Programming Languages and Compilers (CS 421)

Talia Ringer (they/them)

4218 SC, UIUC

https://courses.grainger.illinois.edu/cs421/fa2023/

Based heavily on slides by Elsa Gunter, which were based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Objectives for Today

- 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**)
- Today we will learn the first step of parsing, which is **lexing** those raw input strings into **tokens**

Questions from last week?

Syntax

Meta-discourse

- Language **Syntax** and **Semantics**
- **Syntax**: form
  - Regular Expressions, DFSAs and NDFSAs
  - Grammars
- **Semantics**: meaning
  - Natural Semantics
  - Transition Semantics
- Compilers and interpreters (when correctly implemented) **map** from the **syntax** of programs (as written) to their **semantics** (as executed)

Meta-discourse

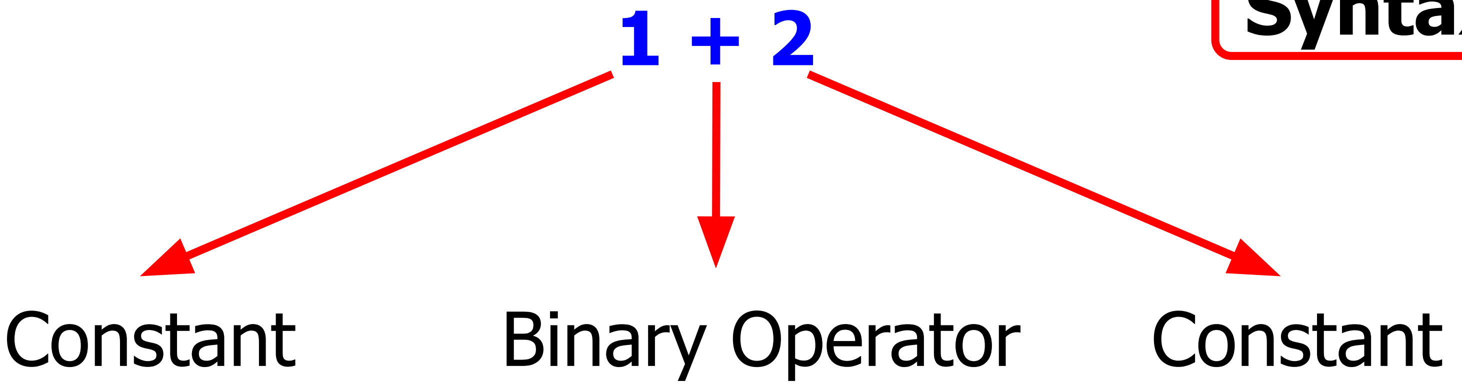
- Language **Syntax** and **Semantics**
- **Syntax**: form
  - Regular Expressions, DFSAs and NDFSAs
  - Grammars
- **Semantics**: meaning
  - Natural Semantics
  - Transition Semantics
- Compilers and interpreters (when correctly implemented) **map** from the **syntax** of programs (as written) to their **semantics** (as executed)

Meta-discourse

- Language **Syntax** and **Semantics**
- **Syntax**: form
  - Regular Expressions, DFSAs and NDFSAs
  - Grammars
- **Semantics**: meaning
  - Natural Semantics
  - Transition Semantics
- Compilers and interpreters (when correctly implemented) **map** from the **syntax** of programs (as written) to their **semantics** (as executed)

Meta-discourse

Syntax



1 + 2

Constant

Binary Operator

Constant

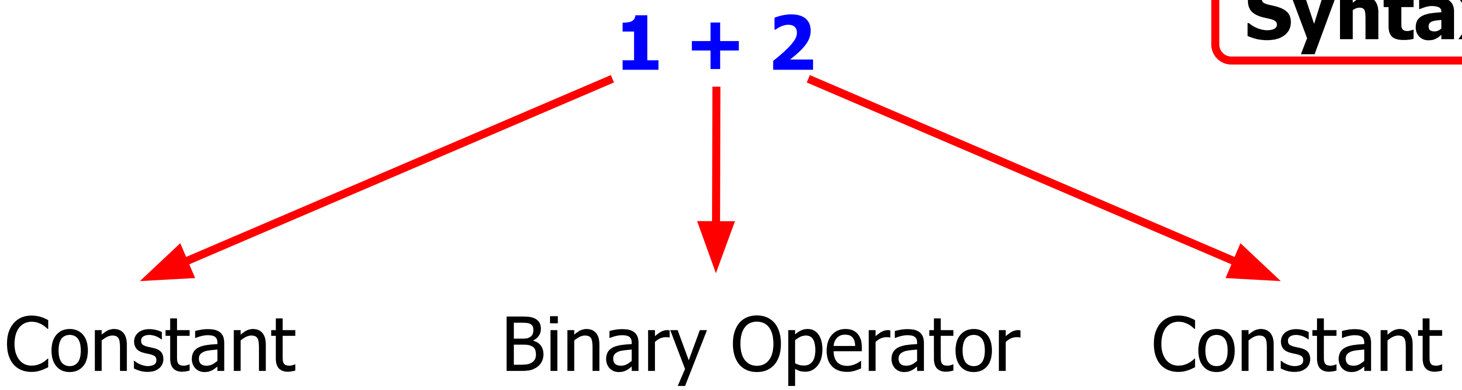
---

1 + 2 = 3

Semantics

Meta-discourse

Syntax



1 + 2

Constant

Binary Operator

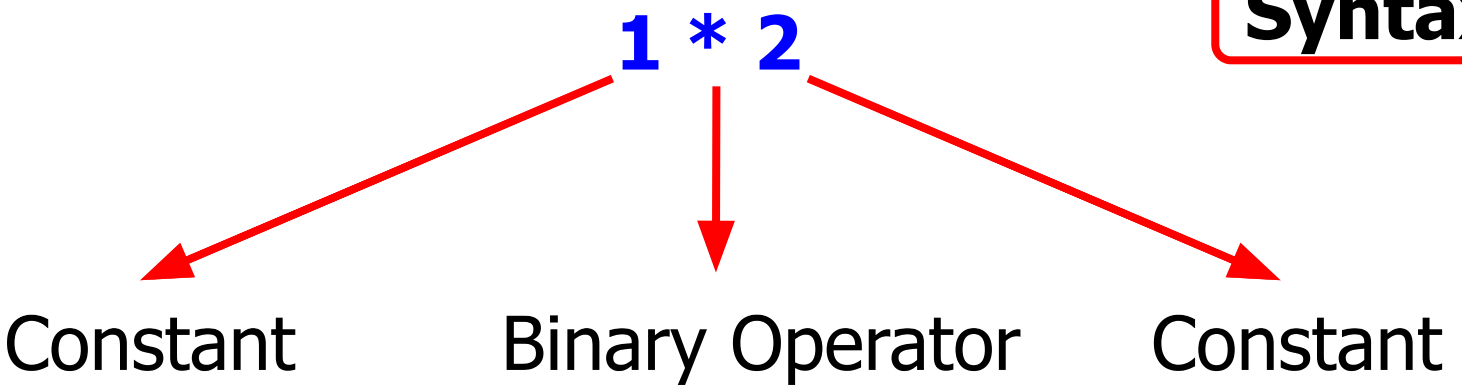
Constant

1 + 2 = 3

Semantics

Meta-discourse

Syntax



1 * 2

Constant

Binary Operator

Constant

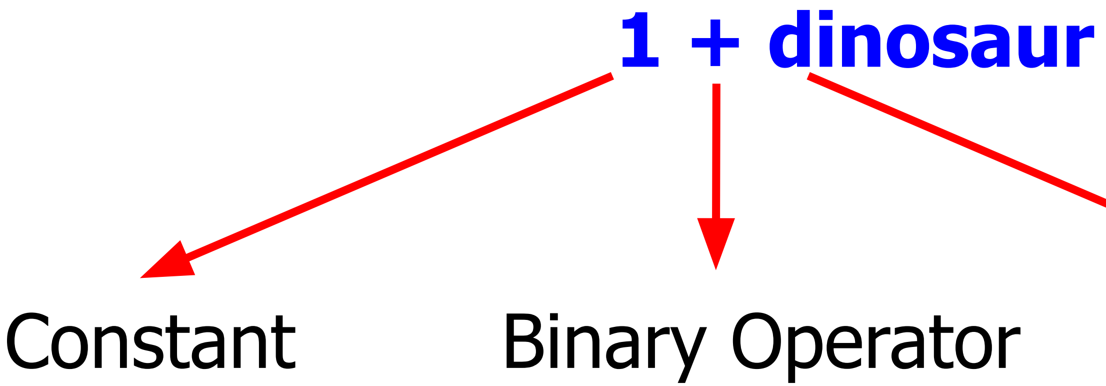
---

1 * 2 = 3

(Bizarre) Semantics

Meta-discourse

Syntax



1 + dinosaur

Constant

Binary Operator

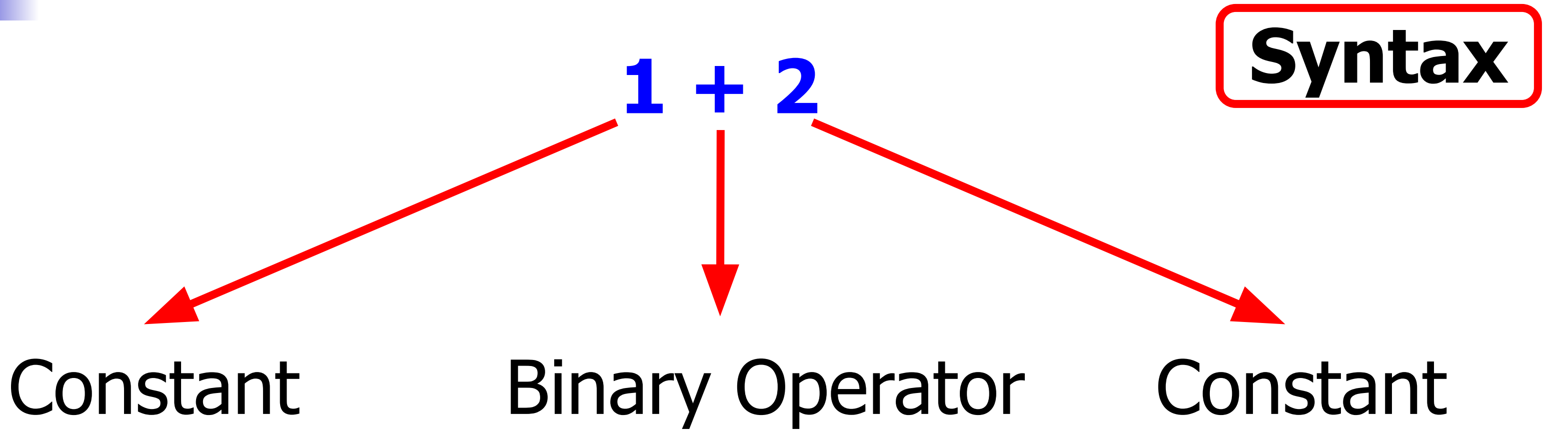
Constant

---

1 + dinosaur = 3

(Bizarre) Semantics

Meta-discourse



Syntax

1 + 2

Constant

Binary Operator

Constant

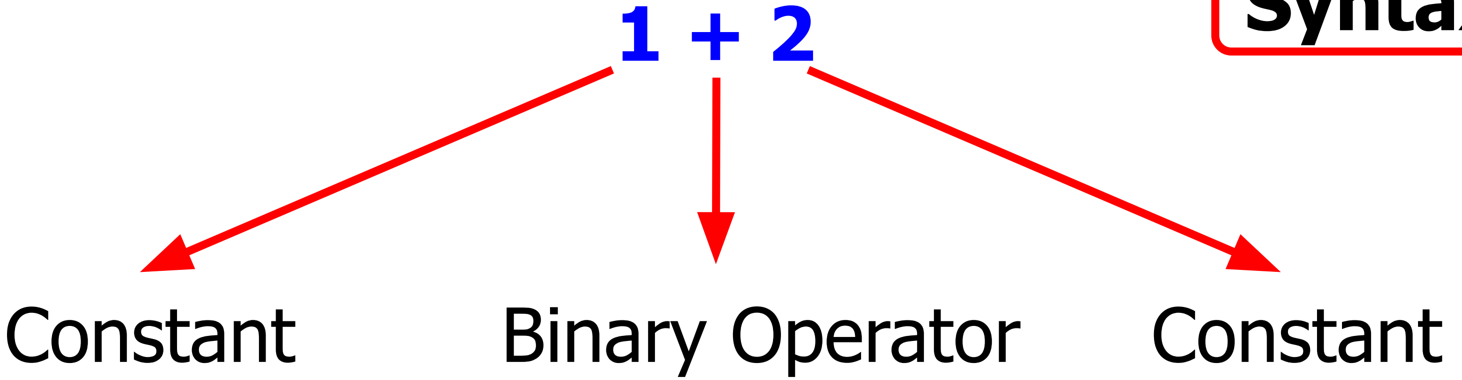
---

1 + 2 = 3

Semantics

Language Syntax

Syntax



1 + 2

Constant

Binary Operator

Constant

Language Syntax

- **Syntax** describe 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**

Questions so far?

Lexing

Lexing and Parsing

- Converting strings (representing programs) to abstract syntax trees done in **two phases**:
  - **Lexing:** Converting **strings** (or streams of characters) into lists/streams of **tokens** (the "words" of the language)
    - Specification Technique: Regular Expressions
  - **Parsing:** Convert lists/streams of **tokens** into **abstract syntax trees**
    - Specification Technique: BNF Grammars

Lexing

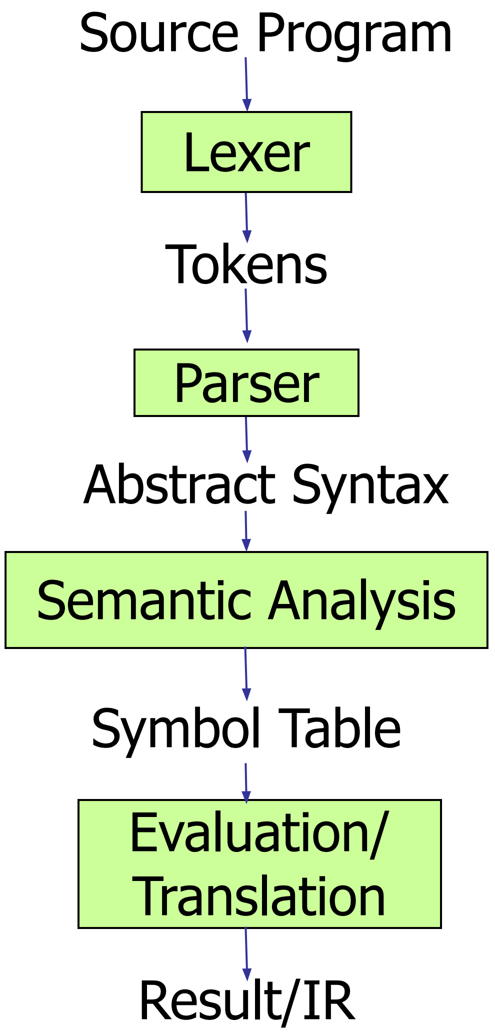
Lexing and Parsing

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

Lexing and Parsing

- Converting strings (representing programs) to abstract syntax trees done in **two phases**:
  - **Lexing:** Converting **strings** (or streams of characters) into lists/streams of **tokens** (the "words" of the language)
    - Specification Technique: Regular Expressions
  - **Parsing:** Convert lists/streams of **tokens** into **abstract syntax trees**
    - Specification Technique: BNF Grammars

Lexing and Parsing



Source Program

Lexer

Tokens

Parser

Abstract Syntax

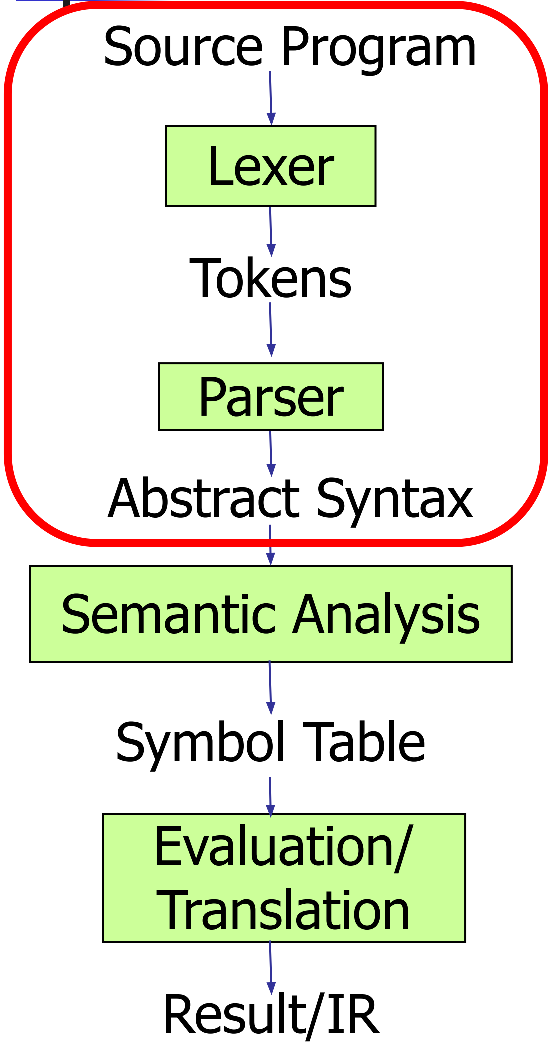
Semantic Analysis

Symbol Table

Evaluation/
Translation

Result/IR

Lexing and Parsing


Source Program

Lexer

Tokens

Parser

Abstract Syntax

Semantic Analysis

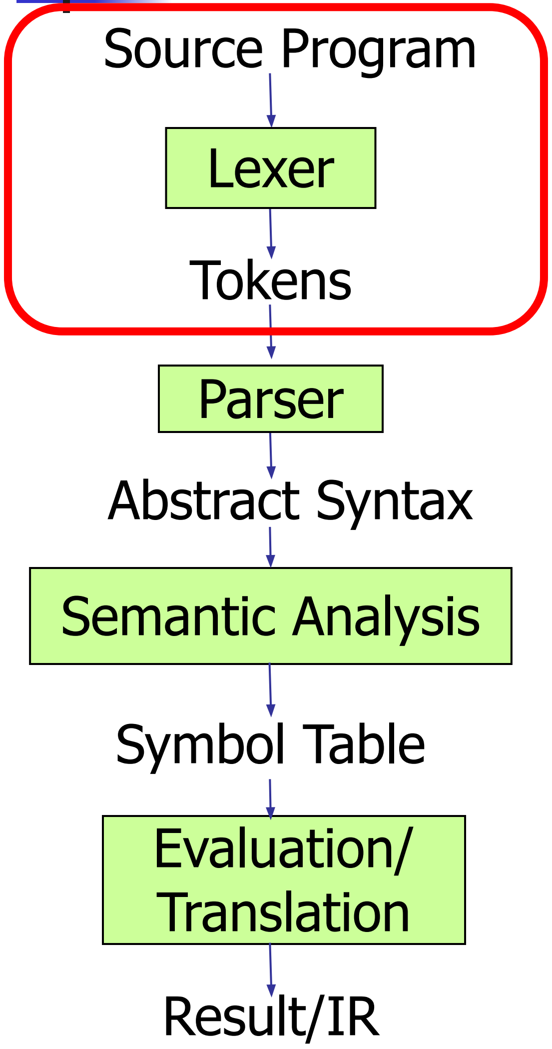
Symbol Table

Evaluation/ Translation

Result/IR

Lexing

Lexing and Parsing



Source Program

Lexer

Tokens

Parser

Abstract Syntax

Semantic Analysis

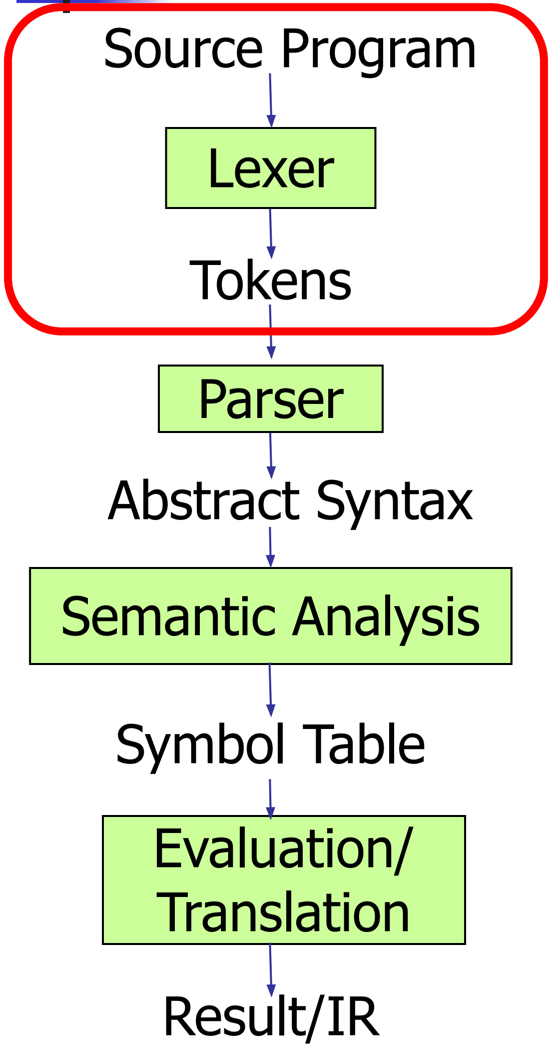
Symbol Table

Evaluation/ Translation

Result/IR

To **lex** our source program and get **tokens**, we need **regular expressions**, **automata**, and a specific kind of **grammar**.

Lexing and Parsing



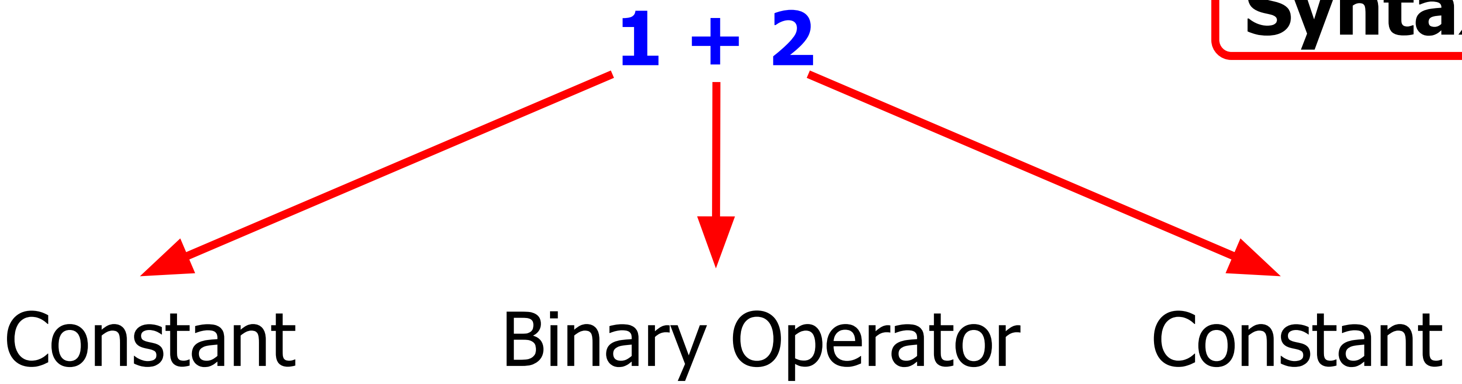
To **lex** our source program and get **tokens**, we need **regular expressions**, **automata**, and a specific kind of **grammar**.

Tokens just tell us what category each part of the input falls into.

Lexing

Lexing and Parsing

Syntax



1 + 2

Constant

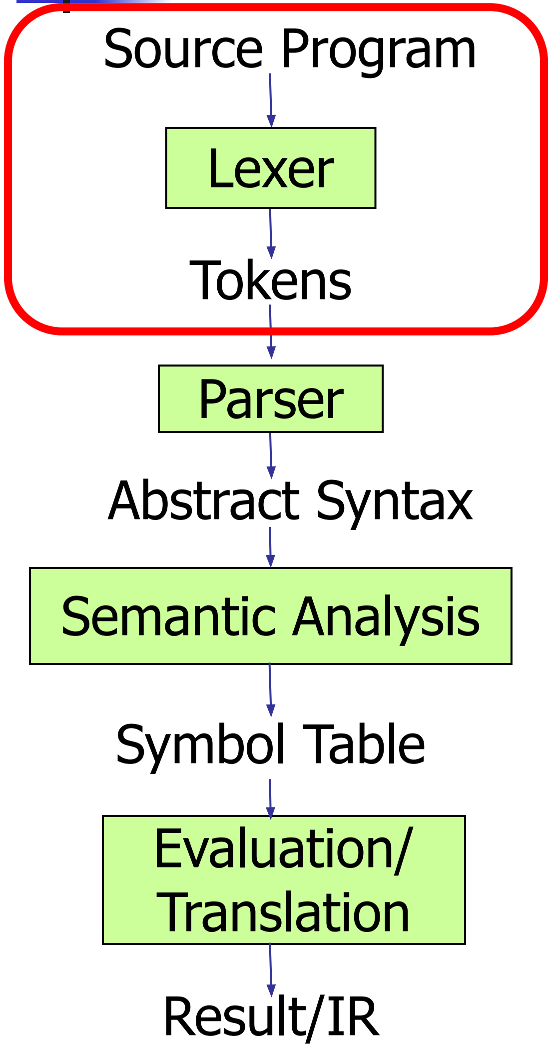
Binary Operator

Constant

Tokens just tell us what category each part of the input falls into.

Lexing

Lexing and Parsing



Source Program

Lexer

Tokens

Parser

Abstract Syntax

Semantic Analysis

Symbol Table

Evaluation/ Translation

Result/IR

To **lex** our source program and get **tokens**, we need **regular expressions**, **automata**, and a specific kind of **grammar**.

Tokens just tell us what category each part of the input falls into.

Regular Expressions

Regular Expressions - Review

- Start with given character set (**alphabet**):
  - For example, {a, b, c, … }
- **Empty Set:**
  - $L(\Phi) = \{ \}$
- **Empty String:**
  - $L(\boldsymbol{\varepsilon}) = \{\text{“”}\}$
- **Literals:** Each **character** is a regular expression
  - Represents the set of one string containing just that character
  - $L(\mathbf{a}) = \{a\}$

Regular Expressions - Review

- Start with given character set (**alphabet**):
  - For example, {a, b, c, … }
- **Empty Set:**
  - $L(\Phi) = \{ \}$
- **Empty String:**
  - $L(\boldsymbol{\varepsilon}) = \{\text{“”}\}$
- **Literals:** Each **character** is a regular expression
  - Represents the set of one string containing just that character
  - $L(\mathbf{a}) = \{a\}$

Regular Expressions - Review

- Start with given character set (**alphabet**):
  - For example, {a, b, c, … }
- **Empty Set:**
  - $L(\Phi) = \{\ \}$
- **Empty String:**
  - $L(\boldsymbol{\varepsilon}) = \{\text{""}\}$
- **Literals:** Each **character** is a regular expression
  - Represents the set of one string containing just that character
  - $L(\mathbf{a}) = \{a\}$

Regular Expressions - Review

- Start with given character set (**alphabet**):
  - For example, {a, b, c, … }
- **Empty Set:**
  - $L(\Phi) = \{\ \}$
- **Empty String:**
  - $L(\boldsymbol{\varepsilon}) = \{\text{“”}\}$
- **Literals:** Each **character** is a regular expression
  - Represents the set of one string containing just that character
  - $L(\mathbf{a}) = \{a\}$

Regular Expressions - Review

- If **x** and **y** are regular expressions, then their **concatenation xy** is a regular expression
  - Represents the set of strings made from first a string described by **x** then a string described by **y**
  - $L(xy) = L(x) \times L(y)$. e.g., if $L(x) = \{a, ab\}$ and $L(y) = \{c, d\}$, then $L(xy) = \{ac, ad, abc, abd\}$
- If **x** and **y** are regular expressions, then their **alternation x** $\vee$ **y** (sometimes **x | y**) is too
  - Represents the set of strings described by **x** or **y**
  - $L(x \vee y) = L(x) \cup L(y)$. e.g., if $L(x) = \{a, ab\}$ and $L(y) = \{c, d\}$, then $L(x \vee y) = \{a, ab, c, d\}$

Regular Expressions - Review

- If **x** and **y** are regular expressions, then their **concatenation xy** is a regular expression
  - Represents the set of strings made from first a string described by **x** then a string described by **y**
  - $L(xy) = L(x) \times L(y)$. e.g., if $L(x) = \{a, ab\}$ and $L(y) = \{c, d\}$, then $L(xy) = \{ac, ad, abc, abd\}$
- If **x** and **y** are regular expressions, then their **alternation x $\vee$ y** (sometimes **x | y**) is too
  - Represents the set of strings described by **x** or **y**
  - $L(x \vee y) = L(x) \cup L(y)$. e.g., if $L(x) = \{a, ab\}$ and $L(y) = \{c, d\}$, then $L(x \vee y) = \{a, ab, c, d\}$

Regular Expressions - Review

- **Grouping:** If **x** is a regular expression, so is **(x)**
  - Represents the same thing as **x**
- **Repeat:** If **x** is a regular expression, then so is **x***
  - It represents strings made from concatenating zero or more strings from **x**
  - $L(x^*) = L(\varepsilon) \cup L(x) \cup (L(x) \times L(x)) \cup \ldots$ e.g., if $L(x) = \{a, ab\}$, then $L(x^*) = \{\text{“”}, a, ab, aa, aab, abab, \ldots\}$

Regular Expressions - Review

- **Grouping:** If **x** is a regular expression, so is (**x**)
  - Represents the same thing as **x**
- **Repeat:** If **x** is a regular expression, then so is **x***
  - It represents strings made from concatenating zero or more strings from **x**
  - $L(x^*) = L(\varepsilon) \cup L(x) \cup (L(x) \times L(x)) \cup \ldots$ e.g., if $L(x) = \{a, ab\}$, then $L(x^*) = \{\text{“”}, a, ab, aa, aab, abab, \ldots\}$

Aside for Math Nerds

- On a given alphabet, these form a **semiring**:
  - Φ is 0 **(additive identity)**
  - ε is 1 **(multiplicative identity)**
  - x ∨ y is x + y **(addition)**
  - xy is x · y **(multiplication)**
  - If curious, try proving the semiring laws :)
- Special kind—a (star-continuous) **Kleene algebra**:
  - The Kleene star x* can be viewed as the infinite sum of powers of x (the **closure**)
  - Furthermore, we have x + x = x **(idempotence)**
  - Imposes a partial ordering and other goodies!

Aside for Math Nerds

- On a given alphabet, these form a **semiring**:
  - Φ is 0 **(additive identity)**
  - ε is 1 **(multiplicative identity)**
  - $x \vee y$ is $x + y$ **(addition)**
  - xy is $x \cdot y$ **(multiplication)**
  - If curious, try proving the semiring laws :)
- Special kind—a (star-continuous) **Kleene algebra**:
  - The Kleene star x^* can be viewed as the infinite sum of powers of x (the **closure**)
  - Furthermore, we have $x + x = x$ **(idempotence)**
  - Imposes a partial ordering and other goodies!

Example Regular Expressions

- **$(0 \vee 1)^*1$**
  - Set of all strings of **0**s and **1**s ending in **1**
  - **{1, 01, 11,…}**
- **a*b(a*)**
  - 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**

Example Regular Expressions

- **$(0 \vee 1)^*1$**
  - Set of all strings of **0**s and **1**s ending in **1**
  - **{1, 01, 11,…}**
- **$a^*b(a^*)$**
  - 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**

Example Regular Expressions

- **$(0 \vee 1)^*1$**
  - Set of all strings of **0**s and **1**s ending in **1**
  - **{1, 01, 11,…}**
- **a*b(a*)**
  - 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**

Example Regular Expressions

- **$(0 \vee 1)^*1$**
  - Set of all strings of **0**s and **1**s ending in **1**
  - **{1, 01, 11,…}**
- **a*b(a*)**
  - 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**

Questions so far?

Equivalently…

Right Regular Grammars

- Subclass of BNF (covered in detail soon)
- Only rules of form
  <nonterminal> ::= <terminal><nonterminal> or
  <nonterminal> ::= <terminal> or
  <nonterminal> ::= ε
- Defines **same class of languages** as **regular expressions**
- Important for writing **lexers** (programs that convert strings of characters into strings of tokens)
- Connection to **nondeterministic finite state automata**: nonterminals ≅ states; rule ≅ edge

Example

- Right regular grammar:

  <Balanced> ::= ε

  <Balanced> ::= 0<OneAndMore>

  <Balanced> ::= 1<ZeroAndMore>

  <OneAndMore> ::= 1<Balanced>

  <ZeroAndMore> ::= 0<Balanced>

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

Example

- Right regular grammar:

  <Balanced> ::= ε

  <Balanced> ::= 0<OneAndMore>

  <Balanced> ::= 1<ZeroAndMore>

  <OneAndMore> ::= 1<Balanced>

  <ZeroAndMore> ::= 0<Balanced>

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

Implementing Regular Expressions

- Regular expressions can be **good** for **generating** strings in language
- They are **not so good** for **recognizing** when a string is in language
- **Problems**:
  - Which option to choose?
  - How many repetitions to make?
- **Answer**: finite state automata
- Should have seen in CS374

Implementing Regular Expressions

- Regular expressions can be **good** for **generating** strings in language
- They are **not so good** for **recognizing** when a string is in language
- **Problems**:
  - Which option to choose?
  - How many repetitions to make?
- **Answer**: finite state automata
- Should have seen in CS374

Implementing Regular Expressions

- Regular expressions can be **good** for **generating** strings in language
- They are **not so good** for **recognizing** when a string is in language
- **Problems**:
    - Which option to choose?
    - How many repetitions to make?
- **Answer**: finite state automata
- Should have seen in CS374

Example: Lexing

- Regular expressions good for **describing lexemes (words)** in a programming language
  - **Identifier** = $(a \vee \ldots \vee z \vee A \vee \ldots \vee Z)$ $(a \vee \ldots \vee z \vee A \vee \ldots \vee Z \vee 0 \vee \ldots \vee 9)^*$
  - **Digit** = $(0 \vee 1 \vee \ldots \vee 9)$
  - **Number** = $0 \vee (1 \vee \ldots \vee 9)(0 \vee \ldots \vee 9)^* \vee$ $\sim (1 \vee \ldots \vee 9)(0 \vee \ldots \vee 9)^*$
  - **Keywords**: if = if, while = while,…

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]
- Could write the regular expression, then translate to DFA by hand, but this is a lot of work
- Better: Write **program** to translate automatically
- **Lex** is such a program (**ocamllex** for ocaml)

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]
- Could write the regular expression, then translate to DFA by hand, but this is a lot of work
- Better: Write **program** to translate automatically
- **Lex** is such a program (**ocamllex** for ocaml)

Lex

How to Lex

- To lex, we need:
  - A way to identify input strings (a **lexing buffer**)
  - A set of **regular expressions** to match against
  - A corresponding **set of actions** to take

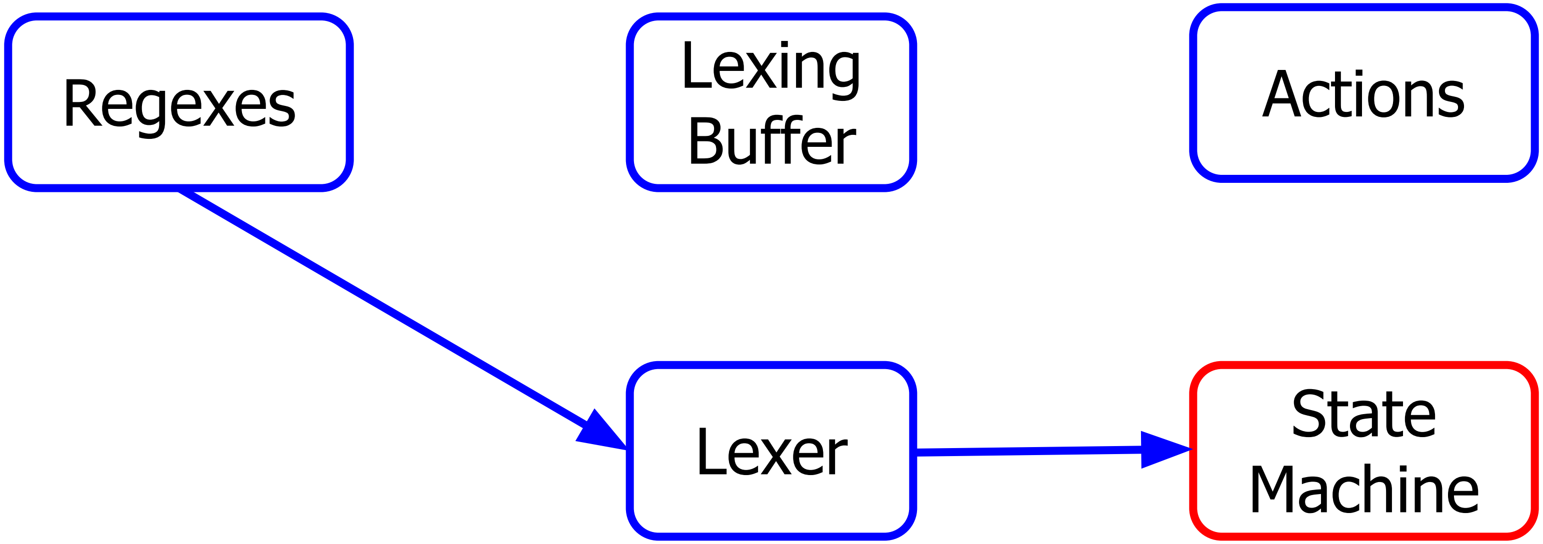
Regexes

Lexing Buffer

Actions

How to Lex

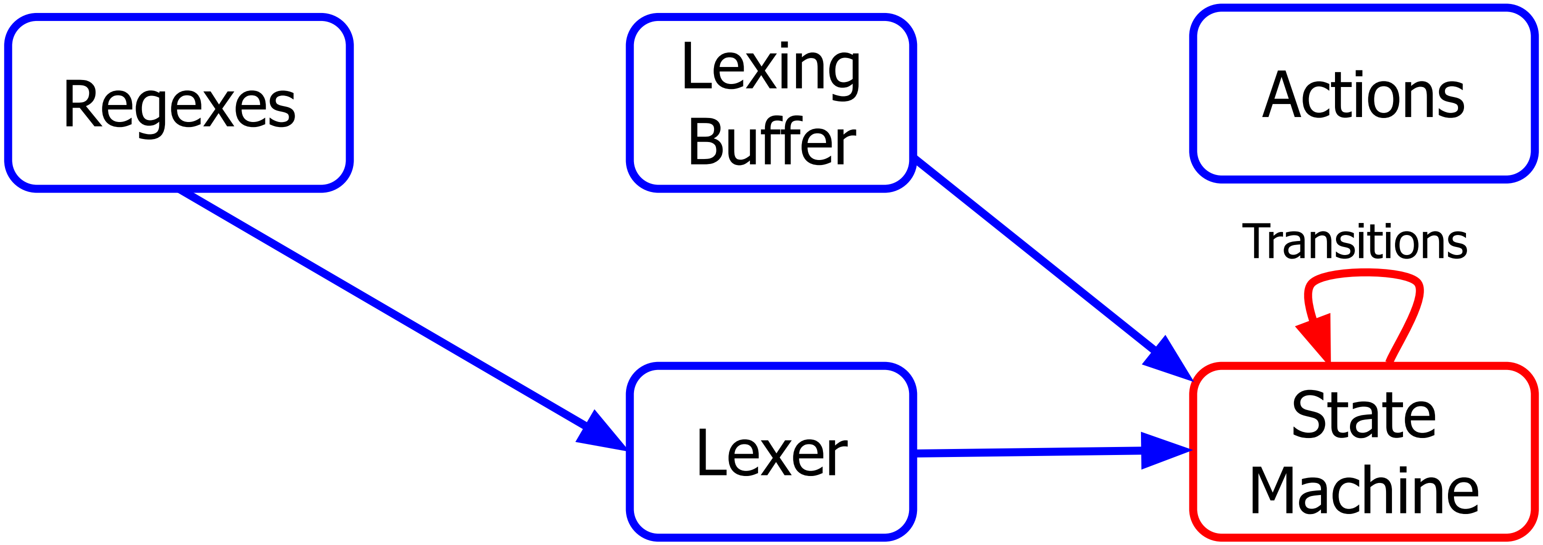
- To lex, we need:
  - A way to identify input strings (a **lexing buffer**)
  - A set of **regular expressions** to match against
  - A corresponding **set of actions** to take



Lex

How to Lex

- To lex, we need:
  - A way to identify input strings (a **lexing buffer**)
  - A set of **regular expressions** to match against
  - A corresponding **set of actions** to take



Regexes

Lexing Buffer

Actions

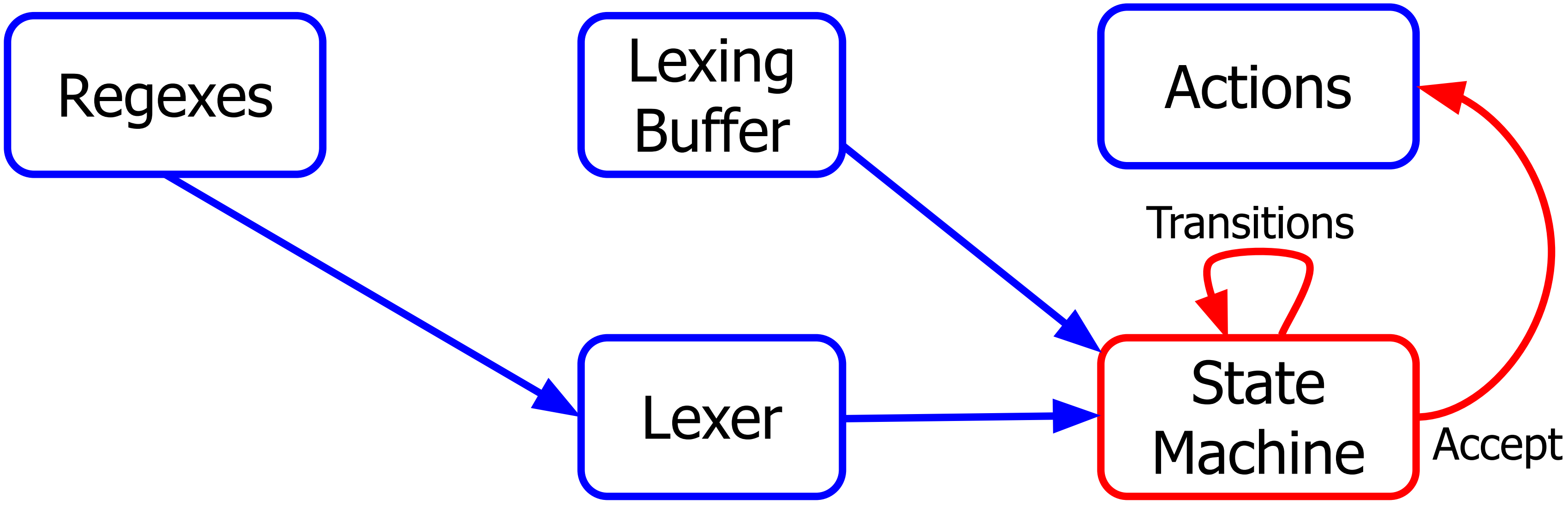
Transitions

Lexer

State Machine

How to Lex

- To lex, we need:
  - A way to identify input strings (a **lexing buffer**)
  - A set of **regular expressions** to match against
  - A corresponding **set of actions** to take



Ocamllex Mechanics

- Put **table** of **regular expressions** and corresponding **actions** (in OCaml) into a file:
  <*filename*>.mll
- **Call**:
  ocamllex <*filename*>.mll
- Produces **OCaml code** for a **lexical analyzer** in
  <*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
  main newlexbuf
}
```

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
}
```

Sample Input

```
let digits = ['0'-'9']+
let chars = ['a'-'z']+
rule main = parse
| digits { print_string "Int\n"}
| digits'.'digits { print_string "Float\n"}
| chars { print_string "String\n"}
| _ { main lexbuf }
{
  let newlexbuf = Lexing.from_channel stdin in
  main newlexbuf
}
```

General Input

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

These contain **arbitrary ocaml code** put at top and bottom of <*filename*>.ml

General Input

{ *header* }

let *ident* = *regexp* …

rule *entrypoint* [*arg1*… *argn*] = parse

| *regexp* { *action* }

| …

| *regexp* { *action* }

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

| …

and …

{ *trailer* }

This **introduces a variable** *ident* for use in later regular expressions

General Input

{ *header* }

let *ident* = *regexp* ...

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

| *regexp* { *action* }

| ...

| *regexp* { *action* }

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

| ...

and ...

{ *trailer* }

This **introduces a variable** *ident* for use in later regular expressions

General Input

{ *header* }

let *ident* = *regexp* …

rule *entrypoint* [*arg1*… *argn*] = parse

| *regexp* { *action* }

| …

| *regexp* { *action* }

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

| …

and …

{ *trailer* }

Each *entrypoint* corresponds to **one lexing function** in <*filename*>.ml

General Input

{ *header* }

let *ident* = *regexp* …

rule *entrypoint* [*arg1*… *argn*] = parse

| *regexp* { *action* }

| …

| *regexp* { *action* }

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

| …

and …

{ *trailer* }

The **name** of that lexing function is the name given for *entrypoint*

General Input

{ *header* }

let *ident* = *regexp* …

rule *entrypoint* [*arg1*… *argn*] = parse

| *regexp* { *action* }

| …

| *regexp* { *action* }

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

| …

and …

{ *trailer* }

Each becomes an OCaml function that takes *n+1* **arguments** …

General Input

{ *header* }

let *ident* = *regexp* …

rule *entrypoint* [*arg1*… *argn*] = parse

| *regexp* { *action* }

| …

| *regexp* { *action* }

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

| …

and …

{ *trailer* }

The first *n* arguments are those defined here explicitly.

General Input

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

Each argument *arg1* … *argn* is available for use in *action*

General Input

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

The extra implicit last argument has type Lexing.lexbuf

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
- $e_1 \mid e_2$: choice - what was $e_1 \vee e_2$
- $[c_1 - c_2]$: choice of any character between first and second inclusive, as determined by character codes
- $[\hat{}c_1 - c_2]$: choice of any character NOT in set

Ocamllex Regular Expression

- e*: same as before
- $e+$: same as $e\ e^*$
- $e?$: option - was $e \vee \varepsilon$
- 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:
https://v2.ocaml.org/releases/4.14/htmlman/lexyacc.html

Example : test.mll

```
{
  (* header *)
  type result = Int of int | Float of float | String of string
}
(* variables for reference in later regular expressions *)
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

```
{
  (* header *)
  type result = Int of int | Float of float | String of string
}
(* variables for reference in later regular expressions *)
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    (* entrypoint called “main” *)
 | (digits)'.'digits as f  { Float (float_of_string f) }
 | digits as n              { Int (int_of_string n) }
 | letters as s             { String s}
 | _                         { main lexbuf }
 {
   (* trailer *)
   let newlexbuf = (Lexing.from_channel stdin) in
   print_newline ();
   main newlexbuf
 }
```

Example : test.mll

```
rule main = parse     (* entrypoint called “main” *)
 | (digits)'.'digits as f  { Float (float_of_string f) }
 | digits as n              { Int (int_of_string n) }
 | letters as s             { String s}
 | _                        { main lexbuf }
 {
   (* trailer *)
   let newlexbuf = (Lexing.from_channel stdin) in
   print_newline ();
   main newlexbuf
 }
```

Example : test.mll

```
rule main = parse     (* entrypoint called “main” *)
 | (digits)'.'digits as f  { Float (float_of_string f) }
 | digits as n              { Int (int_of_string n) }
 | letters as s             { String s}
 | _                        { main lexbuf }
 {
   (* trailer *)
   let newlexbuf = (Lexing.from_channel stdin) in
   print_newline ();
   main newlexbuf
 }
```

Example : test.mll

```
rule main = parse    (* entrypoint called “main” *)
 | (digits)'.'digits as f  { Float (float_of_string f) }
 | digits as n              { Int (int_of_string n) }
 | letters as s             { String s}
 | _                        { main lexbuf }
 {
   (* trailer *)
   let newlexbuf = (Lexing.from_channel stdin) in
   print_newline ();
   main newlexbuf
 }
```

Example : test.mll

```
rule main = parse    (* entrypoint called "main" *)
 | (digits)'.'digits as f  { Float (float_of_string f) }
 | digits as n              { Int (int_of_string n) }
 | letters as s             { String s}
 | _                         { main lexbuf }
 {
   (* trailer *)
   let newlexbuf = (Lexing.from_channel stdin) in
   print_newline ();
   main newlexbuf
 }
```

Example : using generated file

```
# #use "test.ml";;
…
val main : Lexing.lexbuf -> result = <fun>
val __ocaml_lex_main_rec :
  Lexing.lexbuf -> int -> result = <fun>
hi
- : result = String "hi”
```

Example : using generated file

```
# #use "test.ml";;
…
val main : Lexing.lexbuf -> result = <fun>
val __ocaml_lex_main_rec :
  Lexing.lexbuf -> int -> result = <fun>
hi
- : result = String "hi”
```

Example : using generated file

```
# #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”
```

Example : using generated file

```
# #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 : using generated file

```
# let b = Lexing.from_channel stdin;;
# main b;;
hi 673 there
- : result = String "hi"
# main b;;
- : result = Int 673
# main b;;
- : result = String "there"
```

Recall the hidden argument of type lexbuf

Example : using generated file

```
# let b = Lexing.from_channel stdin;;
# main b;;
hi 673 there
- : result = String "hi"
# main b;;
- : result = Int 673
# main b;;
- : result = String "there"
```

Recall the hidden argument of type lexbuf

Example : using generated file

```
# let b = Lexing.from_channel stdin;;
# main b;;
hi 673 there
- : result = String "hi"
# main b;;
- : result = Int 673
# main b;;
- : result = String "there"
```

Recall the hidden argument of type lexbuf

Questions so far?

Your Turn

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

Questions?

Next Class

- **EC2 is up**
- **Quiz 4** on **MP7** is **Tuesday**
  - **Please show up!**
  - **Extra chance for ADT midterm question!**
- **WA7** due next **Thursday**
- All deadlines can be found on **course website**
- Use **office hours** and **class forums** for help