Современный учебник JavaScript

© Илья Кантор

Сборка от 27 апреля 2014 для печати

Внимание, эта сборка может быть устаревшей и не соответствовать текущему тексту.
Актуальный онлайн-учебник, с интерактивными примерами, доступен по адресу http://learn.javascript.ru.

Вопросы по JavaScript можно задавать в комментариях на сайте или на форуме javascript.ru/forum.

Вопросы по сборке, предложения по её улучшению – можно писать мне, по адресу iliakan@javascript.ru .

Глава: Сжатие JavaScript

В файле находится только одна глава учебника. Это сделано в целях уменьшения размера файла, для удобного чтения с устройств.

Содержание

Современные сжиматели JavaScript

Перед выкладыванием JavaScript на «боевую» машину — пропускаем его через минификатор (также говорят «сжиматель»), который удаляет пробелы и по-всякому оптимизирует код, уменьшая его размер.

В этой статье мы посмотрим, как работают современные минификаторы.

Современные сжиматели

Рассматриваемые в этой статье алгоритмы и подходы относятся к минификаторам последнего поколения.

Вот их список:

- Google Closure Compiler [1]
- UglifyJS [2]
- Microsoft AJAX Minifier [3]

Самые широко используемые — первые два, поэтому будем рассматривать в первую очередь их.

Наша цель — понять, как они работают, и что интересного с их помощью можно сотворить.

С чего начать?

Для GCC:

1. Убедиться, что стоит Java [4]
2. Скачать и распаковать http://closure-compiler.googlecode.com/files/compiler-latest.zip , нам нужен файл `compiler.jar`.
3. Сжать файл `my.js`: `java -jar compiler.jar --charset UTF-8 --js my.js --js_output_file my.min.js`

Обратите внимание на флаг `--charset` для GCC. Без него русские буквы будут закодированы во что-то типа `\u1234`.

Google Closure Compiler также содержит песочницу [5] для тестирования сжатия и веб-сервис [6], на который код можно отправлять для

сжатия. Но скачать файл обычно гораздо проще, поэтому его редко где используют.

Для UglifyJS:

1. Убедиться, что стоит Node.js [7]
2. Поставить `npm install uglify-js`.
3. Сжать файл `my.js`: `uglifyjs my.js -o my.min.js`

Что делает минификатор?

Все современные минификаторы работают следующим образом:

1. Разбирают JavaScript-код в синтаксическое дерево.

   Также поступает любой интерпретатор JavaScript перед тем, как его выполнять. Но затем, вместо исполнения кода…

2. Бегают по этому дереву, анализируют и оптимизируют его.
3. Записывают из синтаксического дерева получившийся код.

Как выглядит дерево?

Посмотреть синтаксическое дерево можно, запустив компилятор со специальным флагом.

Для GCC есть даже способ вывести его:

1. Сначала сгенерируем дерево в формате DOT [8] :

   ```
   java -jar compiler.jar --js my.js --use_only_custom_externs --print_tree >my.dot
   ```

   Здесь флаг `--print_tree` выводит дерево, а `--use_only_custom_externs` убирает лишнюю служебную информацию.

2. Файл в этом формате используется в различных программах для графопостроения.

   Чтобы превратить его в обычную картинку, подойдёт утилита `dot` из пакета Graphviz [9]:

   ```
   // конвертировать в формат png
   dot -Tpng my.dot -o my.png

   // конвертировать в формат svg
   dot -Tsvg my.dot -o my.svg
   ```

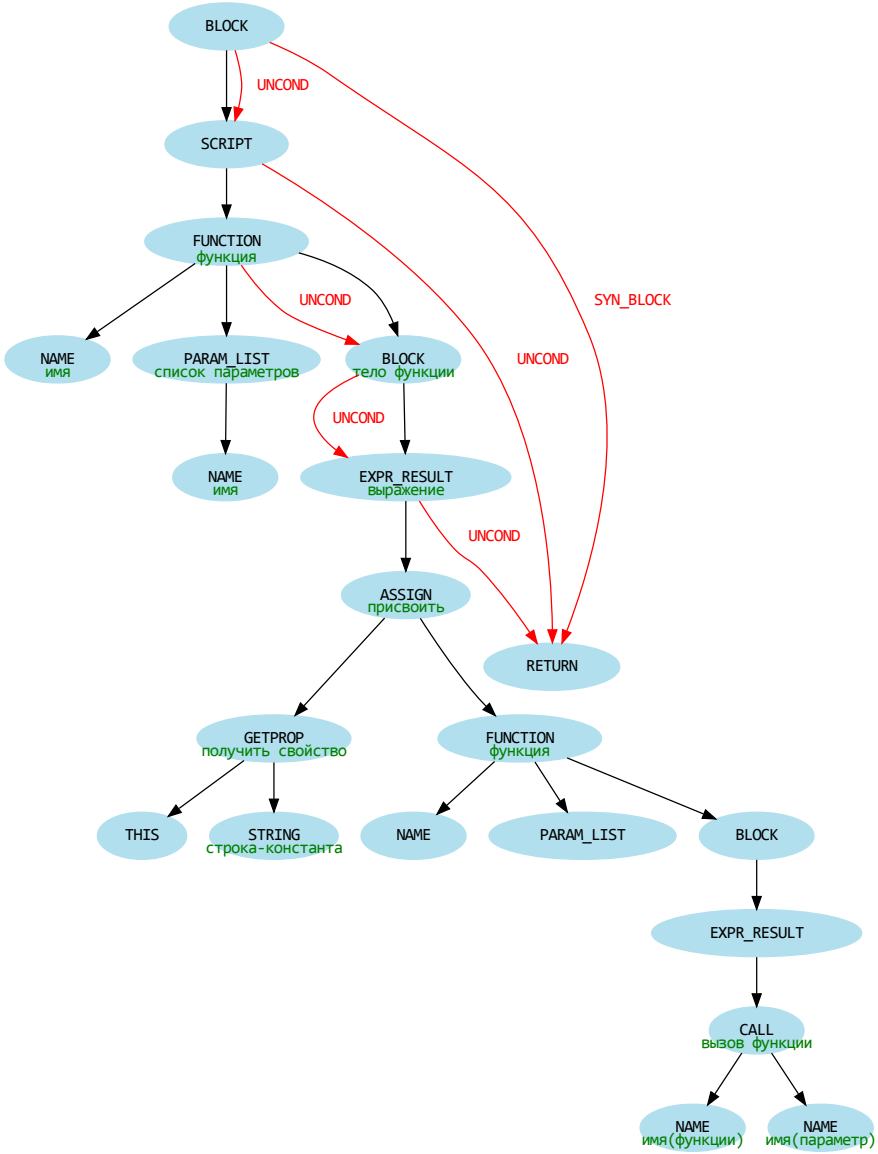
Пример кода `my.js`:

```
function User(name) {

  this.sayHi = function() {
    alert(name);
  };

}
```

Результат, получившееся из `my.js` дерево:



В узлах-эллипсах на иллюстрации выше стоит тип, например `FUNCTION` (функция) или `NAME` (имя переменной). Комментарии к ним на русском языке добавлены мной вручную.

Кроме него к каждому узлу привязаны конкретные данные. Сжиматель умеет ходить по этому дереву и менять его, как пожелает.

Комментарии JSDoc

Обычно когда код превращается в дерево — из него естественным образом исчезают комментарии и пробелы. Они не имеют значения при выполнении, поэтому игнорируются.

Но Google Closure Compiler добавляет в дерево информацию из *комментариев JSDoc*, т.е. комментариев вида `/** ... */`, например:

```
/**
 * Номер минимальной поддерживаемой версии IE
 * @const
 * @type {number}
 */
var minIEVersion = 8;
```

Такие комментарии не создают новых узлов дерева, а добавляются в качестве информации к существующем. В данном случае — к переменной `minIEVersion`.

В них может содержаться информация о типе переменной (`number`) и другая, которая поможет сжимателю лучше оптимизировать код (`const` — константа).

Оптимизации

Сжиматель бегает по дереву, ищет «паттерны» — известные ему структуры, которые он знает, как оптимизировать, и обновляет дерево.

В разных минификаторах реализован разный набор оптимизаций, сами оптимизации применяются в разном порядке, поэтому результаты работы могут отличаться. В примерах ниже даётся результат работы GCC.

Объединение и сжатие констант

До оптимизации:

```
function test(a, b) {
   run(a, 'my'+'string', 600*600*5, 1 && 0, b && 0)
}
```

После:

```
function test(a,b){run(a,"mystring",18E5,0,b&&0)};
```

- `'my' + 'string'` → `"mystring"`.
- `600 * 600 * 5` → `18E5` (научная форма числа, для краткости).
- `1 && 0` → `0`.
- `b && 0` → без изменений, т.к. результат зависит от `b`.

Укорачивание локальных переменных

До оптимизации:

```
function sayHi(name, message) {
  alert(name +" сказал: " + message);
}
```

После оптимизации:

```
function sayHi(a,b){alert(a+" сказал: "+b)};
```

- Локальная переменная заведомо доступна только внутри функции, поэтому обычно её переименование безопасно (необычные случаи рассмотрим далее).
- Также переименовываются локальные функции.
- Вложенные функции обрабатываются корректно.

Объединение и удаление локальных переменных

До оптимизации:

```
1 function test(nodeId) {
2   var elem = document.getElementsById(nodeId);
3   var parent = elem.parentNode;
4   alert(parent);
5 }
```

После оптимизации (GCC):

```
function test(a){a=document.getElementsById(a).parentNode;alert(a)};
```

- Локальные переменные были переименованы.
- Лишние переменные убраны. Для этого сжиматель создаёт вспомогательную внутреннюю структуру данных, в которой хранятся сведения о «пути использования» каждой переменной. Если одна переменная заканчивает свой путь и начинает другая, то вполне можно дать им одно имя.
- Кроме того, операции `elem = getElementsById` и `elem.parentNode` объединены, но это уже другая оптимизация.

Уничтожение недостижимого кода, разворачивание `if`-веток

До оптимизации:

```
01 function test(node) {
02   var parent = node.parentNode;
03
04   if (0) {
05     alert("Привет с параллельной планеты");
06   } else {
07     alert("Останется только один");
08   }
09
10   return;
11
12   alert(1);
13 }
```

После оптимизации:

```
function test(){alert("Останется только один")}
```

- Если переменная присваивается, но не используется, она может быть удалена. В примере выше эта оптимизация была применена к переменной `parent`, а затем и к параметру `node`.
- Заведомо ложная ветка `if(0) { .. }` убрана, заведомо истинная — оставлена.

  То же самое будет с условиями в других конструкциях, например `a = true ? c : d` превратится в `a = c`.

- Код после `return` удалён как недостижимый.

Переписывание синтаксических конструкций

До оптимизации:

```
var i = 0;
while (i++ < 10) {
  alert(i);
}

if (i) {
  alert(i);
}

if (i=='1') {
  alert(1);
} else if(i == '2') {
  alert(2);
} else {
  alert(i);
}
```

После оптимизации:

```
for(var i=0;10>i++;)alert(i);i&&alert(i);"1"==i?alert(1):"2"==i?alert(2):alert(i);
```

- Конструкция `while` переписана в `for`.
- Конструкция `if (i) ...` переписана в `i&&...`.
- Конструкция `if (cond) ... else ...` была переписана в `cond ? ... : ...`.

Инлайнинг функций

Инлайнинг функции — приём оптимизации, при котором функция заменяется на своё тело.

До оптимизации:

```
function sayHi(message) {

  var elem = createMessage('div', message);
  showElement(elem);

  function createMessage(tagName, message) {
    var el = document.createElement(tagName);
    el.innerHTML = message;
    return el;
  }

  function showElement(elem) {
    document.body.appendChild(elem);
  }
}
```

После оптимизации (переводы строк также будут убраны):

```
function sayHi(b){
  var a=document.createElement("div");
  a.innerHTML=b;
  document.body.appendChild(a)
};
```

- Вызовы функций `createMessage` и `showElement` заменены на тело функций. В данном случае это возможно, так как функции используются всего по разу.
- Эта оптимизация применяется не всегда. Если бы каждая функция использовалась много раз, то с точки зрения размера выгоднее оставить их «как есть».

Инлайнинг переменных

Переменные заменяются на значение, если оно заведомо известно.

До оптимизации:

```
(function() {
  var isVisible = true;
  var hi = "Привет вам из JavaScript";

  window.sayHi = function() {
    if (isVisible) {
      alert(hi);
      alert(hi);
      alert(hi);
      alert(hi);
      alert(hi);
      alert(hi);
      alert(hi);
      alert(hi);
      alert(hi);
      alert(hi);
      alert(hi);
      alert(hi);
    }
  }

})();
```

После оптимизации:

```
(function(){
  window.sayHi=function(){
    alert("Привет вам из JavaScript");
    alert("Привет вам из JavaScript");
    alert("Привет вам из JavaScript");
    alert("Привет вам из JavaScript");
    alert("Привет вам из JavaScript");
    alert("Привет вам из JavaScript");
    alert("Привет вам из JavaScript");
    alert("Привет вам из JavaScript");
    alert("Привет вам из JavaScript");
    alert("Привет вам из JavaScript");
    alert("Привет вам из JavaScript");
  };
})();
```

- Переменная `isVisible` заменена на `true`, после чего `if` стало возможным убрать.
- Переменная `hi` заменена на строку.

Казалось бы — зачем менять `hi` на строку? Ведь код стал ощутимо длиннее!

..Но всё дело в том, что минификатор знает, что дальше код будет сжиматься при помощи gzip. Во всяком случае, все правильно настроенные сервера так делают.

Алгоритм работы gzip [10] заключается в том, что он ищет повторы в данных и выносит их в специальный «словарь», заменяя на более короткий идентификатор. Архив как раз и состоит из словаря и данных, в которых дубликаты заменены на идентификаторы.

Если вынести строку обратно в переменную, то получится как раз частный случай такого сжатия — взяли `"Привет вам из JavaScript"` и заменили на идентификатор `hi`. Но gzip справляется с этим лучше, поэтому эффективнее будет оставить именно строку. Gzip сам найдёт дубликаты и сожмёт их.

Плюс такого подхода станет очевиден, если сжать gzip оба кода — до и после минификации. Минифицированный gzip-сжатый код в итоге даст меньший размер.

Разные мелкие оптимизации

Кроме основных оптимизаций, описанных выше, есть ещё много мелких:

- Убираются лишние кавычки у ключей

```
{"prop" : "val" }   =>  {prop:"val"}
```

- Упрощаются простые вызовы `Array/Object`

```
a = new Array()   =>  a = []
o = new Object()  => o = {}
```

  Эта оптимизация предполагает, что `Array` и `Object` не переопределены программистом. Для включения её в UglifyJS нужен флаг `--unsafe`.

- …И еще некоторые другие мелкие изменения кода…

Подводные камни

Описанные оптимизации, в целом, безопасны, но есть ряд подводных камней.

Конструкция `with`

Рассмотрим код:

```
function changePosition(style) {
  var position, test;

  with (style) {
    position = 'absolute';
  }
}
```

Куда будет присвоено значение `position = 'absolute'`?

Это неизвестно до момента выполнения: если свойство `position` есть в `style` — то туда, а если нет — то в локальную переменную.

Можно ли в такой ситуации заменить локальную переменную на более короткую? Очевидно, нет:

```
function changePosition(style) {
  var a, b;

  with (style) {          // а что, если в style нет такого свойства?
    position = 'absolute';// куда будет осуществлена запись? в window.position?
  }
}
```

Такая же опасность для сжатия кроется в использованном `eval`. Ведь `eval` может обращаться к локальным переменным:

```
function f(code) {
  var myVar;

  eval(code); // а что, если будет присвоение eval("myVar = ...") ?

  alert(myVar);
```

Получается, что при наличии `eval` мы не имеем права переименовывать локальные переменные. Причём (!), если функция является вложенноой, то и во внешних функциях тоже.

А ведь сжатие переменных — очень важная оптимизация. Как правило, она уменьшает размер сильнее всего.

Что делать? Разные минификаторы поступают по-разному.

- UglifyJS — не будет переименовывать переменные. Так что наличие `with/eval` сильно повлияет на степень сжатие кода.
- GCC — всё равно сожмёт локальные переменные. Это, конечно же, может привести к ошибкам, причём в сжатом коде, отлаживать который не очень-то удобно. Поэтому он выдаст предупреждение о наличии опасной конструкции.

Ни тот ни другой вариант нас, по большому счёту, не устраивают.

Для того, чтобы код сжимался хорошо и работал правильно, не используем `with` и `eval`.

Либо, если уж очень надо использовать — делаем это с оглядкой на поведение минификатора, чтобы не было проблем.

Условная компиляция IE

В IE поддерживается условное выполнение JavaScript [11] .

Синтаксис: `/*@cc_on код */`. Код выполнится в IE, например:

```
var isIE /*@cc_on =true@*/;

alert(isIE); // true в IE.
```

Там же доступны и дополнительные директивы: `@_jscript_version`, `@if` и т.п., но речь здесь не о том.

Для минификаторов этот «условный» комментарий — всего лишь обычный комментарий. Они его удалят. Получится, что код не поймёт, где же IE.

Что делать?

1. Первое и наиболее корректное решение — не использовать условную компиляцию.
2. Второе, если уж очень надо — применить хак, завернуть его в `eval` или `new Function` (чтобы сжиматель не ругался):

```
var isIE = new Function('', '/*@cc_on return true@*/')();

alert(isIE); // true в IE.
```

В следующих главах мы посмотрим, какие продвинутые возможности есть в минификаторах, как сделать сжатие более эффективным.

Улучшаем сжатие кода

Здесь мы обсудим разные приёмы, которые используются, чтобы улучшить сжатие кода.

Больше локальных переменных

Например, код jQuery обёрнут в функцию, запускаемую «на месте».

```
(function(window, undefined) {
  // ...
  var jQuery = ...

  window.jQuery = jQuery; // сделать переменную глобальной

})(window);
```

Переменные `window` и `undefined` стали локальными. Это позволит сжимателю заменить их на более короткие.

ООП без прототипов

Приватные переменные будут сжаты и заинлайнены.

Например, этот код хорошо сожмётся:

```
function User(firstName, lastName) {
  var fullName = firstName + ' ' + lastName;

  this.sayHi = function() {
    showMessage(fullName);
  }

  function showMessage(msg) {
    alert('**' + msg + '**');
  }
}
```

..А этот — плохо:

```
function User(firstName, lastName) {
  this._firstName = firstName;
  this._lastName = lastName;
}

User.prototype.sayHi = function() {
  this._showMessage(this._fullName);
}


User.prototype._showMessage = function(msg) {
  alert('**' + msg + '**');
}
```

Сжимаются только локальные переменные, свойства объектов не сжимаются, поэтому эффект от сжатия для второго кода будет совсем небольшим.

При этом, конечно, нужно иметь в виду общий стиль ООП проекта, достоинства и недостатки такого подхода.

Сжатие под платформу, `define`

Можно делать разные сборки в зависимости от платформы (мобильная/десктоп) и браузера.

Ведь не секрет, что ряд функций могут быть реализованы по разному, в зависимости от того, поддерживает ли среда выполнения нужную возможность.

Способ 1: локальная переменная

Проще всего это сделать локальной переменной в модуле:

```
(function($) {

    /** @const */
    var platform = 'IE';

    // .....

    if (platform == 'IE') {
        alert('IE');
    } else {
        alert('NON-IE');
    }

})(jQuery);
```

Нужное значение директивы можно вставить при подготовке JavaScript к сжатию.

Сжиматель заинлайнит её и оптимизирует соответствующие IE.

Способ 2: `define`

UglifyJS и GCC позволяют задать значение глобальной переменной из командной строки.

В GCC эта возможность доступна лишь в «продвинутом режиме» работы оптимизатора, который мы рассмотрим далее (он редко используется).

Удобнее в этом плане устроен UglifyJS. В нём можно указать флаг `-d SYMBOL[=VALUE]`, который заменит все переменные `SYMBOL` на указанное значение `VALUE`. Если `VALUE` не указано, то оно считается равным `true`.

Флаг не работает, если переменная определена явно.

Например, рассмотрим код:

```
// my.js
if (isIE) {
  alert("Привет от IE");
} else {
  alert("Не IE :)");
}
```

Сжатие вызовом `uglifyjs -d isIE my.js` даст:

```
alert("Привет от IE");
```

..Ну а чтобы код работал в обычном окружении, нужно определить в нём значение переменной по умолчанию. Это обычно делается в каком-то другом файле (на весь проект), так как если объявить `var isIE` в этом, то флаг `-d isIE` не сработает.

Но можно и «хакнуть» сжиматель, объявив переменную так:

```
// объявит переменную при отсутствии сжатия
// при сжатии не повредит
window.isIE = window.isIE || getBrowserVersion();
```

Убираем вызовы `console.*`

Минификатор имеет в своём распоряжении дерево кода и может удалить ненужные вызовы. Хотя по умолчанию в UglifyJS и GCC такого флага нет, код минификатора можно легко расширить, добавив эту возможность.

- Для UglifyJS функция обхода дерева:

```
function ast_squeeze_console(ast) {
    var w = pro.ast_walker(),
        walk = w.walk,
        scope;
    return w.with_walkers({
        "stat": function(stmt) {
            if(stmt[0] === "call" && stmt[1][0] == "dot" && stmt[1][1] instanceof Array && stmt[1][1][0] == 'name' && stmt[1][1][1] == "console") {
                return ["block"];
            }
            return ["stat", walk(stmt)];
        },
        "call": function(expr, args) {
            if(expr[0] == "dot" && expr[1] instanceof Array && expr[1][0] == 'name' && expr[1][1] == "console") {
                return ["atom", "0"];
            }
        }
    }, function() {
        return walk(ast);
    });
};
```

  Полный код, использующий эту функцию и модуль uglify-js для сжатия с убиранием вызова: myuglify.js [12].

  Вы можете добавить свои любимые опции и использовать его вместо поставляемого «из коробки».

- В GCC соответствующие опции называются `stripNameSuffixes` и `stripTypePrefixes`, но они скрыты при запуске минификатора из командной строки или через веб-сервис.

  Чтобы их использовать, нужно либо поставить утилиту plovr [13], что эквивалентно стрельбе из пушки по воробьям, либо написать свой Java-код для вызова компилятора, который будет ставить опции.

  Выглядеть этот код может примерно так [14].

Аналогичным образом можно убрать и любой другой код, например вызовы вашего собственного логгера.

GCC: продвинутые оптимизации

Продвинутый режим оптимизации google closure compiler включается опцией `--compilation_level ADVANCED_OPTIMIZATIONS`.

Слово «продвинутый» (advanced) здесь, пожалуй, не совсем подходит. Было бы более правильно назвать его «супер-агрессивный-ломающий-ваш-неподготовленный-код-режим». Кардинальное отличие применяемых оптимизаций от обычных (simple) — в том, что они небезопасны.

Чтобы им пользоваться — надо уметь это делать.

Основной принцип продвинутого режима

- Если в обычном режиме переименовываются только локальные переменные внутри функций, то в «продвинутом» — на более короткие имена заменяется все.
- Если в обычном режиме удаляется недостижимый код после `return`, то в продвинутом — вообще весь код, который не вызывается в явном виде.

Например, если запустить продвинутую оптимизацию на таком коде:

```
// my.js
function test(node) {
  node.innerHTML = "newValue"
}
```

Строка запуска компилятора:

```
java -jar compiler.jar --compilation_level ADVANCED_OPTIMIZATIONS --js my.js
```

…То результат будет — пустой файл. Google Closure Compiler увидит, что функция `test` не используется, и с чистой совестью вырежет ее.

А в следующем скрипте функция сохранится:

```
function test(n) {
  alert("this is my test number " + n);
}
test(1);
test(2);
```

После сжатия:

```
function a(b) {
  alert("this is my test number " + b)
}
a(1);
a(2);
```

Здесь в скрипте присутствует явный вызов функции, поэтому она сохранилась.

Конечно, есть способы, чтобы сохранить функции, вызов которых происходит вне скрипта, и мы их обязательно рассмотрим.

Продвинутый режим сжатия не предусматривает сохранения глобальных переменных. Он переименовывает, инлайнит, удаляет вообще все символы, кроме зарезервированных.

Иначе говоря, продвинутый режим (ADVANCED_OPTIMIZATIONS), в отличие от простого (SIMPLE_OPTIMIZATIONS — по умолчанию), вообще не заботится о доступности кода извне и сохранении ссылочной целостности относительно внешних скриптов.

Единственное, что он гарантирует — это внутреннюю ссылочную целостность, и то — при соблюдении ряда условий и практик программирования.

Собственно, за счет такого агрессивного подхода и достигается дополнительный эффект оптимизации и сжатия скриптов.

То есть, продвинутый режим - это не просто «улучшенный обычный», а принципиально другой, небезопасный и обфусцирующий подход к сжатию.

Этот режим является «фирменной фишкой» Google Closure Compiler, недоступной при использовании других компиляторов.

Для того, чтобы эффективно сжимать Google Closure Compiler в продвинутом режиме, нужно понимать, что и как он делает. Это мы сейчас обсудим.

Сохранение ссылочной целостности

Чтобы использовать сжатый скрипт, мы должны иметь возможность вызывать функции под теми именами, которые им дали.

То есть, перед нами стоит задача *сохранения ссылочной целостности*, которая заключается в том, чтобы обеспечить доступность нужных функций для обращений по исходному имени извне скрипта.

Существует два способа сохранения внешней ссылочной целостности: экстерны и экспорты. Мы в подробностях рассмотрим оба, но перед этим необходимо упомянуть о модулях — другой важнейшей возможности GCC.

Модули

При сжатии GCC можно указать одновременно много JavaScript-файлов. «Эка невидаль, » — скажете вы, и будете правы. Да, пока что ничего особого.

Но в дополнение к этому можно явно указать, какие исходные файлы сжать в какие файлы результата. То есть, разбить итоговую сборку на модули.

Так что страницы могут грузить модули по мере надобности. Например, по умолчанию — главный, а дополнительная функциональность — загружаться лишь там, где она нужна.

Для такой сборки используется флаг компилятора `--module имя:количество файлов`.

Например:

```
java -jar compiler.jar --js base.js --js main.js  --js admin.js --module
first:2 --module second:1:first
```

Эта команда создаст модули: first.js и second.js.

Первый модуль, который назван «first», создан из объединённого и оптимизированного кода первых двух файлов (`base.js` и `main.js`).

Второй модуль, который назван «second», создан из `admin.js` — это следующий аргумент `--js` после включенных в первый модуль.

Второй модуль в нашем случае зависит от первого. Флаг `--module second:1:first` как раз означает, что модуль `second` будет создан из одного файла после вошедших в предыдущий модуль (`first`) и зависит от модуля `first`.

А теперь — самое вкусное.

Ссылочная целостность между всеми получившимися файлами гарантируется.

Если в одном функция `doFoo` заменена на `b`, то и в другом тоже будет использоваться `b`.

Это означает, что проблем между JS-файлами не будет. Они могут свободно вызывать друг друга без экспорта, пока находятся в единой

модульной сборке.

Экстерны

Экстерн (extern) — имя, которое числится в специальном списке компилятора. Он должен быть определен вне скрипта, в файле экстернов.

Компилятор никогда не переименовывает экстерны.

Например:

```
document.onkeyup = function(event) { alert(event.type) }
```

После продвинутого сжатия:

```
document.onkeyup = function(a) { alert(a.type) }
```

Как видите, переименованной оказалась только переменная `event`. Такое переименование заведомо безопасно, т.к. `event` — локальная переменная.

Почему компилятор не тронул остального? Попробуем другой вариант:

```
document.blabla = function(event) { alert(event.megaProperty) }
```

После компиляции:

```
document.a = function(a) { alert(a.b) }
```

Теперь компилятор переименовал и `blabla` и `megaProperty`.

Дело в том, что названия, использованные до этого, были во внутреннем списке экстернов компилятора. Этот список охватывает основные объекты браузеров и находится (под именем `externs.zip`) в корне архива `compiler.jar`.

Компилятор переименовывает имя списка экстернов только когда так названа локальная переменная.

Например:

```
window.resetNode = function(node) {
  var innerHTML = "test";
  node.innerHTML = innerHTML;
}
```

На выходе:

```
window.a = function(a) {
  a.innerHTML = "test"
};
```

Как видите, внутренняя переменная `innerHTML` не просто переименована - она заинлайнена (заменена на значение). Так как переменная локальна, то любые действия внутри функции с ней безопасны.

А свойство `innerHTML` не тронуто, как и объект `window` — так как они в списке экстернов и не являются локальными переменными.

Это приводит к следующему побочному эффекту. Иногда свойства, которые следовало бы сжать, не сжимаются. Например:

```
window['User'] = function(name, type, age) {
    this.name = name
    this.type = type
    this.age = age
}
```

После сжатия:

```
window.User = function(a, b, c) {
  this.name = a;
  this.type = b;
  this.a = c
};
```

Как видно, свойство `age` сжалось, а `name` и `type` — нет. Это побочный эффект экстернов: `name` и `type` — в списке объектов браузера, и компилятор просто старается не наломать дров.

Поэтому отметим еще одно полезное правило оптимизации:

Названия своих свойств не должны совпадать с зарезервированными словами (экстернами). Тогда они будут хорошо сжиматься.

Для задания списка экстернов их достаточно перечислить в файле и указать этот файл флагом `--externs <файл экстернов.js>`.

При перечислении объектов в файле экстернов - объявляйте их и перечисляйте свойства. Все эти объявления никуда не идут, они используются только для создания списка, который обрабатывается компилятором.

Например, файл `myexterns.js`:

```
var dojo = {}
dojo._scopeMap;
```

Использование такого файла при сжатии (опция `--externs myexterns.js`) приведет к тому, что все обращения к символам `dojo` и к `dojo._scopeMap` будут не сжаты, а оставлены «как есть».

Экспорт

Экспорт — программный ход, основанный на следующем правиле поведения компилятора.

Компилятор заменяет обращения к свойствам через кавычки на точку, и при этом не трогает название свойства.

Например, `window['User']` превратится в `window.User`, но не дальше.

Таким образом можно *«экспортировать»* нужные функции и объекты:

```
function SayWidget(elem) {
  this.elem = elem
  this.init()
}
window['SayWidget'] = SayWidget;
```

На выходе:

```
function a(b) {
  this.a = b;
  this.b()
}
window.SayWidget = a;
```

Обратим внимание — сама функция `SayWidget` была переименована в `a`. Но затем — экспортирована как `window.SayWidget`, и таким образом доступна внешним скриптам.

Добавим пару методов в прототип:

```
function SayWidget(elem) {
  this.elem = elem;
  this.init();
}

SayWidget.prototype = {
  init: function() {
    this.elem.style.display = 'none'
  },

  setSayHandler: function() {
    this.elem.onclick = function() {
      alert("hi")
    };
  }
}

window['SayWidget'] = SayWidget;
SayWidget.prototype['setSayHandler'] = SayWidget.prototype.setSayHandler;
```

После сжатия:

```
function a(b) {
  this.a = b;
  this.b()
}
a.prototype = {b:function() {
  this.a.style.display = "none"
}, c:function() {
  this.a.onclick = function() {
    alert("hi")
  }
}};
window.SayWidget = a;
a.prototype.setSayHandler = a.prototype.c;
```

Благодаря строке

```
SayWidget.prototype['setSayHandler'] = SayWidget.prototype.setSayHandler
```

метод `setSayHandler` экспортирован и доступен для внешнего вызова.

Сама строка экспорта выглядит довольно глупо. По виду — присваиваем свойство самому себе.

Но логика сжатия GCC работает так, что такая конструкция является экспортом. Справа переименование свойства `setSayHandler`

происходит, а слева — нет.

Планируйте жизнь после сжатия

Рассмотрим следующий код:

```
window['Animal'] = function() {
  this.blabla = 1;
  this['blabla'] = 2;
}
```

После сжатия:

```
window.Animal = function() {
  this.a = 1;
  this.blabla = 2
};
```

Как видно, первое обращение к свойству `blabla` сжалось, а второе (как и все аналогичные) — преобразовалось в синтаксис через точку.
В результате получили некорректное поведение кода.

Так что, используя продвинутый режим оптимизации, планируйте поведение кода после сжатия.

Если где-то возможно обращение к свойствам через квадратные скобки по полному имени — такое свойство должно быть экспортировано.

`goog.exportSymbol` и `goog.exportProperty`

В библиотеке Google Closure Library [15] для экспорта есть специальная функция `goog.exportSymbol`. Вызывается так:

```
goog.exportSymbol('my.SayWidget', SayWidget)
```

Эта функция по сути работает также, как и рассмотренная выше строка с присвоением свойства, но при необходимости создает нужные объекты.

Она аналогична коду:

```
window['my'] = window['my'] || {}
window['my']['SayWidget'] = SayWidget
```

То есть, если путь к объекту не существует — `exportSymbol` создаст нужные пустые объекты.

Функция `goog.exportProperty` экспортирует свойство объекта:

```
goog.exportProperty(SayWidget.prototype, 'setSayHandler', SayWidget.prototype.setSayHandler)
```

Строка выше - то же самое, что и:

```
SayWidget.prototype['setSayHandler'] = SayWidget.prototype.setSayHandler
```

Зачем они нужны, если все можно сделать простым присваиванием?

Основная цель этих функций — во взаимодействии с Google Closure Compiler. Они дают информацию компилятору об экспортах, которую он может использовать.

Например, есть недокументированная внутренняя опция `externExportsPath`, которая генерирует из всех экспортов файл экстернов. Таким образом можно распространять откомпилированный JavaScript-файл как внешнюю библиотеку, с файлом экстернов для удобного внешнего связывания.

Кроме того, экспорт через эти функциями удобен и нагляден.

Если вы используете продвинутый режим оптимизации, то можно взять их из файла base.js Google Closure Library. Можно и подключить этот файл целиком — оптимизатор при продвинутом сжатии вырежет из него почти всё лишнее, так что overhead будет минимальным.

Отличия экспорта от экстерна

Между экспортом и экстерном есть кое-что общее. И то и другое дает возможность доступа к объектам под исходным именем, до переименования.

Но, в остальном, это совершенно разные вещи.

Экстерн	Экспорт
Служит для тотального запрета на переименование всех обращений к свойству. Задумано для сохранения обращений к стандартным объектам браузера, внешним библиотекам.	Служит для открытия доступа к свойству извне под указанным именем. Задумано для открытия внешнего интерфейса к сжатому скрипту.
Работает со свойством, объявленным вне скрипта. Вы не можете объявить новое свойство в скрипте и сделать его экстерном.	Создает ссылку на свойство, объявленное в скрипте.
Если `window` - экстерн, то все обращения к `window` в скрипте останутся как есть.	Если `user` экспортируется, то создается только одна ссылка под полным именем, а все остальные обращения будут сокращены.

Стиль разработки

Посмотрим, как сжиматель поведёт себя на следующем, типичном, объявлении библиотеки:

```
(function(window, undefined) {

  // пространство имен и локальная перменная для него
  var MyFramework = window.MyFramework = {};

  // функция фреймворка, доступная снаружи
  MyFramework.publicOne = function() {
    makeElem();
  };

  // приватная функция фреймворка
  function makeElem() {
    var div = document.createElement('div');
    document.body.appendChild(div);
  }

  // еще какая-то функция
  MyFramework.publicTwo = function() {};

})(window);

// использование
MyFramework.publicOne();
```

Результат компиляции в обычном режиме:

```
// java -jar compiler.jar --js myframework.js --formatting PRETTY_PRINT
(function(a) {
  a = a.MyFramework = {};
  a.publicOne = function() {
    var a = document.createElement("div");
    document.body.appendChild(a)
  };
  a.publicTwo = function() {
  }
})(window);
MyFramework.publicOne();
```

Это — примерно то, что мы ожидали. Неиспользованный метод `publicTwo` остался, локальные свойства переименованы и заинлайнены.

А теперь продвинутый режим:

```
// --compilation_level ADVANCED_OPTIMIZATIONS
window.a = {};
MyFramework.b();
```

Оно не работает! Компилятор попросту не разобрался, что и как вызывается, и превратил рабочий JS-файл в один сплошной баг.

В зависимости от версии GCC у вас может быть и что-то другое.

Всё дело в том, что такой стиль объявления нетипичен для инструментов, которые в самом Google разрабатываются и сжимаются этим минификатором.

Типичный правильный стиль:

```
// пространство имен и локальная перменная для него
var MyFramework = {};

MyFrameWork._makeElem = function() {
  var div = document.createElement('div');
  document.body.appendChild(div);
};

MyFramework.publicOne = function() {
  MyFramework._makeElem();
};

MyFramework.publicTwo = function() {};

// использование
MyFramework.publicOne();
```

Обычное сжатие здесь будет бесполезно, а вот продвинутый режим идеален:

```
// в зависимости от версии GCC результат может отличаться
MyFrameWork.a = function() {
  var a = document.createElement("div");
  document.body.appendChild(a)
};
MyFrameWork.a();
```

Google Closure Compiler не только разобрался в структуре и удалил лишний метод - он заинлайнил функции, чтобы итоговый размер получился минимальным.

Как говорится, преимущества налицо.

Резюме

Продвинутый режим оптимизации сжимает, оптимизирует и, при возможности, удаляет все свойства и методы, за исключением экстернов.

Это является принципиальным отличием, по сравнению с другими упаковщиками.

Отказ от сохранения внешней ссылочной целостности с одной стороны позволяет увеличить уровень сжатия, но требует поддержки со стороны разработчика.

Основная проблема этого сжатия — усложнение разработки. Добавляется дополнительный уровень возможных проблем: сжатие. Конечно, можно отлаживать и сжатый код, для этого придуманы Source Maps [16], но клиентская разработка и без того достаточно сложна.

Поэтому его используют редко.

Как правило, есть две причины для использования продвинутого режима:

1. **Обфускация кода.**

   Если в коде после обычного сжатия ещё как-то можно разобраться, то после продвинутого — уже нет. Всё переименовано и заинлайнено. В теории это, конечно, возможно, но «порог входа» в такой код несоизмеримо выше.

   Судя по виду скриптов на сайтах, созданных Google, сам Google жмет свои скрипты именно продвинутым режимом оптимизации. И библиотека Google Closure Library тоже рассчитана на него.

2. **Хорошие сжатие виджетов, счётчиков.**

Небольшой код, который отдаётся наружу, может быть сжат в продвинутом режиме. Так как он небольшой — все ошибки можно легко исправить, а продвинутый режим гарантирует наилучшее сжатие.

GCC: статическая проверка типов

Google Closure Compiler, как и любой кошерный компилятор, старается проверить правильность кода и предупредить о возможных ошибках.

Первым делом он, разумеется, проверяет структуру кода и сразу же выдает такие ошибки как пропущенная скобка или лишняя запятая.

Но, кроме этого, он умеет проверять типы переменных, используя как свои собственные знания о встроенных javascript-функциях и преобразованиях типов,
так и информацию о типах из JSDoc, указываемую javascript-разработчиком.

Это обеспечивает то, чем так гордятся компилируемые языки — статическую проверку типов, что позволяет избежать лишних ошибок во время выполнения.

Для вывода предупреждений при проверки типов используется флаг `--jscomp_warning checkTypes`.

Задание типа при помощи аннотации

Самый очевидный способ задать тип — это использовать аннотацию. Полный список аннотаций вы найдете в документации [17].

В следующем примере параметр `id` функции `f1` присваивается переменной `boolVar` другого типа:

```
1 /** @param {number} id */
2 function f(id) {
3   /** @type {boolean} */
4   var boolVar;
5
6   boolVar = id; // (!)
7 }
```

Компиляция с флагом `--jscomp_warning checkTypes` выдаст предупреждение:

```
1 f.js:6: WARNING - assignment
2 found   : number
3 required: boolean
4         boolVar = id; // (!)
5         ^
```

Действительно: произошло присвоение значения типа `number` переменной типа `boolean`.

Типы отслеживаются по цепочке вызовов.

Еще пример, на этот раз вызов функции с некорректным параметром:

```
/** @param {number} id */
function f1(id) {
  f2(id); // (!)
}

/** @param {string} id */
function f2(id) { }
```

Такой вызов приведёт к предупреждению со стороны минификатора:

```
f2.js:3: WARNING - actual parameter 1 of f2 does not match formal parameter
found   : number
required: string
  f2(id); // (!)
     ^
```

Действительно, вызов функции `f2` произошел с числовым типом вместо строки.

Отслеживание приведений и типов идёт при помощи графа взаимодействий и выведению (infer) типов, который строит GCC по коду.

Знания о преобразовании типов

Google Closure Compiler знает, как операторы javascript преобразуют типы. Такой код уже не выдаст ошибку:

```
/** @param {number} id */
function f1(id) {
  /** @type {boolean} */
  var boolVar;

  boolVar = !!id
}
```

Действительно - переменная преобразована к типу boolean двойным оператором НЕ.
А код `boolVar = 'test-'+id` выдаст ошибку, т.к. конкатенация со строкой дает тип `string`.

Знание о типах встроенных функций, объектные типы

Google Closure Compiler содержит описания большинства встроенных объектов и функций javascript вместе с типами параметров и результатов.

Например, объектный тип `Node` соответствует узлу DOM.

Пример некорректного кода:

```
/** @param {Node} node */
function removeNode(node) {
  node.parentNode.removeChild(node)
}
document.onclick = function() {
  removeNode("123")
}
```

Выдаст предупреждение

```
f3.js:7: WARNING - actual parameter 1 of removeNode does not match formal parameter
found   : string
required: (Node|null)
  removeNode("123")
             ^
```

Обратите внимание - в этом примере компилятор выдает `required: Node|null`. Это потому, что указание объектного типа (не элементарного) подразумевает, что в функцию может быть передан `null`.

В следующем примере тип указан жестко, без возможности обнуления:

```
/** @param {!Node} node */
function removeNode(node) {
    node.parentNode.removeChild(node)
}
```

Восклицательный знак означает, что параметр обязателен.

Найти описания встроенных типов и объектов javascript вы можете в файле экстернов: `externs.zip` находится в корне архива `compiler.jar`, или в соответствующей директории SVN: http://closure-compiler.googlecode.com/svn/trunk/externs/.

Интеграция с проверками типов из Google Closure Library

В Google Closure Library есть функции проверки типов: `goog.isArray`, `goog.isDef`, `goog.isNumber` и т.п.

Google Closure Compiler знает о них и понимает, что внутри следующего `if` переменная может быть только функцией:

```
var goog = {
  isFunction: function(f) { return typeof f == 'function' }
}

if (goog.isFunction(func)) {
  func.apply(1, 2)
}
```

Сжатие с проверкой выдаст предупреждение:

```
f.js:6: WARNING - actual parameter 2 of Function.apply does not match formal parameter
found   : number
required: (Object|null|undefined)
  func.apply(1, 2)
                ^      ^
```

То есть, компилятор увидел, что код, использующий `func` находится в `if (goog.isFunction(func))` и сделал соответствующий вывод, что это в этой ветке `func` является функцией, а значит вызов `func.apply(1,2)` ошибочен (второй аргумент не может быть числом).

Дело тут именно в интеграции с Google Closure Library. Если поменять `goog` на `g` — предупреждения не будет.

Резюме

Из нескольких примеров, которые мы рассмотрели, должна быть понятна общая логика проверки типов.

Соответствующие различным типам и ограничениям на типы аннотации вы можете найти в Документации Google [18]. В частности, возможно указание нескольких возможных типов, типа `undefined` и т.п.

Также можно указывать количество и тип параметров функции, ключевого слова `this`, объявлять классы, приватные методы и интерфейсы.

Проверка типов javascript, предоставляемая Google Closure Compiler - пожалуй, самая продвинутая из существующих на сегодняшний день.

С ней аннотации, документирующие типы и параметры, становятся не просто украшением, а реальным средством проверки, уменьшающим количество ошибок на production.

Очень подробно проверка типов описана в книге Closure: The Definitive Guide [19], автора Michael Bolin.

GCC: интеграция с Google Closure Library

Google Closure Compiler содержит ряд специальных возможностей для интеграции с Google Closure Library.

Здесь важны две вещи.

1. Для их использования возможно использовать минимум от Google Closure Library. Например, взять одну или несколько функций из библиотеки.
2. GCC — расширяемый компилятор, можно добавить к нему свои «фазы оптимизации» для интеграции с другими инструментами и фреймворками.

Интеграция с Google Closure Library подключается флагом `--process_closure_primitives`, который по умолчанию установлен в `true`. То есть, она включена по умолчанию.

Этот флаг запускает специальный проход компилятора, описанный классом `ProcessClosurePrimitives` и подключает дополнительную проверку типов `ClosureReverseAbstractInterpreter`.

Мы рассмотрим все действия, которые при этом происходят, а также некоторые опции, которые безопасным образом используют символы Google Closure Library без объявления флага.

Преобразование основных символов

Следующие действия описаны в классе `ProcessClosurePrimitives`.

Замена константы `COMPILED`

В Google Closure Library есть переменная:

```
1 /**
2  * @define {boolean} ...
3  */
4 var COMPILED = false;
```

Проход `ProcessClosurePrimitives` переопределяет ее в `true` и использует это при оптимизациях, удаляя ветки кода, не предназначены для запуска на production.

Такие функции существуют, например, в ядре Google Closure Library. К ним в первую очередь относятся вызовы, предназначенные для сборки и проверки зависимостей. Они содержат код, обрамленный проверкой `COMPILED`, например:

```
goog.require = function(rule) {
   // ...
  if (!COMPILED) {
     // основное тело функции
  }
}
```

Аналогично может поступить и любой скрипт, даже без использования Google Closure Library:

```
/** @define {boolean} */
var COMPILED = false

Framework = { }

Framework.sayCompiled = function() {
    if (!COMPILED) {
        alert("Not compressed")
    } else {
        alert("Compressed")
    }
}
```

Для того, чтобы сработало, нужно сжать в продвинутом режиме:

```
Framework = {};
Framework.sayCompiled = Framework.a = function() {
  alert("Compressed");
};
```

Компилятор переопределил `COMPILED` в `true` и произвел соответствующие оптимизации.

Автоподстановка локали

В Google Closure Compiler есть внутренняя опция `locale`

Эта опция переопределяет переменную `goog.LOCALE` на установленную при компиляции.

Для использования опции `locale`, на момент написания статьи, ее нужно задать в Java коде компилятора, т.к. соответствующего флага нет.

Как и `COMPILED`, константу `goog.LOCALE` можно и использовать в своем коде без библиотеки Google Closure Library.

Проверка зависимостей

Директивы `goog.provide`, `goog.require`, `goog.addDependency` обрабатываются особым образом.

Все зависимости проверяются, а сами директивы проверки — удаляются из сжатого файла.

Экспорт символов

Вызов `goog.exportSymbol` задаёт экспорт символа.

Если подробнее, то код `goog.exportSymbol('a',myVar)` эквивалентен
`window['a'] = myVar`.

Автозамена классов CSS

Google Closure Library умеет преобразовывать классы CSS на более короткие по списку, который задаётся при помощи `goog.setCssNameMapping`.

Например, следующая функция задает такой список.

```
goog.setCssNameMapping({
    "goog-menu": "a",
    "goog-menu-disabled": "a-b",
    "CSS_LOGO": "b",
    "hidden": "c"
});
```

Тогда следующий вызов преобразуется в «a a-b»:

```
goog.getCssName('goog-menu') + ' ' + goog.getCssName('goog-menu', 'disabled')
```

Google Closure Compiler производит соответствующие преобразования в сжатом файле и удаляет вызов `setCssNameMapping` из кода.

Чтобы это сжатие работало, в HTML/CSS классы тоже должны сжиматься. По всей видимости, в приложениях Google это и происходит, но соответствующие инструменты закрыты от публики.

Генерация списка экстернов

При объявлении внутренней опции `externExportsPath`, содержащей путь к файлу, в этот файл будут записаны все экспорты, описанные через `goog.exportSymbol`/`goog.exportProperty`.

В дальнейшем этот файл может быть использован как список экстернов для компиляции.

Эта опция может быть полезна для создания внешних библиотек, распространяемых со списком экстернов.

Для её использования нужна своя обёртка вокруг компилятора на Java. Соответствующий проход компилятора описан в классе `ExternExportsPass`.

Проверка типов

В Google Closure Library есть ряд функций для проверки типов. Например: `goog.isArray`, `goog.isString`, `goog.isNumber`, `goog.isDef` и т.п.

Компилятор использует их для проверки типов, более подробно см. GCC: статическая проверка типов [20]

Эта логика описана в классе `ClosureReverseAbstractInterpreter`. Названия функций, определяющих типы, жестко прописаны в Java-коде, поменять их на свои без модификации исходников нельзя.

Автогенерация экспортов из аннотаций

Для этого в Google Closure Compiler есть внутренняя опция `generateExports`.

Эта недокументированная опция добавляет проход компилятора, описанный классом `GenerateExports`.

Он читает аннотации `@export` и создает из них экспортирующие вызовы `goog.exportSymbol/exportProperty`. Название экспортирующих функций находится в классе соглашений кодирования, каким по умолчанию является `GoogleCodingConvention`.

Например:

```
/** @export */
function Widget() {
}
/** @export */
Widget.prototype.hide = function() {
  this.elem.style.display = 'none'
}
```

После компиляции в продвинутом режиме:

```
function a() {
}
goog.d("Widget", a);
a.prototype.a = function() {
  this.b.style.display = "none"
};
goog.c(a.prototype, "hide", a.prototype.a);
```

Свойства благополучно экспортированы. Удобно.

Резюме

Google Closure Compiler содержит дополнительные фичи, облегчающие интеграцию с Google Closure Library. Некоторые из них весьма полезны, но требуют создания своего Java-файла, который ставит внутренние опции.

При обработке символов компилятор не смотрит, подключена ли библиотека, он находит обрабатывает их просто по именам. Поэтому вы можете использовать свою реализацию соответствующих функций.

Google Closure Compiler можно легко расширить, добавив свои опции и проходы оптимизатора, для интеграции с вашими инструментами.

Ссылки

1. Google Closure Compiler https://developers.google.com/closure/compiler/
2. UglifyJS https://github.com/mishoo/UglifyJS
3. Microsoft AJAX Minifier http://ajaxmin.codeplex.com/
4. Java http://java.oracle.com
5. Песочницу http://closure-compiler.appspot.com/home
6. Веб-сервис https://developers.google.com/closure/compiler/docs/gettingstarted_api?hl=ru
7. Node.js http://nodejs.org
8. DOT http://en.wikipedia.org/wiki/DOT_language
9. Graphviz http://www.graphviz.org/
10. Алгоритм работы gzip http://www.gzip.org/algorithm.txt
11. Условное выполнение JavaScript http://msdn.microsoft.com/en-us/library/121hztk3.aspx
12. Myuglify.js http://learn.javascript.ru/files/tutorial/compress/myuglify.js
13. Plovr http://plovr.com/
14. Примерно так https://groups.google.com/d/msg/closure-compiler-discuss/qqdnzikzpWA/8cPHKIR1svgJ
15. Google Closure Library https://developers.google.com/closure/library/
16. Source Maps http://www.html5rocks.com/en/tutorials/developertools/sourcemaps/
17. Документации http://code.google.com/intl/ru/closure/compiler/docs/js-for-compiler.html
18. Документации Google http://code.google.com/intl/ru/closure/compiler/docs/js-for-compiler.html
19. Closure: The Definitive Guide http://www.ozon.ru/context/detail/id/6089988/

20. GCC: статическая проверка типов http://learn.javascript.ru/gcc-check-types