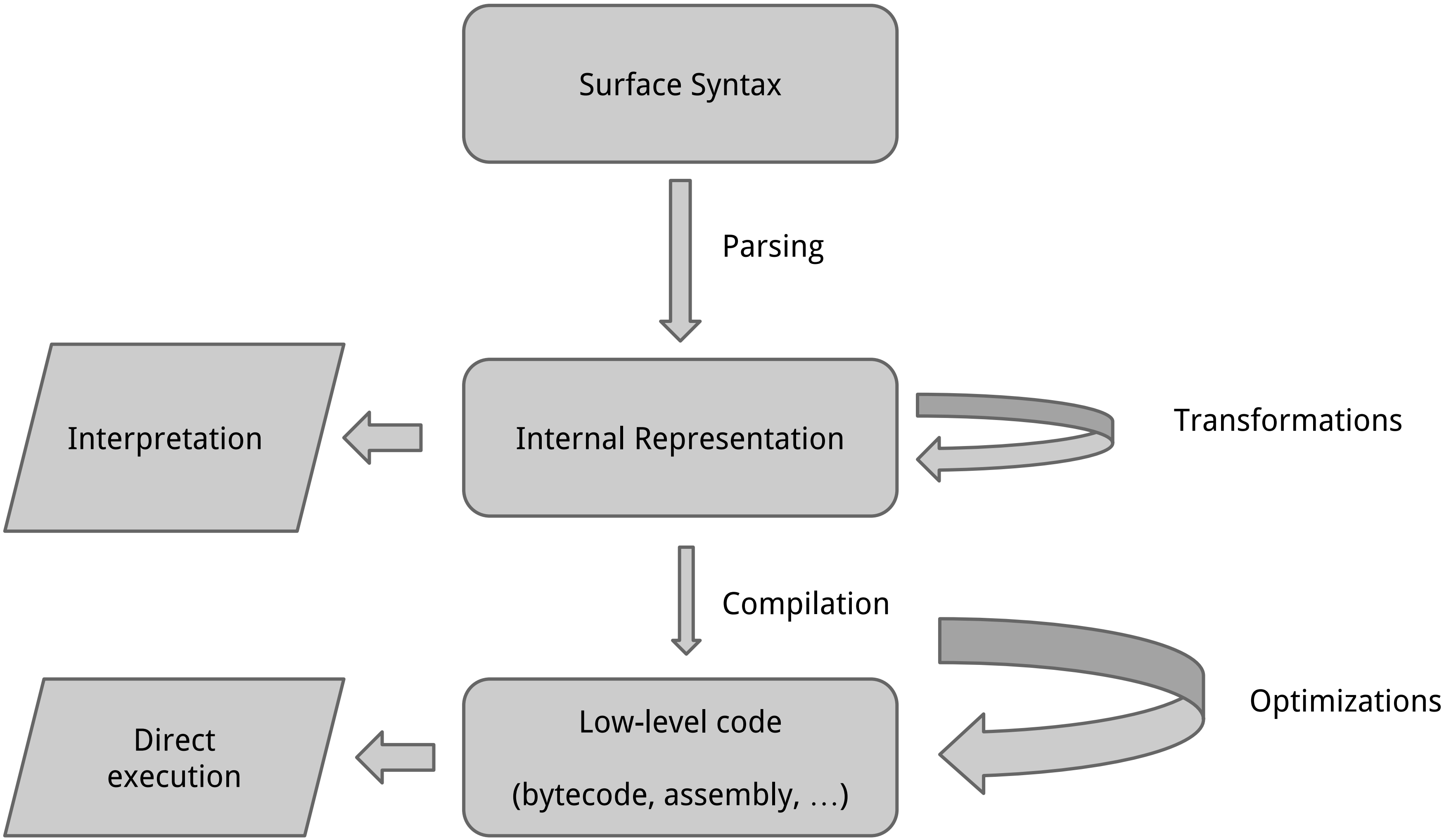
Surface Syntax:

Lexical Analysis

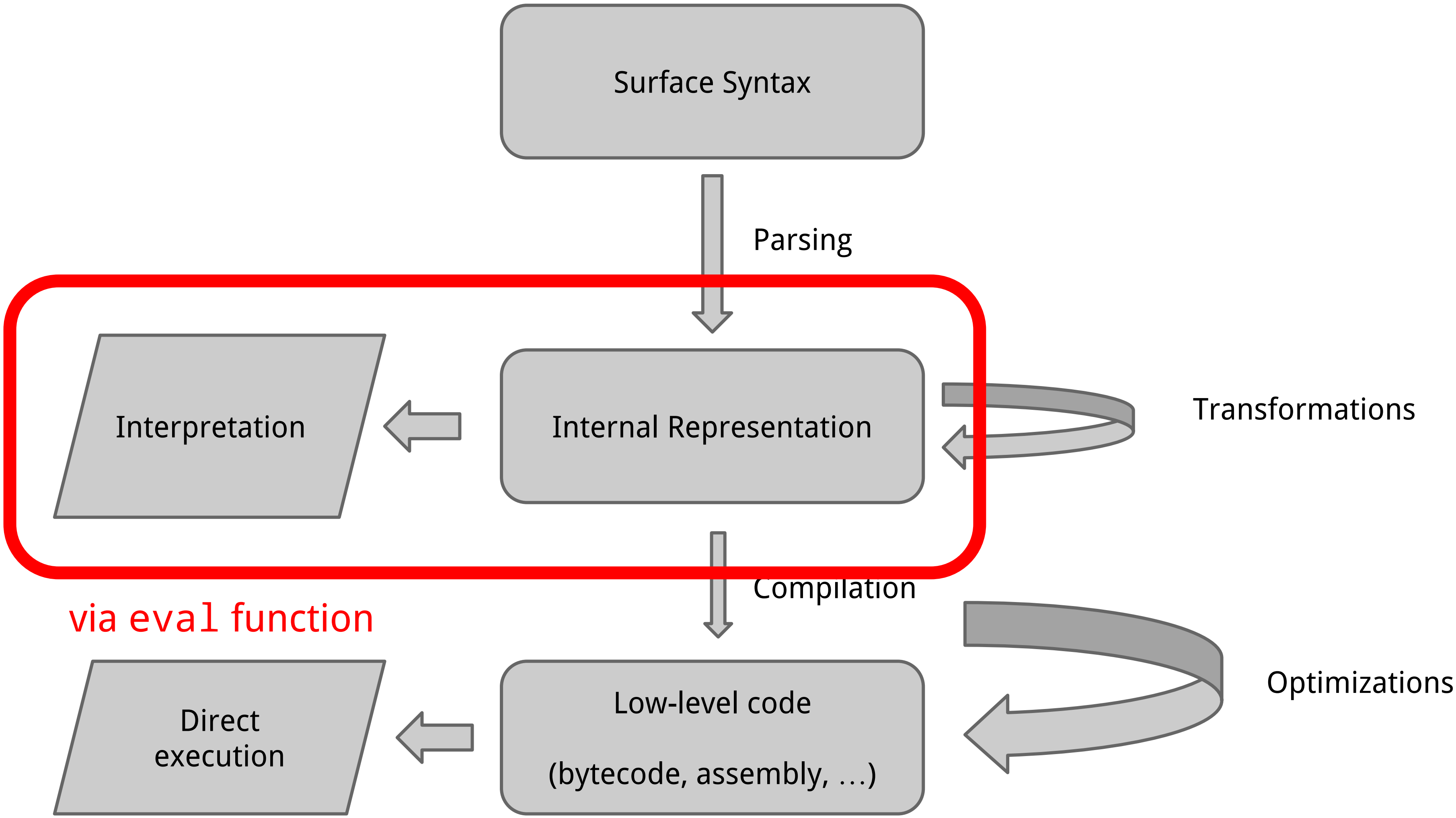
February 4, 2014

Riccardo Pucella

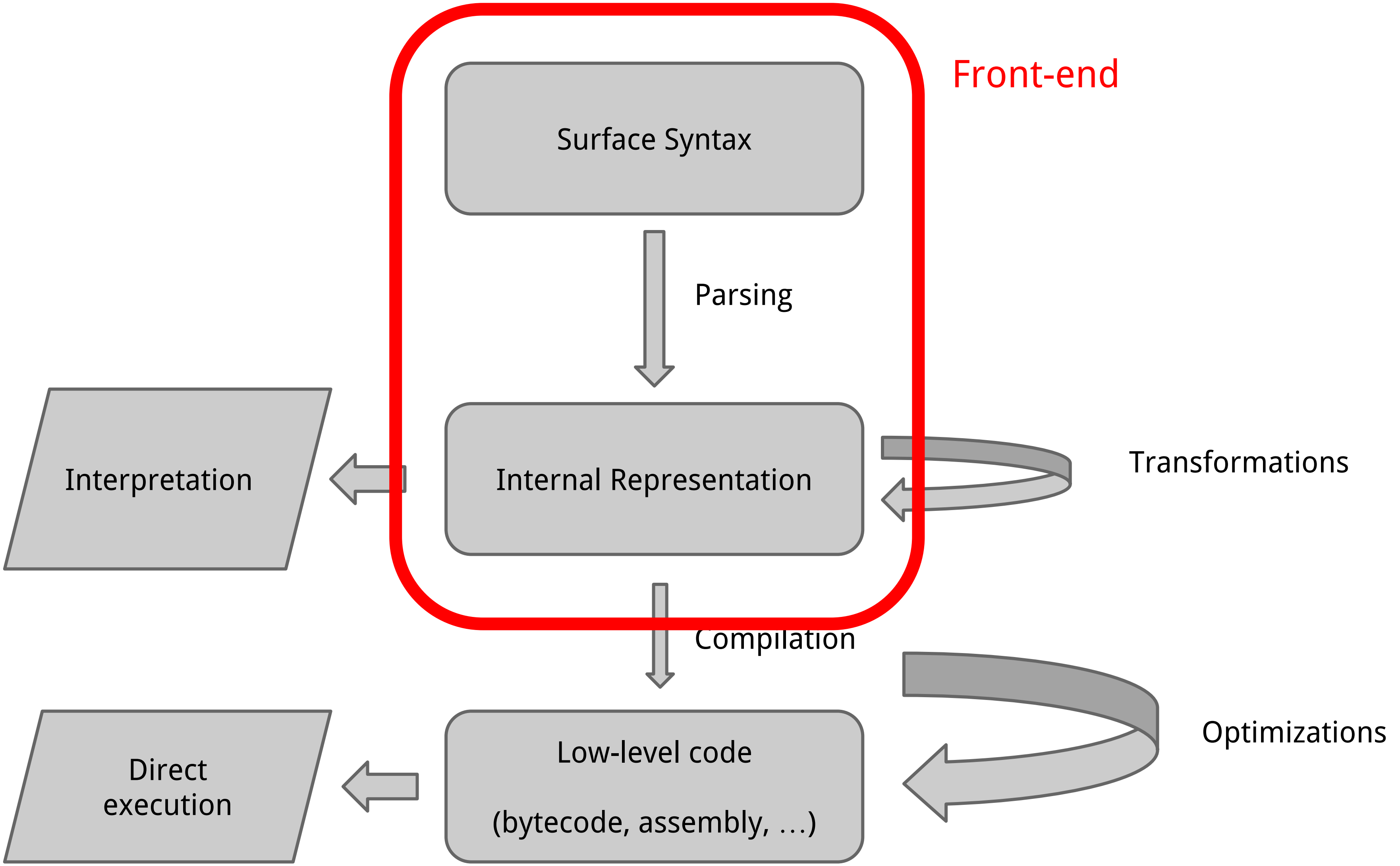
The structure of language execution



The structure of language execution



The structure of language execution



Why surface syntax?

Internal representation:

good for computers

Surface syntax:

good for humans

Why surface syntax?

Internal representation:

good for computers

Surface syntax:

good for ~~humans~~ *programmers*

Front-end

Surface syntax → internal representation

```
let x = 10 + 20
   in x * x
```

- Perform syntax checking
- Sometimes: some type checking

Front-end

Surface syntax → internal representation

```
let x = 10 + 20
   in x * x
```

→

```
ELet (“x”, EAdd (EVal (VInt 10),
                 Eval (VInt 20)),
           EMul (EIdent “x”,
                 EIdent “x”)
```

- Perform syntax checking
- Sometimes: some type checking

Front-end input

What is the input to the front-end?

- files, input from interactive shells, ...
- abstraction: sequences of characters
- key operation:
  get next character from sequence

Two phases of the front-end

Lexical analysis

sequence of characters → sequence of tokens

Parsing

sequence of tokens → internal representation

Same thing happens in natural languages

- phonemes into words into grammatical sentences

Example

```
let x = 10 + 20
 in x * x
```

Example

```
let x = 10 + 20
 in x * x
```

```
l e t ␣ x ␣ = ␣ 1 0 ␣ + ␣ 2 0 ◆ ␣ i n ␣ x ␣ * ␣ x
```

Example

```
let x = 10 + 20
 in x * x
```

```
l e t ␣ x ␣ = ␣ 1 0 ␣ + ␣ 2 0 ◆ ␣ i n ␣ x ␣ * ␣ x
```

```
SYM[let] SYM[x] EQUAL INT[10] PLUS INT[20] SYM[in] SYM[x]
   TIMES SYM[x]
```

Example

```
let x = 10 + 20
 in x * x
```

```
l e t ␣ x ␣ = ␣ 1 0 ␣ + ␣ 2 0 ◆ ␣ i n ␣ x ␣ * ␣ x
```

```
SYM[let] SYM[x] EQUAL INT[10] PLUS INT[20] SYM[in] SYM[x]
   TIMES SYM[x]
```

```
ELet (“x”, EAdd (EVal (VInt 10),
                 EVal (VInt 20)),
      EMul (EIdent “x”,
            EIdent “x”)
```

Example

```
let x = 10 + 20
 in x * x
```

```
l e t ␣ x ␣ = ␣ 1 0 ␣ + ␣ 2 0 ◆ ␣ i n ␣ x ␣ * ␣ x
```

```
KW_LET   SYM[x] EQUAL INT[10] PLUS INT[20] KW_IN   SYM[x]
   TIMES SYM[x]
```

```
ELet (“x”, EAdd (EVal (VInt 10),
                 EVal (VInt 20)),
      EMul (EIdent “x”,
            EIdent “x”)
```

Example

```
let x = 10 + 20
 in x * x
```

```
l e t ␣ x ␣ = ␣ 1 0 ␣
```

Can tokenize in many different ways — practical trade-offs

```
KW_LET   SYM[x] EQUAL INT[10] PLUS INT[20] KW_IN   SYM[x]
   TIMES SYM[x]
```

```
ELet (“x”, EAdd (EVal (VInt 10),
                 EVal (VInt 20)),
      EMul (EIdent “x”,
            EIdent “x”)
```

Example

```
let x = 10 + 20
 in x * x
```

```
l e t ␣ x ␣ = ␣ 1 0 ␣ + ␣ 2 0 ◆ ␣ i n ␣ x ␣ * ␣ x
```

```
KW_LET   SYM[x] EQUAL INT[10] PLUS INT[20] KW_IN   SYM[x]
   TIMES SYM[x]
```

OBVIOUS FACT IS OBVIOUS

tokens choice depends on the surface syntax

```
ELet (

                  EIdent “x”)
```

Tokens

Unit of meaning

- sentences are made up of words
- programs are made up of tokens

Typical tokens:

- integers, floating point numbers, identifiers
- operation symbols + - * =
- punctuation ( ) , .

characters → token : local decision

Lexical analysis

Lexer:

sequence of characters → sequence of tokens

Description of tokens: regular expressions

integer	`/[0-9]+/`
string	`/\“.*\”/`
symbol	`/[a-zA-Z][a-zA-Z0-9]*/`
keyword	`/let/` (e.g.)

Lexer:

sequen

Compact representation for families of strings

Efficient to check if a string is in the family (matching)

Description of tokens: regular expressions

integer	`/[0-9]+/`
string	`/\“.*\”/`
symbol	`/[a-zA-Z][a-zA-Z0-9]*/`
keyword	`/let/` (e.g.)

Lexer

Inputs:

- tokens (and corresponding regexps)
- sequence of characters

Output:

- sequence of tokens
- *or syntax error*

A naive lexer algorithm

```
While characters remain:
   For each token:
      Does token’s regexp match a prefix
      of the characters?
        Yes → output token
                tokenize remaining characters
```

- Works reasonably well for small programs
- Relies on regexp matching

Example: internal representation

```
datatype expr = EVal of value
              | EAdd of expr * expr
              | EMul of expr * expr
              | EIf of expr * expr * expr
              | ELet of string * expr * expr
              | EIdent of string
              | ECall of string * expr

datatype value = VInt of int
               | VBool of bool
```

Example: surface syntax

Simplest surface syntax: S-expressions (LISP)

```
expr := integer
        true
        false
        ( add expr expr )
        ( mul expr expr )
        ( if expr expr expr )
        ( let name expr expr )
        name
        ( call name expr )
```

Example: surface syntax

Simplest surface syntax: S-expressions (LISP)

expr := *integer*
true
false
(**add** *expr* *expr* **)**
(**mul** *expr* *expr* **)**
(**if** *expr* *expr* *expr* **)**
(**let** *name* *expr* *expr* **)**
name
(**call** *name* *expr* **)**

Grammar

More on this next time

Example: tokens

```
datatype token = TINT of int
               | TTRUE
               | TFALSE
               | TADD
               | TMUL
               | TIF
               | TLET
               | TSYM of string
               | TCALL
               | TLPAREN
               | TRPAREN
```

Example: tokens (and regexps)

```
datatype token = TINT of int          /[0-9]+/
               | TTRUE                /true/
               | TFALSE               /false/
               | TADD                 /add/
               | TMUL                 /mul/
               | TIF                  /if/
               | TLET                 /let/
               | TSYM of string       /[a-zA-Z][a-zA-Z0-9]*/
               | TCALL                /call/
               | TLPAREN              /\(/
               | TRPAREN              /\)/
```

Regular expressions in SML

- No built-in regular expressions
- There’s a library
- It’s incredibly painful to use
- But we can write simple wrappers

```
matchRE : string -> char list ->
                  (string * char list) option
```

- No
- The
- It's
- But

Standard way to represent an optional value:

```
datatype 'a option = NONE
                   | SOME of 'a
```

`NONE` = *no value*

`SOME t` = *value* t

```
matchRE : string -> char list ->
                  (string * char list) option
```

`matchRE regexp cs  =`

`NONE` if `cs` doesn't match `regexp`

`SOME (s,cs')` where s is the prefix that matches `regexp` and `cs'` is the list of leftover characters

```
matchRE : string -> char list ->
                 (string * char list) option
```

Example: getToken function

```
getToken : char list ->
                (token option * char list)
```

`getToken cs` returns either:

- `(NONE,cs’)`: no token found, but consumed characters (blanks, comments)
- `(SOME t,cs’)`: `t` is the recognized token and `cs’` is the list of leftover characters

Example: getToken function

```
fun getToken cs =
  (case (matchRE “( |\\n|\\t)+” cs) of
    SOME (_,cs’) => (NONE, cs’)
  | NONE =>
  (case (matchRE “[0-9]+” cs) of
    SOME (s,cs’) => (SOME (TINT (Int.fromString s)), cs’)
  | NONE =>
  (case (matchRE “let” cs)
    SOME (_,cs’) => SOME (TLET, cs’)
  | NONE =>
  (case (matchRE “if” cs)
    SOME (_,cs’) => SOME (TIF, cs’)
  | NONE =>
     . . .
```

Excuse the indentation…

Example: getToken function

```
fun getToken cs =
  (case (matchRE “( |\\n|\\t)+” cs) of
    SOME (_,cs’) => (NONE, cs’)
  | NONE =>
  (case (matchRE “[0-9]+”   ) of
    SOME (s,cs’) => (SOME (     (Int.fromString s)), cs’)
  | NONE =>
  (case (matchRE “let” cs)
    SOME (_,cs’) => SOME (T
  | NONE =>
  (case (matchRE “if” cs)
    SOME (_,cs’) => SOME (T
  | NONE =>
     . . .
```

Excuse the indentation…

Skip whitespace

Example: getToken function

```
fun getToken cs =
  (case (matchRE “( |\\n|\\t)+” cs) of
    SOME (_,cs’) => (NONE, cs’)
  | NONE =>
  (case (matchRE “[0-9]+” cs) of
    SOME (s,cs’) => (SOME (TINT (Int.fromString s)), cs’)
  | NONE =>
  (case (matchRE “let” cs)
    SOME (_,cs’) => SOME (TLET, c
  | NONE =>
  (case (matchRE “if” cs)
    SOME (_,cs’) => SOME (TIF, cs’)
  | NONE =>
      . . .
```

Excuse the indentation…

There are more elegant ways to do this

Example: `lex` function

```
fun lex [] = []
  | lex cs = let
         val (token,cs') = getToken cs
     in
         case token of
           NONE => lex cs'
         | SOME t => t::(loop cs')
     end
```

A more clever lexer algorithm

- Matching a regular expression can be done with a deterministic finite automaton (DFA)

- Lexer algorithm:
  - Compile all token regexps into single large DFA
  - Tag final states with token recognized
  - Run the DFA with character sequence
  - When you hit a final state:
    - output token
    - restart DFA with remaining characters

- Usually implemented via a tool (`lex` family)

Example: input to `ml-lex`

```
datatype token = TLET | TSYM of string | TINT of int | TTRUE | TFALSE
               | TADD | TMUL | TIF | TCALL | TLPAREN | TRPAREN
%%

%structure LangLex

%%

[\n\ \t]+    => (lex());
"("      => (TLPAREN);
")"      => (TRPAREN);
"let"    => (TLET);
"true"   => (TTRUE);
"false"  => (TFALSE);
"call"   => (TCALL);
"if"     => (TIF);
"add"    => (TADD);
"mul"    => (TMUL);
[0-9]+ => (TINT (valOf (Int.fromString yytext)));
[a-zA-Z][a-zA-Z0-9]* => (TSYM yytext);
.        => (print ("ignoring bad character "^yytext); lex());
```

Example: input to `ml-lex`

```
datatype token = TLET | TSYM of string | TINT of int | TTRUE | TFALSE
               | TADD | TMUL | TIF | TCALL | TLPAREN | TRPAREN
%%

%structure LangLex

%%

[\n\ \t]+    => (lex());
"("      => (TLPAREN);
")"      => (TRPAREN);
"let"    => (TLET);
"true"   => (TTRUE);
"false"  => (TFALSE);
"call"   => (TCALL);
"if"     => (TIF);
"add"    => (TADD);
"mul"    => (TMUL);
[0-9]+ => (TINT (valOf (Int.fromString yytext)));
[a-zA-Z][a-zA-Z0-9]* => (TSYM yytext);
.        => (print ("ignoring bad character "^yytext); lex());
```

Spits out ~ 400 lines of SML code

Example: input to `ml-lex`

```
datatype token = TLET | TSYM of string | TINT of int | TTRUE | TFALSE
               | TADD | TMUL | TIF | TCALL | TLPAREN | TRPAREN
%%

%structure LangLex

%%

[\n\ \t]+
"("
")"
"let"
"true"
"false"
"call"
"if"
"add"
"mul"
[0-9]+ =>
[a-zA-Z][
.        =
```

Spits out ~ 400 lines of SML code

Including a function

`makeLexer : (int -> string) -> (unit -> token)`

which takes a function that reads characters from an input source and returns a STATEFUL function that when invoked returns the next token from the source

Now what?

- So we have a lexer...
- Lexer: sequence of characters (from file or interactive shell) ➞ sequence of tokens
- Parsing: take sequence of tokens and construct an element of the internal representation
    - next time