Administrivia

Assignment 6 is due on Friday by 11:59PM.

A gentle reminder that the (W)ithdraw deadline is March 29.

Parsing I: An Introduction

Principles of Programming Languages
Lecture 14

`CAS CS 320`

Objectives

Get a sense of what parsing is, starting with ***lexical analysis***.

Look briefly at the ***general parsing*** problem.

Look at ***recursive-decent*** as a first attempt at a simple parsing procedure.

Keywords

parser generator

lexical analysis

lexeme

token

parsing

recursive-decent

Errata

```
<S> ::= <A><B>
<A> ::= a
<B> ::= b
```

```
<S>           <S>
<A><B>        <A><B>
a<B>          <A>b
ab            ab
```

This grammar is not amgbiguous.

A grammar is ambiguous if it has a sentence with multiple **parse trees.**

A sentence may have multiple derivations just by virtue of the order in which you expand nonterminal symbols.

Practice Problem

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= x | ( <expr> )
```

Write an ADT which represents (parse trees of) sentences in the above grammar.

What do the sentences of this grammar represent?

Recap + Motivation

Recall: BNF Grammars

```
<expr>  ::= <op1> <expr>
          | <op2> <expr> <expr>
          | <var>
<op1>   ::= not
<op2>   ::= and | or
<var>   ::= x | y | z
```

Recall: BNF Grammars

```
<expr>  ::= <op1> <expr>
          | <op2> <expr> <expr>
          | <var>
<op1>   ::= not
<op2>   ::= and | or
<var>   ::= x | y | z
```

tokens (terminal symbols)

Recall: BNF Grammars

```
<expr>    ::= <op1> <expr>
            | <op2> <expr> <expr>
            | <var>
<op1>     ::= not
<op2>     ::= and | or
<var>     ::= x | y | z
```

abstractions (non-terminal symbols)

tokens (terminal symbols)

Recall: BNF Grammars

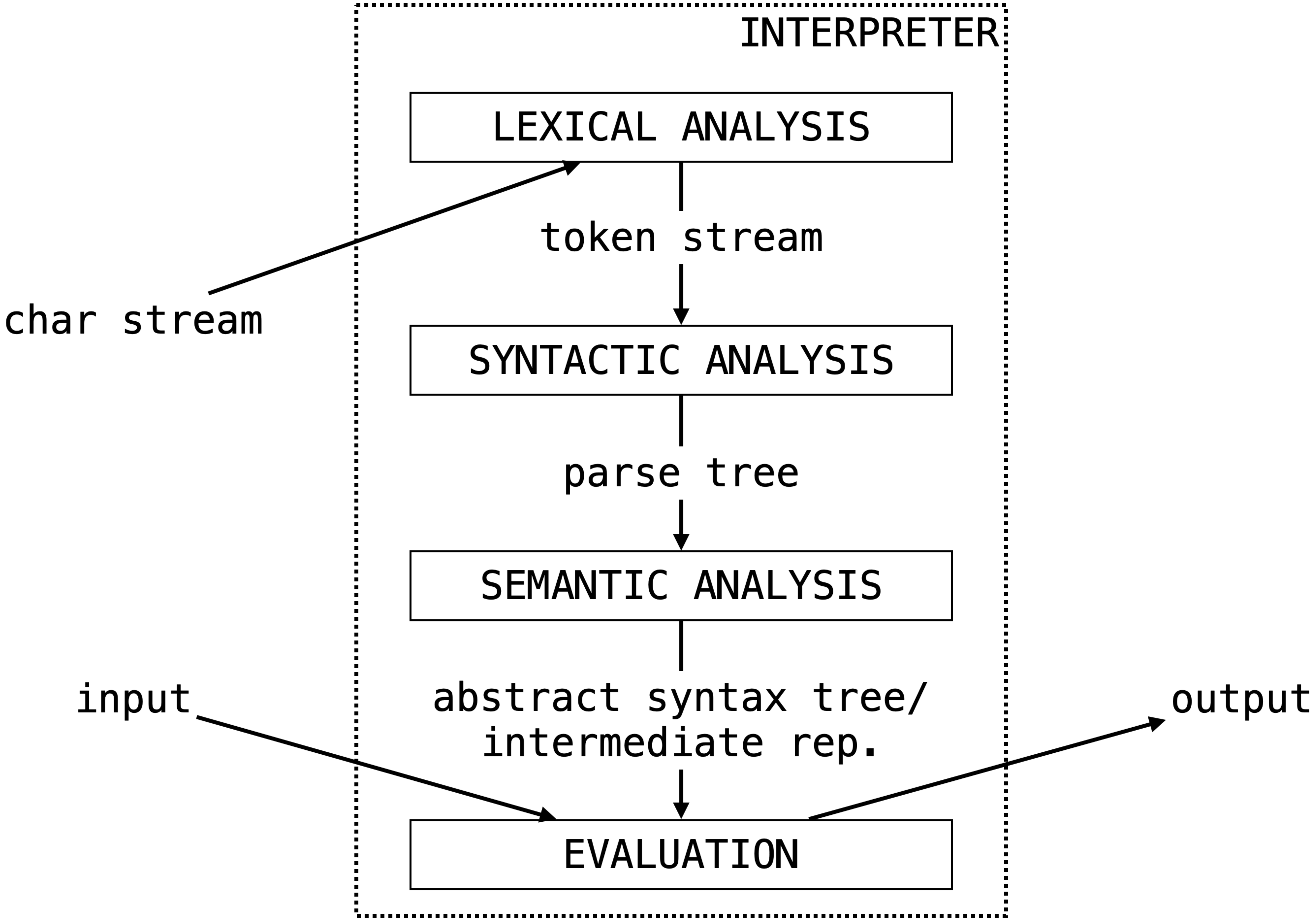
production rules

```
<expr>   ::= <op1> <expr>
           | <op2> <expr> <expr>
           | <var>
<op1>    ::= not
<op2>    ::= and | or
<var>    ::= x | y | z
```

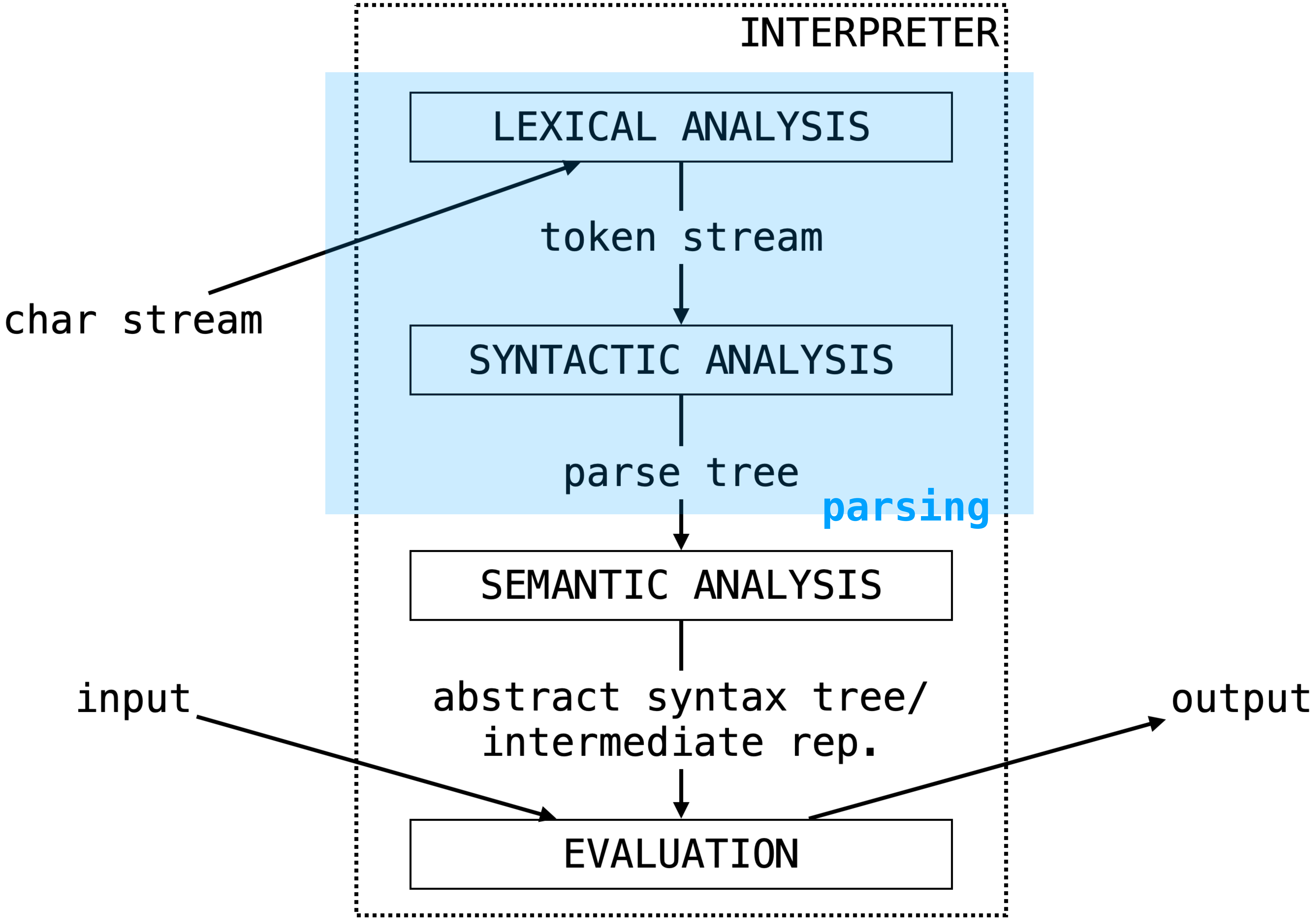
abstractions (non-terminal symbols)

tokens (terminal symbols)

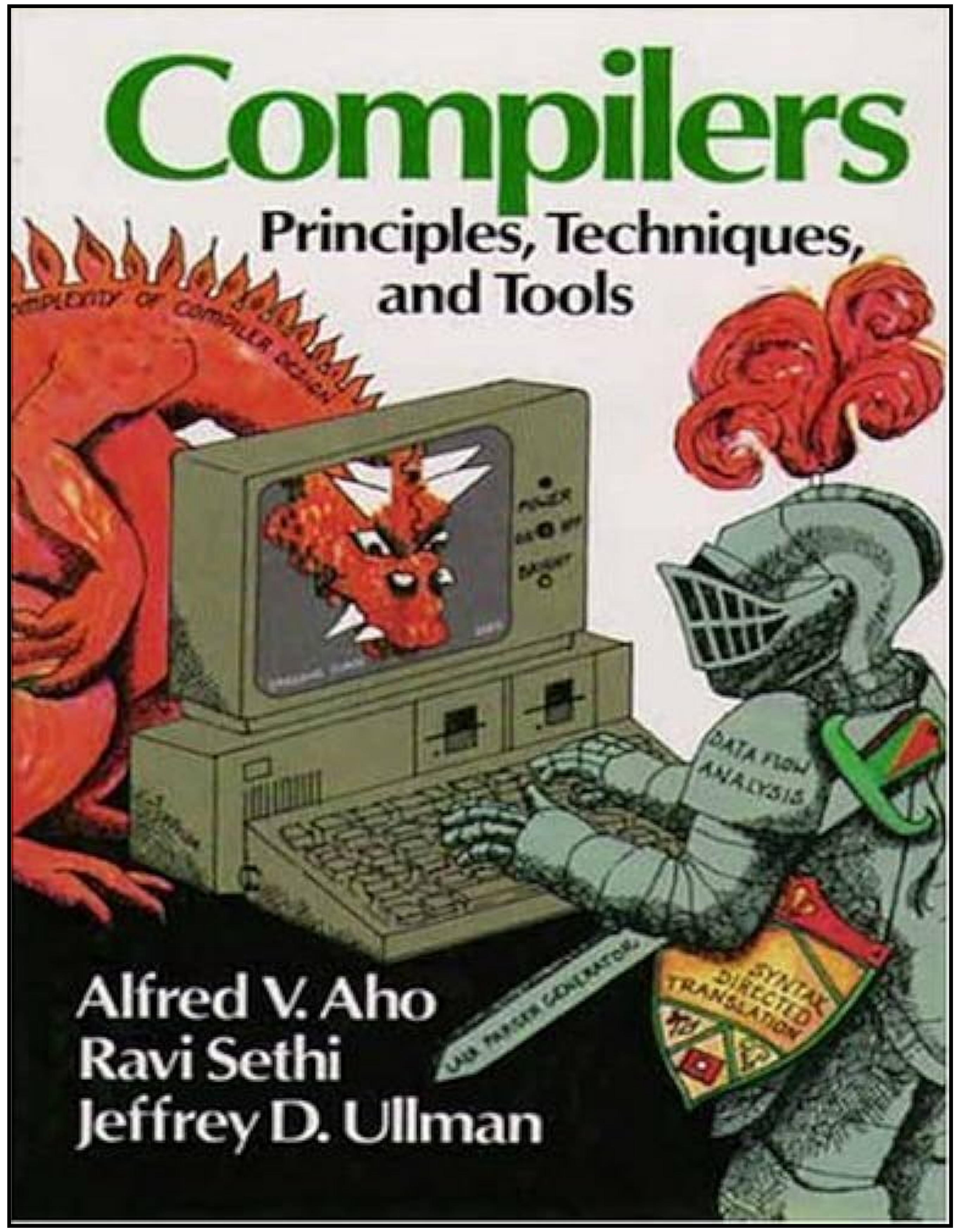
Pure Interpretation: The Picture



Pure Interpretation: The Picture

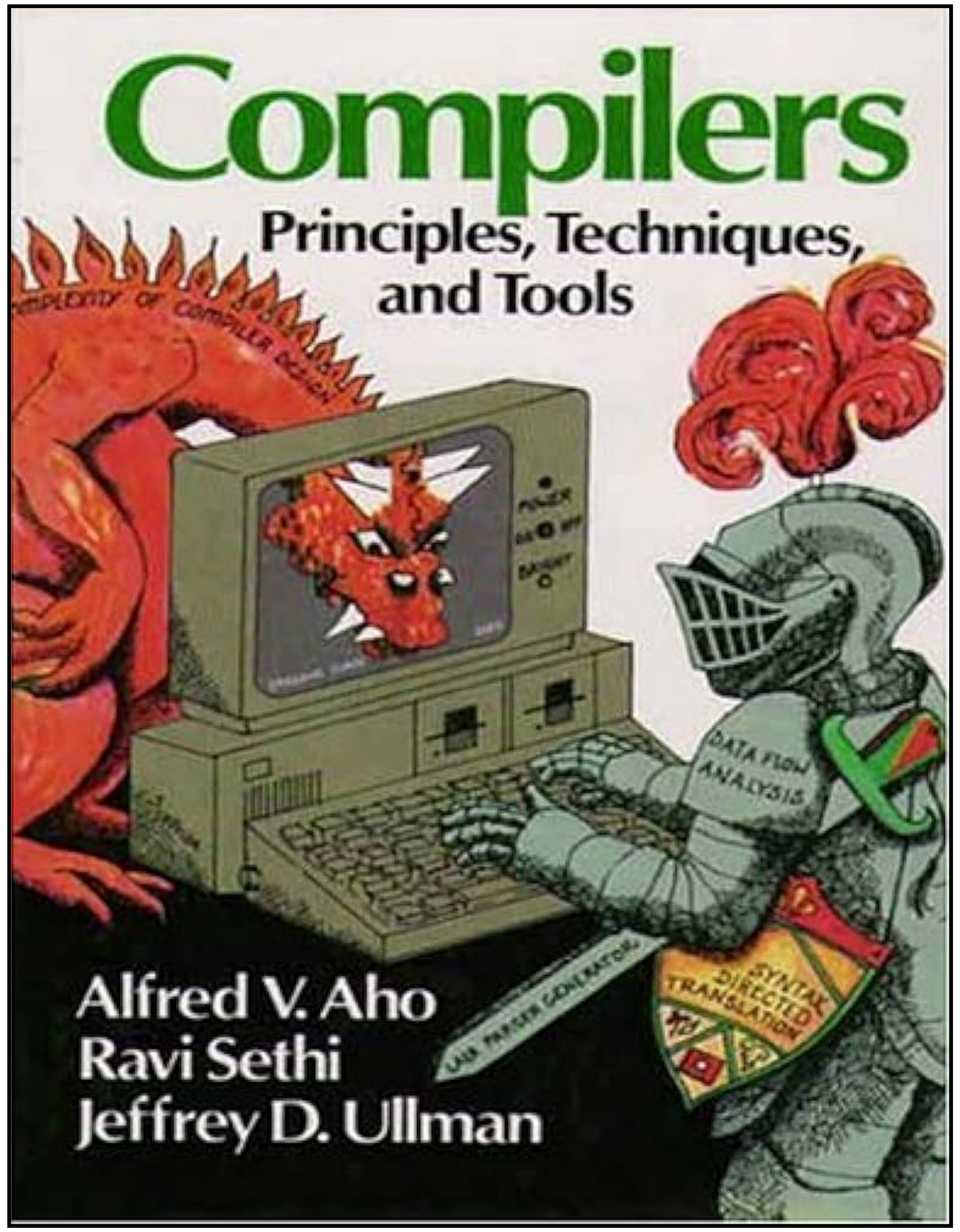


A Note on "History"


Compilers
Principles, Techniques, and Tools
COMPLEXITY OF COMPILER DESIGN
DATA FLOW ANALYSIS
LALR PARSER GENERATOR
SYNTAX DIRECTED TRANSLATION
Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman

A Note on "History"

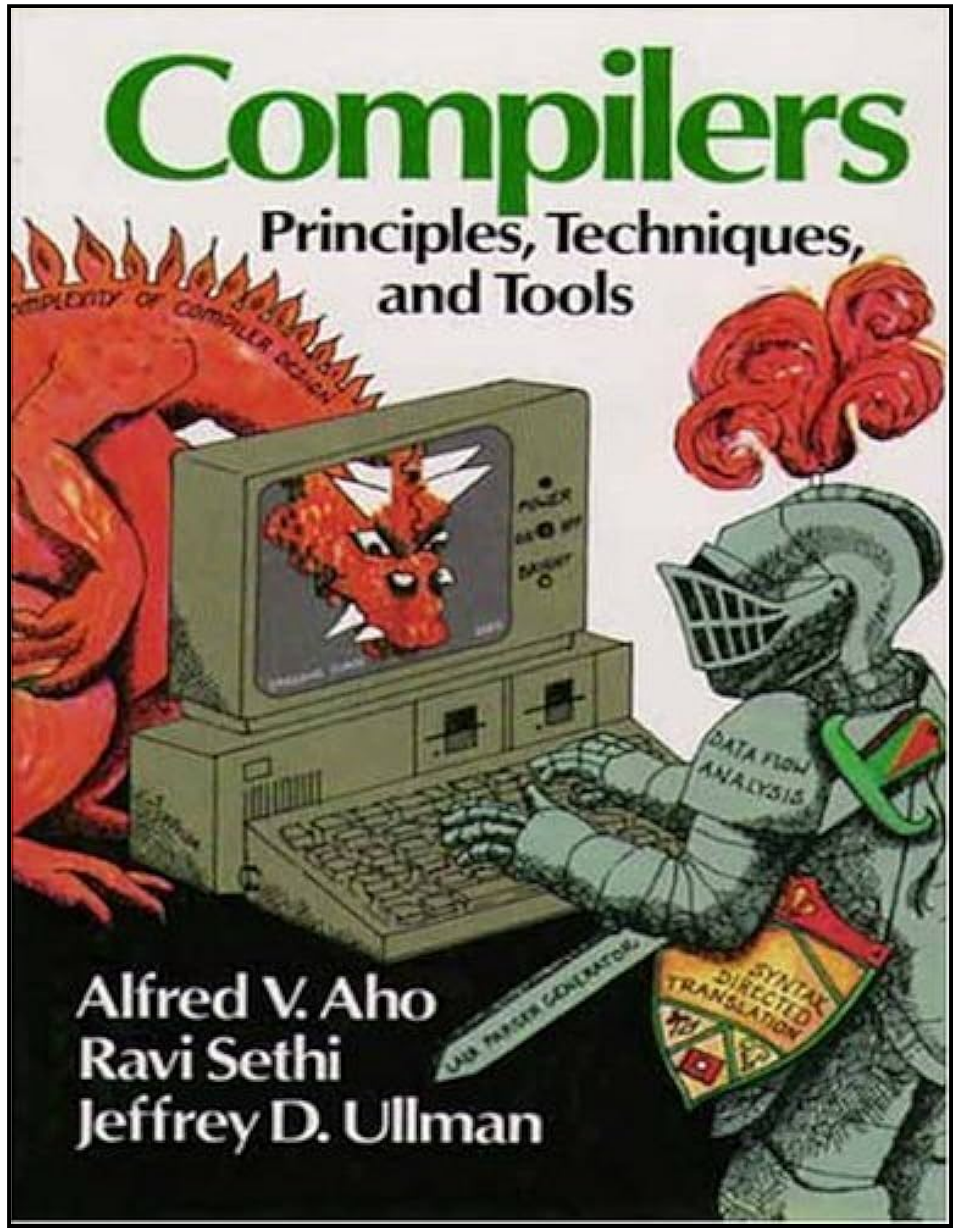
Lexical analysis and parsing are typically associated with ***Compiler Design***.


Compilers
Principles, Techniques, and Tools
COMPLEXITY OF COMPILER DESIGN
DATA FLOW ANALYSIS
SYNTAX DIRECTED TRANSLATION
LALR PARSER GENERATOR
Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman

A Note on "History"

Lexical analysis and parsing are typically associated with ***Compiler Design***.

Compiler design was once a fundamental requirement in CS programs. *This is not really the case anymore.*

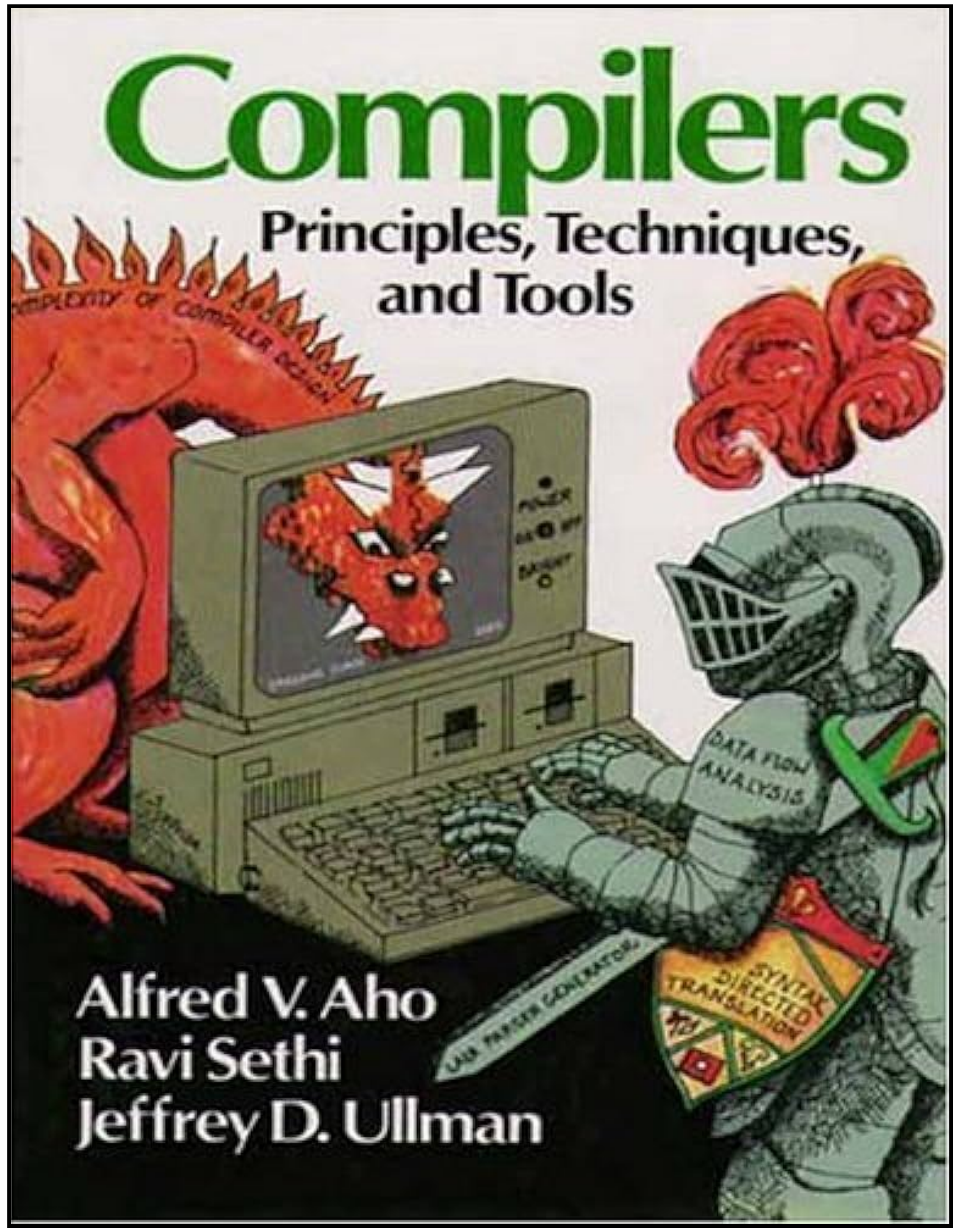

Compilers
Principles, Techniques, and Tools
COMPLEXITY OF COMPILER DESIGN
DATA FLOW ANALYSIS
LALR PARSER GENERATOR
SYNTAX DIRECTED TRANSLATION
Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman

A Note on "History"

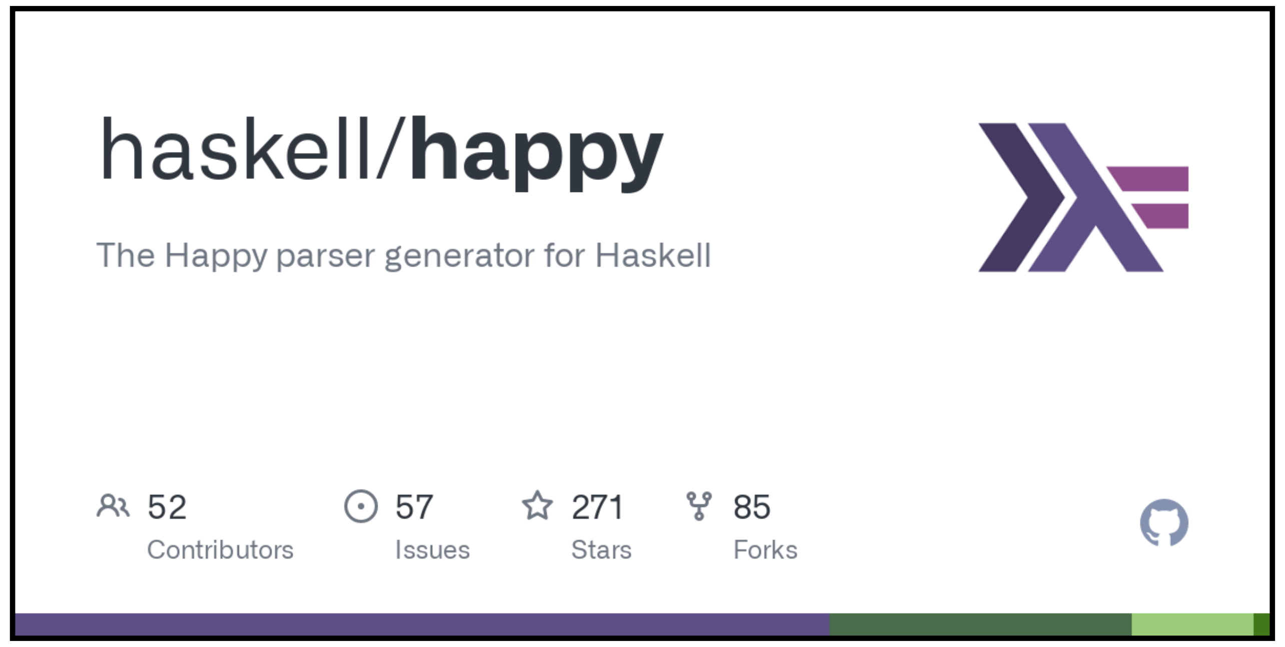
Lexical analysis and parsing are typically associated with ***Compiler Design***.

Compiler design was once a fundamental requirement in CS programs. *This is not really the case anymore.*

Also, we have ***parser generators***.



Parser Generators



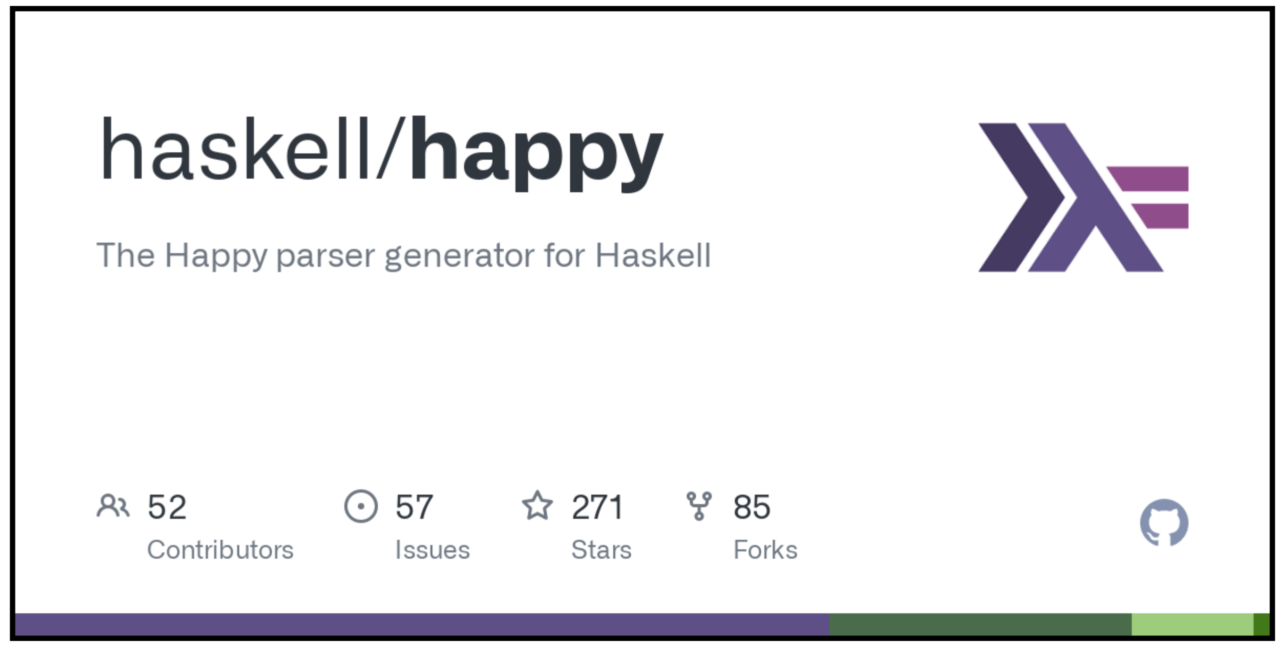
haskell/**happy**

The Happy parser generator for Haskell

52 Contributors · 57 Issues · 271 Stars · 85 Forks

(Beaver)

Parser Generators



haskell/**happy**

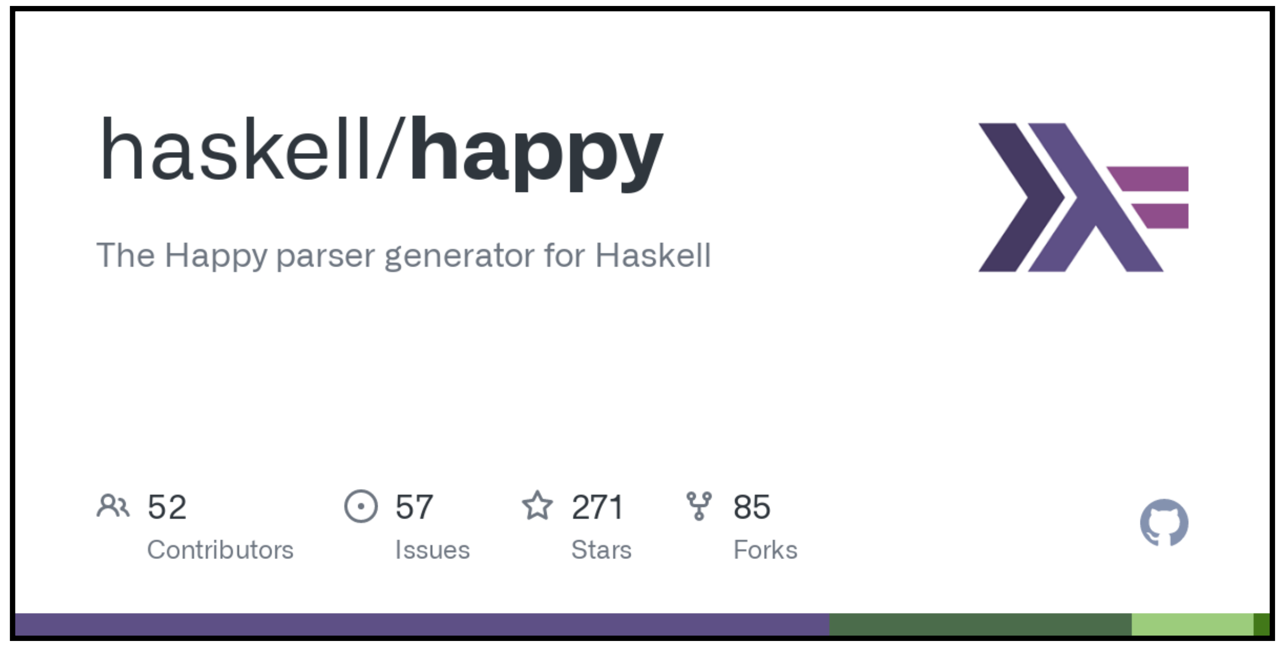
The Happy parser generator for Haskell

52 Contributors 57 Issues 271 Stars 85 Forks

(Beaver)

Parser generators are programs which, given a representation of a language (e.g., as an ***EBNF grammar***), build a parser for you.

Parser Generators



haskell/**happy**

The Happy parser generator for Haskell

52 Contributors 57 Issues 271 Stars 85 Forks

(Beaver)

Parser generators are programs which, given a representation of a language (e.g., as an ***EBNF grammar***), build a parser for you.

(So there was a point to learning (E)BNF for the "real-world")

demo

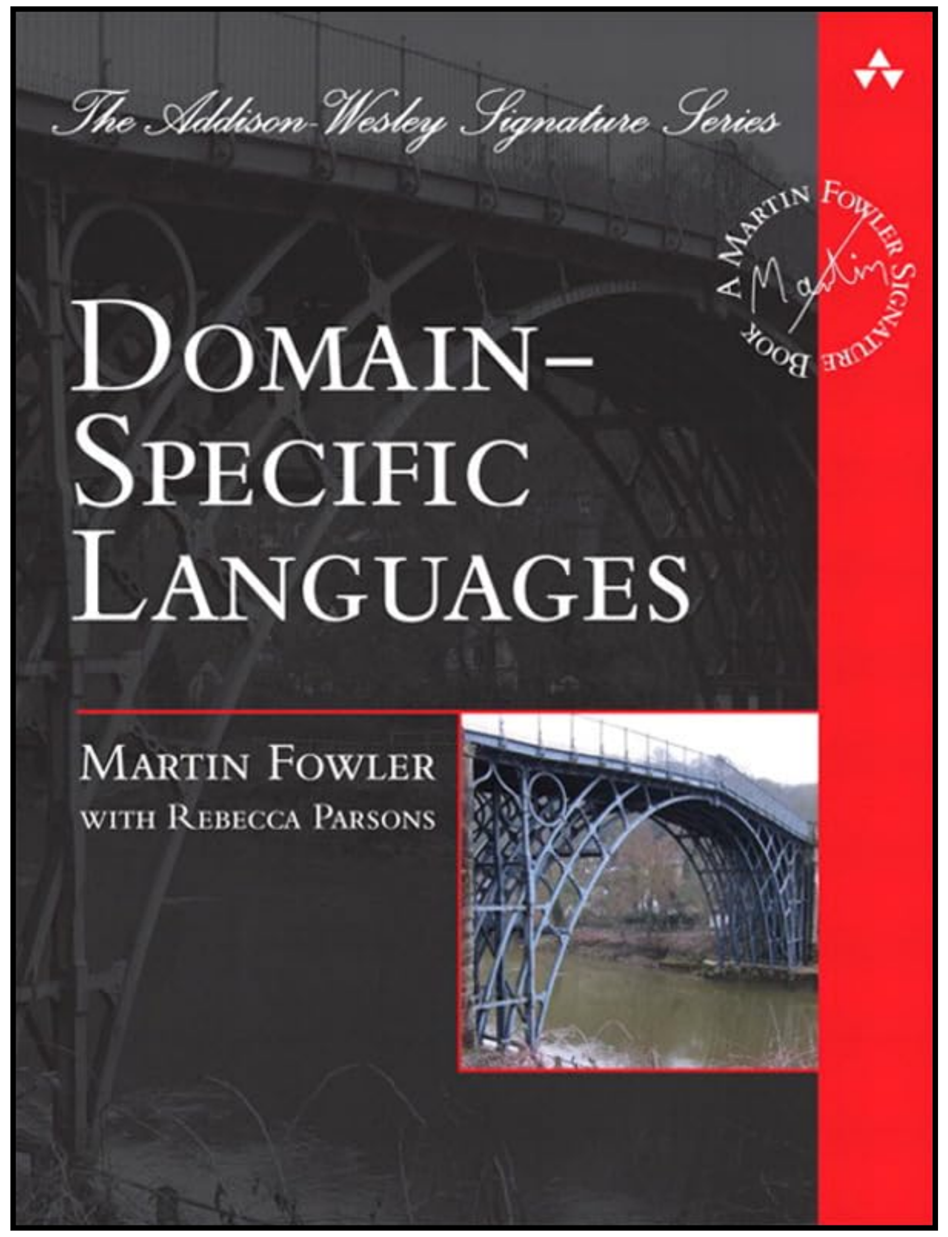
(ANTLR)

An Aside: Domain-Specific Languages

Domain-specific languages (DSLs) are simple programming languages for domain-specific tasks, e.g.

» Emacs Lisp
» SQL

*We need **parsers** for these languages if we want to use them...*


The Addison-Wesley Signature Series
A Martin Fowler Signature Book
Domain-Specific Languages
Martin Fowler
with Rebecca Parsons

SQL

Lexical Analysis

The "Lexing" Problem

"let" ≈ ['l', 'e', 't'] ↦ ***LET***

"fun" ≈ ['f', 'u', 'n'] ↦ ***FUN***

The "Lexing" Problem

"let" ≈ ['l', 'e', 't'] ↦ ***LET***

"fun" ≈ ['f', 'u', 'n'] ↦ ***FUN***

The Goal. *Convert a stream of characters into a stream of tokens.*

The "Lexing" Problem

"let" ≈ ['l', 'e', 't'] ↦ ***LET***

"fun" ≈ ['f', 'u', 'n'] ↦ ***FUN***

The Goal. *Convert a stream of characters into a stream of tokens.*

» Characters are grouped so together so they correspond to the smallest units at the level of the language.

The "Lexing" Problem

"let" ≈ ['l', 'e', 't'] ↦ ***LET***

"fun" ≈ ['f', 'u', 'n'] ↦ ***FUN***

The Goal. *Convert a stream of characters into a stream of tokens.*

» Characters are grouped so together so they correspond to the smallest units at the level of the language.

» Whitespace and comments are ignored.

The "Lexing" Problem

```
"let" ≈ ['l', 'e', 't'] ↦ LET

"fun" ≈ ['f', 'u', 'n'] ↦ FUN
```

The Goal. *Convert a stream of characters into a stream of tokens.*

» Characters are grouped so together so they correspond to the smallest units at the level of the language.

» Whitespace and comments are ignored.

» Syntax errors are caught, when possible.

Lexing vs. Parsing

Lexing vs. Parsing

Lexical Analysis is about small-scale language constructs.

Lexing vs. Parsing

Lexical Analysis is about small-scale language constructs.

- keywords, names, literals

Lexing vs. Parsing

Lexical Analysis is about small-scale language constructs.

- keywords, names, literals

Syntactic Analysis (Parsing) is about large-scale language constructs.

Lexing vs. Parsing

Lexical Analysis is about small-scale language constructs.

- keywords, names, literals

Syntactic Analysis (Parsing) is about large-scale language constructs.

- expressions, statements, modules

Why separate them?

Why separate them?

*Good question...*for simple implementations, we don't.

Why separate them?

*Good question...*for simple implementations, we don't.

But there are benefits for larger projects: *(what do you think?)*

Why separate them?

*Good question...*for simple implementations, we don't.

<u>But there are benefits for larger projects:</u> *(what do you think?)*

- » **Simplicity.** It's easier to think about parsing if we don't need to worry about whitespace, characters, etc.

Why separate them?

*Good question...*for simple implementations, we don't.

But there are benefits for larger projects: *(what do you think?)*

- » **Simplicity**. It's easier to think about parsing if we don't need to worry about whitespace, characters, etc.

- » **Portability**. Files are finicky things, handled differently across different operating systems. Abstracting this away for parsing is just good software engineering.

Lexemes and Tokens

input program:	fun	l	->	l	++	[	100	]
lexemes:	"fun"	"l"	"->"	"l"	"++"	"["	"100"	"]"
tokens:	FUN	(ID "l")	ARR	(ID "l")	(OP "++")	LBRAK	(INT 100)	RBRAK

Lexemes and Tokens

input program:	fun	l	->	l	++	[	100	]
lexemes:	"fun"	"l"	"->"	"l"	"++"	"["	"100"	"]"
tokens:	FUN	(ID "l")	ARR	(ID "l")	(OP "++")	LBRAK	(INT 100)	RBRAK

A **lexeme** is a sequence of characters associated a syntactic unit in a language.

Lexemes and Tokens

input program:	fun	l	->	l	++	[	100	]
lexemes:	"fun"	"l"	"->"	"l"	"++"	"["	"100"	"]"
tokens:	FUN	(ID "l")	ARR	(ID "l")	(OP "++")	LBRAK	(INT 100)	RBRAK

A **lexeme** is a sequence of characters associated a syntactic unit in a language.

A **token** is a lexeme together with information about what kind of unit it is.

Lexemes and Tokens

```
input program:  fun       l    ->       l        ++   [          100   ]
      lexemes: "fun"     "l"  "->"     "l"      "++" "["        "100" "]"
       tokens:  FUN  (ID "l") ARR  (ID "l") (OP "++") LBRAK (INT 100)  RBRAK
```

A **lexeme** is a sequence of characters associated a syntactic unit in a language.

A **token** is a lexeme together with information about what kind of unit it is.

- "12" and "234" are both INT_LITS, whereas "let" is a KEYWORD.

Lexemes and Tokens

input program:	fun	l	->	l	++	[	100	]
lexemes:	"fun"	"l"	"->"	"l"	"++"	"["	"100"	"]"
tokens:	FUN	(ID "l")	ARR	(ID "l")	(OP "++")	LBRAK	(INT 100)	RBRAK

A **lexeme** is a sequence of characters associated a syntactic unit in a language.

A **token** is a lexeme together with information about what kind of unit it is.

- "12" and "234" are both INT_LITS, whereas "let" is a KEYWORD.

We typically represent tokens as an ADT.

One Token at a time

`"  let@#_)($#@_J_@0#GKJ"` $\xrightarrow{\texttt{next\_token}}$ `(LET, "@#_)($#@_J_@0#GKJ")`

`"le x = 2"` $\xrightarrow{\texttt{next\_token}}$ `FAILURE`

One Token at a time

```
"  let@#_)($#@_J_@0#GKJ"  —next_token→  (LET, "@#_)($#@_J_@0#GKJ")

"le x = 2"                —next_token→  FAILURE
```

The approach.

One Token at a time

`"  let@#_)($#@_J_@O#GKJ"` $\xrightarrow{\texttt{next\_token}}$ `(LET, "@#_)($#@_J_@O#GKJ")`

`"le x = 2"` $\xrightarrow{\texttt{next\_token}}$ **FAILURE**

The approach.

» Given a stream of characters, determine if there is a valid lexeme at the beginning.

One Token at a time

`"  let@#_)($#@_J_@0#GKJ"` $\xrightarrow{\texttt{next\_token}}$ `(LET, "@#_)($#@_J_@0#GKJ")`

`"le x = 2"` $\xrightarrow{\texttt{next\_token}}$ **FAILURE**

The approach.

» Given a stream of characters, determine if there is a valid lexeme at the beginning.

» If there is, return its corresponding token and the remainder of the stream.

Question (Conceptual)

`"  let@#_)($#@_J_@0#GKJ"` $\xrightarrow{\texttt{next\_token}}$ `(LET, "@#_)($#@_J_@0#GKJ")`

`"le x = 2"` $\xrightarrow{\texttt{next\_token}}$ **`FAILURE`**

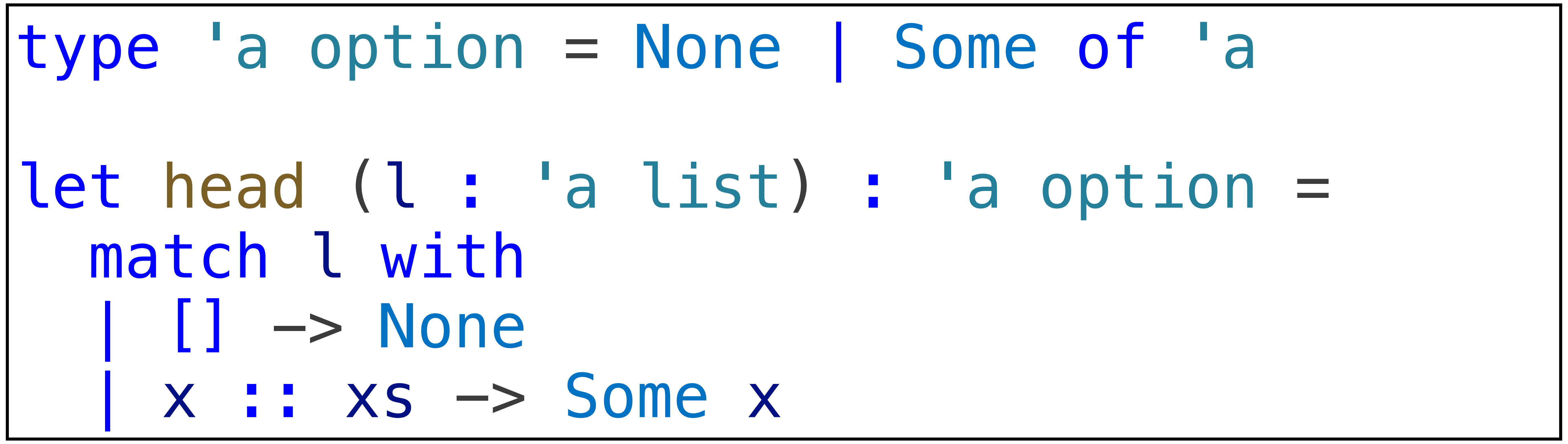
Why do it this way?

Possible Answers

- *What else could we do?* For example, splitting on whitespace could group characters unnecessarily.

- It generalizes nicely to cases when we don't have the entire input program, e.g., if we want to buffer the input, or if we want to combine lexing and parsing.

Recall: Options



```
type 'a option = None | Some of 'a

let head (l : 'a list) : 'a option =
  match l with
  | [] -> None
  | x :: xs -> Some x
```

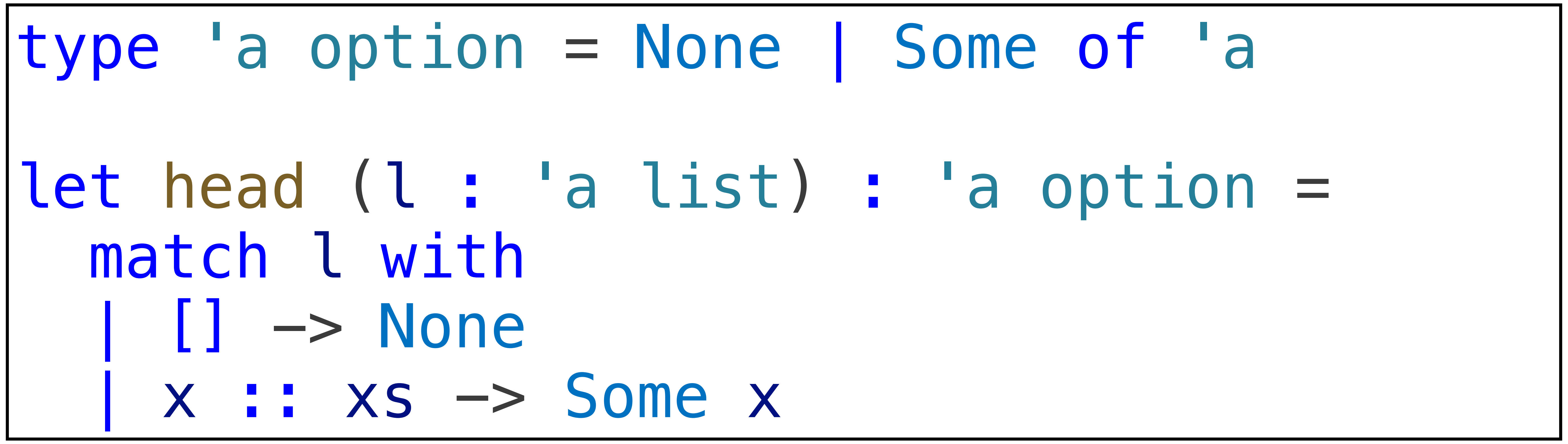
Recall: Options

```
type 'a option = None | Some of 'a

let head (l : 'a list) : 'a option =
  match l with
  | [] -> None
  | x :: xs -> Some x
```

Options are like boxes which *may* hold a value or may be empty.

Recall: Options



```
type 'a option = None | Some of 'a

let head (l : 'a list) : 'a option =
  match l with
  | [] -> None
  | x :: xs -> Some x
```

Options are like boxes which *may* hold a value or may be empty.

This can be useful for defining functions which may not be total.

Next Token

```
let rec next_token (cs : char list) : (token * char list) option = ...
```

Next Token

```
let rec next_token (cs : char list) : (token * char list) option = ...
```

To deal with possible failures of the **next_token** function, we use **options**.

Next Token

```
let rec next_token (cs : char list) : (token * char list) option = ...
```

input stream   next token   rest of stream

To deal with possible failures of the **next_token** function, we use **options**.

Next Token

```
let rec next_token (cs : char list) : (token * char list) option = ...
```

input stream   next token   rest of stream

To deal with possible failures of the **next_token** function, we use **options**.

» If we want to include syntax error messages, we'd need something with more structure (like **results**).

Next Token

```
let rec next_token (cs : char list) : (token * char list) option = ...
```

input stream next token rest of stream

To deal with possible failures of the **next_token** function, we use **options**.

» If we want to include syntax error messages, we'd need something with more structure (like **results**).

» If we wanted to buffer the input, we wouldn't use **lists**.

Tokenizing

```
next_token "let x = 2"     ⟹    Some (LET, " x = 2")
next_token " x = 2"        ⟹    Some (ID "x", " = 2")
next_token " = 2"          ⟹    Some (EQ, " 2")
next_token " = 2"          ⟹    Some (INT 2, "")

tokenize "let x = 2"       ⟹    Some [LET, ID "x", EQ, INT 2]
```

Tokenizing

```
next_token "let x = 2"     ⟹   Some (LET, " x = 2")
next_token " x = 2"        ⟹   Some (ID "x", " = 2")
next_token " = 2"          ⟹   Some (EQ, " 2")
next_token " = 2"          ⟹   Some (INT 2, "")

tokenize "let x = 2"       ⟹   Some [LET, ID "x", EQ, INT 2]
```

Once we have a **next_token** function. The process of turning a list of characters into a list of tokens is simple.

Tokenizing

```
next_token "let x = 2"    ⟹   Some (LET, " x = 2")
next_token " x = 2"       ⟹   Some (ID "x", " = 2")
next_token " = 2"         ⟹   Some (EQ, " 2")
next_token " = 2"         ⟹   Some (INT 2, "")

tokenize "let x = 2"      ⟹   Some [LET, ID "x", EQ, INT 2]
```

Once we have a **next_token** function. The process of turning a list of characters into a list of tokens is simple.

Just apply it a bunch of times until the list is empty, or until an error occurs (returning None).

Practice Problem

Implement the higher-order function **tokenize,** which given

    **next :** char list -> ('a * char list) option
    **cs :** char list

returns the **'a list** of elements gotten by repeatedly applying **next** until the list is empty, returning **None** if next ever returns **None.**

demo

(tokenizing)

Parsing

General Parsing

General Parsing

Problem. *Determine if a given sentence is recognized by a given grammar.*

General Parsing

Problem. *Determine if a given sentence is recognized by a given grammar.*

Two Approaches:

General Parsing

***Problem.** Determine if a given sentence is recognized by a given grammar.*

Two Approaches:

» **Top-Down.** Start with the start symbol, look for the "right" production rules to apply (recursive-descent)

General Parsing

Problem. *Determine if a given sentence is recognized by a given grammar.*

Two Approaches:

» **Top-Down**. Start with the start symbol, look for the "right" production rules to apply (recursive-descent)

» **Bottom-Up**. Start with the sentence, build the tree upwards (dynamic programming).

General Parsing

Problem. *Determine if a given sentence is recognized by a given grammar.*

Two Approaches:

» **Top-Down.** Start with the start symbol, look for the "right" production rules to apply (recursive-descent)

our focus

» **Bottom-Up.** Start with the sentence, build the tree upwards (dynamic programming).

Recursive-Decent (General)

Recursive-Decent (General)

`The Approach.`

Recursive-Decent (General)

The Approach.

» Choose a nonterminal symbol to expand, and apply a production rule

Recursive-Decent (General)

The Approach.

» Choose a nonterminal symbol to expand, and apply a production rule

» In the case of alternative rules, we have to choose an order to apply the rules.

Recursive-Decent (General)

The Approach.

» Choose a nonterminal symbol to expand, and apply a production rule

» In the case of alternative rules, we have to choose an order to apply the rules.

» Backtrack if we get to an sentential form which does not match our sentence.

The Picture

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= x | ( <expr> )
```

```
<expr>
```

```
x + ( x + x )
```

The Picture

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= x | ( <expr> )
```

```
<expr>
<expr2>
```

```
x + ( x + x )
```

The Picture

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= x | ( <expr> )
```

```
<expr>
<expr2>
x
```

```
x + ( x + x )
```

The Picture

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= x | ( <expr> )
```

```
<expr>
<expr2> + <expr>
```

```
x + ( x + x )
```

The Picture

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= x | ( <expr> )
```

```
<expr>
<expr2> + <expr>
x + <expr>
```

```
x + ( x + x )
```

The Picture

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= x | ( <expr> )
```

```
<expr>
<expr2> + <expr>
x + <expr>
x + <expr2>
```

```
x + ( x + x )
```

The Picture

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= x | ( <expr> )
```

```
<expr>
<expr2> + <expr>
x + <expr>
x + <expr2>
x + x
```

```
x + ( x + x )
```

The Picture

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= x | ( <expr> )
```

```
<expr>
<expr2> + <expr>
x + <expr>
x + <expr2>
x + ( <expr> )
```

```
x + ( x + x )
```

The Picture

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= x | ( <expr> )
```

```
<expr>
<expr2> + <expr>
x + <expr>
x + <expr2>
x + ( <expr> )
x + ( <expr2> )
```

```
x + ( x + x )
```

The Picture

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= x | ( <expr> )
```

```
<expr>
<expr2> + <expr>
x + <expr>
x + <expr2>
x + ( <expr> )
x + ( <expr2> )
x + ( x )
```

```
x + ( x + x )
```

The Picture

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= x | ( <expr> )
```

```
<expr>
<expr2> + <expr>
x + <expr>
x + <expr2>
x + ( <expr> )
x + ( <expr2> )
x + ( ( <expr> ) )
```

```
x + ( x + x )
```

The Picture

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= x | ( <expr> )
```

```
<expr>
<expr2> + <expr>
x + <expr>
x + <expr2>
x + ( <expr> )
x + ( <expr2> + <expr> )
```

```
x + ( x + x )
```

The Picture

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= x | ( <expr> )
```

```
<expr>
<expr2> + <expr>
x + <expr>
x + <expr2>
x + ( <expr> )
x + ( <expr2> + <expr> )
x + ( x + <expr> )
```

```
x + ( x + x )
```

The Picture

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= x | ( <expr> )
```

```
<expr>
<expr2> + <expr>
x + <expr>
x + <expr2>
x + ( <expr> )
x + ( <expr2> + <expr> )
x + ( x + <expr> )
x + ( x + <expr2> )
```

```
x + ( x + x )
```

The Picture

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= x | ( <expr> )
```

```
<expr>
<expr2> + <expr>
x + <expr>
x + <expr2>
x + ( <expr> )
x + ( <expr2> + <expr> )
x + ( x + <expr> )
x + ( x + <expr2> )
x + ( x + x )
```

```
x + ( x + x )
```

DONE

Practical Parsing

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= <int> | ( <expr> )
<int>   ::= ...
```

```
type expr
  = Num of int
  | Add of expr * expr
```

```
[LPAR; NUM 2; ADD; NUM 3; RPAR; ADD; NUM 4]  ⟶  Add (Add (Num 2, Num 3), 4)
 (         2  +        3  )     +    4
```

Practical Parsing

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= <int> | ( <expr> )
<int>   ::= ...
```

```
type expr
  = Num of int
  | Add of expr * expr
```

```
[LPAR; NUM 2; ADD; NUM 3; RPAR; ADD; NUM 4]  ⟶  Add (Add (Num 2, Num 3), 4)
 (         2  +        3  )     +    4
```

***Problem.** Convert a stream of tokens into a parse tree (represented as an ADT).*

Practical Parsing

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= <int> | ( <expr> )
<int>   ::= ...
```

```
type expr
  = Num of int
  | Add of expr * expr
```

```
[LPAR; NUM 2; ADD; NUM 3; RPAR; ADD; NUM 4]  →  Add (Add (Num 2, Num 3), 4)
 (         2  +        3  )     +    4
```

Problem. *Convert a stream of tokens into a parse tree (represented as an ADT).*

Note. An ADT does not have to *perfectly* model a grammar. There is no need for parentheses in the above example since it can be captured by the tree structure alone.

Recursive-Decent (Practical)

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= <int> | ( <expr> )
<int>   ::= ...
```

```
type expr
  = Num of int
  | Add of expr * expr
```

```
parseExpr  [NUM 1; ADD; NUM2; LPAR; LPAR]   ⟹  Some (Add (Num 1, Num 2), [LPAR; LPAR])
parseExpr  [NUM 1; ADD; ADD; LPAR]          ⟹  Some (Num 1, [ADD; ADD; LPAR])
parseExpr2 [NUM 1; ADD; NUM2]               ⟹  Some (Num 1, [ADD; NUM2])
parseExpr2 [LPAR; NUM 1]                    ⟹  None
```

Recursive-Decent (Practical)

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= <int> | ( <expr> )
<int>   ::= ...
```

```
type expr
  = Num of int
  | Add of expr * expr
```

```
parseExpr  [NUM 1; ADD; NUM2; LPAR; LPAR]   ⟹  Some (Add (Num 1, Num 2), [LPAR; LPAR])
parseExpr  [NUM 1; ADD; ADD; LPAR]          ⟹  Some (Num 1, [ADD; ADD; LPAR])
parseExpr2 [NUM 1; ADD; NUM2]               ⟹  Some (Num 1, [ADD; NUM2])
parseExpr2 [LPAR; NUM 1]                    ⟹  None
```

The Approach.

Recursive-Decent (Practical)

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= <int> | ( <expr> )
<int>   ::= ...
```

```
type expr
  = Num of int
  | Add of expr * expr
```

```
parseExpr  [NUM 1; ADD; NUM2; LPAR; LPAR]   ⟹  Some (Add (Num 1, Num 2), [LPAR; LPAR])
parseExpr  [NUM 1; ADD; ADD; LPAR]          ⟹  Some (Num 1, [ADD; ADD; LPAR])
parseExpr2 [NUM 1; ADD; NUM2]               ⟹  Some (Num 1, [ADD; NUM2])
parseExpr2 [LPAR; NUM 1]                    ⟹  None
```

The Approach.

» For each production rule, define a subprogram that parses that kind of non-terminal symbol. *(They will be mutually recursive)*

Recursive-Decent (Practical)

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= <int> | ( <expr> )
<int>   ::= ...
```

```
type expr
  = Num of int
  | Add of expr * expr
```

```
parseExpr   [NUM 1; ADD; NUM2; LPAR; LPAR]   ⟹   Some (Add (Num 1, Num 2), [LPAR; LPAR])
parseExpr   [NUM 1; ADD; ADD; LPAR]          ⟹   Some (Num 1, [ADD; ADD; LPAR])
parseExpr2  [NUM 1; ADD; NUM2]               ⟹   Some (Num 1, [ADD; NUM2])
parseExpr2  [LPAR; NUM 1]                    ⟹   None
```

The Approach.

» For each production rule, define a subprogram that parses that kind of non−terminal symbol. *(They will be mutually recursive)*

» Just like with tokenizing, we try to consume as much as possible from the input, returning what is left.

Recursive-Decent (Practical)

```
<expr>  ::= <expr2> | <expr2> + <expr>
<expr2> ::= <int> | ( <expr> )
<int>   ::= ...
```

```
type expr
  = Num of int
  | Add of expr * expr
```

```
parseExpr  [NUM 1; ADD; NUM2; LPAR; LPAR]  ⟹  Some (Add (Num 1, Num 2), [LPAR; LPAR])
parseExpr  [NUM 1; ADD; ADD; LPAR]         ⟹  Some (Num 1, [ADD; ADD; LPAR])
parseExpr2 [NUM 1; ADD; NUM2]              ⟹  Some (Num 1, [ADD; NUM2])
parseExpr2 [LPAR; NUM 1]                   ⟹  None
```

The Approach.

» For each production rule, define a subprogram that parses that kind of non-terminal symbol. *(They will be mutually recursive)*

» Just like with tokenizing, we try to consume as much as possible from the input, returning what is left.

» We're done if we've consumed every token.

demo

(parsing)

Note: The Choice of Rules is Important

```
<expr>  ::= <expr> + <expr2> | <expr2>
<expr2> ::= x | ( <expr> )
```

```
<expr>
```

Note: The Choice of Rules is Important

```
<expr>  ::= <expr> + <expr2> | <expr2>
<expr2> ::= x | ( <expr> )
```

```
<expr>
<expr> + <expr2>
```

Note: The Choice of Rules is Important

```
<expr>  ::= <expr> + <expr2> | <expr2>
<expr2> ::= x | ( <expr> )
```

```
<expr>
<expr> + <expr2>
<expr> + <expr> + <expr2>
```

Note: The Choice of Rules is Important

```
<expr>  ::= <expr> + <expr2> | <expr2>
<expr2> ::= x | ( <expr> )
```

```
<expr>
<expr> + <expr2>
<expr> + <expr> + <expr2>
<expr> + <expr> + <expr> + <expr2>
```

Note: The Choice of Rules is Important

```
<expr>  ::= <expr> + <expr2> | <expr2>
<expr2> ::= x | ( <expr> )
```

```
<expr>
<expr> + <expr2>
<expr> + <expr> + <expr2>
<expr> + <expr> + <expr> + <expr2>
<expr> + <expr> + <expr> + <expr> + <expr2>
⋮
```

In code, this would be an infinite loop.

What's to Come

- How do we deal with (left) associativity?

- How do we deal with precedence?

- Can we make this simpler and more general?