VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

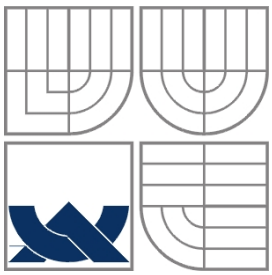
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

SPECIALIZOVANÝ INTERPRET JAZYKA JAVASCRIPT

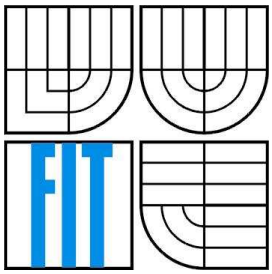
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. Jan Borůvka
AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

SPECIALIZOVANÝ INTERPRET JAZYKA JAVASCRIPT

SPECIALIZED INTERPRET OF JAVASCRIPT LANGUAGE

DIPLMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. Jan Borůvka

VEDOUCÍ PRÁCE
SUPERVISOR

Dr. Ing. Petr Peringer

BRNO 2008

Abstrakt

Cílem diplomové práce je navrhnout a implementovat interpret jazyka JavaScript, který je vytvářen pro potřebu obcházení zatemňovacích obálek některých druhů počítačových virů. Součástí práce je podrobný rozbor vnitřních mechanismů, pomocí kterých je v ECMAScript standardu přesně definováno chování jazyka.

Klíčová slova

Překladač, interpret, programovací jazyk, JavaScript, ECMAScript, uvolňování paměti, gramatika.

Abstract

The aim of this master‘s thesis is to design and implement JavaScript interpreter which is designed for purposes of avoiding obfuscation code of various types of computer viruses. This master’s thesis also comprises a detailed view into the inner mechanism of the ECMAScript standard.

Keywords

Compiler, interpreter, programming language, JavaScript, ECMAScript, garbage collector, grammar

Specializovaný interpret jazyka JavaScript

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Dr. Ing. Petra Peringera.

Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jan Borůvka
15.5.2008

Poděkování

Děkuji tímto vedoucímu mé diplomové práce za přátelský přístup, cenné rady, spoustu věnovaného času při konzultacích a poskytnutí materiálu k přípravě.

Jan Borůvka

Obsah

1 Úvod

V nedávné době nalezli tvůrci JavaScriptových virů nový způsob, jak do počítačů přenést infekci. Jedná se o rychle mutující JavaScriptový kód, který umí uniknout detekci antivirových programů vytvořením obálky kolem vlastního škodlivého kódu. Samotná obálka je samozřejmě na první pohled neškodná, a proto není samotný vir antivirovým programem odhalen. Během provádění kódu programu se ovšem sestavuje řetězec, který představuje nový spustitelný JavaScriptový program. Tento nový program je během vykonávání původního programu spuštěn, přičemž antivirový software již není schopen zjistit, co tento nový spuštěný program obsahuje (jsou spouštěny pomocí vestavěných funkcí JavaScriptu). Hlavním důvodem, proč tyto viry unikají detekci antivirových testů, je fakt, že jsou schopny v čase mutovat. Každý infikovaný počítač obdrží vlastní unikátní javascriptový kód, který je vytvářen zcela náhodně a v reálném čase. Generátor těchto JavaScriptových kódů mění v jednotlivých kopiích jak názvy proměnných a funkcí, tak i způsob jakým se řetězec, představující nový javascriptový program, vytváří.

Způsob, jak se vypořádat s takovýmto typem viru, se sám nabízí. Vytvořit specializovaný javascriptový interpret, kterému bude předán kontrolovaný kód. Tento JavaScriptový interpret bude provádět interpretaci kódu a v případě pokusu o spuštění nového kontextu JavaScriptového programu předá spouštěný kód k analyzování antivirovému programu. V případě, že je daný kód vyhodnocen jako nezávadný, pokračuje interpret dál v provádění kódu. Tímto způsobem se tedy dají obcházet „zatemňovací“ obálky vytvářené kolem potenciálně nebezpečné kódu, a to je také důvod vzniku této diplomové práce, která se vytvářením, takovéhoto specializovaného JavaScriptového interpretu zabývá. Hlavní kapitoly této práce se zabývají následujícími tématy:

- Kapitola 2 se zabývá studiem standardu jazyka ECMAScript. Důraz je kladen především na jednotlivé vnitřní mechanismy tohoto jazyka. Dále je v této kapitole zmíněna lexikální struktura jazyka a jednotlivé datové typy jazyka ECMAScript.
- Kapitola 3 je teoretickým úvodem do oblasti interpretačních překladačů. Také je zde uveden popis jednotlivých logických komponent překladače.
- Kapitola 4 obsahuje návrh jednotlivých tříd, ze kterých se bude skládat výsledný interpretační překladač.
- Kapitola 5 obsahuje implementační detaily výsledného programu. Dále jsou zde uvedeny výsledky testování a seznam pravidel gramatiky, pro které nebyla zatím implementovány sémantické akce.
- Kapitola „Závěr“ je shrnutím dosažených výsledků a naznačením, kterým směrem se měla další práce na projektu ubírat.

2 Jazyk JavaScript

JavaScript je obecně používaný název pro skriptovací jazyk, který byl původně vyvinut firmou Netscape a použit ve stejnojmenném internetovém prohlížeči. Jazyk následně začlenily do svých internetových prohlížečů i konkurenční firmy pod jinými jmény (JScript společnosti Microsoft vkládaný do jejich produktu Internet Explorer). Jazyk byl následně standardizován a pojmenován jako ECMAScript. Podle tohoto standardu postupuji i při tvorbě této diplomové práce. Konkrétně se jedná o standard ECMAScript-262 třetí vydání[6]. V textu budu používat obecně známý název JavaScript.

Menším problémem je fakt, že žádný internetový prohlížeč se zcela přesně neřídí touto specifikací. Během implementace jsem se snažil co nejpřesněji napodobit chování interpretu od společnosti Microsoft (konkrétně interpretu obsaženého v produktu IE7[1]). Mnou implementované odchylky od standardu uvedu později v textu.

JavaScript je interpretovaný programovací jazyk se základními objektově orientovanými schopnostmi, který je dynamicky typovaný. Ze syntaktické pohledu připomíná jazyk na první pohled nejvíce jazyky typu Java, C a C++. Na druhou stranu se od těchto jazyků výrazně liší způsobem, jakým je v programech budována dědičnost a mnoha dalšími aspekty (obor platnosti proměnných a další). Další detaily o jazyku JavaScript, které nejsou v této kapitole uvedeny lze nalézt v [6] a [7].

2.1 Lexikální struktura

Obecně je lexikální struktura jazyka sada nejzákladnější pravidel, definujících tvar proměnných, systém používání poznámek, způsob oddělování jednotlivých základních prvků jazyka a dalších. V této kapitole obecně popíši lexikální strukturu jazyka JavaScript. Přesná specifikace lexikální gramatiky je v ECMAScript specifikaci.

Znaková sada

Pro Javascript je použita znaková sada Unicode. Unicode je znaková sada kódovaná 16bity a znak v tomto kódování může tedy reprezentovat téměř jakýkoliv používaný znak. Interpret, který v rámci diplomové práce vytvářím, může tuto skutečnost zanedbat, jelikož bylo zadavatelem[2] určeno, že od nadřazeného programu bude vytvářený interpret dostávat na vstup pouze znaky ASCII. JavaScript patří mezi jazyky rozlišující malá a velká písmena.

[1] Windows Internet Explorer 7

[2] AVG Technologies CZ, s.r.o.

Oddělovače

Interpret jazyka JavaScript ignoruje veškeré tabulátory, mezery a znaky konce řádku, které oddělují jednotlivé tokeny (jednotlivé lexikální elementy). Tyto oddělovače samozřejmě nejsou ignorovány v případě, že jsou umístěny v rámci řetězce nebo regulárního výrazu.

Automatické vkládání středníků

Ve většině běžných programovacích jazyků je pravidlem, že každý jednoduchý příkaz musí být ukončen středníkem. V případě, že středník není přítomen, je běžně hlášena syntaktická chyba. V jazyce JavaScript je tato skutečnost mírně složitější. Existují případy kdy je možné středník vypustit (respektive je automaticky vložen):

- Příkaz je oddělen od následujícího příkazu znakem ukončující řádek
- Za příkazem se vyskytuje token „}“

Ukázka platného JavaScript kódu:

```
{ 1
2 } 3
```

Ukázka neplatného JavaScript kódu:

```
{ 1 2 } 3
```

Vzhledem k faktu, že tuto vlastnost jazyka nelze efektivně postihnout příslušnou gramatikou, jedná se z hlediska syntaktické analýzy (během níž by se mělo na případnou chybu přijít) o relativně velkou komplikaci.

Komentáře

V JavaScriptu je možné použít komentáře obdobným způsobem jako například u jazyků C nebo C++. Za komentář je tedy považováno cokoliv, co se nachází mezi dvojznakem „//“ a koncem řádku. Dále jakýkoliv text mezi znaky /* a */ je také považován za komentář.

Příklady komentářů:

```
// Komentář na jeden řádek
/* Komentář na
více
řádků */
```

Identifikátory

V jazyce JavaScript se identifikátory používají pro identifikaci proměnných, funkcí a pro vytváření návěští. Identifikátory musí začínat písmenem, podtržítkem „_” nebo znakem dolaru „$“. Znak dolaru je ve specifikaci zamýšlen pro použití v kódu, který je automaticky generován. Další znaky

identifikátoru mohou obsahovat navíc číslice. ECMAScript specifikace také povoluje sekvence escape Unicode. Tedy sekvence \uxxxx, kde x je šestnáctkové číslo. Unicode escape sekvence tedy určuje šestnácti-bitovou hodnotu Unicode znaku. Opět se Unicode hodnotami nemusím ve své práci zabývat, jelikož na vstup mého programu se budou dostávat pouze ASCII hodnoty.

Rezervovaná slova

JavaScript obsahuje tzv. rezervovaná slova, která v programu nemohou zastávat funkci identifikátorů. Tato rezervovaná slova jsou rozdělena do dvou skupin. První skupinu ukazuje tabulka 2.1. Jedná se o slova, která jsou využívaná jako klíčová slova – jsou tedy přímo součástí syntaxe jazyka.

break	else	new	var
case	finally	return	void
catch	for	switch	while
continue	function	this	with
default	if	throw	
delete	in	try	
do	instanceof	typeof	

Tabulka 2.1 – Rezervovaná slova

Druhou skupinu rezervovaný slov tvoří slova, která jsou do budoucna plánována jako klíčová. Tyto slova tedy také nelze použít jako identifikátory, i když nikde v syntaxi programu nejsou zastoupena. Výčet budoucích klíčových slov je v tabulce 3.1.

abstract	enum	int	short
boolean	export	interface	static
byte	extends	long	super
char	final	native	synchronized
class	float	package	throws
const	goto	private	transient
debugger	implements	protected	volatile
double	import	public	

Tabulka 2.2 -budoucí rezervovaná slova

2.2 Datové typy

Jak již bylo řečeno, je JavaScript jazykem dynamicky typovaným. Hodnota literálů, proměnných a různých vnitřních struktur může nabývat hodnot z celkem devíti různých typů (Nedefinovaný, Null, Boolean, Řetězec, Číslo, Objekt, Odkaz, Seznam a typ Dokončení). Datové typy JavaScriptu se dají rozdělit do dvou kategorií.

První kategorií jsou uživatelské datové typy. Jde o typy, které může uživatel jazyka JavaScript vytvářet a používat během vývoje aplikací, ať už jako literály nebo proměnné. Jedná se o tyto typy:

Nedefinovaný, Null, Boolean, Řetězec, Číslo, Objekt. Druhou kategorií datových typů jsou interní datové typy. Tyto datové již nejsou přímou součástí jazyka (respektive programátor je nemůže vytvářet), ale jsou definovány v ECMAScript standardu a jsou pomocí nich popisovány sémantické akce pro jednotlivé nonterminály. Jsou tedy používány pouze jako mezivýsledky během interpretace, ale je nutné je také implementovat. Jedná se o typ Odkaz, Seznam a typ Dokončení.

2.2.1 Uživatelské datové typy

Nedefinovaný typ

Nedefinovaný typ má přesně jednu možnou hodnotu, a to nedefinovanou. Každé proměnné, které není přidělena žádná hodnota, je přidělena nedefinovaná hodnota. Takto jsou inicializovány například hodnoty vzniklé pomocí příkazu **var**.

Typ Null

Null typ má právě jednu hodnotu zvanou null.

Typ Boolean

Boolean typ představuje logickou hodnotu. Může nabývat pouze hodnot true a false.

Typ Řetězec

Typ Řetězec je množina všech konečných seřazených posloupností elementů. Každý element je reprezentován šestnácti-bitovým neznaménkovým typem integer. Typ Řetězec je používán pro reprezentaci textu při běhu Javascriptového programu. Řetězce se uzavírají buď mezi dvojici uvozovek, nebo dvojici apostrofů.

Typ Číslo

Výčet hodnot typu číslo je dán hodnotami 64bitového formátu standardu IEEE 754. Prakticky ekvivalentem je například typ double jazyka C++, který bude také pro reprezentaci této hodnoty během implementace používán.

Typ Objekt

Objekt je představován množinou vlastností. Každá vlastnost má jméno, hodnotu a množinu atributů.

Atributy vlastností

Vlastnost může mít nula nebo více atributů z následující množiny

Atribut	Popis
ReadOnly	Pokusy o přepsání vlastnosti při vykonávání Javascript kódu budou ignorovány. Vlastnost s tímto atributem se může měnit díky akcím, které provede prostředí, kde se kód vykonává
DontEnum	Vlastnost nemůže být použita pro konstrukci typu for-in
DontDelete	Pokusy o smazání této vlastnosti budou ignorovány
Internal	Interní vlastnosti nemají jméno a nejsou přímo přístupné skrz operátor zpřístupnění. Tyto vlastnosti a metody jsou zde pouze z čistě vnitřních interpretačních důvodů.

Tabulka 2.3 - Popis atributů vlastností

Atribut Internal je v ECMAScript standardu sice definován, ale z hlediska implementace by nebylo příliš vhodné odlišovat vlastnosti pouze skrz atribut. Jednotlivé interní vlastnosti jsou tedy přímo vlastnostmi třídy (myšleno třída v jazyce C++), která daný JavaScriptový objekt reprezentuje a v tabulce Tabulka 2.3 jsem tento atribut uvedl jen pro úplnost.

Interní metody a vlastnosti

Interní metody a vlastnosti nejsou přímou součástí jazyka (tedy k nim nemůžeme přistupovat přímo). Jsou definovány čistě z vnitřních interpretačních důvodů. Tabulka 2.4 uvedena níže popisuje atributy a chování jednotlivých metod a vlastností objektů jádra JavaScriptového interpretu. Z pohledu tvůrce JavaScriptových programů je sice tato tabulka naprosto nezajímavá, ale pro tvorbu JavaScriptového interpretu je velice podstatná, jelikož popis chování je ve specifikaci jazyka uváděn právě za použití těchto vlastností.

Vlastnost	Atributy	Popis
Prototype	-	Prototyp daného objektu
Class	-	Řetězec udávající druh objektu
Value	-	Interní stav asociovaný s objektem
Get	(Jméno vlastnosti)	Vrátí hodnotu vlastnosti
Put	(Jméno vlastnosti, Hodnota)	Nastaví specifikovanou vlastnost na určitou hodnotu
CanPut	(Jméno vlastnosti)	Vrátí booleovskou hodnotu určující, jestli operace Put provedená se Jménem vlastnosti může být úspěšná. Tedy jestli objekt má daný typ vlastnosti a daná vlastnost nemá atribut ReadOnly.
HasProperty	(Jméno vlastnosti)	Vrátí booleovskou hodnotu indikující, má-li objekt vlastnost daného jména.
Delete	(Jméno vlastnosti)	Odstraní specifikovanou vlastnost z objektu.
DefaultValue		Vráti počáteční hodnotu objektu (měla by to být primitivní hodnota).
Construct	Seznam argumentů předaných volajícím objektem	Vytvoří objekt. Přístupné skrz operator new. Objekty implementující tuto metodu se nazývají konstruktory.
Call	Seznam argumentů předaných volajícím objektem	Provede kód asociovaný s tímto objektem. Tato metoda je vyvolána při interpretaci volání funkce. Objekty implementující tuto metodu se nazývají funkce.
HasInstance	(Hodnot)	Pouze Function objekty implementují tuto vlastnost.
Scope	-	Řetězec oborů platnosti definuje v jakém prostředí se Function objekt vykonává.
Match	(řetězec, index)	Test na regulární výraz a navrácení hodnoty MatchResult.

Tabulka 2.4 – Popis interních vlastností objektu

2.2.2 Interní datové typy

Typ Odkaz

Jedná se v podstatě o odkaz na vlastnost objektu. Typ Odkaz se skládá ze dvou komponent: bázového objektu a jména vlastnosti. Tento typ je určen k popisu chování operátorů, jako jsou delete, typeof, this (při volání funkcí) a operátoru přiřazení.

Typ seznam

Tento typ je používán při vyhodnocování seznamu argumentů při použití operátoru new a při volání funkcí. Jedná se v podstatě o seřazenou sekvenci hodnot.

Typ Dokončení

Je použit pro popis chování příkazů typu break, continue, return a throw, které vykonávají nelokální přesuny toku kódu interpretace. Hodnoty typu Dokončení jsou trojice hodnot (**typ**, **hodnota**, **cíl**). **Typ** může nabývat jedné z hodnot normal, break, continue, return nebo throw. **Hodnota** je jakákoliv Javascript hodnota, nebo hodnota prázdná a **cíl** je Javascript identifikátor nebo prázdna hodnota (tedy v podstatě řetězec).

2.3 Proměnné

JavaScript patří mezi tzv. dynamicky typované jazyky a tedy její proměnné mohou postupně obsahovat data jakýchkoliv typů. Je tedy v pořádku do proměnné postupně přiřadit nejprve hodnotu typu boolean a hned v dalším příkaze například hodnotu typu číslo nebo řetězec.
Proměnná lze deklarovat explicitně pomocí klíčového slova **var**. Není-li takto deklarovaná proměnná rovnou inicializována je její hodnota nastavena na hodnotu **undefined**. Pokusíme-li se do proměnné, která ještě nebyla deklarována přiřadit nějakou hodnotu, interpret tuto proměnnou implicitně deklaruje a přiřadí ji rovnou danou hodnotu. V JavaScriptu je povoleno provádět redeklarace proměnných. V případě, že je nedeklarovaná proměnná použita pro čtení (tedy ve výrazu, jako parametr funkce atd.) je interpretem nahlášena chyba.

I když to není na první pohled zřejmé, pojem proměnná je mírně zavádějící, jelikož všechny proměnné a funkce jsou přidávány jako vlastnosti k některému z objektů (a vlastně tedy nejsou proměnnými, ale vlastnostmi objektů). Tedy například, i pokud deklarujeme nějakou proměnnou v globální části kódu, nevědomky vlastně přidáváme vlastnost globálnímu objektu. Přesnější popis způsobu přidělování proměnných k objektům provedu v následující kapitole.

2.4 Kontext vykonávaní

Ve chvíli, kdy se začne s vykonáváním zdrojového kódu, řízení programu vstupuje do tzv. kontextu vykonávání. Kromě globálního kódu je i při každém volání funkce nebo konstruktor, vytvářen nový kontext vykonávání. Tyto kontexty vykonávání tvoří logicky zásobník. Nové kontexty vykonávání jsou vkládány vždy na vrchol zásobníku. Po dokončení vykonávání aktuálního kontextu, se pokračuje vykonáváním kontextu, který byl v zásobníku pod aktuálním.

Při vytváření kontextu vykonávání je postupně vytvořen řetězec oborů platnosti, přidělena hodnota klíčového slova this a provedení procesu konkretizace proměnných (inicializace těchto hodnot probíhá různým způsobem v závislosti na typu spouštěného kontextu). Než přistoupím k samotnému popisu procesu inicializace spouštěných kontextů pro jednotlivé typy musím provést několik definic.

2.4.1 Globální objekt

Globální objekt je unikátní objekt, který je vytvořen JavaScript interpretem ještě před vstupem do jakéhokoliv spustitelného kontextu. Po inicializaci má globální objekt následující vlastnosti:

- Vestavěné objekty – jako jsou Math, String, Date, parseInt atd. Všechny tyto vlastnosti (objekty) mají atribut DontEnum.
- Dále obsahuje vlastnosti definované klientským prostředím. To může zahrnovat i vlastnosti odkazující na samotný globální objekt. Například u prohlížeče Internet Explorer je vlastnost globálního objektu **window**, odkaz na globální objekt (tedy na sebe sama).

Během provádění samotného kódu může být obsah vlastností tohoto globálního objektu libovolně měněn.

2.4.2 Aktivační objekt

Tento objekt je vytvořen při vytvoření nového kontextu vykonávání, který je asociován se spustitelným kódem funkce. Aktivační objekt je inicializován s vlastností **argument** a atributem DontDelete. Inicializační hodnota této vlastnosti je objekt, který bude popsán níže.

Aktivační objekt je objektem používaným pouze pro provádění interpretace. Tedy přímo v JavaScriptovém programu můžeme přistupovat pouze k jeho vlastnostem, ale přímo k němu ne.

2.4.3 Objekt Arguments

Tento objekt je opět vytvořen při vytvoření nového kontextu vykonávání, který je asociován se spustitelným kódem funkce. Vytvořen a inicializován je následovně:

- Hodnota interní vlastnosti Prototype, bude předán odkaz na hodnotu základního prototypového objektu (který bude popsán dále v textu).

- Hodnota nově vytvořené vlastnosti callee, je odkazem na právě prováděnou funkci. Díky tomuto je možné například volat nepojmenované funkce. Tato vlastnost má atribut DontEnum.
- Hodnota nově vytvořené vlastnosti length je počet parametrů, s kterými je funkce volána. Vlastnost length má opět jeden atribut – DontEnum.
- Dále je vytvořena jedna vlastnost pro každé nezáporné číslo menší než length. Každé této vlastnosti je předaná hodnota příslušného parametru funkce. Díky tomuto lze funkci volat s neomezeným počtem (i nepojmenovaných) parametrů.

2.4.4 Typy kódu kontextu vykonávání

V JavaScriptu existují tři typy kódu spouštěného kontextu:

- Globální kód – tímto je myšlen zdrojový text programu, tedy části analyzované jako Program. Globální kód pouze neobsahuje části analyzované jako FunctionBody.
- Kód funkce Eval – zdrojový text předávaný vestavěné funkci eval. Tedy přesněji řečeno, pokud je funkci eval předán jako parametr řetězec, je tento řetězec považován za Javascript Program.
- Kód funkcí – je zdrojový text analyzovaný jako FunctionBody. Případně je za to samé považován poslední argument vestavěné funkce Function, se kterým je nakládáno jako by se jednalo o FunctionBody. FunctionBody neobsahuje žádný zdrojový text, který by obsahoval část analyzovanou jako vnořené FunctionBody .

2.4.5 Konkretizace proměnných

Každý spouštěný kontext má k sobě asociovaný objekt proměnných. Kdykoliv se v daném spouštěném kontextu deklaruje proměnná nebo funkce, je přidána právě k tomuto objektu proměnných jako jeho vlastnost. Některé vlastnosti nového objektu proměnných mohou být opět závislé na typu kódu spouštěného kontextu. Postup vytváření objektu proměnných je prováděn v tomto pořadí.

- *Pro kód funkcí* – Pro každý formální parametr funkce přidáme do objektu vlastnost s názvem identifikátoru parametru. Hodnoty parametrů jsou předané pomocí speciálního objektu **arguments**, který zajistí volající objekt. Pokud volající objekt předá menší počet parametrů, než je formální počet parametrů, je parametrům, na které „nezbylo“ nastavena hodnota na **undefined**. Pokud více formálních parametrů sdílí stejné jméno, jsou tyto parametry identické. Jejich hodnota je následně nastavena na hodnotu, která je přidělena poslednímu z identických parametrů.
- *Pro deklaraci funkcí v kódu* – Pro každou deklarovanou funkci v kódu je opět přidána vlastnost do objektu proměnných, která se jmenuje stejně jako identifikátor funkce. Hodnota

této vlastnosti je nastavena na návratovou hodnotu vzniklou při procesu vytváření funkčního objektu (bude popsáno dále v textu). Pokud již vlastnost s daným jménem existuje, jednoduše bude přepsána.

- *Pro každé deklarování proměnné* – Opět vytvoříme vlastnost objektu proměnných se jménem identifikátoru nové proměnné. Hodnota deklarované proměnné bude buď undefined, nebo inicializační hodnota. Pokud již objekt proměnných obsahuje vlastnost daného jména a jedná se o formální parametr nebo o funkci, tak bude deklarování proměnné ignorováno.

2.4.6 Řetězec oborů platnosti

Řetězec oborů platnosti je jedna z vlastností kontextu vykonávání. Jedná se o seznam objektů, které jsou prohledávány při vyhodnocování hodnoty identifikátoru. Identifikátor je postupně hledán ve všech objektech řetězce oborů platnosti. Pro každý typ kódu spouštěného kontextu je obor platnosti inicializován různě (bude uvedeno dále). Během provádění kontextu vykonávání je řetězec oborů platnosti možno ovlivňovat pouze pomocí příkazů **with** a **catch**.

2.4.7 Vytvoření kontextu vykonávání

V této kapitole uvedu postup při vytváření kontextu vykonávání, pro různé typy kódu kontextu vykonávání.

Globální kód

- Řetězec oborů platnosti je vyroben a inicializován tak, že obsahuje pouze globální objekt
- Proces konkretizace proměnných je proveden za použití globálního objektu jako objektu proměnných a s použitím atributu vlastností DontDelete.
- Hodnota klíčového slova **this** nastavena na globální objekt

Kód funkce eval

Ve chvíli kdy řízení programu vstoupí do kódu kontextu vykonávání funkce eval, je předchozí aktivní kontext vykonávání použit pro určení řetězce oborů platnosti, hodnoty klíčového slova this a objektu proměnných.

Kód funkcí

- Řetězec oborů platnosti je inicializován, tak aby obsahoval aktivační objekt, následovaný objekty, které jsou uloženy ve vlastnosti **Scope** funkčního objektu.
- Proces konkretizace proměnných je proveden za použití aktivačního objektu. Tento aktivační objekt zde zastupuje obekt proměnných. Vlastnosti jsou přidávány s hodnotou atributu DontDelete.
- Hodnota klíčového slova **this**, je předaná volajícím kontextem.

2.5 Objekty

Jak už bylo řečeno výše objekt je tvořen množinou vlastností, kde každá vlastnost obsahuje jméno, hodnotu a atribut. Hodnotami mohou být jak primitivní datové typy, tak odkazy na další objekty nebo metody. Vlastnosti objektu jsou zpřístupnitelné pomocí operátoru „**.**“.

2.5.1 Vytváření objektů

Objekty jsou vytvářeny pomocí tzv. konstruktorů. Konstruktory jsou funkce, které inicializují daný objekt a jsou volány pomocí operátoru **new.** Dále je konstruktoru předán odkaz na nově vytvořený prázdný objekt jako hodnota klíčového slova **this**.

JavaScript podporuje řadu vestavěných konstruktorů, které mohou vytvářet jak prázdné objekty (tedy objekty bez vlastností), tak objekty, které jsou inicializovány složitějším způsobem. Metody jsou funkce JavaScriptu, které jsou volány prostřednictvím objektu. Funkce je přiřazena k objektu jednoduchým přiřazením funkce k nějaké vlastnosti objektu. Jelikož to u běžných programovacích jazyků není zcela běžné, uvedu příklad:

```
function PlochaObdelniku () { return this.vyska*this.sirka }
function Obdelnik (vyska,sirka) {
      this.sirka = sirka;
      this.vyska = vyska;
      this.plocha = PlochaObdelniku
}
obdelnik = new Obdelnik (4,5);
vymer = obdelnik.plocha ();
```

2.5.2 Prototypování

JavaScript neobsahuje přímo vlastní třídy, ale místo toho podporuje výše zmíněné konstruktory. Ke každému konstruktoru je asociována vlastnost „prototype“, která je použita pro implementaci prototypově založené dědičnosti a sdílení vlastností. Každý objekt vytvořený pomocí určitého konstruktoru má implicitně odkaz na prototype vlastnost(tedy objekt uložený v této vlastnosti) tohoto konstruktoru. Tento odkaz má uložený v interní Prototype vlastnosti. K této vlastnosti nemá programátor u takto vyrobeného objektu přímý přístup. Dědění se projevuje automaticky během vyhledávání hodnot vlastností. Ve chvíli, kdy interpret musí vyhledávat nějakou hodnotu vlastnosti objektu, postupuje následovně. Nejdříve začne prohledávat prostor svých přímých vlastností. Nenajde-li mezi nimi požadovanou vlastnost, pokračuje v prohledávání dál v objektu určeném interní vlastností Prototype. Nenajde-li určenou vlastnost ani v tomto objektu, pokračuje takto dál, případně přes další objekty. Tento postup ovšem platí pouze pro čtení vlastností. Pokud se například

pokoušíme zapsat do vlastnosti **o.v** a objekt **o** vlastnost **v** nemá, interpret již neprochází celý prototype řetězec pro vyhledávání dané vlastnosti, ale rovnou vytvoří vlastnost **v** v objektu **o** přímo.

2.5.3 Konverze datových typů

Pokud se při vykonávání JavaScript kódu použije datový typ, který je v daném kontextu neočekávaný, JavaScript se pokusí provést automatickou konverzi na typ, který je danou operací očekáván. Tímto je myšlena například situace, kdy testujeme v podmínce `if` řetězcovou hodnotu. V následující tabulce 2.5 uvedu všechny případy automatických konverzí.

Konvertovaná hodnota	**Výsledek automatické konverze**			
	Logická hodnota	Řetězec	Číslo	Objekt
Undefined	false	„undefined“	NaN	Vyvolána výjimka
Null	false	„null“	+0	Vyvolána výjimka
true	true	„true“	1	objekt Boolean s interní hodnotou true
false	false	„false“	+0	objekt Boolean s interní hodnotou false
+0,-0	false	„0“ nebo „-0“	beze změny	objekt Number
NaN	false	„NaN“	beze změny	objekt Number
Nekonečno	true	„Infinity“	beze změny	objekt Number
Záporné nekonečno	true	„-Infinity“	beze změny	objekt Number
Ostatní čísla	true	Číslo převedené na řetězec	beze změny	objekt Number s interní hodnotou rovnou danému číslu
Neprázdný řetězec	true	beze změny	Číselná hodnota řetězce (pokud lze), nebo NaN	objekt String
Prázdný řetězec	false	beze změny	0	objekt String
objekt	true	Zavolá se metoda objektu toString	Postupně se zavolají metody valueOf , toString a toNumber. Nezdaří-li se je použita hodnota NaN	beze změny

Tabulka 2.5 – Automatická konverze hodnot

2.6 Bezpečnostní rizika jazyka JavaScript

Z pohledu aplikace, která je zamýšlena jako budoucí součást programu, který je schopen provádět různé antivirové testy, jsou hlavními bezpečnostními riziky takové součásti jazyka, které znemožňují efektivní činnost jednotlivým testů antivirového programu. Při studiu jazyka JavaScript jsem narazil na dvě vestavěné funkce, které lze potencionálně využít pro obcházení antivirových programů. Nebezpečí obou tkví ve faktu, že přijímají jako vstupní parametry řetězce, které jsou transformovány na nový vykonavatelný JavaScriptový kód. Bez interpretace nelze tedy nikdy dopředu odhadnout, co takto vytvořený kód bude vykonávat. Škodlivý kód vznikající na tomto principu je většinou navíc vytvářen z jednoho dobře známého viru a pomocí různých zatemňovacích obálek složených především z těchto dvou funkcí následně transformován na kód, který na první pohled vypadá relativně bezpečně.

Objekt Function

První nebezpečnou konstrukcí je vestavěný objekt Function. Tento vestavěný objekt lze volat jako funkci i konstruktor. V případě volání tohoto objektu jsou z prvních N-1 parametrů vytvořeny formální parametry nové funkce a N-tý parametr je tělem této nové funkce. Pro takto vzniklou funkci musí být samozřejmě provedena syntaktická analýza. Taková to funkce je následně vrácena jako návratová hodnota. Pro větší názornost uvádím krátkou ukázku použití takové funkce:

```
var F = Function(“a“, “b“,“return a+b;“);
// je ekvivalentem k
// var F = function(a,b){return a+b;};
 print(F(1,2));
// na obrazovku bude vytištěno „3“
```

Funkce eval()

Druhou problémovou funkcí je metoda eval globálního objektu. Volání této funkce spustí zcela nový kontext vykonávání, ve kterém je prováděn JavaScriptový program daný argumentem funkce eval. Tento argument je typu řetězec a opět pro něj musí být provedena, před spuštěním nového kontextu vykonávání, syntaktická analýza. Pro větší názornost uvedu opět krátkou ukázku :

```
a = 1;
eval(“b=2;function Secti(a,b){return a+b;};”);
print(Secti(a,b));
// na obrazovku bude vytištěno „3“
```

A jak tedy vypadají samotné zatemňovací obálky? Většinou jde o větší množství různě dlouhých řetězců, nad kterými jsou prováděny různé operace (sčítání, operace unescape apod.). Takto

vzniklý řetězec je předán funkci eval, kde se provádějí další operace s řetězci (většinou v relativně dlouhých cyklech). Výsledný řetězec, který již obsahuje samotný škodlivý kód je zapsán do html dokumentu. Pro činnost antivirového programu je tedy důležité, aby měl zajištěn přístup k řetězcům, které vznikají nebo jsou zapisovány při provádění funkce eval (případně Function).

Samotný způsob a principy škodlivé činnosti jednotlivých druhů JavaScriptových virů nejsou pro vypracování této diplomové práce zásadním tématem. Informací je případně dostatek na [4]

3 Interpretační překladač

Pro spouštění zdrojového kódu napsaného ve vyšším programovacím jazyce existují dva přístupy. Prvním z nich je převedení programu do ekvivalentního strojového kódu. Překladač kompilující tímto způsobem se nazývá kompilátor nebo také kompilační překladač. Jeho výhodou je to, že analýza zdrojového kódu (mnohdy dosti časově náročná) se provádí pouze jednou a poté se již pouze spouští ekvivalentní program ve strojovém kódu. Druhou variantou je převést kód zdrojového textu do nějaké vnitřní reprezentace a s její pomocí přímo provádět příslušné akce. Tento typ překladače se nazývá interpret neboli interpretační překladač.

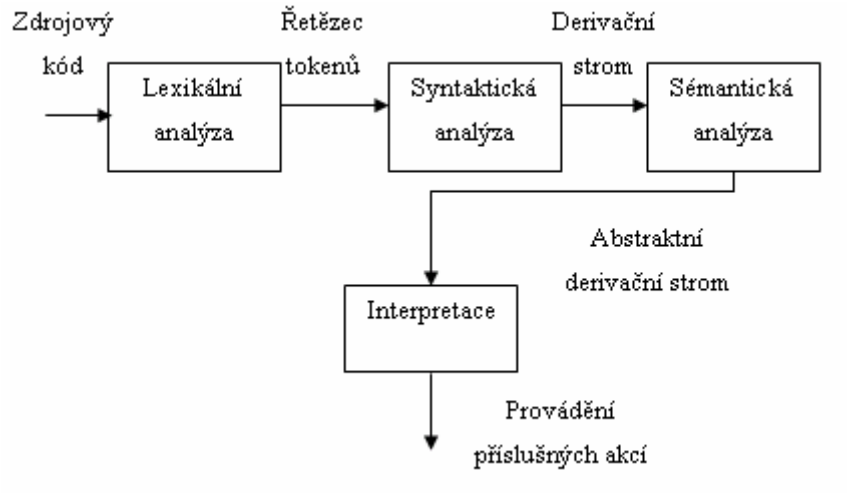
Zde se přímo nabízí otázka, proč si přímo nevystačíme pouze s jedním z uvedených přístupů. Pokud od spouštěného programu požadujeme maximální rychlost, bude zřejmě lepší využít služby kompilačních překladačů. Kompilační překladače provádějí analýzu a překlad zdrojového programu pouze jednou a nezávisle na samotném spouštění programu. Díky tomu lze provádět i důkladnější, časově náročnější kontroly. Naproti tomu interpret provádí analýzu zdrojového programu při každém spuštění aplikace a logicky se tedy rychlost takovéhoto programu musí snižovat. Použití interpretačního překladače také zvyšuje paměťové nároky spouštěného programu, jelikož je nutné udržovat v paměti při běhu programu také samotný interpret. Proč tedy vůbec používat interpretační překladač? Jendou z výhod interpretačních překladačů je jejich přenositelnost mezi různými platformami, máme-li totiž nainstalovaný překladač určitého zdrojového jazyka na různých platformách, každý z nich umí přijmout tentýž zdrojový program. V praxi se ovšem oba typy velice často vzájemně překrývají a vznikají tak překladače, které využívají výhod obou typů. Dále uvádím tabulku srovnávající jednotlivé vlastnosti interpretačního a kompilačního překladače. Symbolem "+" jsem označil překladač, jenž danou vlastnost zvládá obecně lépe.

Vlastnost	Kompilátor	Interpret
Paměťové nároky	+	
Přenositelnost zdrojového kódu		+
Rychlost výsledné aplikace	+	

Tabulka 3.1- srovnání vlastností interpretu a překladače

3.1 Struktura interpretačního překladače

Základní blokové schéma interpretačního překladače je uvedeno na obr.3.1. Jednotlivé bloky schématu hlavně ukazují, jaké jednotlivé úlohy musí interpretační překladač zvládnout během své činnosti. U jednotlivých implementací různých interpretačních překladačů se mohou některé bloky překrývat a jiné obsahovat složitější operace, než které vyjadřuje toto blokové schéma.



Obrázek 3.1-Blokové schéma interpretačního překladače

3.1.1 Lexikální analýza

Lexikální analyzátor zpracovává po znacích zdrojový soubor a snaží se rozeznávat jednotlivé logické celky zdrojového programu, tzv. lexémy. Informace o lexémech dále poskytuje syntaktickému analyzátoru pomocí "tokenů". Tokeny většinou nesou informaci o typu lexémy, kterou reprezentují a popřípadě její hodnotu (nemusí být vždy potřeba). Lexikální analyzátor se také stará o vypouštění komentářů ze zdrojového textu a informuje uživatele o výskytu chyb zjištěných již během lexikální analýzy. Většinou bývá implementován pomocí konečného automatu.

Definice konečného automatu

Konečný automat je pětice[3] :

M= (Q,Σ,R,s,F) ,kde

- Q je konečná množina stavů
- Σ je vstupní abeceda automatu
- R je konečná množina pravidel tvaru pa $\rightarrow$ q, kde p,q $\in$ Q, a $\in \Sigma \cup \{\varepsilon\}$. Symbol ε vyjadřuje prázdný řetězec. Striktně matematicky vzato je R relace z $Q \times (\Sigma \cup \{\varepsilon\})$ do Q
- s $\in$ Q je startovní stav
- F $\subseteq$ Q je množina konečných stavů

Nástroje pro tvorbu konečných automatů

Pro jeho tvorbu lze také použít některý z řady nástrojů pro automatickou tvorbu lexikálních analyzátorů jako je např. Lex. Já jsem se tento nástroj rozhodl použít, jelikož je nesporné, že tvorba lexikálního analyzátoru následně probíhá výrazně rychleji a je možné, v případě potřeby, jednodušeji pozměňovat vlastnosti lexikálního analyzátoru.

[3] převzato z [2])

3.1.2 Syntaktická analýza

Před samotným popisem syntaktické analýzy, bude vhodné uvést definici bezkontextové gramatiky.

Definice bezkontextové gramatiky

Bezkontextová gramatika je čtveřice[4] G = (N, T, P, S), kde

- N je konečná množina nonterminálních symbolů.
- T je konečná množina terminálních symbolů, přičemž $N \cap T = \emptyset$.
- P je konečná podmnožina kartézského součinu $N \times (N \cup T)^*$. Místo tohoto relačního zápisu zapisujeme pravidla P ve tvaru $A \rightarrow x$, kde $A \in N$ a $x \in (N \cup T)^*$.
- $S \in N$ je výchozí symbol gramatiky.

Syntaktická analýza je proces, při kterém se překladač snaží rozpoznat, zda zdrojový text tvoří věty odpovídající gramatice daného jazyka. Pokud překladač rozpozná během činnosti nějakou chybu, nahlásí ji a obvykle provede určité zotavení, aby mohl pokračovat a případně najít nějaké další chyby.

Existují dva základní přístupy k návrhu syntaktického analyzátoru - syntaktická analýza shora dolů nebo zdola nahoru. Tyto názvy odpovídají postupu při vytváření derivačního stromu. Při překladu metodou zdola nahoru se snažíme posloupnost terminálních symbolů postupně redukovat pomocí pravidel gramatiky až na startovní nonterminál. Naopak při překladu metodou shora dolů vycházíme ze startovního nonterminálu gramatiky a ten se snažíme postupným derivováním expandovat na řetězec terminálních symbolů, které odpovídají terminálním symbolům ze zdrojového textu. Uvedeným dvěma metodám také odpovídají dvě základní třídy gramatik - LL gramatika pro metodu shora dolů a LR gramatika pro metodu zdola nahoru.

Kromě kontroly, zda je zdrojový text syntakticky správný, je důležité, aby byly zaznamenávány informace potřebné v další fázi překladu. Syntaktický analyzátor by měl tedy zaznamenávat pravidla, která používal při vytváření syntaktického stromu a argumenty příslušející jednotlivým pravidlům. Například je nutné znát hodnoty literálů a jména identifikátorů. Tyto informace jsou použity během sémantické analýzy pro vyhodnocování vlastností, které nelze postihnout bezkontextovými gramatikami.

Vzhledem k faktu, že jsem při návrhu počítal s využitím automatického nástroje pro tvorbu syntaktického analyzátoru Bison, nebudu uvádět bližší detaily spojené s návrhem a teorií. V případě zájmu lze příslušnou teorii a postupy dohledat v [1].

3.1.3 Sémantická analýza

Sémantická analýza provádí kontroly, které nelze postihnout během syntaktické analýzy. Tedy například kontrola jsou-li volané funkce (konstruktory) již deklarovány. Tedy kontrola, není-li volána

[4] Převzato z [2]

funkce, která nebyla do té doby deklarována. Dále jsou prováděny typové kontroly během vyhodnocování výrazů, kontrola správného typu operandu u operátorů, kontrola zda jsou používané proměnné deklarovány (je-li to daným jazykem požadováno) apod. U běžných jazyků do této kategorie patří i kontrola redeklarace proměnných.

Jelikož je JavaScript jazykem dynamicky typovaným nelze sémantickou analýzu provádět jako jakýsi separovaný krok. To, co u klasických překladačů nazýváme sémantickou analýzou, probíhá v interpretačním překladači jazyka JavaScript průběžně při interpretaci, vždy když je potřeba.

3.2 Uvolňování paměti

Uvolňování paměti je část JavaScriptu, která není striktně daná ECMAScript normou, ale obecně se počítá s tím, že si interpret po sobě „uklidí“ nepoužívané místo. A proto musím podobný mechanismus implementovat i ve své diplomové práci.

Při uvolňování paměti je zásadní úlohou pro mechanismus uvolňování paměti, určení kdy je možné dealokovat určité místo v paměti. Je zřejmé, že muže být ovlivňováno pouze místo v paměti, které již není jakýmkoliv způsobem přístupné pomocí proměnných, objektů atd. V dalších kapitolách rozeberu základní metody uvolňování paměti.

3.2.1 Uvolňování paměti počítáním odkazů

Tato metoda je založena na počítání jednotlivých odkazů na daný objekt v paměti. Tedy každý objekt si počítá počet objektů (proměnných), které na něj odkazují.

Ve chvíli, kdy je objekt interpretem vytvořen a odkaz na něj uložen do některé proměnné, nastaví si objekt počet odkazů na jedna. Při každém zkopírování odkazu na daný objekt se počítadlo odkazů zvýší o jedna. Naopak při uložení jiné hodnoty do proměnné, která obsahovala odkaz na daný objekt, je počítadlo odkazů tohoto objektu sníženo o jedna. To samé se děje samozřejmě také při zániku proměnné (např. u lokálních proměnných). Dosáhne-li počet odkazů na objekt čísla nula, můžeme bezpečně a s jistotou uvolnit paměť, kterou si daný objekt pro svou činnost alokoval. Každá metoda má své výhody a nevýhody. U této metody jsou výhody zřejmé na první pohled:

- Snadná implementace
- Odstranění objektu z paměti ihned, jak je to možné

Bohužel, i když je tuto metodu snadné implementovat a ve spoustě případů funguje výborně, existují případy, díky nimž se tato metoda dá řadit spíše mezi nepoužitelné. Hlavním problémem metody počítání odkazů, jsou totiž cyklicky se odkazující objekty. Představme si hypotetickou situaci, kdy máme objekt A, odkazující na objekt B, který odkazuje na objekt C, který opět odkazuje na objekt A. Nastane-li nyní situace, že na tento cyklus neexistuje z vnějšku žádná reference, jsou v podstatě tyto tři objekty nedostupné, ale mechanismus uvolňování paměti to není schopen zaregistrovat, jelikož proměnné na sebe stále odkazují.

3.2.2 Uvolňování paměti označením a odstraněním

Metoda uvolňování paměti pomocí označování a odstraňování je prováděna, jak již název napovídá, ve dvou krocích. Nejprve musí mechanismus uvolňování paměti postupně procházet stromem proměnných a označovat všechny objekty, které jsou těmito proměnnými odkazovány. Pokud jsou odkazovanými hodnotami pole nebo objekty, pokračuje se samozřejmě v označování i ve vlastnostech

objektů a prvcích polí. Takto jsou tedy označeny všechny objekty, které jsou dostupné. Ve druhé fázi interpret projde všechny objekty a smaže všechny, které nejsou označené v předchozím kroku.

Tato metoda je již pro použití výhodnější, jelikož se dokáže vypořádat i s cyklicky se odkazujícími objekty. Na druhou stranu se zde objevuje jiná nevýhoda. Tento mechanismus není schopen dealokovat paměť objektu ve chvíli, kdy ztratí poslední referenci. Dealokace probíhá hromadně vždy v cyklech, tedy zastaví se činnost samotného programu a začne proces označování a odstraňování. Spouštěcí podmínkou pro tuto činnost může být například dosažení určitého velikosti alokované paměti programem.

4 Návrh

Jelikož jsem se s návrhem a implementací takto rozsáhlého interpretu nikdy předtím nesetkal, probíhal návrh a implementace inkrementálně. Jak již bylo v předchozích kapitolách řečeno, lexikální i syntaktický analyzátor bude implementován pomocí automatických nástrojů Flex a Bison. Důvody jsou zřejmé, oba nástroje šetří čas při implementaci a je snadné v případě potřeby upravovat vlastnosti jak lexikální, tak syntaktické analýzy.

Samotný standard ECMAScript popisuje jazyk pomocí definic sémantiky pro jednotlivá gramatická pravidla. Budu se tedy také snažit držet přesného postupu, který je uveden ve standardu, abych se vyvaroval zbytečných chyb. Při vyhodnocování jednotlivých uzlů je většinou (pokud se nejedná o terminální uzly) samozřejmě zapotřebí vyhodnotit podřízené uzly (zastupující nonterminály v pravidlech gramatiky). Takto se pokračuje až k nejnižším patrům syntaktického stromu. Původně jsem při návrhu zamýšlel provádět interpretaci rekurzivně přímo z této stromové struktury. Během návrhu došlo k mírné změně zadání a bylo nutné, aby bylo možné interpretaci programu provádět po určitých elementárních krocích. Samozřejmě i za této podmínky by bylo možné provádět interpretaci přímo ze stromové struktury, ale vznikalo by zde nepříjemně mnoho problémů. Nakonec jsem se tedy rozhodl použít již připravenou stromovou strukturu pouze pro generování mnou definovaných instrukcí, které budou následně sekvenčně interpretovány v rámci jednotlivých kontextů vykonávání.

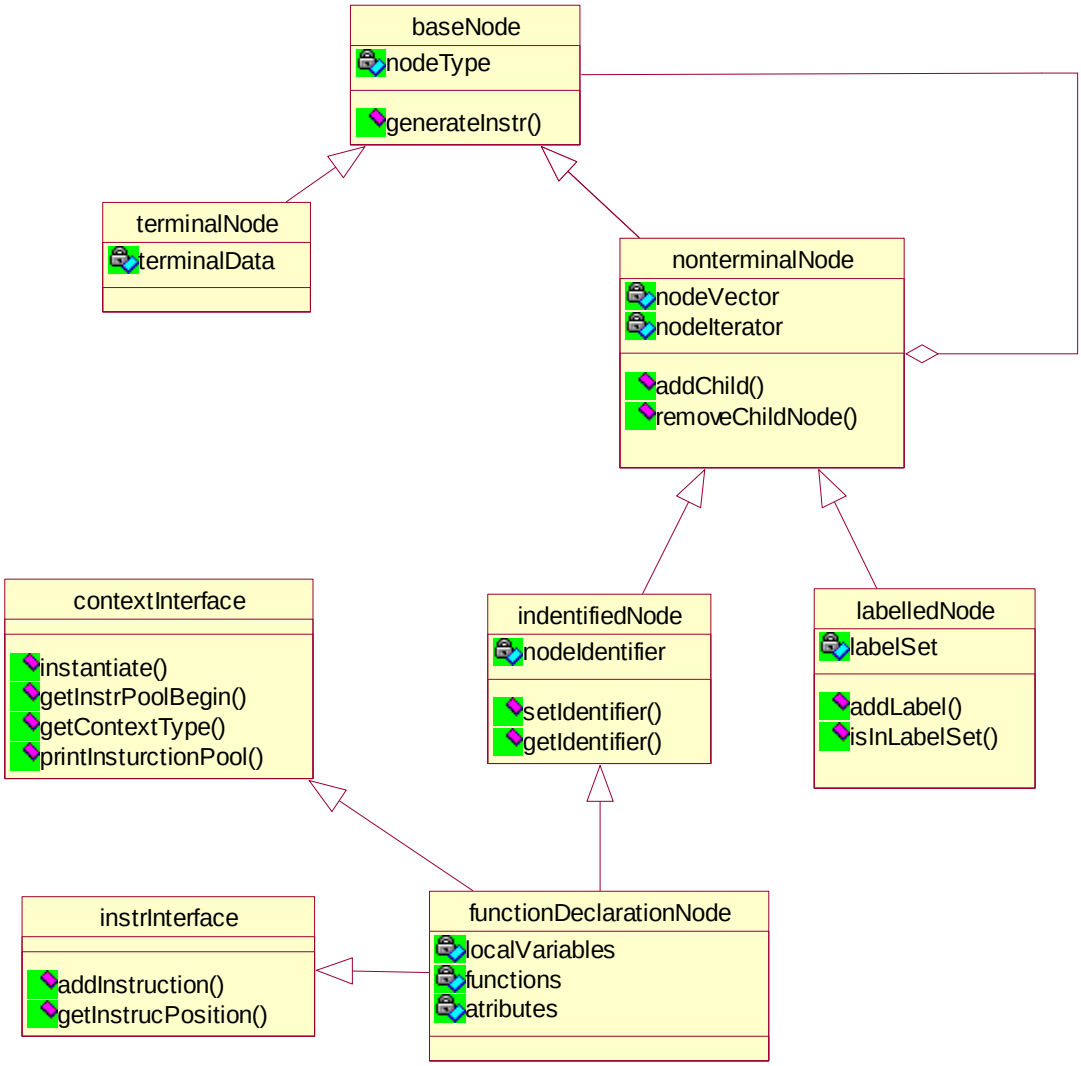
ECMAScript specifikace dále při popisu jazyka využívá některé datové struktury a objekty, které bude nutné implementovat. Jedná se hlavně o struktury kontextu vykonávání (popsána v kapitole 2.4.) , obecného JavaScript objektu (popsán v kapitole 2.2.6.) a strukturu jež je schopna vyjádřit množinu všech možných datových hodnot jazyka JavaScript(popsáno v kapitole 2.2). Dále je potřeba implementovat mechanismy, díky nimž bude možno vytvářet jednotlivé interní datové objekty (zde mám na mysli především Aktivační objekt, objekt Arguments).

Samotný syntaktický strom je vytvořen během syntaktické analýzy, prováděné pomocí programu Bison. Na jednotlivé uzly syntaktického stromu, jsou samozřejmě také případně navázány hodnoty, které k nim logicky patří. Tedy například k literálu čísla hodnota čísla atp. Syntaktický strom je složen z objektů odvozených od třídy baseNode. Dále v textu bude tato a další důležité třídy popsány.

4.1 Návrh struktury syntaktického stromu

Pro návrh této třídy jsem se nechal mírně inspirovat návrhovým vzorem Composite. Jedná se o návrhový vzor, který řeší situaci, kdy máme strukturu složenou z jednoduchých a složených objektů a ke všem z nich potřebujeme mít zajištěný jednotný přístup.

V mém konkrétním případě jde o stromovou reprezentaci terminálních a neterminálních pravidel gramatiky. Každý z nich potřebuje ke svému životu různé datové struktury a operace. Všechny tyto uzly (terminální i neterminální) musí mít implementovánu operaci generateInstr, skrz kterou se provádí generování instrukcí pro jednotlivé prvky gramatiky, který jsou uzly reprezentovány. Na obrázku 4.1 je uvedeno schéma třídní hierarchie, které budu dále popisovat.



Obrázek 4.1– Třídní hierarchie syntaktického stromu

Třída baseNode

Základní třídou hierarchie je abstraktní třída baseNode, která obsahuje již výše zmíněnou „virtuální“ operaci generateInstr. Tuto operaci musejí implementovat všichni následníci této třídy. Voláním této metody se provede vygenerování instrukcí pro příslušný terminální nebo nonterminální uzel. Návratová hodnota této metody je iterátor na první instrukci, kterou byla uzlem vygenerována. Tato návratová hodnota je potřeba pro případ skoku z jiné instrukce na začátek tohoto bloku instrukcí.

Jelikož každý z následníků třídy baseNode bude vytvářen pro určitou množinu pravidel gramatiky, které mají podobné metody vyhodnocování, je nutné specifikovat typ uzlu, který je v daném objektu zastoupen. Toto je prováděno pomocí atributu nodeType, který je nastaven při vytváření objektu konstruktorem.

Třída terminalNode

Třída terminalNode je třídou představující terminální uzly gramatiky a je odvozená (dědí) z třídy baseNode. Třída obsahuje atribut terminalData, ve kterém má uzel uloženy další data (hodnoty čísel, řetězců, názvy identifikátoru). Datový typ tohoto atributu je opět odkaz na objekt třídy nodeData. Tato třída slouží pro vyhodnocování terminálu literálů (číslo, řetězec, boolean,null), klíčového slova this a identifikátorů. Jelikož se již jedná o konkrétní třídu, musí samozřejmě obsahovat implementaci virtuální metody generateInstr.

Třída nonterminalNode

Třídou, která přestavuje základní třídu pro neterminální uzly je třída nonterminalNode. Jak je zřejmé z obrázku 4.1, je tato třída odvozená od třídy baseNode. Jelikož se jedná o třídu, obsahující odkazy na další uzly (konkrétně odkazy na objekty třídy baseNode), musí obsahovat strukturu, do které se budou tyto odkazy ukládat (std. kontejner jazyka C++ vector) a metody, pomocí kterých se budou uzly s objekty této třídy asociovat. Dále uvedu základní operace, které je možno provádět nad třídou:

- addChildNode – Metoda vloží nový uzel na konec seznamu potomků. Parametrem této funkce je odkaz na objekt typu baseNode.
- removeChildNode – Metoda odstraní první uzel ze seznamu potomků.

Třídy IdentifiedNode a labelledNode

Dále v třídní hierarchii následují třídy uzlů, které ke svému vyhodnocení potřebují některé specifické informace. Jedná se o třídy IdentifiedNode a labelledNode. Obě třídy jsou odvozeny od třídy NonterminalNode.

labelledNode slouží pro vyhodnocování nonterminálů, které mimo jiné slouží jako cíle pro skoky typu **break** a **continue**. Pro svoji činnost potřebuje poskytovat možnost k sobě asociovat návěští. Toto je prováděno pomocí metody addLabel. Metoda isInLabelSet naopak vrací booleovskou hodnotu, která udává, je-li návěští uvedené v argumentu metody v daném uzlu zastoupeno. Množina návěští je pro jednotlivé typy uzlů inicializována jako prázdná, s výjimkou uzlů představující nonterminály příkazů switch, a příkazů iterace (for,while apod). Do množiny návěští těchto uzlů je pří inicializaci vložen prázdný řetězec (Návěští jsou popisována pomocí řetězců).

IdentifiedNode představuje uzel, ke kterému je možno přiřadit jméno. Nastavit, případně získat pojmenování uzlů lze skrz metody setIdentifier a getIdentifier.

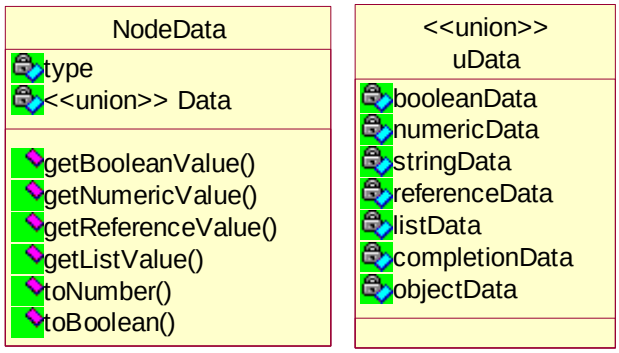
Třída functionNode

Trochu více pozornosti si zaslouží functionNode uzel. Jak je možné vidět na obrázku Obrázek 4.1, uzel functionNode dědí po uzlu identifiedNode. Tedy jednotlivé funkce a proměnné deklarované na globální úrovni, počet a názvy jednotlivých atributů funkce a samozřejmě i samotné tělo funkce (reprezentované nějakým syntaktickým stromem). Při prvním volání funkce je ze syntaktického stromu vygenerován seznam instrukcí. Dále tato třída uzlu zodpovědná za úkon konkretizace proměnných, tak jak je uvedeno v kapitole 2.4.5. Jelikož tato třída obsahuje téměř všechny informace potřebné ke spuštění funkce, mohlo by se zdát, že by bylo ideální, aby skrz dědění z bázové třídy představující JavaScriptový objekt, zároveň představovala objekt funkce. Bohužel jelikož jedna funkce (myšleno jedna část kódu programu) může, v závislosti na přiděleném řetězci oborů platnosti, vykonávat dvě rozdílné činnosti, nemůže functionNode zároveň zastávat funkci objektu funkce. Podrobněji bude problematika funkcí a přidělování řetězců oboru platnosti budu věnováno v kapitole 5.2.

4.2 Reprezentace JavaScriptové datové hodnoty

Jelikož je JavaScript jazykem dynamicky typovaným, bylo vhodné, abych vytvořil třídu, která dokáže představovat jakoukoliv hodnotu jazyka JavaScript. Také by mněla poskytovat metody potřebné pro automatickou konverzi datových typů a vyhodnocování jednotlivých operandů. Takovou třídou je třída nodeData. V podstatě se jedná o třídu obalující „union“ strukturu, která má jako položky uvedeny jednotlivé datové typy, které jsou uvedeny jako datové typy pro jazyk JavaScript. Tyto datové typy jsou uvedeny v kapitole 2.2.

Dále v textu budu „get“ metodami označovat množinu metod, které vracejí datovou hodnotu struktury. Třída obsahuje „get“ metodu pro každý jeden datový typ (viz. obrázek 4.3). Pokud je volána „get“ metoda pro typ, který není momentálně ve struktuře obsažen, je vyvolána interní výjimka. Pro případ, že chceme pouze zjistit, je-li instance struktury nodeData určitého typu, obsahuje třída množinu metod, kterými se můžeme dotazovat na konkrétní uživatelské datové typy (metody isBoolean, isNumber apod). „Set“ metody (tedy metody nastavující datovou hodnotu objektu) nejsou obsaženy. Datová hodnota objektu se v případě potřeby mění pomocí metod, které představují jednotlivé operace (+,- .. atd.), nebo pomocí metod, které zajišťují konverzi datového typu. Pro každý uživatelský datový typ je zde připravena jedna konverzní metoda (toBoolean,toObject apod.). Opět zde platí, že je-li prováděna typově nepovolená operace, je vyvolána výjimka identifikující tuto operaci. Počáteční typ dat, který je v objektu typu nodeData obsažen, je dán parametrem konstruktoru. Na obrázku 4.2 uvádím schéma třídy nodeData a struktury union, kterou využívá. Výčet metod z množiny „get“ a výčet všech operací typových konverzí není úplný z důvodu zbytečného zvětšování obrázku.



Obrázek 4.2 - Třída nodeData a struktura uData

4.3 Návrh třídy realizující kontext vykonávání

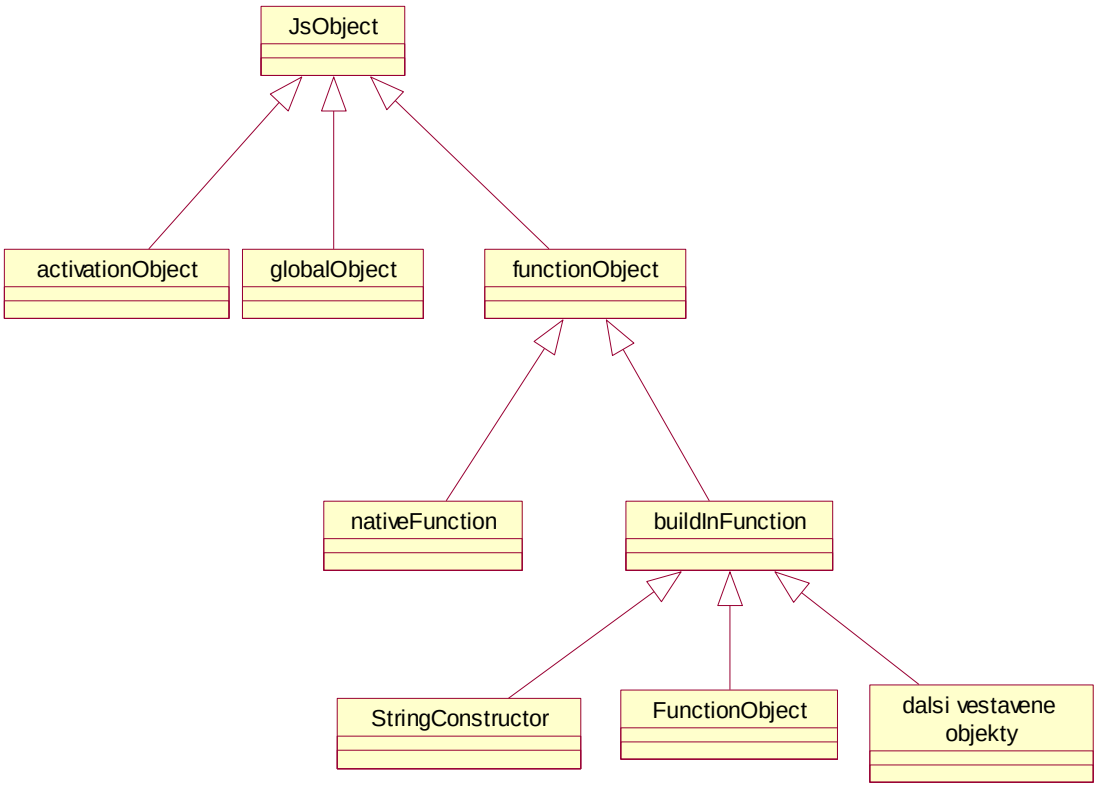
Tato třída má za úkol realizovat kontext vykonávání, tak jak je popsán v kapitole 2.4. V podstatě se jedná o třídu, která uchovává základní informace o vykonávaném kontextu.

Ve formě atributů třída uchovává řetězec oborů platnosti, hodnotu klíčového slova **this**, typ spouštěného kontextu, velikost zásobníku před vykonáním příkazu (uchováváno pro případ, kdy by byla uvnitř JavaScriptového příkazu vyvolána výjimka) a instrukci, která bude prováděna jako další v pořadí. Dále popíši jednotlivé metody třídy context:

- *getThis* – Metoda, která vrací hodnotu klíčového slova this. Návratová hodnota je tedy typu jsObject objekt.
- *findIdentifier* – Parametrem této metody je hodnota typu String. Jedná se o metodu, která prochází objekty zásobníku aktuálního řetězce oborů platnosti a vyhledává vlastnost objektu, která má stejné jméno jako parametr předaný funkci. Po prohledání všech vlastností daného objektu, je prohledán i objekt, který je uložen v interní prototype vlastnosti daného objektu (atd. pro celý prototype řetěz). Pokud metoda identifikátor daného jména nalezne, vrátí jeho hodnotu jako výsledek, jinak hodnotu **null**.
- *deleteIdentifier* – Funkce provede prohledání řetězce oborů platnosti zásobníku obdobně jako ve funkci findIdentifier. V případě, že vlastnost daného jména nalezne, odebere tuto vlastnost z objektu.
- *setNextStmtInstruction* – Jako aktuální instrukce je nastavena instrukce, kterou začíná další příkaz zdrojového kódu. Používáno pro přeskakování příkazů při volání výjimky (bude upřesněno dále v textu).
- *getInstrucParam* – Vráti hodnotu asociovanou s instrukcí.
- *nextInstruction* – Další instrukce v pořadí se stane instrukcí aktuální (posun iterátoru).
- *setActualInstruction* – Nastaví hodnotu předanou parametrem jako aktuální.
- *addToScope* – vloží položku na začátek řetězce oborů platnosti.
- *removeFromScope* – Odstraní první položku řetězce oborů platnosti.

4.4 Reprezentace JavaScript objektů

JavaScriptový objekt je v interpretu reprezentován množinou tříd, jejichž společná bázová třída je třída jsObject. Na obrázku 4.4 je ukázáno strukturální uspořádání a závislosti těchto tříd. Vzhledem k množství jednotlivých atributů a metod jsem se rozhodl tuto třídní hierarchii zobrazit bez nich. V podkapitolách podrobněji rozepíšu účel pouze nějakým způsobem významnějších tříd (tedy nebudu podrobně popisovat třídy reprezentující jednotlivé vestavěné objekty).



Obrázek 4.3 – Třídní hierarchie JavaScriptových objektů

4.4.1 Třída jsObject

Tato třída je bázovou třídou pro celou hierarchii tříd představující různé JavaScriptové objekty. Poskytuje metody, které jsou nutné pro pohodlnou a účelnou manipulaci s tímto typem objektů. Velká část těchto metod je virtuálních[5].

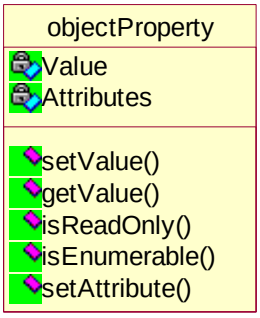
Základní schopností tohoto objektu je ukládat jednotlivé vlastnosti. Toto bude realizováno pomocí asociativního pole. Konkrétně se bude jednat o std. kontejner jazyka C++ map. Do tohoto kontejneru budou vkládány dvojice řetězec, datová hodnota. Datová hodnota bude typu

[5] Viz [11]

objectProperty (tato třída je znázorněna na obr.4.5). Dále uvedu přehled základních metod, pomocí kterých je manipulováno s touto třídou:

- *addProperty* – Funkce slouží k přidání nové vlastnosti do objektu. Jsou jí předány dva parametry. Jeden určuje jméno proměnné, druhý její hodnotu (datový typ objectProperty).
- *deleteProperty* – Slouží k vymazání vlastnosti daného jména. Návratovou hodnotou je hodnota typu boolean určující zda se operace smazání vlastnosti vydařila, či nikoliv
- *hasProperty* – Vrací true, pokud objekt obsahuje vlastnost daného jména.
- *setProperty*,*getProperty* – Vrátí, případně nastaví, hodnotu dané vlastnosti. Narozdíl od metod addProperty je zde předávána/vracena hodnota typu nodeData (hodnota není vytvářena pouze měněna).
- *isActivationObject*, *isVariableObject*, – Metoda vrací booleovskou hodnotu, určující patří-li objekt do dané kategorie. Pro bázovou třídu vracejí tyto metody implicitně false. Následníci této třídy mohou tuto metodu případně přepsat.
- *isConstructor*, *isFunction* – Metoda vrací, může-li být objekt volán jako funkce (konstruktor). Díky těmto metodám vím, mohu-li si dovolit přetypovat objekt typu jsObject na objekt typu functionObject. Opět je tato metoda definována v bázové třídě tak, aby vracela hodnotu false a případně je tuto metodu možné přepsat v některé třídě, která z této bázové třídy dědí.
- *mark* – Pomocí této metody je objekt označen, před spuštěním procesu uvolňování paměti, která probíhá pomocí metody „označ a uvolni“.
- *setParnamentlyMark* – Takto jsou označeny objekty, které jsou vytvořeny jako parametry instrukcí. Jelikož by bylo zdlouhavé procházet postupně všechny vygenerované instrukce a provádět označování pomocí metody mark, při každém spuštění procesu uvolňování paměti, je při vytváření objektu za účelem parametrizace instrukce volána tato metoda, která nastaví permanentní příznak označení objektu. Takto označený objekt již nemusí být následně označován metodou mark před uvolňováním paměti.
- isMarked – Metoda vrátí true, pokud má objekt nastaven příznak mark nebo pernamentlyMark.

Na obrázku 4.4 je znázorněna třída objectProperty. Typ atributu Value je nodeData. Dále zde třídu Property nebudu podrobněji nerozepisovat, jelikož důvod a účel jejich atributů a metod je snad dostatečně zřejmý.



Obrázek 4.4 – Základní metody a atributy tříd Property a JavaScriptObject

4.4.2 Třída functionObject

Třída functionObject je bázovou třídou pro všechny objekty, které mohou být zároveň volány jako funkce nebo konstruktory. Třídy odvozené od této musí implementovat metody callThis a constructThis. Každá z těchto metod má dva argumenty. Prvním argumentem je ukazatel na seznam reálných argumentů volané funkce (konstruktoru). Druhým argumentem je objekt, který bude představovat během volání hodnotu klíčového slova **this**. Obě funkce mají návratovou hodnotu typu tFUNC_RET_VALUE, což je struktura skládající se ze tří položek:

- Ukazatel na syntaktický strom funkce – ukazatel na uzel typu functionNode(popsán v kapitole 4.1), který obsahuje veškeré potřebné informace, s výjimkou řetězce oborů platnosti, k spuštění nového kontextu vykonávání pro volání funkce (konstruktoru).
- Ukazatel na řetězec oborů platnosti – je druhou částí nutnou pro spuštění nového kontextu vykonávání. Důvod proč řetězec oborů platnosti není možno asociovat s objektem třídy functionNode je podrobněji popsán v kapitole 5.1.
- Návratová hodnota funkce – objekt třídy nodeData představující návratovou hodnotu vestavěné funkce.

Tyto tři položky pokrývají všechny možnosti návratových hodnot funkcí a je tak zajištěn jednotný přístup pro všechny typy funkcí (uživatelská, vestavěná, eval, Function). Výsledek tohoto volání také určuje typ funkce, tedy je-li potřeba pro danou funkci zřizovat nový kontext vykonávání (uživatelská, eval) nebo pouze vložit výsledek po volání funkce na zásobník (vestavěná funkce nebo funkce Function). Pro lepší orientaci v problematice uvedu dvě tabulky. První tabulka ukazuje způsob, jak se podle návratové hodnoty z metody callThis určuje typ právě vykonávané funkce. V druhé tabulce jsou vypsány reakce interpretu v závislosti na zjištěném typu funkce. Tabulky jsou vytvořeny pouze pro volání funkce callThis – pro volání metody constructThis (tedy pokud je funkce volána jako konstruktor) je chování ve většině případů obdobné.

Položka struktury tFUNC_RET_VALUE			Typ funkce
Ukazatel na syntaktický strom funkce	Ukazatel na řetězec oborů platnosti	Návratová hodnota funkce	
NULL	NULL	Hodnota	Vestavěná funkce
not NULL	not NULL	hodnota undefined	Uživatelsky definovaná funkce
not NULL	NULL	hodnota undefined	vestavěná funkce eval
not NULL	NULL	hodnota jiná než undefined	vestavěná funkce Function

Tabulka 4.1 – Identifikovaný typ funkce v závislosti na návratové hodnotě metody callThis

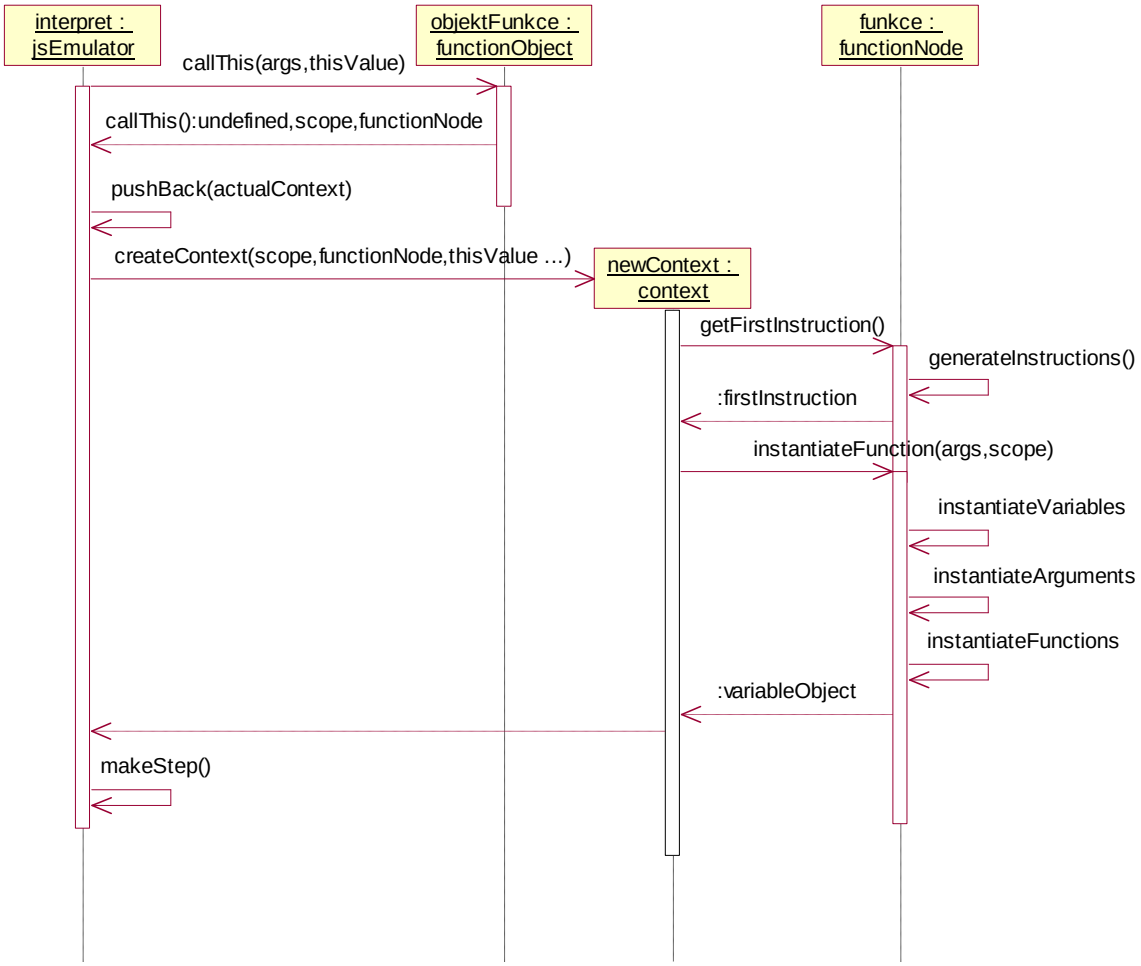
Typ funkce	Popis reakce interpretačního překladače
Vestavěná funkce	Vloží návratovou hodnotu funkce na zásobník.
Uživatelsky definovaná funkce	Vytvoří a spustí nový kontext vykonávání. Řetězec oborů platnosti a syntaktický strom, z kterého budou převzaty instrukce je daná návratovou hodnotou metody callThis.
Vestavěná funkce eval	Vytvoří a spustí nový kontext vykonávání. Syntaktický strom nového kontextu vykonávání je určen návratovou hodnotou metody callThis. Ostatní parametry kontextu vykonávání zůstanou stejné jako pro aktuální kontext vykonávání.
Vestavěná funkce Function	Je vytvořen nový objekt, který lze zároveň volat jako funkci (syntaktický strom patřící k dané funkci je opět dán návratovou hodnotou po volání metody callThis). K tomuto objektu je asociován aktuální řetězec oborů platnosti. Takto vytvořený objekt je umístěn na zásobník.

Tabulka 4.2 – Reakce interpretu v závislosti na typu volané funkce

Jelikož stále nemusí být zcela jasné jak volání funkcí JavaScriptu probíhá, doplním vysvětlení ještě o diagramy interakcí, na kterých by mělo být názorně vidět rozdíl mezi voláním vestavěné a uživatelem definované funkce.

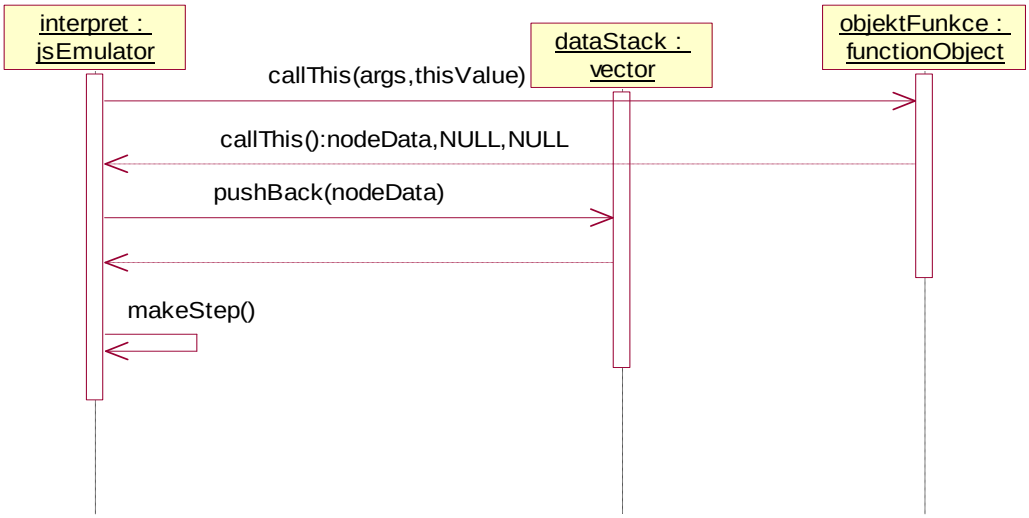
První diagram (Obrázek 4.5) zobrazuje průběh volání uživatelsky definované funkce. Nejprve je provedena metoda callThis volaného objektu, která nám jako výsledek vrátí řetězec oborů platnosti a ukazatel na příslušný functionNode. Nenulová hodnota těchto ukazatelů zároveň jednoznačně určeno, že jde o uživatelem definovanou funkci. Tyto dva údaje spolu s dalšími (hodnota klíčového slova this, typem vytvářeného kontextu apod.) jsou použity pro tvorbu nového kontextu vykonávání.

Při vytváření nového kontextu jsou skrz objekt třídy functionNode vygenerovány instrukce (pouze pokud je funkce volána poprvé) a provedena instanciace proměnných a funkcí, tedy vytvořen nový variable objekt, který je umístěn na začátek řetězce oborů platnosti. Po vytvoření nového kontextu začne interpret postupně provádět jednotlivé instrukce tohoto nového kontextu.



Obrázek 4.5 – Průběh volání uživatelsky definované funkce

Druhý diagram (Obrázek 4.6) taktéž zobrazuje průběh funkce, tentokrát ovšem vestavěné. Při volání této funkce je pouze zavolána metoda callThis volaného objektu. Výsledkem tohoto volání je navrácení struktury tFUNC_RET_VALUE popsané výše, přičemž hodnota ukazatelů na functionNode a řetězec oborů platnosti v této struktuře je rovna nule, čímž je jednoznačně určeno, že šlo o vestavěnou funkci. Třetí složka návratové hodnoty (objekt třídy nodeData) je následně uložena na zásobník a interpret pokračuje v provádění instrukcí stávajícího kontextu.



Obrázek 4.6 - Průběh volání vestavěné funkce

Třída nativeFunction

Jedná se o třídu, jejichž instance reprezentují uživatelsky definované funkce. Jelikož dědí od třídy functionObject musí implementovat metody callThis a constructThis. Ty jsou implementovány tak, že vrací vždy ukazatel na functionNode a řetěz oborů platnosti (tyto hodnoty jsou do objektu vloženy při vytváření objektu a jsou neměnné).

Třída builtInFunction

Opět se jedná o třídu, jejíž instance reprezentují funkce. Jak ovšem již název napovídá, jedná se v tomto případě o funkce vestavěné. Při vytváření toho typu objektu je nutné v konstruktoru předat ukazatele na funkce, které budou zastávat funkci a konstruktor. Takové funkce musí mít samozřejmě návratovou hodnotu a parametry typově stejné jako funkce callThis a constructThis. CallThis a constructThis metody následně pouze předávají parametry funkcím, které jsou implementacemi příslušných vestavěných funkcí, a následně vracejí příslušnou návratovou hodnotu (kromě vestavěných funkcí eval a Function je výsledkem vždy nějaká datová hodnota).

4.5 Třída objectStore

Každý interpretační překladač, a ani můj nebude výjimkou, by měl mít nějakým způsobem implementován způsob, kterak uvolnit za běhu programu z paměti objekty, které byly vytvořeny, ale již nejsou dále žádným způsobem odkazovány. V kapitole 3.2 jsem teoreticky rozebral nejzákladnější možné přístupy k dané problematice. Z nabízených variant jsem si pro implementaci vybral variantu „označ a uvolni“ (v cizí literatuře označovanou také jako „Mark and sweep“). Vybral jsem si jí,

jelikož je oproti druhé metodě schopna uvolnit opravdu všechny neodkazované objekty a je zároveň relativně jednoduchá na implementaci.

Proces uvolňování paměti je prováděn s využitím třídy objectStore, která představuje jakési skladiště všech vzniklých JavaScriptových objektů a zároveň následně provádí „čistění skladu“. Tato třída je implementována jako singleton a to hlavně z důvodu následné snadné implementace. Takto je totiž možné předávat tomuto skladišti reference na objekty jednoduše z konstruktoru třídy jsObject, jež je bázovou třídou pro všechny JavaScriptové objekty. Během ukládání tohoto objektu do skladiště je zároveň provedena kontrola, nebyla-li přesažena prahová hranice objemu objektů ve skladišti. V případě, že daná hranice byla překročena, jsou všechny viditelné objekty skrz instanci třídy jsEmulator označeny. Následně jsou sekvenčně testovány všechny objekty ve skladě a smazány ty, které nebyly v předchozím kroku označeny. Dále uvedu jednotlivé metody, pomocí nichž se komunikuje s instancí třídy objectStore:

- *getInstance* – Statická metoda, která vrátí ukazatel na singleton instanci třídy objectStore
- *putObject* – Metoda pomocí níž se provede registrace právě vzniklého objektu v tomto skladišti. Metoda má jeden argument, kterým je právě registrovaný objekt. Zároveň je metodou v případě, že byla přesažena kritická hranice objemu objektů, provedeno označení (pomocí metody markAll objektu třídy jsEmulátor) a uvolnění nepotřebných objektů.
- *registerEmulator* – Skrz tuto metodu je předán instanci třídy objectStore objekt typu jsEmulator, který má na starosti „označovací“ část procesu uvolňování paměti. Předání je provedeno samozřejmně pomocí parametru metody.

4.6 Třída jsEmulator

Třída jsEmulator je třída obalující celý mechanismus zajišťující interpretaci zdrojového kódu a komunikaci nadřazeným programem. Logicky je tato třída tvořena mechanismem pro interpretaci jednotlivých instrukcí, zásobníkem kontextů vykonávání a datovým zásobníkem. Datový zásobník je struktura, na které se nacházejí mezivýsledky po vykonání předchozích instrukcí. Jednotlivé položky datového zásobníku jsou typu nodeData. Tento zásobník je sdílený všem jednotlivým kontextům vykonávání. Jak již bylo zmíněno, třída musí zajišťovat možnost interpretace ve dvou režimech:

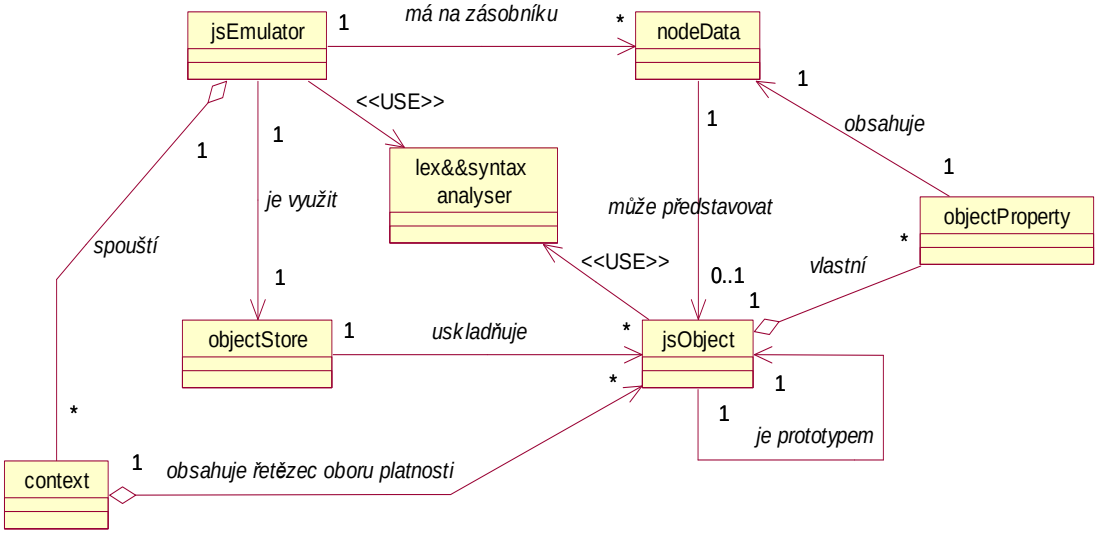
- Provedení určitého počtu elementárních kroků – při samotném interpretování si vzhledem k účelu, pro který je tato práce určena, není možné dovolit situaci, kdy se na vstup dostane JavaScriptový program obsahující nějaký typ nekonečné smyčky („`while (true);`“ apod.). Z tohoto důvodu bylo zadavatelem určeno, že aplikace musí podporovat i tento způsob interpretace, jakožto způsob pro vyhnutí se těmto nekonečným smyčkám.
- Provedení interpretace celého programu – minimálně jde o režim, který bude vhodné využívat při testování samotného interpretu.

Přehled důležitých veřejných metod poskytovaných třídou jsEmulátor:

- *Init* - Jedná se o metodu, která má jako vstupní parametr soubor se zdrojovým kódem JavaScriptového programu. Metoda provede kompletní inicializaci (případně reinicializaci) všech objektů a struktur nutných pro interpretaci. Dále je během této metody provedena syntaktická analýza. Tedy naplnění kořenového uzle syntaktického stromu.
- *Emulate* – Tato metoda spustí interpretaci vstupního programu (bude proveden celý).
- *makeSteps* – Začne se provádět interpretace vstupního programu. Bude proveden parametrem zadaný počet kroků instrukcí. Pozn.: Podle složitosti se může instrukce skládat i z více než jednoho kroku.

V době návrhu nebyl ještě zadavatelem přesně dán způsob, kterým interpret komunikuje s nadřazeným objektem. To je důvod proč zde funkce pro takovou komunikaci zatím chybí.

Následující diagram objektů, je zde uveden pro lepší pochopení vazeb mezi jednotlivými objekty a logické struktury celého JavaScriptového interpretu za běhu programu. Kvůli zjednodušení a přehlednosti zde nejsou uvedeny veškeré typy instancí JavaScriptových objektů, ale je zde zastoupena pouze instance třídy, ze které ostatní uzly dědí. Šipky s popiskem <<USE>> jsou zde pro doplnění celkové logické struktury překladače. Naznačují skutečnost, že instance třídy jsEmulator a jsObject nemají přímé reference na syntaktický (lexikální) analyzátor, jelikož lexikální ani syntaktický analyzátor nejsou součástí žádné třídy. Jedná se pouze o moduly, které poskytují třídám jsEmulator a jsObject funkci yyparse, zajišťující syntaktickou analýzu zdrojového textu. Dále zde kvůli zjednodušení také není zastoupena instance třídy představující syntaktický strom. Ten je vytvořen při procesu syntaktické analýza a použit pro vytvoření prvního kontextu vykonávání. Skrz tento kořenový uzel syntaktického stromu je kontextu vykonávání také přidělen iterátor na první prováděnou instrukci.



Obrázek 4.7 – zjednodušený diagram objektů interpretačního překladače

5 Implementace

V této kapitole uvedu nejprve některé implementační detaily, které byly v návrhu pouze lehce nastíněny nebo nebyly uvedeny vůbec. V dalších podkapitolách jsou umístěny podrobnosti o použitých technologiích, testování interpretu a částech jazyka, které zatím nebyly implementovány.

Vlastní interpretování příkazů

Jak již bylo uvedeno, je problém interpretace zdrojového kódu jazyka ECMAScript postupně převeden na problém interpretace posloupnosti jakýchsi elementárních instrukcí. Tyto instrukce se vždy skládají z typu instrukce a případně z parametru instrukce. Parametr instrukce nemá v hlavičkovém souboru implicitně uveden typ (respektive je typu ukazatel na `void`). Takto jsem to proved, jelikož jsem už od počátku věděl, že instrukce budou mít značné množství různých parametrů (objekty, literály, čísla, ukazatele na instrukce apod.). Také jsem si zpočátku nebyl zcela jistý, kolik různých typů parametrů bude potřeba. Takové to přetypovávání není samozřejmě úplně nejbezpečnější a při implementaci jsem si musel dávat pozor, aby k jednotlivým typům instrukcí patřil správný typ parametru instrukce. Pro větší přehlednost jsem v deklaraci výčtu jednotlivých typů instrukcí (soubor instrInterface.h) celkem podrobně okomentoval každý typ instrukce. Vždy je uveden typ parametru, změna velikosti zásobníku po provedení instrukce, případně další detaily. Při každém použití parametru instrukce, je nutné daný parametr přetypovat. Samotné interpretování instrukcí je prováděno pomocí jednoho rozsáhlého `switch` příkazu, kde ke každému typu instrukce je přiřazen kód, který se pro danou instrukci provede.

Pro lepší představu o tom jak vypadají a jaké akce provádějí jednotlivé instrukce, uvádím tabulku Tabulka 5.1 ukazující určitý typický vzorek instrukcí, které během interpretace používám. Sémantické akce nonterminálu v ECMAScript standardu jsou definovány pomocí sekvence určité elementárních akcí a právě tyto elementární akce většinou (pokud to při práci se zásobníkem bylo alespoň trochu efektivní) představují jednotlivé, mnou definované instrukce. Další instrukce manipulují se zásobníkem, případně provádějí skokové operace. Celkový počet instrukcí je kolem 75. V této kapitole jsem pojednával především o implementaci těchto souborů:

- instrInterface.h,.cpp – konstruktor a destruktor struktury tINSTRUCTION, výčet typů jednotlivých instrukcí.
- jsEmulator.h,.cpp – v těchto souborech je deklarována a definována třída jsEmulator, jež zajišťuje interpretaci jednotlivých instrukcí. V kapitole 4.4, kde je uveden návrh třídy jsEmulátor je zmíněné, že třída jsEmulátor by měla komunikovat s nadřazeným programem. Tato část nebyla zatím implementována, jelikož jsem se zatím nedohodl se zadavatelem, jakým způsobem bude tato komunikace prováděna.

Typ instrukce	Parametr instrukce	Popis akce
GET_VALUE	bez parametru	Jde o jednu z velmi frekventovaných akcí využívanou při popisování nonterminálů ECMAScript standardem. Konvertuje datový typ vršku zásobníku z typu reference na některý uživatelský datový typ.
PUSH_LITERAL	nodeData *	Vloží na zásobník kopii hodnoty, která je uložená v parametru instrukce.
POP_VALUE	bez parametru	Odstraní vrchol zásobníku.
PUT_VALUE	bez parametru	Vloží data, která jsou na zásobníku do vlastnosti objektu určenou referencí, která je uložena na zásobníku pod touto hodnotou.
TO_BOOLEAN	bez parametru	Změní datový typ a provede příslušné konverze hodnoty na vrcholu zásobníku.
JUMP_IF_TRUE	tINSTR_ITERATOR *	Interpret skočí na instrukci danou parametrem, pokud je na vrcholu zásobníku hodnota **true**.
OP_PLUS	bez parametru	Vezme horní hodnotu na zásobníku a připočte jí k hodnotě uložené na zásobníku pod touto hodnotou.
CALL_FUNCTION	bez parametru	Vytvoří nový kontext vykonávání. Argumenty funkce jsou uloženy na vrcholu zásobníku.
CONTEXT_END	bez parametru	Interpret ukončí vykonávání aktuálního kontextu. Jsou provedeny některé další akce, které se liší podle typu kontextu. Pokud je např. aktuální kontext funkcí a zároveň nebyla použita konstrukce **return**, je na zásobník uložena (návratová) hodnota **undefined**.

Tabulka 5.1 – Ukázka několika typických instrukcí

Implementace syntaktického stromu

Syntaktický strom se skládá z terminálních a neterminálních uzlů. K terminálním uzlům, již bylo pravděpodobně vše řečeno v návrhu. Nonterminální uzly mají obecně n-potomků – podle počtu terminálů a nonterminálů v daném pravidle, proto jsem se nesnažil vytvářet zvlášť unární, binární, atd. stromy, ale jednotlivé podstromy nonterminálu se ukládají jednoduše do kontejneru standardní knihovny vector. Každý konkrétní prvek stromu má k sobě asociovaný typ, který určuje, jaký prvek gramatiky představuje.

Každá třída představující určitou množinu nonterminálů syntaktického stromu, je také zodpovědná za generování příslušných instrukcí. Generování instrukcí je prováděno rekurzivně od příslušného startovního nonterminálu. Startovním nonterminálem v tomto případě nemusí být pouze nonterminál pokrývající celý program. Instrukce pro těla funkcí jsou generována až ve chvíli, kdy je daná funkce poprvé volána. V každém případě rekurzivní generování instrukcí započne nějaká instance třídy functionNode. Do této třídy je také ukládána příslušná posloupnost instrukcí. Jak už bylo zmíněno, každá třída reprezentující nějakou část syntaktického stromu, musí implementovat metodu generateInstr, která má jako parametr ukazatel na instanci rozhraní generateInstr (tedy na instanci třídy functionNode, která toto rozhraní implementuje). Skrz tento parametr jsou přidávány jednotlivé instrukce do seznamu instrukcí příslušného nonterminálního uzlu. Návratová hodnota z této funkce je iterátor na první instrukci, která během generováni instrukcí pro příslušný nonterminál vznikla. Této návratové hodnoty je využíváno pro nastavení parametru pro určité typy skokových instrukcí. Při samotném generování jsem se pokoušel vždy co nejpřesněji držet (pokud to bylo alespoň trochu efektivní) postupu uvedeného v ECMAScript standardu. Pokud jsem se výjimečně odchýlil od tohoto postupu, je to zdůrazněno pomocí komentáře ve zdrojovém kódu a samozřejmě je to provedeno tak, aby bylo výsledné chování programu stejné.

Syntaktický strom je vytvářen a instrukce generovány za pomocí tříd obsažených v těchto souborech:

- baseNode.h,.cpp – abstraktní předek všech tříd, z nichž se skládá syntaktický strom.
- nonterminalNode.h(.cpp), terminalNode.h, (.cpp) – tyto třídy představují nonterminál, respektive terminál syntaktického stromu.
- identifiedNode.h (.cpp), labelledNode.h (.cpp), functionNode.h (.cpp) – další specifické uzly syntaktického stromu.
- nodeFactory.h(.cpp) – obsahují třídu nodeFactory, která zprostředkovává a zajišťuje jednotný přístup při vytváření jednotlivých uzlů syntaktického stromu.

Vestavěné objekty jazyka JavaScript

ECMAScript definuje řadu vestavěných objektů. Vzhledem k časovým možnostem a relativně velkému množství objektů, jsem se omezil na implementaci pouze těch objektů a metod, které jsou obecně pro fungování JavaScriptového interpretu nezbytné, nebo byly používány ve virech, které mně byly zadavatelem práce dodány. Výčet implementovaných objektů a metod je v kapitole 5.3.

Pro každý vestavěný objekt byla vytvořena samostatná třída odvozená od třídy builtInObject, která byla implementována pomocí návrhového vzoru singleton. Tento návrhový vzor zajišťuje vytvoření daného objektu právě v jedné instanci. To je z hlediska implementace důležité hlavně pro provázání některých vestavěných objektů a jejich prototypů. Takto mně byla ulehčena práce za situace, kdy vytvářený vestavěný objekt potřebuje získat referenci na jiný vestavěný objekt (např. pro uložení daného objektu jako svého prototypu). Mnohdy jsou navíc tyto vazby mezi vytvářeným vestavěným objektem a jeho prototypovým objektem cyklické (prototypový objekt má vlastnost construct, která je referencí na objekt jehož je prototypem). Implementace jednotlivých vestavěných objektů jsou umístěný v těchto souborech:

- builtInFunction.h (.cpp) – je bázovou třídou pro konkrétní vestavěné objekty
- globalObject.h (.cpp) – implementace Global objektu (ECMAScript standard kap. 15.1).
- booleanObject.h (.cpp) – implementace Boolean objektu a příslušných prototypových objektů (ECMAScript standard kap. 15.6).
- stringObject.h (.cpp) – implementace String objektu a příslušných prototypových objektů (ECMAScript standard kap. 15.5).
- numberObject.h (.cpp) – implementace Number objektu a příslušných prototypových objektů (ECMAScript standard kap. 15.7).
- objectObject.h (.cpp) – implementace Object objektu a příslušných prototypových objektů (ECMAScript standard kap. 15.7).

Implementace procesu uvolňování paměti

Při implementaci jsem se v podstatě držel navrhnutého způsobu řešení. Přesto je implementace pouze dosti provizorní, jelikož způsob, jakým se rozhoduje o okamžiku, kdy začne proces uvolňování paměti, je naprosto nevyhovující. V současném stavu reaguje objekt zajišťující uvolňování paměti na překročení určité prahové hodnoty počtu vytvořených objektů. Tato prahová hodnota je v případě potřeby posunuta. Tato hodnota je testována vždy před vytvořením nového objektu. Tento způsob byl zvolen pouze pro naprostou triviálnost implementace. V budoucnu by bylo jistě vhodnější, aby interpret reagoval spíše na překročení určité velikosti celkového alokovaného prostoru nebo na nedostatek systémových prostředků.

5.1 Použité technologie

Implementace byla prováděna pod operačním systémem Windows XP v jazyce C++. Tento programovací jazyk byl vybrán zadavatelem práce. Konkrétně bylo použito výbojové prostředí Visual Studio 2005 s příslušným překladačem. Během implementace byla využita pouze standardní knihovna jazyka C++. Dále byly při implementaci použity nástroje Flex (verze 2.5.4) a Bison (verze 2.1), jejichž výstup je přeložitelný pomocí překladače jazyka C++.

Ač byl celý interpretační překladač vyvíjen pod systémem Windows, je vzhledem k faktu, že jsou použity pouze standardní knihovny jazyka C++, možný jeho bezproblémový přenos i pod systémem Unix a pravděpodobně i na jiné systémy, kde je nainstalován nějaký překladač jazyka C++ podporující standardní knihovny. U systémů pro něž neexistuje nějaký příslušná varianta programů Flex nebo Bison, musí být provedeno přenesení již vygenerovaných souborů obsahující moduly lexikálního a syntaktického analyzátoru. Pro pohodlný překlad je v adresáři \src připraven makefile.

Nástroje pro tvorbu překladačů

V této kapitole popíši nástroje, které jsem použil pro tvorbu lexikálního a syntaktického analyzátoru. Jsou jimi programy Flex a Bison. Konkrétně tyto dva nástroje jsem si vybral z těchto důvodů. Prvním důvodem je, že jsem se s oběma nástroji setkal při svém studiu a tedy jsem s nimi měl bohaté zkušenosti. Druhým důvodem je skutečnost, že se jedná o nástroje, které jsou využívány pro tvorbu překladačů již dlouhou dobu a je zde tedy docela velká záruka, že nástroje sami o sobě neobsahují chyby, které by mně mohli při tvorbě překladače působit potíže. V neposlední řadě je významná i rozsáhlá vývojářská komunita, která se kolem těchto dvou nástrojů na internetu vyskytuje. Oba dva programy navíc spolu dokáží spolupracovat.

5.1.1 Flex

Flex je program určený ke generování programů v jazyce C, schopných rozeznávat regulární výrazy (v mém případě tedy lexikální analyzátor). Vstupem Flexu je textový soubor se specifikací úlohy, kterou má generovaný výstupní soubor ovládat.

Vstupní soubor Flexu se skládá ze tří částí. Jednotlivé sekce jsou odděleny dvojznakem „%%“. První sekce se nazývá definiční. Obsahuje deklarace identifikátorů (tedy zástupné identifikátory pro regulární výrazy) a. Mezi znaky „%{„ a „%}“ můžeme v první sekci vkládat kód v jazyce C, který bude beze změn okopírován do výsledného zdrojového kódu (je to tedy vhodná sekce pro vložení hlavičkových souborů a globálních proměnných).

Druhá sekce je složena z dvojic regulární výraz – kód v jazyce C, který říká, co se má vykonat, narazíme-li na příslušný regulární výraz. Klasicky zde provedeme vrácení typu právě nalezeného

tokenu a případně jeho hodnotu, kterou uložíme do proměnné yylval. Tyto informace jsou nadřazenému programu (tedy syntaktickému analyzátoru - Bisonu) předány po volání funkce yylex().

Do třetí sekce může uživatel vložit kód, který bude beze změny okopírován do generovaného souboru. Jsem jsou typicky vkládány různé funkce. Podrobný popis lze nalézt v [8].

Implementace lexikálního analyzátoru

Implementace lexikálního analyzátoru pomocí nástroje lex byla relativně bezproblémová. Jedinou výjimkou byl problém s rozlišením literálu pro regulární výraz od operátorů „/=“ a „/“. Lexém regulárního výrazu je nástrojem lex vyhledán pro vzor (uvádím pouze ve zjednodušené formě, přesná pravidla pro vyhledání tohoto typu lexému jsou ve zdrojovém souboru JS_Lex.l):

“/”{ZnakyRegulárníhoVýrazu}+”/”PříznakyRegulárníhoVýrazu

Problém tedy nastává, pokud zdrojový text JavaScriptového programu má například tento tvar: „a = 3 / 3 / 3;“. Nástroj Lex v takovémto případě pošle syntaktickému analyzátoru na vstup postupně tokeny „3“ a token literálu regulárního výrazu „/ 3 /” (pozn. lex dává prioritu vzorům s větší délkou – proto je pro tento vstup navrácen literál regulárního výrazu). Na úrovni ECMAScript specifikace je toto řešeno, takto. Pokud lze ze syntaktického hlediska použít “/” nebo „/=“, použijeme je. Jinak v daném syntaktickém kontextu použijeme literál regulárního výrazu. Toto je ovšem s využitím nástroje Bison dosti těžko realizovatelné (je-li to vůbec možné) a z tohoto důvodu musím provést rozhodnutí již na úrovni lexikálního analyzátoru. Rozhodnutí jedná-li se o literál regulárního výrazu, nebo o dělení („/“ nebo „/=“), je provedeno čistě na základě typu předchozího tokenu. Lexém bude vyhodnocen jako dělení, pokud bude předcházející lexém typově roven jednomu z těchto lexémů: Identifikátor, číslo, řetězec, regulární výraz, „)“, ‘++‘, ‘--', ‘]‘, ‘}‘, ‘false‘, ‘true‘, ‘null‘, ‘super‘ nebo ‘this‘. Ve všech ostatních případech bude lexém vyhodnocen jako regulární výraz. Návrh řešení tohoto problému byl převzat z [10].

5.1.2 Bison

Bison je program, který podle zadané gramatiky generuje syntaktický analyzátor v jazyce C. Soubor popisující činnost se opět skládá ze tří částí. Gramatika je zadávána ve tvaru pravidlo gramatiky – kód v jazyce C, který se vykoná po redukování daného pravidla. Detailní popis nástroje lze nalézt v [9].

Implementace syntaktického analyzátoru

Při implementaci kostry syntaktického analyzátoru pomocí programu Bison, jsem narazil na několik problémů v gramatice, které jsem musel řešit. Problémy způsoboval především nestandardní zápis, některých gramatických pravidel ve specifikaci, které nelze přímo vkládat jako pravidla vstupního souboru programu Bison.

Prvním nestandardním zápisem gramatiky je část gramatiky rozlišující literály objektu a funkce od bloku a deklarace funkce. Jedná o zápis tohoto gramatického pravidla:

ExpressionStatement -> [další token nepatří mezi {**{, function**}] Expression **;**

Toto gramatické pravidlo v podstatě říká, že žádný příkaz, který začíná výrazem, nesmí na začátku obsahovat lexémy „{“ a „function“. Tedy v podstatě nesmí příkaz začínat literálem funkce nebo literálem objektu. Je to z důvodu zabránění nejednoznačnosti gramatiky mezi literálem funkce a deklarací funkce a mezi literálem objektu a blokem. Jelikož Bison neobsahuje možnost explicitní kontroly následujících tokenů během kontroly gramatického pravidla, musel jsem použít relativně neelegantní metody. Zavedl jsem celou novou větev gramatiky vytvořenou speciálně pro příkaz výrazu. Tato část gramatiky neobsahuje již možnost začlenění tokenů „{“ a „function“ do nejlevější části výrazu. Zároveň je tato gramatická větev postavena tak, aby se v ostatních jejich částech mohly objevovat i výše zmíněné tokeny.

Druhá část gramatiky, s jejíž úpravou jsem se musel vypořádat je nestandardní část gramatiky, zapsaná s využitím terminálu, který by neměl být součástí syntaktické gramatiky. V souvislosti s tímto problémem budu diskutovat tyto gramatická pravidla:

ReturnStatement -> return [no LineTerminator here] Expressionopt ;

BreakStatement -> break [no LineTerminator here] Expressionopt ;

ContinueStatement -> continue [no LineTerminator here] Expressionopt ;

ThrowStatement -> throw [no LineTerminator here] Expressionopt ;

V následujícím textu budu označovat tyto klíčová slova jako problémová: `break`, `continue`, `throw` a `return`. Výše uvedená pravidla obsahují ve specifikaci terminál, který jinak sám o sobě nemůže proniknout mezi terminály syntaktického stromu. Pro vyřešení prvního problému jsem v lexikálním analyzátoru nadefinoval další typ lexému pro každé problémové slovo. Tento lexém byl vždy popsán regulárním výrazem:

(Problémové_slovo) (Mezery+Tabulátory)* (KonecŘádku)

Tyto lexémy měli při lexikální analýze vyšší prioritu, než jejich varianty bez konce řádku. K vyřešení problému již pouze stačilo přidat tyto gramatická pravidla:

ReturnStatement -> return_EOL

BreakStatement -> break_EOL

ContinueStatement -> continue_EOL

ThrowStatement -> throw_EOL

Další komplikace při výstavbě syntaktického analyzátoru je spojena s vyřešením problému týkajícího se automatického vkládání středníků (popsáno v kapitole 2.1.). Problém opět způsobuje fakt, že o syntaktické korektnosti může rozhodovat lexém, který není součástí gramatiky (tedy konec řádky). Řešení je tedy opět výsledkem určité kooperace lexikálního a syntaktického analyzátoru. Do původní gramatiky jazyka jsem přidal ke všem pravidlům, u nichž je možnost automatického vkládání středníků, její variaci, která místo terminálního symbolu středníku obsahuje speciální terminální symbol programu bison „error“. Pravidlo, s tímto terminálním symbolem, je redukováno pokud nelze vykonat žádnou jinou akci (přesun nebo redukci). Tento terminální symbol se běžně

používá v souvislosti se zotavením po chybě. V mém případě, ale toto pravidlo vyvolá spuštění funkce, která zkontroluje, je-li na daném místě možno vložit středník. To je možné, pokud má lexikální analyzátor nastaven příznak indikující konec řádky před aktuálním lexémem, nebo pokud byl terminální symbol „error“ použít místo terminálu „}“. Pokud je středník „vložen“, následuje standardní redukce pravidla gramatiky. V opačném případě je vstup vyhodnocen jako syntakticky nesprávný.

5.2 Implementovaní odlišností od ECMAScript standardu

Téměř každá implementace ECMAScript jazyka se nějakým způsobem odlišuje od standardu, a jelikož tato diplomová práce je zamýšlena jako nástroj pro odkrývání zatemňovacích schránek virů, bylo vhodné, aby se přinejmenším podobala některé implementaci tohoto standardu. Po dohodě se zadavatelem této diplomové práce jsem vybral implementaci pro produkt Internet Explorer 7 (dále jen IE7). Jelikož jsem nikde nenašel zcela přesnou specifikaci tohoto jazyka (některé informace je možné nalézt v [7]), nebylo možné implementovat všechny odlišnosti, na které jsem přišel a to hlavně z důvodu, že jsem nebyl ve většině případů schopen přijít na způsob, jakým byla nedodržena specifikace ECMAScirpt standardu. Nakonec se mně povedlo implementovat pouze rozdílný způsob jakým se IE7 staví k funkcím definovaným v rámci výrazu. U některých dalších odlišností jsem měl i hrubou představu o tom jakým způsobem je standard nedodržen, ale obával jsem, že by můj způsob implementace mohl mít následky v jiných částech programu, který se jinak snaží držet standardu, a proto jsem další odlišnosti raději neimplementoval.

Definice funkcí ve výrazu

Dle ECMAScript standardu lze definovat funkci i uvnitř výrazu, ale identifikátor funkce je viditelný pouze uvnitř těla této funkce (pro případ rekurzivního volání této funkce). Z jiného místa programu by neměl být identifikátor této funkce viditelný. Interpret prohlížeče IE7 ovšem dovoluje volat funkce definované ve výrazu odkudkoliv (tedy i před samotným definováním této funkce). Z této odlišnosti plyne pro implementaci další nepříjemný fakt. Pokud je totiž možné funkce ve výrazu odkazovat odkudkoliv, je možné, že nastane případ, kdy jedno tělo funkce bude pro dvě různá volání (myšleno samozřejmě se stejnými parametry) provádět dvě odlišné interpretace těla funkce. Jak je to možné? Pro snadnější a ilustrativnější vysvětlení uvedu příklad:

```
//Zdrojový kód programu
a = {x:10}; // literál objektu
x = 5;
with(a) // Na začátek řetězce oborů platnosti se vloží objekt a
{
c =   function B()
```

```
      {
            return x;
      }
}
document.writeln(c());
document.writeln(B());
-- výstup interpretace programu IE7
10 5
```

Postup vytváření objektů (funkcí) je takovýto. Při vstupu do programu je vytvořena funkce `B` (toto je v rozporu s ECMAScript standardem). Jedna z hodnot nutná při vytváření funkce je i aktuální řetězec oborů platnosti, tento řetězec obsahuje v našem případě pouze jeden prvek a tím je globální objekt. Dále je ve zdrojovém textu příkaz `with`, který vloží na začátek řetězce oboru použití objekt `a`. Následuje příkaz v jehož rámci je vytvořena další funkce, která má stejné tělo a parametry jako funkce `B`. Rozdíl je právě v řetězci oboru použití, který kromě globálního objektu obsahuje také objekt `a`. Následně tedy funkce uložená v proměnné `B` vrátí výsledek 5 (proměnná `x` je nalezena jako vlastnost globálního objektu) a funkce uložená v proměnné c vrátí výsledek 10 (proměnná `x` je nejprve vyhledávána v objektu `a`). Způsob jakým je tento problém řešen je podrobněji vysvětlen v kapitole 4.4.2 .

5.3 Implementované části jazyka

Jelikož je JavaScript jazykem relativně rozsáhlým, věděl jsem již po zadání diplomové práce, že nebude v mojich silách dokončit implementaci interpretu v plném rozsahu standardu ECMAScript. Při implementaci jsem se proto zaměřil prioritně na takové jazykové konstrukce, které byly použity v kódech JavaScriptových virů (byly mi dodány zadavatelem). Následně jsem postupně implementoval další nejběžnější jazykové konstrukce. Z vestavěných objektů jazyka JavaScript jsem implementoval pouze ty, které byli nutné k implementaci jiných vnitřních mechanismům, případně pokud se vyskytovali v některém z virů. Dále v této podkapitole uvedu tabulku, která obsahuje přehled implementovaných/neimplementovaných konstrukcí jazyka. Symbolem „A“ budou označeny ty konstrukce, které jsou implementovány. „N“ bude značit opak.

Obecné konstrukce	Implementováno
Identifier, this, Literály	A (s výjimkou literálu regulárního výrazu)
Object literál	A
Array literál	N
new operator	A
function call	A
Zpřístupnění vlastnosti objektu	A
Operátory	**Implementováno**
Postfixové	A
Unární	A (s výjimkou operátoru „~“)
Multiplikativní	A
Aditivní	A
Bitové	N
Operátory porovnání (<,<= atd.)	A (s výjimkou operátoru `instanceof`)
Operátory rovnosti (= =,!=,= = =,atd.)	A
Binární bitové operátory	N
Podmínkový operátor (?:), operátor ‘,‘	A
Příkazy	**Implementováno**
Blok, Deklarace proměnných, prázdný příkaz	A
Výraz	A
if,if-else	A
Iterační příkazy	A (výjimkou konstrukce for-each)
try-catch-finally	A
with	A

Tabulka 5.2 – Implementované konstrukce jazyka

I pro konstrukce, které jsou zde uvedeny jako neimplementované, je provedena syntaktická analýza. Syntaktický analyzátor je tedy kompletní pro všechny konstrukce jazyka JavaScript. Nedokončenost některých konstrukcí se projeví, až ve chvíli, kdy se program pokouší vygenerovat instrukce pro neimplementovaný nonterminál (terminál). V takovém případě je na výstup vypsáno chybové hlášení informující o nedokončeném procesu generování instrukcí.

Výčet implementovaných vestavěných objektů a metod

V tabulce Tabulka 5.3 Tabulka 2.3jsem vypsal jednotlivé implementované vestavěné objekty jazyka JavaScript a k nim příslušné metody. Sémantika příslušných metod je uvedena v ECMAScript specifikaci.

Název objektu	Implementované metody objektu
Global	eval, print, unescape
Object	
Object prototype	toString, toLocaleString
String	
String prototype	toString, valueOf, charAt, indexOf
Number	
Number prototype	toString, valueOf
Boolean	
Boolean prototype	toString, valueOf

Tabulka 5.3 – Výčet implementovaných vestavěných objektů

5.4 Testování interpretačního překladače

Tato diplomová práce je koncipovaná jako specializovaný interpret, který je určený pro obcházení zatemňovacích schránek některých počítačových virů, ale jelikož nemůžu (před případným zařazením do reálně fungujícího produktu zbývá ve vývoji ještě relativně dlouhá cesta) ukázat fungování interpretu ve spolupráci s nadřazeným programem, použiji pro demonstraci funkčnosti a otestování správného chování implementovaných tříd aplikaci, která se chová jako obyčejný JavaScriptový interpret.

Po zkompilování zdrojových souborů je tedy vytvořena aplikace, která očekává jediný parametr, kterým by měl být soubor obsahující zdrojový text napsaný v jazyce JavaScript. Aplikace následně provede interpretaci tohoto zdrojového textu a ukončí svojí činnost.

Jelikož je potenciálním vstupem interpretu nekonečná množina JavaScriptových programů, není samozřejmě možné otestovat všechny potencionální vstupy. Vzhledem k tomuto faktu je téměř jisté, že řada potencionálních chyb zůstane i přes testování neošetřena.

K testování interpretu jsem použil sadu testů, která se snaží pokrýt co možná nejrozmanitější spektrum konstrukcí jazyka. Jelikož je navíc JavaScript jazykem dynamicky typovaným jsou pro každý operátor vytvořeny testy pro všechny kombinace datových typů. Jednotlivé testy pokrývají pouze konstrukce, které jsou interpretem podporovány (tedy byly implementovány). Testy jsou

zaměřeny především na správnost interpretace. Vzhledem k účelu, pro který je práce vytvářena není primárně důležité testovat správnost chybových hlášení apod.

V adresáři \test\t jsou umístěny jednotlivé testovací soubory[6]. Ke každému vstupnímu testovacímu souboru jsem vytvořil soubor, ve kterém je umístěn očekávaný výstup. Ve většině případů je očekávaný výstup shodný dle ECMAScript standardu. Pro tyto případy jsem očekávané výstupy vytvořil pomocí prohlížeče Mozilla Firefox 5, která má odchylek od standardu relativně málo. Pro vstupy, u kterých jsem věděl, že jsou prohlížečem IE7 interpretovány jinak a zároveň jsem dokázal toto nestandardní chování ve svém interpretu napodobit jsem vzorové výstupy tvořil pomocí výstupů prohlížeče IE7. Kromě těchto dvojic vstupní soubor-očekávaný výstup je v adresáři \test umístěn také dávkový soubor test_win.bat, jež postupně pošle na vstup interpretu všechny testovací soubory. Výstupy interpretu jsou následně porovnány s příslušnými očekávanými výstupy. Zpráva o testování je ukládána do souboru result.txt. Tento soubor obsahuje výpis souborů, u kterých byla zjištěna špatná interpretace a celkovou statistiku o úspěšných a celkově provedených testech.

Výsledky testování

Výsledky testování pro aktuální verzi interpretu jsou uloženy v souboru „result.txt“. Celkově bylo testováno 271 vstupních souborů. Z nich bylo jako správně interpretových označeno celkem 258 souborů. Úspěšnost testu byla tedy celkově 95,2%. Většina z chybných interpretací byla způsobena špatným způsobem formátování čísel (čísla s desetinou čárkou, případně čísla s exponentem), což jsem vyhodnotil jako relativně podřadnou chybu a rozhodl se jí prozatím neopravovat. Další část chyb při interpretaci byla způsobena absencí implementace některých vestavěných objektů. Z těch důležitějších budu jmenovat objekty Error a Function. Jelikož se jedná o rozsáhlý jazyk, jsem s výsledy testů poměrně spokojen.

[6] Většina souborů převzata z projektu PJS[12]

6 Závěr

Cílem práce bylo navrhnout a implementovat interpretační překladač jazyka JavaScript. Vzhledem k rozsáhlosti JavaScriptu nebyl interpret implementován v celém rozsahu, ale povedlo se pokrýt všechny důležité jazykové konstrukce a všechny vestavěné objekty, jež byly zapotřebí pro zprovoznění hlavních vnitřních mechanismů tohoto jazyka. Podrobněji je výpis implementovaných/ neimplementovaných částí jazyka uveden v kapitole 5.3 . S výsledky testování funkčnosti interpretace jazyka jsem byl relativně spokojen. Interpretace byla prováděna rozdílně většinou pouze v případech, kdy bylo potřeba vypsat číselné hodnoty s konkrétním formátováním.

Během vypracovávání diplomového projektu jsem se důkladně seznámil se specifikací jazyka JavaScript. Dále jsem se seznámil s vnitřními mechanismy, pomocí kterých je standardem popisována interpretace jednotlivých uzlů gramatiky. Se znalostí těchto mechanismů jsem provedl základní návrh interpretu jazyka JavaScript. Jelikož jsem se s návrhem takto rozsáhlého interpretu zatím nesetkal, musel jsem během implementace tento návrh mírně upravovat. Základní struktura návrhu zůstala ovšem nezměněna. Takto navrhnutý interpret byl následně implementován.

Směr dalšího možného vývoje tohoto projektu je také celkem zřejmý. V prvé řadě je potřeba dodělat chybějící jazykové konstrukce a vestavěné objekty. Dále je potřeba se zaměřit na různé optimalizace programu, jelikož rychlost interpretace není v současné době ideální. Absence jakýchkoliv optimalizací je způsobena hlavně doporučením zadavatele, abych se primárně zaměřil na kompletnost implementace. Hlavní pozornost bych při optimalizacích zaměřil především na proces generování a interpretace instrukcí, kde budou pravděpodobně největší výkonnostní mezery. Také by bylo vhodné připravit komplexnější testy, které by odhalily další chyby. Zejména testování případů, kdy je na vstup programu vložen neplatný zdrojový kód jazyka JavaScript, není v současné době v dostatečné míře ošetřeno.

Literatura

[1] AHO, Alfred V., et al. *Compilers : Principles, Techniques, & Tools*. Boston : Pearson Education, c2007.

[2] MEDUNA, A., Podklady k předmětu VYP, VUT v Brně, FIT 2005

[3] HASHIM, Habiballa. *Překladače*. Ostavská univerzita v Ostravě, 2005.

[4] HÁK, Igor. *Viry.cz* [online]. c1998-2007 [cit. 2008-05-01]. Dostupný z WWW: <http://www.viry.cz/>.

[5] FLANAGAN, D., *JavaScript Kompletní průvodce*. Praha : Computer Press 2002.

[6] *Standard ECMA-262 : ECMAScript Language Specification 3 rd edition* [online], 1999. Dostupný z WWW: <http://www.ecma-international.org/publications/standards/Ecma-262.htm>.

[7] *Microsoft Developer Network : JScript Language Reference* [online]. c2008 [cit. 2008-03-13]. Dostupný z WWW: <http://msdn.microsoft.com/en-us/library/yek4tbz0(VS.85).aspx>.

[8] *Flex : The Fast Lexical Analyzer* [online]. 1997-2008 , last updated 2008-02-26 [cit. 2008-05-01]. En. Dostupný z WWW: <http://flex.sourceforge.net/>.

[9] *Bison : GNU parser generator* [online]. 1998-2007 , last updated 2007-04-13 [cit. 2008-04-01]. Dostupný z WWW: <http://www.gnu.org/software/bison/>.

[10] *JavaScript 2.0* [online]. 1999-2000 , last updated 2000-6-6 [cit. 2008-04-23]. Dostupný z WWW: <http://www.mozilla.org/js/language/js20-2000-07/>.

[11] *Wikipedia : The Free Encyclopedia : Virtual function* [online]. c1991-2006, 2008-04-30 [cit. 2008-05-04]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Virtual_function>.

[12] SOYTURK, Mehmet Yavuz Selim. *PJS : Parrot JavaScript* [online]. [2007] [cit. 2008-04-03]. Dostupný z WWW: <http://users.fulladsl.be/spb1622/pjs/>.

Seznam příloh

Příloha 1. Obsah přiloženého CD

- \bin\windows – Spustitelný kód JavaScriptového interpretu (kompilované pod systémem Windows).
- \bin\unix – Spustitelný kód JavaScriptového interpretu (kompilované pod systémem Unix).
- \doc\ – Technická zpráva ve formátech .doc a .pdf.
- \src\ - Zdrojové kódy jazyka C++, nástroje Bison, nástroje Flex a příslušný makefile.
- \test\ - Script pro spuštění testování interpretačního překladače. Dále soubor s výsledky testování, který vznikl použitím binárního souborů uloženého ve složce \bin\.
- \test\t – Soubory určené k testování interpretačního překladače a příslušné očekávané výstupy.

Příloha 2. Gramatika použitá pro syntaktickou analýzu

Jedná se o mírně upravenou ECMAScript gramatiku, která je bez problémů přijímána LALR syntaktickým analysátorem Bison. Pravidla pomocí nichž jsem se vypořádal s automatickým vkládáním středníku, zde nejsou obsažena, jelikož bez bližšího popisu by nikomu příliš nedávala smysl. Popsána jsou v kapitole 5.1.2 . Dále se na pravých stranách vyskytují pravidla, která obsahují koncovku „_OPT“. Tato koncovka naznačuje fakt, že se zde daný nonterminál/terminál vyskytuje pouze volitelně.

Levá strana pravidla	Pravá strana pravidla
Program	StatementList
FunctionBody	StatementList
StatementList	Statement
	StatementList Statement
Statement	;
	FunctionDecl
	Block
	VarStatement
	ExprStatement
	IfStatement
	IterationStatement
	ContinueStatement
	ReturnStatement
	BreakStatement
	WithStatement
	SwitchStatement
	LabelledStatement
	ThrowStatement
	TryStatement
ThrowStatement	THROW Expr ;
TryStatement	TRY Block Catch
	TRY Block Finally
	TRY Block Catch Finally
Catch	CATCH (IDENTIFIER) Block
finally	FINALLY Block
LabelledStatement	IDENTIFIER : Statement
SwitchStatement	SWITCH (Expr) CaseBlock

CaseBlock	{ CaseClauses_OPT }
	{ CaseClauses_OPT DefaultClause CaseClauses_OPT }
CaseClauses	CaseClause
	CaseClauses CaseClause
CaseClause	CASE Expr : StatementList
	CASE Expr :
DefaultClause	DEFAULT : StatementList
	DEFAULT :
WithStatement	WITH (Expr) Statement
BreakStatement	BREAK Identifier_OPT ;
ReturnStatement	RETURN Expr_OPT ;
ContinueStatement	CONTINUE Identifier_OPT ;
ExprStatement	ExprNoFunc ;
IfStatement	IF (Expr) Statement
	IF (Expr) Statement ELSE Statement
IterationStatement	DO Statement WHILE (Expr) SEMICOLON
	WHILE (Expr) Statement
	FOR (ExprNoIn_OPT ; Expr_OPT ; Expr_OPT) Statement
	FOR (VAR VarDeclListNoIn ; Expr_OPT ; Expr_OPT) Statement
	FOR (LeftHandSideExpr IN Expr)Statement
	FOR(VAR VarDeclNoIn IN Expr) Statement
VarStatement	VAR VarDeclList ;
VarDeclList	VarDecl
	VarDeclList , VarDecl
VarDeclListNoIn	VarDeclNoIn
	VarDeclListNoIn,VarDeclNoIn
VarDecl	IDENTIFIER Initialiser_OPT
VarDeclNoIn	IDENTIFIER InitialiserNoIn_OPT
Initialiser	EQUAL AssigExpr
InitialiserNoIn	EQUAL AssigExprNoIn
AssigExpr	ConditionalExpr
	LeftHandSideExpr AssigOperator AssigExpr
AssigExprNoFunc	ConditionalExprNoFunc
	LeftHandSideExprNoFunc AssigOperator AssigExpr

AssigExprNoIn	ConditionalExprNoIn
	LeftHandSideExpr AssigOperator AssigExprNoIn
ConditionalExpr	LogicalORExpr
	LogicalORExpr ? AssigExpr : AssigExpr
ConditionalExprNoFunc	LogicalORExprNoFunc
	LogicalORExprNoFunc ? AssigExpr : AssigExpr
ConditionalExprNoIn	LogicalORExprNoIn
	LogicalORExprNoIn ? AssigExprNoIn : AssigExprNoIn
LogicalORExpr	LogicalANDExpr
	LogicalORExpr || LogicalANDExpr
LogicalORExprNoFunc	LogicalANDExprNoFunc
	LogicalORExprNoFunc || LogicalANDExpr
LogicalORExprNoIn	LogicalANDExprNoIn
	LogicalORExprNoIn || LogicalANDExprNoIn
LogicalANDExpr	BitwiseORExpr
	LogicalANDExpr && BitwiseORExpr
LogicalANDExprNoFunc	BitwiseORExprNoFunc
	LogicalANDExprNoFunc && BitwiseORExpr
LogicalANDExprNoIn	BitwiseORExprNoIn
	LogicalANDExprNoIn && BitwiseORExprNoIn
BitwiseORExpr	BitwiseXORExpr
	BitwiseORExpr | BitwiseXORExpr
BitwiseORExprNoFunc	BitwiseXORExprNoFunc
	BitwiseORExprNoFunc | BitwiseXORExpr
BitwiseORExprNoIn	BitwiseXORExprNoIn
	BitwiseORExprNoIn | BitwiseXORExprNoIn
BitwiseXORExpr	BitwiseANDExpr
	BitwiseXORExpr ^ BitwiseANDExpr
BitwiseXORExprNoFunc	BitwiseANDExprNoFunc
	BitwiseXORExprNoFunc ^ BitwiseANDExpr
BitwiseXORExprNoIn	BitwiseANDExprNoIn
	BitwiseXORExprNoIn ^ BitwiseANDExprNoIn
BitwiseANDExpr	EqualityExpr
	BitwiseANDExpr & EqualityExpr
BitwiseANDExprNoFunc	EqualityExprNoFunc

	BitwiseANDExprNoFunc & EqualityExpr
BitwiseANDExprNoIn	EqualityExprNoIn
	BitwiseANDExprNoIn & EqualityExprNoIn
EqualityExpr	RelationalExpr
	EqualityExpr == RelationalExpr
	EqualityExpr != RelationalExpr
	EqualityExpr === RelationalExpr
	EqualityExpr !== RelationalExpr
EqualityExprNoFunc	RelationalExprNoFunc
	EqualityExprNoFunc == RelationalExpr
	EqualityExprNoFunc != RelationalExpr
	EqualityExprNoFunc === RelationalExpr
	EqualityExprNoFunc !== RelationalExpr
EqualityExprNoIn	RelationalExprNoIn
	EqualityExprNoIn == RelationalExprNoIn
	EqualityExprNoIn != RelationalExprNoIn
	EqualityExprNoIn === RelationalExprNoIn
	EqualityExprNoIn !== RelationalExprNoIn
RelationalExprNoFunc	ShiftExprNoFunc
	RelationalExprNoFunc < ShiftExpr
	RelationalExprNoFunc > ShiftExpr
	RelationalExprNoFunc <= ShiftExpr
	RelationalExprNoFunc >= ShiftExpr
	RelationalExprNoFunc INSTANCEOF ShiftExpr
	RelationalExprNoFunc IN ShiftExpr
RelationalExprNoIn	ShiftExpr
	RelationalExprNoIn < ShiftExpr
	RelationalExprNoIn > ShiftExpr
	RelationalExprNoIn <= ShiftExpr
	RelationalExprNoIn >= ShiftExpr
	RelationalExprNoIn INSTANCEOF ShiftExpr
RelationalExpr	ShiftExpr
	RelationalExpr < ShiftExpr
	RelationalExpr > ShiftExpr
	RelationalExpr <= ShiftExpr

	RelationalExpr >= ShiftExpr
	RelationalExpr INSTANCEOF ShiftExpr
	RelationalExpr IN ShiftExpr
ShiftExpr	AdditiveExpr
	ShiftExpr << AdditiveExpr
	ShiftExpr >> AdditiveExpr
	ShiftExpr >>> AdditiveExpr
ShiftExprNoFunc	AdditiveExprNoFunc
	ShiftExprNoFunc << AdditiveExpr
	ShiftExprNoFunc >> AdditiveExpr
	ShiftExprNoFunc >>>AdditiveExpr
AdditiveExpr	MultiplicativeExpr
	AdditiveExpr + MultiplicativeExpr
	AdditiveExpr - MultiplicativeExpr
AdditiveExprNoFunc	MultiplicativeExprNoFunc
	AdditiveExprNoFunc + MultiplicativeExpr
	AdditiveExprNoFunc - MultiplicativeExpr
MultiplicativeExpr	UnaryExpr
	MultiplicativeExpr * UnaryExpr
	MultiplicativeExpr \ UnaryExpr
	MultiplicativeExpr % UnaryExpr
MultiplicativeExprNoFunc	UnaryExprNoFunc
	MultiplicativeExprNoFunc * UnaryExpr
	MultiplicativeExprNoFunc \ UnaryExpr
	MultiplicativeExprNoFunc % UnaryExpr
UnaryExpr	PostfixExpr
	DELETE UnaryExpr
	VOID UnaryExpr
	TYPEOF UnaryExpr
	++ UnaryExpr
	-- UnaryExpr
	+ UnaryExpr
	- UnaryExpr
	~ UnaryExpr
	! UnaryExpr

UnaryExpr	PostfixExprNoFunc
	DELETE UnaryExpr
	VOID UnaryExpr
	TYPEOF UnaryExpr
	++ UnaryExpr
	-- UnaryExpr
	+ UnaryExpr
	- UnaryExpr
	~ UnaryExpr
	! UnaryExpr
PostfixExpr	LeftHandSideExpr
	LeftHandSideExpr ++
	LeftHandSideExpr --
PostfixExprNoFunc	LeftHandSideExprNoFunc
	LeftHandSideExprNoFunc ++
	LeftHandSideExprNoFunc --
LeftHandSideExpr	NewExpr
	CallExpr
LeftHandSideExprNoFunc	NewExprNoFunc
	CallExprNoFunc
NewExpr	MemberExpr
	NEW NewExpr
NewExprNoFunc	MemberExprNoFunc
	NEW NewExpr
CallExpr	MemberExpr Arguments
	CallExpr Arguments
	CallExpr [Expr]
	CallExpr . IDENTIFIER
CallExprNoFunc	MemberExprNoFunc Arguments
	CallExprNoFunc Arguments
	CallExprNoFunc [Expr]
	CallExprNoFunc . IDENTIFIER
Arguments	( )
	(ArgumentList)
ArgumentList	AssigExpr

	ArgumentList , AssigExpr
MemberExpr	PrimaryExpr
	FunctionExpr
	MemberExpr [Expr]
	MemberExpr . IDENTIFIER
	NEW MemberExpr Arguments
MemberExprNoFunc	PrimaryExprNoObj
	MemberExprNoFunc [Expr]
	MemberExprNoFunc . IDENTIFIER
	NEW MemberExpr Arguments
PrimaryExpr	THIS
	IDENTIFIER
	Literal
	ArrayLiteral
	ObjectLiteral
	REGULAR_EXPR
	(Expr)
PrimaryExprNoObj	THIS
	IDENTIFIER
	Literal
	ArrayLiteral
	REGULAR_EXPR
	(Expr)
Literal	NULL
	BooleanLiteral
	NUMERIC_LITERAL
	STRING_LITERAL
BooleanLiteral	TRUE
	FALSE
ArrayLiteral	[Elision_OPT]
	[ElementList]
	[ElementList , Elision_OPT]
Elision	,
	Elision ,
ElementList	Elision_OPT AssigExpr

	ElementList , Elision_OPT AssigExpr
ObjectLiteral	{ }
	{ PropNameAndValueList }
PropNameAndValueList	PropName : AssigExpr
	PropNameAndValueList , PropName AssigExpr
PropName	IDENTIFIER
	STRING_LITERAL
	NUMERIC_LITERAL
Expr	AssigExpr
	Expr , AssigExpr
ExprNoFunc	AssigExprNoFunc
	ExprNoFunc DASH AssigExpr
ExprNoIn_OPT	ExprNoIn
	ε (eps.pravidlo)
ExprNoIn	AssigExprNoIn
	ExprNoIn , AssigExprNoIn
AssigOperator	=
	*
	\
	%
	+
	-
	<<
	>>
	>>>
	&=
	^=
	|=
Block	{ }
	{ StatementList }
FunctionExpr	FUNCTION IDENTIFIER (FormalParameterList_OPT) { FunctionBody }
	FUNCTION (FormalParameterList_OPT) { FunctionBody }
FunctionDecl	FUNCTION IDENTIFIER (FormalParameterList_OPT) { FunctionBody }

	FUNCTION (FormalParameterList_OPT) { FunctionBody }
FormalParameterList	IDENTIFIER
	FormalParameterList , IDENTIFIER