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

General Input

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

Ocamllex 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

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 / e_2`: choice - what was $e_1 \lor e_2$
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 $ident = \text{regexp}$
- $e_1$ as $id$: binds the result of $e_1$ to $id$ to be used in the associated action

Example : test.mll

```ocaml
{ 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

```ocaml
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.
h i 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 MP8
  - 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 }
| eof { [] }
| _ { main lexbuf }

and comment = parse
  close_comment { main lexbuf }
| _ { comment 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 }

10/23/14
19

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 }

10/23/14
20

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

10/23/14
21

SAMPLE GRAMMAR

- Language: Parenthesized sums of 0’s and 1’s
  - <Sum> ::= 0
  - <Sum> ::= 1
  - <Sum> ::= <Sum> + <Sum>
  - <Sum> ::= (<Sum>)

10/23/14
22

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

10/23/14
23
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
  \[ X ::= yZw \text{ and } Z ::= v \]
  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:
  \[ <Sum> \Rightarrow \]

- Pick a non-terminal
  \[ <Sum> \Rightarrow \]

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

Pick a non-terminal:

<Sum> => <Sum> + <Sum >

Pick a rule and substitute:

<Sum> ::= ( <Sum> )
<Sum> => <Sum> + <Sum >
=> ( <Sum> ) + <Sum>

Pick a non-terminal:

<Sum> => <Sum> + <Sum >
=> ( <Sum> ) + <Sum>

Pick a rule and substitute:

<Sum> ::= <Sum> + <Sum>
<Sum> => <Sum> + <Sum >
=> ( <Sum> + <Sum> ) + <Sum>

Pick a rule and substitute:

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

**Pick a non-terminal:**

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

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

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

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

BNF Derivations

**Pick a rule and substitute:**

- \(<\text{Sum}>::= 0\)

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

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

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

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

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

BNF Derivations

**Pick a non-terminal:**

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

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

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

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

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

BNF Derivations

**Pick a rule and substitute:**

- \(<\text{Sum}>::= 0\)

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

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

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

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

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

BNF Derivations

**(0 + 1) + 0 is generated by grammar**

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

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

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

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

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

BNF Derivations

**<Sum>::= 0 | 1 | <Sum> + <Sum> | (Sum)\)**

\[
<\text{Sum}> \Rightarrow
\]
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 $\text{nonterminal} ::= \text{terminal} \cdot \text{nonterminal}$ 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:
  \[
  \begin{align*}
  \text{<Balanced>} & ::= \epsilon \\
  \text{<Balanced>} & ::= 0 \cdot \text{<OneAndMore>} \\
  \text{<Balanced>} & ::= 1 \cdot \text{<ZeroAndMore>} \\
  \text{<OneAndMore>} & ::= 1 \cdot \text{<Balanced>} \\
  \text{<ZeroAndMore>} & ::= 0 \cdot \text{<Balanced>}
  \end{align*}
  \]
- 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 $X ::= y \mid z$
  - Abbreviates $X ::= y, X ::= z$
- Options: $X ::= y \cdot \{v\} \ast z$
  - Abbreviates $X ::= yvz, X ::= yz$
- Repetition: $X ::= y \cdot \{v\} \ast z$
  - Can be eliminated by adding new nonterminal $V$ and rules $X ::= yz, X ::= yVz, V ::= v, V ::= vV$

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:
  \[
  \begin{align*}
  \text{<exp>} & ::= \text{<factor>} \\
  & \mid \text{<factor>} + \text{<factor>} \\
  \text{<factor>} & ::= \text{<bin>} \\
  & \mid \text{<bin>} \ast \text{<exp>} \\
  \text{<bin>} & ::= 0 \mid 1
  \end{align*}
  \]
- Problem: Build parse tree for $1 \ast 1 + 0$ as an $\text{<exp>}$
Example cont.

- $1 \times 1 + 0$: $<$exp$>$

$<$exp$>$ is the start symbol for this parse tree

Example cont.

- $1 \times 1 + 0$: $<$exp$>$

Use rule: $<$exp$> ::= <$factor$>$

Example cont.

- $1 \times 1 + 0$: $<$exp$>$

Use rule: $<$factor$> ::= <bin> * $<$exp$>$

Example cont.

- $1 \times 1 + 0$: $<$exp$>$

Use rules: $<$bin$> ::= 1$ and $<$exp$> ::= <$factor$> + <$factor$>

Example cont.

- $1 \times 1 + 0$: $<$exp$>$

Use rules: $<$bin$> ::= 1 | 0
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:
  
  ```
  <exp> ::= <factor> | <factor> + <factor>
  <factor> ::= <bin> | <bin> * <exp>
  <bin> ::= 0 | 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.
- Can be represented as

  ```
  Factor2Exp
  (Mult(One,
       Plus(Bin2Factor One,
            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**

\[
\begin{align*}
0 + 1 + 0 & \\
<\text{Sum}> & \rightarrow <\text{Sum}> + <\text{Sum}> \\
& \rightarrow 0 + <\text{Sum}> \\
& \rightarrow <\text{Sum}> + 0 \\
& \rightarrow 0 + 1 + 0
\end{align*}
\]

**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