

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

10/22/24

1

Unification Problem

Given a set of pairs of terms (“equations”)

$$\{(s_1, t_1), (s_2, t_2), \dots, (s_n, t_n)\}$$

(the *unification problem*) does there exist a substitution σ (the *unification solution*) of terms for variables such that

$$\sigma(s_i) = \sigma(t_i),$$

for all $i = 1, \dots, n$?

10/22/24

2

Uses for Unification

- Type Inference and type checking
- Pattern matching as in OCaml
 - Can use a simplified version of algorithm
- Logic Programming - Prolog
- Simple parsing

10/22/24

3

Unification Algorithm

- Let $S = \{(s_1 = t_1), (s_2 = t_2), \dots, (s_n = t_n)\}$ be a unification problem.
- Case $S = \{ \}$: $\text{Unif}(S) = \text{Identity function}$ (i.e., no substitution)
- Case $S = \{(s, t)\} \cup S'$: Four main steps

10/22/24

4

Unification Algorithm

- **Delete:** if $s = t$ (they are the same term) then $\text{Unif}(S) = \text{Unif}(S')$
- **Decompose:** if $s = f(q_1, \dots, q_m)$ and $t = f(r_1, \dots, r_m)$ (same f , same m !), then $\text{Unif}(S) = \text{Unif}(\{(q_1, r_1), \dots, (q_m, r_m)\} \cup S')$
- **Orient:** if $t = x$ is a variable, and s is not a variable, $\text{Unif}(S) = \text{Unif}(\{(x = s)\} \cup S')$

10/22/24

5

Unification Algorithm

- **Eliminate:** if $s = x$ is a variable, and x does not occur in t (the occurs check), then
 - Let $\varphi = \{x \rightarrow t\}$
 - $\text{Unif}(S) = \text{Unif}(\varphi(S')) \circ \{x \rightarrow t\}$
 - Let $\psi = \text{Unif}(\varphi(S'))$
 - $\text{Unif}(S) = \{x \rightarrow \psi(t)\} \circ \psi$
 - Note: $\{x \rightarrow a\} \circ \{y \rightarrow b\} = \{y \rightarrow (\{x \rightarrow a\}(b))\} \circ \{x \rightarrow a\}$ if y not in a

10/22/24

6

Tricks for Efficient Unification

- Don't return substitution, rather do it incrementally
- Make substitution be constant time
 - Requires implementation of terms to use mutable structures (or possibly lazy structures)
 - We won't discuss these

10/22/24

7

Example

- x, y, z variables, f, g constructors
- Unify $\{(f(x) = f(g(f(z), y))), (g(y, y) = x)\} = ?$

10/22/24

8

Example

- x, y, z variables, f, g constructors
- $S = \{(f(x) = f(g(f(z), y))), (g(y, y) = x)\}$ is nonempty
- Unify $\{(f(x) = f(g(f(z), y))), (g(y, y) = x)\} = ?$

10/22/24

9

Example

- x, y, z variables, f, g constructors
- Pick a pair: $(g(y, y) = x)$
- Unify $\{(f(x) = f(g(f(z), y))), (g(y, y) = x)\} = ?$

10/22/24

10

Example

- x, y, z variables, f, g constructors
- Pick a pair: $(g(y, y) = x)$
- Orient: $(x = g(y, y))$
- Unify $\{(f(x) = f(g(f(z), y))), (g(y, y) = x)\} =$
Unify $\{(f(x) = f(g(f(z), y))), (x = g(y, y))\}$
by Orient

10/22/24

11

Example

- x, y, z variables, f, g constructors
- Unify $\{(f(x) = f(g(f(z), y))), (x = g(y, y))\} = ?$

10/22/24

12

Example

- x, y, z variables, f, g constructors
- $\{(f(x) = f(g(f(z), y))), (x = g(y, y))\}$ is non-empty
- Unify $\{(f(x) = f(g(f(z), y))), (x = g(y, y))\} = ?$

10/22/24

13

Example

- x, y, z variables, f, g constructors
- Pick a pair: $(x = g(y, y))$
- Unify $\{(f(x) = f(g(f(z), y))), (x = g(y, y))\} = ?$

10/22/24

14

Example

- x, y, z variables, f, g constructors
- Pick a pair: $(x = g(y, y))$
- Eliminate x with substitution $\{x \rightarrow g(y, y)\}$
 - Check: x not in $g(y, y)$
- Unify $\{(f(x) = f(g(f(z), y))), (x = g(y, y))\} = ?$

10/22/24

15

Example

- x, y, z variables, f, g constructors
- Pick a pair: $(x = g(y, y))$
- Eliminate x with substitution $\{x \rightarrow g(y, y)\}$
- Unify $\{(f(x) = f(g(f(z), y))), (x = g(y, y))\} =$
Unify $\{(f(g(y, y)) = f(g(f(z), y)))\}$
 - $\{x \rightarrow g(y, y)\}$

10/22/24

16

Example

- x, y, z variables, f, g constructors
- Unify $\{(f(g(y, y)) = f(g(f(z), y)))\}$
 - $\{x \rightarrow g(y, y)\} = ?$

10/22/24

17

Example

- x, y, z variables, f, g constructors
- $\{(f(g(y, y)) = f(g(f(z), y)))\}$ is non-empty
- Unify $\{(f(g(y, y)) = f(g(f(z), y)))\}$
 - $\{x \rightarrow g(y, y)\} = ?$

10/22/24

18

Example

- x, y, z variables, f, g constructors
- Pick a pair: $(f(g(y, y)) = f(g(f(z), y)))$

- Unify $\{(f(g(y, y)) = f(g(f(z), y)))\}$
 - $\{x \rightarrow g(y, y)\} = ?$

10/22/24

19

Example

- x, y, z variables, f, g constructors
- Pick a pair: $(f(g(y, y)) = f(g(f(z), y)))$
- Decompose: $(f(g(y, y)) = f(g(f(z), y)))$ becomes $\{(g(y, y) = g(f(z), y))\}$

- Unify $\{(f(g(y, y)) = f(g(f(z), y)))\}$
 - $\{x \rightarrow g(y, y)\} =$
- Unify $\{(g(y, y) = g(f(z), y))\} \circ \{x \rightarrow g(y, y)\}$

10/22/24

20

Example

- x, y, z variables, f, g constructors
- $\{(g(y, y) = g(f(z), y))\}$ is non-empty

- Unify $\{(g(y, y) = g(f(z), y))\}$
 - $\{x \rightarrow g(y, y)\} = ?$

10/22/24

21

Example

- x, y, z variables, f, g constructors
- Pick a pair: $(g(y, y) = g(f(z), y))$

- Unify $\{(g(y, y) = g(f(z), y))\}$
 - $\{x \rightarrow g(y, y)\} = ?$

10/22/24

22

Example

- x, y, z variables, f, g constructors
- Pick a pair: $(f(g(y, y)) = f(g(f(z), y)))$
- Decompose: $(g(y, y) = g(f(z), y))$ becomes $\{(y = f(z)); (y = y)\}$

- Unify $\{(g(y, y) = g(f(z), y))\} \circ \{x \rightarrow g(y, y)\} =$
Unify $\{(y = f(z)); (y = y)\} \circ \{x \rightarrow g(y, y)\}$

10/22/24

23

Example

- x, y, z variables, f, g constructors

- Unify $\{(y = f(z)); (y = y)\} \circ \{x \rightarrow g(y, y)\} = ?$

10/22/24

24

Example

- x, y, z variables, f, g constructors
- $\{(y = f(z)); (y = y)\} \circ \{x \rightarrow g(y, y)\}$ is non-empty
- Unify $\{(y = f(z)); (y = y)\} \circ \{x \rightarrow g(y, y)\} = ?$

10/22/24

25

Example

- x, y, z variables, f, g constructors
- Pick a pair: $(y = f(z))$
- Unify $\{(y = f(z)); (y = y)\} \circ \{x \rightarrow g(y, y)\} = ?$

10/22/24

26

Example

- x, y, z variables, f, g constructors
- Pick a pair: $(y = f(z))$
- Eliminate y with $\{y \rightarrow f(z)\}$
- Unify $\{(y = f(z)); (y = y)\} \circ \{x \rightarrow g(y, y)\} =$
Unify $\{(f(z) = f(z))\}$
o $\{y \rightarrow f(z)\} \circ \{x \rightarrow g(y, y)\} =$
Unify $\{(f(z) = f(z))\}$
o $\{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\}$

10/22/24

27

Example

- x, y, z variables, f, g constructors
- Unify $\{(f(z) = f(z))\}$
o $\{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\} = ?$

10/22/24

28

Example

- x, y, z variables, f, g constructors
- $\{(f(z) = f(z))\}$ is non-empty
- Unify $\{(f(z) = f(z))\}$
o $\{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\} = ?$

10/22/24

29

Example

- x, y, z variables, f, g constructors
- Pick a pair: $(f(z) = f(z))$
- Unify $\{(f(z) = f(z))\}$
o $\{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\} = ?$

10/22/24

30

Example

- x, y, z variables, f, g constructors
- Pick a pair: $(f(z) = f(z))$
- Delete
- Unify $\{(f(z) = f(z))\}$
 - $\{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\} =$
- Unify $\{\} \circ \{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\}$

10/22/24

31

Example

- x, y, z variables, f, g constructors
- Unify $\{\} \circ \{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\} = ?$

10/22/24

32

Example

- x, y, z variables, f, g constructors
- $\{\}$ is empty
- Unify $\{\} =$ identity function
- Unify $\{\} \circ \{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\} =$
 $\{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\}$

10/22/24

33

Example

- Unify $\{(f(x) = f(g(f(z), y))), (g(y, y) = x)\} =$
 $\{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\}$

$$\begin{aligned} f(x) &= f(g(f(z), y)) \\ \rightarrow f(g(f(z), f(z))) &= f(g(f(z), f(z))) \end{aligned}$$

$$\begin{aligned} g(y, y) &= x \\ \rightarrow g(f(z), f(z)) &= g(f(z), f(z)) \end{aligned}$$

10/22/24

34

Example of Failure: Decompose

- Unify $\{(f(x, g(y)) = f(h(y), x))\}$
- Decompose: $(f(x, g(y)) = f(h(y), x))$
- = Unify $\{(x = h(y)), (g(y) = x)\}$
- Orient: $(g(y) = x)$
- = Unify $\{(x = h(y)), (x = g(y))\}$
- Eliminate: $(x = h(y))$
- Unify $\{(h(y) = g(y))\} \circ \{x \rightarrow h(y)\}$
- No rule to apply! Decompose fails!

10/22/24

35

Example of Failure: Occurs Check

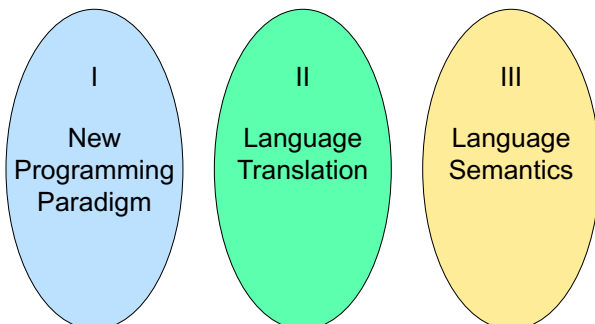
- Unify $\{(f(x, g(x)) = f(h(x), x))\}$
- Decompose: $(f(x, g(x)) = f(h(x), x))$
- = Unify $\{(x = h(x)), (g(x) = x)\}$
- Orient: $(g(x) = x)$
- = Unify $\{(x = h(x)), (x = g(x))\}$
- No rules apply.

10/22/24

36

Programming Languages & Compilers

Three Main Topics of the Course

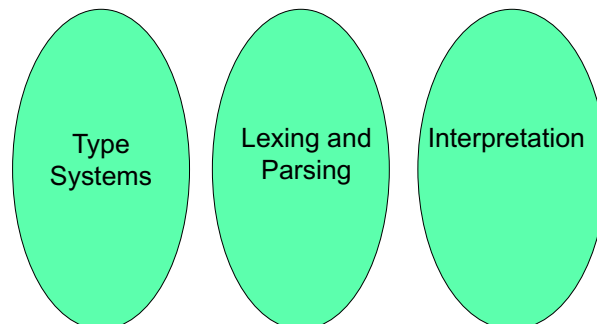


10/22/24

37

Programming Languages & Compilers

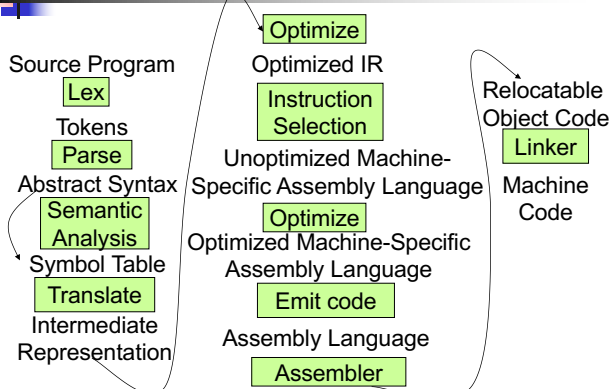
II : Language Translation



10/22/24

38

Major Phases of a Compiler



Modified from "Modern Compiler Implementation in ML", by Andrew Appel

Where We Are Going Next?

- We want to turn strings (code) into computer instructions
- Done in phases
- Turn strings into abstract syntax trees (parse)
- Translate abstract syntax trees into executable instructions (interpret or compile)

10/22/24

40

Meta-discourse

- Language Syntax and Semantics
- Syntax
 - Regular Expressions, DFSAs and NDFSAs
 - Grammars
- Semantics
 - Natural Semantics
 - Transition Semantics

10/22/24

41

Language Syntax

- Syntax is the description of which strings of symbols are meaningful expressions in a language
- It takes more than syntax to understand a language; need meaning (semantics) too
- Syntax is the entry point

10/22/24

42

Syntax of English Language

- Pattern 1

Subject	Verb
David	sings
The dog	barked
Susan	yawned

- Pattern 2

Subject	Verb	Direct Object
David	sings	ballads
The professor	wants	to retire
The jury	found	the defendant guilty

10/22/24

43

Elements of Syntax

- Character set – previously always ASCII, now often 64 character sets
- Keywords – usually reserved
- Special constants – cannot be assigned to
- Identifiers – can be assigned to
- Operator symbols
- Delimiters (parenthesis, braces, brackets)
- Blanks (aka white space)

10/22/24

44

Elements of Syntax

- Expressions
 - if ... then begin ... ; ... end else begin ... ; ... end
- Type expressions
 - $type_{expr_1} \rightarrow type_{expr_2}$
- Declarations (in functional languages)
 - let *pattern* = *expr*
- Statements (in imperative languages)
 - $a = b + c$
- Subprograms
 - let *pattern*₁ = *expr*₁ in *expr*

10/22/24

45

Elements of Syntax

- Modules
- Interfaces
- Classes (for object-oriented languages)

10/22/24

46

Lexing and Parsing

- Converting strings to abstract syntax trees done in two phases
 - Lexing:** Converting string (or streams of characters) into lists (or streams) of tokens (the “words” of the language)
 - Specification Technique: Regular Expressions
 - Parsing:** Convert a list of tokens into an abstract syntax tree
 - Specification Technique: BNF Grammars

10/22/24

47

Formal Language Descriptions

- Regular expressions, regular grammars, finite state automata
- Context-free grammars, BNF grammars, syntax diagrams
- Whole family more of grammars and automata – covered in automata theory

10/22/24

48

Grammars

- Grammars are formal descriptions of which strings over a given character set are in a particular language
- Language designers write grammar
- Language implementers use grammar to know what programs to accept
- Language users use grammar to know how to write legitimate programs

10/22/24

49

Regular Expressions - Review

- Start with a given character set – **a, b, c...**
- $L(\epsilon) = \{""\}$
- Each character is a regular expression
 - It represents the set of one string containing just that character
 - $L(a) = \{a\}$

10/22/24

50

Regular Expressions

- If **x** and **y** are regular expressions, then **xy** is a regular expression
 - It represents the set of all strings made from first a string described by **x** then a string described by **y**
- If $L(x) = \{a, ab\}$ and $L(y) = \{c, d\}$
then $L(xy) = \{ac, ad, abc, abd\}$

10/22/24

51

Regular Expressions

- If **x** and **y** are regular expressions, then **x ∨ y** is a regular expression
 - It represents the set of strings described by either **x** or **y**
- If $L(x) = \{a, ab\}$ and $L(y) = \{c, d\}$
then $L(x \vee y) = \{a, ab, c, d\}$

10/22/24

52

Regular Expressions

- If **x** is a regular expression, then so is **(x)**
 - It represents the same thing as **x**
- If **x** is a regular expression, then so is **x***
 - It represents strings made from concatenating zero or more strings from **x**

If $L(x) = \{a, ab\}$ then $L(x^*) = \{ "", a, ab, aa, aab, abab, \dots \}$
- ϵ
 - It represents $\{""\}$, set containing the empty string
- \emptyset
 - It represents $\{ \}$, the empty set

10/22/24

53

Example Regular Expressions

- **(0 ∨ 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) ∨ (10))***
 - You tell me
- Regular expressions (equivalently, regular grammars) important for lexing, breaking strings into recognized words

10/22/24

54

Right 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
 $\langle \text{nonterminal} \rangle ::= \epsilon$
- 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 \cong states; rule \cong edge

10/22/24

55

Example

- Right regular grammar:
 $\langle \text{Balanced} \rangle ::= \epsilon$
 $\langle \text{Balanced} \rangle ::= 0 \langle \text{OneAndMore} \rangle$
 $\langle \text{Balanced} \rangle ::= 1 \langle \text{ZeroAndMore} \rangle$
 $\langle \text{OneAndMore} \rangle ::= 1 \langle \text{Balanced} \rangle$
 $\langle \text{ZeroAndMore} \rangle ::= 0 \langle \text{Balanced} \rangle$
- Generates even length strings where every initial substring of even length has same number of 0's as 1's

10/22/24

56

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

10/22/24

57

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 = $0 \vee (1 \vee \dots \vee 9)(0 \vee \dots \vee 9)^* \vee (1 \vee \dots \vee 9)(0 \vee \dots \vee 9)^*$
 - Keywords: if = if, while = while,...

10/22/24

58

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]

10/22/24

59

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

10/22/24

60

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.

10/22/24

61

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.

10/22/24

62

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`

10/22/24

63

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
  main newlexbuf
  }
```

10/22/24

64

General Input

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

10/22/24

65

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

10/22/24

66

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*

10/22/24

67

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
- `e1 / e2`: choice - what was $e_1 \vee e_2$

10/22/24

68

Ocamllex Regular Expression

- `[c1 - c2]`: choice of any character between first and second inclusive, as determined by character codes
- `[^c1 - c2]`: choice of any character NOT in set
- `e*`: same as before
- `e+`: same as `e e*`
- `e?`: option - was $e \vee \varepsilon$
- `(e)`: same as `e`

10/22/24

69

Ocamllex Regular Expression

- `e1 # e2`: the characters in `e1` but not in `e2`; `e1` and `e2` must describe just sets of characters
- `ident`: abbreviation for earlier reg exp in `let ident = regexp`
- `e1 as id`: binds the result of `e1` to `id` to be used in the associated *action*

10/22/24

70

Ocamllex Manual

- More details can be found at
Version for ocaml 4.07:
<https://v2.ocaml.org/releases/4.07/htmlman/lexyacc.html>
- Current version (ocaml 4.14)
<https://v2.ocaml.org/releases/4.14/htmlman/lexyacc.html>
(same, except formatting, I think)

10/22/24

71

End of Lect 18

10/22/24

72

Example : test.mll

```
{ 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 +
```

10/22/24

73

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 }
```

10/22/24

74

Example

```
# #use "test.mll";
...
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?!?

10/22/24

75

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"
```

10/22/24

76

Your Turn

- Work on MP8
 - Add a few keywords
 - Implement booleans and unit
 - Implement Ints and Floats
 - Implement identifiers

10/22/24

77

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 `ocaml yacc`
- Side Benefit: can add "state" into lexing
- Note: already used this with the `_` case

10/22/24

78

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 }
```

10/22/24

79

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

10/22/24

80

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 }
```

10/22/24

81

Dealing with comments

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

10/22/24

82

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 }
```

10/22/24

83

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 }
```

10/22/24

84

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 }
```

10/22/24

85

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

10/22/24

86

Sample Grammar

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

10/22/24

87

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*

10/22/24

88

BNF Grammars

- BNF rules (aka *productions*) have form $\mathbf{X} ::= \gamma$
where **X** is any nonterminal and γ 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

10/22/24

89

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

10/22/24

90

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

10/22/24

91

BNF Derivations

- Start with the start symbol:

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

10/22/24

92

BNF Derivations

- Pick a non-terminal

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

10/22/24

93

BNF Derivations

- Pick a rule and substitute:
 - $\langle \text{Sum} \rangle ::= \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

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

10/22/24

94

BNF Derivations

- Pick a non-terminal:

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

10/22/24

95

BNF Derivations

- Pick a rule and substitute:

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

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

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

10/22/24

96

BNF Derivations

- Pick a non-terminal:

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

10/22/24

97

BNF Derivations

- Pick a rule and substitute:

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

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

10/22/24

98

BNF Derivations

- Pick a non-terminal:

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

10/22/24

99

BNF Derivations

- Pick a rule and substitute:

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

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

10/22/24

100

BNF Derivations

- Pick a non-terminal:

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

10/22/24

101

BNF Derivations

- Pick a rule and substitute:

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

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

10/22/24

102

BNF Derivations

- Pick a non-terminal:

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

10/22/24

103

BNF Derivations

- Pick a rule and substitute

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

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

10/22/24

104

BNF Derivations

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

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

10/22/24

105

BNF Derivations

- Pick a non-terminal:

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

10/22/24

106