AST FTW

may 2023,

José Pedro Dias

What is an AST?

AST stands for abstract syntax tree.
It's a structured representation of source code
for a given language.

Arkadium

JS and ASTs

ASTs can be generated and used on any programming language.
Javascript has been leveraging of ASTs for some time now due to the ever-evolving nature of the browser ecosystem, mainly to:

- allow old browsers to support newer syntax by means of rewriting it with older, more supported building blocks babel
- to test drive ES language feature proposal implementations TC39
- adhering to code best practices, normalizing code formatting eslint, prettier...)

Arkadium

Scope for today

The ideas presented here can be applied to other languages.
Even for Javascript & Typescript, there are multiple parsers and transformer solutions.

Today we'll focus on using JSCodeShift to manipulate Typescript code: it's powerful and covers both languages we profusely use at Arkadium (TS being a superset of JS).

Arkadium

What about text search and replace? 1/2

We may want to assess the consumers of a given class method to see if we can deprecate it (of games repos using core, for example).
Or maybe we found a use case where the method potentially misbehaves and want to see how it's being exercised.

Oftentimes we can make do with search, particularly if you're regexp skills are great and that feature isn't being creatively exercised...

What about text search and replace? 2/2

The thing is, imports can be renamed, instances can be assigned any variable name, function calls can receive complex inline arguments.

And if you want to take the extra step of not only finding but massaging the found hits, its unlikely you'll be able to do so using search and replace!

It's always a tradeoff. Do we need to pick some examples or find ALL calls? Is searching by text returning too many false hits? Is this to be used only once?

What does it look like

https://ts-ast-viewer.com/

https://astexplorer.net/

Note: *the format we'll be using for our transformations slightly differs, but this visualization is similar enough for demo purposes*

Common AST applications

code in lang X → AST → search for patterns in the tree

code in lang X → AST → manipulate tree → changed code in lang X

code in lang X → AST → manipulate tree → equivalent code in lang Y

Practical examples

example: Search

Find `require` calls. Tag them with todo for replacing them with ESM imports later

```
const fs = require('fs');
```

```
const fs = require('fs');// TODO
```

avoidRequires.ts

example: Edit

The `BitmapText` constructor options argument changed from core 1 to core 2.

Could rewrite it for simple use cases?

```
const bt = new BitmapText('some text', {
    font: '32px sans-serif'
});
```

```
new BitmapText('some text', {
    fontSize: 32,
    fontName: 'sans-serif'
});
```

changeConstructorCall.ts

Arkadium

example: Complex manipulations

Group all imports from the same package. Rename packages. Rename a subset of deprecated symbols (Container to IOCContainer). Avoid targeting symbols via dist paths. Sort imported symbols.

```
import {Inject} from '@arkadium/game-core';
import {AnimateViewBase} from '@arkadium/game-core/dist/Base/AnimateViewBase';
import {Container} from '@arkadium/game-core';
```

```
import { AnimateViewBase, Inject, IOCContainer } from '@arkadium/game-core-engine';
```

groupPackages.ts

Arkadium

Other considerations

Limitations

AST may result in counterintuitive structures for reasoning and manipulation.

As an example:
Comments in our AST are second-class citizens. Their nodes can't be freely placed, but instead they're children of their surrounding node (leading/trailing array of comments).
Whitespace-only lines aren't even captured on most JS ASTs.
Recast cleverly mitigates some of this by only recomputing AST subtree nodes which have been changed.

Be practical

A small piece of code may spawn a very complex AST.

Your subject code may feature too much variability, resulting in super complex manipulating code.

Time box your experience - not working? Drop it!

A Transformation recipe

1/2

Create an example file with the desired code to change.

Add some similar expressions which should be false positives.

You may target a source code repo instead (make sure to start from committed state)

Check the AST you get from it. example, using recast

- What are the node types you want to look for and navigate from?
- Which properties do you need to filter by to properly contain the change?

Arkadium

2/2

(if you need to ADD new nodes as part of the transformation)
look for which node types you need to call and their arguments.

Create your transformation code. Exercise each change you do:

- find the proper nodes and print or tag them.
- change them.
- add / delete.

Now target a larger corpus.

Questions?

Thank you!