CS 252:

Advanced Programming Language Principles

Macros & Sweet.js

Prof. Tom Austin

San José State University

Let's say we wanted to use classes in JavaScript, but we are using an old interpreter…

We'd like to have something like:

```
class Person {
  constructor(name) {
    this.name = name;
  }

  say(msg) {
    console.log(this.name +
                " says: " + msg);
  }
}
```

But what we have to type is:

```
function Person(name) {
  this.name = name;
}

Person.prototype.say = function(msg) {
  console.log(this.name +
              " says: " + msg);
}
```

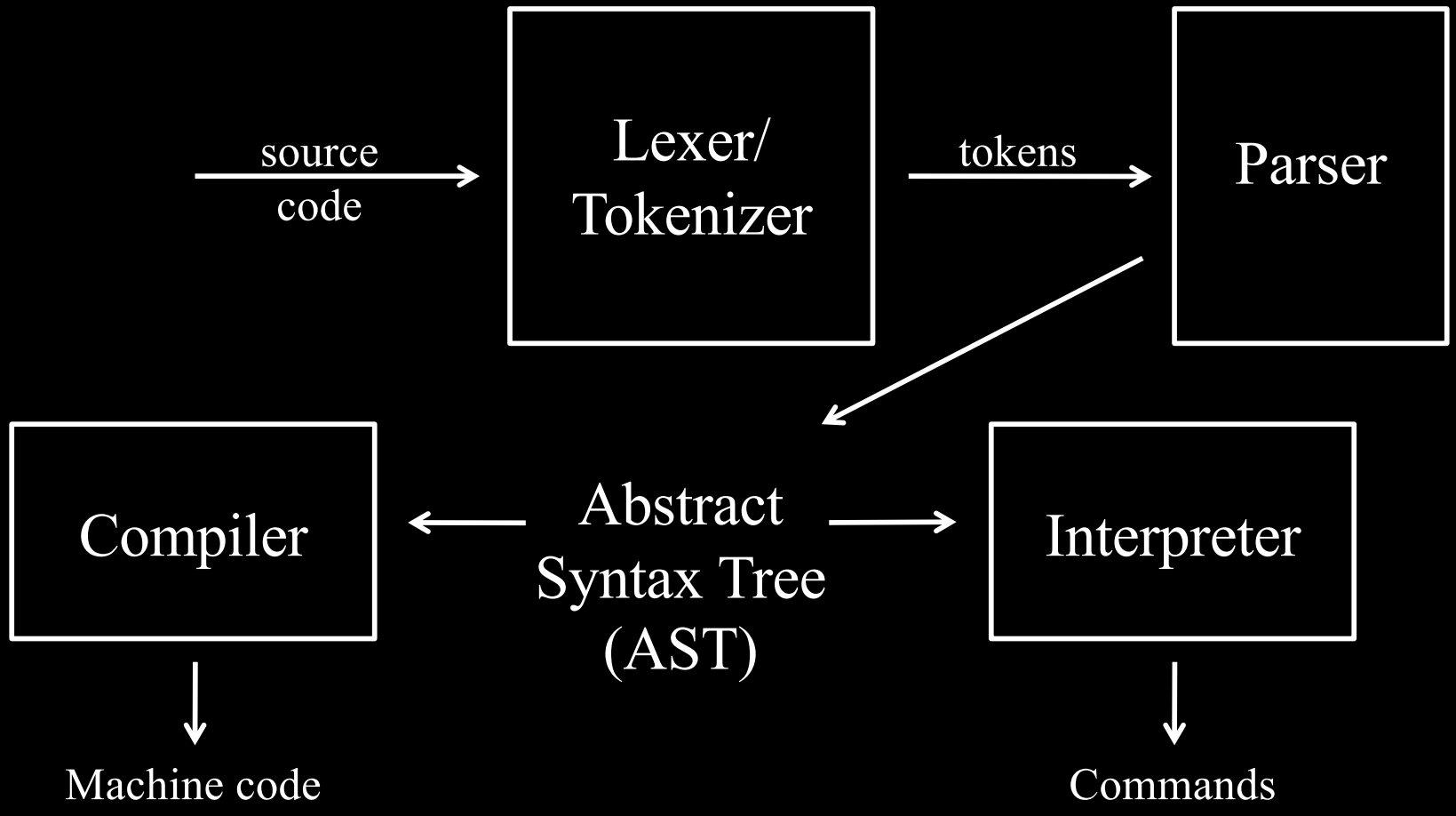
We want to *expand* our code with classes to a version of JavaScript understood by the interpreter.

Introducing macros…

What is a macro?

- Short for *macroinstruction*.
- Rule specifies how **input sequence** maps to a **replacement sequence**.

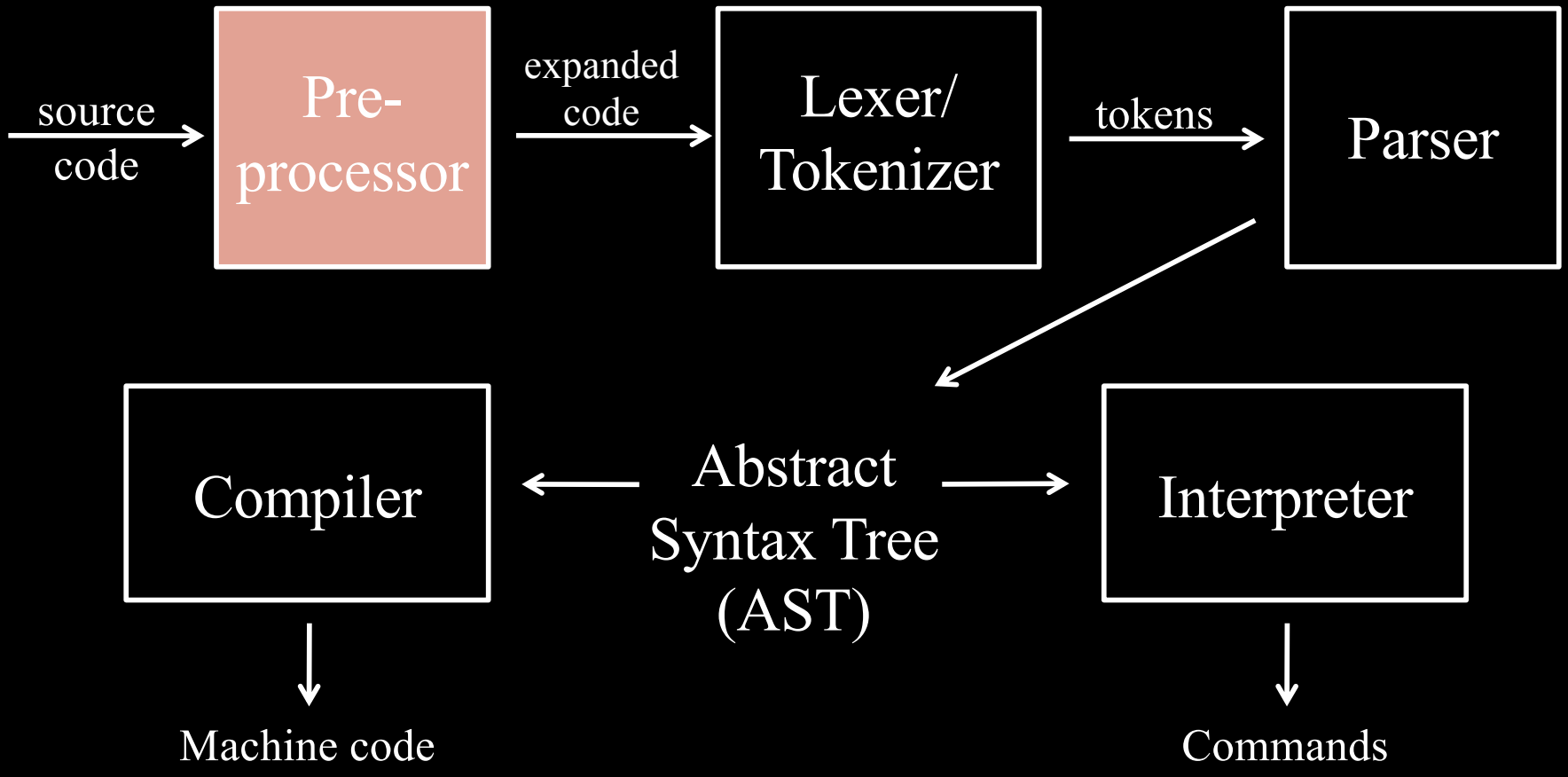
A Review of Compilers



Macros in C

- C *preprocessor*
- *Text substitution* macros
  - text is converted to text.
- Embedded languages are similar
  - PHP, Ruby's erb, etc.

Some variants work at the token level, but the concept is the same.



C preprocessor example

```
#define PI 3.14159

#define SWAP(a,b) {int tmp=a;a=b;b=tmp;}

int main(void) {
    int x=4, y=5, diam=7, circum=diam*PI;
    SWAP(x,y);
}
```

```
int main(void) {
    int x=4, y=5, diam=7, circum=diam*PI;
    SWAP(x,y);
}
```


```
int main(void) {
  int x=4, y=5, diam=7,
      circum=diam*3.14159;
  {int tmp=x;x=y;y=tmp;};
}
```

Problems with C macros

(in class)

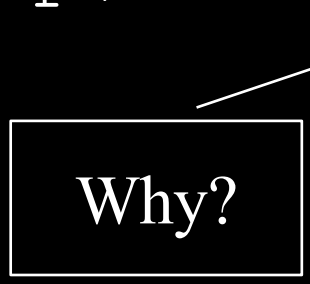
Many macro systems suffer from *inadvertent variable capture.*

Let's look at an example…

Hygiene

Hygienic macros are macros whose expansion is guaranteed not to cause the *accidental capture of identifiers*.

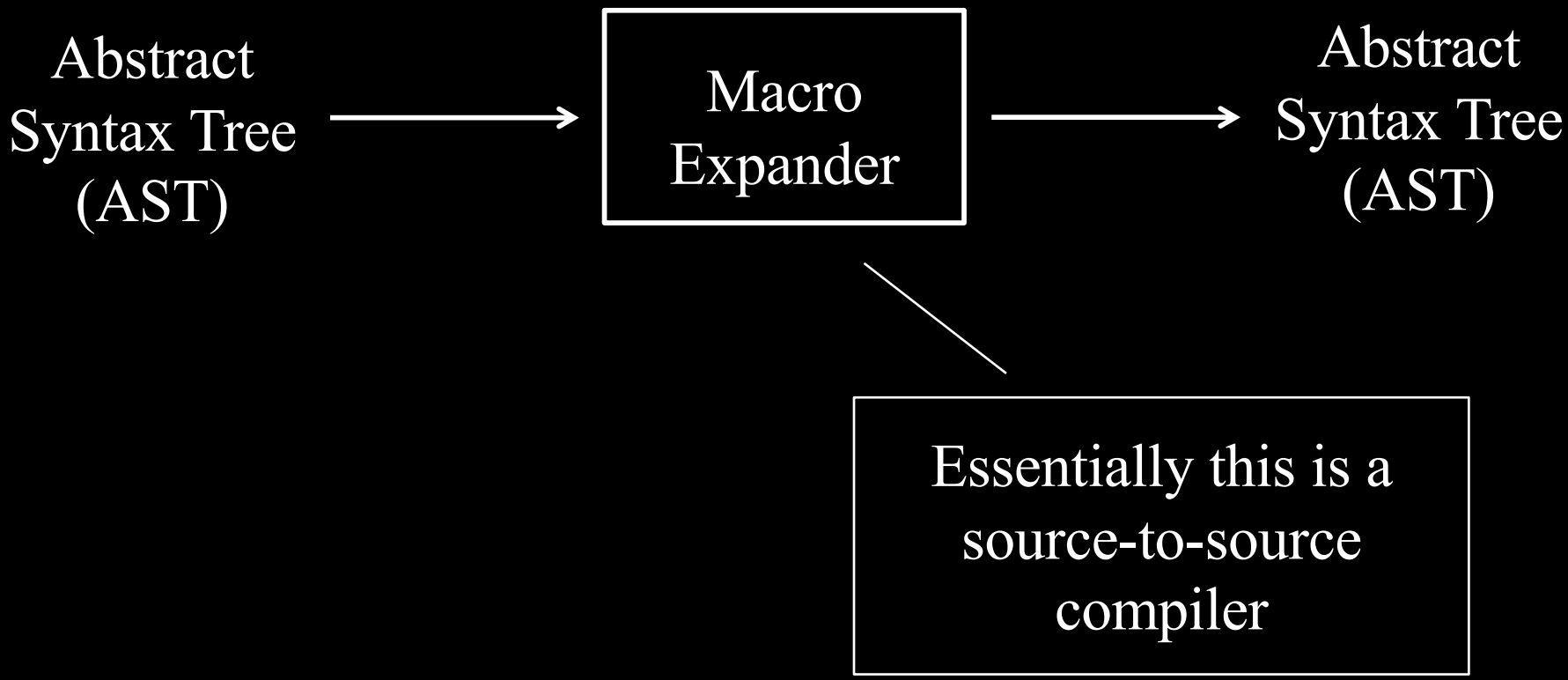
```
//macro should be on one line
#define SWAP(a,b) { int tmp=a;
                    a=b; b=tmp; }
int main(void) {
    int x=4, y=5, tmp=7;
    SWAP(x,y);   // Swaps x&y
    SWAP(x,tmp); // tmp unchanged
}
```



Syntactic macros

- Work at the level of abstract syntax trees
- From the Lisp family
  - Why Lisp? Because Lisp programs *are* ASTs
- Powerful, but expensive
- *Hygiene* is still a major concern, but is perhaps easier to address at that level

Macro expansion process



Macros in Racket

```
(define-syntax-rule (swap x y)
  (let ([tmp x])
    (set! x y)
    (set! y tmp)))
```

Macros in Racket

```
(define-syntax-rule (swap x y)
  (let ([tmp x])
    (set! x y)
    (set! y tmp)))
```

Macros in Racket

```
(define-syntax-rule (swap x y)
  (let ([tmp x])
    (set! x y)
    (set! y tmp)))
```

Macros in Racket

```
(define-syntax-rule (swap x y)
  (let ([tmp x])
    (set! x y)
    (set! y tmp)))
(let ([a 7][b 3])
  (swap a b)
  (displayln a)
  (displayln b))
```

Expanded code

```
(define-syntax-rule (swap x y)
  (let ([tmp x])
    (set! x y)
    (set! y tmp)))
(let ([a 7][b 3])
  (let ([tmp a])
    (set! a b)
    (set! b tmp))
  (displayln a)
  (displayln b))
```

Macros for JavaScript

- No standard macro system for JavaScript.
- Sweet.js has been gaining interest.
- Recent redesign.
- http://sweetjs.org/
- https://www.sweetjs.org/doc/tutorial.html

Sweet.js high-level

- Source-to-source compiler for JavaScript.
  - Other s2s compilers for JS:
    - TypeScript
    - CoffeeScript
    - Dart (though also has a VM)
- Project backed by Mozilla
- Concepts borrowed from Racket

Prototypal Inheritance

```
var Droid = {
  speak: function() {
    console.log(">>Beep, boop<<, " +
        "I am " + this.name);
  },
  create: function(name) {
    var clone = Object.create(this);
    clone.name = name;
    return clone;
  },
};
```

We create new droids like so:

```
var areToo =
    Droid.create('R2-D2');
```

but we are used to calling:

```
var bb8 = new Droid('BB8');
```

Macro

```
syntax new = function (ctx) {
  let ident = ctx.next().value;
  let params = ctx.next().value;
  return #`${ident}.create ${params}`;
}

var bb8 = new Droid('BB8');
```

Translated version

```
var Droid_0 = { speak: function speak() {
    console.log(">>Beep, boop<<, I am "
        + this.name);
  }, create: function create(name_8) {
    var clone_9 = Object.create(this);
    clone_9.name = name_8;
    return clone_9;
  } };
var bb8_7 = Droid_0.create("BB8");
```

Installing Sweet.js

From a Unix/Dos command line:

```
$npm install -g @sweet-js/cli
$npm install @sweet-js/helpers
```

Invoking Sweet.js

- Compile your code:
  ```
  $sjs myfile.js -d out/
  ```
- Then you may run the output file normally:
  ```
  $node out/myfile.js
  ```

```
syntax swap = function (ctx) {
  var a = ctx.next().value;
  var b = ctx.next().value;
  return #`var tmp =${a}; ${a}=${b}; ${b}=tmp;`;
}

var a = 10; var b = 20;
console.log("a:" + a + " b:" + b);
swap a b;
console.log("a:" + a + " b:" + b);
```

Iterating over syntax

```
syntax square = function (ctx) {
  var inCtx = ctx.contextify(ctx.next().value);
  var result = #``;
  var stx;
  for (stx of inCtx) {
    result = result.concat(
        #`${stx} = ${stx}*${stx};`);
    inCtx.next(); // Eating comma
  }
  return result;
}
var a = 1; var b = 2; var c = 3;
square(a, b, c);
console.log("a:"+a+" b:"+b+" c:"+c);
```

Output

```
var a_5 = 1;
var b_6 = 2;
var c_7 = 3;
a_5 = a_5 * a_5;
b_6 = b_6 * b_6;
c_7 = c_7 * c_7;
console.log("a:" + a_5 + " b:" +
        b_6 + " c:" + c_7);
```

Sweet.js helper functions

```
import { isStringLiteral }
    from '@sweet-js/helpers' for syntax;
syntax m = function(ctx) {
    if (isStringLiteral(ctx.next().value))
        return #``'a string'``;
    else
        return #``'not a string'``;
}
m 'foo';
m 42;
var s = "hello";
m s;
```

Adding classes to JavaScript.
(in class)

Lab

Create a **`rotate`** macro in Sweet.js that works like the **`swap`** macro, except that it takes an *arbitrary number of arguments*.

There is no starter code for this lab.