プログラミング実験第三

TinyJavaScript コンパイラの作り直し

1311216
Rathore Amogh
岩崎研究室

Contents

1 はじめに

1.1 背景

TinyJavaScript は JavaScript の一部機能を制限んしたサブセットのことである [1]。TinyJavaScript の元のコンパイラは Mozilla の SpiderMonkey Parser API [2] を使用していた。しかし、SpiderMonkey Parser は絶えてしまって、TinyJavaScript のコンパイラの開発も続けられなくなった。だから、TinyJavaScript コンパイラを新しいパーサを使用して作りなおす必要が出てきた。このレポートは TinyJavaScript コンパイラを Node JS で作る実験について述べる。

1.2 実験の目的

この実験では TinyJavaScript のコンパイラを NodeJS と Esprima (ECMAScript Parsing Infrastructure for Multipurpose Analysis) [3] を使用して実装する。それで、コンパイラの作り方と構造について学習する。最後に、実験で作った新しいコンパイラと TinyJavaScript の [1] で述べてるコンパイラの比較を行う。

1.3 実装の方針

実装には Esprima という ECMAScript[6] Parser を使用する。JavaScript は ECMAScript の dialect であるので、ECMAScript のパーサは JavaScript の構文に対して正しくパーシングをされる。参考のため [1] で述べてる TinyJavaScript Compiler の実装を使う。もう少し具体的に言うと、Esprima を使って TinyJavaScript のソースコードを抽象構文木に変換する。それから、抽象構文木をトラバースして SSJSVM(これについて後で述べる) の命令列を生成する。

2 TinyJavaScript と SSJSVM

2.1 TinyJavaScript

TinyJavaScript は JavaScript の以下の機能をサポートしない [1]

- with 文
- delete 文
- グローバル変数宣言時の var の省略
- for in 文
- switch 文
- 名前付きの関数定義

[1] より TinyJavaScript の文法規則が図 1 通りである。

⟨ プログラム ⟩ ::= ⟨ 複変数宣言文 ⟩ ⟨ 複文 ⟩

⟨ 複変数宣言文 ⟩ ::= ⟨ 変数宣言文 ⟩∗

⟨ 変数宣言文 ⟩ ::= var ⟨ 識別子 ⟩ | var ⟨ 識別子 ⟩ = ⟨ 式 ⟩

⟨ 複文 ⟩ ::= ⟨ 文 ⟩∗

⟨ 文 ⟩ ::= ⟨ 式 ⟩ ; | ⟨ 梱包文 ⟩ | ⟨*if* 文 ⟩
| ⟨*while* 文 ⟩ | ⟨*do* 文 ⟩
| ⟨*for* 文 ⟩ | return ⟨ 式 ⟩ ;

⟨ 梱包文 ⟩ ::= {⟨ 複文 ⟩}

⟨*if* 文 ⟩ ::= if(⟨ 式 ⟩) ⟨ 文 ⟩
| if(⟨ 式 ⟩) ⟨ 文 ⟩ else ⟨ 文 ⟩

⟨*while* 文 ⟩ ::= while(⟨ 式 ⟩) ⟨ 文 ⟩

⟨*do* 文 ⟩ ::= do ⟨ 文 ⟩ while(⟨ 式 ⟩);

⟨*for* 文 ⟩ ::= for(⟨ 式 ⟩ ; ⟨ 式 ⟩ ; ⟨ 式 ⟩ ;) ⟨ 文 ⟩

⟨ 式 ⟩ ::= ⟨ 関数定義式 ⟩ | ⟨ 関数呼び出し式 ⟩ | ⟨ メンバー式 ⟩
| ⟨ 代入式 ⟩ | ⟨*new* 式 ⟩ | ⟨ 前置単項式 ⟩ | ⟨ 後置単項式 ⟩
| ⟨ 二項式 ⟩ | ⟨ 三項式 ⟩ | ⟨ 文字列リテラル ⟩ | ⟨ 数値 ⟩
| ⟨ 識別子 ⟩

⟨ 関数定義式 ⟩ ::= function(){ ⟨ プログラム ⟩ }
| function(⟨ 識別子 ⟩ [, ⟨ 識別子 ⟩]∗){ ⟨ プログラム ⟩ }

⟨ 関数呼び出し式 ⟩ ::= ⟨ 式 ⟩ () | ⟨ 式 ⟩ (⟨ 式 ⟩ [, ⟨ 式 ⟩]∗)

⟨ メンバー式 ⟩ ::= ⟨ 式 ⟩ [⟨ 式 ⟩] | ⟨ 式 ⟩ . ⟨ 識別子 ⟩

⟨ 代入式 ⟩ ::= ⟨ 左辺値 ⟩ ⟨ 代入演算子 ⟩ ⟨ 式 ⟩

⟨ 代入演算子 ⟩ ::= = | += | *= | /= | %=

⟨ 左辺値 ⟩ ::= ⟨ 識別子 ⟩ | ⟨ メンバー式 ⟩

⟨ 前置単項式 ⟩ ::= ++ ⟨ 左辺値 ⟩ | -- ⟨ 左辺値 ⟩ | - ⟨ 式 ⟩ | + ⟨ 式 ⟩
| typeof ⟨ 式 ⟩ | void ⟨ 式 ⟩

⟨ 後置単項式 ⟩ ::= ⟨ 左辺値 ⟩ ++ | ⟨ 左辺値 ⟩ --

⟨ 二項式 ⟩ ::= ⟨ 式 ⟩ ⟨ 二項演算子 ⟩ ⟨ 式 ⟩

⟨ 二項演算子 ⟩ ::= + | - | / | * | % | < | <= | > | >= | == | === | && | || | ,

⟨ 文字列リテラル ⟩ ::= " ⟨ 英数字 ⟩ ∗ "

⟨ 識別子 ⟩ ::= ⟨ 英字 ⟩ ⟨ 英数字 ⟩∗

Figure 1: TinyJavaScript の文法規則

2.2 SSJSVM

SSJSVM は Server-side JavaScript Virtual Machine の略称である。すなわち、サーバで動く JavaScript の仮想機械のことである。本実験では、TinyJavaScript のソースコードを SSJSVM の命令列に変換するコンパイラを作る。

3 パーサの選び方

実験の目的は TinyJavaScript のコンパイラを新しいパーサ (SpiderMonkey 1.6 でないパーサ) を使用して作るということである。だから、プログラミングを始める前にパーサを選ぶ必要がある。以下の特徴を持つパーサを使いたい。

1. ECMAScript の新しいバージョンをサポートする
2. 構文解析を正しくやってくれる
3. 使いやすい
4. ドキュメントとサポートがいい

いろいろ調べた結果、２つの案が出てきた。

- ECMAScript の文法を揃えて ANTLR[7] (ANother Tool for Language Recognition) を使ってパーサを生成する
- Esprima を使う

ANTLR を使用すると ANTLR 用の文法が必要となる。その文法はいろいろなソースがインターネットで提供してるが、オフィシャルではないので正しさは保証できない。あとドキュメントなども全然提供されてなくて、使いにくいと判断した。ところで、Esprima はとてもアクチブなプロジェクトであって、投稿者が多い。だから、最後に Esprima を使うと決める。

4 Esprima

4.1 特徴

Esprima [3] は JavaScript で書かれてる ECMAScript のパーシングインフラストラクチャである。Esprima の主な特徴は以下のとおりである [3]。

4.2 例

Esprima には「parse」メソッドがあって、そのメソッドにソースコードをストリングで渡すと抽象構文木が JS オブジェクトで返される。

```
var a = 2;
```

上の変数宣言のコードの抽象構文木オブジェクトの JSON は以下のようである。

```
{
    "type": "Program",
    "body": [
        {
            "type": "VariableDeclaration",
            "declarations": [
                {
                    "type": "VariableDeclarator",
                    "id": {
                        "type": " Identifier ",
                        "name": "a"
                    },
                    "init": {
                        "type": "Literal",
                        "value": 2,
                        "raw": "2"
                    }
                }
            ],
            "kind": "var"
        }
    ],
    "sourceType": "script"
}
```


```
var a = ["amogh", "rathore", 21];
```

配列なら以下のようになる。


```
{
    "type": "Program",
    "body": [
        {
            "type": "VariableDeclaration",
            "declarations": [
                {
                    "type": "VariableDeclarator",
                    "id": {
                        "type": " Identifier ",
                        "name": "a"
                    },
                    "init": {
                        "type": "ArrayExpression",
                        "elements": [
                            {
```

```
                    "type": "Literal",
                    "value": "amogh",
                    "raw": "\"amogh\""
                },
                {
                    "type": "Literal",
                    "value": "rathore",
                    "raw": "\"rathore\""
                },
                {
                    "type": "Literal",
                    "value": 21,
                    "raw": "21"
                }
            ]
        }
    }
],
"kind": "var"
}
],
"sourceType": "script"
}
```


上からわかるように、各構文は「type」というプロパーティが付いている。そのプロパーティから構文の種類 (BNF の非終端記号と一緒) がわかる。

- ECMAScript 6 (ECMA-262 [6]) 全体をサポート
- Estree プロジェクトの標準を対応する構文木フォーマット
- よくテストされたパーサ [4]
- オープンソース [5]

5 コンパイラの設計

本実験で作るコンパイラは、ソースコードを 3 つの段階で仮想機械の命令列に変換する。その段階は以下の通りである。

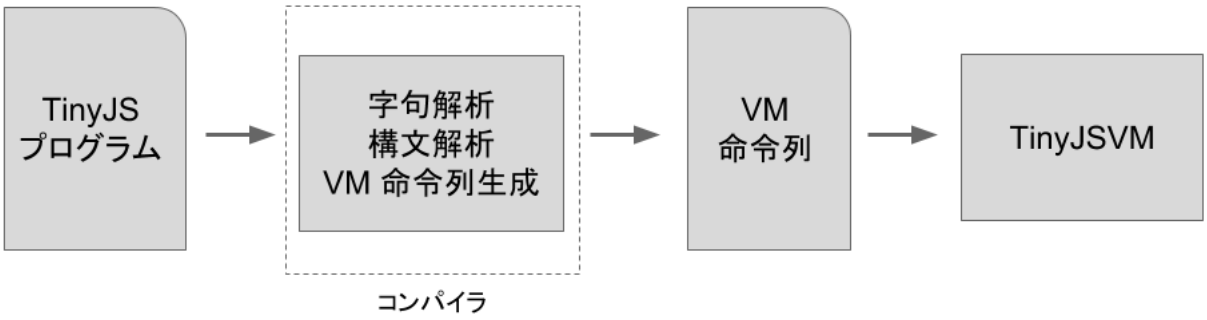

Figure 2: コンパイルの 3 つの段階

1. ソースコードの字句解析を行う
2. 字句列から抽象構文木を生成する
3. 抽象構文木を仮想機械の命令列に変換する

本実験で使う Esprima パーサは段階 1 と段階 2 の処理を行ってくれる。Esprima は抽象構文木を JS オブジェクトにして返す。

6 コンパイラの実装

この説でコンパイラの実装を述べる。コンパイラは各構文に対して別の変換を行うので、各構文に対しての処理をこれから述べる。

6.1 Literal タイプ

Esprima の Literal タイプは数値、真偽値や undefined などの定数、文字列、正規表現を含んでいる。

6.1.1 数値

数値の命令列を生成するために、まずは整数かどうかを調べる。なぜなら、SSJSVM で整数と浮動小数点数に対して別の命令が存在するから。コンパイラは NodeJS で書くので、JS の Number.isSafeInteger() メソッドを使って、整数かどうかを判断する。整数なら fixnum、整数でないなら number という命令を使う。

例：var a = 3; の'2' のところは以下のようになる

```
...
fixnum 2 2
...
```

整数でない場合は、すなわち var a = 2.5; の場合

```
...
number 2 2.5
...
```

のようになる。

6.1.2 真偽値や undefined などの定数

この場合は、SSJSVM の「specconst」という命令を使う。specconst 命令は引数で定数を以下のように受け取る。

定数	SSJSVM で使う引数
true	true
false	false
null	null
undefined	undefined

例：var a = true; は以下のようになる

```
...
specconst 2 true
...
```

6.1.3 文字列

文字列の場合、SSJSVM の「string」命令を使う。
例：var a = "amogh";

```
...
string 2 "amogh"
...
```

6.1.4 正規表現

この場合、SSJSVM の「regexp」命令を使う。JavaScript の正規表現のフラッグを整数のビットに保持して渡す。
例：var a = /d(b+)d/g; の場合

```
...
regexp 2 1 "d(b+)d"
...
```

のようになる。ここで、2はレジスターで、1は正規表現のフラッグを表す。1の意味はフラッグgが立ててることを表す。

6.2 バイナリ式 (BinaryExpression タイプ)

バイナリ式の場合、まずは演算子の左辺と右辺を評価して、2つのレジスタに保持する。次に、2つのレジスタの値をオペランドにして、演算子に対応する適当な命令を使う。演算子と命令は以下の通りである。

演算子	命令
<	lessthan
<=	lessthanequal
>	greaterthan
>=	greaterthanequal
==	JS なので複雑な手順を使わないといけないが、equal 命令を使う
===	eq
+	add
-	sub
*	mul
/	div
%	mod
&	bitand
|	bitor
<<	leftshift
>>	rightshift
>>>	unsignedrightshift

例： 5 + 10; の場合以下のようになる

```
...
fixnum 3 1
fixnum 4 2
add 2 3 4
...
```

6.3 変数 (Identifier タイプ)

変数の環境をまず求める。それは、local か global か arg のいずれかである。環境を求めたあと命令を以下のように決める。

6.4 代入 (AssignmentExpression タイプ)

まずは、左辺の種類を求める。すなわち、変数かプロパーティかを判断する。

環境	命令
local	getlocal
global	getglobal
arg	getarg

6.4.1 変数の場合 (左辺は Identifier タイプ)

まずは変数の環境を求める。次に、環境によって、以下のように命令を決める。

環境	命令
local	setlocal
global	setglobal
arg	setarg

例：a = "some string"; の場合

```
...
string  2 "some string"
string  5 "a"
setglobal  5 2
...
```

6.4.2 プロパーティの場合 (左辺は MemberExpression タイプ)

ここはまた 2 つ種類がある。左辺は . を使った代入と、左辺は [ ] を使った代入。

. を使った代入だとまずは . の左辺の変数 (Identifier) をコンパイルしてレジスタ t1 に保持する。その後、= の右辺の式をコンパイルしてレジスタ t2 に保持する。それから、. の右辺のプロパーティに対して string 命令を使ってそれをレジスタ t3 に保持する。最後に setprop t1 t2 t3 命令を使ってプロパーティに代入する。

例：a.prop = 10; の場合は以下のようになる

```
string  7 "a"
getglobal  3 7
fixnum 2 10
string  5 "prop"
setprop 3 5 2
```

[ ] を使った場合は、[ ] の左辺の変数をコンパイルしてレジスタ t1 に保持する。[ ] の中身をコンパイルしてレジスタ t3 に保持する。= の右辺をコンパイルしてレジスタ t2 に保持する。最後に、setprop t1 t2 t3 命令を使ってプロパーティに代入する。

6.5 ConditionalExpression タイプ

e1?e2:e3 を考える。２つのラベル L1 と L2 を用意する。まずは、e1 をコンパイルして結果をレジスタ t に保持する。次に、t が false なら L1 にジャンプする命令を生成する。その後は e2 をコンパイルして、ラベル L2 にジャンプする命令を生成する。それは、t が true なら e3 に行かないからである。それから、ラベル L1 をつけて、e3 をコンパイルする。最後にラベル L2 をつける。
例： a > b ? 3 : 5; の場合は以下のようになる。

```
...
string  6 "a"
getglobal  4 6
string  7 "b"
getglobal  5 7
lessthan  3 5 4
jumpfalse 3 3
fixnum 2 3
jump 2
fixnum 2 5
...
```

---

7 評価

8 終わりに

References

[1] 高田 祥. *ARM* 上で動作する *JavaScript* 処理系の実装. 電気通信大学 電気通信学部情報工学科 ソフトウェア学講座. January, 2011.

[2] SpiderMonkey 1.6
http://www-archive.mozilla.org/js/spidermonkey/release-notes/

[3] Esprima
http://esprima.org/

[4] Esprima のテスト情報
http://esprima.org/test/ci.html

[5] Esprima のソースコード
https://github.com/jquery/esprima

[6] ECMAScript
http://www.ecma-international.org/publications/standards/Ecma-262.htm

[7] ANTLR
http://www.antlr.org/

[8] EStree プロジェクト
https://github.com/estree/estree