Example Regular Expressions

- \((0|1)^*1\)
  - The set of all strings of 0's and 1's ending in 1, \{1, 01, 11, \ldots\}
- \(a^*b(a^*)\)
  - The set of all strings of a's and b's with exactly one b
- \(((01) \lor (10))^*\)
  - You tell me

Regular expressions (equivalently, regular grammars) important for lexing, breaking strings into recognized words

Example

Regular grammar:
- \(<Balanced> ::= \varepsilon\>
- \(<Balanced> ::= 0<OneAndMore>\>
- \(<Balanced> ::= 1<ZeroAndMore>\>
- \(<OneAndMore> ::= 1<Balanced>\>
- \(<ZeroAndMore> ::= 0<Balanced>\>

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 \lor b \lor \ldots \lor z \lor A \lor B \lor \ldots \lor Z) \,(a \lor b \lor \ldots \lor z \lor A \lor B \lor \ldots \lor Z \lor 0 \lor 1 \lor \ldots \lor 9)^*
  - Digit = (0 \lor 1 \lor \ldots \lor 9)
  - Number = 0 \lor (1 \lor \ldots \lor 9)(0 \lor \ldots \lor 9)^* \lor \sim (1 \lor \ldots \lor 9)(0 \lor \ldots \lor 9)^*
  - Keywords: if = if, while = while, \ldots

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.

Mechanics

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

```ocaml
{ header }
let ident = regexp ...
rule entrypoint [arg1... argn] = parse
  regexp { action }
| ...
| regexp { action }
and entrypoint [arg1... argn] = parse ...
{ trailer }
```

Ocamllex 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`

Ocamllex Regular Expression

- `[^c1 - c2]`: choice of any character NOT in set
- `e*`: same as before
- `e+`: same as `e e*`
- `e?:` option - was `e1 v ε`

Ocamllex Regular Expression

- `[c1 - c2]`: choice of any character between first and second inclusive, as determined by character codes
- `e1 # e2`: the characters in `e1` but not in `e2`; `e1` and `e2` must describe just sets of characters
- `ident`: abbreviation for earlier regexp in `let ident = regexp`
- `e1 as id`: binds the result of `e1` to `id` to be used in the associated `action`
More details can be found at
http://caml.inria.fr/pub/docs/manual-ocaml/
lexyacc.html

Example : test.mll

```ocaml
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 +

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

```ocaml
# #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

```ocaml
# 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
- Mainly useful when you can make your lexer be your parser
- OCaml yacc parser needs tokens one at a time

Example

```ocaml
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}
| eof                     { [] }
| _                        { main lexbuf }
```

Dealing with nested comments

```
| open_comment         { comment lexbuf}
| eof                   { [] }
| _ { main lexbuf }
and comment = parse
   close_comment      { main lexbuf }
| _                     { comment lexbuf }
```

```
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
- <Sum> ::= 0
- <Sum> ::= 1
- <Sum> ::= <Sum> + <Sum>
- <Sum> ::= (<Sum>)

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
  X ::= y
  where 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: <Sum>
- Start symbol = <Sum>
- <Sum> ::= 0
- <Sum> ::= 1
- <Sum> ::= <Sum> + <Sum>
- <Sum> ::= (<Sum>)
- Can be abbreviated as
  <Sum> ::= 0 | 1
  | <Sum> + <Sum> | (<Sum>)

BNF Derivations

- Given rules
  \( X ::= yZw \) and \( Z ::= v \)
  we may replace \( Z \) by \( v \) to say
  \( X \Rightarrow yZw \Rightarrow yvw \)
- 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:
  \( <Sum> \Rightarrow \)

BNF Derivations

- Pick a non-terminal
  \( <Sum> \Rightarrow \)

BNF Derivations

- Pick a rule and substitute:
  - \( <Sum> ::= <Sum> + <Sum> \)
  \( <Sum> \Rightarrow <Sum> + <Sum> \)
BNF Derivations

- Pick a rule and substitute:
  - \(<\text{Sum}> ::= ( \langle\text{Sum}\rangle )\)
  - \(<\text{Sum}> \Rightarrow <\text{Sum}> + <\text{Sum}>\)
    \[\Rightarrow ( \langle\text{Sum}\rangle ) + <\text{Sum}>\]

BNF Derivations

- Pick a non-terminal:
  - \(<\text{Sum}> \Rightarrow <\text{Sum}> + <\text{Sum}>\)
    \[\Rightarrow ( \langle\text{Sum}\rangle ) + <\text{Sum}>\]

BNF Derivations

- Pick a rule and substitute:
  - \(<\text{Sum}> ::= <\text{Sum}> + <\text{Sum}>\)
  - \(<\text{Sum}> \Rightarrow <\text{Sum}> + <\text{Sum}>\)
    \[\Rightarrow ( \langle\text{Sum}\rangle ) + <\text{Sum}>\]
    \[\Rightarrow ( \langle\text{Sum} +\rangle + <\text{Sum}\rangle ) + <\text{Sum}>\]

BNF Derivations

- Pick a non-terminal:
  - \(<\text{Sum}> \Rightarrow <\text{Sum}> + <\text{Sum}>\)
    \[\Rightarrow ( \langle\text{Sum}\rangle ) + <\text{Sum}>\]
    \[\Rightarrow ( \langle\text{Sum} +<\text{Sum}\rangle ) + <\text{Sum}>\]

BNF Derivations

- Pick a rule and substitute:
  - \(<\text{Sum}> ::= 1\)
  - \(<\text{Sum}> \Rightarrow <\text{Sum}> + <\text{Sum}>\)
    \[\Rightarrow ( \langle\text{Sum}\rangle ) + <\text{Sum}>\]
    \[\Rightarrow ( \langle\text{Sum} +1\rangle + <\text{Sum}\rangle ) + <\text{Sum}>\]
    \[\Rightarrow ( \langle\text{Sum} +1\rangle + <\text{Sum}\rangle ) + <\text{Sum}>\]
BNF Derivations

- Pick a rule and substitute:
  - \(<\text{Sum}> ::= 0\)
  - \(<\text{Sum}> ::= <\text{Sum}> + <\text{Sum}>\)
    - \(\Rightarrow ( <\text{Sum}> ) + <\text{Sum}>\)
    - \(\Rightarrow ( <\text{Sum}> + <\text{Sum}> ) + <\text{Sum}>\)
    - \(\Rightarrow ( <\text{Sum}> + 1 ) + <\text{Sum}>\)
    - \(\Rightarrow ( <\text{Sum}> + 1 ) + 0\)

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.

BNF Derivations

- Pick a non-terminal:
  - \(<\text{Sum}> ::= <\text{Sum}> + <\text{Sum}>\)
    - \(\Rightarrow ( <\text{Sum}> ) + <\text{Sum}>\)
    - \(\Rightarrow ( <\text{Sum}> + <\text{Sum}> ) + <\text{Sum}>\)
    - \(\Rightarrow ( <\text{Sum}> + 1 ) + <\text{Sum}>\)
    - \(\Rightarrow ( <\text{Sum}> + 1 ) + 0\)

BNF Derivations

- (0 + 1) + 0 is generated by grammar
  - \(<\text{Sum}> ::= 0 | 1 | <\text{Sum}> + <\text{Sum}> | ( <\text{Sum}> )\)
  - \(<\text{Sum}> ::= 0 | 1 | <\text{Sum}> + <\text{Sum}> | ( <\text{Sum}> )\)

BNF Derivations

- 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
  \(<\text{nonterminal}>::=<\text{terminal}><\text{nonterminal}>\)
  or
  \(<\text{nonterminal}>::=<\text{terminal}>\)
- 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 \(\equiv\) states; rule \(\equiv\) edge

Extended BNF Grammars

- Alternatives: allow rules of from \(X::=y|z\)
  - 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 \(X::=yz, X::=yVz, V::=v, V::=vV\)

Example

- Regular grammar:
  \(<\text{Balanced}>::=\varepsilon\)
  \(<\text{Balanced}>::=0<\text{OneAndMore}>\)
  \(<\text{Balanced}>::=1<\text{ZeroAndMore}>\)
  \(<\text{OneAndMore}>::=1<\text{Balanced}>\)
  \(<\text{ZeroAndMore}>::=0<\text{Balanced}>\)
- Generates even length strings where every initial substring of even length has same number of 0’s as 1’s

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:
  \(<\text{exp}>::=\text{<factor>}\)
    \| \text{<factor>} + \text{<factor>}\)
  \(<\text{factor}>::=\text{<bin>}\)
    \| \text{<bin>} * \text{<exp>}\)
  \(<\text{bin}>::=0\ | \ 1\)
- Problem: Build parse tree for \(1 * 1 + 0\) as an \(<\text{exp}>\)

Example cont.

- \(1 * 1 + 0: \ <\text{exp}>\)
  \(<\text{exp}>\) is the start symbol for this parse tree
Example cont.

\[ 1 \times 1 + 0: \quad \text{<exp>}
\]

\[ \quad \text{<factor>}
\]

Use rule: \( \text{<exp>} ::= \text{<factor>} \)

Example cont.

\[ 1 \times 1 + 0: \quad \text{<exp>}
\]

\[ \quad \text{<factor>}
\]

\[ \quad \text{<bin>} \quad \ast \quad \text{<exp>}
\]

\[ 1 \quad \text{<factor>} \quad + \quad \text{<factor>}
\]

Use rule: \( \text{<factor>} ::= \text{<bin>} \ast \text{<exp>} \)

Example cont.

\[ 1 \times 1 + 0: \quad \text{<exp>}
\]

\[ \quad \text{<factor>}
\]

\[ \quad \text{<bin>} \quad \ast \quad \text{<exp>}
\]

\[ 1 \quad \text{<factor>} \quad + \quad \text{<factor>}
\]

\[ 1 \quad \text{<factor>} \quad + \quad \text{<factor>}
\]

Use rules: \( \text{<bin>} ::= 1 \) and \( \text{<exp>} ::= \text{<factor>} + \text{<factor>} \)

Example cont.

\[ 1 \times 1 + 0: \quad \text{<exp>}
\]

\[ \quad \text{<factor>}
\]

\[ \quad \text{<bin>} \quad \ast \quad \text{<exp>}
\]

\[ 1 \quad \text{<factor>} \quad + \quad \text{<factor>}
\]

\[ 1 \quad \text{<factor>} \quad + \quad \text{<factor>}
\]

Use rule: \( \text{<factor>} ::= \text{<bin>} \)

Example cont.

\[ 1 \times 1 + 0: \quad \text{<exp>}
\]

\[ \quad \text{<factor>}
\]

\[ \quad \text{<bin>} \quad \ast \quad \text{<exp>}
\]

\[ 1 \quad \text{<factor>} \quad + \quad \text{<factor>}
\]

\[ 1 \quad \text{<factor>} \quad + \quad \text{<factor>}
\]

Use rules: \( \text{<bin>} ::= 1 \mid 0 \)

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:
  \[ \langle \text{exp} \rangle \ ::= \langle \text{factor} \rangle \mid \langle \text{factor} \rangle + \langle \text{factor} \rangle \]
  \[ \langle \text{factor} \rangle \ ::= \langle \text{bin} \rangle \mid \langle \text{bin} \rangle \ast \langle \text{exp} \rangle \]
  \[ \langle \text{bin} \rangle \ ::= 0 \mid 1 \]
- type \( \text{exp} = \text{Factor2Exp of factor} \)
  | Plus of factor \ast factor
  and \( \text{factor} = \text{Bin2Factor of bin} \)
  | Mult of bin \ast exp
  and \( \text{bin} = \text{Zero} \mid \text{One} \)

Example cont.
- 1 * 1 + 0:
  \[
  \begin{align*}
  \langle \text{exp} \rangle & \quad \langle \text{factor} \rangle \\
  \langle \text{bin} \rangle & \quad \ast \\n  \langle \text{factor} \rangle & \quad + \\n  \langle \text{bin} \rangle & \quad \\
  \langle \text{bin} \rangle & \quad \\
  \langle \text{bin} \rangle & \quad
  \end{align*}
  \]
- Can be represented as
  \[
  \text{Factor2Exp} \left( \text{Mult}(\text{One}, \text{Plus}(\text{Bin2Factor One}, \text{Bin2Factor Zero})) \right)
  \]

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

```
<Sum>  
  <Sum> <Sum> <Sum>
  <Sum> <Sum> 0 <Sum> <Sum>
  0     1        0
```

Example

- What is the result for:
  
  \[ 3 + 4 \times 5 + 6 \]

Possible answers:

- \[ 41 = ((3 + 4) \times 5) + 6 \]
- \[ 47 = 3 + (4 \times (5 + 6)) \]
- \[ 29 = (3 + (4 \times 5)) + 6 = 3 + ((4 \times 5) + 6) \]
- \[ 77 = (3 + 4) \times (5 + 6) \]

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