Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC

https://courses.engr.illinois.edu/cs421/sp2023

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha
EOH!

- Engineering Open House is Friday and Saturday
- They are taking this room

- On Friday only, we will meet in Zoom only:
  - Zoom Info: Meeting Id: 838 6324 1301
  - Passcode: cs421
  - URL: [https://illinois.zoom.us/j/83863241301?pwd=U2dtRm9RUmhVQUw4d3dFOVJxNHY4UT09](https://illinois.zoom.us/j/83863241301?pwd=U2dtRm9RUmhVQUw4d3dFOVJxNHY4UT09)
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

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_newline ();
  main newlexbuf  }
Example

```ocaml
# use "test.ml";;
...
val main : Lexing.lexbuf -> result = <fun>
val __ocaml_lex_main_rec : Lexing.lexbuf -> int ->
    result = <fun>
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"
```
Problem

- How to get lexer to look at more than the first token at one time?
- Answer: *action* has to tell it to -- recursive calls
  - Not what you want to sew this together with ocamlyacc
- 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

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 = "(*"
lет 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\} |
| eos | \{ [] \} |
| _ | \{ 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
- Pushdown automata
- Whole family more of grammars and automata – covered in automata theory
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*
Sample Grammar

- Language: Parenthesized sums of 0’s and 1’s
- $<\text{Sum}> ::= 0$
- $<\text{Sum}> ::= 1$
- $<\text{Sum}> ::= <\text{Sum}> + <\text{Sum}>$
- $<\text{Sum}> ::= (<\text{Sum}>)$
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 derivation

- Derivation called right-most if always replace the right-most non-terminal
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

- Start with the start symbol:

\[ \langle \text{Sum} \rangle \Rightarrow \]
BNF Derivations

- Pick a non-terminal

\texttt{<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>`
  - `=> ( <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:
  - \(<\text{Sum}\> ::= <\text{Sum}\> + <\text{Sum}\>

\(<\text{Sum}\> \Rightarrow <\text{Sum}\> + <\text{Sum}\>

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

\Rightarrow ( <\text{Sum}\> + <\text{Sum}\> ) + <\text{Sum}\>
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`

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

Pick a non-terminal:

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

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

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

- Alternatives: allow rules of from $X ::= y \mid z$
  - Abbreviates $X ::= y$, $X ::= z$
- Options: $X ::= y \{v\} z$
  - Abbreviates $X ::= y \lor z$, $X ::= y z$
- Repetition: $X ::= y \{v\}^* z$
  - Can be eliminated by adding new nonterminal $V$ and rules $X ::= y z$, $X ::= y V z$, $V ::= v$, $V ::= v V$
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:

\[
\begin{align*}
\texttt{<exp>} & \ ::= \texttt{<factor>} \\
& \quad \mid \texttt{<factor>} + \texttt{<factor>}
\end{align*}
\]

\[
\begin{align*}
\texttt{<factor>} & \ ::= \texttt{<bin>} \\
& \quad \mid \texttt{<bin>} \times \texttt{<exp>}
\end{align*}
\]

\[
\begin{align*}
\texttt{<bin>} & \ ::= \ 0 \mid 1
\end{align*}
\]

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

- \[ 1 \times 1 + 0: \quad \langle \text{exp} \rangle \]

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

1 * 1 + 0: \[\langle \text{exp} \rangle \]
\[
\begin{array}{c}
\langle \text{factor} \rangle \\
\end{array}
\]

Use rule: \(\langle \text{exp} \rangle ::= \langle \text{factor} \rangle\)
Example cont.

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

\[<\text{factor}>\]

\[<\text{bin}> * <\text{exp}>\]

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

1 * 1 + 0:  \(<exp>\)

\(<factor>\)

\(<bin>\)  *  \(<exp>\)

1  \(<factor>\)  +  \(<factor>\)

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

1 * 1 + 0:

Use rule: \(<factor> ::= <bin>\)
Example cont.

1 * 1 + 0:

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

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

\(<\text{factor}>\)

\(<\text{bin}>\) * \(<\text{exp}>\)

1 \(<\text{factor}>\) + \(<\text{factor}>\)

\(<\text{bin}>\)

1 \(<\text{bin}>\)

1 0

Fringe of tree is string generated by grammar
Your Turn: 1 * 0 + 0 * 1

```
<exp>
/   |   \
<fact>  +  <fact>
/   |   \   /   |   \\
<b> * <e> <b> * <e>
```