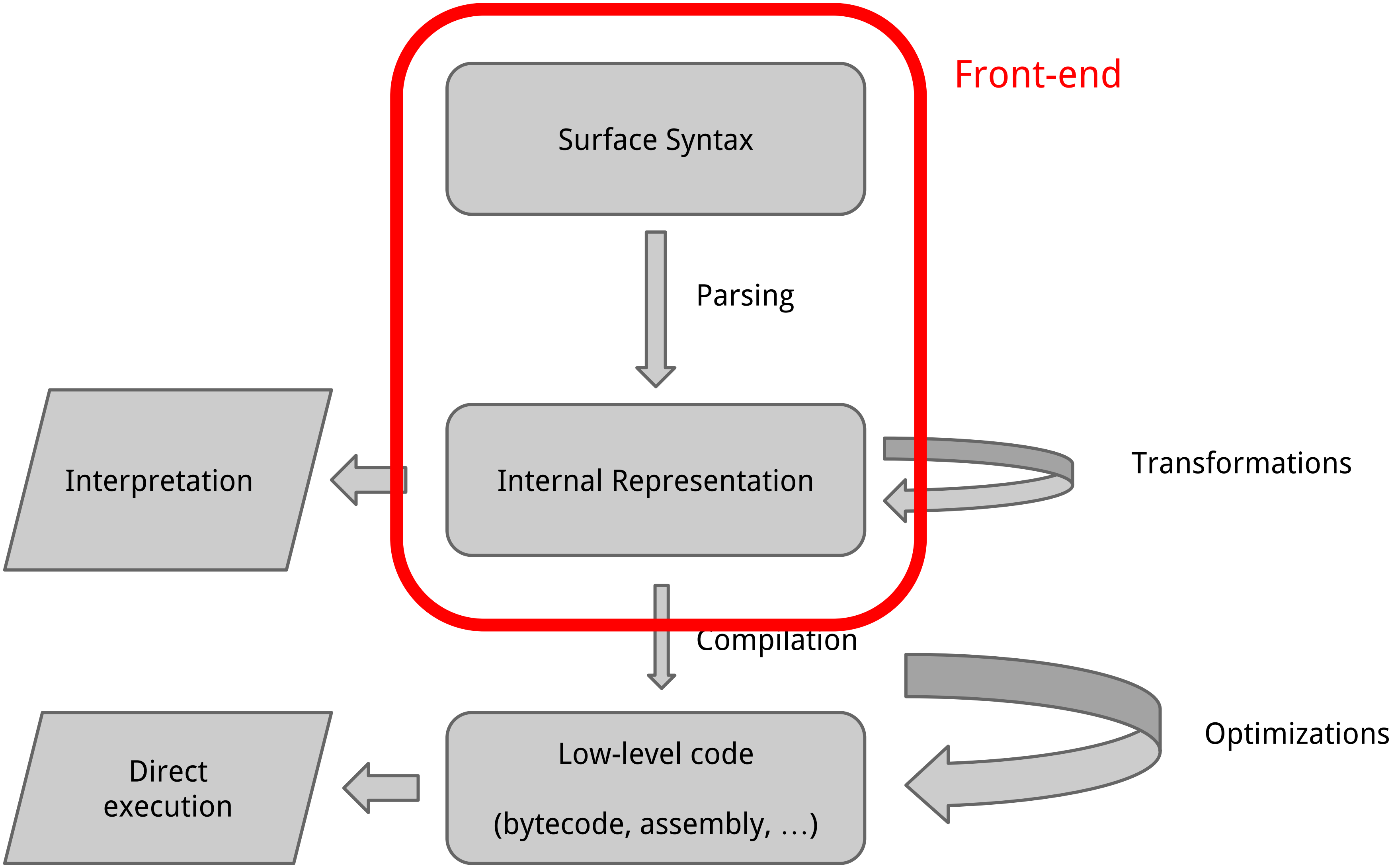
Surface Syntax:

Parsing

February 11, 2014

Riccardo Pucella

The structure of language execution



Front-end

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

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

Surface syntax

tokenization

Tokens

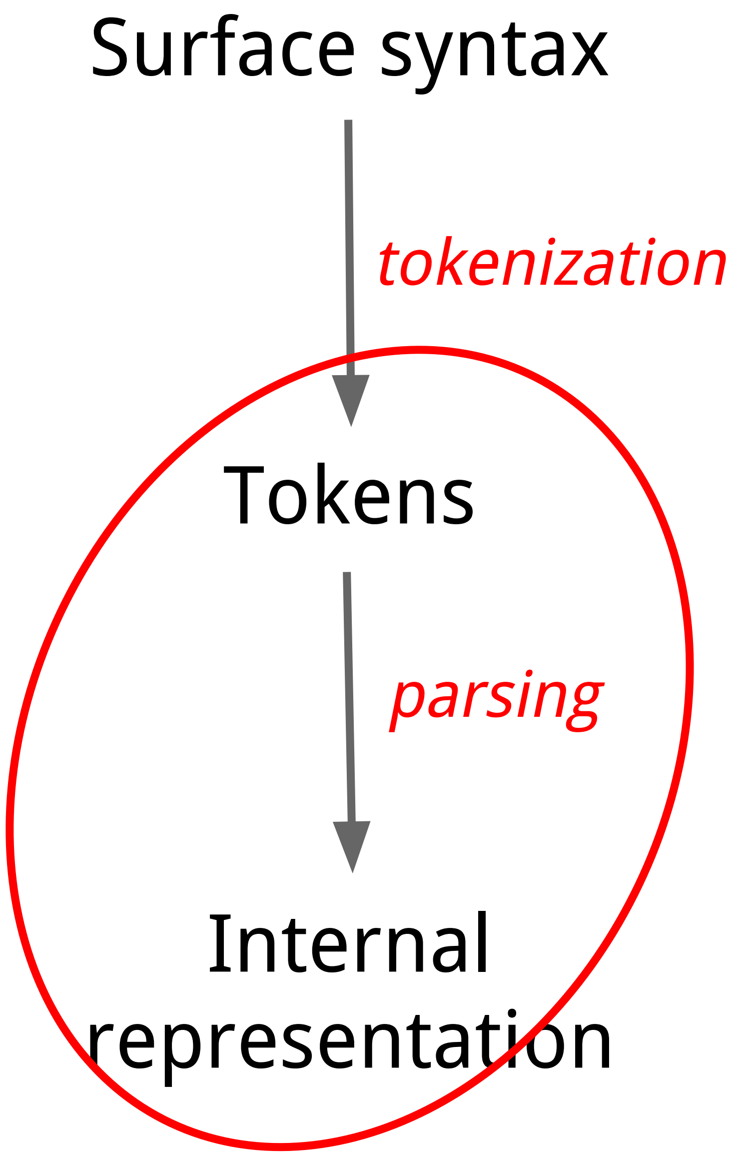
parsing

Internal representation

Front-end

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

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



Recall: Our 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 list

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

Parsing

- Identify valid token sequences
  - E.g. valid: `T_LET T_SYM T_EQUAL T_INT T_IN T_SYM`
  - E.g. not: `T_LET T_SYM T_SYM T_EQUAL T_LET …`

- Map valid token sequences to elements of the internal representation
  - E.g. `ELet (...)`

- Anything that does that is a parser

- How to describe valid token sequences?

Parsing

This is a HUGE field

A lot of work in programming languages, linguistics, natural language processing, and computational theory

This is merely to give you a taste

- Anything that does that is a parser
- How to describe valid token sequences?

Grammars

A grammar is a list of production rules for how to expand nonterminal symbols in terms of terminal symbols and other nonterminals

- terminal symbols: tokens
- nonterminal symbols: sequences of tokens

Think of English:

- A *sentence* is a *noun phrase* followed by a *verb phrase*
- A *noun phrase* is …

Example: Scheme surface syntax

```
expr ::= integer
         symbol
         true
         false
         ( + expr expr )
         ( * expr expr )
         ( let ( ( symbol expr ) ) expr )
         ( symbol expr )
```

Example: `(let ((x (+ 10 20))) (* x x))`

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

Example: Scheme surface syntax

```
expr ::= integer
         symbol
         true
         false
         ( + expr expr )
         ( * expr expr )
         ( let ( ( symbol expr ) ) expr )
         ( symbol expr )
```

nonterminals

Example: `(let ((x (+ 10 20))) (* x x))`

For: `ELet ("x", EAdd (EVal (VInt 10), Eval (VInt 20)),`
`EMul (EIdent "x", EIdent "x")`

Example: Scheme surface syntax

expr ::= ***integer***
symbol
true
false
(**+** *expr* *expr* **)**
(***** *expr* *expr* **)**
(**let** **(** **(** ***symbol*** *expr* **)** **)** *expr* **)**
(***symbol*** *expr* **)**

terminals

Example: `(let ((x (+ 10 20))) (* x x))`

For: `ELet ("x", EAdd (EVal (VInt 10), Eval (VInt 20)),`
`EMul (EIdent "x", EIdent "x")`

Example: Scheme surface syntax

```
expr ::= T_INT
         T_SYM
         T_TRUE
        T_FALSE
         T_LPAREN T_PLUS expr expr T_RPAREN
         T_LPAREN T_TIMES expr expr T_RPAREN
         T_LPAREN T_LET T_LPAREN T_LPAREN
            T_SYM expr T_RPAREN T_RPAREN expr T_RPAREN
         T_LPAREN T_SYM expr T_RPAREN
```

Really should be expressed in terms of tokens

For clarity, I will not do so in these slides

where:

```
datatype token = T_INT of int | T_PLUS | T_TIMES | T_LET
               | T_SYM of string | T_LPAREN | T_RPAREN
```

Two approaches to parsing

TOP-DOWN

- recursive descent
- coded by hand
- flexible
- can be slow

- good for simple grammars

BOTTOM-UP

- table-based
- generated by tools (`yacc`, `bison`, `antlr`)
- fast

- production systems

Recursive-descent parsers

- For every terminal T:
  - define a function *expect_T* that can match T
- For every nonterminal NT:
  - define a function *parse_NT* that can match a sequence of tokens via a rule for NT

Predictive parsers

A class of simple recursive-descent parsers

- Applicable when a token uniquely identifies which rule to choose for each nonterminal
    - can generalize to *k* tokens
- This can be tricky to recognize

A grammar with that property is called LL (1)

(Generalizes to LL(*k*) — there are precise definitions, which I'll skip)

Example: Scheme surface syntax

```
expr ::= integer
         symbol
         ( let ( ( symbol expr ) ) expr )
```

Example: a predictive parser

```
expect_INT :    token list -> token list
expect_SYM :    token list -> token list
expect_LET :    token list -> token list
expect_LPAREN : token list -> token list
expect_RPAREN : token list -> token list

parse_expr :    token list -> token list
```

Example: a predictive parser

```
fun expect_INT ((T_INT _)::ts) = ts
  | expect_INT _ = perror “expect_INT”

fun expect_SYM ((T_SYM _)::ts) = ts
  | expect_SYM _ = perror “expect_SYM”

fun expect_LET (T_LET::ts) = ts
  | expect_LET _ = perror “expect_LET”

...
```

Example: a predictive parser

```
fun parse_expr ((T_INT _)::ts) = ts
  | parse_expr ((T_SYM _)::ts) = ts
  | parse_expr (T_LPAREN::ts) = let
      val ts = expect_LET ts
      val ts = expect_LPAREN ts
      val ts = expect_LPAREN ts
      val ts = expect_SYM ts
      val ts = parse_expr ts
      val ts = expect_RPAREN ts
      val ts = expect_RPAREN ts
      val ts = parse_expr ts
      val ts = expect_RPAREN ts
    in  ts  end
  | parse_expr _ = perror “parse_expr”
```

Creating parse results

```
expr ::= integer
         symbol
         ( let ( ( symbol expr ) ) expr )
```

Creating parse results

```
expr ::= integer_i ──→ EVal (VInt i)
         symbol
         ( let ( ( symbol expr ) ) expr )
```

Creating parse results

$$\begin{aligned} expr ::= \; & \textbf{integer}_{i} \longrightarrow \texttt{EVal (VInt } i\texttt{)} \\ & \textbf{symbol}_{s} \longrightarrow \texttt{EIdent } s \\ & \textbf{(let ((} \textbf{symbol} \; expr \textbf{)) } expr \textbf{)} \end{aligned}$$

Creating parse results

$expr$ ::= $\textbf{integer}_i \longrightarrow$ `EVal (VInt` i`)`

$\textbf{symbol}_s \longrightarrow$ `EIdent` s

(let (($\textbf{symbol}_s$ $expr_{e1}$ **))** $expr_{e2}$**)** $\longrightarrow$ `ELet (`s`,`$e1$`,`$e2$`)`

Creating parse results

expr ::= ***integer***$_i$ → EVal (VInt *i*)

symbol$_s$ → EIdent *s*

(let ((***symbol***$_s$ *expr*$_{e1}$ **))** *expr*$_{e2}$**)**

→ ELet (*s*,*e1*,*e2*)

These sort of grammars are often called *attribute grammars*

Creating parse results

```
expect_INT :    token list -> token list
expect_SYM :    token list -> token list
expect_LPAREN : token list -> token list
expect_RPAREN : token list -> token list
expect_LET :    token list -> token list

parse_expr :    token list -> token list
```

Creating parse results

```
expect_INT :    token list -> (int * token list)
expect_SYM :    token list -> (string * token list)
expect_LPAREN : token list -> token list
expect_RPAREN : token list -> token list
expect_LET :    token list -> token list

parse_expr :    token list -> (expr * token list)
```

Creating parse results

```
fun expect_INT ((T_INT _)::ts) = ts
  | expect_INT _ = perror “expect_INT”

fun expect_SYM ((T_SYM _)::ts) = ts
  | expect_SYM _ = perror “expect_SYM”

fun expect_LET (T_LET::ts) = ts
  | expect_LET _ = perror “expect_LET”

...
```

Creating parse results

```
fun expect_INT ((T_INT i)::ts) = (i, ts)
  | expect_INT _ = perror "expect_INT"

fun expect_SYM ((T_SYM s)::ts) = (s, ts)
  | expect_SYM _ = perror "expect_SYM"

fun expect_LET (T_LET::ts) = ts
  | expect_LET _ = perror "expect_LET"

...
```

Creating parse results

```
fun parse_expr ((T_INT _)::ts) = ts
  | parse_expr ((T_SYM _)::ts) = ts
  | parse_expr (T_LPAREN::ts) = let
      val ts = expect_LET ts
      val ts = expect_LPAREN ts
      val ts = expect_LPAREN ts
      val ts = expect_SYM ts
      val ts = parse_expr ts
      val ts = expect_RPAREN ts
      val ts = expect_RPAREN ts
      val ts = parse_expr ts
      val ts = expect_RPAREN ts
    in  ts  end
  | parse_expr _ = perror “parse_expr”
```

Creating parse results

```
fun parse_expr ((T_INT i)::ts) = (EVal (VInt i), ts)
  | parse_expr ((T_SYM s)::ts) = (EIdent s, ts)
  | parse_expr (T_LPAREN::ts) = let
      val ts = expect_LET ts
      val ts = expect_LPAREN ts
      val ts = expect_LPAREN ts
      val (s,ts) = expect_SYM ts
      val (e1,ts) = parse_expr ts
      val ts = expect_RPAREN ts
      val ts = expect_RPAREN ts
      val (e2,ts) = parse_expr ts
      val ts = expect_RPAREN ts
    in  (ELet (s,e1,e2), ts)  end
  | parse_expr _ = perror “parse_expr”
```

Example: Scheme surface syntax

```
expr ::= integer
         symbol
         ( let ( ( symbol expr ) ) expr )
```

Example: Scheme surface syntax

```
expr ::= integer
         symbol
         ( let ( ( symbol expr ) ) expr )
        ( + expr expr )
        ( * expr expr )
        ( symbol expr )
```

Example: Scheme surface syntax

```
expr ::= integer
         symbol
         ( let ( ( symbol expr ) ) expr )
        ( + expr expr )
        ( * expr expr )
        ( symbol expr )
```

Token does not uniquely determine the rule to apply

Left factorization

Sometimes we can transform a grammar into one that can be parsed by a predictive parser.

Left factorization:

- a kind of distributive law for grammars
- rules that start with common tokens...
  - can be replaced by a single rule with those tokens...
  - and a new nonterminal for the different leftovers

Example: Scheme surface syntax

```
expr ::= integer
         symbol
         ( let ( ( symbol expr ) ) expr )
        ( + expr expr )
        ( * expr expr )
        ( symbol expr )
```

Example: Scheme surface syntax

```
expr ::= integer
         symbol
         ( expr_seq

expr_seq ::= let ( ( symbol expr ) ) expr )
             + expr expr )
             * expr expr )
             symbol expr )
```

Updated parser

```
expect_INT :      token list -> (int * token list)
expect_SYM :      token list -> (string * token list)
expect_LPAREN :   token list -> token list
expect_RPAREN :   token list -> token list
expect_LET :      token list -> token list

parse_expr :      token list -> (expr * token list)
parse_expr_seq :  token list -> (expr * token list)
```

Updated parser

```
fun parse_expr ((T_INT i)::ts) = (EVal (VInt i), ts)
  | parse_expr ((T_SYM s)::ts) = (EIdent s, ts)
  | parse_expr (T_LPAREN::ts) = parse_expr_seq ts
  | parse_expr _ = perror “parse_expr”
```

Updated parser

```
fun parse_expr ((T_INT i)::ts) = (EVal (VInt i), ts)
  ...

and parse_expr_seq (T_LET::ts) = let
      val ts = expect_LPAREN ts
      val ts = expect_LPAREN ts
      val (s,ts) = expect_SYM ts
      val (e1,ts) = parse_expr ts
      val ts = expect_RPAREN ts
      val ts = expect_RPAREN ts
      val (e2,ts) = parse_expr ts
      val ts = expect_RPAREN ts
    in  (ELet (s,e1,e2), ts)  end
  | parse_expr_seq (T_PLUS::ts) = let
      val (e1,ts) = parse_expr ts
      val (e2,ts) = parse_expr ts
      val ts = expect_RPAREN ts
    in  (EAdd (e1,e2), ts)   end
  | ...
```

Parsing with backtracking

Some grammars cannot be parsed with a predictive parser

General recursive-descent parser

Attempt to parse:

- if it succeeds, done
- if it fails, try to parse a different way
    - backtracking

Example: ML-like surface syntax

```
expr ::= integer
         symbol
         ( expr )
        let symbol = expr in expr
        symbol ( expr )
        expr + expr
        expr * expr
```

Example: ML-like surface syntax

```
expr ::= integer
         symbol
         ( expr )
        let symbol = expr in expr
        symbol ( expr )
        expr + expr
        expr * expr
```

Grammar with *left recursion*

Bad for recursive-descent parsers — WHY?

Examp

expr

Can eliminate left recursion by rewriting the grammar:

```
expr ::= integer
         expr + expr
         expr * expr
```

```
expr ::= integer
         integer expr_rest

expr_rest ::= + expr
              * expr
```

The full algorithm is a bit tricky.

Grammar with *left recursion*

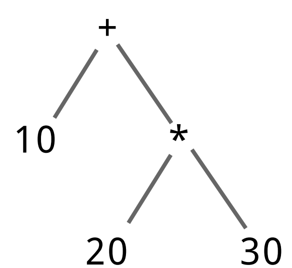
Bad for recursive-descent parsers — WHY?

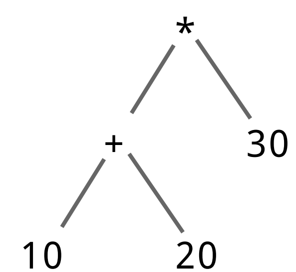
Ambiguities

Thankfully, there's another problem with the grammar: ambiguities

A grammar is ambiguous if some sequences of tokens can parse in more than one way

10 + 20 * 30





Ambiguities

Thankf ... he
gram

Another classic ambiguity

```
expr ::= ...
         if expr then expr else expr
         if expr then expr
```

Consider `if a then if b then c else d`

`if a then (if b then c) else d` ?

`if a then (if b then c else d)` ?

A gra ... of
token

10 +

Ambiguities

Than he
gra

A g of
tok

1(

Ambiguities are *nasty*

There is no generic way of dealing with them

You have to understand your grammar well, and know which of the possible parses is the one you want

For a recursive-descent parser: modify the grammar so that the parse you want is the only one found — or the first one found

Example: ML-like surface syntax

```
expr ::= integer
         symbol
         ( expr )
        let symbol = expr in expr
        symbol ( expr )
        expr + expr
        expr * expr
```

Example: ML-like surface syntax

```
  expr ::= let symbol = expr in expr
           symbol ( expr )
           term + term
            term

  term ::= factor * factor
           factor

factor ::= integer
           symbol
           ( expr )
```

No left recursion anymore (accident, but nice accident)

Example: ML-like surface syntax

```
expect_INT :      token list -> (int * token list) option
expect_SYM :      token list -> (string * token list)
option
expect_LPAREN :   token list -> (token list) option
expect_RPAREN :   token list -> (token list) option
expect_LET :      token list -> (token list) option
expect_IN :       token list -> (token list) option
expect_PLUS :     token list -> (token list) option
expect_TIMES :    token list -> (token list) option


parse_expr :      token list -> (expr * token list) option
parse_term :      token list -> (expr * token list) option
parse_factor :    token list -> (expr * token list) option
```

Example: ML-like surface syntax

```
fun expect_INT ((T_INT i)::ts) = SOME (i,ts)
  | expect_INT _ = NONE

fun expect_SYM ((T_SYM s)::ts) = SOME (s,ts)
  | expect_SYM _ = NONE

fun expect_LET (T_LET::ts) = SOME ts
  | expect_LET _ = NONE

...
```

Example: ML-like surface syntax

`parse_expr` tries to parse an *expr*

```
fun parse_expr ts =
  (case parse_expr_1 ts
     of NONE =>
        (case parse_expr_2 ts
           of NONE =>
              (case parse_expr_3 ts
                 of NONE => parse_expr_4 ts
                  | s => s)
            | s => s)
       | s => s)
```

Example: ML-like surface syntax

`parse_expr` tries to parse an *expr*

```
fun parse_expr ts =
  (case parse_expr_1 ts
     of NONE =>
        (case parse_expr_2 ts
           of NONE =>
              (case parse_expr_3 ts
                 of NONE => parse_expr_4 ts
                  | s => s)
            | s => s)
```

`parse_expr_1`	tries to parse	**let** *symbol* **=** *expr* **in** *expr*
`parse_expr_2`	tries to parse	*symbol* **(** *expr* **)**
`parse_expr_3`	tries to parse	*term* **+** *term*
`parse_expr_4`	tries to parse	*term*

Example: ML-like surface syntax

`parse_expr_3` tries to parse *`term`* **+** *`term`*

```
fun parse_expr_3 ts =
  (case parse_term ts
     of NONE => NONE
      | SOME (e1,ts) =>
        (case expect_PLUS
           of NONE => NONE
            | SOME ts =>
              (case parse_term ts
                 of NONE => NONE
                  | SOME (e2,ts) =>
                           SOME (EAdd (e1,e2),ts))))
```

Example: ML-like surface syntax

parse_expr_3 tries to parse *term* **+** *term*

```
fun parse_expr_3 ts =
  (case parse_term ts
     of NONE => NONE
      | SOME (e1 t
         (case e
            of N
             | S
               (
```

This is somewhat mind-numbing

(Then again, parsing is mind-numbing)

Alternatives:

(1) can generate recursive-descent parsers

(2) can use parser combinators

Bottom-up parsing

- Top-down: *I expect an expr, do I have the tokens to make one?*

Bottom-up parsing

- Bottom-up: *I have an integer followed by a T_PLUS followed by an integer, what does that make?*

Try to match sequences of tokens with rules

- done via a PDA-like state machine
- if you think recursive-descent parsers are mind-numbing…
    - state machines are worse
    - usually handled via a tool (`yacc`,`bison`,`antlr`)