Vol.10 No. 2 OKTOBER 2014

ISSN 1907-0837

JURNAL ILMIAH

RealTech

Teknik Informatika | Teknik Industri | Teknik Elektro

Fakultas Teknik
Universitas Katolik De La Salle Manado

Realtech	Vol. 10	No. 2	Hal 1-99	Manado Oktober 2014	ISSN 1907-0837

ISSN 1907-0837

Jurnal Realtech

Volume 10 Nomor 2 Oktober 2014

Pelindung :

Rektor

Unika De La Salle Manado

Penasehat :

PembantuRektor

Unika De La Salle Manado

Penanggung Jawab :

Dekan Fakultas Teknik Unika De La Salle Manado

Sidang Penyunting :

Dr. Ir. Rila Mandala, M.Eng. (ITB)

Ir. RinaldiMunir, MT. (ITB)

Ir. NoldiWatuna, MM.

Debby Paseru, ST., MMSI., M.Ed.

Rubby Padang, SKom.

Gerald Rawis, ST., MM.

PrudensyFebreine, ST.

Ronald Rachmadi, ST., MT.

LianlyRompis, ST.

Alamat Sekretariat / Redaksi :

Sekretariat Jurnal Realtech

Fakultas Teknik

Universitas Katolik De La Salle Manado

Kairagi I Kombos Manado 95000

Telp. 0431-877512, 871971, 871957

E-mail: realtech_dlsu@yahoo.com

Jurnal Realtech merupakan jurnal ilmiah sebagai bentuk pengabdian dalam hal pengembangan bidang Teknologi Informasi, Teknik Elektro dan Teknik Industri dan bidang terkait lainnya.

Jurnal Realtech diterbitkan oleh Fakultas Teknik Universitas Katolik De La Salle Manado. Redaksi mengundang para profesional dari dunia usaha, pendidikan dan peneliti untuk menulis mengenai perkembangan ilmu di bidang yang berkaitan dengan Teknologi Informasi, Teknik Elektro dan Teknik Industri.

Jurnal Realtech diterbitkan 2 (dua) kali dalam 1 tahun pada bulan April dan Oktober. Edisi pertama terbit Juli 2005. Harga berlangganan Rp. 25.000,-/eksemplar dan Rp. 35.000,-/eksemplar (untuk luar Pulau Sulawesi).

ISSN 1907-0837

JurnalRealtech

Volume 10 Nomor 2 Oktober 2014

Daftar Isi Kumulatif

Volume 10 Nomor 2

PERANCANGAN DAN IMPLEMENTASI BAHASA PEMROGRAMAN "PANIKI"

Patrix Rembang[1], Debby Paseru[2], Gerald Rawis[3]
Program Studi Teknik Informatika - Universitas Katolik De La Salle Manado[1,2,3]
Email : patrixcr@gmail.com[1], dpaseru@unikadelasalle.ac.id[2], geraldrawis@yahoo.com[3]

Abstract

The programming language is a technology used to control the computer. Most existing programming languages have a keyword in English because it was created by programmers outside Indonesia. Many programming languages are used in education also have syntax and semantics are relatively complex, and rooted in procedural programming paradigm. In addition, sometimes some people are also reluctant to try to learn a programming language because they sometimes have to do the installation various software rather complicated.

To that end, the author designed and made the implementation of a programming language that has the keyword in Indonesian language, has a simple syntax and semantics, and adopting a functional programming paradigm, to be used for education and to introduce the functional programming paradigm. The author also makes REPL is a web-based software to allow users to try out the programming language via a web browser.

In this research, the author uses the system development methodology Rational Unified Process (RUP) and Unified Modelling Language (UML) as a graphical notation to describe the system being designed. HTML and Javascript used to implement REPL and a programming language interpreter.

Keywords: Programming Language, Functional Programming

Abstrak

Bahasa pemrograman adalah teknologi yang digunakan untuk mengontrol komputer. Kebanyakan bahasa pemrograman yang ada memiliki *keyword* bahasa Inggris karena diciptakan oleh *programmer* di luar Indonesia. Banyak bahasa pemrograman yang digunakan dalam edukasi juga memiliki sintaks dan semantik yang relatif kompleks, dan berakar pada paradigma *procedural programming*. Selain itu, kadang sebagian orang juga enggan mencoba mempelajari sebuah bahasa pemrograman karena mereka kadang harus melakukan proses instalasi berbagai perangkat lunak yang agak rumit.

Untuk itu, penulis merancang dan membuat implementasi sebuah bahasa pemrograman yang memiliki *keyword* berbahasa Indonesia, memiliki sintaks dan semantik yang sederhana, dan mengadopsi paradigma *functional programming*, agar bisa digunakan untuk edukasi dan untuk memperkenalkan paradigma pemrograman *functional*. Penulis juga membuat REPL yaitu sebuah perangkat lunak berbasis *web* agar pengguna bisa mencoba bahasa pemrograman ini melalui *web browser*.

Dalam penelitian ini, penulis menggunakan metodologi pengembangan sistem *Rational Unified Process* (RUP) dan *Unified Modelling Language* (UML) sebagai notasi grafis untuk menggambarkan sistem yang akan dibuat. HTML dan Javascript digunakan untuk mengimplementasikan REPL dan *interpreter* bahasa pemrograman.

Kata kunci : Bahasa pemrograman, *Functional Programming*

1. PENDAHULUAN

Kemajuan teknologi informasi membuat semakin banyak pekerjaan manusia melibatkan komputer. Hal ini tentunya membuka peluang kerja bagi para *programmer*. Bahasa pemrograman adalah sebuah bahasa yang bisa digunakan oleh manusia untuk mengendalikan komputer. Dengan bahasa pemrograman, *programmer* bisa membuat perangkat lunak mulai dari *game*, aplikasi *word processor*, sistem operasi, aplikasi *web*, dan banyak lagi.

Namun, kebanyakan bahasa pemrograman yang digunakan dalam edukasi memiliki sintaks dan semantik yang terlalu kompleks. Misalnya, dalam bahasa pemrograman C, untuk melakukan operasi aritmatika dasar seperti penambahan saja, *programmer* harus hati-hati dalam memilih tipe data dari variabel yang menjadi *operand* pada

operator penambahan tersebut, karena itu bisa menghasilkan hasil yang tidak diinginkan. Salah satu contohnya adalah program C berikut:

```
#include <stdio.h>
int main()
{
    int x = -1;
    unsigned int y = 2;
    if (x < y) {
        printf("x kurang dari y\n");
    } else {
        printf("x lebih dari y\n");
    }
    return 0;
}
```

Kode program di atas akan mengeluarkan *string* "x lebih dari y", jika di-*compile* dengan *compiler* yang mengimplementasikan standar ANSI C, karena aturan konversi aritmatik dalam standar ANSI C.

Tujuan penelitian ini adalah membuat bahasa pemrograman untuk edukasi yang memiliki sintaks dan semantik sederhana sehingga bisa lebih mudah dimengerti oleh orang-orang yang baru mempelajari pemrograman komputer, serta membuat REPL berbasis *web* sehingga pengguna bisa mencoba bahasa pemrograman ini secara langsung tanpa harus menginstalasi perangkat lunak tambahan.

Adapun batasan masalah dalam penelitian ini antara lain:

1. *Standard library* yang akan dibuat hanya akan mencakup fungsi aritmatika, logika, perbandingan dan manipulasi struktur data dasar. Struktur data dasar yang dimaksud adalah *list* dan *hash table.*
2. *Command-line tool* yang berupa *interpreter* dan REPL hanya bisa dijalankan di Linux.
3. Tipe data yang akan dibuat hanyalah Angka, Nilai logika, Deretan karakter, Larik, dan Larik Asosiatif.

2. STUDI PUSTAKA

2.1. Bahasa Pemrograman

Bahasa pemrograman adalah bahasa yang digunakan untuk mengontrol komputer. Bahasa pemrograman juga dapat diartikan sebagai notasi untuk mendeskripsikan algoritma dan struktur data [1]. Bahasa pemrograman terdiri dari sintaks dan semantik. Sintaks dari suatu bahasa pemrograman berarti struktur atau aturan penulisan program dalam bahasa tersebut. Sedangkan semantik merujuk kepada arti dari deretan karakter yang legal dari suatu bahasa pemrograman. Bahasa pemrograman yang dapat melakukan semua komputasi yang dapat dilakukan, yaitu dapat mensimulasikan mesin Turing universal, disebut *Turing complete* [2]. Mesin Turing adalah sebuah model komputasi yang dapat mensimulasikan komputer. Mesin Turing terdiri dari *finite control* yang memiliki *state,* dan pita yang dapat ditulisi simbol. Kemampuan komputer dan mesin Turing adalah sama, yaitu hanya dapat menerima bahasa yang *recursively enumberable* [9].

Pada awal munculnya komputer yang bisa diprogram pada 1940-an, bahasa pemrograman yang ada berupa kode-kode mesin. *Operator* komputer membolak-balikkan saklar untuk menyimpan kode ini ke memori. Pemrograman komputer dengan cara ini sangat rentan terhadap kesalahan. Setelah itu, muncul bahasa *assembly,* yang lebih dapat dipahami manusia dibandingkan dengan bahasa mesin. Bahasa pemrograman ini menggunakan beberapa simbol *mnemonic* untuk mewakili instruksi dan lokasi memori. *Assembler* adalah program yang dapat mengubah bahasa *assembly* ke bahasa mesin. Bahasa *assembly* juga tidak *portable,* karena *hardware* yang memiliki set instruksi yang berbeda akan memiliki dialek bahasa *assembly* yang berbeda pula. Pada tahun 1950-an, muncullah bahasa pemrograman FORTRAN yang dibuat oleh John Backus, yang lebih *high-level* daripada *assembly,* untuk mempermudah kinerja *programmer* [4].

Bahasa pemrograman biasanya digolongkan berdasarkan paradigma yang didukung oleh fitur-fitur yang ada dalam bahasa pemrograman tersebut, walaupun pengelompokkan bahasa pemrograman berdasarkan paradigma ini tidak disetujui oleh semua akademisi [5]. Contoh-contoh paradigma yang ada adalah [6]:

1. *Functional.* Bahasa pemrograman *functional* membuat model komputasi berdasarkan definisi rekursif dari fungsi-fungsi. Bahasa pemrograman ini terinspirasi dari λ-calculus, sebuah model komputasi formal yang dikembangkan Alonzo Church pada tahun 1930-an. Program dianggap sebagai fungsi dari masukan ke keluaran, yang didefinisikan dari fungsi yang lebih sederhana. Bahasa pemrograman yang biasanya digolongkan dalam paradigma ini adalah LISP, ML, dan Haskell. Beberapa fitur yang dimiliki bahasa pemrograman yang dikategorikan *functional* adalah *higher-order function* [7], *first-class*

function, anonymous function, closure, dan pengurangan pengubahan *state* dalam program (berbeda dengan bahasa pemrograman berparadigma prosedural/*Von Neumann*). Karena itu biasanya bahasa pemrograman *functional* tidak memiliki konstruksi iterasi seperti *for* dan *while loop*, karena basis kedua konstruksi tersebut adalah mengubah *state* dari suatu variabel. Iterasi biasanya dilakukan lewat rekursi atau menggunakan *higher-order function* seperti *map, filter,* dan *fold.* Bahasa pemrograman seperti Haskell bahkan sampai menghilangkan pengubahan *state* seluruhnya [8].

2. *Logic.* Bahasa pemrograman *logic* terinspirasi dari logika predikat. Bahasa pemrograman ini memodelkan komputasi sebagai suatu uji coba untuk menemukan sekumpulan nilai yang memenuhi relasi-relasi tertentu yang telah dispesifikasikan, menggunakan pencarian melalui daftar aturan logika. Prolog adalah bahasa pemrograman *logic* yang paling terkenal.
3. *Object-oriented.* Bahasa pemrograman *object-oriented* adalah bahasa pemrograman yang berakar dari Simula 67. Daripada memodelkan komputasi sebagai operasi prosesor monolitik pada memori yang monolitik, bahasa pemrograman yang digolongkan dalam jenis ini biasanya menggambarkan komputasi sebagai interaksi objek-objek semiindependen, di mana setiap objek memiliki *state* sendiri dan subrutin yang mengatur state tersebut. Smalltalk adalah contoh bahasa pemrograman *object-oriented* yang murni.
4. *Von Neumann/Procedural.* Inti bahasa pemrograman ini adalah modifikasi variabel. Berbeda dengan bahasa pemrograman *functional* yang berbasis pada ekspresi yang memiliki nilai, bahasa pemrograman berparadigma ini berbasis pernyataan yang mempengaruhi komputasi selanjutnya melalui *side effect* dari pengubahan nilai di memori. Contoh bahasa pemrograman yang dikategorikan ke dalam paradigma ini adalah C, Ada, Fortran.

Selain berdasarkan paradigma, bahasa pemrograman juga sering dikelompokkan berdasarkan fitur-fitur dan semantik di dalamnya. Pengelompokkan yang paling umum adalah *static typing* dan *dynamic typing, eager evaluation* dan *lazy evaluation, lexical scoping* dan *dynamic scoping.*

Static typing adalah fitur di mana pengecekan tipe variabel dilakukan pada saat waktu kompilasi (*compile-time*). Contoh bahasa pemrograman yang menggunakan *static typing* adalah PL/SQL, Ada, C, dan Pascal. Pada *dynamic typing,* pengecekan tipe variabel dilakukan pada saat eksekusi program (*runtime).* Contoh bahasa pemrograman yang menggunakan *dynamic typing* adalah Javascript, Perl, dan Ruby (Feuerstein and Priby 2009). *Static* dan *dynamic typing* ini adalah salah satu fitur yang paling sering diperdebatkan oleh para *programmer.* Pendukung *static typing* berargumen bahwa keuntungan *static typing* adalah deteksi awal kesalahan pemrograman, dokumentasi program yang lebih baik dalam bentuk anotasi tipe, lebih banyak kemungkinan untuk dilakukan optimisasi oleh *compiler,* dan meningkatnya efisiensi *runtime.* Sedangkan pendukung *dynamic typing* berargumen bahwa *static typing* terlalu kaku, dan bahasa pemrograman dengan *dynamic typing* sangat ideal untuk digunakan sebagai alat untuk membuat sistem dengan *requirement* yang sering berubah atau berinteraksi dengan sistem yang sering berubah tanpa bisa diprediksi. *Dynamic typing* juga memungkinkan fitur dinamis program seperti *dynamic loading, runtime reflection,* dan lain-lain [9].

Eager evaluation merujuk kepada evaluasi argumen kepada suatu fungsi sebelum argumen tersebut disubtitusikan ke dalam suatu fungsi, sedangkan *lazy evaluation* adalah penundaan evaluasi argumen dalam suatu fungsi [10]. Contoh implementasi bahasa pemrograman yang menggunakan *eager evaluation* adalah JVM untuk Java, GCC untuk C, MRI untuk Ruby, V8 dan SpiderMonkey untuk Javascript, CPython untuk Python, dan masih banyak lagi. Sedangkan implementasi bahasa pemrograman yang menggunakan *lazy evaluation* adalah GHC untuk Haskell.

Scope adalah istilah yang berarti ruang lingkup suatu nama variabel. *Scope* memetakan antara himpunan nama ke himpunan nilai dan prosedur dari suatu himpunan pernyataan di dalam program. *Lexical scoping* berarti dalam suatu *scope,* setiap nama merujuk kepada deklarasi leksikal terdekat. Artinya, jika ada nama variabel **s** yang digunakan dalam *scope* tertentu, **s** merujuk kepada **s** yang dideklarasikan dalam *scope* tersebut,

jika ada. Jika tidak, **s** merujuk kepada deklarasi dari **s** yang berada pada *scope* luar yang paling dekat. Sedangkan dalam *dynamic scoping*, variabel bebas merujuk kepada variabel yang sama dengan variabel bebas tersebut yang paing terakhir diciptakan pada saat *runtime* [11].

Bahasa pemrograman biasanya diimplementasikan menggunakan *compiler, interpreter,* atau *Just-In-Time compiler.* Implementasi bahasa pemrograman dalam bentuk *compiler* biasanya memiliki tahap *lexical analysis, syntactic analysis, semantic analysis, optimization,* dan *code generation* [12], sedangkan *interpreter* biasanya memiliki tahap *lexical analysis, syntactic analysis,* dan *execution/evaluation.*

Lexical analysis adalah fase di mana sebuah komponen implementasi bahasa pemrograman yang bernama *lexical analyzer* membaca serangkaian karakter yang merupakan bagian dari program menjadi rangkaian yang berarti yang disebut *lexeme. Lexical analyzer* atau yang biasa disebut *lexer* ini menghasilkan *token* sebagai *output.*

Syntactic analysis atau *Parsing* adalah fase di mana *token* yang dihasilkan *lexer* diubah oleh *syntactic analyzer* atau juga sering disebut *parser* menjadi struktur berbentuk pohon yang melambangkan struktur gramatikal dari rangkaian *token* tersebut. Struktur pohon ini disebut *parse tree.* Pembuatan *parse tree* ini dilakukan dengan cara mencocokkan *token* yang diterima *parser* dengan *grammar* yang bisa diterima oleh *parser* tersebut.

Teknik *parsing* sering dibagi ke dalam 2 kategori, yaitu *top-down parsing* dan *bottom-up parsing. Top-down parsing* adalah teknik di mana *parse tree* dibuat mulai dari atas ("akar") ke bawah ("daun"), *depth-first.* Teknik ini juga bisa dipandang sebagai penurunan *input string* dari kiri (*Leftmost derivation).* Sedangkan *bottom-up parsing* adalah teknik di mana *parse tree* dibuat mulai dari bawah ("daun") ke atas ("akar").

Untuk mendeskripsikan sintaks bahasa pemrograman yang biasanya *context free*, maka secara formal sering digunakan *Context-free Grammar* (CFG). Sayangya, CFG bisa menghasilkan *parse tree* yang ambigu karena sifat CFG yang nondeterministik. Akibatnya, *parser* tidak bisa benar-benar menerima bahasa yang dideskripsikan oleh CFG, tetapi hanya subset dari CFG. Karena itu, ada juga *grammar* lain seperti *Parsing Expression Grammar* (PEG). PEG mirip dengan CFG tetapi memberikan beberapa batasan agar tidak mungkin tercipta *parse tree* yang ambigu. Tetapi PEG juga memiliki kelemahan seperti tidak bisa mendeskripsikan aturan produksi yang *left-recursive* [13].

Semantic analysis adalah fase di mana komponen yang bernama *semantic analyzer* melakukan pengecekan semantik dari program. Bagian penting dari *semantic analysis* adalah pengecekan tipe data variabel, di mana *compiler* melakukan pengecekan apakah tiap operator mendapatkan *operand* yang sesuai.

Optimization adalah fase di mana representasi program dioptimisasi. Biasanya, jika *target platform* dari suatu *compiler* adalah bahasa *assembly*, sebelum fase *optimization* *syntax tree* diubah menjadi kode yang mirip kode *assembly.* Kode ini lalu dioptimisasi oleh fase *optimization* ini.

Code generation adalah fase di mana representasi program yang telah melalui fase-fase sebelumnya diubah menjadi kode bagi *target platform* dari *compiler* tersebut. Kode ini bisa berupa *machine code, assembly,* atau bahasa lain yang lebih *high-level* [14].

Evaluation atau *execution* adalah fase di mana *interpreter* mengeksekusi representasi program yang biasanya dalam bentuk *syntax tree* [15].

2.2. *Read-Evaluate-Print-Loop* (REPL)

Read-Evaluate-Print-Loop atau REPL adalah perangkat lunak yang menerima masukan dari pengguna berupa program, mengevaluasi atau mengeksekusi program tersebut, lalu mencetak hasil evaluasi tersebut, kemudian mengulangi proses ini terus menerus [16]. Untuk membuat REPL ini, penulis menggunakan teknologi *web* agar pengguna bisa mencoba bahasa pemrograman ini lewat *web browser* secara langsung.

Untuk itu, penulis menggunakan *HyperText Markup Language,* sebuah bahasa *markup* yang pada awalnya digunakan untuk mendeskripsikan *content* dalam bentuk teks secara elektronik [17]; *Cascading StyleSheet,* yaitu bahasa *stylesheet* yang digunakan untuk mengubah gaya tampilan visual HTML [18]; dan Javascript, bahasa pemrograman yang dapat dieksekusi oleh berbagai *web browser*, dan digunakan di situs-situs *web* modern [19].

3. ANALISIS DAN PERANCANGAN

3.1. Target Pengguna

Penulis bertujuan menjadikan calon-calon *programmer* Indonesia yang baru mau mempelajari pemrograman, maupun *programmer* yang baru mau mempelajari paradigma pemrograman fungsional sebagai target pengguna.

Preliminary Project Requirement

1. Persyaratan Pengembangan
   - Menggunakan metodologi pengembangan RUP.
   - Menggunakan UML untuk pemodelan sistem.

2. Persyaratan Fungsional
   - Bagian sitem yang bertugas menjadi *interpreter* dapat mengeksekusi program yang ditulis dalam bahasa pemrograman yang dibuat penulis
   - Bagian sistem yang menjadi REPL dapat menerima *input* dari pengguna yang berupa program yang ditulis dalam bahasa pemrograman yang dibuat penulis, dan menampilkan hasil eksekusi program tersebut kembali kepada pengguna.

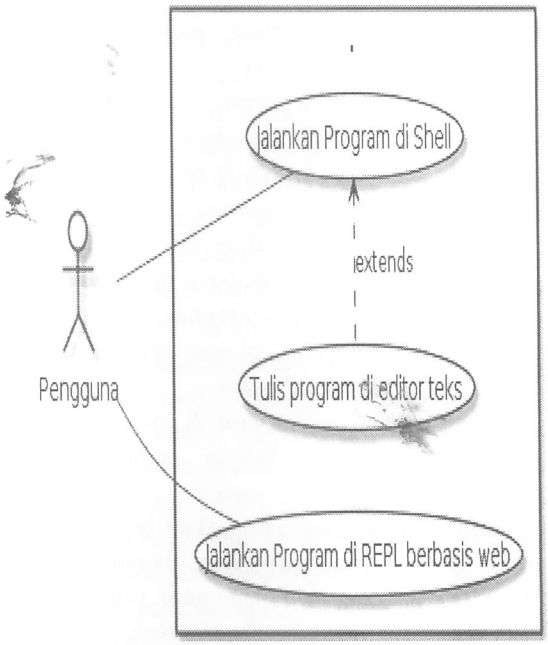
3. Persyaratan Non Fungsional
   - Operational Requirements
     - REPL berbasis *web* yang dibuat penulis bisa diakses melalui *web browser*
     - *Command line tool* yang akan dijalankan di *shell* dari sistem operasi dapat dijalankan secara *stand alone* di *distro* Linux modern yang dapat menjalankan Node.js.
   - *Performance Requirements*
     - Sistem harus dapat menanggapi *input* program yang dibuat oleh pengguna (yang tidak memiliki instruksi iterasi dan cukup kecil) di bawah 3 detik.
   - *Cultural or Political Requirements*
     - Sistem harus menggunakan Bahasa Indonesia

3.2. Perancangan

Deskripsi sistem yang akan dibangun adalah sebagai berikut :

1. Functional Modelling

Pemodelan fungsional digambarkan dengan *Use Case Diagram*. Use Case mendeskripsikan semua fungsionalitas yang bisa digunakan oleh pengguna yang akan dibangun dalam perangkat lunak.

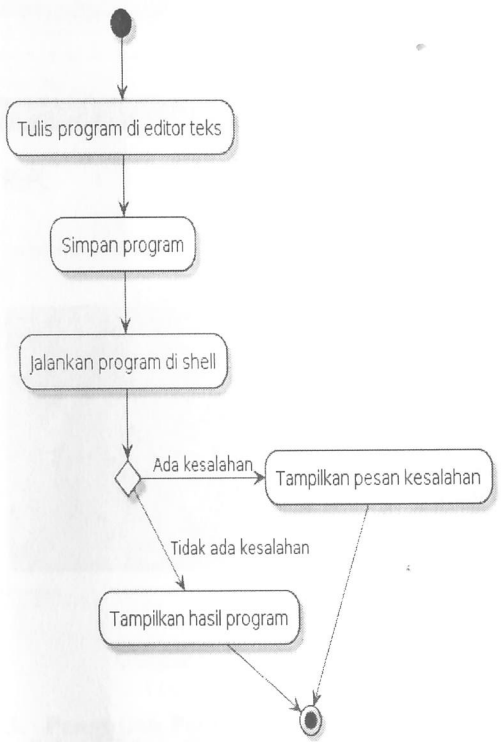


Gambar 1. *Use Case Diagram*

2. Behavioural Modelling

Behaviour modelling dideskripsikan lewat *Activity Diagram*. Diagram ini menggambarkan alur penggunaan perangkat lunak yang akan dibuat.

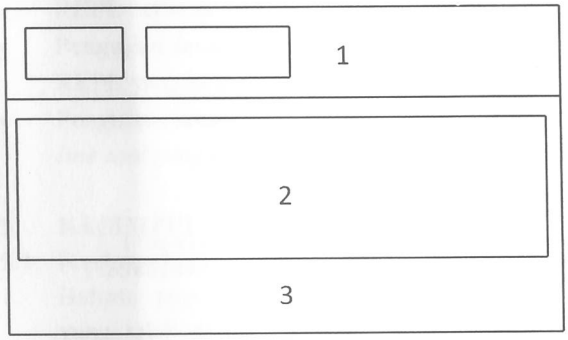
Gambar 2. *Activity Diagram* Jalankan program di REPL



Gambar 3. *Activity Diagram* Jalankan program di *shell*

3.3. Perancangan Antarmuka

Antarmuka grafis yang akan digunakan pengguna adalah REPL berbasis *web*. Rancangan tampilan awalnya adalah sebagai berikut:



Gambar 4. Rancangan antarmuka REPL berbasis *web*

Keterangan:

1. *Navigation Bar*. Tempat di mana navigasi perangkat lunak ini berada
2. REPL. Tempat di mana pengguna bisa mencoba bahasa pemrograman.
3. Penjelasan tentang aspek-aspek bahasa pemrograman.

Selain antarmuka berbasis *web* yang dibuat untuk menampilkan REPL, penulis juga merancang sintaks dari bahasa pemrograman yang telah dibuat. Sintaks ini dapat dipandang sebagai antarmuka *programmer* yang akan menggunakan bahasa pemrograman ini. Sintaks bahasa pemrograman tersebut dispesifikasikan sebagai berikut, menggunakan notasi *Parsing Expression Grammar* (PEG):

```
program <- mlws statements
statements <- statement* expr? ws
ws <- ign*
mlws <- mlign*
ign <- [ \t] / comment
mlign <- [ \t\n\r] / comment
comment <- ";" [^;]* ";"
statement <- = expr ws "\n" mlws
expr <- def / funApp / id / cond / func /
number / boolean / string / list / hash /
block
id <- = [a-zA-Z0-9_!@#`~$%^&*=+|<.?/-]+
string <- = "\"" "\"" / "\""
(!unescapedquote .)* unescapedquote
unescapedquote <- [^\\] "\""
boolean <- "Benar" / "Salah"
def <- id ws ":=" mlws expr
func <- "fungsi" mlws "(" mlws idList mlws
")" mlws expr
idList <- id restId*
restId <- mlws id
number <- "-"? [0-9]+ frac?
frac <- "," [0-9]+
block <- "{" mlws statements "}"
funApp <- id ws "(" mlws exprList mlws ")"
/ func ws "(" mlws aexprList mlws ")"
exprList <- expr restExpr*
restExpr <- mlws expr
list <- "[" mlws "]" / "[" mlws exprList
mlws "]"
cond <- "jika" mlws expr mlws "maka" mlws
expr mlws "atau" mlws expr
hash <- "{#" mlws hashContent* mlws "}"
hashContent <- id mlws ":" mlws expr mlws
```

Keterangan:

1. program adalah awal program
2. expr adalah ekspresi
3. comment adalah komentar
4. id adalah simbol

4. IMPLEMENTASI

4.1. Kode Program

Bahasa pemrograman yang digunakan untuk mengimplementasikan perangkat lunak ini adalah Javascript. Berikut ini adalah salah satu kode program yang berfungsi menerima *Abstract Syntax Tree* (AST) dan mengevaluasi nilai yang dihasilkan (eval.js):

```
var Evaluator = (function() {
var is = function (obj, type) {
```

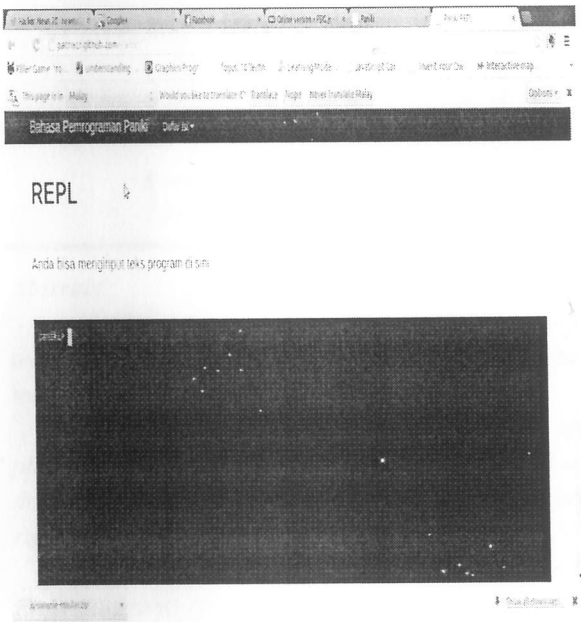
```
var clas =
Object.prototype.toString.call(obj).slice(8
, -1);
return obj !== undefined && obj !== null &&
clas === type;
};

  var lookup = function (id, env) {
if (!(env.hasOwnProperty('bindings')))
throw new Error(id + " tidak
ditemukan");
if (env.bindings.hasOwnProperty(id)) return
env.bindings[id];
return lookup(id, env.outer);
};

var bind = function (id, val, env) {
if (!(env.hasOwnProperty('bindings'))) {
env.bindings = {};
env.outer = {};
}
if (env.bindings.hasOwnProperty(id)) throw
new Error(id+"telah
didefinisikan sebelumnya");
env.bindings[id] = val;
};
var eval = function (expr, env) {
if (is(expr, 'Array')) {
var res = 0;
for (var i = 0, j = expr.length; i < j;
i++) {
res = eval(expr[i], env);
}
return res;
}
if (is(expr, 'Number')) {
return expr;
}
if (is(expr, 'String')) {
return lookup(expr, env)
}
switch (expr.form) {
case 'bool':
return expr.value;
case 'string':
return expr.value;
case 'list':
var resList = [];
for (var i = 0, j = expr.value.length; i <
j; i++) {
resList[[i]] = eval(expr.value[i], env);
}
return resList;
case 'hash':
var resHash = {};
for (var p in expr.value) {
if (expr.value.hasOwnProperty(p))
resHash[p] = eval(expr.value[p],
env);
}
return resHash;
case 'cond':
var antecedent = eval(expr.antc, env);
if(!is(antecedent, "Boolean"))
throw new Error("Kondisi di dalam Implikasi
harus berupa Nilai
Logika");
return antecedent ? eval(expr.cons, env) :
eval(expr.alt, env);
case 'def':
var val = eval(expr.expr, env);
bind(expr.id, val, env);
return val;
case '->':
if (expr.params.length === 1) {
return function() {
var bnd = {};
bnd[expr.params[0]] = arguments[0];
var newEnv = { bindings: bnd, outer: env };
return eval(expr.body, newEnv);
};
} else {
return function() {
var bnd = {};
bnd[expr.params[0]] = arguments[0];
var newEnv = {bindings: bnd, outer: env};
expr.params.shift();
return eval(expr, newEnv);
}
}
case 'app':
var func = eval(expr.func, env);
var arg;
for (var i = 0, j = expr.args.length; i <
j; i++) {
var arg = eval(expr.args[i], env);
if (!is(func, "Function"))
throw new Error("Jumlah argumen melebihi
jumlah parameter");
func = func.call(undefined, arg);
}
return func;
}
};

return {
eval: eval,
lookup: lookup,
bind: bind
};
})();
if (typeof module !== 'undefined') {
module.exports.Evaluator = Evaluator;
}
```

4.2. Hasil implementasi Aplikasi

Hasil yang didapat dalam implementasi perangkat lunak ini adalah:

- REPL berbasis *web*.
- REPL yang bisa dijalankan di *shell* Linux.
- *Interpreter* dalam bentuk *command-line tool* yang bisa dijalankan di Linux.

Di bawah ini tampilan REPL berbasis *web*:



Gambar 5. REPL berbasis *web*

4.3. Pengujian Perangkat Lunak

Pengujian dilakukan untuk mengetahui apakah perangkat lunak sudah dibuat sesuai dengan maksud awal pembuatannya. Pengujian yang dilakukan adalah:

- Pengujian *correctness* dari *interpreter* yang telah dibuat melalui *automated unit tests*.
- Pengujian secara manual terhadap antarmuka REPL berbasis *web*
- Pengujian secara manual terhadap antarmuka REPL yang bisa dijalankan di *shell* Linux
- Pengujian secara manual terhadap *command-line tool* yang bisa dijalankan di Linux.

5. KESIMPULAN DAN SARAN

5.1. Kesimpulan

1. Bahasa pemrograman "Paniki" dan REPL yang telah dibuat bisa membantu pengguna tingkat pemula untuk belajar pemrograman yang menggunakan sintaks berbahasa Indonesia.
2. Bahasa Pemrograman "Paniki" dan REPL yang dibuat bisa membantu mengenalkan konsep-konsep *functional programming* kepada pengguna.

5.2. Saran

1. *Standard Library* dapat dikembangkan lagi agar *programmer* bisa berinteraksi dengan *database,* dapat berinteraksi dengan *Document Object Model*, dan lain-lain.
2. Tipe data dapat ditambahkan lagi.
3. Bisa ditambahkan jenis struktur data selain *list* dan *hash table*.

6. DAFTAR PUSTAKA

1. Kedar, S. and Thakare S. (2009). *Principles of Programming Languages*. Pune: Technical Publications.
2. Guttag, J. (2008). *Lecture 2: Core Elements of a Program*. Available at http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-5600sc-introduction-to-computer-science-and-programming-spring-2011/unit-1/lecture-2-core-elements-of-a-program [Accessed 9 October 2012].
3. Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2000). *Introduction to Automata Theory, Languages, and Computation*. (2nd edn). Boston: Addison-Wesley.
4. Louden, K. C. and Lambert, K. A. (2011). *Programming Languages: Principles and Practices*. (3rd edn). Boston: Course Technology.
5. Krishnamurti, S. (2008). *Teaching Programming Languages in a Post-Linnaean Age*. SIGPLAN Workshop on Undergraduate Programming Language Curricula 2008, Tucson, 7-13 Juni 2008. New York: ACM.
6. Scott, M. L. (2009). *Programming Language Pragmatics*. (3rd edn). Burlington: Morgan Kauffman.
7. Crockford, D. (2012). *Monads and Gonads*. Available at http://www.yuiblog.com/blog/2012/12/13/yuiconf-2012-talk-douglascrockford-on-monads-and-gonads-evening-keynote [Accessed 10 December 2012].
8. Stewart, D. (2006). *A brief introduction to Haskell*. Available at http://www.haskell.org/haskellwiki/A_brief_introduction_to_Haskell [Accessed 10 December 2012].
9. Meijer, E. and Drayton, P. (2004). Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages. *Revival of dynamic languages Workshop at OOPSLA 2004*. Vancouver, 24-28 Oktober 2004. New York: Springer.
10. Krishnamurti, S. (2012). *Programming Languages: Application and Interpretation*. (2nd edn). Providence: Shriram Krishnamurti.

11. Cooper, K. and Torczo, L. (2011). *Engineering a Compiler*. (2nd edn). Burlington: Morgan Kauffman.
12. Aiken, A. (2012). *Structure of a Compiler*. Available at https://class.coursera.org/compilers-2012-selfservice/lecture/index [Accessed 9 October 2012].
13. Ford, B. (2004). *Parsing Expression Grammars: A Recognition-Based Syntactic Foundation*. The 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Venice, 14-16 Januari 2004. New York: ACM.
14. Aho, A., Lam, M., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, & Tools*. Boston: Addison-Wesley.
15. Norvig, P. (2010). *(How to Write a (Lisp) Interpreter (in Python))*. Available at http://norvig.com/lispy.html [Accessed 9 October 2012).
16. Siebel, P. (2005). *Practical Common Lisp*. New York: Apress.
17. Francis, M. N. (2008). *The Basics of HTML*. Available at http://dev.opera.com/articles/view/12-the-basics-of-html [Accessed 10 October 2012].
18. Mozilla. (2012). *CSS*. Available at https://developer.mozilla.org/En/CSS [Accessed 10 October 2012].
19. Chugh, R., Meister, J. A., Jhala, R. and Lerner, S. (2009). *Staged Information Flow for Javascript*. Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation, Dublin, 15-20 Juni 2009. New York: ACM.