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 (注: 試作版 (10/17 rev2) です)

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 (Dependancy 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。

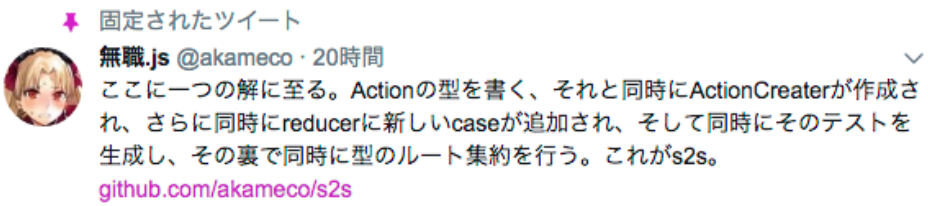
生産性の圧倒的向上

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。人間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
>
> 生産性や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。

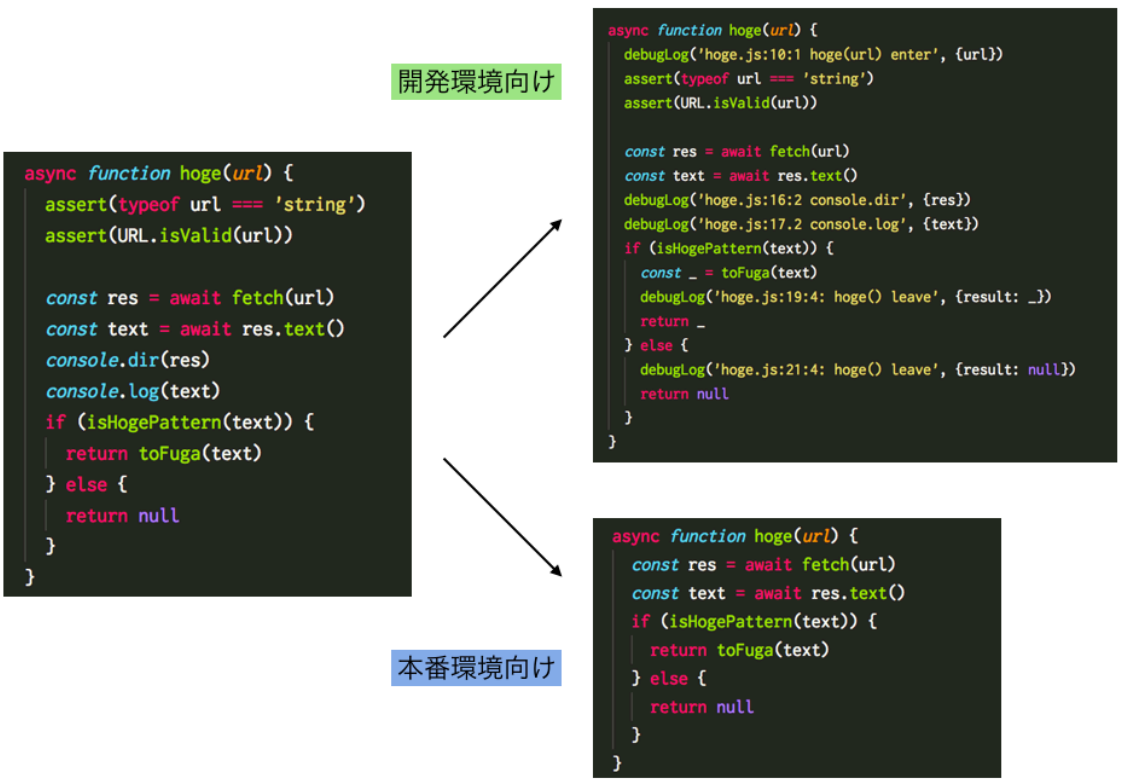
「CPU の歓声が聞こえる」ほどの天才のひとならばアセンブラでとても高性能・高効率なコードを書けるのでしょうが、生産性は最悪といってもいいです。天才以外はメンテナンスが難しく開発期間もかかるでしょう。少なくともそういう一部の特殊な人以外が書くアセンブラはコンパイラのはき出すものより性能が劣る時代です。

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、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としての仕事でしょう。

■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。細かい部分が色々と違う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
```

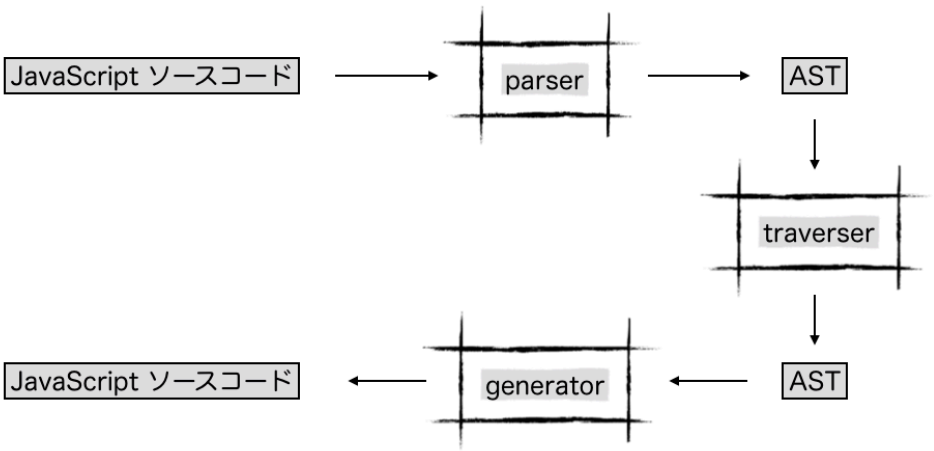
最後のオプション `-S` は `--save` の省略形で、プログラムの動作時に依存するという情報を package.json に記録するためのものです。もし開発時にしか依存しないパッケージであれば`-D` もしくは `--save-dev` を指定するといいでしょう。

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ましょう

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、これを使いましょう。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

[*5] 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

内でさくっと作れます。

リスト 1: 変換サンプル

```
 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。

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

[*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。

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。

またソースコードは https://github.com/erukiti/ast-book-sample に置いてあります。

目次

第1章

AST 解説

まずは 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 を実際に眺めてみましょう

AST を実際に見るには AST Explorer [*2] 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 関連の開発ではここが定番です。

babylon7 を選んで、１ ＋ ２ を食わせてみましょう。

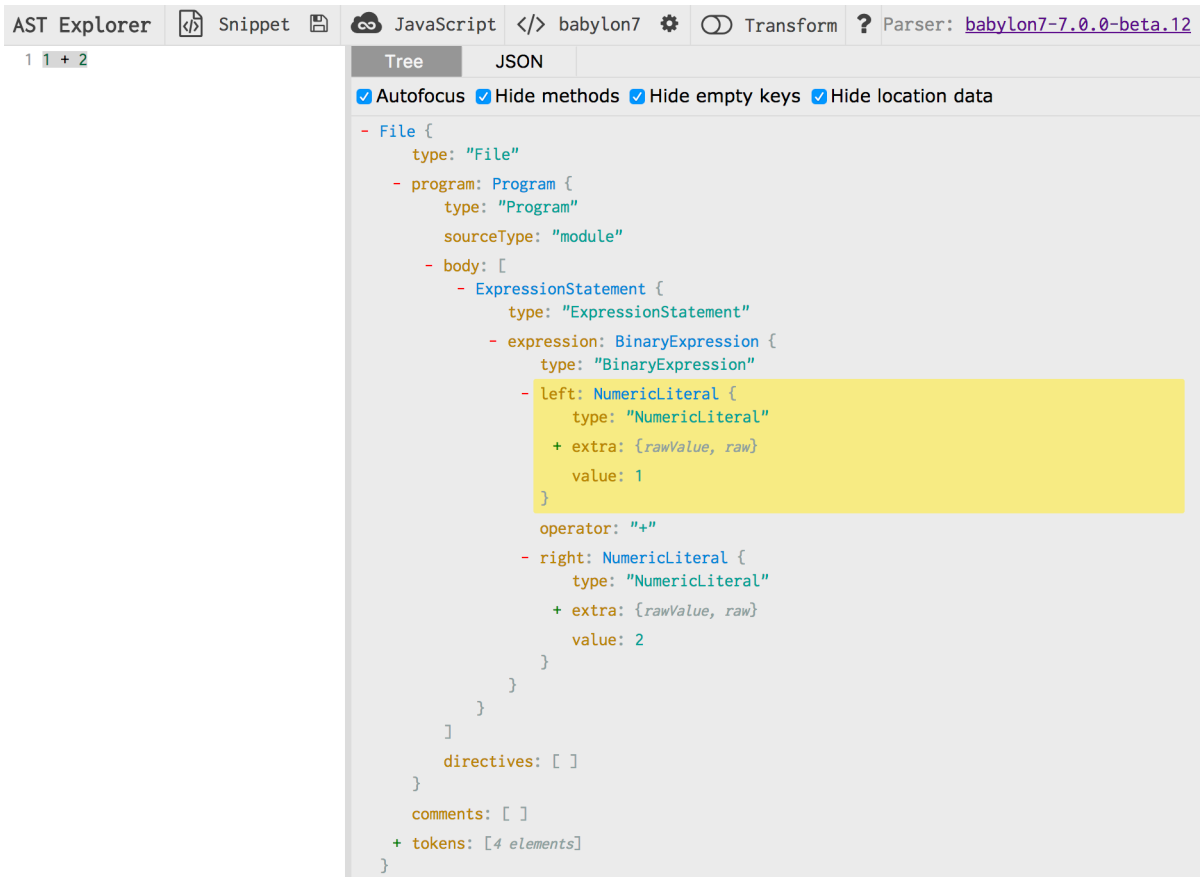


図 1.1　AST explorer

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

[*2] http://astexplorer.net/

AST はノード[*3]が集まって出来あがったツリーです。

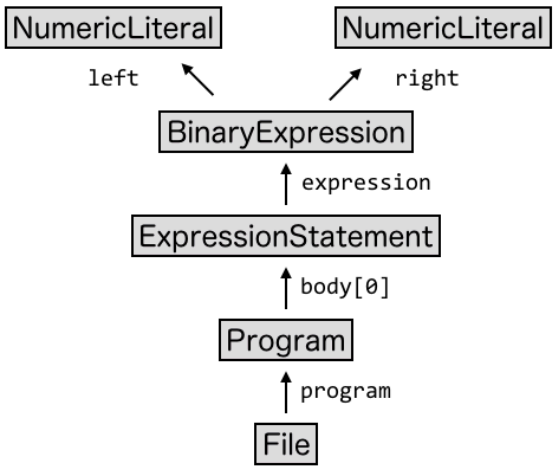


図 1.2　AST のツリー構造

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。そこに `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「文 (Statement)」の集まりです。プログラミング言語の世界で「文」というのは命令のことで、宣言だったり演算だったり様々なものです。JavaScript の場合は上から順に文を実行[*4]します。`Program` の `body` はそういった `Statement` の配列です。

`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。中身は `BinaryExpression`、つまり二項演算子です。

`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` です。つまり `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 `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です。

`NumericLiteral` はそこで完結していてそれ以上先をたどれない先端です。[*5]

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[*6]用に Parser API[*7]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 ESTree 仕様[*8]というものに進化しました。

[*3] 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ツリー構造の幹や枝や葉っぱなどを統合したものとして扱われることが多いです。
[*4] 関数宣言やクラス宣言は中身こそ実行しませんが、関数を宣言して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。
[*5] ツリー構造ではよく leaf node と呼ばれますね
[*6] 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
[*7] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Parser_API
[*8] https://github.com/estree/estree

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。

様々な 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。

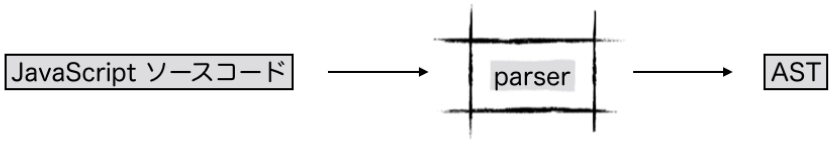
抽象構文木 (AST) はしがらみから脱却しています。ソースコード上の意味をもとに作られているツリーなので、表記方法という具象的な情報を切り落として考えられるから抽象なのです。

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 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: babylon をつかってみる

```
 1: const babylon = require('babylon')
 2:
 3: const ast = babylon.parse('1 + 2 * (3 + 4)')
 4: console.log(ast)
 5:
 6: /* 結果
 7: Node {
 8:   type: 'File',
 9:   start: 0,
10:   end: 15,
11:   loc:
12:    SourceLocation {
13:      start: Position { line: 1, column: 0 },
14:      end: Position { line: 1, column: 15 } },
15:   program:
16:    Node {
17:      type: 'Program',
18:      start: 0,
19:      end: 15,
20:      loc: SourceLocation { start: [Object], end: [Object] },
21:      sourceType: 'script',
22:      body: [ [Object] ],
23:      directives: [] },
24:   comments: [] }
25: */
```

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ないようです。

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

他には start，end，loc というプロパティもありますが、これら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: }
```

Node の子 Node 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

いつでも AST Explorer で AST を見られるとは限らないので、まずは 1 つツールを紹介します。babel-log[*9] 及び ast-pretty-print[*10] というツールです。[*11]

実のところ babel-log の `log()` 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` 以外に出したいなどといった場合に有効です。ちょっとしたショートハンドみたいなものですね。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: babel-log or ast-pretty-print

```
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: console.log('Node 型')
7: log(ast)
8: // console.log(printAST(ast, true))
```

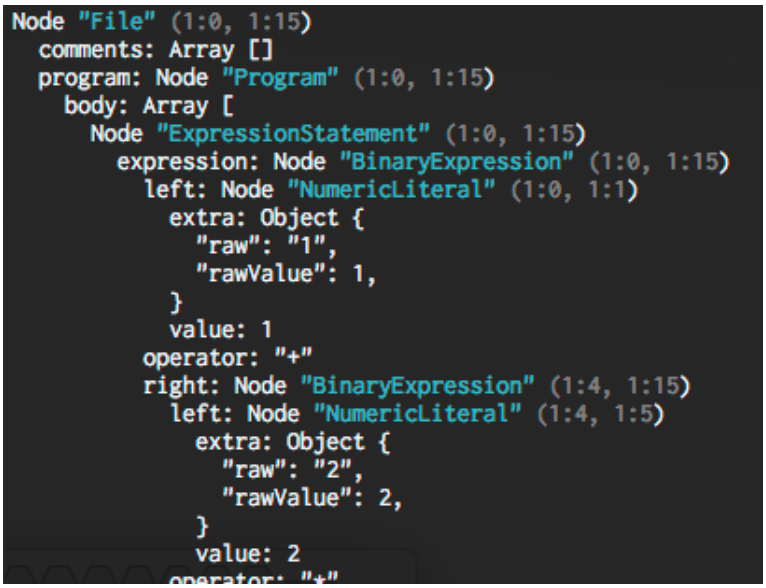


図 1.3 babel-log 結果

[*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の中身を再帰的ににたどって 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: replacer 関数追加版

```
 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: AST アウトライン

```
{
    "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、たとえば関数が自分自身を呼び出すことです。再帰処理に向いたデータ構造を再帰を

使わずにフラットにコードを書こうとすると、おそらくキューやスタックを使ったり複雑な仕掛けが必要になることが多いです。

それでは再帰処理に向いたデータ構造はツリー構造のデータ、たとえば JSON や JS のプレーンなオブジェクトなどです。

リスト 1.10: ツリーを再帰処理する

```
 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ましょう

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: 自作トラバーサ

```
 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)
27:     }
28:   })
29:
```

[*12] いつか筆者が表を書くかもしれません

```
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。

14 行目から 26 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。

30 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。

また、既存のトラバーサだと機能的に不満がある場合はもちろん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ことと、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きましょう

リスト 1.12: AST 計算機のビジター関数オブジェクト

```
 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: 呼び出し階層

```
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: 完成版

```
 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: babel-core

```
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。

AST の加工はプラグイン任せなので、リスト 2.1 では何もしないプラグインなので特に加工されることもありませんが、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。

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

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

```
{
    "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

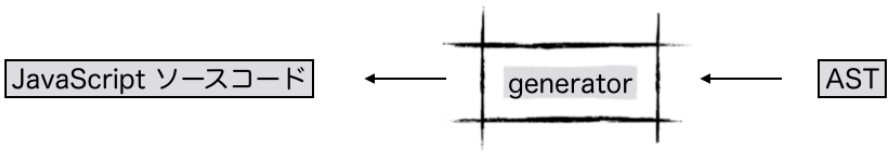
```
1: const plugins = [
2:   ['plugin-A', {hoge: 1}],
3:   'plugin-B',
4:   ['plugin-C']
5: ]
```

`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[*2]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。



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

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

リスト 2.4: babel-generator

```
 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。[*3]

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

リスト 2.5: prettier を使って生成する

```
 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: []})
```

[*3] 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

```
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を無視して AST を返しても大丈夫です。

リスト 2.6: ダミー文字列を指定した場合

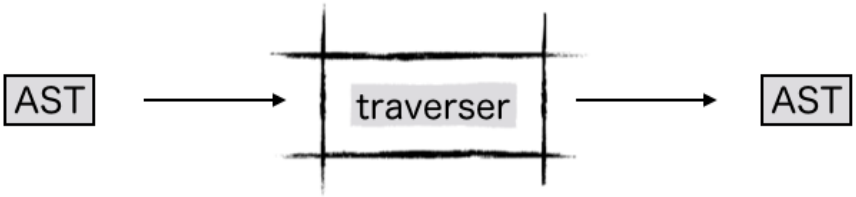
```
const prettier = require('prettier')
const {transformFromAst} = require('babel-core')

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[*4] 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。

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



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: traverse

```
 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。[*5]

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。

リスト 2.8: visitor の指定方法

```
1: const {parse} = require('babylon')
2: const traverse = require('babel-traverse').default
3:
4: const src = '1 + 2'
```

[*5] 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 require/import もできます。

```
 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 Node の探査開始時の `enter` と、探査終了時の `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[*6]があります。

`enter/exit` 以外の場合は、`NumericLiteral` や `BinaryExpression` のような Node 型の type[*7]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。[*8]

リスト 2.9: 結果

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

[*6] `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。

[*7] 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。

[*8] 言ってみれば、ちょっとした shorthand ですね

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: NodePath inspector

```
const {parse} = require('babylon')
const traverse = require('babel-traverse').default

const src = 'const a = 1; hoge(a)'

const ast = parse(src)

const inspectProps = prop => {
  const propType = typeof prop

  if (propType === 'string') {
    return `'${prop}'`
  } else if (propType !== 'object' || !prop) {
    return prop
  } else if (prop.constructor.name === 'Object') {
    return JSON.stringify(prop)
  } else if (prop.constructor.name === 'Array') {
    return `[${prop.map(value => inspectProps(value)).join(' ,')}]`
  } else {
    if ('type' in prop) {
      return `Object(${prop.constructor.name}) ${prop.type}`
    } else {
      return `Object(${prop.constructor.name})`
    }
  }
}

const visitor = {
  // enter(nodePath) {
  // 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。
  CallExpression(nodePath) {
    console.log(`enter ${nodePath.type}`)
    Object.keys(nodePath).sort().forEach(key => {
      console.log(`  ${key}:  ${inspectProps(nodePath[key])}`)
    })
  },
}

traverse(ast, visitor)

/* 結果
enter CallExpression
  container:  Object(Node) ExpressionStatement
  context:  Object(TraversalContext)
  contexts:  [Object(TraversalContext)]
  data:  {}
  hub:  undefined
  inList:  false
  key:  'expression'
  listKey:  undefined
  node:  Object(Node) CallExpression
  opts:  {"CallExpression":{"enter":[null]},"_exploded":true,"_verified":true}
  parent:  Object(Node) ExpressionStatement
  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: */
```

Node 型を取得するには node です。visitor 関数を書くときには、おそらく一番参照するでしょう。

CallExpression の親は ExpressionStatement です。Node 型で親を取得するなら parent で NodePath 型で取得するなら parentPath を使います。

親にとっての自分のキーは parentKey で取得できます。今回は expression ですが、BinaryExpression の子 Node であれば、left, right が入っているでしょう。

CallExpression の Node には callee(呼び出される関数・メソッド) というプロパティが入っていますが、これの NodePath バージョンを取得する簡単な方法は get メソッドです。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

メンバー	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[*9]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。

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

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

リスト 2.11: babel-types で AST を生成する

```
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。[*10]

`t.numericLiteral` などには全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: babel-types で型判定

```
 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
```

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

[*10] 間に babel-core の `transformFromAst` を挟むといいでしょう。
[*11] エイリアス (alias) といいます

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・メンバーなど何かの名前	`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

試供版

試供版はここまでとなります。

もし良ければ https://goo.gl/forms/EgAl6KFY04kiNKBy1 のアンケートにお答えいただけると、今後の同人誌や商業誌での執筆に反映できるかもしれません。

図 2.1　試供版アンケート

本書の完成版は 10/22 開催の技術書典 3 (https://techbookfest.org/event/tbf03) の え 13 東京ラビットハウス https://techbookfest.org/event/tbf03/circle/5099247972122624 で頒布予定です。サークルチェックを付けていただければ部数の目安になります (10/18 13:00 までなら部数を変更できます)。

`erukiti@gmail.com` 宛までメールを送るか、https://twitter.com/erukiti 宛にでもご連絡を頂ければ、取り置きなどにも対応させていただきます。

また、https://rabbit-house.tokyo/books/javascript-ast にて本書のサポートを行います。