Understanding “code is data is code”

Mikey Austin

1st May 2017

Intro - “code is data”

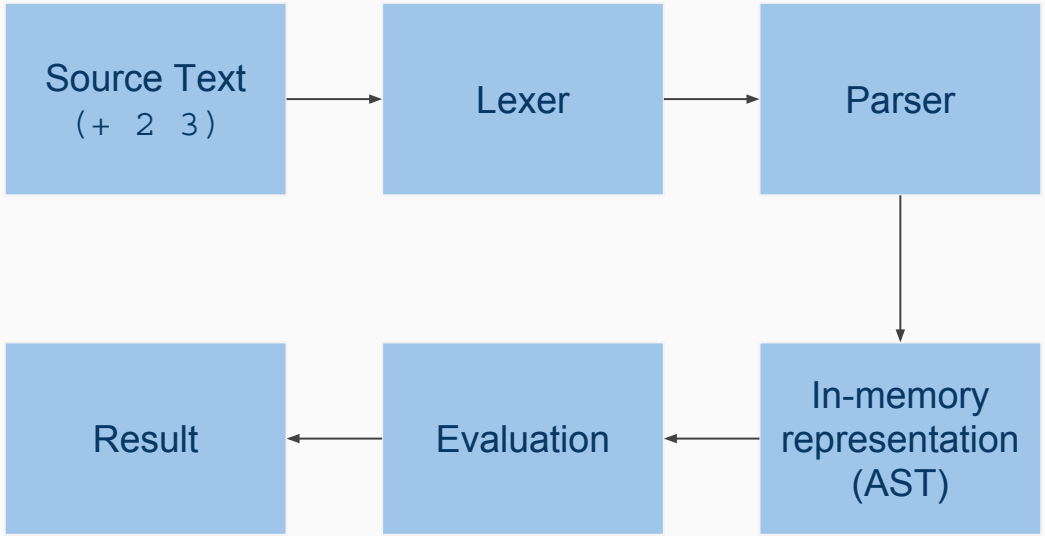
Let’s break down this ethereal statement:

- Syntax
- Parsing
- In-memory representation
- Evaluation

Stu; another lisp dialect

- github.com/mikey-austin/stu
- Embeddable, garbage-collected lisp dialect
- Macros, built-in partial evaluation, pluggable memory allocators, …
- Hack with us! =)

Typical pipeline



Lexer

Lisp is really light on syntax.

Splits text into a stream of ***tokens***, to be processed by the parser.

Token stream also useful for other applications, such as multi-line REPL input, eg:

```
stu> (map
>      (λ (a) (* 2 a))
>      '(1 2 3 4))
(2 4 6 8)
```

```
"[" |
"]" |
"(" |
")" |
"'" |
"`" |
"," |
"@" |
"." { return yytext[0]; }

\"(\\.|[^"])*\"    { yylval.s = yytext; return STRING; }
-?[0-9]+           { yylval.i = atol(yytext); return INTEGER; }
-?[0-9]+\.[0-9]+   { yylval.f = atof(yytext); return FLOAT; }
"#t"               { yylval.i = 1; return BOOLEAN; }
"#f"               { yylval.i = 0; return BOOLEAN; }

[a-zA-Z0-9._*+~#/=<>?!&\x80-\xf3-]+ { yylval.str = yytext; return SYMBOL;
}

";".*

[ \t\n]

.
```

Parser

Lisp is simple to parse!

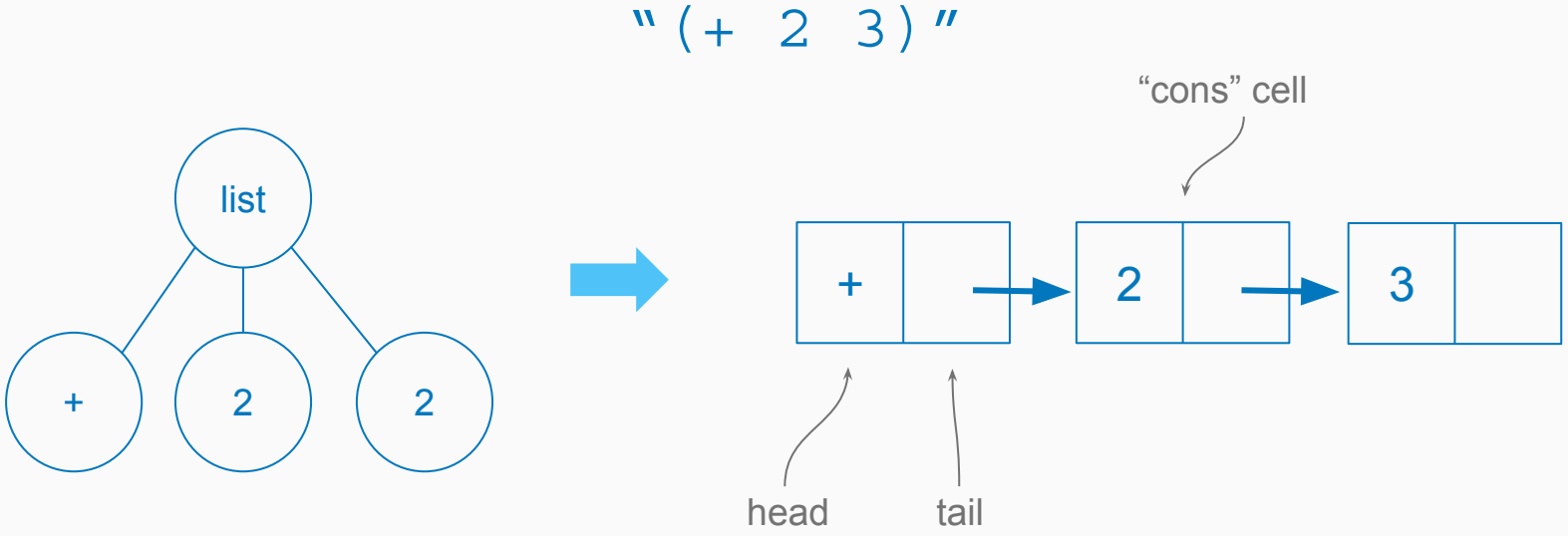
List processing in every sense.

Bison generates bottom up parser.

We build up the AST as we traverse the parse tree.

```
stu:
    | forms                   { Svlist_push(*list, $1); }
    ;

forms: forms sexp             { Svlist_push(*list, $1); $$ = $2; }
    | sexp                    { $$ = $1; }
    ;

list: '(' ')'                 { $$ = NIL; }
    | '(' elements ')'        { $$ = $2; }
    | '(' sexp '.' sexp ')'   { $$ = Sv_cons(stu, $2, $4); }
    ;

elements: sexp                { $$ = Sv_cons(stu, $1, NIL); }
    | sexp elements           { $$ = Sv_cons(stu, $1, $2); }
    ;

sexp: atom                    { $$ = $1; }
    | list                    { $$ = $1; }
    | '\'' sexp               { $$ = Sv_cons(stu, Sv_new_sym(stu, "quote"),
                                          Sv_cons(stu, $2, NIL)); }
    ;

atom: INTEGER                 { $$ = Sv_new_int(stu, $1); }
    | FLOAT                   { $$ = Sv_new_float(stu, $1); }
    | STRING                  { $$ = Sv_new_str(stu, $1); }
    | SYMBOL                  { $$ = Sv_new_sym(stu, $1); }
    | RATIONAL                { $$ = Sv_new_rational(stu, $1.n, $1.d); }
    | BOOLEAN                 { $$ = Sv_new_bool(stu, (short) $1); }
    ;
```

Token Stream to AST



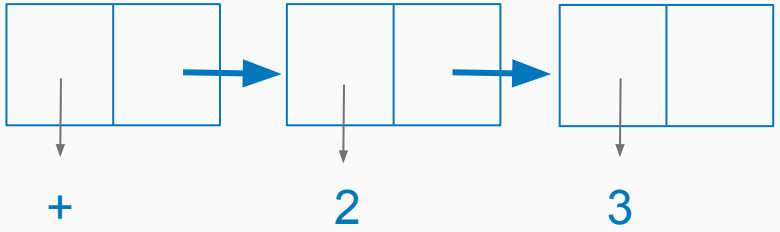
Evaluating ASTs

- AST nodes are dynamically typed in stu
- Could be an *integer, string, symbol, lambda, cons cell, …*
- **Eval takes an AST and returns an AST**
- … so we can take any object and *treat it as code* by evaluating it
- It is easier to *work with* code

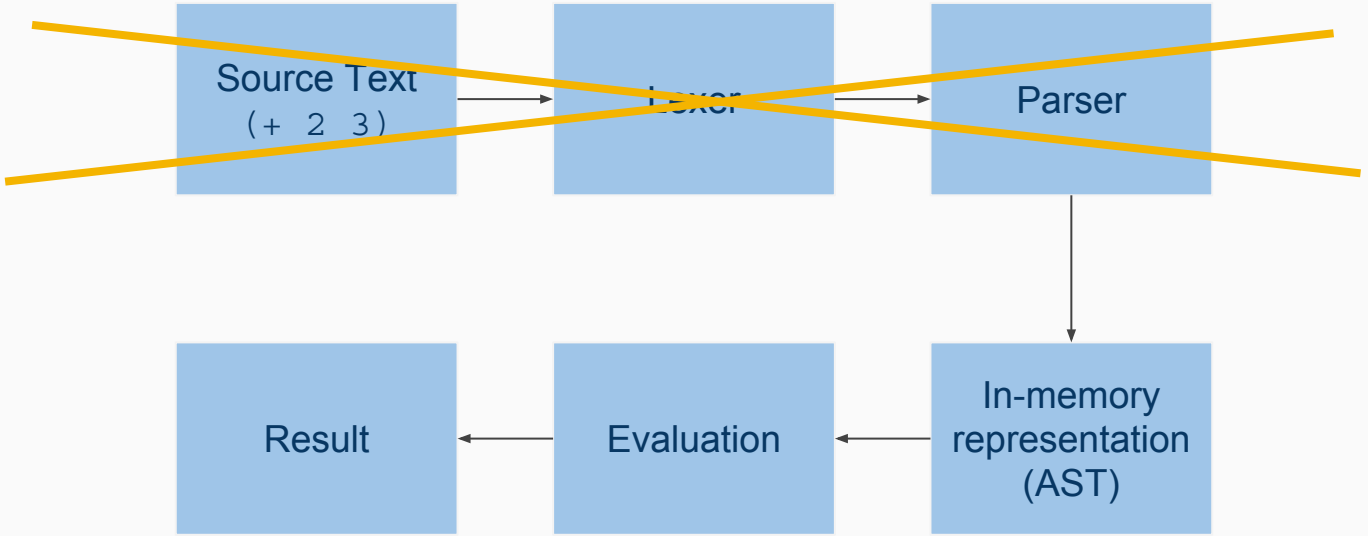
Code Writing Code

“Homoiconicity”

```
"(+ 2 3)"
```



Jump Straight to the AST



In Other Languages

- ASTs dramatically different from syntactic representation
- Effectively no facilities in dynamic languages (eg perl, python, etc.) for manipulating ASTs
- Next best thing is to invoke the full pipeline as described earlier; string manipulation

Subtle yet Powerful

Downsides to the alternatives:

String manipulation is inconvenient

Escaping issues

Eval'ing strings in some languages *kind-of* "considered harmful", if that's your bag

Some benefits:

- Reflective capabilities; code walkers
- Generation; eg Emacs keyboard macros to lisp code
- Enables powerful macro systems

Thanks!

Mikey Austin

Code for stu
github.com/mikey-austin/stu