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

- $(0 \lor 1)^*1$
  - The set of all strings of 0’s and 1’s ending in 1, \{1, 01, 11,...\}

- $a^*b(a^*)$
  - 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
Regular Grammars

- Subclass of BNF (covered in detail soon)
- Only rules of form
  \[ <\text{nonterminal}> ::= <\text{terminal}> <\text{nonterminal}> \] or
  \[ <\text{nonterminal}> ::= <\text{terminal}> \] or
  \[ <\text{nonterminal}> ::= \epsilon \]
- 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
Example

- Regular grammar:
  \[
  \begin{align*}
  \langle \text{Balanced} \rangle &::= \varepsilon \\
  \langle \text{Balanced} \rangle &::= 0 \langle \text{OneAndMore} \rangle \\
  \langle \text{Balanced} \rangle &::= 1 \langle \text{ZeroAndMore} \rangle \\
  \langle \text{OneAndMore} \rangle &::= 1 \langle \text{Balanced} \rangle \\
  \langle \text{ZeroAndMore} \rangle &::= 0 \langle \text{Balanced} \rangle 
  \end{align*}
  \]

- 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 ∨ b ∨ ... ∨ z ∨ A ∨ B ∨ ... ∨ Z) (a ∨ b ∨ ... ∨ z ∨ A ∨ B ∨ ... ∨ Z ∨ 0 ∨ 1 ∨ ... ∨ 9)*
  - Digit = (0 ∨ 1 ∨ ... ∨ 9)
  - Number = 0 ∨ (1 ∨ ... ∨ 9)(0 ∨ ... ∨ 9)* ∨ ~ (1 ∨ ... ∨ 9)(0 ∨ ... ∨ 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>.mll`
- Call `ocamllex `<filename>.mll` produces Ocaml code for a lexical analyzer in file `<filename>.ml`
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_channels stdin) in
  print_string "Ready to lex.\n";
  main newlexbuf
}
General Input

```latex
\{ \textit{header} \} \\
\text{let } \textit{ident} = \textit{regexp} \ldots \\
\text{rule } \textit{entrypoint} [\textit{arg1} \ldots \textit{argn}] = \text{parse} \\
\hspace{1cm} \textit{regexp} \{ \textit{action} \} \\
\hspace{2cm} | \ldots \\
\hspace{3cm} | \textit{regexp} \{ \textit{action} \} \\
\text{and } \textit{entrypoint} [\textit{arg1} \ldots \textit{argn}] = \text{parse} \ldots \text{and} \ldots \\
\{ \textit{trailer} \} 
```
Ocamllex Input

- *header* and *trailer* contain arbitrary ocaml code put at top an bottom of `<filename>.ml`

- let *ident* = *regexp* ... Introduces *ident* for use in later regular expressions
Ocamlllex 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

- 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
- $e_1 \lor e_2$: choice - what was $e_1 \lor e_2$
Ocamllex Regular Expression

- \([c_1 - c_2]\): choice of any character between first and second inclusive, as determined by character codes
- \([\sim c_1 - c_2]\): choice of any character NOT in set
- \(e^*\): same as before
- \(e^+\): same as \(e\ e^*\)
- \(e?\): option - was \(e_1 \lor \varepsilon\)
Ocamllex 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
More details can be found at

Example : test.mll

```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.mll

```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

```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
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                    { [] }
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

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

- Start with a set of characters, \( a, b, c, \ldots \)
  - We call these *terminals*
- Add a set of different characters, \( X, Y, Z, \ldots \)
  - 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
  \[ X ::= yZw \text{ and } Z ::= \nu \]
  we may replace \( Z \) by \( \nu \) to say
  \[ X \Rightarrow yZw \Rightarrow y\nu w \]

- Sequence of such replacements called \emph{derivation}

- Derivation called \emph{right-most} if always replace the right-most non-terminal
BNF Derivations

- Start with the start symbol:

\[ <\text{Sum}> \Rightarrow \]
BNF Derivations

- Pick a non-terminal

<Sum> =>
BNF Derivations

- Pick a rule and substitute:
  - `<Sum> ::= <Sum> + <Sum>`

  `<Sum>` => `<Sum> + <Sum>`
BNF Derivations

- Pick a non-terminal:

\[
<\text{Sum}> \Rightarrow <\text{Sum}> + <\text{Sum}>
\]
BNF Derivations

- Pick a rule and substitute:
  - `<Sum> ::= ( <Sum> )`

```
<Sum>  =>  <Sum> + <Sum>
 =>  ( <Sum> ) + <Sum>
```
BNF Derivations

- Pick a non-terminal:

\[ <\text{Sum}> \Rightarrow <\text{Sum}> + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> ) + <\text{Sum}> \]
BNF Derivations

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

- Pick a non-terminal:

\[<\text{Sum}> \Rightarrow <\text{Sum}> + <\text{Sum}>\]
\[\Rightarrow ( <\text{Sum}> ) + <\text{Sum}>\]
\[\Rightarrow ( <\text{Sum}> + <\text{Sum}> ) + <\text{Sum}>\]
BNF Derivations

- Pick a rule and substitute:
  - `<Sum>` ::= 1

`<Sum>` => `<Sum>` + `<Sum>`
  => ( `<Sum>` ) + `<Sum>`
  => ( `<Sum>` + `<Sum>` ) + `<Sum>`
  => ( `<Sum>` + 1 ) + `<Sum>`
BNF Derivations

- Pick a non-terminal:

\[
\text{<Sum>} \Rightarrow \text{<Sum> + <Sum>}
\]

\[
\Rightarrow (\text{<Sum>}) + \text{<Sum>}
\]

\[
\Rightarrow (\text{<Sum> + <Sum>}) + \text{<Sum>}
\]

\[
\Rightarrow (\text{<Sum> + 1}) + \text{<Sum>}
\]
BNF Derivations

- Pick a rule and substitute:
  - `<Sum>` ::= 0

  `<Sum>` => `<Sum>` + `<Sum>`
  => ( `<Sum>` ) + `<Sum>`
  => ( `<Sum>` + `<Sum>` ) + `<Sum>`
  => ( `<Sum>` + 1 ) + `<Sum>`
  => ( `<Sum>` + 1 ) + 0
BNF Derivations

- Pick a non-terminal:

\[
\text{<Sum>} \Rightarrow \text{<Sum>} + \text{<Sum>}
\]

\[\Rightarrow (\text{<Sum>}) + \text{<Sum>}\]

\[\Rightarrow (\text{<Sum>} + \text{<Sum>}) + \text{<Sum>}\]

\[\Rightarrow (\text{<Sum>} + 1) + \text{<Sum>}\]

\[\Rightarrow (\text{<Sum>} + 1) + 0\]
BNF Derivations

- Pick a rule and substitute

  - `<Sum>` ::= 0

  `<Sum>`  =>  `<Sum>` + `<Sum>`
  => ( `<Sum>` ) + `<Sum>`
  => ( `<Sum>` + `<Sum>` ) + `<Sum>`
  => ( `<Sum>` + 1 ) + `<Sum>`
  => ( `<Sum>` + 1 ) 0
  => ( 0 + 1 ) + 0
BNF Derivations

(0 + 1) + 0 is generated by grammar

<Sum> => <Sum> + <Sum>
=> ( <Sum> ) + <Sum>
=> ( <Sum> + <Sum> ) + <Sum>
=> ( <Sum> + 1 ) + <Sum>
=> ( <Sum> + 1 ) + 0
=> ( 0 + 1 ) + 0
\(<\text{Sum}\> ::= 0 \mid 1 \mid <\text{Sum}\> + <\text{Sum}\> \mid (<\text{Sum}\>)\)

\(<\text{Sum}\> = >
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
  \[
  \text{<nonterminal>} ::= \text{<terminal>} \text{<nonterminal>} \text{ or } \\
  \text{<nonterminal>} ::= \text{<terminal>} \text{ or } \\
  \text{<nonterminal>} ::= \epsilon
  \]
- 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 \(\approx\) states; rule \(\approx\) 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 from $X ::= y/z$
  - Abbreviates $X ::= y, X ::= z$
- Options: $X ::= y[\nu]z$
  - Abbreviates $X ::= y\nu z, X ::= yz$
- Repetition: $X ::= y\{\nu\}^*z$
  - Can be eliminated by adding new nonterminal $V$ and rules $X ::= yz, X ::= yVz, V ::= \nu, V ::= \nu W$
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
Consider grammar:

\[ \texttt{<exp>} ::= \texttt{<factor>}, \quad \texttt{<factor>} ::= \texttt{<bin>}, \quad \texttt{<bin>} ::= 0 \mid 1 \]

\[ \texttt{<exp>} ::= \texttt{<factor>} \mid \texttt{<factor>} + \texttt{<factor>} \mid \texttt{<bin>} \ast \texttt{<exp>} \]

Problem: Build parse tree for \(1 \ast 1 + 0\) as an \texttt{<exp>}. 
Example cont.

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

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

- $1 \times 1 + 0$: $<\text{exp}>$
  
  $<\text{factor}>$

Use rule: $<\text{exp}> ::= <\text{factor}>$
Example cont.

1 * 1 + 0:  \( \text{<exp>} \)

Use rule: \( \text{<factor>} ::= \text{<bin>} \times \text{<exp>} \)
Example cont.

1 * 1 + 0: \( <\text{exp}> \)

\[
\begin{array}{c}
\text{<bin> } \ast \text{<exp>} \\
1 \quad \text{<factor>} \quad + \quad \text{<factor>}
\end{array}
\]

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

- $1 \times 1 + 0$: 

  \[
  \begin{array}{c}
  \text{<exp>}
  \\
  \text{<factor>}
  \\
  \text{<bin> * <exp>}
  \\
  1 <factor> + <factor>
  \\
  \text{<bin> + <bin>}
  \end{array}
  \]

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

10/22/15
Example cont.

1 * 1 + 0:

- `<exp>`
  - `<factor>`
    - `<bin>` * `<exp>`
      - 1 * 1 + 0
        - `<factor>` + `<factor>`
          - `<bin>` 1 + `<bin>` 0

Use rules: `<bin> ::= 1 | 0`
Example cont.

1 * 1 + 0:  

\[
\begin{array}{c}
\text{<exp>}
\end{array}
\]

\[
\begin{array}{c}
\text{<factor>}
\end{array}
\]

\[
\begin{array}{c}
\text{<bin>}
\end{array}
\text{ * } \begin{array}{c}
\text{<exp>}
\end{array}
\]

\[
\begin{array}{c}
\text{1}
\end{array}
\]

\[
\begin{array}{c}
\text{<factor>}
\end{array}
\]

\[
\begin{array}{c}
\text{+}
\end{array}
\]

\[
\begin{array}{c}
\text{<factor>}
\end{array}
\]

\[
\begin{array}{c}
\text{<bin>}
\end{array}
\text{ <bin>}
\]

\[
\begin{array}{c}
\text{1}
\end{array}
\]

\[
\begin{array}{c}
\text{0}
\end{array}
\]

Fringe of tree is string generated by grammar
Your Turn: $1 \times 0 + 0 \times 1$
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:
  \[
  \begin{align*}
  \langle \text{exp} \rangle & ::= \langle \text{factor} \rangle \mid \langle \text{factor} \rangle + \langle \text{factor} \rangle \\
  \langle \text{factor} \rangle & ::= \langle \text{bin} \rangle \mid \langle \text{bin} \rangle \ast \langle \text{exp} \rangle \\
  \langle \text{bin} \rangle & ::= 0 \mid 1
  \end{align*}
  \]

- type \( \text{exp} = \text{Factor2Exp of factor} \mid \text{Plus of factor} \ast \text{factor} \)
  and \( \text{factor} = \text{Bin2Factor of bin} \mid \text{Mult of bin} \ast \text{exp} \)
  and \( \text{bin} = \text{Zero} \mid \text{One} \)
Example cont.

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

\[
\text{<factor>}
\]

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

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

\[
\text{<bin>} \quad 1 \quad \text{<bin>} \quad 0
\]
Example cont.

- Can be represented as

\[
\text{Factor2Exp} \\
\text{(Mult(One,} \\
\quad \text{Plus(Bin2Factor One,} \\
\quad \quad \text{Bin2Factor Zero))})
\]
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:

$$3 + 4 \times 5 + 6$$
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:

$$7 - 5 - 2$$
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