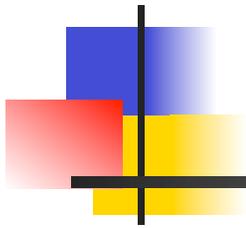


Programming Languages and Compilers (CS 421)

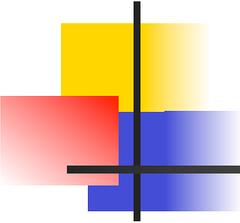


Elsa L Gunter

2112 SC, UIUC

<http://courses.engr.illinois.edu/cs421>

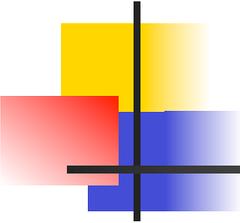
Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha



Example

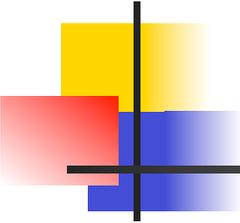
- x, y, z variables, f, g constructors

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



Example

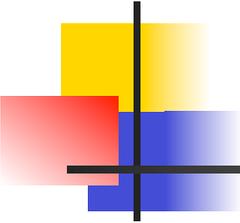
- x, y, z variables, f, g constructors
- $S = \{(f(x) = f(g(f(z), y))), (g(y, y) = x)\}$ is nonempty
- Unify $\{(f(x) = f(g(f(z), y))), (g(y, y) = x)\} = ?$



Example

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

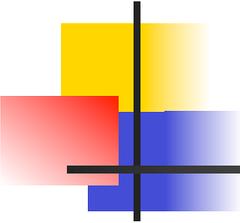
- Unify $\{(f(x) = f(g(f(z), y))), (g(y, y) = x)\} = ?$



Example

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

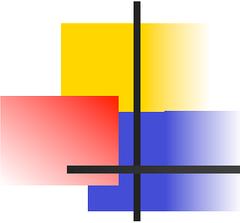
- Unify $\{(f(x) = f(g(f(z), y))), (g(y, y) = x)\} =$
Unify $\{(f(x) = f(g(f(z), y))), (x = g(y, y))\}$
by Orient



Example

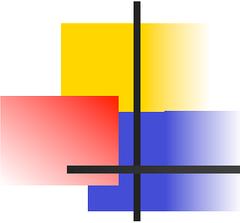
- x, y, z variables, f, g constructors

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



Example

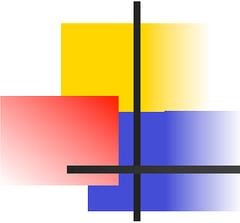
- x, y, z variables, f, g constructors
- $\{(f(x) = f(g(f(z), y))), (x = g(y, y))\}$ is non-empty
- Unify $\{(f(x) = f(g(f(z), y))), (x = g(y, y))\} = ?$



Example

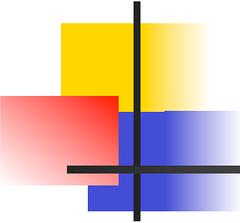
- x, y, z variables, f, g constructors
- Pick a pair: $(x = g(y, y))$

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



Example

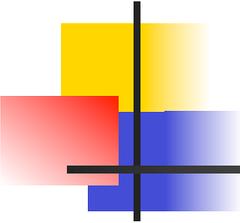
- x, y, z variables, f, g constructors
- Pick a pair: $(x = g(y, y))$
- Eliminate x with substitution $\{x \rightarrow g(y, y)\}$
 - Check: x not in $g(y, y)$
- Unify $\{(f(x) = f(g(f(z), y))), (x = g(y, y))\} = ?$



Example

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

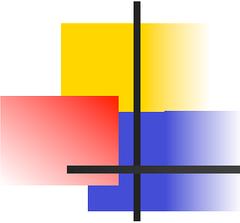
- Unify $\{(f(x) = f(g(f(z), y))), (x = g(y, y))\} =$
Unify $\{(f(g(y, y)) = f(g(f(z), y)))\}$
 - $\{x \rightarrow g(y, y)\}$



Example

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

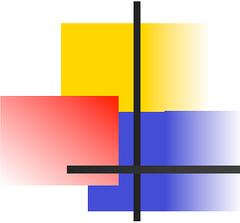
- Unify $\{(f(g(y, y)) = f(g(f(z), y)))\}$
 - $\{x \rightarrow g(y, y)\} = ?$



Example

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

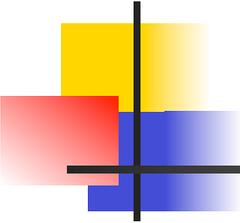
- Unify $\{(f(g(y, y)) = f(g(f(z), y)))\}$
 - $\{x \rightarrow g(y, y)\} = ?$



Example

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

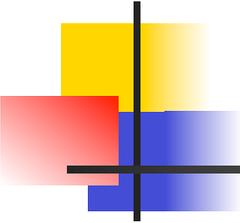
- Unify $\{(f(g(y, y)) = f(g(f(z), y)))\}$
 - $\{x \rightarrow g(y, y)\} = ?$



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 $\{(g(y, y) = g(f(z), y))\}$

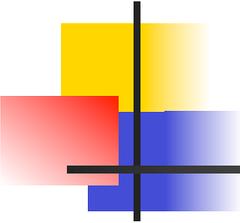
- Unify $\{(f(g(y, y)) = f(g(f(z), y)))\}$
 - $\{x \rightarrow g(y, y)\} =$
Unify $\{(g(y, y) = g(f(z), y))\} \circ \{x \rightarrow g(y, y)\}$



Example

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

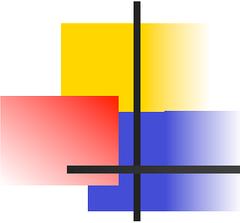
- Unify $\{(g(y, y) = g(f(z), y))\}$
 - $\{x \rightarrow g(y, y)\} = ?$



Example

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

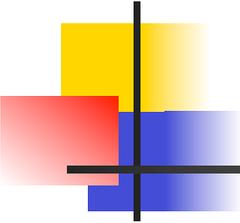
- Unify $\{(g(y, y) = g(f(z), y))\}$
 - $\{x \rightarrow g(y, y)\} = ?$



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 $\{(y = f(z)); (y = y)\}$

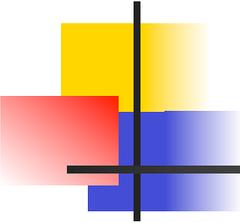
- Unify $\{(g(y, y) = g(f(z), y))\} \circ \{x \rightarrow g(y, y)\} =$
Unify $\{(y = f(z)); (y = y)\} \circ \{x \rightarrow g(y, y)\}$



Example

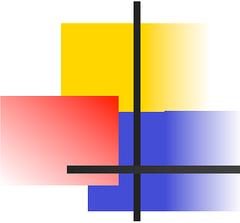
- x, y, z variables, f, g constructors

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



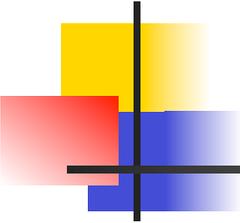
Example

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



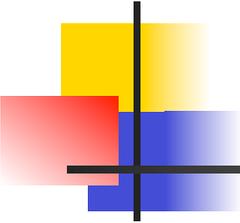
Example

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



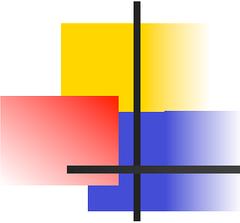
Example

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



Example

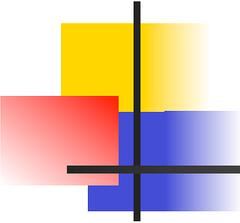
- x, y, z variables, f, g constructors
- Unify $\{(f(z) = f(z))\}$
 - $\{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\} = ?$



Example

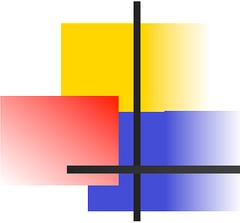
- x, y, z variables, f, g constructors
- $\{(f(z) = f(z))\}$ is non-empty

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



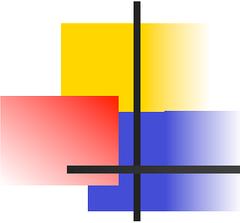
Example

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



Example

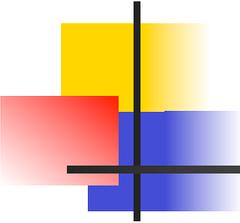
- x, y, z variables, f, g constructors
- Pick a pair: $(f(z) = f(z))$
- Delete
- Unify $\{(f(z) = f(z))\}$
 - $\{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\} =$
Unify $\{\}$ ○ $\{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\}$



Example

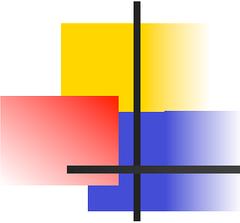
- x, y, z variables, f, g constructors

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



Example

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



Example

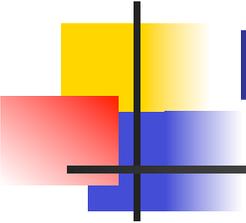
- Unify $\{(f(x) = f(g(f(z), y))), (g(y, y) = x)\} = \{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\}$

$$f(x) = f(g(f(z), y))$$

$$\rightarrow f(g(f(z), f(z))) = f(g(f(z), f(z)))$$

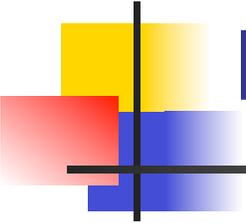
$$g(y, y) = x$$

$$\rightarrow g(f(z), f(z)) = g(f(z), f(z))$$



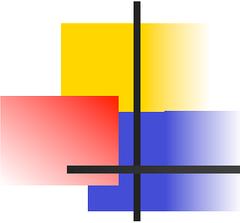
Example of Failure: Decompose

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



Example of Failure: Occurs Check

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



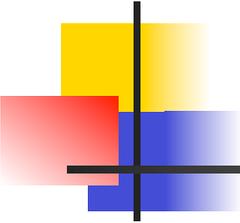
Programming Languages & Compilers

Three Main Topics of the Course

I
New
Programming
Paradigm

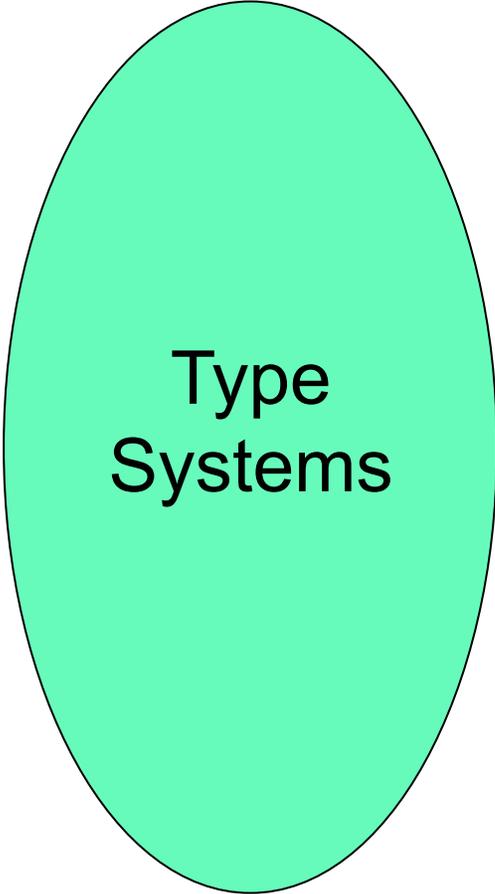
II
Language
Translation

III
Language
Semantics

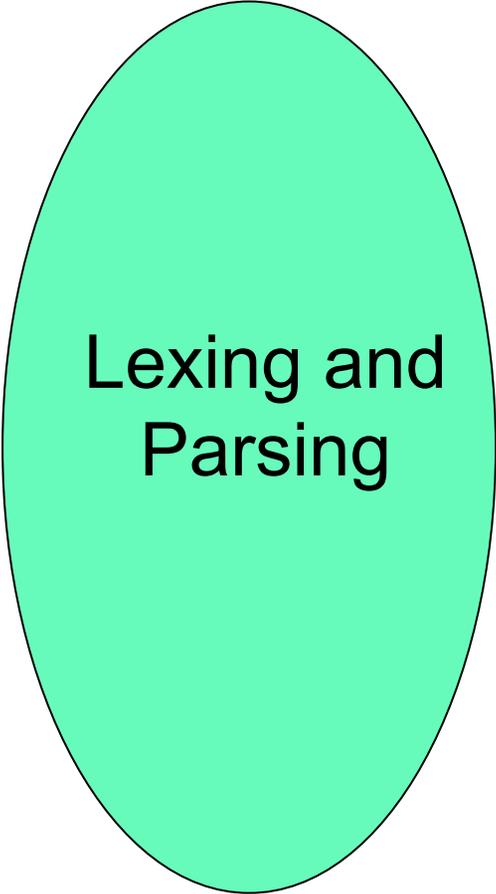


Programming Languages & Compilers

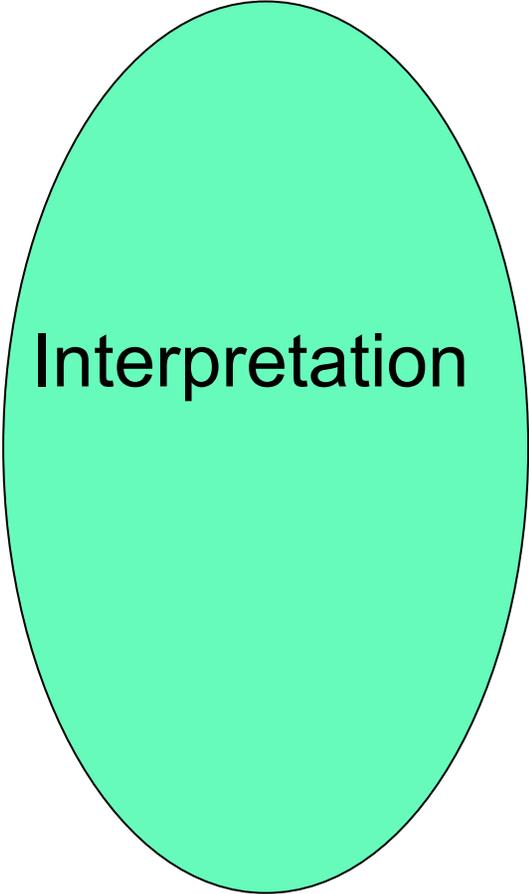
II : Language Translation



Type
Systems



Lexing and
Parsing



Interpretation

Major Phases of a Compiler

Source Program

Lex

Tokens

Parse

Abstract Syntax

Semantic
Analysis

Symbol Table

Translate

Intermediate
Representation

Optimize

Optimized IR

Instruction
Selection

Unoptimized Machine-
Specific Assembly Language

Optimize

Optimized Machine-Specific
Assembly Language

Emit code

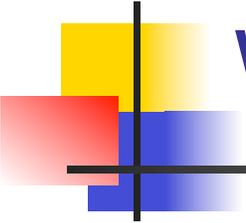
Assembly Language

Assembler

Relocatable
Object Code

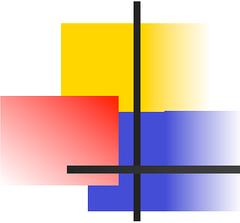
Linker

Machine
Code



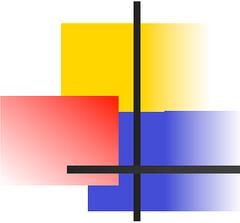
Where We Are Going Next?

- 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)



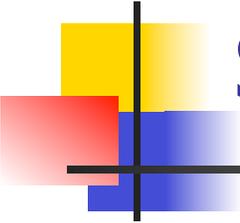
Meta-discourse

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



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



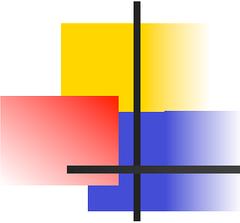
Syntax of English Language

- Pattern 1

Subject	Verb
<i>David</i>	<i>sings</i>
<i>The dog</i>	<i>barked</i>
<i>Susan</i>	<i>yawned</i>

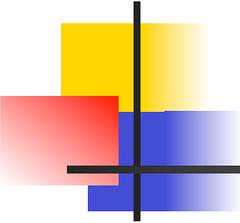
- Pattern 2

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



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)



Elements of Syntax

- Expressions

if ... then begin ... ; ... end else begin ... ; ... end

- Type expressions

typexpr₁ -> typexpr₂

- Declarations (in functional languages)

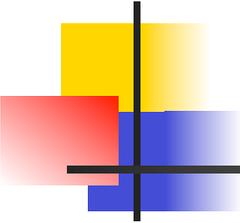
let *pattern* = *expr*

- Statements (in imperative languages)

a = b + c

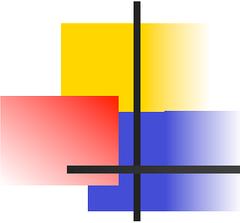
- Subprograms

let *pattern₁* = *expr₁* in *expr*



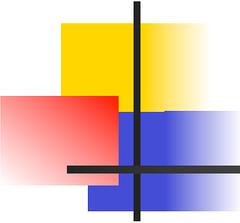
Elements of Syntax

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



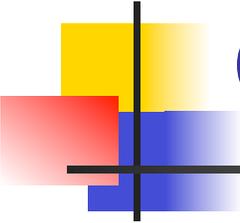
Lexing and Parsing

- 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



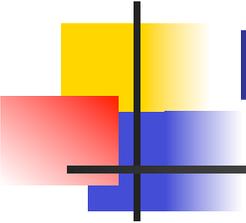
Formal Language Descriptions

- 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



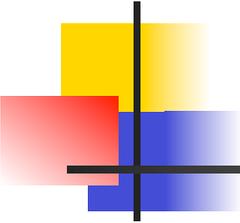
Grammars

- 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



Regular Expressions - Review

- Start with a given character set –
a, b, c...
- Each character is a regular expression
 - It represents the set of one string containing just that character

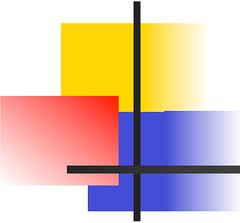


Regular Expressions

- 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**

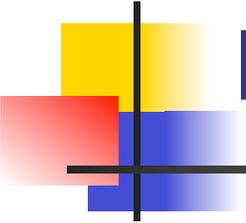
If $x = \{a, ab\}$ and $y = \{c, d\}$ then $xy = \{ac, ad, abc, abd\}$.

- 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 $x = \{a, ab\}$ and $y = \{c, d\}$ then $x \vee y = \{a, ab, c, d\}$



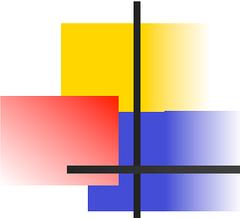
Regular Expressions

- 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 $x = \{a, ab\}$ then $x^* = \{\text{""}, a, ab, aa, aab, abab, \dots\}$
- ϵ
 - It represents $\{\text{""}\}$, set containing the empty string
 - ϕ
 - It represents $\{\}$, the empty set



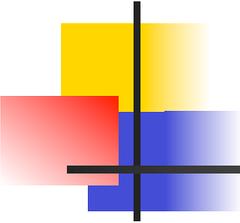
Example Regular Expressions

- **$(0 \vee 1)^* 1$**
 - 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
- **$((01) \vee (10))^*$**
 - You tell me
- Regular expressions (equivalently, regular grammars) important for lexing, breaking strings into recognized words



Regular Grammars

- Subclass of BNF (covered in detail 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
 $\langle \text{nonterminal} \rangle ::= \varepsilon$
- 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



Example

- Regular grammar:

$\langle \text{Balanced} \rangle ::= \varepsilon$

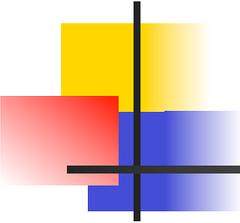
$\langle \text{Balanced} \rangle ::= 0 \langle \text{OneAndMore} \rangle$

$\langle \text{Balanced} \rangle ::= 1 \langle \text{ZeroAndMore} \rangle$

$\langle \text{OneAndMore} \rangle ::= 1 \langle \text{Balanced} \rangle$

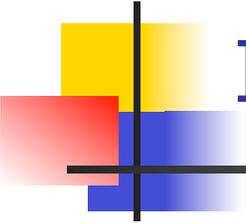
$\langle \text{ZeroAndMore} \rangle ::= 0 \langle \text{Balanced} \rangle$

- Generates even length strings where every initial substring of even length has same number of 0's as 1's



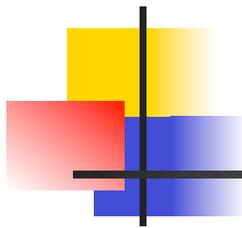
Example: Lexing

- 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 = $0 \vee (1 \vee \dots \vee 9)(0 \vee \dots \vee 9)^* \vee \sim (1 \vee \dots \vee 9)(0 \vee \dots \vee 9)^*$
 - 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 CS373 / CS374

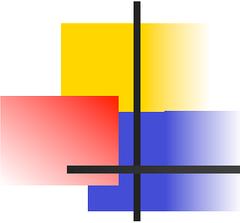


Lexing

- Different syntactic categories of “words”:
tokens

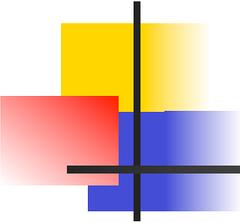
Example:

- 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]



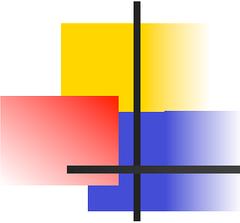
Lex, ocamllex

- 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



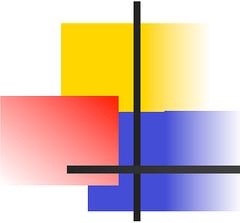
How to do it

- 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

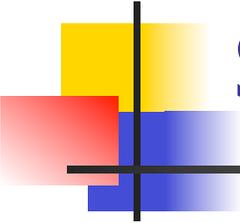
- The lexer will take the regular expressions and generate a state machine.
- The state machine will take our lexing buffer and apply the transitions...
- If we reach an accepting state from which we can go no further, the machine will perform the appropriate action.



Mechanics

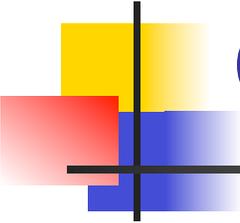
- Put table of reg exp and corresponding actions (written in ocaml) into a file *<filename>.ml*
- Call

```
ocamllex <filename>.ml
```
- 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

{ *header* }

let *ident* = *regex* ...

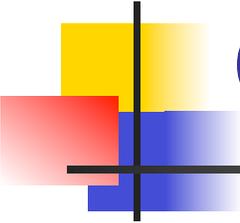
rule *entrypoint* [*arg1*... *argn*] = parse
 regex { *action* }

| ...

| *regex* { *action* }

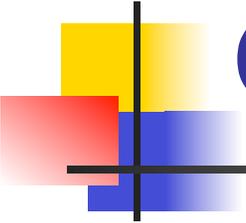
and *entrypoint* [*arg1*... *argn*] =
 parse ...and ...

{ *trailer* }



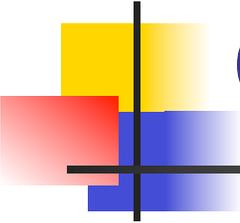
Ocamlex Input

- *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



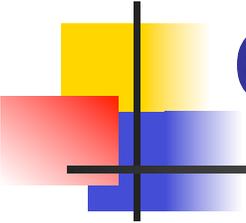
Ocamlex Input

- *<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*



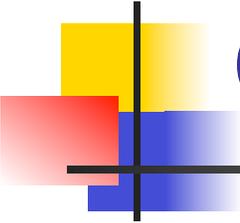
Ocamlex Regular Expression

- 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 $e_1 \vee e_2$



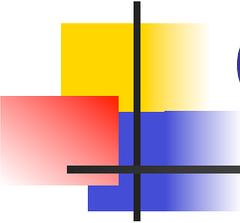
Ocamlex Regular Expression

- $[c_1 - c_2]$: choice of any character between first and second inclusive, as determined by character codes
- $[^c_1 - c_2]$: choice of any character NOT in set
- e^* : same as before
- $e+$: same as $e e^*$
- $e?$: option - was $e_1 \vee \varepsilon$



Ocamlex Regular Expression

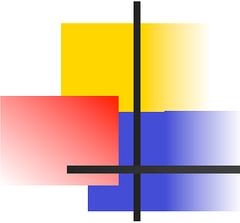
- $e_1 \# e_2$: the characters in e_1 but not in e_2 ; e_1 and e_2 must describe just sets of characters
- *ident*: abbreviation for earlier reg exp in `let ident = regexp`
- e_1 *as* *id*: binds the result of e_1 to *id* to be used in the associated *action*



Ocamllex Manual

- More details can be found at

<http://caml.inria.fr/pub/docs/manual-ocaml/lex yacc.html>



Example : test.ml

```
{ type result = Int of int | Float of float |  
  String of string }
```

```
let digit = ['0'-'9']
```

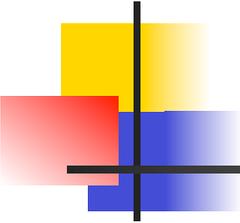
```
let digits = digit +
```

```
let lower_case = ['a'-'z']
```

```
let upper_case = ['A'-'Z']
```

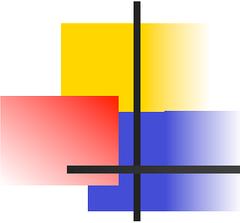
```
let letter = upper_case | lower_case
```

```
let letters = letter +
```



Example : test.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

```
# #use "test.ml";;
```

```
...
```

```
val main : Lexing.lexbuf -> result = <fun>
```

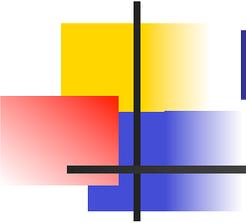
```
val __ocaml_lex_main_rec : Lexing.lexbuf -> int ->  
  result = <fun>
```

Ready to lex.

hi there 234 5.2

```
- : result = String "hi"
```

What happened to the rest?!?



Example

```
# let b = Lexing.from_channel stdin;;
```

```
# main b;;
```

```
hi 673 there
```

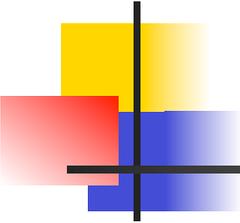
```
- : result = String "hi"
```

```
# main b;;
```

```
- : result = Int 673
```

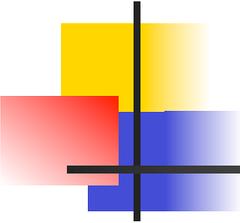
```
# main b;;
```

```
- : 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
- Side Benefit: can add “state” into lexing
- Note: already used this with the `_` case



Example

rule main = parse

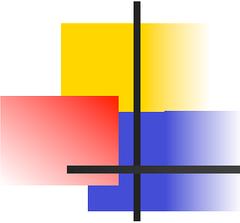
(digits) '.' digits as f { Float
(float_of_string f) :: main lexbuf }

| digits as n { Int (int_of_string n) ::
main lexbuf }

| letters as s { String s :: main
lexbuf }

| eof { [] }

| _ { main lexbuf }



Example Results

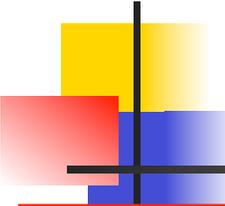
Ready to lex.

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 = "("*
```

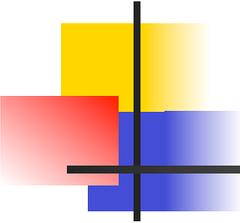
```
let close_comment = "*"
```

```
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 }
```



Dealing with comments

| open_comment { comment lexbuf }

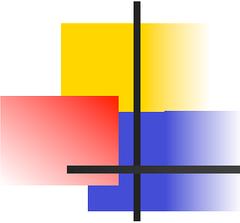
| eof { [] }

| _ { main lexbuf }

and comment = parse

close_comment { main lexbuf }

| _ { comment lexbuf }



Dealing with nested comments

rule main = parse ...

| open_comment { comment 1 lexbuf }

| eof { [] }

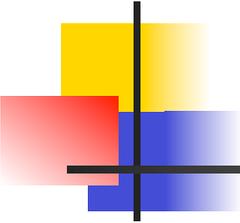
| _ { main lexbuf }

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 }

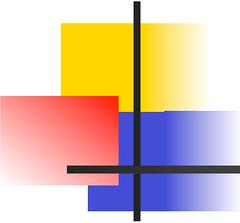
| digits as n { Int (int_of_string n) :: main
lexbuf }

| 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 }