Jurnal Ilmiah Komputer dan Informatika (KOMPUTA)
Edisi. .. Volume. .., Bulan 20.. ISSN : 2089-9033

KOMPILER UNTUK PEMROGRAMAN DALAM BAHASA INDONESIA

Gun Gun Febrianza Amd.Kom[1], Ken Kinanti Purnamasari S.Kom., M.T[2]

[1]Teknik Informatika – Universitas Komputer Indonesia
Jl. Dipatiukur 112-114 Bandung
E-mail : gungunfebrianza@gmail.com

ABSTRAK

Bahasa pemrograman digunakan manusia agar dapat mengendalikan komputer, namun secara dasar komputer hanya mampu memahami bahasa mesin. Sehingga bahasa pemrograman tingkat tinggi dibuat untuk menyederhanakan pemrograman komputer.

Semakin sederhana maka semakin tinggi abstraksi dan kedetailan yang disembunyikan di dalamnya. Bahasa pemrograman tingkat tinggi masih bisa diklasifikasikan, terdapat pemrograman berbahasa Inggris dan pemrograman berbahasa natif. Pada penelitian ini pemrograman Berbahasa Indonesia dibuat untuk menyederhanakan bahasa pemrograman agar mudah difahami, diingat dan dipelajari oleh mereka yang memahami bahasa natif di Indonesia. Sehingga Kompiler untuk pemrograman berbahasa Indonesia dengan paradigma pemrograman berorientasi objek kami buat.

Paradigma pemrograman berorientasi objek digunakan agar beberapa permasalahan pada paradigma pemrograman sebelumnya juga ikut dipecahkan. Kompiler untuk pemrograman berbahasa Indonesia sendiri menggunakan skema penerjemahan dua fase, menggunakan sebuah mesin eksekusi agar program yang dihasilkan kompiler dapat berjalan diberbagai platform baik itu sistem operasi Linux atau Windows 32 atau 64 bit.**Kata**

kunci : Bahasa Pemrograman Tingkat Tinggi, Pemrograman Berbahasa Indonesia, Kompiler, Pemrograman Berorientasi Objek

1. PENDAHULUAN

Bahasa Pemrograman adalah sebuah notasi untuk menjelaskan sebuah komputasi pada manusia dan juga pada suatu mesin komputer[1]. Ada berbagai macam bahasa pemograman yang bisa diklasifikasikan sebagian diantaranya dapat diklasifikasikan berdasarkan abstraksi mulai dari bahasa tingkat rendah sampai tingkat tinggi, berdasarkan penerjemah terdapat bahasa pemrograman untuk interpreter dan bahasa pemrograman untuk kompiler, dari segi bahasa terdapat pemrograman berbahasa Inggris dan pemrograman berbahasa natif, berdasarkan paradigma terdapat pemrograman berorientasi objek, pemrograman imperatif, pemrograman deklaratif dan sebagainya.

Untuk bisa membuat kompiler yang mampu menerjemahkan bahasa pemrograman berbahasa Indonesia dengan gaya pemrograman berorientasi objek, diperlukan tehnik analisis leksikal, analisis sintak, analisis semantik dan penghasil kode secara khusus yang bisa digunakan untuk menghadapi spesifikasi tersebut.

Sehingga pada penelitian ini kami mengembangkan kompiler untuk pemrograman berbahasa Indonesia yang diberi nama PBI (Pemrograman berbahasa Indonesia) sebagai bahasa pemrograman untuk *general-purpose*. Alasan kuat lainya pembuatan bahasa Indonesia sebagai bahasa pemrograman dikarenakan Indonesia merupakan negara dengan jumlah penduduk terbesar ke empat di dunia. Sehingga memiliki potensi yang cukup besar untuk mendapatkan pengembangan, kontribusi dan pemanfaatan pemrograman berbahasa Indonesia untuk berbagai penelitian.

Penelitian ini dibagi menjadi beberapa bagian, pada bagian ke 2 dijelaskan latar belakang penelitian, beberapa penelitian tentang pemrograman berbahasa natif diberbagai belahan dunia dan perkembangan pemrograman berbahasa natif di Indonesia sendiri yang menjadi dorongan kuat penelitian ini dibuat. Pada bagian ke 3 dijelaskan proses kompilasi yang dilakukan pada PBI, pada bagian ke 4, metode untuk melakukan analisis leksikal, pada bagian ke 5 dijelaskan metode untuk melakukan analisis sintak dan optimasi, pada bagian ke 6 dijelaskan metode untuk melakukan analisis semantik dan penghasil kode, pada bagian ke 7 dijelaskan mesin eksekusi yang digunakan dan pada bagian ke 8 terdapat kesimpulan untuk penelitian selanjutnya.

2. ISI PENELITIAN

2.1 Latar Belakang

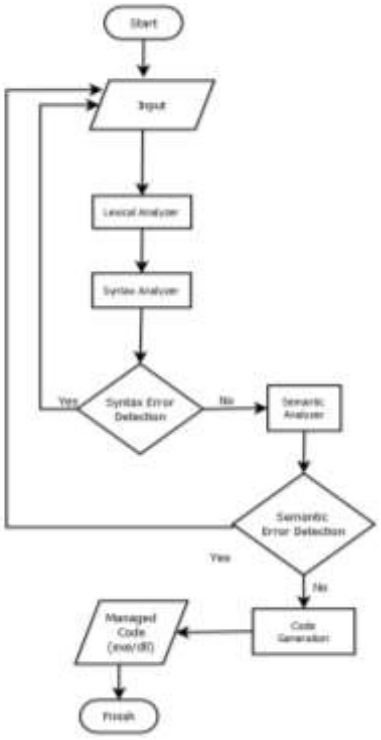
Seluruh bahasa pemrograman yang mengglobal didominasi menggunakan bahasa Inggris karena faktor sejarah dan linguistik. Dari segi faktor sejarah bahasa Inggris dalam dunia komputer adalah *lingua franca* (bahasa umum) dikarenakan karakter *encoding* pertama kali didesain untuk bahasa inggris yaitu ASCII pada tahun 1960[2]. Dari segi linguistik bahasa Inggris memiliki karakter yang singkat dan tidak memiliki *diacritical marks* (aksen karakter).

Meskipun begitu, *non-english based programming language* tetap eksis (*Lexico* dalam bahasa Spanyol, *Aheui* dalam bahasa Korea, *Sako* dalam bahasa Polandia dan *Rapira* dalam bahasa Rusia[3]). Di Indonesia penelitian penggunaan bahasa Indonesia sebagai bahasa pemrograman telah dilakukan. Diantaranya adalah bahasa KILANG yang ada dalam buku mengenai pembuatan Interpreter berbahasa Indonesia yang berjudul KILANG 002: BASIC dalam Bahasa Indonesia yang terbit pada tahun 1984[5], kemudian bahasa ALIN untuk ekspresi Aljabar Linier dalam buku Tuntunan Praktis Pemrograman Merekayasa Interpreter yang terbit pada tahun 1995[6], dan BAIK (Bahasa Anak Indonesia untuk Komputer) yang dirilis pertama kali pada tahun 2008[7], untuk menerjemahkan dari bahasa tingkat tinggi saat ini ketiganya masih menggunakan Interpreter.

Sehingga saat ini di Indonesia belum ada kompiler yang secara khusus digunakan untuk menerjemahkan pemrograman berbahasa Indonesia. Kompiler yang mampu menerjemahkan pemrograman berbahasa Indonesia untuk bisa memproduksi *program* dan *library* yang bisa digunakan secara simultan. Sebuah *executable* digunakan agar program yang dibuat dengan sebuah bahasa bisa langsung dieksekusi komputer dan *Dynamic Link Library* agar *library* yang dibuat dengan sebuah bahasa *class & method* di dalamnya bisa dimanfaatkan oleh program lainya. Selanjutnya keluaran *executable* atau *dynamic link library* bisa diproduksi dan digunakan pada multisistem operasi tanpa harus menggunakan lagi penerjemah.

2.2 Proses Kompilasi

Saat kompilasi dilakukan ada tahap yang harus dilalui yang akan dilakukan oleh beberapa program di dalam kompiler mulai dari program *Lexical Analyzer* untuk memproduksi *token*, program *Syntax Analyzer* untuk melakukan *parsing* pada setiap *token*, selanjutnya proses *Tree Traversal* dilakukan oleh program *Semantic Analyzer* untuk menjelajahi representasi *data tree* atau *list data* hasil dari *parsing* untuk membangun *symbol table*. Penggunaan *symbol table* untuk melakukan *Semantic Analysis* agar bisa melakukan *Code Generation*. Target bahasa yang akan diproduksi adalah *IL Assembly* atau CIL (*Common Intermediate Language*).



Gambar 1. Proses Kompilasi

CIL atau *Common Intermediate Language* adalah *term* yang digunakan *European Computer Manufacturers Association* (*ECMA*) untuk membuat sebuah *standard international* dalam *CLI* (*Common Language Infrastructure*). *IL Assembly* memiliki *instruction* yang juga sama dimiliki oleh *native assembly language*, sebagai contoh jika ingin melakukan operasi penjumlahan maka menggunakan *add instruction*, untuk pengurangan menggunakan *sub instruction*, dan *IL Assembly* juga memiliki *instruction set* yang sama seperti pada *native assembly language* pada umumnya. Di bawah ini adalah contoh kode *mnemonic* dan *opcode* pada *IL Assembly*.

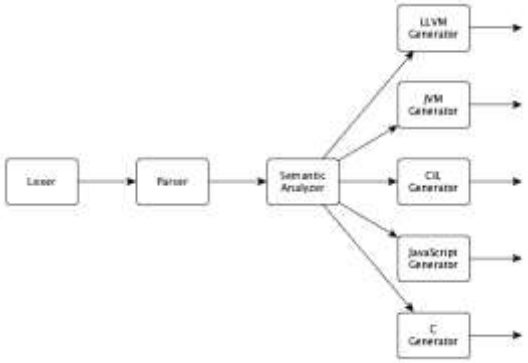
Tabel 1. *Sample Mnemonic* dan *Opcode IL*

Mnemonic	***Opcode***	***Purpose***
Add	0x58	Menjumlahkan dua item yang berada pada *evaluation stack*
Add.ovf	0xd6	
Add.ovf.un	0xd7	

Instruksi Add.ovf memiliki kemampuan untuk mendeteksi *overflow* dan Add.ovf.un juga sama untuk mendeteksi *overflow* hanya saja untuk tipe data *unsigned*.

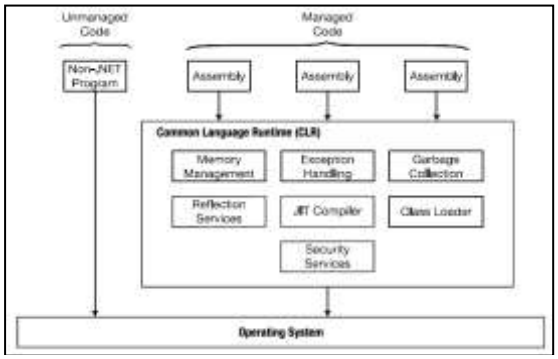
Kompiler pada PBI menggunakan skema *two-stage translations*, hasil produksi *IL Assembly* pada *code generation* akan diterjemahkan kembali oleh IL *Assembler* (ILASM) untuk menghasilkan *managed code* baik itu sebuah *executable* ataupun *dynamic link library*. Di bawah ini adalah *list two stage*

translation yang ada saat ini dan kompiler untuk PBI menggunakan *CIL Generator*.



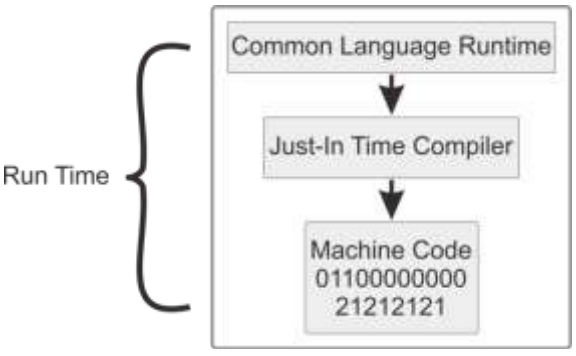
Gambar 2. Skema Penerjemahan 2 Fase

Selanjutnya *managed code* akan dieksekusi di atas mesin eksekusi *Common Language Runtime* (CLR).



Gambar 3. Eksekusi *Managed Code* di dalam *CLR*

Di dalam *Common Language Runtime* terdapat *Just-In Time Compiler* yang akan melakukan *JIT Compilation, managed code* akan diubah ke dalam bahasa mesin agar dapat dieksekusi oleh *CPU*.



Gambar 4. *Runtime Execution* Pada *CLR*

Modern-day Compilers menggunakan *Two-stage Translation* untuk menghadapi tantangan perbedaan bahasa *Assembly* pada setiap processor, sebab setiap *Architectures* memiliki bahasa *Assembly* yang berbeda-beda sebagai contoh pada *Architectures Intelx86* dan *ARM* keduanya memiliki *Assembly Language* yang berbeda.

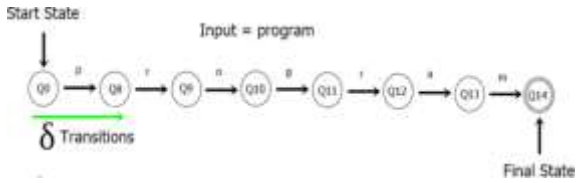
2.3 Analisis Leksikal

Lexical Analyzer adalah salah satu program di dalam kompiler yang digunakan untuk melakukan analisis leksikal pada fase pertama kompilasi. *Lexical Analyzer* akan menerima masukan berupa *source code* dalam bentuk *character stream*. Selanjutnya *character stream* akan diproses untuk memproduksi sebuah *token*, setiap *token* mempunyai *pattern* yang akan menghasilkan *lexeme* yang berbeda beda untuk menghasilkan sebuah *keyword, identifier, number, float, string* dan sebagainya.

Tabel 2. *Token* dan *Lexeme*

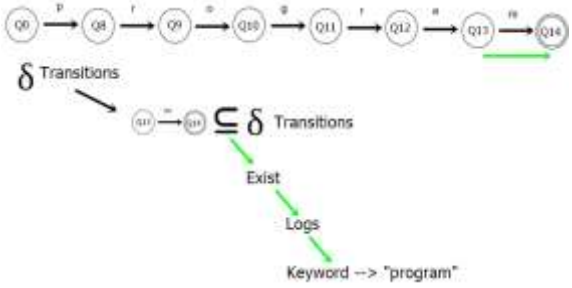
Token	***Lexeme***
Kata Kunci	bulat32, bulat64, pecahan, kalimat, logika, kosong, utama
Pengena;	*Variable1, var, var_1*
Tititk Koma	;
Kurung Kiri	(
Kurung Kanan	)

Metode Analisis leksikal menggunakan *Deterministic Finite Automata*. Sebuah *DFA* terdiri dari 5 *tuple* $M = (Q, \Sigma, \delta, q, F)$
Q = *Finite set* dari *"internal state"*
Σ = *Finite set symbol* yang dipanggil "*input alphabet*"
δ = *lowercase delta* sebagai notasi fungsi *transition*
q0 ∈ Q = q0 elemen dari Q yang menjadi *start state*
F ⊆ Q = F *single* elemen atau *subset* dari *set* Q untuk *final state*



Gambar 5. Komponen *DFA*

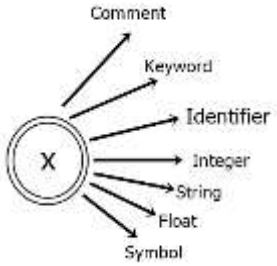
Jika *input* adalah *string* program maka *state* yang dilaluinya akan sampai pada Q14 yang menjadi *final state*. Sebab setiap *transition* yang terjadi merupakan dari *subset transtitions* yang diketahui oleh mesin DFA.



Gambar 6. *Input* sampai pada *Final State*

Setiap *final state* akan memberikan *input* yang akan menjadi *token* sebuah atribut tipe, baik itu

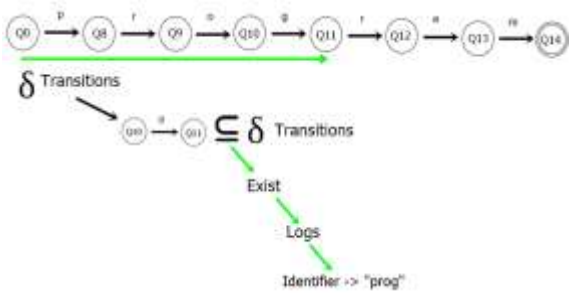
berupa *comment, keyword, identifier, string, integer, float* atau *symbol*.



Gambar 7. *Final State* memberi *attribute*

Setiap *token* selanjutnya akan memiliki beberapa *properties* diantaranya adalah :
1. ID *Token*
2. Nilai *Token*
3. Tipe *Token*
4. Indeks *Token*
5. Baris *Token*

Jika *input string* tidak sampai pada *final state* maka *input* gagal dikenali dan diberi *attribute* sebagai *identifier*.



Gambar 8. *Input* tidak sampai pada *Final State*

2.4 Analisis Sintak

Syntax Analyzer adalah program di dalam kompiler yang digunakan untuk melakukan analisis sintak. *Syntax Analyzer* menerima keluaran dari *lexical analyzer* yaitu serangkaian *token. Syntax Analysis* seringkali disebut *parsing*[7]. Selanjutnya serangkaian *token* tersebut akan diuji susunannya, dengan mencocokan *grammar* yang telah ditetapkan untuk membangun hirarki terstruktur yang disebut dengan *syntax tree* atau *list data*. Metode analisis sintak menggunakan model matematis *set theory* dan meminjam beberapa istilah dalam *regular expression*.

Di dalam PBI terdapat istilah ***Set of order*** yaitu serangkaian struktur sintaksis yang direpresentasikan dalam bentuk *data numeric* di dalam sebuah *set*. Contoh :

<Deklarasi Program> = {1,2,3} (4.1)

Data numeric 1,2,3 adalah *member* dari *set* <Deklarasi Program>, setiap *data numeric* merepresentasikan sebuah *token*. Misalkan, Angka 1 merupakan representasi dari *terminal* ‘program’, angka 2 merupakan representasi dari *terminal identifier* (pengenal) dan angka 3 merupakan *terminal left curly* (kurawal kiri). Jika enumerasi dilakukan maka :

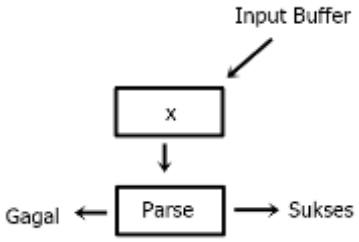
<Deklarasi Program> = {1,2,3}
<Deklarasi Program> =
{’program’, ‘([_]?[a-zA-Z] [0-9])’, ‘{’}

Identifier adalah jenis *token* yang mampu menghasilkan *lexeme* yang berbeda-beda sehingga untuk bisa menulisnya diperlukan notasi *Regular Expression*.

Di dalam PBI juga terdapat istilah *Sequence of Order*, sebuah susunan token yang telah ditentukan posisinya. Setiap *set of order* mempunyai *sequence of order* yang akan digunakan untuk melakukan analisis sintak. Pada *Set of Order* sebelumnya jika diubah kedalam *sequence of order* maka :

1 → 2 → 3 (*Correct Order*) (4.2)

Pada notasi diatas *parser* akan memulai melakukan analisis sintak pada angka 1 dan notasi → digunakan untuk menjelaskan arus *parsing* dari kiri ke kanan pada *sequence of order*. Selanjutnya *parser* akan membaca *sequence of order* satu persatu untuk menguji urutan kebenaranya.
Read Input Buffer → *Parse Current Token (Target Order)* → *Check Correct Order*



Gambar 9. *Parsing Sequence of Order*

Untuk *error recovery* dalam PBI ada dua tipe *error* yang akan terjadi jika menggunakan *Set of Order* :

a. *Error* karena *token* dalam *Input Buffer* masih anggota *sequence of order*
Jika *Input Buffer* = {3,2,3} maka proses *parsing* yang terjadi adalah sebagai berikut :
Parse token 3 gagal! Solusi dibutuhkan token 1.

b. *Error* karena *token* dalam *Input Buffer* bukan anggota *sequence of order*
Jika *Input Buffer* = {1,4,3} maka proses parsing yang terjadi adalah sebagai berikut :
Parse token 1! Berhasil!
Parse token 4 gagal! Solusi dibutuhkan token 2!

Di dalam PBI juga terdapat *Alternative Parsing* yaitu *optional correct order* dan *multi correct order*. Pada *sequence of order* dibawah ini angka 1 bersifat *optional correct order* :

1? → 2 → 3 (4.3)

Optional correct order ditandai dengan notasi '?' sehingga *token* tersebut ada ataupun tidak ada di dalam *sequence of order*, *parser* tidak akan memberikanya *error* secara sintaksis. *Multi Correct Order* ditandai dengan sekumpulan *data numeri*c di dalam [n l.. n], dengan n adalah token dan 'l' simbol pemisah antar token lainya. Sebuah *token* yang ada di dalam *multi correct order* salah satunya bisa diterima oleh *parser*. Sehingga jika terdapat *sequence of order* seperti dibawah ini maka :

1? → 2 → 3 → [7 l 8 l 9] → 4 → 5 → 6 (4.4)

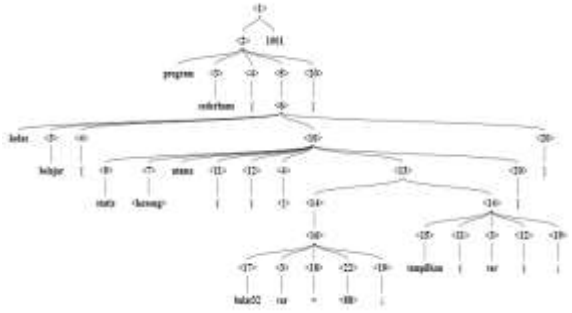
Jika *token* 1 tidak ada maka *Error Recovery* tidak akan terjadi pada *optional correct order*. Jika token 7, 8 atau 9 muncul setelah token 3 maka *Error Recovery* tidak akan terjadi pada *multi correct order* namun jika salah satunya tidak ada maka akan terjadi *error recovery* dengan *multi solution*. Solusi dibutuhkan token 7, 8 atau 9.

Setiap *parser* melewati *set of order* misal dari *set of order* deklarasi program menuju deklarasi kelas maka secara hirarki kumpulan terminal dan non-terminal akan membentuk representasi *data tree* atau *data list* yang akan digunakan untuk Analisis Semantik.

2.5 Analisis Semantik

Semantic Analyzer adalah program di dalam kompiler yang digunakan untuk melakukan analisis semantik. *Semantic Analyzer* akan melakukan cek *error* dimana *error* tersebut adalah tipe *semantic error* yang tidak bisa dideteksi oleh *syntax analyzer*. Sebagai contoh jika terdapat sebuah *variable* dengan *identifier* yang sama hal ini tidak bisa dideteksi oleh *syntax analyzer* tapi mampu dideteksi oleh *semantic analyzer* dengan bantuan *symbol table*.

Pada PBI, *semantic analyzer* adalah fase untuk menganalisis kesalahan kode yang tidak bisa dilakukan *syntax analyzer* sebelum memproduksi IL *Assembly*. Analisis Semantik dilakukan terlebih dahulu agar sumber kode yang akan diterjemahkan bersih dari kesalahan kode. Di bawah ini adalah keseluruhan *parse tree* yang terdiri dari sekumpulan *terminal* dan *non-terminal nodes* dihasilkan dari *syntax analyzer* yang selanjutnya akan digunakan untuk melakukan *tree traversal*.



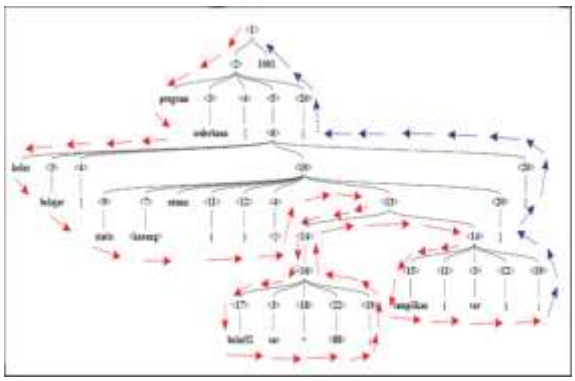
Gambar 10. Representasi *Tree*

Representasi abstrak *non-terminal* pada *node* ditandai dengan *node id* di dalam kurung siku kiri dan kurung siku kanan dan representasi *terminal* dapat dilihat secara langsung.

Tabel 3. *Node ID* dan *Abstraction*

***Node* ID**	***Abstraction***
<1>	<Program>
<2>	<Deklarasi Program>
<3>	<Pengenal>
<4>	<Kurawal Kiri>
<5>	<Kelas>
<20>	<Kurawal Kanan>
<6>	<Deklarasi Kelas>
<10>	<Metode Utama>
<9>	<Nilai Balik>
<7>	<Ciri>
<11>	<Kurung Kiri>
<12>	<Kurung Kanan>
<13>	<Spernyataan>
<14>	<Pernyataan>
<16>	<Deklarasi Variabel>
<17>	<Tipe Primitif>
<18>	<Sama Dengan>
<22>	<Bilangan Bulat>
<19>	<Titik Koma>
<15>	<Metode Masukan Keluaran>
<11>	<Kurung Kiri>

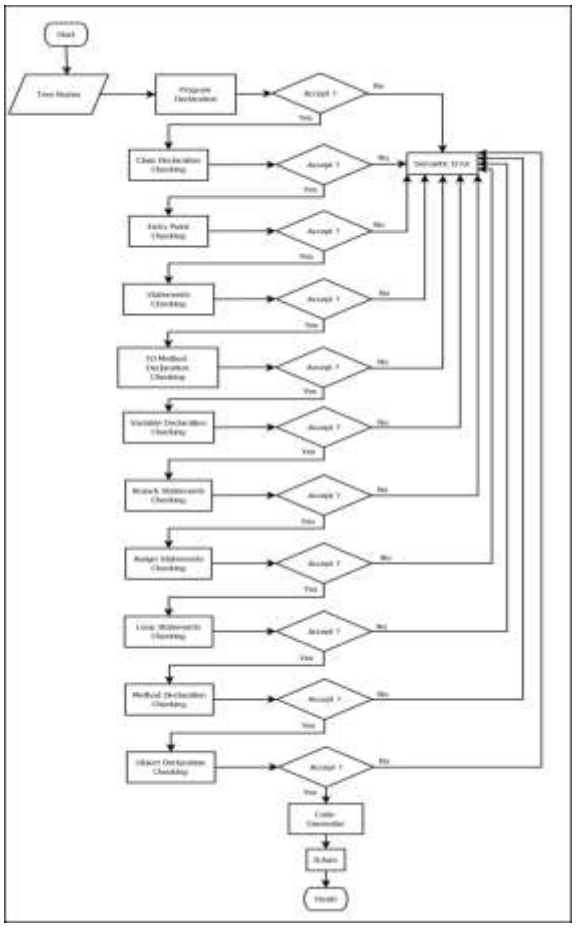
Proses analisis semantik dimulai dari *root* untuk mendapatkan informasi yang akan digunakan untuk mengirimkan informasi kepada *symbol tabel*, sehingga proses *semantic analysis* lebih dalam bisa dilakukan sebelum kompiler melakukan proses *code generation.*



Gambar 11. Proses Analisis Semantik

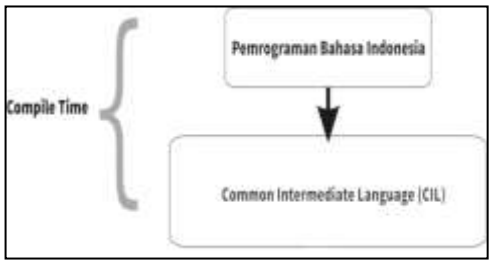
Pada gambar di atas proses *tree traversal* dilakukan secara *depth-first* dari kiri kekanan dimulai dari *root* paling atas, ditandai dengan arah panah berwarna merah. *Rule Node* yang pertama adalah <program> selanjutnya masuk ke dalam *rule node* dari <Deklarasi Program> dan seterusnya hingga mengakses setiap *terminal* atau *leaves* (sekumpulan *token* di dalam *parse tree)* sampai kembali lagi ke dalam *root*. Proses dari bawah ke atas ditandai dengan arah panah berwarna biru.

Ketika *tree traversal* mengakses *node* tertentu di dalam *data tree*, terdapat proses dimana informasi diberikan kepada *symbol table* sambil melakukan proses *semantic analysis* dan *code generation*.



Gambar 12. Alur Proses Analisis Semantik

Proses *semantic analysis* dilakukan agar setiap *rules node* bebas dari *semantic error* saat proses *tree traversal* untuk memproduksi *Common Intermediate Language* (CIL).



Gambar 13. Proses *Code Generation*

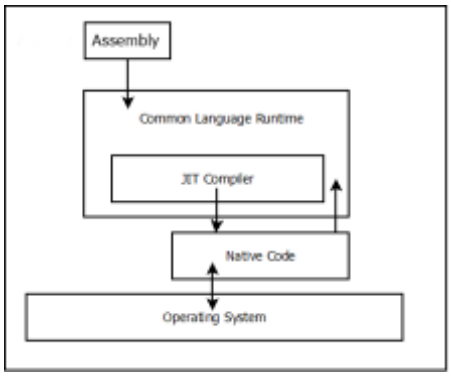
2.6 Mesin Eksekusi

Common Language Runtime adalah jantung dari *.Net Framework*. CLR adalah sebuah *execution environment* atau *execution engine* yang digunakan untuk memuat dan mengeksekusi sebuah kode. Sebuah kode yang berjalan di atas CLR disebut dengan *managed code* dan sebuah kode yang dibuat khusus untuk *win32 executable* dan *dynamic link libraries* (DLLs) disebut dengan *unmanaged code*. Dibandingkan dengan *unmanaged code*, pada *managed code* memiliki keunggulan *automatic memory management*, *type safety*, *security*, dan kesempatan untuk berjalan pada *platform* lainya yang didukung oleh CLR.

Selain itu *Common Language Runtime* juga menyediakan berbagai pelayanan diantaranya adalah :

1. Memuat dan mengeksekusi sebuah program atau *application*.
2. *Memory isolation* pada sebuah program atau *application*.
3. Verifikasi *type safety*.
4. Menterjemahkan *IL Assembly* ke dalam *native code*.
5. Menyediakan *Metadata*.
6. Manajemen *exception* dan *error*.

Dalam CLI (*Common Language Infrastructure*) hasil *assembled* dari *IL Assembly* selanjutnya disebut *Assembly*. Dalam setiap *Assembly* terdapat *Metadata* dan *IL Code*. Sebuah *CLR-compliant compiler* akan menerima lebih dari satu *input* untuk memproduksi sebuah *assembly*.



Gambar 14. Model Eksekusi *CLR*

Terminologi *Assembly* sendiri dalam *Common Language Infrastructure* sudah termasuk *managed code*, *icon* dan *resources* yang digunakanya. Selanjutnya ketika sebuah *assembly* dieksekusi akan dilakukan penerjemahan kembali oleh *JIT Compiler*. Masukan bahasa tingkat tinggi dari pemrograman berbahasa Indonesia setelah *two stage translation* keluarannya disebut dengan *Assembly* baik itu dalam bentuk *executable* atau *dynamic link library*.

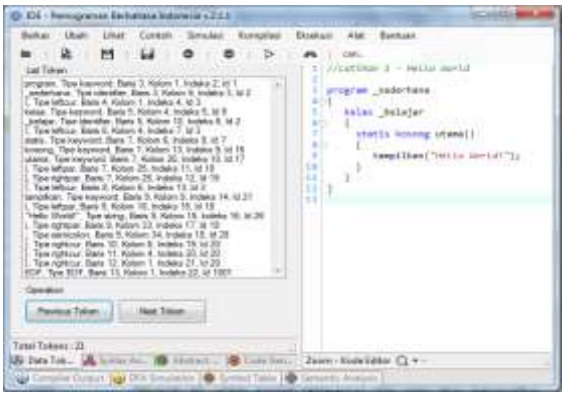
3. PENUTUP

Kompiler untuk PBI hasil dari desain metode sebelumnya :



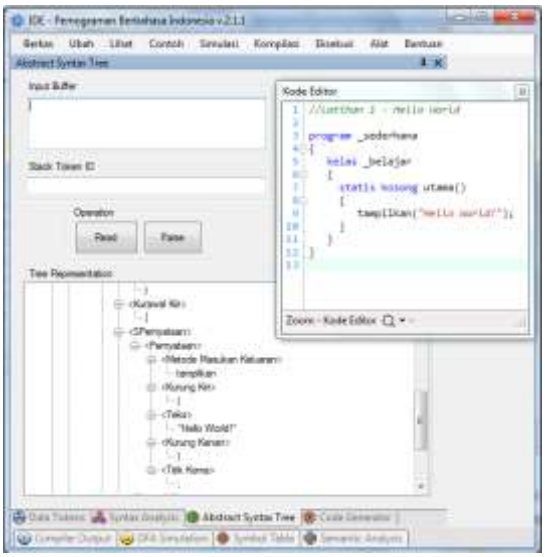
Gambar 15. Tampilan *Code Editor*

Tampilan *token* hasil Analisis Leksikal :



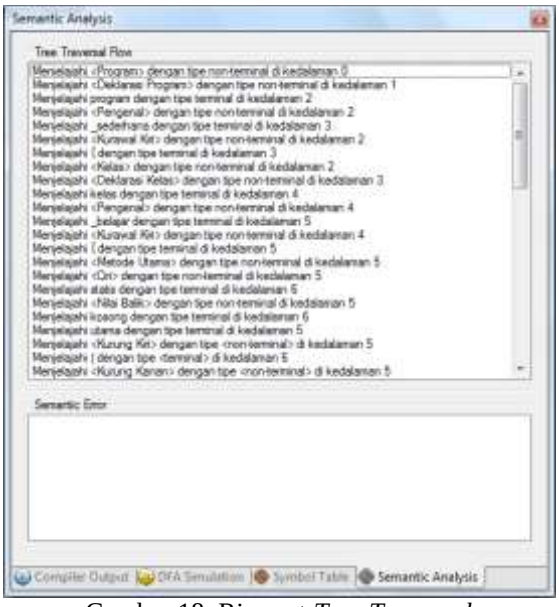
Gambar 16. Daftar *Token*

Data Tree hasil *Parsing* :



Gambar 17. *Data Tree* Hasil Analisis Sintak

Dibawah ini riwayat *tree traversal* pada Analisis Semantik.



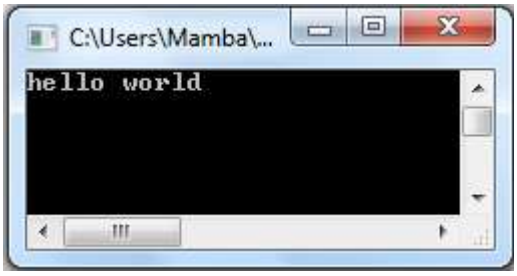
Gambar 18. Riwayat *Tree Traversal*

Di bawah ini adalah hasil *code generation* yang didapatkan :



Gambar 19. Hasil *Code Generation*

Di bawah ini adalah keluaran *executable* yang telah dieksekusi.



Gambar 20. *Executable* dieksekusi

3.1 Kesimpulan

Berdasarkan hasil pengujian pada 50 variasi kode sumber untuk PBI (Pemrograman Berbahasa Indonesia) proses menerjemahkan bahasa pemrograman tingkat tinggi berbahasa Indonesia berhasil dilakukan. PBI juga berhasil memproduksi keluaran *executable* dan *dynamic link library*

menggunakan pemrograman bahasa tingkat tinggi berbahasa Indonesia.

3.2 Saran

Terdapat beberapa saran agar penelitian ini bisa lebih baik kedepanya, diantaranya adalah :

1. Pengembangan *IDE Compiler* untuk pemrograman berbahasa Indonesia yang mampu berjalan pada sistem operasi *Linux* dan *Macintosh*.
2. Pengembangan *IDE Compiler* untuk pemrograman berbahasa Indonesia yang mendukung aplikasi berbasis GUI (*Graphic Unit Interface*).

DAFTAR PUSTAKA

[1] Alfred, Aho "Introduction", *Compiler Principles, Techniques, & Tools*, : Pearson, 2006. pp 1

[2] Fraser, John "Assesment of The English Language", "*English : The Prototypical World Language for The Twenty First Century*", : Lulu, 2009. pp 107

[3] Shampa Banik, Md Monjurul Islam, Md Azher Uddin, "New Approach to Develop Bilingual Compiler", *Journal of Engineering Research and Applications*,. pp 257-261, February, 2014.

[4] Dali S. Naga, "Kilang 002 : Basic Dalam Bahasa Indonesia," Perpusnas.go.id, 1984 [Online]. Tersedia : http://onesearch.perpusnas.go.id/Record/IOS1-INLIS000000000029667 [Diakses : 31 Januari 2017]

[5] Dali S. Naga, "Kilang 002 : Basic Dalam Bahasa Indonesia," Perpusnas.go.id, 1984 [Online]. Tersedia : http://onesearch.perpusnas.go.id/Record/IOS1-INLIS000000000029667 [Diakses : 31 Januari 2017]

[6] Hassanudin, Haris, "BAIK (Bahasa Anak Indonesia Untuk Komputer) - Programming Language Based on Indonesian Lexical Parsing For Multitier Web Development", Jurnal Ilmu Komputer dan Informasi, Volume 3, pp 47

[7] Puntambekar. "Overview of Compilation," *Compiler Design*. India : Technical Publication Pune. 2009. pp 4