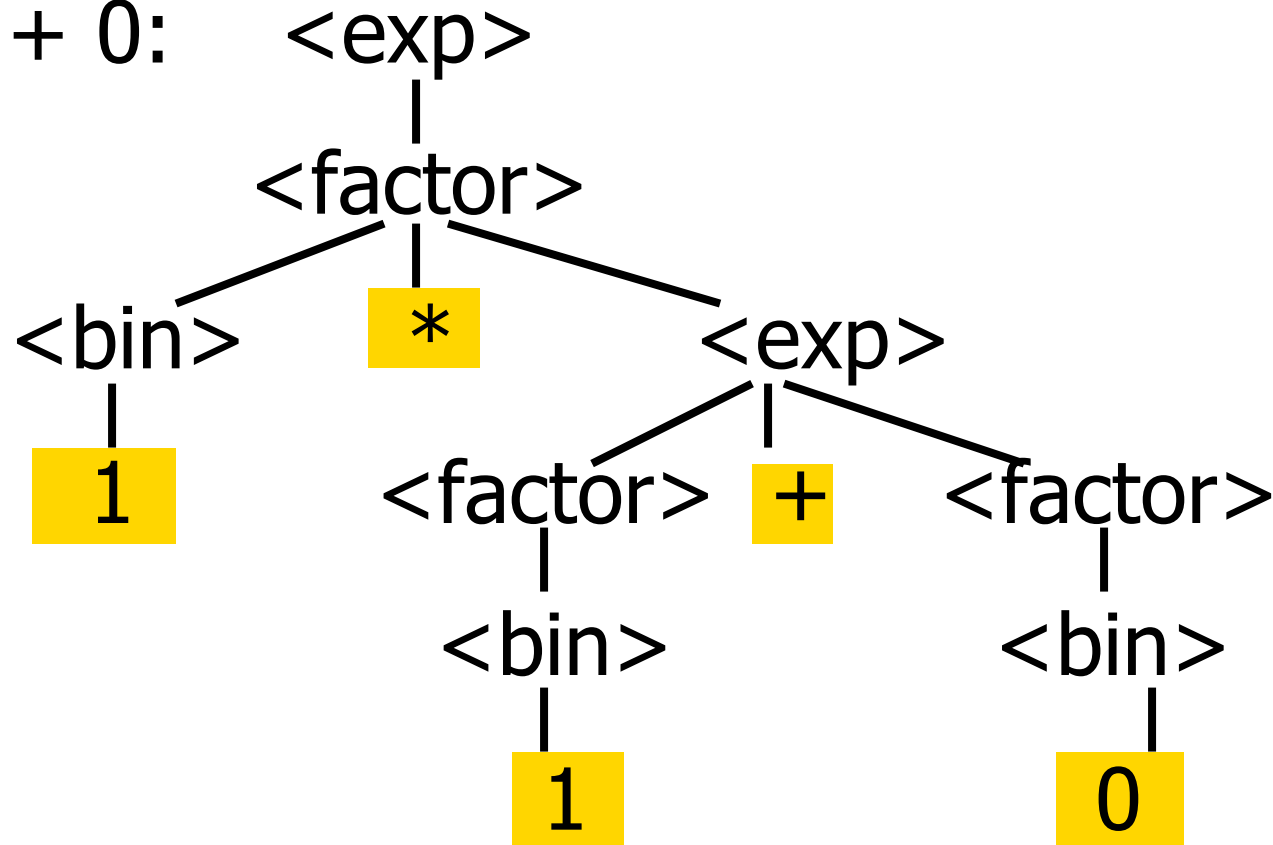
- $1 * 1 + 0$:



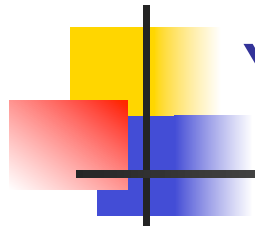
Use rules: $\langle \text{bin} \rangle ::= 1 \mid 0$

Example cont.

- 1 * 1 + 0:



Fringe of tree is string generated by grammar



Your Turn: $1 * 0 + 0 * 1$



Parse Tree Data Structures

- Parse trees may be represented by OCaml datatypes
- One datatype for each nonterminal
- One constructor for each rule
- Defined as mutually recursive collection of datatype declarations



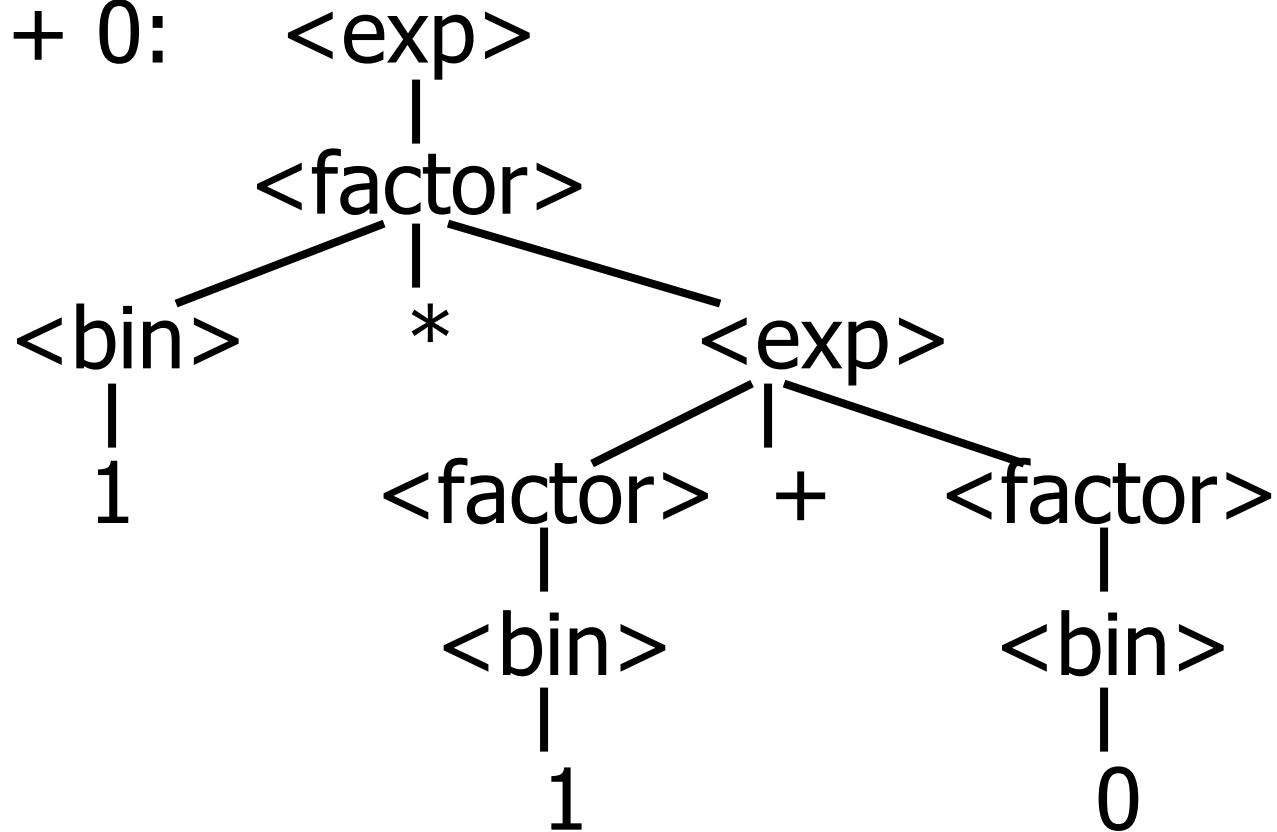
Example

- Recall grammar:
 $\langle \text{exp} \rangle ::= \langle \text{factor} \rangle \mid \langle \text{factor} \rangle + \langle \text{factor} \rangle$
 $\langle \text{factor} \rangle ::= \langle \text{bin} \rangle \mid \langle \text{bin} \rangle * \langle \text{exp} \rangle$
 $\langle \text{bin} \rangle ::= 0 \mid 1$
- type exp = Factor2Exp of factor
 | Plus of factor * factor
and factor = Bin2Factor of bin
 | Mult of bin * exp
and bin = Zero | One



Example cont.

- $1 * 1 + 0$:





Example cont.

- Can be represented as

Factor2Exp

(Mult(One,

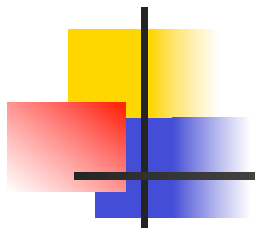
Plus(Bin2Factor One,

Bin2Factor Zero)))



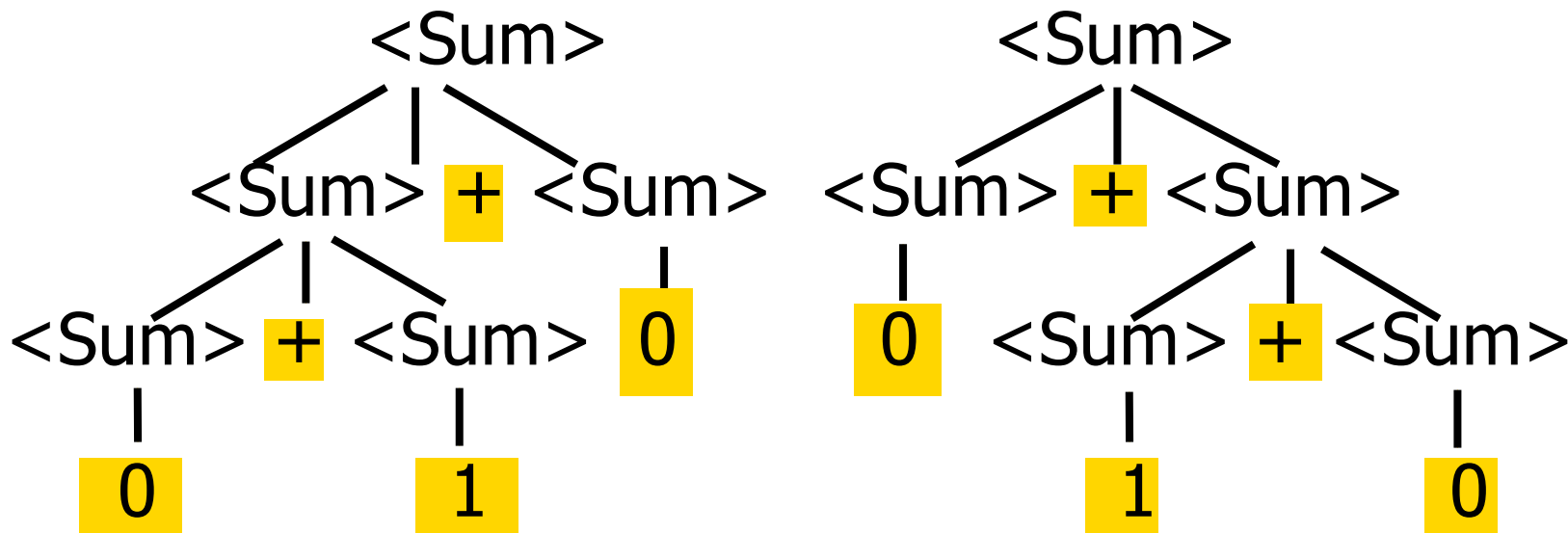
Ambiguous Grammars and Languages

- A BNF grammar is *ambiguous* if its language contains strings for which there is more than one parse tree
- If all BNF's for a language are ambiguous then the language is *inherently ambiguous*



Example: Ambiguous Grammar

■ $0 + 1 + 0$





Example

- What is the result for:

$$3 + 4 * 5 + 6$$



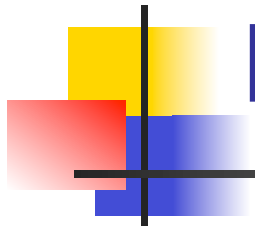
Example

- What is the result for:

$$3 + 4 * 5 + 6$$

- Possible answers:

- $41 = ((3 + 4) * 5) + 6$
- $47 = 3 + (4 * (5 + 6))$
- $29 = (3 + (4 * 5)) + 6 = 3 + ((4 * 5) + 6)$
- $77 = (3 + 4) * (5 + 6)$



Example

- What is the value of:

$$7 - 5 - 2$$



Example

- What is the value of:

$$7 - 5 - 2$$

- Possible answers:
 - In Pascal, C++, SML assoc. left
 $7 - 5 - 2 = (7 - 5) - 2 = 0$
 - In APL, associate to right
 $7 - 5 - 2 = 7 - (5 - 2) = 4$



Two Major Sources of Ambiguity

- Lack of determination of operator precedence
- Lack of determination of operator associativity
- Not the only sources of ambiguity