ES6 Destructuring Binding/Assignment Design

Generally speaking, ES6 destructuring will be implemented by desugaring in the parser. No new kinds of AST nodes will be created by this effort (besides Spread, needed already).

Owners

Andy Wingo -- wingo@igalia.com
Dmitry Lomov -- dslomov@chromium.org

Expansion of destructuring productions in the ES6 grammar

Production	Source	Expansion

LexicalBinding	`let [x, y = 42] = z`	`let t1 = GetIterator(z)`
		`let x = GetNext(t1)`
		`let t2 = GetNext(t1)`
		`let y = t2 === undefined ? 42 : t2`

	`const { x, y = 42 } = z`	`let t1 = ToObject(z)`
		`const x = t1.x`
		`let t2 = t1.y`
		`const y = t2 === undefined ? 42 : t2`

	`let [ {x}, { y = 42 } ] = z`	`let t1 = GetIterator(z)`
		`let t2 = ToObject(GetNext(t1))`
		`let x = t2.x`

		`let t3 = ToObject(GetNext(t1))`
		`let t4 = t3.y`
		`let y = t4 === undefined ? 42 : t4`

	`let [...x] = z`	`let t1 = z`
		`let t2 = []`
		`let t3 = 0`
		`for (let t4 of t1)`
		`  t2[t3++] = t4`
		`let x = t2`

VariableDeclaration	`var [x, y = 42] = z`	`var x, y;` `[x, y = 42] = z;`

ForBinding	`for (let [x, y=42] of z) {`
	`  let x = 10`
	`}`
		`for (let t of z) {`
		`  let [x, y = 42] = t;`
		`  { let x = 10 }`
		`}`

CatchParameter	`try {} catch ([x, y=42]) {}`	`try {} catch (t) {`
		`  let [x, y = 42] = t;`
		`}`

FormalParameter	`function ([x, y = 42]) {}`	`function(t) { let [x, y = 42] = t; }`[1]
	`([x, y = 42]) => {}`	`(t) => { let [x, y = 42] = t }`[2]

AssignmentExpression	`[x, y.foo = 42] = z`	`do {`
		`  let t1 = GetIterator(z)`
		`  x = GetNext(t1)`
		`  let t2 = GetNext(t1)`
		`  y.foo = t2 === undefined ? 42 : t2`
		`}`

Modifications to the parser

Though not identical, pattern syntax and expression syntax are very close. This is lovely on a language level. Unhappily, though, this similarity makes it not always apparent to the parser whether it's parsing an expression or a pattern.

---

[1] Note that the precise scoping rules for default initializer expressions of FormalParameter have yet to reach the main specification. Also note that the presence of a destructuring pattern or default initializer in function parameters causes the arguments object not to alias the formal parameters.

[2] Arrow function formals have the same caveat regarding scoping, but they don't have an arguments binding, so the arguments mapping comment does not apply.

V8's parser is a recursive-descent template that returns values of type Expression. We plan to modify the parser in a few ways. Firstly, we add another return value to the parser function: a set of flags indicating whether the parsed expression is valid as

- an arrow function argument;
- an assignment destructuring pattern;
- a binding destructuring pattern;
- an expression.

Some motivating examples:
Valid as an expression but not a pattern: [x()]
Valid as a pattern but not an expression: {x=y}
Valid as an assignment pattern but not a binding pattern: [x.y]
Valid as a pattern (within a generator or sloppy fn) but not an arrow fn parameter: [x=(yield)]

Every place in the parser that recursively descends to parse a sub-expression will have to check these flags to determine that the parsed node is actually a valid expression.

Likewise, the continuations in the parser that expect patterns will have to verify the appropriate flags, or otherwise signal a syntax error. However instead of returning a "Destructuring binding/assignment" node, they will instead desugar into a sequence of more primitive statements (see table above).

Dependencies on future work

This strategy requires "do" expressions in order to be able to desugar destructuring assignment, as assignment is an expression.

SpreadExpression, not yet present, will be the parsing of the ...y in [...y].

Reinterpreting parsed AST nodes is tricky

The plan is to parse patterns as ArrayLiteral and other standard AST node types, then revisit those nodes to desugar to other AST node kinds. However, parsing can have side effects, and parsing a reinterpreted AST might not have the same side effects. Currently, for example, expressions are allocated contiguous, locally unique "bailout ids", and the function itself accumulates a count of materialized literals and (if a constructor) the expected number of properties on the constructed object. All of this state must be rolled back or reinterpreted somehow.

Instead of trying to account for this state, we are going to try to move stateful calculations to a new tree-visiting post-pass, which will be run by full-codegen, crankshaft, or turbofan as

needed, that will compute bailout ids and other needed state. In this way it should be easier to transplant (e.g.) initializers of arrow function parameters from the outer to the inner function, without worrying about state. There has been some concern about performance impacts of a post-pass; we will benchmark and proceed if a parser-only worst-case regression is less than a few percent.

Order of implementation

1. Modify the parser to compute bailout ids and other stateful items in a post-pass
2. Modify the parser to have the extra “flags” return argument, and modify continuations to check the flags. Open question: use flags for parsing arrow functions as well?
3. Add support for LexicalBinding.
4. VariableDeclaration
5. ForBinding
6. CatchParameter
7. FormalParameter for function
8. AssignmentExpression (requires do expressions)
9. FormalParameter for arrow functions