簡単JavaScript AST入門

ASTで1桁上の生産性

誰でも簡単にソース解析・加工

東京ラビットハウス

えるきち 著

はじめに

この本は JavaScript AST の入門書です。AST とはソースコードを扱いやすいように加工されたデータ構造のことです。AST を操作するとソースコードの変更・削除・挿入や解析ができます。

JavaScript においては AST は難しいものではありません。AST を使ってお手軽に JavaScript をハックできるツールを作ってみましょう。

対象読者

簡単！ 専門知識不要！

自動化を進めたい人、JavaScript をハックするようなツールを書きたい、Babel プラグインを作りたいという人向けです。

あまり高度なことは書かないようにしています。コンパイラ関係の本はゴツいものばかりなので、お手軽な同人誌として書いてみたかったのが筆者のモチベーションです。

ただし JavaScript 自体については説明をしません。筆者が書いた本ですが「最新 JavaScript 開発 ES2017 対応モダンプログラミング (技術書典シリーズ (NextPublishing))」http://amzn.to/2xTUxnz という本では ECMAScript 2017 について詳しく解説しています。

JavaScript AST がなぜ簡単なのか？

本来の AST はコンパイラの内部表現に過ぎないため言語利用者の大半には縁のないものですが、JavaScript は他の処理系とは違う歴史を持っているため AST が身近です。それはブラウザの互換性との戦いの歴史から、トランスパイル、つまりソースコードをソースコードに変換するのが一般的だという特殊な事情です。トランスパイルで有名な Babel は、ECMAScript2017 (JavaScript の最新言語仕様) などをウェブブラウザ上で問題なく動作するように変換します。AST はその過程で大活躍するのです。プラグインで自由に AST を操作できますし、Babel はトランスパイラというツールというだけではなく、便利なライブラリ群でもあるのです。

Babel 以外にも有力な AST エコシステムがありますが、もともと ESTree という標準仕様があるため、Babel に限らず JavaScript AST エコシステムでは知識や経験が使い回しできます。そういった手厚い AST のエコシステムのおかげで、他の言語では考えられないくらい簡単に AST を使えて、ソースコードの解析・加工などができてしまうのです。

ところが、AST というと「難しい」という印象をお持ちの方も多いのでは無いでしょうか？ AST はとても可能性を秘めているというのに、ほとんどの JavaScript プログラマが AST を触ったことがないのです。分厚いコンパイラの本を読む必要もありません。是非本書を読んでカジュアルに JavaScript をハックしてみましょう。

AST でできること

AST を使って何ができるでしょうか？ AST を使うとソースコードの解析や加工ができます。対象のソースコードに手を付けずにハックできます。

デバッグやテストに便利なツールを作る

第 2 章では、対象のソースコードを一切変更せずに動的に依存性注入 (Dependency Injection) するというハックを、たったの 100 行程度で実装しています。

これにより、ユニットテストに向いていないソースコードの一部分を切り出してテストコードを書いたりモックを注入したりできます。テストの無いソースコードにテストを導入するのは一般的には面倒ですが、動的にソースコードを書き換えれば雑に一部を切り取って簡単にテストを順次追加していけるのです。

ソースコードの整形

JavaScript では、ESlint[*1] や prettier というソースコードの整形ツールがよく使われますがこれらにも AST が使われます。

コメントの活用

AST ではコメントを簡単に取得できるため、コメントをさまざまな目的に活用できます。JavaDoc のような、コメントに関数やクラスの仕様ドキュメントを書く風習ですが、そういったツールも JavaScript AST を使えば簡単に作れます。付録 A で軽く触れています。

ソースコードの静的解析や最適化

テストのカバレッジの取得や、lint ツールなどさまざまな静的解析ツールには一般的に AST が使われています。AST のエコシステムの中にはソースコードの静的解析をしてくれるもの、支援するものなどもあります。JavaScript が小規模の目的にしか使えないおもちゃだった時代はとっくの昔に終わりました。カジュアルさを残しつつも、バグを減らすための仕組みを使いましょう。

第 4 章では 50 行以内でできるお手軽なソースコードの最適化を実践しています。さらに 50 行ほどを追加

[*1] ESLint は元々ソースコードの整形を目的としたものではありませんが、最近ではよくソースコードの整形目的にも使われています。

してさらなる最適化についても書いています。

ソースコードの難読化

静的解析や最適化の技術を応用したものですが、ウェブサーバーから JavaScript のソースコードを取得して動かすウェブブラウザの仕組み上、ソースコードそのままを配信したくないこともあります。ソースコードを読みづらくする難読化や、配信するファイルサイズを減らす為の軽量化などが可能です。

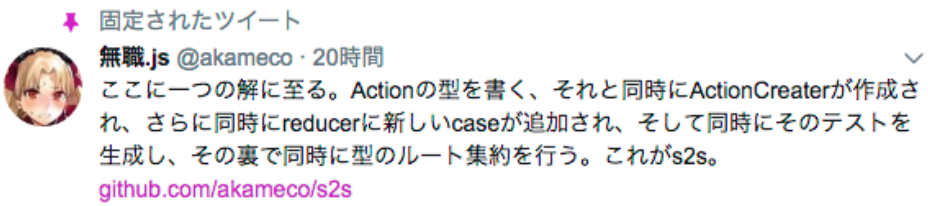
生産性の圧倒的向上

エンジニアの 3 大美徳の 1 つに「怠惰」というものがあります。コンピュータにできることは頑張って人間がしてはいけない。自動化して自分自身は楽をしようというものです。

Rails で有名になった DRY(Don't Repeat Yourself) という言葉があります。人間が同じものを何回も書くと、大なり小なりあれど必ず生産性を阻害します。繰り返しは人間が手作業でするものではありません。自動化することの大切さを DRY 思想では説いています。人力で頑張って運用するのはエンジニアとしてはとても恥ずべきことです。

できるエンジニアの生産性は 1 桁も 2 桁も違うという言葉もありますが、AST をうまく使いこなせば動作速度やソースコード管理のしやすさを犠牲にせずにそれが可能です。

たとえば、s2s[*2]という便利なツールがあります。s2s は Source to Source の略称で、ファイル変更を検知して babel プラグインで加工されたソースコードをリアルタイムにはき出すものです。s2s 作者の akameco さん[*3]は React-Redux において DRY なコーディングをしています。



https://twitter.com/akameco/status/916294919450275840

詳しくは Qiita の S2S タグを読むといいでしょう。[*4]

これに関してもアイデア次第です。何度も何度も書くコード、コピペ、そういったものから解放されましょう。s2s や AST でプログラミングは完全に進化します。

■コラム: 生産性、性能、トレードオフ

生産性と性能はトレードオフになることが多いです。

性能はアルゴリズムによっては何桁も変わります。アルゴリズムを間違えると、数十秒で終わる計算が人類が滅亡しても終わらない計算になってしまうこともあります。

https://www.youtube.com/watch?v=Q4gTV4r0zRs

[*2] https://github.com/akameco/s2s
[*3] https://twitter.com/akameco
[*4] https://qiita.com/tags/S2S

アルゴリズムの選択を間違えなかったとしても性能の最適化は、生産性を犠牲にしがちです。よく「早すぎる最適化」と呼ばれる問題です。

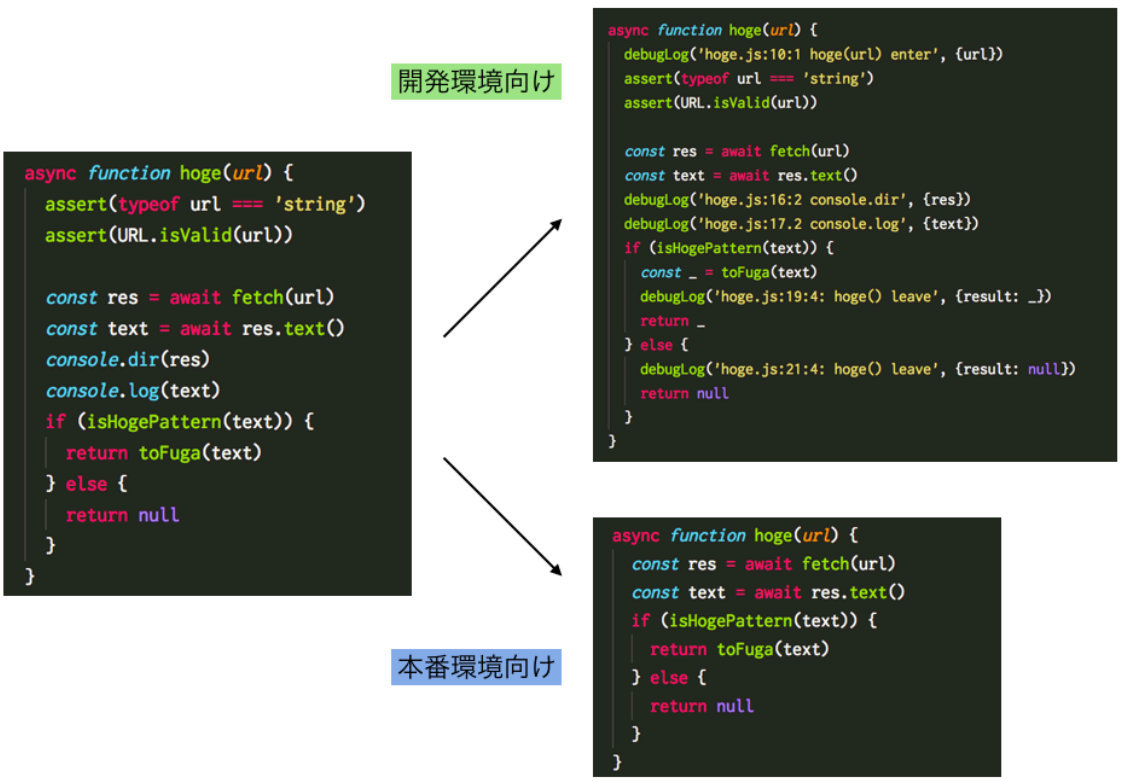
「CPU の歓声が聞こえる」ほどの天才のひとならばアセンブラでとても高性能・高効率なコードを書けるのでしょうが、生産性は最悪といってもいいです。天才以外はメンテナンスが難しく開発期間もかかるでしょう。少なくともそういう一部の特殊な人以外が書くアセンブラはコンパイラのはき出すものより性能が劣る時代です。

また、メモリ効率はいいけど動作速度が劣る、動作速度は速いけどメモリ効率が劣るみたいなトレードオフもしばしば発生します。

他にも「お金」というファクターも考える必要がありますね。開発人員のアサイン、開発期間、システムアーキテクチャなどなど、ソフトウェアの世界はトレードオフに満ちています。

ワンソースで開発向けと本番向けを分ける

ここまでに書いたことの応用ですが、ワンソースで開発環境向けと本番環境向けにそれぞれ変換することがとても簡単にできます。デバッグコードなんかを残しておくと性能面での問題やセキュリティの問題など色々発生すると思うひともいるかもしれませんが、AST でソースコードを加工すればそんな問題は無くなります。



デバッグコードや assert を手作業で除去するの面倒ですし、人間は間違いを犯す生き物です。そういうものは極力コンピュータにやらせるのがエンジニアの仕事でしょう。

■コラム: 動的変換

Babel でよくある事例は静的にトランスパイルして、その結果をウェブサーバーで配信するようなものです。

動的に処理することもできます。たとえば Node.js のような環境では `require` をハックすることで変換されたソースコードを読み込めます。動的なソースコードの変換というと `eval` のようですが `eval` のような制限はありません。ソースコードを変更する範囲においては何でもできます。うまく設計すれば、セキュリティ面でも安全にできます。

AST

ここまでで用途について説明してきました。それでは実際に AST というものがどういうものなのか見ていきましょう。

注意

AST を操作するツールは大きく分けて二系統ありますが本書では Babel/Babylon 系で説明します。もう 1 つの巨大勢力である Esprima 系について本書では触れませんが、基本的な考え方はどちらも同じです。細かい違いが色々あるだけなので応用は効くはずです。

また Babel/Babylon は、執筆の現時点 (10/13) ではバージョン 6 が安定版で、7 が開発中の beta なのですが、TypeScript に対応したり色々と美味しいので、せっかくなのでここでは 7 を前提に説明します。

```
$ npm install babel-core@next -S
```

Babel7 系は`@next` を指定する必要がありますが、もし Babel7 が安定版になっていれば不要です。

```
$ npm install babel-core -S
```

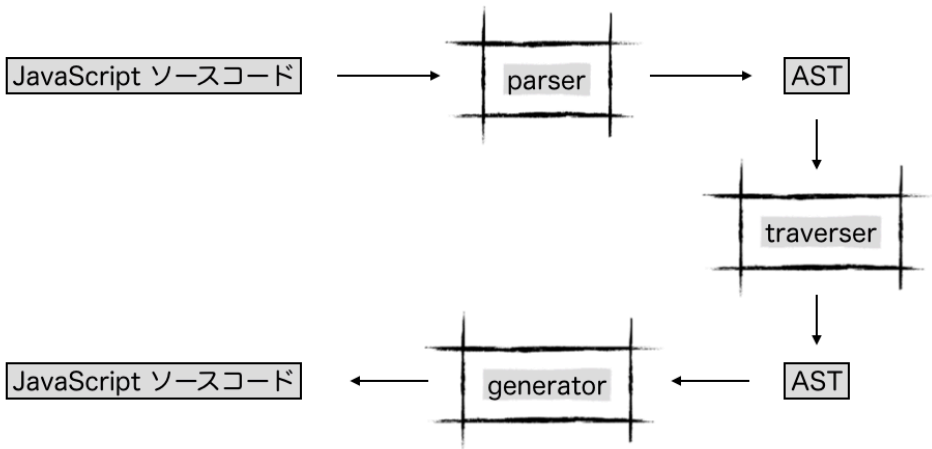
最後のオプション `-S` は `--save` の省略形で、プログラムの動作時に依存するという情報を package.json に記録するためのものです。もし開発時にしか依存しないパッケージであれば`-D` もしくは `--save-dev` を指定するといいでしょう。

ソースコードに関しては ECMAScript2017 (ECMA-262 8 版) を前提に記述しています。Node.js は v8.6.0 を対象としています。10 月中に v8 系が安定版になっているはずなので、おそらく本書が出る 10/22 には安定版になっているでしょう (なってないかもしれませんが、10 月中には安定版になっている予定です)。

本書では、これ以後@nextをつけませんので、適宜読み替えてください。あと筆者はyarnをほとんど使ったことがないので、すみませんがyarnをお使いのひとも適宜読み替えてください。

ライフサイクル

ASTを扱うツールは大まかにわけて3種類あります。パーサー・トラバーサ[*5]・ジェネレータです。



パーサー / parser

JavaScript や AltJS など[*6]のコードを解析して AST を作るツールです。Babylon や Esprima が該当します。

JavaScript の AST で楽ができるのは、このパーサーが面倒な部分をひととおり引き受けてくれるからなのです。

トラバーサ / traverser

AST を再帰的にいじるツールがトラバーサです。AST はツリー構造なので再帰的に探索するのがセオリーです。それをとても楽にするためのものです。

特に babel-traverser はとても優秀で、ある程度の静的解析まで行ってくれるという至れり尽くせりなものです。

ジェネレータ / generator

AST から JavaScript のソースコードを生成するツールがジェネレータです。babel-generator などがこれに該当します。

実際にサンプルを見てみよう

ここまでライフサイクルの説明をしてきて、それを台無しにする感じではありますが、babel-core の`transform`関数なら、パーサー・トラバーサ・ジェネレータの工程をひととおり面倒をみてくれるのでこれ

[*5] ウォーカー (walker) ともいいます。
[*6] 頑張れば他の言語や CSS とかでもいけます

を使ったソースコードを見てみましょう。

二項演算子を強制的にかけ算にしてしまうというサンプルです。20 行以内でさくっと作れます。

リスト 1: pre/babel-sample.js

```
 1: const {transform} = require('babel-core')
 2:
 3: const src = '1 + 2'
 4:
 5: const plugin = ({types: t}) => ({
 6:   visitor: {
 7:     BinaryExpression: (nodePath) => {
 8:       if (nodePath.node.operator !== '*') {
 9:         const newAst = t.binaryExpression('*', nodePath.node.left, nodePath.node.right)
10:         nodePath.replaceWith(newAst)
11:       }
12:     }
13:   }
14: })
15:
16: const {code} = transform(src, {plugins: [plugin]})
17: console.log(code) // --> 1 * 2;
```

5 行目の plugin というのはどういうことでしょうか？ じつはこのコードはれっきとした Babel のプラグインなのです。ソースコードを変換するとき、パーサー・トラバーサ・ジェネレータを個別に叩くよりは、プラグインを作って babel-core の transform を叩くのが実は一番てっとり早いためです。

6 行目の visitor はビジターパターン[*7]のビジターです。AST を再帰的に辿って、visitor オブジェクトの中に該当するラベルの関数があればそれを呼び出します。7 行目のラベルでは、BinaryExpression (二項演算子) を見つけた時に呼ばれる関数を定義しています。本書ではこれをビジター関数と呼びます。

8 行目の if 文で演算子が '*' 以外という判定をします。

9 行目で新しい AST を作成します。t.binaryExpression は 3 つの引数を渡しますが、1 つめが演算子の文字列で今回はかけ算なので'*'です。2 つめは二項演算子の左側、3 つめが右側を指します。nodePath.node.left という字面から想像できるかもしれませんが、元のソースの左側と右側を指します。[*8]

10 行目では自分自身のノードを newAst で置換します。

16 行目の transform では、Babel プラグインである plugin を使ってソースコード、今回は 1 + 2 を変換した結果 1 * 2; を返しています。

■コラム: ビジターパターン

再帰アルゴリズムを使う場合、自前で全部行うと、再帰するための処理と対象の処理 (前述の例だと BinaryExpression を置換するコード、ビジター関数) が混ざってしまいます。もちろん単純なものであればそれでも構いませんが、責務 (役割) を複数もった関数はたいてい複雑になりすぎて、メンテしづらくなります。特に AST のようなものを扱う場合いろいろ面倒なので「責務の分離」をした方が圧倒的に楽です。

[*7] https://ja.wikipedia.org/wiki/Visitor_%E3%83%91%E3%82%BF%E3%83%BC%E3%83%B3

[*8] 引数の順序を逆にすれば、当然左と右が入れ替わりますね。

ビジターパターンはそういう責務の分離をするためのデザインパターンですが、GoF という言語仕様が乏しい時代に生まれたデザインパターンで、当時は複雑な仕組みで実現するものでした。しかし、第一級関数という仕組みが言語に備わっていれば難しく考えなくても大丈夫です。

元々 JavaScript は関数型言語を作りたかった人によって生まれた言語のため、関数型言語によく見られるような第一級関数 (ファーストクラス関数)、つまり関数を自由自在にやりとりするのが当たり前にできます。再帰処理を行うトラバーサにビジター関数を渡せばいいだけです。前述の例だとビジター関数を集めたオブジェクトです。

トラバーサは AST を再帰的に辿って BinaryExpression を見つけるとさきほどのビジター関数を呼び出します。`1 + 2` という単純なコードでも複数のノードから成り立ってますが、BinaryExpression 以外は勝手に処理してくれるので、プログラマが集中したい責務だけ記述すればいいのです。

本書のサポート

https://rabbit-house.tokyo/books/javascript-ast にて本書のサポートを行います。感想や間違いの指摘などございましたら `erukiti@gmail.com` 宛までメールを送るか、https://twitter.com/erukiti 宛にメンションを飛ばすなどしていただけたら幸いです。

ソースコードの扱い

この本に登場するソースコードは CC0[*9]とします。つまり自由にソースコードを使って構いません。
また https://github.com/erukiti/ast-book-sample にてソースコードを公開しています。

[*9] https://creativecommons.org/share-your-work/public-domain/cc0

目次

第1章

AST 解説

この章では AST[*1] について解説します。前半部分では AST の取得と使い方の説明、後半部分では実際に AST を使う実践をそれぞれ説明します。

1.1 AST を実際に眺めてみよう

AST を実際に眺めるには AST Explorer http://astexplorer.net/ を使いましょう。babylon 以外の AST パーサーや他の言語にも色々対応しています。AST 関連の開発ではこれを使うのが定番です。

babylon7 を選んで、1 + 2 を食わせてみましょう。

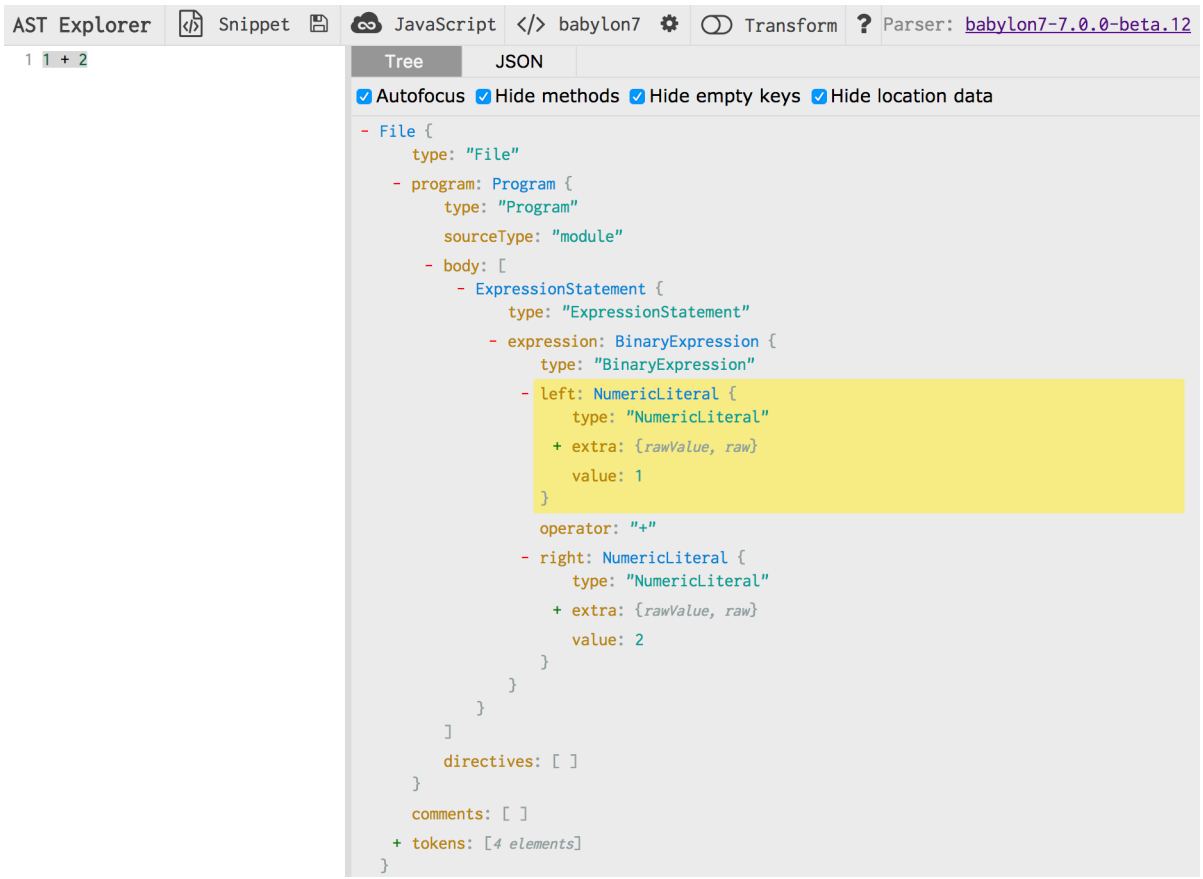


図 1.1 AST explorer

[*1] Abstract Syntax Tree の略称で日本語では抽象構文木といいます。

AST はノード[*2]が集まって出来あがったツリーです。

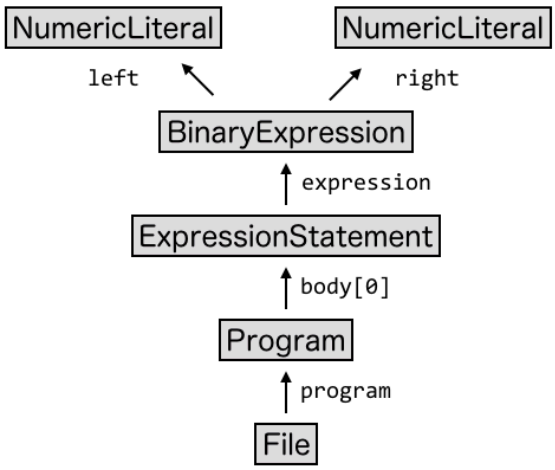


図 1.2　AST のツリー構造

図 1.2 はツリーというには貧弱ですがなんとなく木っぽいなと思ってください。ツリーの根っこは `File` というノードです。根っこから 1 つ上の幹として `Program` があります。この 2 つはいってみればコード全体を指すものです。

JavaScript のソースコードは何らかの Statement の集まりです。Statement はプログラミング言語の世界では「文」と呼ばれます。関数呼び出しや変数の書き換え、クラス定義なども文です。JavaScript の場合は上から順に文を実行[*3]します。`Program` の `body` はまさにその `Stamement` の配列で枝分かれしていきますが、今回は body の中身が 1 つです。

`ExpressionStatement` はこのプログラム唯一の文です。式そのものが文になる場合のプレースホルダーのようなものです。`ExpressionStatement` の中身は `BinaryExpression`、つまり二項演算子[*4]です。

`BinaryExpression` には 3 つの重要なプロパティがあります。演算子を指す `operator` で今回だと `'+'` という足し算を指す文字列が入っています。残りは `left` と `right` で `operator` の左と右です。今回は `1 + 2` という式なので、`left` には `1` を指すリテラルである `NumericLiteral` が入っています。リテラルというのはプログラミングの世界では、ソースコード上に直接書かれる数値や文字列のことです。

`NumericLiteral` はそこで完結していてそれ以上先を辿れない先端、つまり行き止まりです。

1.1.1　JavaScript における AST とは

JavaScript において AST エコシステムの歴史は Mozilla Firefox から始まります。Firefox のリフレクション[*5]用に Parser API[*6]が生み出されたのですが、この API が返すオブジェクトを標準化されて、皆が使える

[*2] ノードとは節 (ふし) です。プログラミングの世界ではノードというと根っこのルートノードや、葉っぱのリーフノードや、その間にあるものもすべてノードとして扱います。
[*3] 関数宣言やクラス宣言は、登録するという命令を実行しています。中身は呼び出されるまで実行されません。
[*4] a + b とか 1 * 2 とか hoge == fuga のような、「左」「演算子」「右」の構造のものを二項演算子といいます。
[*5] プログラムが自分自身の情報を読み取ったり書き換えたりすることです
[*6] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Parser_API

ESTree 仕様[*7]というものに進化しました。

リスト 1.1: Program ノードの定義

```
extend interface Program {
  sourceType: "script" | "module";
  body: [ Statement | ModuleDeclaration ];
}
```

リスト 1.1 は ESTree での Program というノードの定義が書かれています。Program には sourceType というキーで `script` か `module` という文字列が入ります。body というキーで `Statement` か `ModuleDeclaration` の配列が入ります。

さまざまな JavaScript の AST 操作ツールは基本的に ESTree ベースです。

1.1.2 Babel/Babylon (Acorn) 系と Esprima 系

JavaScript AST では大きく分けて Esprima 系と Babel/Babylon (Acorn) 系があります。大まかな構造は先ほど述べたように ESTree で共有されているのですが、細かい違いがありパーサー・トラバーサ・ジェネレータは系統を合わせる必要があります。特に Babel/Babylon では ESTree を拡張したような形となっています。

以前であれば Esprima 系が良かったようですが、今となってはどちらも成熟度が高く、どうせ世の中 Babel を使う事例ばかりなので筆者は Babel/Babylon でいいという考えに至りました。そのため本書では Babel/Babylon 系を中心に解説しますが Esprima 系でも根本的な考え方は変わらないので十分応用は効くはずです。

■コラム: Abstract Syntax Tree とは？　abstract とは？

ソースコードを内部表現に変換する過程は主に、字句解析と構文解析が必要です。

字句解析は予約語や変数名・記号などに分解する過程です。古来からある lex や、最近だと PEG (JavaScript でなら peg.js が有名です) といった字句解析に強いツールを使って、トークンという単位に分解します。

- PEG: https://ja.wikipedia.org/wiki/Parsing_Expression_Grammar
- peg.js: https://pegjs.org/

構文解析では、トークンをそれぞれの意味で分析して構文木や抽象構文木を作ります。

抽象じゃない構文木とは何でしょうか？ それはトークンを元に、括弧やリテラルの書き方など意味が同じなのに表現のことなるもの、そういったしがらみに左右されるのが構文木です。

抽象構文木 (AST) はしがらみから脱却しています。ソースコード上の意味をもとに作られているツリーなので、表記方法という具象的な情報を切り落として考えられるから抽象なのです。

コンパイラの教科書や大学の授業じゃなければ抽象じゃない構文木を意識することはないでしょう。

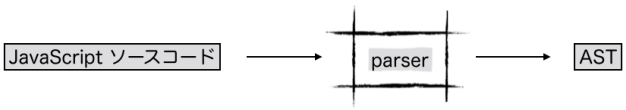
AST はしがらみから脱却はしていますが、ソースコード上の位置情報だったり、表記方法といった具

[*7] https://github.com/estree/estree

象情報も保持しているので実用上の問題もありません。もちろんそれらは無視することができます！

1.2 Babylon

さて、実際に Babel 系パーサーである Babylon[*8]を使って AST に触れてみましょう。



パーサーは JavaScript のソースコードを AST に変換してくれるものです。

```
$ npm i babylon -S
```

使い方はとても簡単です。`const ast = babylon.parser(sourceCode)` のように、引数にソースコードを入れて関数を呼び出したら AST が返ってきます。簡単ですよね！

リスト 1.2: chapter1/try-babylon7.js

```
const babylon = require('babylon')

const ast = babylon.parse('1 + 2 * (3 + 4)')
console.log(ast)

/* 結果
Node {
  type: 'File',
  start: 0,
  end: 15,
  loc:
   SourceLocation {
     start: Position { line: 1, column: 0 },
     end: Position { line: 1, column: 15 } },
  program:
   Node {
     type: 'Program',
     start: 0,
     end: 15,
     loc: SourceLocation { start: [Object], end: [Object] },
     sourceType: 'script',
     body: [ [Object] ],
     directives: [] },
  comments: [] }
*/
```

[*8] https://github.com/babel/babylon

返ってくる AST はツリー構造な Node 型オブジェクトです。Node 型自体は実質ただのプロパティ保持用のクラスで特にメソッドをもっているわけでもありません。

Node 型はかならず `type` という種別を示すプロパティがあり、`File`，`Program`，`BinaryExpression` などの文字列が入っています。

1.2.1 位置情報

`start`，`end`，`loc` というプロパティはソースコード上の位置情報を示しています。

`start`，`end` は渡したソースコードのバイト数での位置情報で、`loc` はソースコードの表示上の位置情報です。

リスト 1.3: loc

```
loc:
 SourceLocation {
   start: Position { line: 1, column: 0 },
   end: Position { line: 1, column: 15 } },
```

`line` は行番号で、1 から始まるオリジン 1 で、`column` は列でオリジン 0 です。byte 単位での位置情報よりは、line/column の方がエラー表示なんかには便利ですね。

1.2.2 子 Node の辿り方

AST はツリー構造ですが、子 Node のキーは必ずしも同じではありません。それぞれの Node の種類ごとに異なるキーで辿る必要があります。たとえば、Babel 系 AST での一番トップは File Node です。`program` というプロパティに Program Node を持っています。Program Node は、`body` に Statement や ModuleDeclaration の配列を持っています。

リスト 1.4: File Node

```
1: {
2:   "type": "File",
3:   "program": <Program Node>,
4:   "comments": []
5: }
```

リスト 1.5: Program Node

```
1: {
2:   "type": "Program",
3:   "sourceType": "script",
4:   "body": [<Statement node>, ....]
5: }
```

子 Node の持ち方には 2 パターンあります。File の program のように直接 Node 型のオブジェクトが入っているケースと、Program の body のように Node 型の配列というケースです。

traverser を使わずに自前で AST を辿る場合、それぞれのキーごとの値が Node 型か、Node 型の配列かを判別するのが手っ取り早いですが、Node の型ごとの子 Node のリストをもつという手もあります。

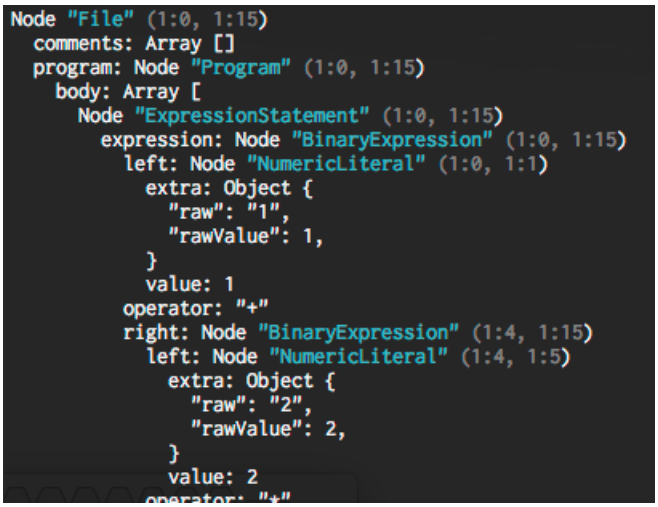
1.2.3 AST を見るお手軽な方法その 1

まずは AST を見るツールを紹介します。babel-log[*9] と ast-pretty-print[*10] です。[*11]実のところ babel-log の `log(ast)` は、`console.log(printAST(ast, true))` を呼び出すだけです。`printAST` は結果を文字列として取得できるので `console.log` を使いたくない時に便利です。目的に合わせてどちらかを選べばいいでしょう。

```
$ npm i babel-log ast-pretty-print -S
```

リスト 1.6: chapter1/babel-log.js

```
1: const babylon = require('babylon')
2: const log = require('babel-log')
3: // const printAST = require('ast-pretty-print')
4:
5: const ast = babylon.parse('1 + 2 * (3 + 4)')
6: log(ast)
7: // console.log(printAST(ast, true))
```



[*9] https://github.com/babel-utils/babel-log
[*10] https://github.com/babel-utils/ast-pretty-print
[*11] このツール、akameco さんに教えていただきました。ありがとうございます！

1.2.4 AST を見るお手軽な方法その 2

AST を見るツールをさっくり作っちゃうのも手です。その場合、JSON.sringify() を使うとお手軽なのですが、余分な情報が多いので第二引数の replacer 関数を使います。

リスト 1.7: JSON.stringify()

```
1: JSON.stringify(obj, replacer, '  ')
```

JSON.stringify は第 1 引数で指定したオブジェクトを、プロパティごとに再帰的にたどって JSON 化するのですが、第 2 引数の replacer が関数ならば、プロパティをどう扱うかは毎回 replacer 関数にお伺いを立ててくれます。

replacer 関数にはプロパティのキーである key と中身の value が渡ってきます。戻り値によって表示が変わるのですが、undefined を返せばそのプロパティは無かったことになります。元の値を返す、あるいは別の値を返すとその値が表示されます。

replacer 関数に null を渡した場合 JSON.stringify(obj, null, ' ') と比較してみると挙動について理解がすすむでしょう。ちなみに第 3 引数は渡した文字列でインデントをしてくれます。これが無いとインデントなしで詰め込まれた 1 行の JSON が返ってくるので使いづらいです。

リスト 1.8: chapter1/print-ast-outline.js

```
 1: const babylon = require('babylon')
 2:
 3: const ast = babylon.parse('1 + 2 * (3 + 4)')
 4:
 5: const isNode = obj => {
 6:   // Node もしくは Node の配列は必ず object 型です。
 7:   if (typeof obj !== 'object') {
 8:     return false
 9:   }
10:
11:   // 配列の中に Node が含まれていれば、配列自体を Node 型と判定します。
12:   if (Array.isArray(obj)) {
13:     return obj.find(v => isNode(v)) !== undefined
14:   }
15:
16:   return obj && 'constructor' in obj && obj.constructor.name === 'Node'
17: }
18:
19: const replacer = (key, value) => {
20:   if (!key || key === 'type' || isNode(value)) {
21:     return value
22:   }
23:
24:   return undefined
25: }
26:
27: console.log(JSON.stringify(ast, replacer, '  '))
```

これで、AST のアウトラインが見えるようになりました。

■コラム: Node 型を判定する方法

Babylon では残念ながら Node 型が export されてないので instanceof で判定できません。

constructor.name が'Node'かどうかを見ています。

別の手段としてはダックタイピング的に、type が string だったり、start, end が数値、loc などを見てオブジェクトの中身が AST のノードっぽいかを判定するというのもあります。この方法をとるときは、isNodeLike みたいな関数名を付けられることがあります。中身が Node っぽければ厳密な型を気にしないという考え方ですね。

`1 + 2 * (3 + 4)` をリスト 1.2 で生成した AST ならば以下のような出力になります。

リスト 1.9: chapter1/print-ast-outline.result.json

```
{
    "type": "File",
    "program": {
      "type": "Program",
      "body": [
        {
          "type": "ExpressionStatement",
          "expression": {
            "type": "BinaryExpression",
            "left": {
              "type": "NumericLiteral"
            },
            "right": {
              "type": "BinaryExpression",
              "left": {
                "type": "NumericLiteral"
              },
              "right": {
                "type": "BinaryExpression",
                "left": {
                  "type": "NumericLiteral"
                },
                "right": {
                  "type": "NumericLiteral"
                }
              }
            }
          }
        }
      ]
    }
  }
```

自作すると何が嬉しいかというと、色々いじって出力をカスタマイズできます！

■コラム: 再帰処理

再帰処理とは関数が自分自身を呼び出すことです。直接呼び出さずに他の関数を経由するパターンも

あります。

再帰処理に向いたデータ構造を再帰無しで書こうとすると、キューやスタックといった別の高度な仕掛けを使うことになるでしょう。それでは再帰処理に向いたデータ構造とは何かというと再帰データ構造です。ツリー構造、リスト構造などが該当します。これらに共通するのは自分 (Node) が別の Node への参照をもっていることです。

JavaScript で身近なツリー構造には、JSON やプレーンなオブジェクトがあります。

リスト 1.10: chapter1/tree-recursive.js

```
 1: const obj = {
 2:   hoge: {
 3:     fuga: [1, 2, 3]
 4:   },
 5:   piyo: 'ぴよ',
 6:   foo: {
 7:     bar: {
 8:       baz: null
 9:     }
10:   }
11: }
12:
13: const objToString = (node, indent = 0) => {
14:   const leading = ' '.repeat(indent)
15:
16:   if (typeof node === 'object' && node) {
17:     return Object.keys(node).map(key => {
18:       return `${leading}${key}:\n` +
19:              `${objToString(node[key], indent + 2)}`
20:     }).join('\n')
21:   }
22:
23:   return `${leading}${node}`
24: }
25:
26: console.log(objToString(obj))
```

objToString が最初に呼び出される時は obj を引数に渡して、indent を省略しているので indent は 0 です。その時点で node の中身は obj そのままなので、処理は 17 行目に渡ります。Object.keys(node) では hoge, piyo, foo の配列が得られるので 19 行目では objToString(obj.hoge, 2) が呼び出されます。

今度は node は obj.hoge を指しているのでまたオブジェクトです。先ほどと同じ流れで 19 行目で objToString(obj.hoge.fuga, 4) が呼び出されます。

obj.hoge.fuga は配列なのですが、配列はオブジェクトです。これまでと同じように 19 行目で objToString(obj.hoge.fuga[0], 6) が呼び出されます。

obj.hoge.fuga[0] は数値ですが、インデントされた文字列としての’ 1’を返します。これを配列の個数分繰り返して obj.hoge.fuga が処理した文字列が返ります。

obj.hoge には fuga 以外は無いので obj.hoge.fuga が返した文字列を加工した文字列を返します。

ここまでで、hoge の処理は完了です。次は piyo に映って同じ流れで処理して、最後に foo を処理します。

AST ではツリー構造を相手にするため、この流れを頭に入れておく必要があります。コツはデバッガ

を使うなり机上デバッグなり、手段はなんでもいいですが、実際にどういう引数で objToString が呼び出されるか、return するときは何を返すのかを観察することです。たとえば、VSCode のデバッガでステップ実行をすると一目瞭然ですね。

1.2.5 AST の調べ方

Babylon 公式の仕様書 https://github.com/babel/babylon/blob/master/ast/spec.md か、babel-types の定義 https://github.com/babel/babel/tree/master/packages/babel-types/src/definitions を読むのが一番確実です。もちろん AST explorer を活用して調べるのも手です。[*12]

1.3 実際に AST を使ってみよう

AST について説明してきました。リスト 1.9 にでてきた種類の Node だけ知っていれば四則演算なら簡単に行えます。作ってみましょう。

1.3.1 トラバーサを自作してみよう

今回はトラバーサ、つまり再帰的に Node を辿って AST をいじるためのツールも自作してみましょう。

リスト 1.11: chapter1/traverser.js

```
 1: const code = '1 + 2 * (3 + 4)'
 2:
 3: // その Node に対応するソースソースコードを取得するヘルパー関数
 4: const getCode = node => code.substr(node.start, node.end - node.start)
 5:
 6: const traverser = (node, exitVisitor, indent = 0) => {
 7:   console.log(`${' '.repeat(indent)}enter: ${node.type} '${getCode(node)}'`)
 8:   if (!(node.type in exitVisitor)) {
 9:     console.error(`unknown type ${node.type}`)
10:     console.log(JSON.stringify(node, null, '  '))
11:     process.exit(1)
12:   }
13:
14:   const res = {}
15:   // Node の中身を舐める
16:   Object.keys(node).forEach(key => {
17:     // Node 型じゃないのでたどらない
18:     if (!isNode(node[key])) {
19:       return
20:     }
21:
22:     if (Array.isArray(node[key])) {
23:       // Node 型の配列なのでそれぞれ再帰する
24:       res[key] = node[key].map(v => traverser(v, exitVisitor, indent + 2))
25:     } else {
26:       res[key] = traverser(node[key], exitVisitor, indent + 2)
```

[*12] いつか筆者が表を書くかもしれません

```
27:     }
28:   })
29:
30:   console.log(`${' '.repeat(indent)}exit:  ${node.type} '${getCode(node)}'`)
31:   //ビジター関数を呼び出してその結果を返す
32:   return exitVisitor[node.type](node, res, indent)
33: }
```

exitVisitor は対象の Node から出るときに呼び出されるコールバック関数 callback であるビジター関数のつまったオブジェクトです。このオブジェクトに登録されていない Node を見つけてしまった場合は、エラー終了します。

16 行目から 28 行目では traverser の再帰処理を行っています。リスト 1.8 で作った isNode 関数を使って node のそれぞれのプロパティが Node 型か Node 型の配列なら再帰的に探索するのです。

32 行目ではビジター関数を呼び出した結果を返しています。

本来 traverser には入るとき・出るとき両方に対応してるものですが、今回は出るときの処理だけで済むので、入る時の処理は省略しています。

■コラム: 車輪の再発明

車輪の再発明、つまりすでにあるものを新しく作ることを嫌う人はいますが、筆者は少なくとも習得過程では有益だと考えます。ソースコードリーディングや改造もいいですが、一からロジックを組み立てる練習をしておくと、中身の理解が捗ります。

既存のトラバーサに機能的な不満があるなら、改造や作り直しをすることもあるでしょう。そのときに何を優先したいのか、コストの兼ね合いでどうするか考えましょう。

1.3.2 トラバーサから呼び出すためのビジター関数オブジェクトを書いてみよう

リスト 1.12: chapter1/visitor.js

```
 1: const exitVisitor = {
 2:   File: (node, res) => res.program,
 3:   Program: (node, res) => res.body,
 4:   ExpressionStatement: (node, res) => {
 5:     const expr = node.expression
 6:     return `${getCode(node)} = ${res.expression}`
 7:   },
 8:   BinaryExpression: (node, res, indent) => {
 9:     console.log(`${' '.repeat(indent)} ${res.left} ${node.operator} ${res.right}`)
10:     const {left, right} = res
11:     switch (node.operator) {
12:       case '+': return left + right
13:       case '*': return left * right
14:       case '-': return left - right
15:       case '/': return left / right
16:       case '%': return left % right
17:       default: throw new Error('対応してない二項演算子')
```

```
18:     }
19:   },
20:   NumericLiteral: (node, res, indent) => {
21:     console.log(`${' '.repeat(indent)}  value: ${node.value}`)
22:     return node.value
23:   }
24: }
```

リスト 1.12 はビジター関数オブジェクトですが、一番最初に呼び出されるビジター関数は子 Node が存在しない `NumericLiteral` です。それ以外の Node の場合は traverser が再帰処理をしているので、子 Node がすべて解決したあと[*13]になるので、`BinaryExpression` が出るときはすでに子 Node の `res.left`, `res.right` には数値が入っているのでそのまま計算できます。

リスト 1.13: chapter1/ast-calc.result.txt

```
enter: File '1 + 2 * (3 + 4)'
enter: Program '1 + 2 * (3 + 4)'
  enter: ExpressionStatement '1 + 2 * (3 + 4)'
    enter: BinaryExpression '1 + 2 * (3 + 4)'
      enter: NumericLiteral '1'
      exit:  NumericLiteral '1'
        value: 1
      enter: BinaryExpression '2 * (3 + 4)'
        enter: NumericLiteral '2'
        exit:  NumericLiteral '2'
          value: 2
        enter: BinaryExpression '3 + 4'
          enter: NumericLiteral '3'
          exit:  NumericLiteral '3'
            value: 3
          enter: NumericLiteral '4'
          exit:  NumericLiteral '4'
            value: 4
        exit:  BinaryExpression '3 + 4'
         3 + 4
      exit:  BinaryExpression '2 * (3 + 4)'
       2 * 7
    exit:  BinaryExpression '1 + 2 * (3 + 4)'
     1 + 14
  exit:  ExpressionStatement '1 + 2 * (3 + 4)'
exit:  Program '1 + 2 * (3 + 4)'
exit:  File '1 + 2 * (3 + 4)'

1 + 2 * (3 + 4) = 15
```

リスト 1.13 の、enter と exit を良く見ておいてください。どういう順番で処理が行われるかが分かるはずです。

1.3.3 完成版

完成版では計算するコードは引数で渡すようにしています。

[*13] 深さ優先探索と言います。

```
$ node ast-calc.js '1 + 2 * (3 + 4)'
```

のように起動してみてください。

リスト 1.14: chapter1/ast-calc.js

```
 1: const {parse} = require('babylon')
 2:
 3: const code = process.argv.slice(2).join(' ')
 4:
 5: const isNode = obj => {
 6:   if (typeof obj !== 'object') {
 7:     return false
 8:   }
 9:
10:   if (Array.isArray(obj)) {
11:     return obj.find(v => isNode(v)) !== undefined
12:   }
13:
14:   while (obj && 'constructor' in obj) {
15:     if (obj.constructor.name === 'Node') {
16:       return true
17:     }
18:     obj = Object.getPrototypeOf(obj)
19:   }
20:   return false
21: }
22:
23: const getCode = node => code.substr(node.start, node.end - node.start)
24:
25: const traverser = (node, exitVisitor, indent = 0) => {
26:   console.log(`${' '.repeat(indent)}enter: ${node.type} '${getCode(node)}'`)
27:   if (!(node.type in exitVisitor)) {
28:     console.error(`unknown type ${node.type}`)
29:     console.log(JSON.stringify(node, null, '  '))
30:     process.exit(1)
31:   }
32:
33:   const res = {}
34:   Object.keys(node).forEach(key => {
35:     if (!isNode(node[key])) {
36:       return
37:     }
38:
39:     if (Array.isArray(node[key])) {
40:       res[key] = node[key].map(v => traverser(v, exitVisitor, indent + 2))
41:     } else {
42:       res[key] = traverser(node[key], exitVisitor, indent + 2)
43:     }
44:   })
45:
46:   console.log(`${' '.repeat(indent)}exit:  ${node.type} '${getCode(node)}'`)
47:   return exitVisitor[node.type](node, res, indent)
48: }
49:
50: const exitVisitor = {
51:   File: (node, res) => res.program,
52:   Program: (node, res) => res.body,
53:   ExpressionStatement: (node, res) => {
```

```
54:     const expr = node.expression
55:     return `${getCode(node)} = ${res.expression}`
56:   },
57:   BinaryExpression: (node, res, indent) => {
58:     console.log(`${' '.repeat(indent)} ${res.left} ${node.operator} ${res.right}`)
59:     const {left, right} = res
60:     switch (node.operator) {
61:       case '+': return left + right
62:       case '*': return left * right
63:       case '-': return left - right
64:       case '/': return left / right
65:       case '%': return left % right
66:       default: throw new Error('対応してない二項演算子')
67:     }
68:   },
69:   NumericLiteral: (node, res, indent) => {
70:     console.log(`${' '.repeat(indent)}  value: ${node.value}`)
71:     return node.value
72:   }
73: }
74:
75: const results = traverser(parse(code), exitVisitor)
76: console.log('')
77: results.forEach(result => console.log(result))
```

第2章

Babel 系エコシステム弾丸ツアー

第 1 章 では実際に Babylon を使って AST に触れてみました。AST エコスシステムはパーサーだけではありません。Babel 系ツールをひととおり軽く説明します。もっとも大半のツールはとてもシンプルなんで、ほとんど説明することはありません。

2.1 babel-core

序章でも使った babel-core[*1] は名前のとおり Babel の機能の本体です。パーサー・トラバーサ・ジェネレータの全部を含んだものです。ソースコードを渡せば Babel プラグインを読み込んで実際にコードを変換してくれます。このとき、AST とソースコード両方を生成するのでさらに色々手を加えることもできます。

```
$ npm i babel-core -S
```

リスト 2.1: chapter2/try-babel-core.js

```
1: const {transform} = require('babel-core')
2:
3: const sourceCode = '1 + 2'
4: const opts = {plugins: []}
5:
6: const {code, ast, map} = transform(sourceCode, opts)
7: // code: 変換後のソースコード
8: // ast: 変換後の AST
9: // map: ソースマップ
```

`transform` には、ファイルから読み込む `transformFile`, `transformFileSync`, `transfomFromAst` などの亜種もあります。くわしくは https://babeljs.io/docs/usage/api/ をご覧ください。

`transform` では AST の加工はプラグイン任せです。リスト 2.1 ではプラグインを指定していないため、何も加工されません[*2]が、序章に書いたサンプルのようにさっくりプラグインを作ってちょちょいとソースコードを変換する用途か、既存のプラグインを使うのにとても便利です。

[*1] https://babeljs.io/docs/core-packages/

[*2] コード自体は加工されませんが、スタイルは変わります。これはオプション次第ですがソースコードのスタイルが変わる可能性があります。

2.1.1 .babelrc

Babel では設定を.babelrc というファイルに JSON 形式で記述するようになっています。

リスト 2.2: .babelrc

```
{
    "presets": ["env", "power-assert"]
}
```

たとえば、このような設定ファイルがよく使われます。このオプションは transform の引数でも同じです。

表 2.1 主なオプション

オプション名	デフォルト	中身
ast	true	AST を出力するか？
babelrc	true	既にある.babelrc を読み込むかどうか。.babelrc を無視したければ false を指定
code	true	code を出力するか？ (AST のみなら false)
generatorOpts	{}	babel-generator の generate 関数に渡すオプション
inputSourceMap	null	ソースマップを明示的に指定する
parseOpts	{}	Babylon の parse に渡すオプション
plugins	[]	プラグインを配列で指定
presets	[]	プリセットを配列で指定
sourceMaps	false	ソースマップを出力するかどうか？ 'inline' だとソース埋め込み

オプションを全部書くと切りがないので、詳しくは https://babeljs.io/docs/usage/api/ をご覧ください。

リスト 2.3: プラグインオプション

```
1: const plugins = [
2:   ['plugin-A', {hoge: 1}],
3:   'plugin-B',
4:   ['plugin-C']
5: ]
```

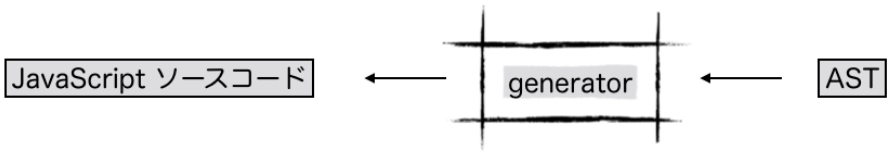
plugins という大本の配列があって、3 パターンの形式でプラグインを指定します。最初の plugin-A はオプションを指定する唯一の方法です。オプションを指定しなければ plugin-B の形式が良いでしょう。

AST 操作ツールの開発ではあまり.babelrc を意識しないとは思いますが、覚えておいて損はありません。

2.2 babel-generator

babel-generator[*3]は Babel 系のジェネレータ、つまり AST からソースコードを生成するツールです。

[*3] https://www.npmjs.com/package/babel-generator




```
$ npm i babel-generator -S
```

リスト 2.4: chapter2/try-babel-generator.js

```
 1: const generate = require('babel-generator').default
 2:
 3: const ast = {
 4:   type: 'ExpressionStatement',
 5:   expression: {
 6:     type: 'BinaryExpression',
 7:     operator: '+',
 8:     left: {type: 'NumericLiteral', value: 1},
 9:     right: {type: 'NumericLiteral', value: 2}
10:   }
11: }
12:
13: const {code, map} = generate(ast)
14: console.log(code)
15: // 1 + 2;
```

`generate` に指定する AST は、Babylon でパースしたもの、babel-core の `transform` で出した AST や、babel-types で生成した AST などが使えます。

2.3 prettier

prettier は JavaScript を初めとした色々な言語に対応したソースコードの整形ツールです。本来の使い方はソースコードを整形してそのままソースコードを書き戻すものですが、Babel 系 AST に元々対応しているためジェネレータとしても使えます。[*4]

```
$ npm i prettier -S
```

リスト 2.5: chapter2/try-prettier.js

[*4] 本来は別に Babel ファミリーというわけではないです。https://github.com/prettier/prettier

```
 1: const {transform} = require('babel-core')
 2: const prettier = require('prettier')
 3:
 4: const src = 'console.log("hoge");'
 5:
 6: const code = prettier.format(src, {
 7:   semi: false,
 8:   singleQuote: true,
 9:   parser(text) {
10:     const {ast} = transform(text, {plugins: []})
11:     return ast
12:   }
13: })
14:
15: console.log(code)
16: // --> console.log('hoge')
```

リスト 2.5 ではソースコードからソースコードに変換しているだけですが、parser 関数は AST さえ返せばそれがコードに変換されるものなので、引数とは無関係な AST を返しても大丈夫です。

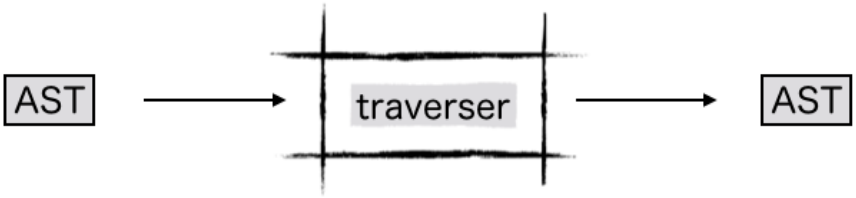
リスト 2.6: chapter2/try-prettier-with-dummy.js

```
const prettier = require('prettier')
const {transformFromAst} = require('babel-core')

const rawAst = {
  type: 'Program',
  body: [{
    type: 'ExpressionStatement',
    expression: {
      type: 'BinaryExpression',
      operator: '+',
      left: {type: 'NumericLiteral', value: 1, extra: {raw: '1'}},
      right: {type: 'NumericLiteral', value: 2, extra: {raw: '2'}},
    }
  }]
}

const {ast} = transformFromAst(rawAst)

const code = prettier.format('dummy', {
  semi: false,
  singleQuote: true,
  parser(text) {
    return ast
  }
})

console.log(code)
// 1 + 2
```

元々がソースコードを整形するツールなので少し使い方が面倒ではありますが、babel-generator の代わりにいかがでしょうか。

2.4 babel-traverse

babel-traverse[*5]は Babel 系のトラバーサ、つまり AST を再帰的に解析したり加工したりできるツールです。



前の章では説明のために自作トラバーサを使いましたが、再起処理でこうやれば処理ができるという最低限のサンプルに過ぎません。ちゃんとしたトラバーサには色々と便利な機能があるので普通は既存のトラバーサを使います。

```
$ npm i babel-traverse -S
```

やはり使い方は簡単です。

リスト 2.7: chapter2/try-babel-traverse.js

```
 1: const babylon = require('babylon')
 2: const traverse = require('babel-traverse').default
 3:
 4: const ast = babylon.parse('1 + 2')
 5:
 6: const visitor = {
 7:   BinaryExpression: (nodePath) => {
 8:     console.log(nodePath.node) // --> Node {....}
 9:   }
10: }
11:
12: traverse(ast, visitor)
```

7 行目では引数に NodePath 型が入っています。詳しくは後述しますが NodePath 型は、Node 型だけだと足りない情報を色々と補ってくれるものです。[*6]

2.4.1 visitor

`visitor` はビジター関数の詰まったオブジェクトです。

[*5] https://www.npmjs.com/package/babel-traverse

[*6] Node 型は Babylon で内部定義された型で、NodePath は babel-traverse が定義する型です。NodePath は export されているので参照できます。

リスト 2.8: chapter2/visitor.js

```
 1: const {parse} = require('babylon')
 2: const traverse = require('babel-traverse').default
 3:
 4: const src = '1 + 2'
 5:
 6: const ast = parse(src)
 7: const visitor = {
 8:   enter(nodePath) {
 9:     console.log(`enter: ${nodePath.type}`)
10:   },
11:   exit(nodePath) {
12:     console.log(`exit:  ${nodePath.type}`)
13:   },
14:   NumericLiteral: {
15:     enter(nodePath) {
16:       console.log('NumericLiteral enter')
17:     },
18:     exit(nodePath) {
19:       console.log('NumericLiteral exit')
20:     }
21:   },
22:   BinaryExpression(nodePath) {
23:     console.log('BinaryExpression enter')
24:   },
25: }
26:
27: traverse(ast, visitor)
```

ビジター関数はある Node の処理開始時の `enter` と終了時の `exit` をそれぞれ指定できます。

まず `visitor` 直下にある `enter/exit` は、全 Node での、入ったときと出るときに呼び出されます。ただしこれは強力ですが Babel プラグインにはメインの `visitor` で指定するとエラーになるという制約[*7]があります。

`enter/exit` 以外の場合は、`NumericLiteral` や `BinaryExpression` のような Node 型の type[*8]を指定します。このラベル指定のとき、中身がオブジェクトであれば `enter`, `exit` のどちらかを指定してビジター関数を登録します。中身がそのまま関数であれば、`enter` として扱われます。[*9]

リスト 2.9: chapter2/visitor.result.txt

```
 1: enter: Program
 2: enter: ExpressionStatement
 3: enter: BinaryExpression
 4: BinaryExpression enter
 5: enter: NumericLiteral
 6: NumericLiteral enter
 7: exit:  NumericLiteral
 8: NumericLiteral exit
 9: enter: NumericLiteral
10: NumericLiteral enter
11: exit:  NumericLiteral
```

[*7] `nodePath.traverse(innerVisitor)` という呼び出しをすると `enter`, `exit` も使えます。ちょっとした抜け道ですね。

[*8] じつは他にも指定ができますがここでは置いておきましょう。

[*9] 言ってみれば、ちょっとした shorthand ですね

```
12: NumericLiteral exit
13: exit:  BinaryExpression
14: exit:  ExpressionStatement
15: exit:  Program
```

さて、visitor 関数は第 1 引数が NodePath 型だと説明しましたが、もう少し詳しく中を見てみましょう。NodePath 側は循環参照要素も多くそのまま見ようとするとやたら長いので、ダイジェストで表示できるコードを作ってみましょう。

リスト 2.10: chapter2/nodepath-inspector.js

```
 1: const {parse} = require('babylon')
 2: const traverse = require('babel-traverse').default
 3:
 4: const src = 'const a = 1; hoge(a)'
 5:
 6: const ast = parse(src)
 7:
 8: const inspectProps = prop => {
 9:   const propType = typeof prop
10:
11:   if (propType === 'string') {
12:     return `'${prop}'`
13:   } else if (propType !== 'object' || !prop) {
14:     return prop
15:   } else if (prop.constructor.name === 'Object') {
16:     return JSON.stringify(prop)
17:   } else if (prop.constructor.name === 'Array') {
18:     return `[${prop.map(value => inspectProps(value)).join(' ,')}]`
19:   } else {
20:     if ('type' in prop) {
21:       return `Object(${prop.constructor.name}) ${prop.type}`
22:     } else {
23:       return `Object(${prop.constructor.name})`
24:     }
25:   }
26: }
27:
28: const visitor = {
29:   // enter(nodePath) {
30:   // ひとまず CallExpression だけ見てますが、enter とかを指定してみてもいいでしょう。
31:   CallExpression(nodePath) {
32:     console.log(`enter ${nodePath.type}`)
33:     Object.keys(nodePath).sort().forEach(key => {
34:       console.log(`  ${key}:  ${inspectProps(nodePath[key])}`)
35:     })
36:   },
37: }
38:
39: traverse(ast, visitor)
40:
41: /* 結果
42: enter CallExpression
43:   container:  Object(Node) ExpressionStatement
44:   context:  Object(TraversalContext)
45:   contexts:  [Object(TraversalContext)]
46:   data:  {}
47:   hub:  undefined
```

```
48:   inList:  false
49:   key:  'expression'
50:   listKey:  undefined
51:   node:  Object(Node) CallExpression
52:   opts:  {"CallExpression":{"enter":[null]},"_exploded":true,"_verified":true}
53:   parent:  Object(Node) ExpressionStatement
54:   parentKey:  'expression'
55:   parentPath:  Object(NodePath) ExpressionStatement
56:   removed:  false
57:   scope:  Object(Scope)
58:   shouldSkip:  false
59:   shouldStop:  false
60:   skipKeys:  {}
61:   state:  undefined
62:   type:  'CallExpression'
63:   typeAnnotation:  null
64: */
```

NodePath 型の変数 nodePath から Node 型を取得するには nodePath.node です。visitor 関数を書くときに最も使うでしょう。

CallExpression の親は ExpressionStatement です。親を Node 型で取得するなら nodePath.parent を使い、NodePath 型でならば nodePath.parentPath を使います。

親から見た自分のキーは nodePath.parentKey で取得できます。今回のキーは'expression'ですが、BinaryExpression の子 Node であれば、left か right が入っているでしょう。

CallExpression には、呼び出される関数やメソッドを指す callee というメンバーが含まれていて、Node 型で取得するなら nodePath.node.callee でいいのですが、NodePath 型を取得するなら nodePath.get メソッドを使います。nodePath.get('callee') で、callee の NodePath が取得できます。

Block や Program のように配列で管理されている中の一部であれば、inList が true になり、key が配列のインデックスになります。

表 2.2　NodePath 型

プロパティ	中身
type	Node の type と同じものが文字列として入っている
node	Node オブジェクト
parent	自分の親の Node オブジェクト
parentPath	自分の親の NodePath オブジェクト
parentkey	自分の親から自分にアクセスするためのキー
get(pathname)	pathname で指定した名前の NodePath を取得する
inList	自分が配列管理されているか？ (true or false)
key	inList が true なら配列での添え字 (整数)

大体は表 2.2 がよく使われるものです。

他にとても強力なものとしては scope の指す Scope 型があります。その Node が属するスコープにおいてどの変数が宣言されているか、その変数が実際に使われているか、再代入が発生しているかどうかなさまざまな静的解析の情報を得られます。詳しくは第 5 章で説明します。

2.5 babel-types

babel-types[*10]は、AST を生成したり、Node の種別を判別できる便利なヘルパーのライブラリです。

```
$ npm i babel-types -S
```

リスト 2.11: chapter2/try-babel-types.1.js

```
1: const t = require('babel-types')
2: const generate = require('babel-generator').default
3:
4: const ast = t.binaryExpression('*', t.numericLiteral(1), t.numericLiteral(2))
5: const {code} = generate(ast)
6: console.log(code) // --> 1 * 2
```

`t.binaryExpression` や `t.numericLiteral` はなんとなく分かると思いますが、Node 型の type の先頭を小文字にしたものです。実際には https://babeljs.io/docs/core-packages/babel-types/ を読むといいですが、読まなくてもだいたい作りたい type さえわかれば先頭を小文字にするだけでいけるでしょう。

これで生成した AST は Node 型ではなく単なるプレーンなオブジェクトです。babel-traverse の AST 変換・追加メソッドや、babel-generator に喰わせることはできますが、prettier には対応していません。

`t.numericLiteral` などにはすべて判定用の関数とバリデーション用の関数が用意されています。

`NumericLiteral` の判定ならば `t.isNumericLiteral` です。`BinaryExpression` なら `isBinaryExpression` ですね。

さらに `NumericLiteral` は `Literal` でもあるため[*11]、`t.isLiteral` で判定することもできます。

リスト 2.12: chapter2/try-babel-types.2.js

```
 1: const {parse} = require('babylon')
 2: const traverse = require('babel-traverse').default
 3: const t = require('babel-types')
 4:
 5: const src = '1 + 2'
 6: const ast = parse(src)
 7:
 8: traverse(ast, {
 9:   BinaryExpression: (nodePath) => {
10:     const {left, right, operator} = nodePath.node
11:     if (t.isLiteral(left) && t.isLiteral(right)) {
12:       console.log(eval(`${left.value} ${operator} ${right.value}`))
13:     }
14:   }
15: })
16: // --> 3
```

[*10] https://babeljs.io/docs/core-packages/babel-types/

[*11] エイリアス (alias) といいます

実際に Babel ではどういう type が存在しているかは AST explorer にソースコードを喰わせるのもいいですが、babel-types のソースコードを見るのが一番早いです。

https://github.com/babel/babel/tree/master/packages/babel-types/src/definitions にはそれぞれの種別ごとの定義が書かれています。

表 2.3　type 抜粋

type 名	内容	例	エイリアス
AssignmentExpression	代入式	`left = right`	Expression
BinaryExpression	二項演算子	`left * right`	Expression
Identifier	変数・プロパティなど何かの名前	`let hoge` の `hoge`	Expression など
StringLiteral	文字列リテラル	`"hoge"` や `'fuga'`	Literal, Expression など
NumericLiteral	数値リテラル	`42` や `10.4`	Literal, Expression など

2.6　参照リンク

Babel 関連やるときは大体、babeljs.io か github のリポジトリとにらめっこになります。本書ではそれらを読んで得た知見を記しているつもりですが、限界もあるので一次ソース重要ということで参照リンクです。

- https://babeljs.io/docs/core-packages/
- https://babeljs.io/docs/usage/api/
- https://github.com/thejameskyle/babel-handbook/blob/master/translations/en/plugin-handbook.md
- https://github.com/babel/babel/tree/master/packages
- https://github.com/babel/babylon/blob/master/ast/spec.md
- https://github.com/babel/babel/tree/master/packages/babel-types/src/definitions

第3章

Babel プラグイン

序章で作ったように、Babel のプラグインは第 2 章で紹介したツール群があればとても簡単にプラグインを作れます。前半部分では Babel プラグインの作り方を説明します。後半では実際にプラグインを作ってみましょう。

3.1 作り方

序章では visitor の入ったオブジェクトを返す関数を作りましたが、実はオブジェクトそのままでもプラグインとしては有効です。ただ、色々と便利なので関数の方がいいです。

リスト 3.1: chapter3/babel-plugin-object.js

```
 1: const {transform} = require('babel-core')
 2:
 3: const plugin = {
 4:   visitor: {
 5:     BinaryExpression: nodePath => {
 6:       console.log(nodePath.node.operator) // --> +
 7:     },
 8:   },
 9: }
10:
11: transform('1 + 2', {plugins: [plugin]})
```

関数によるプラグインであれば、引数に渡ってくるのはオブジェクトで babel-core が require している babel-traverse や babel-types, babel-template などが詰まっています。

つまり Babel のプラグインを作る場合、自前で babel-traverse や babel-types をインストールしたり require する必要はありません。なぜか Babylon は含まれていないのでもしも Babylon で parse したい場合は自前で require する必要があります。

リスト 3.2: chapter3/babel-plugin-func.js

```
 1: const {transform} = require('babel-core')
 2:
 3: const plugin = babel => {
 4:   const {traverse, types: t, template, version} = babel
 5:   console.log(version)                          // --> 7.0.0-beta.3
 6:
 7:   return {
 8:     visitor: {
 9:       BinaryExpression: nodePath => {
10:         console.log(t.isExpression(nodePath)) // --> true
11:         console.log(nodePath.node.operator)   // --> +
```

```
12:       },
13:     },
14:   }
15: }
16:
17: transform('1 + 2', {plugins: [plugin]})
```

babel プラグインのオブジェクトのプロパティは visitor だけではありません。

リスト 3.3: name

```
 1: const plugin = {
 2:   inherits,
 3:   visitor,
 4:   name: 'my-plugin-name',
 5:   pre() {
 6:     //前処理
 7:   },
 8:   post() {
 9:     //後処理
10:   }
11: }
```

それぞれ見ていきましょう。

3.1.1 name

name でプラグインの名前を付けます。リスト 3.3 なら my-plugin-name という名前のプラグインになります。

3.1.2 pre

pre は traverse する前に呼び出される関数です。

リスト 3.4: chapter3/pre.js

```
 1: const {transform} = require('babel-core')
 2:
 3: const plugin = babel => {
 4:   return {
 5:     pre() {
 6:       console.log('pre', this.constructor.name) // --> pre PluginPass
 7:       this.hoge = 'hoge'
 8:     },
 9:     visitor: {
10:       Program: (nodePath, state) => {
11:         console.log(state.constructor.name) // --> PluginPass
12:         console.log(state.hoge)             // --> hoge
13:       },
14:     },
15:   }
16: }
```

```
17:
18: transform('1 + 2', {plugins: [plugin]})
```

pre の中での this は babel-core で定義される PluginPass 型で、ビジター関数の第 2 引数に渡されます。
注意点として this で処理しなければならないので、アロー関数で記述することはできません。

3.1.3 post

post は traverse した後に呼び出される関数です。

リスト 3.5: chapter3/post.js

```
 1: const {transform} = require('babel-core')
 2:
 3: const plugin = babel => {
 4:   return {
 5:     pre() {
 6:       this.hoge = 'hoge'
 7:     },
 8:     visitor: {
 9:       Program: (nodePath, state) => {
10:         state.hoge = 'ほげ'
11:       },
12:     },
13:     post() {
14:       console.log(this.hoge) // --> ほげ
15:     }
16:   }
17: }
18:
19: transform('1 + 2', {plugins: [plugin]})
```

ビジター関数では state を書き換えることもできます。post ではそれをそのまま取得できるので、たとえばコード解析なんかに利用できます。

3.1.4 inherits

Babel ではあるプラグインがあることを前提に動作させることができます。
たとえば、babel-plugin-syntax-typescript を指定すれば、TypeScript を処理できるプラグインになります。

```
$ npm i babel-plugin-syntax-typescript -S
```

リスト 3.6: chapter3/inherits.js

```
 1: const {transform} = require('babel-core')
 2: const syntaxTypeScript = require('babel-plugin-syntax-typescript').default
 3:
 4: const plugin = babel => {
 5:   return {
 6:     inherits: syntaxTypeScript,
 7:     visitor: {
 8:       VariableDeclarator: (nodePath) => {
 9:         console.log(nodePath.node.id.typeAnnotation.type) // --> TSTypeAnnotation
10:       },
11:     },
12:   }
13: }
14:
15: transform('let hoge: Hoge', {plugins: [plugin]})
```

`TSTypeAnnotation` は TypeScript の型アノテーションの名前です。inherits を指定しなければこれを取得することはできません。

`babel-plugin-syntax-flow` というプラグインもありますが、TypeScript と Flow はかち合うので、どちらか片方しか指定することができません。

3.2 traverse を叩いたときの state との違い

ビジター関数の state はじつは traverse を叩くときの第 2 引数そのものです。プラグインの場合は babel-core が traverse に PluginPass を指定しているので state は PluginPass 型のオブジェクトなのです。

それ以外の事例、traverse を直接叩く場合は第 2 引数に任意のオブジェクトを渡せます。

リスト 3.7: chapter3/traverse-state.js

```
 1: const {transform} = require('babel-core')
 2:
 3: const plugin = babel => {
 4:   return {
 5:     pre() {
 6:       this.hoge = 'hoge'
 7:     },
 8:     visitor: {
 9:       Program: (nodePath, state) => {
10:         nodePath.traverse({
11:           BinaryExpression: (innerPath, innerState) => {
12:             console.log('inner', innerState) // --> {fuga: 'FUGA'}
13:           }
14:         }, {fuga: 'FUGA'})
15:         console.log(state.constructor.name) // --> PluginPass
16:         console.log(state.hoge)             // --> hoge
17:       },
18:     },
19:   }
20: }
21:
22: transform('1 + 2', {plugins: [plugin]})
```

3.3 プラグインオプションの取得方法

プラグインのオプションは PluginPass 型の opts に含まれているので、メインのビジター関数であれば、`state.opts` で取得できます。

リスト 3.8: chapter3/state-opts.js

```
const {transform} = require('babel-core')

const plugin = babel => {
  return {
    visitor: {
      Program: (nodePath, state) => {
        console.log(state.opts) // --> { hoge: 'hoge' }
      },
    },
  }
}

const plugins = [
  [plugin, {hoge: 'hoge'}]
]

transform('1 + 2', {plugins})
```

babel プラグインのオプションは少し特殊な指定方法をしています。配列の 1 つめにプラグイン、2 つめのオプションを渡すのです。[*1]

3.4 Babel プラグインとして Injector プラグインを作ってみる

Babel プラグインの作り方を説明してきました。それではさっそく実際にプラグインを作ってみましょう。

今回つくる Injector プラグインは DI を実現するもので、指定したソースコードに手を加えず、中の変数宣言・関数宣言・クラス宣言を置換したり、コードの最後に新しいコードを追加できるというものです。テストやデバッグに便利なものです。

Injector プラグインでは、まさに AST という題材をお手軽に楽しめると思います。

3.4.1 DI (Dependency Injection)

Dependency Injection は日本語でいうと依存性注入ですが、依存性というのは実際には何らかのオブジェクトやプリミティブ値や関数です。対象を動かすときに依存する物だから依存性です。

たとえば Node.js でファイルを読み込むには、`const fs = require('fs')` で fs のモジュールを読み込んで、`fs.readFile('hoge.js')` のように fs モジュールのファイルを読み込む関数を叩くのが一般的です。

このとき `const fs = require('dummy-fs')` というように、実際のファイル I/O を発生させないダミー (モック) を流し込めれば、対象のソースコードを変更しなくてもユニットテストしやすくなります。これを動的書き換えや AST を使わずにやるとするとどうでしょうか？ たとえばクラスのコンストラクタに fs を渡

[*1] 他の言語であればタプルというデータ構造が用意されていたりするのですが…。

すようにする仕組みなんかがよくあるパターンですね。DI はもともとデザインパターンの一種です。

JavaScript には AST というとても強い味方がいるのでスマートに DI を実現できます。

3.4.2 変数定義を置換してみる

まずは変数定義を置き換えてみます。

変数定義は VariableDeclaration と VariableDeclarator の二段構成です。`var a, b = 0` のような定義をした場合、1 つの VariableDeclaration (`kind` が`'var'`) の下に `a` を ID としてもつ VariableDeclarator と `b` を ID としてもつ VariableDeclarator がそれぞれぶら下がります。

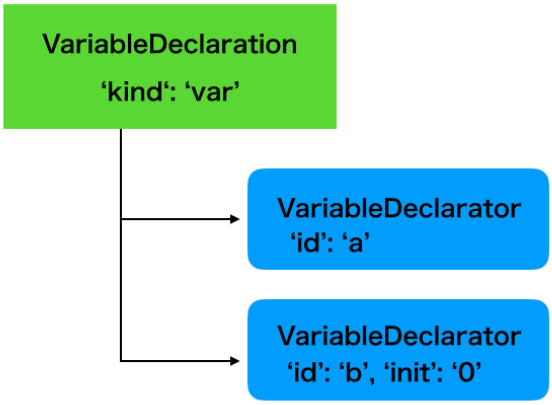


図 3.1 VariableDeclaration と VariableDeclarator

VariableDeclarator の init には変数の初期化の Node が入っているので、ここを置換すれば変数定義を変更できます。

リスト 3.9: chapter3/replace-variable-init.js

```
 1: const {transform} = require('babel-core')
 2: const {parseExpression} = require('babylon')
 3:
 4: const source = 'const hoge = require("hoge")'
 5:
 6: const targetId = 'hoge'
 7: const replaceCode = 'require("dummy-hoge")'
 8:
 9: const plugin = ({types: t, template}) => {
10:   return {
11:     visitor: {
12:       VariableDeclarator: (nodePath, state) => {
13:         if (t.isIdentifier(nodePath.node.id) && nodePath.node.id.name === targetId) {
14:           const newAst = parseExpression(replaceCode)
15:           nodePath.get('init').replaceWith(newAst)
16:         }
17:       },
18:     },
19:   }
20: }
21:
```

```
22: console.log(transform(source, {plugins: [plugin]}).code)
23: // --> const hoge = require("dummy-hoge");
```

もともとのソースでは `require("hoge")` で初期化されていた `hoge` を `require("dummy-hoge")` に置き換えました。もちろん `require` に限らずさまざまな式に置換できます。

`t.isIdentifier(nodePath.node.id)` を使って、id が Identifier かどうかを確かめています。もしも `const {hoge} = require('hoge')` のように、オブジェクトの分割代入の場合、`nodePath.node.id` には `ObjectPattern` が入っていたりしてとても面倒です。実用に使うなら ObjectPattern の場合の解析も必要になりますが、今回は単純に Identifier が入っていると決め打ちしてしまいましょう。

`newAst` は書き換えるコードの AST です。babel-template を使えば簡単に生成できます。

`nodePath.get('init')` で `VariableDeclarator` の `init` の NodePath を取得しています。さきほども説明したとおり、`init` は初期化式の Node なので、これを `replaceWith(newAst)` で置き換えます。

たった数行で変数定義の初期化式を書き換えることができました。Babel エコシステムのおかげです。

3.4.3 関数定義を置換してみる

次に、関数定義を書き換えてみましょう。関数定義は FunctionDeclaration です。今度は一部を書き換えるだけじゃなくて関数定義の全体を書き換えてみましょう。

リスト 3.10: chapter3/replace-function.js

```
 1: const {transform} = require('babel-core')
 2:
 3: const source = 'function hoge() {return 1}'
 4:
 5: const targetId = 'hoge'
 6: const replaceCode = 'function hoge() {return 2}'
 7:
 8: const WasCreated = Symbol('WasCreated')
 9:
10: const plugin = ({types: t, template}) => {
11:   return {
12:     visitor: {
13:       FunctionDeclaration: (nodePath, state) => {
14:         if (nodePath[WasCreated] || !t.isIdentifier(nodePath.node.id)) {
15:           return
16:         }
17:         if (nodePath.node.id.name === targetId) {
18:           const newAst = template(replaceCode)()
19:           nodePath.replaceWith(newAst)
20:           nodePath[WasCreated] = true
21:         }
22:       },
23:     }
24:   }
25: }
26:
27: console.log(transform(source, {plugins: [plugin]}).code)
28: // --> function hoge() {
29: //       return 2;
30: //     }
```

WasCreated はそれが作られたものなのかを判定するためのキーで nodePath[WasCreated] = true でマークを付けて、if (nodePath[WasCreated] || ...) でマークが付いていれば処理をそこでやめます。すでに置換処理後だからです。

Node や NodePath のどちらにマークを付けても実用上問題はありません。

今回は関数定義を関数定義に置き換えていますが、replaceCode に任意の文を書いておけばそれに書き換わります。

3.4.4 クラス定義も置き換えてみよう

クラス定義は関数定義と内容がほぼ同じです。

visitor のラベルには'FunctionDeclaration|ClassDeclaration'のように複数の識別子をまとめる指定方法もあるので、その書き方に直してみましょう。

リスト 3.11: chapter3/replace-function-class.js

```
 1: const {transform} = require('babel-core')
 2:
 3: const source = 'class Hoge {}'
 4:
 5: const targetId = 'Hoge'
 6: const replaceCode = 'class Hoge{ hoge() { return "hoge" } }'
 7:
 8: const WasCreated = Symbol('WasCreated')
 9:
10: const plugin = ({types: t, template}) => {
11:   return {
12:     visitor: {
13:       'FunctionDeclaration|ClassDeclaration': (nodePath, state) => {
14:         if (nodePath[WasCreated] || !t.isIdentifier(nodePath.node.id)) {
15:           return
16:         }
17:         if (nodePath.node.id.name === targetId) {
18:           const newAst = template(replaceCode)()
19:           nodePath.replaceWith(newAst)
20:           nodePath[WasCreated] = true
21:         }
22:       },
23:     }
24:   }
25: }
26:
27: console.log(transform(source, {plugins: [plugin]}).code)
28: // --> class Hoge {
29: //       hoge() {
30: //         return "hoge";
31: //       }
32: //
33: //     }
```

3.4.5 コードを最初や最後に挿入する

コードを一番最初や最後に挿入するのであればとても簡単です。Program Node を exit する時に、nodePath.pushContainer (先頭なら nodePath.unshiftContainer) というメソッドを叩けば良いのです。

リスト 3.12: chapter3/insert-last.js

```
const {transform} = require('babel-core')

const source = 'console.log(1)'

const insertCode = 'console.log(2)'

const plugin = ({types: t, template}) => {
  return {
    visitor: {
      Program: {
        exit: (nodePath, state) => {
          const newAst = template(insertCode)()
          nodePath.pushContainer('body', newAst)
        }
      },
    }
  }
}

console.log(transform(source, {plugins: [plugin]}).code)
// --> console.log(1);
//     console.log(2);
```

それ以外の位置へ挿入したい場合は、どうやって場所を指定するか悩ましいものです。行番号とかだと対象のソースコードを変更すると簡単に壊れてしまいます。

3.4.6 オプションで指定できるようにする

さて、`targetId` や `replaceCode` で直接指定するのはサンプルコードだからです。プラグインとして実用的に作るならプラグインオプションで指定できるようにします。

リスト 3.13: option

```
const options = {
  replace: {
    'const hoge =': '1 + 2',
    'function fuga': 'function piyo() {}',
    'class Foo': 'class Foo {}'
  },
  insert: {
    last 'exports.hooks.hoge = hoge'
  }
}

transform(src, {
  plugins: [
    [plugin, options]
```

```
15:   ]
16: })
```

オプションは PluginPass 型に含まれているので、リスト 3.13 で指定したオプションなら、ビジター関数の第 2 引数経由で `state.opts.replace['const hoge =']` や `state.opts.insert.last` という形で取得できます。

3.4.7 完成版

ここまで書いたことでひととおりプラグインを作ることができます。リスト 3.14 で Babel プラグインとして完成です。

これに手を加えた筆者版の DI プラグインを、https://github.com/erukiti/babel-plugin-dependency-injection というリポジトリで公開しています。

リスト 3.14: chapter3/babel-plugin-di.js

```
 1: const {parseExpression} = require('babylon')
 2:
 3: const WasCreated = Symbol('WasCreated')
 4:
 5: const plugin = ({types: t, template, version}) => {
 6:   const visitor = {
 7:     Program: {
 8:       exit: (nodePath, state) => {
 9:         if (state.inserters.last) {
10:           const newAst = template(state.inserters.last)()
11:           nodePath.pushContainer('body', newAst)
12:         }
13:       },
14:     },
15:     VariableDeclarator: (nodePath, state) => {
16:       const {kind} = nodePath.parent
17:
18:       if (t.isIdentifier(nodePath.node.id)) {
19:         const replaceCode =
20:           state.replacers[`${kind} ${nodePath.node.id.name} =`]
21:         if (replaceCode) {
22:           const newAst = parseExpression(replaceCode)
23:           nodePath.get('init').replaceWith(newAst)
24:         }
25:       }
26:     },
27:     'FunctionDeclaration|ClassDeclaration': (nodePath, state) => {
28:       if (nodePath[WasCreated] || !t.isIdentifier(nodePath.node.id)) {
29:         return
30:       }
31:       const optId = {
32:         FunctionDeclaration: 'function',
33:         ClassDeclaration: 'class',
34:       }
35:
36:       const replaceCode =
37:         state.replacers[`${optId[nodePath.type]} ${nodePath.node.id.name}`]
38:       if (replaceCode) {
39:         const newAst = template(replaceCode)()
```

```
40:         nodePath.replaceWith(newAst)
41:         nodePath[WasCreated] = true
42:       }
43:     },
44:   }
45:
46:   return {
47:     name: 'dependency-injection',
48:     visitor,
49:     pre() {
50:       this.inserters = Object.assign({}, this.opts.insert)
51:       this.replacers = Object.assign({}, this.opts.replace)
52:     },
53:   }
54: }
55:
56: module.exports = plugin
```

3.4.8 動作確認

リスト 3.15: chapter3/test-di.js

```
 1: const {transform} = require('babel-core')
 2:
 3: const opts = {
 4:   replace: {
 5:     'const hoge =': '"const hoge replaced"',
 6:     'function fuga': 'function fuga() {console.log("function fuga replaced")}',
 7:     'class Piyo': `
 8:       class Piyo {
 9:         constructor() {
10:           console.log('class Piyo replaced')
11:         }
12:         get() {
13:           return 'piyo'
14:         }
15:       }
16:     `,
17:   },
18:   insert: {
19:     last: 'module.exports.Piyo = Piyo',
20:   },
21: }
22:
23: const src = `
24: const hoge = 'hoge'
25:
26: console.log(hoge)
27:
28: function fuga() {
29:     console.log('fuga')
30: }
31:
32: fuga()
33:
34: class Piyo {
35:     constructor() {
```

```
36:         console.log('piyo')
37:     }
38:
39:     get() {
40:         return null
41:     }
42: }
43: `
44:
45: console.log('before:')
46: console.log(src)
47:
48: const {code} = transform(src, {
49:   plugins: [[require('./babel-plugin-di.js'), opts]],
50: })
51: console.log('\nafter:')
52: console.log(code)
```

3.5 Babel プラグインをパッケージ化する

Babel プラグインは通常、npm のパッケージとして使われます。npm パッケージを作成するためにはまず最低限 package.json が必要です。

リスト 3.16: chapter3/package.json

```
 1: {
 2:     "name": "babel-plugin-sample-di",
 3:     "version": "0.1.0",
 4:     "description": "Babel plugin sample",
 5:     "main": "src/index.js",
 6:     "scripts": {
 7:     },
 8:     "keywords": ["babel", "plugin"],
 9:     "repository": {
10:     },
11:     "author": "",
12:     "license": ""
13:   }
```

最低限のプロパティは説明しますが、一度は https://docs.npmjs.com/files/package.json に目を通しておく方がいいでしょう。

3.5.1 name

パッケージ名です。Babel プラグインには、先頭に `babel-plugin-`と付ける風習になっています。npm パッケージを公開するときは、他のパッケージ名とぶつからないように気をつける必要があります。

3.5.2 version

パッケージ公開をする時は前回よりも必ずバージョンを上げる必要があります。バージョンナンバーは、セマンティックバージョン[*2]に従いましょう。`x.y.z` というバージョンであれば、x がメジャーバージョンで y がマイナーバージョン、z がパッチバージョンです。

もし互換性が途切れる変更がある場合は必ずメジャーバージョンを上げましょう。最近は互換性が途切れなくても、年次リリースという形で、毎年メジャーバージョンを上げるプロダクトも多いです。

マイナーバージョンは基本的には互換性は変わらず機能追加やある程度の大きな修正をした時に上げます。

パッチバージョンはちょっとした修正レベルのときにあげます。

3.5.3 description

簡単な説明です。npm で公開する場合よほどのことが無い限りは英語です。プライベートなパッケージとかであれば日本語でも構いません。

3.5.4 main

エントリポイントです。トランスパイルの必要がない場合は元々のソースを指して、トランスパイルが必要な場合はトランスパイル後のファイルを指すようにしましょう。

3.5.5 scripts

たとえばトランスパイルが必要であれば `build` を指定することが多いです。

```
$ npm run build
```

3.5.6 keywords

npm に登録されるキーワードです。

3.5.7 repository

Github のリポジトリなどを指定しましょう。

3.5.8 author

作者の名前と連絡先です。

[*2] http://semver.org/lang/ja/

3.5.9 license

MIT, Apacahe-2.0 などを選ぶといいでしょう。https://spdx.org/licenses/ の一覧にある `Identifier` を選ぶ必要があります。

3.6 npm publish

https://www.npmjs.com/signup でサインアップをしてから、`npm adduser` コマンドを叩きます

```
$ npm adduser
```

あとは動作確認を済ませたら公開するだけです。

```
$ npm publish
```

バージョンアップも同じコマンドです。

3.7 Babel プラグインの自動テスト

https://github.com/babel-utils/babel-plugin-tester がオススメらしい[*3]です。

3.8 require hack

さて、プラグインとしてはこれで完成ですが動的に書き換えると便利なので、そのようにしてみましょう。

```
$ npm i babel-register -S
```

リスト 3.17: babel-register を使ったやつ

```
1: // injectorPlugin と injectorOptions を設定しておく
2:
3: require('babel-register')({
4:     plugins: [injectorPlugin, injectorOptions]
5: })
```

これだけでさっくりです。

[*3] 筆者はまだ試せていないのでなんともいえません。時間ができれば試して Qiita にでも記事を書いてみたいところです。