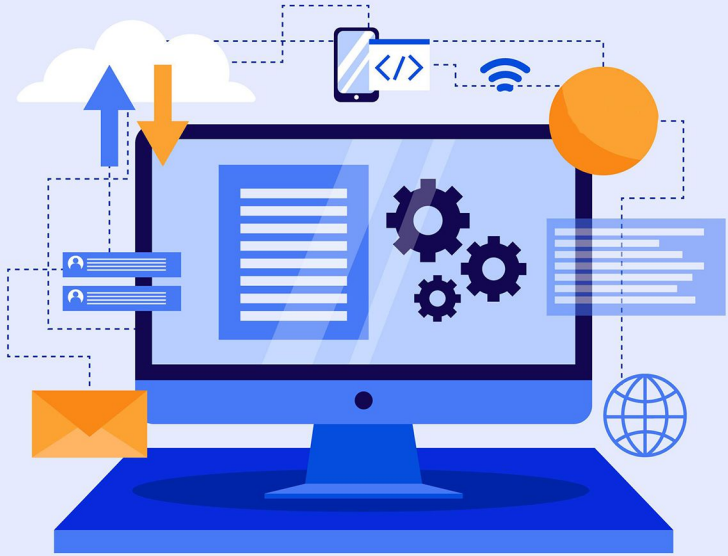
Fathur Rahman, S.Kom., M.Kom
As'ary Ramadhan, S.Kom., M.Cs
Yusri Ikhwani, S.Kom., M.Kom



RANCANGAN DAN TEKNIK KOMPILASI

LEXICAL ANALYSIS | SYNTAX ANALYSIS | INTERPRETASI
TYPE CHECKING | CODE GENERATION | REGISTER ALLOCATION
INTERMEDIATE - CODE GENERATION

RANCANGAN DAN TEKNIK KOMPILASI

(Dasar-dasar Rancangan dan Teknik Kompilasi)

FATHUR RAHMAN., S.Kom., M.Kom

AS'ARY RAMADHAN., S.Kom., M.Cs

YUSRI IKHWANI., S.Kom., M.Kom

UNIVERSITAS ISLAM KALIMANTAN MUHAMMAD ARSYAD AL BANJARI BANJARMASIN

2022

RANCANGAN DAN TEKNIK KOMPILASI
(Dasar-dasar Rancangan dan Teknik Kompilasi)

Penulis:
Fathur Rahman, S.Kom., M.Kom,
As'ary Ramadhan, S.Kom., M.Cs,
Yusri Ikhwani, S.Kom., M.Kom

Editor:
Taufiq Hidayah

Penyunting:
Antoni Pardede, S.Si.,M.Si.,Ph.D.

Tata letak:
Aris Setia Noor, S.E., M.Si

Desain sampul:
M.Fikri Ansari, S.Kom

Penerbit:
Universitas Islam Kalimantan Muhammad Arsyad Al Banjari, Banjarmasin

Redaksi: Jl. Adhyaksa No2, Kayutangi, Banjarmasin 70123

Cetakan Pertama 2022
ISBN: 978-623-7583-83-7

KATA PENGANTAR

Segala puji kami haturkan pada Allah subhanhu wata ’ala yang telah memberi tak berhingga banyak nikmat dan karunia Nya. sehingga penyusunan buku ini akhirnya bisa diselesaikan.

Semakin meningkatnnya kebutuhan buku Kompilasi khususnya "Rancangan dan Teknik Kompilasi", terlihat dari matakuliah "Teknik Kompilasi" hampir diberikan di seluruh perguruan tinggi khususnya jurusan Teknik Informatika , membuat kami berkeinginan untuk menyusun materi-materi yang kami ambil dari beberapa buku teknik kompilasi. Kami berharap dengan tersusunya buku ini bisa memenuhi kebutuhan para mahasiswa akan buku teknik kompilasi khususnya yang berbahasa indonesia.

Buku referensi yang digunakan dalam menyusun buku ini sebagian besar (khususnya bab 1, 4, 7) adalah buku karangan Torben Mogensen dengan judul "*Introduction to Compiler Design*" dan sebagian besarnya lagi (khususnya bab 2, 5, 6) diambil dari buku karangan Alfred V. Aho,dkk. dengan judul "*Compilers, Principles, Techniques, & Tools*" serta bab 3 diambil dari artikel Yohan.es dengan judul "Membuat Interpreter dan Compiler".

Kami mengucapkan banyak terima kasih pada kedua orang tua atas segala bimbingan, do’a dan pengorbanannya, juga pada keluarga kami atas kebersamaan dan dukungannya. Mudah-mudahan semuanya tadi tidak sia-sia dan mendapatkan imbalan yang sebesar-besarnya dari Allah subhanahu wata’ala. Amiin..

Akhir kata, sekali lagi mudah-mudahan buku ini bisa memberi manfaat bagi siapa saja yang membutuhkannya.

Banjarmasin, 2022
Penulis,

Fathur Rahman, S.Kom.,M.Kom
As’ary Ramadhan, S.Kom.,M.Cs
Yusri Ikhwani, S.Kom.,M.Kom

DAFTAR ISI

UNIVERSITAS ISLAM KALIMANTAN
MUHAMMAD ARSYAD AL BANJARI BANJARMASIN

Jl.Adhyaksa No.2 Kayu Tangi Banjarmasin 70123. Telp/Facs (0511) 3304852. www.uniska-bjm.ac.id

FORMULIR
RENCANA PEMBELAJARAN SEMESTER (RPS)

No. Dokumen | No. Revisi | Hal. 1 dari 2

RENCANA PEMBELAJARAN SEMESTER

Mata Kuliah : Teknik Kompilasi	Semester : 5	SKS : 2	Kode : INT345	
Program Studi : Teknik Informatika	Dosen Pengampu/Penanggungjawab : Fathur Rahman, S.Kom, M.Kom			
Mata Kuliah Prasyarat				
Capaian Pembelajaran Lulusan	Sikap : Membangun aplikasi perangkat lunak yang berkaitan dengan pengetahuan ilmu komputer Keterampilan Umum : Merancang dan mengembangkan program aplikasi untuk memanipulasi model gambar, grafis, citra, desktop dan web serta dapat memvisualisasikannya Keterampilan Khusus : Membangun aplikasi perangkat lunak dalam berbagai area yang berkaitan dengan bidang robotic, pengenalan suara, sistem cerdas dan bahasa natural Pengetahuan : Membangun dan mengevaluasi perangkat lunak dalam berbagai area, termasuk yang berkaitan dengan interaksi antara manusia dan komputer			
Capaian Pembelajaran Matakuliah	Memahami konsep-konsep algoritma dan kompleksitas, meliputi konsep-konsep sentral dan kecakapan yang dibutuhkan untuk merancang, menerapkan dan menganalisis algoritma untuk menyelesaikan masalah. (khususnya di bidang compiler)			
Deskripsi Matakuliah	Dapat memahami konsep dasar teknik kompilasi, meliputi fungsi, komponen, tahapan-tahapan dan mekanisme kerjanya, serta cara perancangan suatu kompiler sederhana.			

UNIVERSITAS ISLAM KALIMANTAN
MUHAMMAD ARSYAD AL BANJARI BANJARMASIN

Jl.Adhyaksa No.2 Kayu Tangi Banjarmasin 70123. Telp/Facs (0511) 3304852. www.uniska-bjm.ac.id

FORMULIR
RENCANA PEMBELAJARAN SEMESTER (RPS)

No. Dokumen | No. Revisi | Hal. 1 dari 2

Referensi	Referensi Utama : 1. Alfred V. Abo, Monica S. Lam, Ravi Sethi, &Jeffrey D. Ullman 1986, Compilers : principles, techniques, and tools / 2nd ed Pearson Addison Wesley 2. Torben Ægidius Mogensen. Introduction to Compiler Design, Second Edition. Undergraduate Topics in Computer Science. Springer, 2017. 3. Yohanes Nugroho. Membuat interpreter/compiler itu mudah, 2019. https: //yohan.es/compiler/, Last accessed on 2021-11-30.	

Pert. ke-	Sub-CPMK	Bahan Kajian	Indikator	Metode Pembelajaran	Pengalaman Belajar	Penilaian (Jenis dan Kriteria)	Bobot	Waktu	Referensi
1	Mampu menjelaskan arti, tujuan, definisi compiler dan interpreter	• Kegunaan Kompiler • Struktur Kompiler • Perbedaan compiler dan interpreter	**Indikator:** • Kelengkapan dan kebenaran mengenai konsep compiler dan interpreter	• Discovery learning • Cooperative learning • Small group discussion	• Mahasiswa mampu memahami poses compiler dan interpreter	**Jenis Penilaian:** Tes lisan atau berupa kuis **Kriteria:** • Ketepatan dalam menjawab	8%	• TM; 2x45 = 90 menit • BT; 2x50 = 100 menit • BM; 2x50 = 100 menit	

2	Mampu menjelaskan arti definisi dan konsep notasi dan bahasa	• Istilah dalam Bahasa (alfabet, huruf, kata, token lexem/besaran leksik, tata bahasa, bahasa, pengenal bahasa) • Hirarki	**Indikator:** • Kelengkapan dan kebenaran penjelasan mengenai notasi dan bahasa	• Discovery learning • Cooperative learning • Small group discussion	Mahasiswa mampu memahami konsep notasi dan Bahasa.	**Bentuk Penilaian:** Tes tertulis **Kriteria:** • Ketepatan dalam mengerjakan soal	12%	• TM; 2x45 = 90 menit • BT; 2x50 = 100 menit • BM; 2x50 = 100 menit	

		Chomsky							
3	Mampu, Mengetahui & Memahami Bahasa Reguler	• Kelas Reguler Grammar dan beberapa bentuk ekspresi Reguler	**Indikator:** • Kelengkapan dan kebenaran penjelasan mengenai konsep Reguler Grammar	• Discovery learning • Cooperative learning • Small group discussion	Mahasiswa mampu memahami konsep Bahasa Grammar	**Bentuk Penilaian:** Tes tertulis **Kriteria:** • Ketepatan dalam menjawab	10%	• TM; 2x45 = 90 menit • BT; 2x50 = 100 menit • BM; 2x50 = 100 menit	
4	Mampu menjelaskan arti definisi dari Context Free	- Tata bahasa Context Free dan hubungannya sebagai parser (penganalisa sintak)	**Indikator:** • Kelengkapan dan kebenaran penjelasan mengenai konsep Context Free Grammar	• Discovery learning • Cooperative learning • Small group discussion	Mahasiswa mampu memahami konsep Context Free atau Grammar Bebas.	**Bentuk Penilaian:** Tes lisan / tertulis **Kriteria:** • Ketepatan dalam menjawab	10%	• TM; 2x45 = 90 menit • BT; 2x50 = 100 menit • BM; 2x50 = 100 menit	
5	Mampu Menjelaskan Tahapan-tahapan Kompilasi	- Tahapan Kompilasi mempunyai 6 (enam) Tahapan. - Urutan Prosedur Tahapan Kompilasi	**Indikator:** • Kelengkapan dan kejelasan tentang tahapan kompilasi.	• Discovery learning • Cooperative learning • Small group discussion	Mahasiswa mampu memahami Tahapan Kompilasi	**Bentuk Penilaian:** Tes tertulis **Kriteria:** • Ketepatan dalam menjawab	10%	• TM; 2x45 = 90 menit • BT; 2x50 = 100 menit • BM; 2x50 = 100 menit	

6	Mampu Menjelaskan dari Analisa Leksikal	- Ekivalensi tata bahasa reguler - Ekspresi reguler - Otomata hingga	**Indikator:** • Kelengkapan dan kejelasan tentang	• Discovery learning • Cooperative learning	Mahasiswa mampu memahami tentang Analisa	**Bentuk Penilaian:** Tes lisan / tertulis **Kriteria:** • Ketepatan	10%	• TM; 2x45 = 90 menit • BT; 2x50 = 100 menit	

		- Tugas scanner (penganalisa leksikal)	Analisa Leksikal	• Small group discussion	Leksikal	dalam menjawab		• BM; 2x50 = 100 menit	
7	Mampu menjelaskan& Memahami Parser (Penganalisa Sintaksis)	• Metode dan teknik-teknik Parsing • Recursive-predictive parser dan implementasinya dalam bahasa tertentu	**Indikator:** • Kelengkapan dan kejelasan tentang Analisa Sintaksis	• Discovery learning • Cooperative learning • Small group discussion	Mahasiswa mampu memahami tentang Analisa Sintaksis	**Bentuk Penilaian:** Tes tertulis **Kriteria:** • Ketepatan dalam menjawab	10%	• TM; 2x45 = 90 menit • BT; 2x50 = 100 menit BM; 2x50 = 100 menit	
8	UJIAN TENGAH SEMESTER								
9 - 10	Mampu Menjelaskan dan Mengetahui Analisa Semantik, Kode Antara, dan Pembangkitan Kode	• Analisa Semantik • Kode Antara • Pembangkitan Kode	**Indikator:** • Kelengkapan dan kejelasan tentang Analisa Semantik, Kode Antara dan Pembangkitan Kode	• Discovery learning • Cooperative learning • Small group discussion	Mahasiswa mampu memahami tentang Analisa Semantik, Kode Antara, dan Pembangkitan Kode	**Bentuk Penilaian:** Tes tertulis **Kriteria:** • Ketepatan dalam menjawab	10%	• TM; 2x45 = 90 menit • BT; 2x50 = 100 menit BM; 2x50 = 100 menit	

11	Mampu menjelaskan dan Mengetahui serta Memahami Teknik Optimasi	• Dependensi Optimasi • Optimasi Lokal • Optimasi Global	**Indikator:** • Kelengkapan dan kejelasan tentang Teknik Optimasi	• Discovery learning • Cooperative learning • Small group discussion	Mahasiswa mampu memahami tentang Teknik Optimasi.	**Jenis Penilaian:** Tes lisan atau berupa kuis **Kriteria:** • Ketepatan dalam menjawab	5%	• TM; 2x45 = 90 menit • BT; 2x50 = 100 menit BM; 2x50 = 100 menit	

12	Mampu menjelaskan arti, definisi dan konsep intermediate Code Generator	- Gambaran Umum Kode Antara (Intermediate Code Generator) - Syntax Directed Translation (SDT) - Syntax Tree - Three Address Code - N-Tuple	**Indikator:** • Kelengkapan dan kejelasan mengenai Intermediate Code Generator	• Discovery learning • Cooperative learning • Small group discussion	Mahasiswa mampu memahami tentang Intermediate Code Generator	**Jenis Penilaian:** Tes lisan atau berupa kuis **Kriteria:** • Ketepatan dalam menjawab	5%	• TM; 2x45 = 90 menit • BT; 2x50 = 100 menit • BM; 2x50 = 100 menit	
13	Mampu menjelaskan arti, definisi dan konsep Code Generator	- Gambaran Umum Kode Pembangkit / Code Generator)	**Indikator:** • Kelengkapan dan kejelasan mengenai Code Generator	• Discovery learning • Cooperative learning • Small group discussion •	Mahasiswa mampu memahami tentang Code Generator	**Jenis Penilaian:** Tes lisan atau berupa kuis **Kriteria:** • Ketepatan dalam menjawab	5%	• TM; 2x45 = 90 menit • BT; 2x50 = 100 menit • BM; 2x50 = 100 menit	
14	Mampu menjelaskan dan memahami Penanganan Kesalahan / Error Handler	- Kesalahan program - Penanganan Kesalahan	**Indikator:** • Kelengkapan dan kejelasan tentang Penganan Kesalahan	• Discovery learning • Cooperative learning • Small group discussion	Mahasiswa mampu memahami tentang Penanganan Kesalahan	**Jenis Penilaian:** Tes lisan atau berupa kuis **Kriteria:** • Ketepatan dalam menjawab	5%	• TM; 2x45 = 90 menit • BT; 2x50 = 100 menit BM; 2x50 = 100 menit	

15	Mampu Menjelaskan dan Mengetahui Perbedaan	• Pengertian Interpreter beserta contoh dan implementasinya. • Pengertian Kompiler	**Indikator:** • Kelengkapan dan kejelasan tentang	• Discovery learning • Cooperative learning	Mahasiswa mampu memahami tentang	**Bentuk Penilaian:** Tes tertulis **Kriteria:** • Ketepatan	5%	• TM; 2x45 = 90 menit • BT; 2x50 = 100 menit	

	Interperter dan Kompiler	beserta contoh dan implementasinya	Perbedaan interpreter dan Kompiler	• Small group discussion	Perbedaan Interpreter dan Kompiler	dalam menjawab		BM; 2x50 = 100 menit	
16	UJIAN AKHIR SEMESTER								

Daftar Referensi:

1. Alfred V. Abo, Monica S. Lam, Ravi Sethi, &Jeffrey D. Ullman 1986, Compilers : principles, techniques, and tools / 2nd ed Pearson Addison Wesley
2. Torben Ægidius Mogensen. Introduction to Compiler Design, Second Edition. Undergraduate Topics in Computer Science. Springer, 2017.
3. Yohanes Nugroho. Membuat interpreter/compiler itu mudah, 2019. https://yohan.es/compiler/, Last accessed on 2021-11-30.

Tugas mahasiswa dan penilaiannya:

1.
2.

Mengetahui
Ketua Program Studi Teknik Informatika

Dr. Ir. H. M. Muflih, M.Kom

Banjarmasin, 2021
Dosen Pengampu/Penanggung jawab MK

Fathur Rahman, S.Kom, M.Kom

BAB 1
PENGANALISA LEKSIKAL

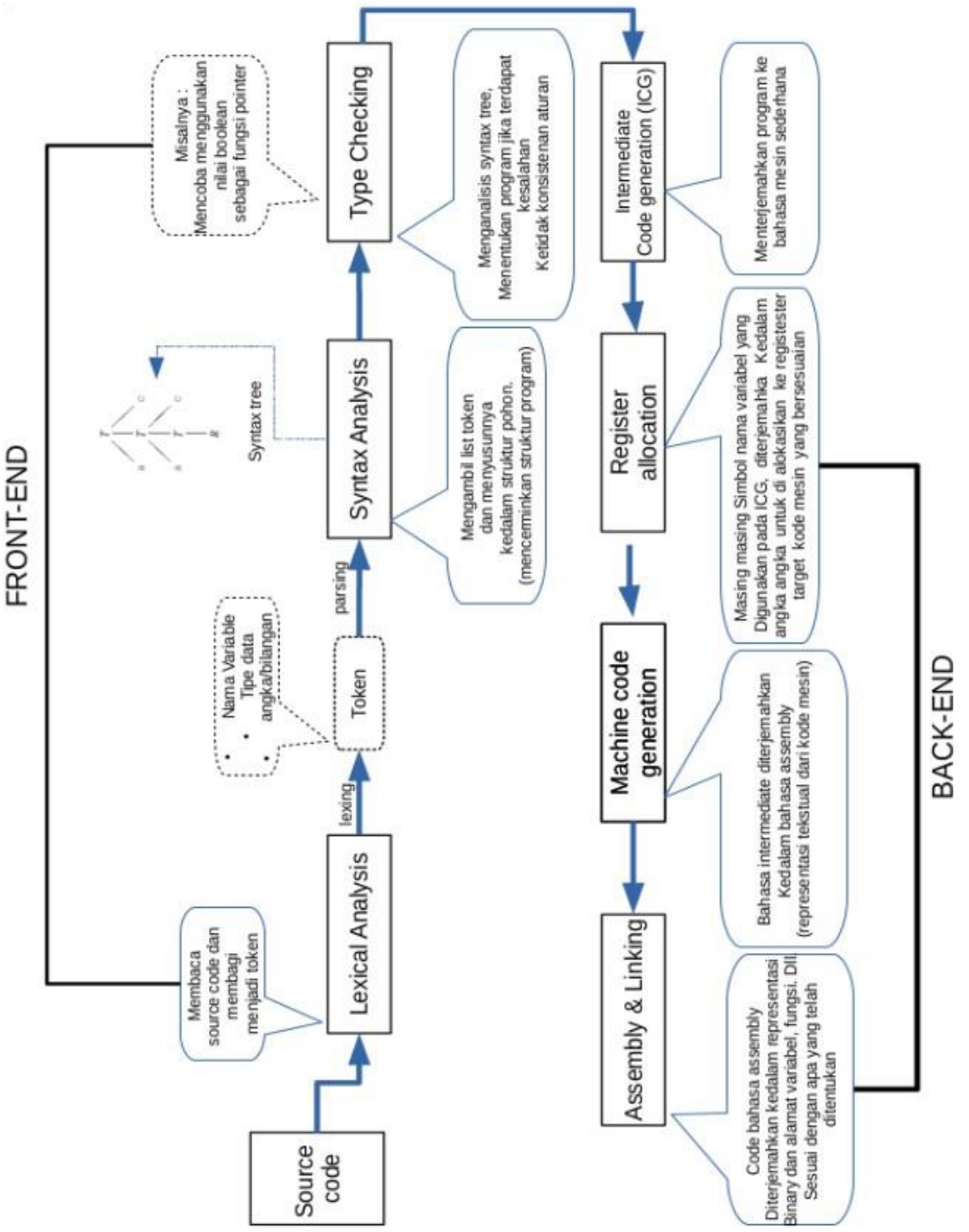
1.1 Pendahuluan

Agar dapat mereduksi kompleksitas dari rancang bangun komputer dibuatlah perintah program yang paling sederhana tetapi dapat melakukan suatu perintah dengan sangat cepat. Program untuk sebuah komputer harus dibangun dengan mengkombinasikan perintah program yang sangat sederhana kedalam suatu program yang disebut dengan bahasa mesin. Kekurangan dari bahasa mesin adalah rawan akan kesalahan dan membutuhkan usaha yang besar untuk dipelajari Maka sebagai gantinya adalah dengan menggunakan bahasa pemrograman tingkat tinggi. Namun bahasa pemrograman tingkat tinggi akan sangat berbeda dari bahasa mesin saat dijalankan pada komputer. Jadi untuk menjembatani hal ini maka hadirlah sebuah program yang disebut dengan *compiler*. Pada kompiler bisa dilakukan optimisasi atau peningkatan kualitas kode yang bisa dijalankan. Ada yang dioptimasi supaya lebih cepat, ada yang supaya lebih kecil, ada yang dioptimasi untuk sistem dengan banyak processor. Kalau interpreter susah tidakbisa dioptimasikan. Kompiler membutuhkan linker untuk menggabungkan kode objek dengan berbagai macam library demi menghasilkan suatu kode yang bisa dijalankan oleh mesin. Kalau interpreter tidak butuh linker. Interpreter cocok untuk membuat atau menguji coba modul atau sub-routine atau program-program kecil. Kalau compiler agak repot karena untuk mengubah suatu modul atau kode objek kecil, maka harus dilakukan proses linking atau penggabungan kembali semua objek dengan library yang diperlukan.

> "Kompiler adalah penerjemah program yang ditulis dengan bahasa pemrograman tingkat tinggi kebahasa program tingkat rendah yang bisa dimengerti oleh komputer."
> -(Torben et al, 2017).

Untuk keperluan yang memperhatikan kompleksitas waktu bahasa mesin masih digunakan, karena bahasa pemrograman tingkat tinggi mungkin masih berjalan agak lambat karena Kompilator bahasa pemrograman tingkat tinggi melakukan proses kompilasi dengan cara menganalisis kode sumber secara keseluruhan. Akan tetapi kompiler yang dirancang dengan struktur program yang bagus akan mampu menterjemahkan secara cepat mendekati kecepatan bahasa mesin yang kita tulis. Kompiler berbeda dengan assembler karena biasanya kompilator akan menghasilkan kode obyek (object code) yang bisa berupa file executable pada sisten operasi Windows atau file bin pada sisten operasi berbasis Unix.

Tapahan pada kompiler secara umum dapat disusun seperti berikut, namun dibeberapa kompiler tahapan ini mungkin akan tampak sedikit berbeda. Lihat Gambar 1.1.



Gambar 1.1: Tahapan Umum Teknik Kompilasi

1.1.1 Cara Kerja Compiler

Sudah ada banyak bahasa di dunia ini, untuk apa belajar membuat interpreter atau compiler untuk sebuah bahasa?

1. Belajar membuat compiler merupakan latihan pemrograman yang bagus. Untuk membuat compiler, kita perlu mengetahui parsing, abstract syntax tree, dan aneka hal lain yang memerlukan algoritma dan struktur data yang kompleks.

2. Pengetahuan pembuatan compiler terutama parsing dan pembuatan abstract syntax tree dapat diaplikasikan ke berbagai bidang lain. Contoh aplikasi kedua ilmu tersebut misalnya source-to-source translators (penerjemahan otomatis dari satu bahasa pemrograman ke bahasa lain, misalnya f2c yang menerjemahkan FORTRAN ke C), refactoring tools, reengineering tools, metrics tools (mengukur jum lah baris kode/fungsi, dsb untuk metrik perangkat lunak), consistency checkers (memeriksa apakah kode program konsisten dengan aturan), dan lain-lain. Beberapa contoh aplikasi lain dari ilmu yang dipelajari ketika membuat compiler bisa dilihat di http://progtools.comlab.ox.ac.uk/members/torbjorn/thesis

Tidak terlalu banyak orang yang belajar membuat compiler untuk general purpose language, kebanyakan orang belajar untuk membuat Domain specific language (DSL). DSL adalah bahasa yang digunakan hanya untuk aplikasi tertentu. Contoh DSL misalnya bahasa scripting di game, atau macro di word processor tertentu, bahkan Cascading Style Sheeet (CSS) pada HTML/XML juga bisa dipandang sebagai DSL. DSL biasanya jauh lebih sederhana dibandingkan general purpose language, karena tujuannya spesifik untuk domain tertentu.

Compiler membaca sebuah source code dalam bentuk teks, menyatukan karakter-karakter yang berhubungan menjadi token, lalu memeriksa apakah token-token tersebut memenuhi grammar, setelah itu compiler akan memeriksa semantik input, dan membuat output dalam sebuah bahasa (yang umumnya adalah assembly). Jika outputnya adalah assembly maka proses berikutnya adalah assembling yang dilakukan dengan assembler untuk menghasilkan bahasa mesin. Proses terakhir untuk membuat executable file dilakukan oleh linker.

Mungkin terlalu banyak kata-kata asing di kalimat tersebut, secara singkat:

- *source code* adalah kode program tekstual dalam bahasa tertentu.

- *token* adalah kumpulan karakter yang memiliki arti khusus. Misalnya token 123 adalah token angka seratus dua puluh tiga, tidak dipandang sebagai karakter terpisah 1, 2, dan 3

- *grammar* adalah tata bahasa. Misalnya dalam program kalkulator 11 23 merupakan sesuatu yang benar, tapi 12 + / 4 tidak benar (plus sebelum bagi tidak memiliki makna)

- *assembly* adalah bahasa sederhana yang mudah diterjemahkan ke bahasa mesin. Tools untuk menerjemahkannya adalah assembler, dan prosessnya namanya assembling.

- *executable file* adalah file program yang siap dijalankan

- *linker* adalah tools yang akan menggabungkan kode mesin yang kita tulis dengan library

- *library* adalah kumpulan kode yang disatukan dalam sebuah file yang dapat digunakan oleh sebuah program. Biasanya programmer tidak akan menulis sendiri kode untuk menghitung sinus, kosinus, dsb. Kode-kode standar ini biasanya ada pada sebuah library.

1.2 Penganalisa Leksikal

Kata "leksikal" dapat diartikan "hal – hal yang berkaitan dengan kata". Dalam istilah bahasa pemrograman kata-kata merupakan entitas seperti nama variabel , bilangan, tipe data. Entitas dalam artian lama (tradisional) disebut dengan token. Penganalisa leksikal kadang juga disebut dengan *lexer*, atau *scanner* tugasnya adalah mengambil string inputan dan membagi string tersebut mejadi rentetan token. Tujuan utama dari scanner adalah untuk mempermudah pemrosesan pada tahapan penganalisa sintaksis. Spesifikasi scanner biasanya ditulis menggunakan ekspresi regular (regular expression) atau secara otomatis menghasilkan notasi lexical analyzer berdasarkan regexps. Notasi alajabar digunakan untuk mendeskripsikan himpunan string. Hasil dari proses scanner berada dalam kelas program yang sederhana biasa disebut dengan finite automata. *Reguler expression* dapat di- konversikan ke dalam *finite automata*.

1.3 Regular Expression

Regular expression (regexps) digunakan untuk menyatakan himpunan string. Himpunan string disebut dengan bahasa (*language*). Contoh himpunan string adalah semua himpunan integer konstan atau semua himpunan nama-nama variabel. Contoh penggunaan regexps sebagai berikut:

1. Jika a adalah regexps maka a menyatakan bahasa (*language*) yang terdiri dari string a. Bahasa ini kita sebut $L(a)$.

2. Jika r dan s adalah regexps, maka bentuk rs merupakan *concatenation* yang juga merupakan regexps menyatakan bahasa terhadap semua kemungkinan string diperoleh dari *concatenating* bahasa string yang dinyatakan dengan r ke sebuah bahasa string yang dinyatakan dengan s. Bahasa ini kita sebut $L(rs)$

3. Jika r adalah regexps, pengulangan r^* juga merupakan regexps yang menyatakan bahasa terdiri dari string didapat dengan cara menggabungkan *concate* lebih dari 0 string instance dinyatakan bersamaan dengan r. Bahasa ini kita sebut $L(r^*)$.

   Perlu diperhatikan bahwa $r^0 = E$, merupakan string kosong (*empty string*) dengan panjang 0; $r^1 = r$, $r^2 = rr$, $r^3 = rrr$, dan seterusnya; r^* menyatakan bilangan tak terhingga pada string terhingga.

4. ε adalah regexp yang menyatakan bahasa dan hanya memuat string kosong.

5. Jika a adalah regexps. Bahasa ini kita sebut $L(\ ) = \emptyset$ Perlu diperhatikan bahwa jika L adalah himpunan string, maka L^* adalah himpunan seluruh string untuk seluruh regexps termasuk regexp ε.

6. Jika r adalah regexps, maka (r) juga sebuah regexps yang menyatakan kesamaan bahasa. Tanda kurung hanya digunakan untuk mengelompokkan.

7. 0 adalah regexps yang menyatakan string tunggal 0

8. 1 adalah regexps yang menyatakan string tunggal 1

9. 0|1 adalah regexps yang menyatakan 2 bahasa string 0 dan 1.

10. (0|1) adalah regexps, menyatakan bahasa yang sama terhadap 2 string 0 dan 1

11. $(0|1)^*$ adalah regexps yang menyatakan semua bahasa string, termasuk juga string kosong,0 dan 1: $\varepsilon, 0, 1, 00, 01, 10, 000, 001, 010, 011, ..., 000111, ...$

12. $1(0|1)^*$ adalah regexps yang menyatakan semua bahasa string 1 dan 0 yang diawali dengan 1.

13. $0|1(0|1)^*$ adalah regexps yang menyatakan bahasa yang terdiri dari semua bilangan biner (mengecualikan didepan 0 yang tidak diperlukan)

> [!] Perlu diperhatikan bahwa dalam melakukan konstruksi regexps pengulangan $*$ lebih didahulukan kemudian penggabungan (*concatenation*) dan terakhir pengubahan (*alternation*). Contoh : $01 * 0|1$ equivalent dengan $(0(1*)0)|(1*)$

Untuk dapat mengetahui pernyataan-pernyataan himpunan string kita dapat menggunakan aturan derivasi berikut ini:

Regexp	Aturan derivasi	Keterangan informal
a		sebuah huruf string tunggal a. Tidak dikenakan aturan derivasi karena sudah menjadi string
E	$E \Rightarrow$	String kosong
$s\|t$	$s\|t \Rightarrow s$ $s\|t \Rightarrow t$	s atau t. Memungkinkan untuk melakukan perkalian dengan derivasi yang berbeda
st	$st \Rightarrow s^1t^1$, jika $s \Rightarrow s^1$ dan $t \Rightarrow t^1$	sesuatu diturunkan dari s diikuti dengan sesuatu diturunkan dari t
s^*	$s^* \Rightarrow$ $s^* \Rightarrow$ ( )	penggabungan terhadap sembarang bilangan string (termasuk 0) diturunkan dari s. Bilangan tergantung dari seberapa banyak aturan kedua digunakan

Contoh

L(a(b|c)) ={ab, ac}
karena,
a(b|c) ⇒ a(b) = ab dan a(b|c) ⇒ a(c) = ac

Contoh-2

L((a|b)*) ={ab}
karena,
(a|b)* ⇒ (a|b)(a|b)* ⇒ a(a|b)* ⇒ (a|b)(a|b)* ⇒ ab(a|b)* ⇒

Tanda kurung digunakan untuk membedakan yang mana mesti dikerjakan terlebih dahulu, sama seperti pada kesepakatan aljabar bahwa perkalian didahulukan daripada penambahan. Misal $3+4\times5$ equivalent terhadap $3+(4\times5)$ bukan $(3+4)\times5$. Untuk regexps sendiri *cleene closure (*)* didahulukan daripada *concate*. Misalnya $a|ab^*$ equivalent terhadap $a|(a(b*))$.

Latihan !!!

Diberikan regexps $s = (a|b)(c|d|E)$

1. Menggunakan aturan derivasi , tunjukkan bahwa $L(s)$ berisi string ac. $L(s) = \{ac\}$

2. Nyatakan semua himpunan string $L(s)$ yang terdapat pada regexps $(a|b)(c|d|E)$.

1.3.1 Shorthand

Seperti pada bahasa pemrograman, kita sering memberikan penggunaan nama pada suatu hal misalnya: $D = 1|2|3|4|5|6|7|8|9$ dapat dinyatakan dengan $0|D(D|0)*$ artinya adalah pernyataan bahasa bilangan alam yang equivalent terhadap $0|(1|2|3|4|5|6|7|8|9)(1|2|3|4|5|6|7|8|9|0)*$. Beberapa contoh penggunaan regexps dalam bahasa pemrograman misalnya regexps **if** merupakan *concate* dari regexps **i** dan f. Nama variabel pada bahasa C terdiri dari huruf, digit, dan simbol garis bawah, penamaan variabel harus dimulai dengan huruf atau garis bawah.

$(r\|s)\|t = r\|s\|t = r\|(s\|t)$	$\|$ adalah asosiatif
$s\|t = t\|s$	$\|$ adalah komutatif
$s\|s = s$	$\|$ adalah idempoten
$s? = s\|E$	berdasarkan definisi
$(rs)t = rst = r(st)$	*concate* adalah asosiatif
$sE = s = Es$	E merupakan element netral untuk *concate*
$r(s\|t) = rs\|rt$	distributif *concatenation* $\|$
$(r\|s)t = rt\|st$	distributif *concatenation* $\|$
$(s^*)^* = s^*$	$*$ adalah idempoten
$s^*s^* = s^*$	0 atau lebih dua kali tetap 0 atau lebih
$ss^* = s^+ = s^*s$	berdasarkan definisi
$(s^+)^+ = s^+$	$^+$ adalah idempoten

terkadang juga digunakan beberapa aturan umum untuk menyatakan derivasi refleksif dan transitif.

$s \Rightarrow s$	Derivasi refleksif
$r \Rightarrow t$ if $r \Rightarrow s$ dan $s \Rightarrow t$	Derivasi transitif

1.4 Non deterministic finite automata (NDFA)

Non deterministic finite automaton terdiri dari hipunan state S. Salah satu dari state ini, $s_0 \in S$, disebut dengan *starting state* automaton, dan subset state $F \subseteq S$ merupakan *final state*. Sebagai tambahan, kita mempunyai himpunan T terhadap transisi. Masing-masing transisi t terhubung secara berpasangan terhadap state s_1 dan s_2 dan telah diberi label dengan sebuah simbol. Simbol ini berupa *character* c dan ε berasal dari alphabet Σ. Simbol epsilon mengindikasi sebuah *transisi-epsilon*. Transisi dari state s ke state t dengan sebuah simbol c yang terdapat pada transisi tersebut, dituliskan dengan $s^c t$. Berikut keterangan notasi grafik dari finite automata :

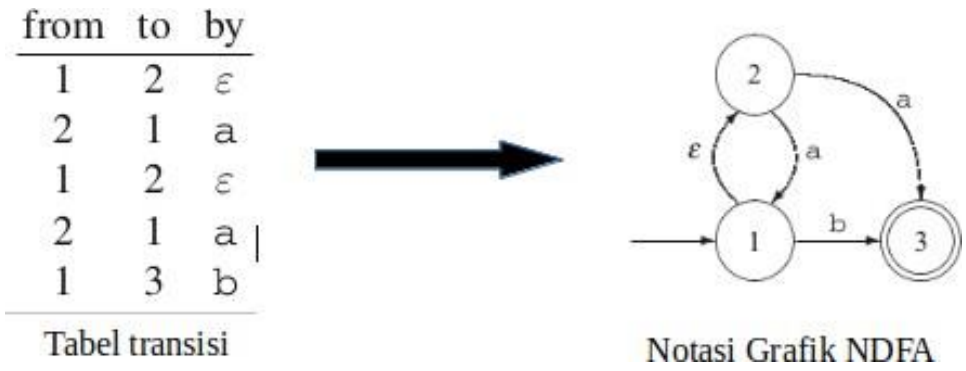
1. State berbentuk lingkaran, secara opsional berisi bilangan atau nama sebagai identifikasi state
2. Final state berbentuk dua lingkaran.
3. Tanda panah dari luar automata dan tidak memiliki label adalah inisial state
4. Tanda panah yang menghubungkan dua state disebut transisi.

5. Tanda panah yang diberi lebel dengan simbol epsilon (ε) adalah pemicu (*trigger*) transisi.

untuk memulai state kita dapat melakukan salah satu dari langkah berikut:

1. Mengikuti transisi epsilon untuk berpindah ke state selanjutnya
2. membaca karakter inputan kemudian mengikuti transisi yang sudah diberi label untuk berpindah ke state selanjutnya.

saat semua inputan string telah terbaca dan berkahir pada final state artinya automaton berhasil mengenali inputan bahasa string. Kita dapat menganalogikan proses ini dengan sebuah pemecahan masalah atau solusi dari sebuah labirin, dimana pada labirin tersebutterdapat simbol-simbol yang tertulis pada masing-masing koridor secara bersesuaian yang nanti mengarahkan kita untuk menemukan pintu keluar (string berhasil dikenali oleh NFA). Contoh NDFA mampu mengenali bahasa string aab yang dinyatakan dengan regexps $a^*(a\ b)$ dapat dilihat pada Gambar. 1.2, namun proses dari metode ini akan banyak memakan waktu karena segala kemungkinan perpindahan transisi dilakukan untuk membuat NFA menjadi sesuai dan efisien dalam mengenali bahasa string aab.

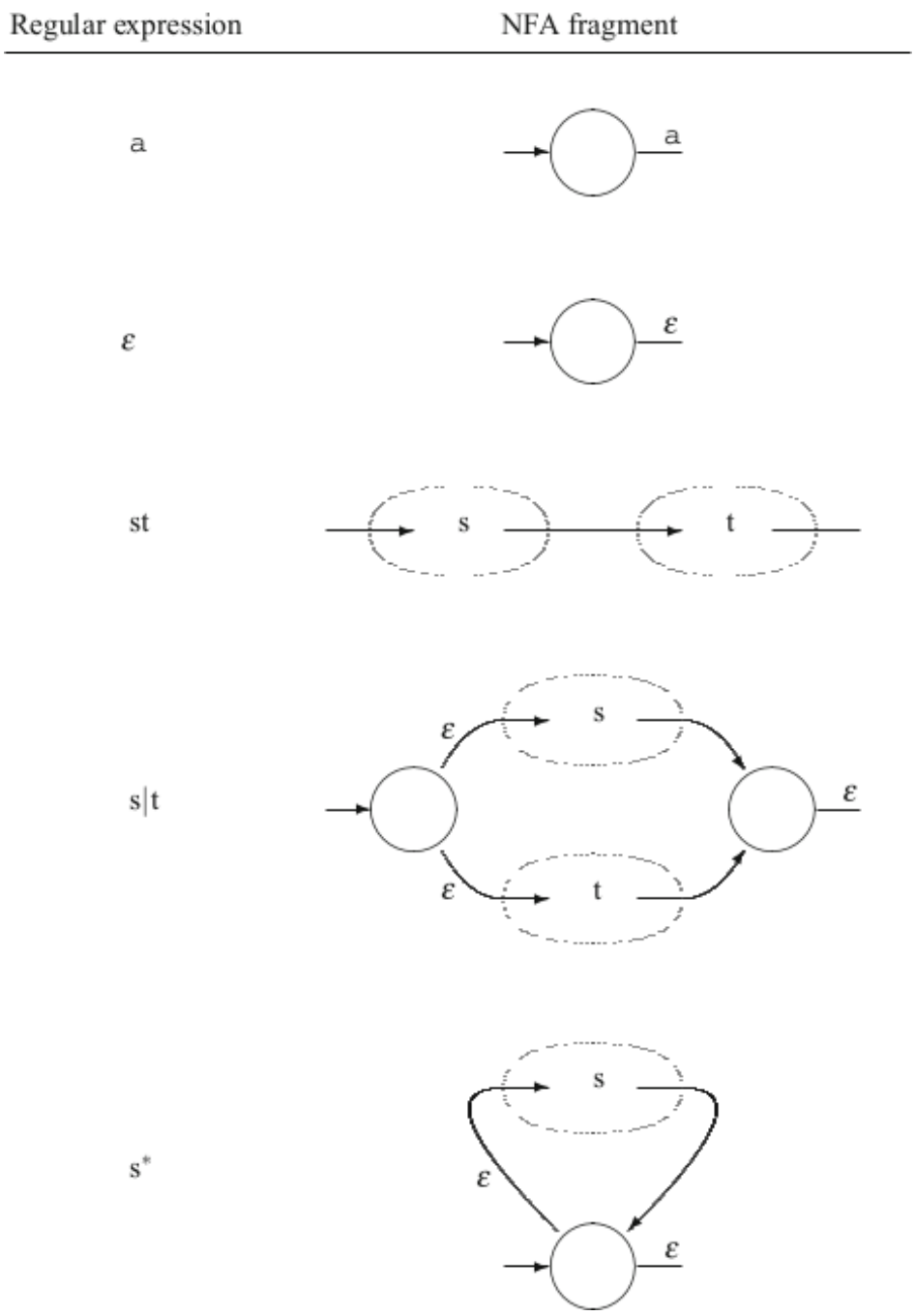


Gambar 1.2: Notasi Grafik NDFA

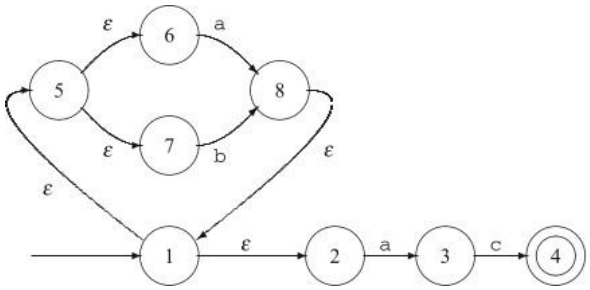
1.4.1 Konversi regular expression ke NFA

Sebuah potongan NFA terdiri dari beberapa state dengan transisinya dan sebagai tambahan dua transisi yang tidak lengkap: satu menunjuk kearah masuk potongan dan satunya lagi menunjuk kearah luar potongan. Setengah transisi yang mengarah masuk ke state tidak diberi label, namun setengah transisi yang mengarah keluar diberi label dengan simbol epsilon ε. Setengah transisi pada potongan NFA yang mengarah masuk dan keluar ini digunakan untuk menghubungkan antar potongan NFA atau merekatkan antar state. Kita dapat menggunakan tabel konversi 1.3 untuk membuat notasi grafik NFA terhadap regexps. Pembuatan notasi NFA mengikuti struktur dari regexps yaitu dengan pertama membuat potongan NFA untuk subexpression, dan menggabungkan potongan tesebut ke potongan NFA pada keseluruhan regexp. Bentul oval dengan garis putus-putus adalah potongan NFA *subexpression* dengan setengah transisi mengarah masuk kekiri dan setengah transisi mengarah keluar kekanan. Pada Potongan NFA *subexpression* yang bebentuk oval dengan garis putus-putus simbol tidak ditampilakan (simbol tersembunyi didalam oval). Ketika potongan NFA telah selesai dirancang terhadap keseluruhan regexps,rancangan ini diselesaikan dengan transisi mengarah keluar yang terhubung ke sebuah final state. Perlu diperhatikan bahwa NFA yang rancang menggunakan metode ini hanya memiliki satu final state yang ditambahkan pada akhir rancangan walaupun NFA itu sendiri memungkinkan untuk memiliki beberapa final state. Contoh NFA yang dirancang dengan cara ini untuk regexps $(a\ b)^*ac$ dapat dilihat pada Gambar 1.4. Pemberian nomor urutan pada tiap-tiap state dimulai pada cleen closure $(a\ b)^*$ kemudian berlanjut ke *concate* ac. Pada akhir

state berbentuk dua lingkaran yang menyatakan (*final state*).



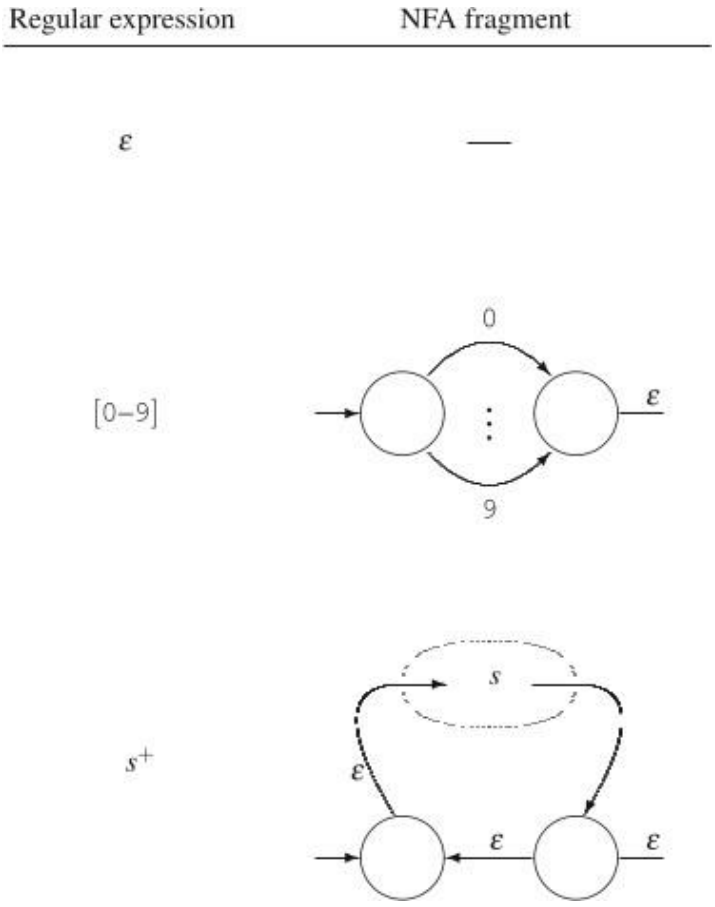
Gambar 1.3: Tabel Konversi



Gambar 1.4: NFA untuk regexps $(a|b)^*ac$

1.4.2 Optimisasi

Kita dapat menggunkan rancangan yang ada pada Gambar 1.3 untuk regexps apapun dengan memperluas semua stenography, sebagai contohnya:melakukan konversi s^+ ke ss^*, [0-9] ke $0|1|2|...|9$, $s?$ ke $s|\varepsilon$, dan seterusnya. Bagaimanapun, ini akan menghasilkan dalam NFA yang

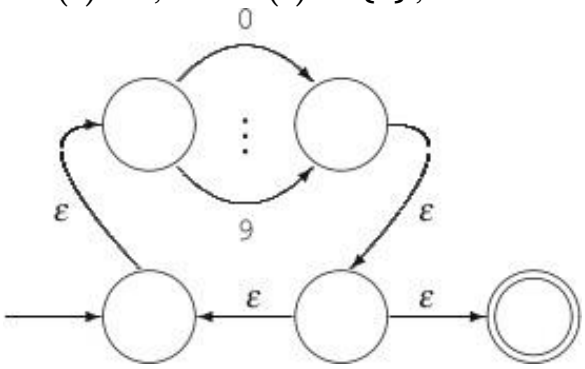


Gambar 1.5: Optimisasi NFA

sangat besar untuk beberapa *expression*. Jadi kita akan menggunakan beberapa rancangan optimasi terhadap *shorthand* yang ditunjukkan pada Gambar 1.5. Sebagai tambahan, kami menunjukkan rancangan alternatif untuk regexps ε. Pada rancangan ini tidak mengikuti seluruh formula yang digunakan seperti pada Gambar 1.3.Karena tidak terdapat dua setengah transisi (*half-transistion*). Sebaliknya, notasi garis-segment bermaksud melakukan identifikasi bahwa potongan NFA untuk ε hanya menghubungkan setengah transisi terhadap potongan NFA yang

digabungkan dengannya. Pada rancangan [0-9], maksud dari verikal elips untuk melakukan identifikasi bahwa ada transisi untuk masing-masing digit dalam [0-9]. Rancangan ini digeneralisasi dengan cara yang jelas terhadap himpunan-himpunan karater, sebagai contoh,[a-zA-Z0-9]. NFA untuk $[0\text{-}9]^+$ dapat dilihat pada Gambar 1.6. Perlu diperhatikan saat NFA ini dioptimalkan, NFA tersebut belum optimal. Kamu dapat (dalam beberapa cara berbeda) membuat sebuah NFA untuk bahasa ini hanya menggunakan dua state.Bahasa kosong (*empty language*) yaitu bahasa yang tidak mengandung string dapat dikenali dengan sebuah DFA (DFA apapun yang tidak memiliki final state dapat menerima bahasa ini), tetapi tidak dapat didefinisikan oleh regexps apapun menggunakan rancangan pada pembahasan 1.3. Konsekuensi logisnya, persamaan antara DFA dan regexps menjadi tidak lengkap. Untuk memperbaiki hal ini, diperkenalkan sebuah regexp baru ($\emptyset$) yang dinyatakan dengan bahasa $L(\emptyset) = \emptyset$.

Sebagai latihan-: (1). Jika $L(\emptyset) = \emptyset$, maka $L(\emptyset)^* = \{\varepsilon\}$, Jelaskan mengapa?

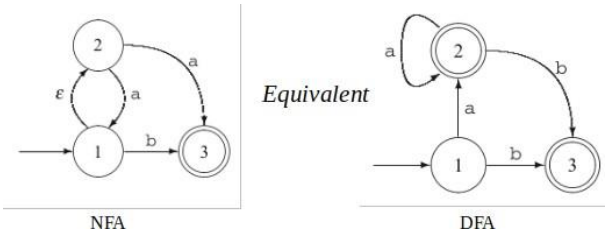


Gambar 1.6: NFA yang dioptimalkan untuk $[0\text{-}9]^+$

1.5 Deterministic Finite Automata (DFA)

DFA merupakan teori komputasi dan cabang dari ilmu komputer teoritis. DFA adalah Finite-state Machine atau mesin keadaan terbatas yang menerima atau menolak string dari simbol dan hanya menghasilkan perhitungan unik dari otomata untuk setiap string yang di masukan. DFAjuga dikenal sebagai Deterministic Finite Acceptor (DFA), Deterministic Finite State Machine (DFSM), atau Deterministic Finite State Automaton (DFSA). DFA didefinisikan sebagai sebuah konsep matematika abstrak digunakan untuk menyelesaikan permasalahan pada penganalisa leksikal. DFA adalah kasus khusus pada NFA yang mematuhi sejumlah batasan berikut :

- Tidak terdapat transisi epsilon.
- Tidak terdapat dua transisi luar berlabel identik pada state yang sama.
- Gambar 1.7 menunjukkan DFA yang ekivalen terhadap NFA pada Gambar 1.4.



Gambar 1.7: NFA Ekivalen DFA

pada gambar DFA tersebut tidak terdapat beberapa pilihan ke state berikutnya seperti halnya pada NFA yang memiliki pilihan state yaitu state satu dengan transisi input simbol ε ke state dua dan dari state dua dengan transisi input simbol a ke state satu. Pada DFA State dan transisi (*input symbol*) berikutnya ditentukan secara unik, dan karena inilah automata disebut sebagai *deterministic*. Sebagian besar penganalisa leksikal menggunakan DFA dalam pemrosesan bahasa.

Hubungan transisi pada DFA adalah sebuah fungsi parsial yang dapat dituliskan sebagai sebuah fungsi $move(s, c)$ dimana pada state s terdapat transisi dengan simbol input c menuju ke state selanjutnya. Namun jika pada state s tidak terdapat state selanjutnya maka $move(s, c)$ merupakan fungsi tak tentu. DFA yang dihasilkan dapat lebih besar (secara exponensial) dibandingkan dengan NFA. Dalam prakteknya yaitu saat menyatakan token pada bahasa pemrograman, peningkatatan dalam ukuran biasanya rata-rata. Itulah menjadi sebab kebanyakan penganalisa leksikal didasarkan pada DFA.

Latihan !!

1. Buatlah digram transisi DFA (deterministic finite automata) untuk mengenali tanda kurung yang bersesuaian dengan jumlah kedalaman bersarang 3. sebagai contoh : ε,(), (()(())) atau (()())()() tetapi bukan seperti (((()))) atau seperti (()(()(()))).
2. Buatlah digram transisi DFA untuk mengenali himpunan string {a. b} dengan ketentuan transisi b sebanyak 3 kali (dan ada sejumlah transisi a).

1.6 Konversi NFA ke DFA

Ide dari konversi ini adalah bahwa himpunan state yang berbeda pada NFA menjadi state tunggal berbeda pada DFA yang akan kita bangun. Kita akan menetapkan *epsilon closure* sebagai perluasan himpunan state NFA yang dapat dicapai dari state tersebut menggunakan sejumlah transisi epsilon. Definisi secara formal-nya adalah : $X = M \cup \{t | s \in X \text{ dan } s^{\varepsilon}t \in T\}$, dimana T adalah himpunan transisi NFA, M adalah himpunan state NFA dan X adalah solusi dari himpunan persamaan.

1.6.1 Penyelesaian Himpunan Persamaan

Secara umum, himpunan persamaan nilai tunggal atas variabel X memiliki bentuk $X = F(X)$, dimana F adalah sebuah fungsi dari himpunan ke himpunan. Untuk memasukkan persamaan $X = M \cup \{t | s \in X \text{ dan } s^{\varepsilon}t \in T\}$ kedalam bentuk $X = F(X)$ untuk sebuah fungsi f, kita beri definisi F_M menjadi $F_M(X) = M \cup \{t | s \in X \text{ dan } s^{\varepsilon}t \in T\}$.

Mungkin akan ada beberapa solusi untuk persamaan $X = F_M(X)$. Sebagai contoh jika NFA memiliki state berpasangan yang masing-masing saling terhubung dengan transisi epsilon, tambahkan state berpasangan ini ke solusi yang akan membuat pasangan state sebagai solusi baru.

F_M memiliki properti yang pada dasarnya adalah metode solusi kita: jika $X \subseteq Y$ maka $F_M(X) \subseteq F_M(Y)$. Kita dapat katakan bahwa F_M adalah *monotonic*. Ketika kita memiliki sebuah persamaan berupa $X = F(X)$ dan F adalah monotonic, setidaknya kita dapat menemukan solusi persamaan dengan cara berikut: Pertama kali kita tebak bahwa solusi berupahimpunan kosong $\emptyset$ dan cek untuk melihat apakah itu benar: kita dapat bandingkan $\emptyset$ dengan $F(\emptyset)$. Jika dalam hal ini adalah sama (*equal*), kita selesai dan adalah solusinya. Jika tidak, kita gunakan properti-properti berikut:

- setidaknya solusi S memenuhi persamaan $S = F(S)$
- $\emptyset \subseteq S$ menyatakan secara tidak langsung (*implies*) bahwa $F(\emptyset) \subseteq F(S)$

untk menyimpulkan bahwa $F(\emptyset) \subseteq S$. Sebagai konsekuensi logis, $F(\emptyset)$ adalah S atau subset dari S, dengan demikian kita dapat menggunakan itu sebagai tebakan baru.Dalam bentuk rantai adalah $\emptyset \subseteq F(\emptyset) \subseteq F(F(\emptyset)) \subseteq \ldots$ Jika rentetan titik element apapun identik dengan sebelumnya, kita memiliki titik tetap (*fixed-point*) yaitu himpunan S sedemikian rupa $S = F(S)$. Titik tetap pada rentetan ini setidaknya adalah solusi dari persamaan atau istilahnya adalah *himpunan inklusi*. Karena disini terjadi perulangan fungsi sampai mecapai titik tetap, kita sebut proses ini sebagai perulangan titik tetap (*fixed-point iteration*).

Kita akan gunakan metode ini untuk mengkalkulasikan epsilon-closure terhadap himpunan {1} berkenaan dengan NFA yang ditunjukkan pada Gambar 1.4. Karena kita ingin mencari $\varepsilon\text{—}closure(\{1\})$, $M = \{1\}$ maka $F_M = F_{(1)}$. Kita mulai dengan malakukan penebakan bahwa X adalah himpunan kosong:

$$\begin{aligned} F_{\{1\}}(\emptyset) &= \{1\} \cup \{t | s \in \{\emptyset\} \text{ dan } s^{\varepsilon}t \in T\} \\ &= \{1\} \end{aligned}$$

Karena $\emptyset = \{1\}$, perulangan dilanjutkan:

$$\begin{aligned} F\{1\}(F\{1\}(\emptyset)) &= F_{\{1\}}(\{1\}) \\ &= \{1\} \cup \{t | s \in \{1\} \text{ dan } s^{\varepsilon}t \in T\} \\ &= \{1\} \cup \{2,5\} = \{1,2,5\} \end{aligned}$$

$$\begin{aligned} F_{\{1\}}(F_{\{1\}}(F_{\{1\}}(\emptyset))) &= F_{\{1\}}(\{1,2,5\}) \\ &= \{1\} \cup \{t | s \in \{1,2,5\} \text{ dan } s^{\varepsilon}t \in T\} \\ &= \{1\} \cup \{2,5,6,7\} = \{1,2,5,6,7\} \end{aligned}$$

$$\begin{aligned} F_{\{1\}}(F_{\{1\}}(F_{\{1\}}(F_{\{1\}}(\emptyset)))) &= F_{\{1\}}(\{1,2,5,6,7\}) \\ &= \{1\} \cup \{t | s \in \{1,2,5,6,7\} \text{ dan } s^{\varepsilon}t \in T\} \\ &= \{1\} \cup \{2,5,6,7\} = \{1,2,5,6,7\} \end{aligned}$$

Sekarang kita telah mencapai titik tetap (*fixed-point*) dan solusi kita telah didapatkan, Secara logis kita dapat menyimpulkan bahwa ε-*closure*$(1) = \{1, 2, 5, 6, 7\}$. Kita dapat membuat sebuah titip tetap yang dioptimalkan dengan mengeksploitasi tersebut tidak hanya sebagai monotonic,tetapi juga sebagai distributif: $F(X \cup Y) = F(X) \cup F(Y)$. Selama perulangan element-element ditambahkan ke himpunan, dalam perulangan berikutnya hanya diperlukan mengkalkulasi F untuk element-element baru dan hasilnya tambahkan ke himpunan. Dalam contoh diatas, kita dapatkan

$$\begin{aligned} F_{\{1\}}(\emptyset) &= \{1\} \cup \{t | s \in \{\emptyset\} \text{ dan } s^{\varepsilon}t \in T\} \\ &= \{1\} \end{aligned}$$

$$\begin{aligned} F_{\{1\}}(\{1\}) &= \{1\} \cup \{t | s \in \{\emptyset\} \text{ dan } s^{\varepsilon}t \in T\} \\ &= \{1\} \cup \{2, 5\} = \{1, 2, 5\} \end{aligned}$$

$$\begin{aligned} F_{\{1\}}(\{1, 2, 5\}) &= F_{\{1\}}(\{1\}) \cup F_{\{1\}}(\{2, 5\}) \\ &= \{1, 2, 5\} \cup (\{1\} \cup \{t | s \in \{\emptyset\} \text{ dan } s^{\varepsilon}t \in T\}) \\ &= \{1, 2, 5\} \cup (\{1\} \cup \{6, 7\}) = \{1, 2, 5, 6, 7\} \end{aligned}$$

$$\begin{aligned} F_{\{1\}}(\{1, 2, 5\}) &= F_{\{1\}}(\{1\}) \cup F_{\{1\}}(\{2, 5\}) \\ &= \{1, 2, 5\} \cup (\{1\} \cup \{t | s \in \{\emptyset\} \text{ dan } s^{\varepsilon}t \in T\}) \\ &= \{1, 2, 5\} \cup (\{1\} \cup \{6, 7\}) = \{1, 2, 5, 6, 7\} \end{aligned}$$

$$\begin{aligned} F_{\{1\}}(\{1, 2, 5, 6, 7\}) &= F_{\{1\}}(\{1, 2, 5\}) \cup F_{\{1\}}(\{6, 7\}) \\ &= \{1, 2, 5, 6, 7\} \cup (\{1\} \cup \{t | s \in \{6, 7\} \text{ dan } s^{\varepsilon}t \in T\}) \\ &= \{1, 2, 5, 6, 7\} \cup (\{1\} \cup \emptyset) = \{1, 2, 5, 6, 7\} \end{aligned}$$

1.6.2 Algoritma Subset Construction

Diasumsikan sebuah NFA N dengan state S, permulaan state $s_0 \in S$, final state $F \subseteq S$, transisi T , dan alpabet Σ, kita akan membuat ekivalen DFA D dengan state S^I, permulaan state s^I_0, final state F^I, dan fungsi transisi *move* dengan:

$$\begin{aligned} s^I_0 &= \varepsilon\text{-}closure(\{s_0\}) \\ move(s^I, c) &= \varepsilon\text{-}closure(\{t | s \in s^I \text{ dan } s^c t \in T\}) \\ S^I &= \{s^I_0\} \cup \{move(s^I, c) | s^I \in S^I, c \in \Sigma\} \\ F^I &= \{s^I \in S^I | s^I \cap F \neq \emptyset\} \end{aligned}$$

DFA dan NFA menggunakan alfabet Σ yang sama.

Penjelasan:

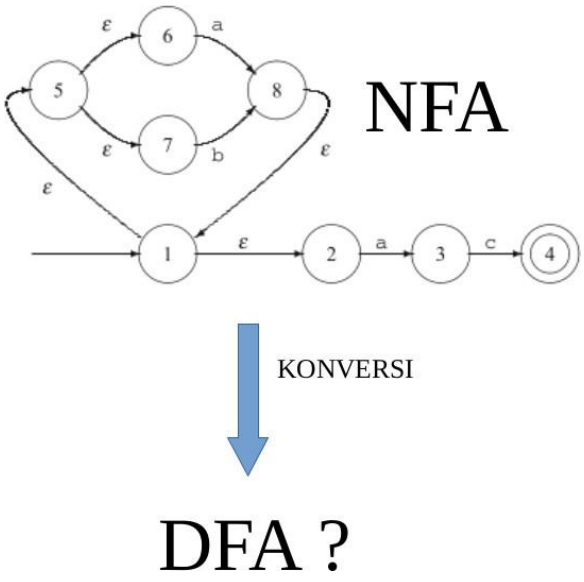
- State awal s^I_0 DFA adalah epsilon-closure pada himpunan state awal s_0 NFA, yaitu state yang dapat dicapai dari s_0 hanya dengan transisi-epsilon.

- Trasisi dengan simbol c pada DFA diselesaikan dengan mencari himpunan state s NFA yang terdiri dari state DFA, menggabungkan himpunan state yang diperoleh dari NFA, dan akhirnya mentutup transisi-epsilon pada NFA.

- Himpunan state S^I pada DFA adalah himpunan state DFA yang dicapai dari s^I_0 menggunakan fungsi *move*. S^I didefinisikan sebagai himpunan persamaan yang dapat diselesaikan dengan cara seperti yang sudah dijelaskan sebelumnya pada bagian 1.6.1.

- s^I pada DFA adalah final state jika ada satu (setidaknya) state NFA dalam s^I merupakan final state.

Sebagai contoh, kita akan mencoba mengkonversikan NFA berikut 1.8 ke DFA. Inisial state



Gambar 1.8: NFA ke DFA

DFA adalah $\varepsilon\text{-}closure(\{1\})$, yang baru saja selesai kita kalkulasikan menjadi $s^I_0 = \{1, 2, 5, 6, 7\}$. Sehingga kita peroleh fungsi *move* sebagai berikut:

$$\begin{aligned} move(s^I_0, a) &= \varepsilon\text{-}closure(\{t | s \in \{1,2,5,6,7\} \text{ dan } s^a t \in T\}) \\ &= \varepsilon\text{-}closure(\{3, 8\}) \\ &= \{3, 8, 1, 2, 5, 6, 7\} \\ &= s^I_1 \end{aligned}$$

$$\begin{aligned} move(s^I_0, b) &= \varepsilon\text{-}closure(\{t | s \in \{1,2,5,6,7\} \text{ dan } s^b t \in T\}) \\ &= \varepsilon\text{-}closure(\{8\}) \\ &= \{8, 1, 2, 5, 6, 7\} \\ &= s^I_2 \end{aligned}$$

$$\begin{aligned} move(s^I_0, c) &= \varepsilon\text{-}closure(\{t | s \in \{1,2,5,6,7\} \text{ dan } s^c t \in T\}) \\ &= \varepsilon\text{-}closure(\{\}) \\ &= \{\} \end{aligned}$$

$$move(s_1^I, a) = \varepsilon\text{-}closure(\{t | s \in \{3, 8, 1, 2, 5, 6, 7\} \text{ dan } s^a t \in T\})$$
$$= \varepsilon\text{-}closure(\{3, 8\})$$
$$= \{3, 8, 1, 2, 5, 6, 7\}$$
$$= s_1^I$$

$$move(s_1^I, b) = \varepsilon\text{-}closure(\{t | s \in \{3, 8, 1, 2, 5, 6, 7\} \text{ dan } s^b t \in T\})$$
$$= \varepsilon\text{-}closure(\{8\})$$
$$= \{8, 1, 2, 5, 6, 7\}$$
$$= s_2^I$$

$$move(s_1^I, c) = \varepsilon\text{-}closure(\{t | s \in \{3, 8, 1, 2, 5, 6, 7\} \text{ dan } s^c t \in T\})$$
$$= \varepsilon\text{-}closure(\{4\})$$
$$= \{4\}$$
$$= s_3^I$$

$$move(s_2^I, a) = \varepsilon\text{-}closure(\{t | s \in \{8, 1, 2, 5, 6, 7\} \text{ dan } s^a t \in T\})$$
$$= \varepsilon\text{-}closure(\{3, 8\})$$
$$= \{3, 8, 1, 2, 5, 6, 7\}$$
$$= s_1^I$$

$$move(s_2^I, b) = \varepsilon\text{-}closure(\{t | s \in \{8, 1, 2, 5, 6, 7\} \text{ dan } s^b t \in T\})$$
$$= \varepsilon\text{-}closure(\{8\})$$
$$= \{8, 1, 2, 5, 6, 7\}$$
$$= s_2^I$$

$$move(s_2^I, c) = \varepsilon\text{-}closure(\{t | s \in \{8, 1, 2, 5, 6, 7\} \text{ dan } s^c t \in T\})$$
$$= \varepsilon\text{-}closure(\{\})$$
$$= \{\}$$

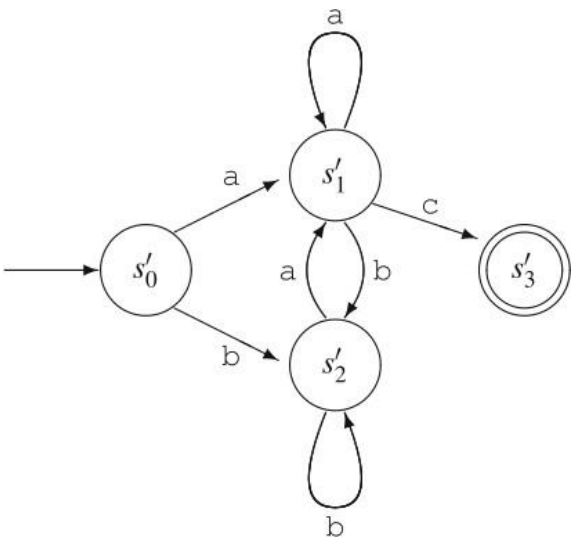
$$move(s_3^I, a) = \varepsilon\text{-}closure(\{t | s \in \{4\} \text{ dan } s^a t \in T\})$$
$$= \varepsilon\text{-}closure(\{\})$$
$$= \{\}$$

$$move(s_3^I, b) = \varepsilon\text{-}closure(\{t | s \in \{4\} \text{ dan } s^b t \in T\})$$
$$= \varepsilon\text{-}closure(\{\})$$
$$= \{\}$$

$$move(s_3^I, c) = \varepsilon\text{-}closure(\{t | s \in \{4\} \text{ dan } s^c t \in T\})$$

$$= \varepsilon\text{-}closure(\{\})$$
$$= \{\}$$

hanya s'_3 berisi final state 4 NFA, dengan demikian state inilah yang dijadikan final state pada DFA tersebut. Gambar 1.9 menunjukkan DFA hasil konversi NFA.



Gambar 1.9: DFA hasil konversi NFA

Latihan !!

Terdapat regular expression (regexps) $a^*(a|b)$:

1. Buatlah digram transisi NFA dari regexps tersebut menggunakan metode konversi regexps ke NFA
2. Konversi NFA yang telah dibuat tadi ke DFA menggunakan algoritme *subset construction*.

1.6.3 Meminimalkan DFA

DFA pada Gambar 1.9 belum dalam bentuk minimal, walaupun hanya memiliki empat state. Dengan mudah dapat kita lihat s'_0 dan s'_2 adalah state DFA dengan transisi yang identik dan tidak saling menerima. Oleh karenanya kita dapat memperpendek (*collapse*) state stersebut menjadi state tunggal sehingga diperoleh DFA dengan tiga state. DFA yang telah dibuat dari regular expression melalui NFA sering dalam bentuk non-minimal.

Ada beberapa aturan yang dapat digunakan dalam meminimalkan DFA, yaitu:

- Final state tidak ekivalen terhadap yang bukan final state.
- Jika dua state s_1 dan s_2 memiliki transisi dengan simbol c yang sama ke state t_1 dan t_2 maka dapat dikatan bahwa s_1 dan s_2 tidak ekivalent. Ini juga berlaku jika hanya salah satu state s_1 atau s_2 terdapat sebuah simbol c pada transisi yang telah ditentukan.

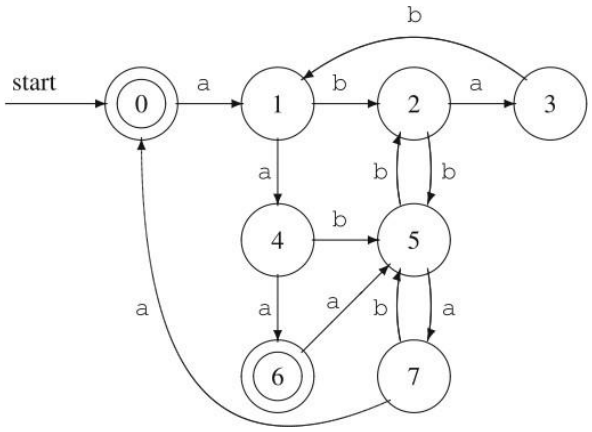
Kita dapat membawa aturan ini kedalam algoritme berikut: State awal pada DFA minimal adalah

Algoritme DFA Minimisasi

1: Inisialisasi kedalam dua grup, final state dan F dan non-final state S (S/F)
2: Pilih sembarang grup G yang tidak diberikan tanda dan kita cek apakah grup G tersebut konsisten atau tidak, jika ya, berikan penanda, namun jika tidak, kita partisi kedalam subgrup maksimal konsisten dan mengganti G dengan subgrup tersebut.
3: Jika semua grup sudah bertanda, proses selesai, dan sisa grup adalah state-state DFA minimal. Jika tidak, kembali ke step 2.

grup yang berisi state awal orijinal, dan grup apapun yang berisi final state adalah final state untuk DFA minimal.

Kita akan menggunakan DFA pada Gambar 1.10 sebagai contoh minimisasi.



Gambar 1.10: Non-minimal DFA

1. **Task 1:**
   Inisialisasi state-state kedalam 2 grup
   $S/F = \{1, 2, 3, 4, 5, 7\} / \{0, 6\} = \{1, 2, 3, 4, 5, 7\}$
   $G1 = \{0, 6\}$
   $G2 = \{1, 2, 3, 4, 5, 7\}$Task 2:
   Pilih grup yang belum bertanda (G1), cek konsistensi dengan table transisi, jika konsisten berikan tanda pada grup.

G1	a	b
0	G2	-
6	G2	-

2. **Task 2:**
   Pilih sisa yang belum bertanda yaitu (grup G2) dan buat tabel transisi.

G2	**a**	**b**
1	G2	G2
2	G2	G2
3	-	G2

4	G1	G2
5	G2	G2
7	G1	G2

group G2 tidak konsisten karena ditandai dengan subgrup yang berbeda-beda yaitu state $\{1, 2, 5\}$ state $3 = \{3\}$ state $4, 7$. Jika tidak konsisten kita lakukan partisi grup menjadi subgrup berdasarkan konsistensi state pada subgrup tersebut. Dengan demikian diperolehsubgrup baru sebagai beriku:

Subgrup baru

G1 = {0, 6}
G3 = {1, 2, 5}
G4 = {3}
G5 = {4, 7}

3. **Task 3:**
Pilih G3 sebagai urutan selanjutnya :

G3	a	b
1	G5	G3
2	G4	G3
5	G5	G3

G3 tidak konsisten, kembali lakukan partisi dan diperoleh subgrup baru

Subgrup baru

G1 = {0, 6}
G4 = {3}
G5 = {4, 7}
G6 = {1, 5}
G7 = {2}

4. **Task 4:**
Pilih urutan subgrup G4:

G4ab
3-G6

5. **Task 5:**
   Pilih G5 sebagai urutan selanjutnya :

G5	a	b
4	G1	G6
7	G1	G6

   G5 konsisten , berikan penanda pada subgrup tesebut kemudian lanjutkan memilih urutan subgrup berikutnya, yaitu G6 :

6. **Task 6:**
   Pilih subgroup G6, dan buat tabel transisi.

G6	a	b
1	G5	G7
5	G5	G7

   G6 konsisten, jadi sisa subgrup yang belum bertanda adalah G1 yang merupakannon-singleton, berikan tanda pada G1 non-singletone.

   -r G1 non-singleton

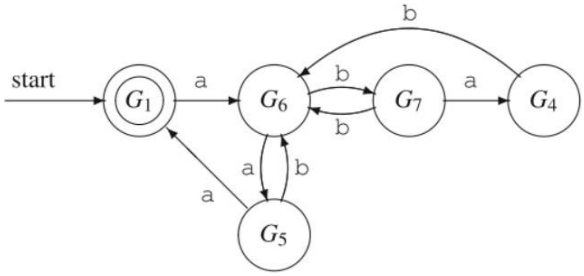
G1	a	b
0	G6	-
6	G6	-

7. **Task 7:**
   Pilih subgroup G7, dan buat tabel transisi.

G7	a	b
2	G4	G6

   Semua subgroup sudah bertanda (G1, G5, G6), kecuali (G7, G4) yang merupakan *singleton*.

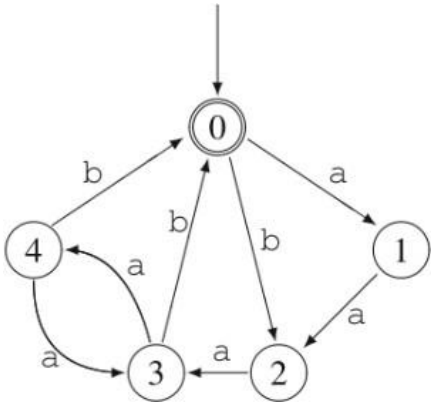
Dalam membangun sebuah transisi diagram minimal DFA dimulai dengan subgrup yang sudah bertanda dan lanjutkan pada subgrup (state) singleton. Hasil dari proses minimisasi ini dapat dilihat pada Gambar 1.11 .



Gambar 1.11: Minimal DFA

Latihan !

1. Minimalkan DFA berikut 1.12



Gambar 1.12: non-minimal DFA

1.6.4 Dead State

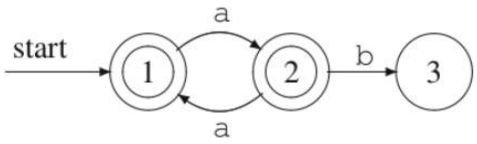
Dead state adalah sebuah state yang tidak berakhir pada final state, dead state tidak tejadi pada DFA yang telah dibuat dari NFA tanpa dead state. NFA yang memiliki dead state tidak dapat dikonversi menggunakan regexps. Jika terdapat DFA yang tidak diketahui asal usulnya, kemungkinan DFA tersebut memiliki dead state dan transisi yang tidak terdefinisi. Jika kita memasukan DFA yang memiliki dead state kedalam algoritma minimal DFA, konsekuesinya akan mempartisi DFA tersebut yang mana sebenar tidak diperlukan serta merubah groupnya menjadi tidak konsisten.

Terdapat 2 solusi untuk menangani permasalah ini :

3. Pastikah bahwa tidak terdapat dead states, ini diapat dipastikan dengan invariant, seperti yang terjadi pada DFA yang telah dihasilkan dari regexps dengan metode yang telah ditunjukkan pada pembahasan sebelumnya, atau menghapua dead state tersebut secara explisit sebelum diminimalkan. Dead state dapat dengan mudah ditemaukan dengan cara mengalisis pencapaian transisi state akhir pada graph searah. (jika kamu tidak dapat

mencapai sebuah final state dari state s, maka s adalah sebuah dead state). pada contoh dibawah ini, state 3 merupakan dead state dan dapat dihapus (termasuk transisinya). Hal ini membuat state 1 dan 2 tetap pada grup yang sama selama proses meminimalkan.

4. Pastikan bahwa tidak terdapat transisi yang tidak terdefinisi. Ini bisa didapatkan dengan menambahkan sebuah dead state baru (yang memiliki transisi terhadap itu sendiri padasemua simbol) dan mengganti semua transisi yang tidak terdefinisi dengan transisi ke dead state ini. Setealah minimisasi, group yang memiliki penambahan dead state akan tetap memiliki semua dead state dari DFA aslinya. Sekarang group ini dapat dihapus dari minimal DFA (yang sekali lagi akan memiliki transisi yang tidak didefinisikan). Pada contoh yang dintunjukkan pada Gambar 1.13, sebuah state baru state 4 (bukan final state) telah ditambahkan. State 1 memiliki transisi ke state 4 dengan simbol b, state 3 memilik sebuah transisi ke state 4 dengan simbal a dan b. Dan state 4 memilik transisi kedirinya sendiri, a dan b. Setelah minimisasi, state 1 dan state 2 menjadi tergabung. Seperti state 3 dan 4, karena state 4 adalah dead state, semua state digabungkan. Sehingga kita dapaat menghapus gabungan state 3 dan 4 dari hasil automaton yang diminimalkan.



Gambar 1.13: DFA-Dead State

1.7 Tokens

Leksikal tokens atau tokens merupakan sebuah rentetan dari karakter-karakter yang dapat diperlakukan sebagai unit tata bahasa (grammar) pada bahasa pemrograman. Contoh :

- Tipe token (“id” {a-z, A-Z, 0-9}, “num” {0-9}, “real”, . . .)
- Token -pemberian tanda baca ((“if”, “void”, “return”, “begin”, “call”, “const”, “do”, “end”, “if”, “odd”, “procedure”, “then”, “var”, “while”. . .))
- Alphabetic tokens (“keywords”)
- Symbol tokens (‘+’, ‘–’, ‘*’, ‘/’, ‘=’, ‘(’, ‘)’, ‘,’, ‘;’, ‘.’, ‘:=’, ‘<’, ‘<=’, ‘< >’, ‘>’, ‘>=’.....)

yang bukan token, Contoh: *Comments, preprocessor directive, macros, blanks, tabs, newline, . . .*

Contoh penggunaan atribut token:

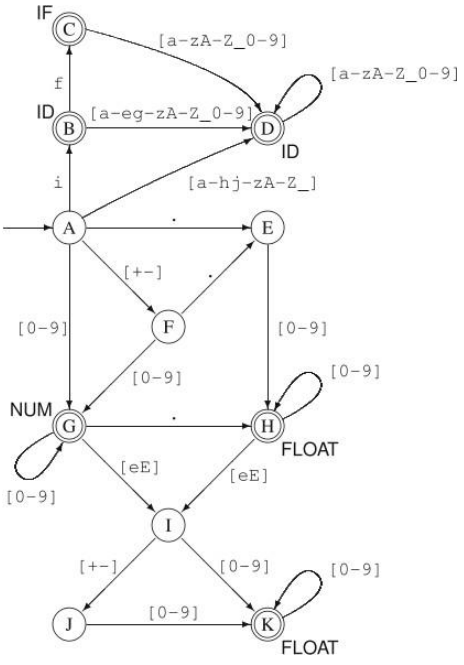
- c = a + b
  TOKENS adalah ‘c’, ‘=’, ‘a’, ‘+’, ‘b’
  TOKENS sebagai tiga attribut
  <id , pointer to symbol table entry for c>
  <EQ, >
  <id , pointer to symbol table entry for a>

<PLUS, >
<id , pointer to symbol table entry for b>

- this_book_by_vivek_sharma = (project - manager)
  TOKENS adalah 'this_book_by_vivek_sharma', '=', '(', 'project', '-' , 'manager'
  TOKENS dengan tiga attribut
  <id , pointer to symbol table entry for this_book_by_vivek_sharma><assign_op , >
  < *LP,* >
  <id pointer to symbol table entry for project>
  <MINUS , >
  <id pointer to symbol table entry for manager>
  <RP , >

1.8 Lex Generator

Sejauh ini kita telah mempalajari cara mengkonversi regular expression yang dinyatakan kedalam DFA melalui NFA. Selanjutnya kita akan mecoba mempelajari sebuah tool yang disebut dengan *Lex compiler*. *Lex compiler* mengubah pola inputan kedalam sebuah transisi digram dan menghasilkan code dalam sebuah file *lex.yy.c* dibalik layar dari proses *Lex compiler* dapat dilihat pada Gambar 1.14 berikut: sebagai contoh misal lexing dari string if17, kita malaui state



Gambar 1.14: Kombinasi DFA untuk bebera token

A, B, C, dan berakhir pada state D sebagai final state dari transisi tersebut.

1.8.1 Install Lex dan YACC pada Sistem Operasi Windows

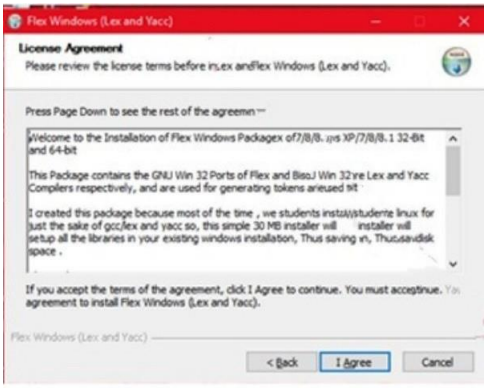
Lex yang merupakan Lexical Analyzer Generator bermaksud men-scan kode yang ditulis. Sementara pasangannya Yacc (yet another compilers compiler) berfungsi melakukan parsing berdasarkan grammar. Dengan lex and yacc ini kita dapat membuat kompiler sendiri yang akan memerintahkan komputer menjalankan instruksi sesuai keinginan kita. Misalnya jika dalam bahasa pemrograman rata-rata menggunakan kata "if-else", maka kita bisa saja menggantinya menjadi "jika-maka" setelah mengaturnya dengan Lex and Yacc ini. Silahkan unduh sourcecode nya lewat Google, lalu instal.

1.8.2 Cara Install

Lex and Yacc dapat berjalan di banyak platform. Untuk mudahnya di sini akan kita coba pada Windows 10. Setelah memperoleh kode sumbernya, klik ganda hingga muncul informasi bahwa akan diinstal Lex and Yacc 1.15.

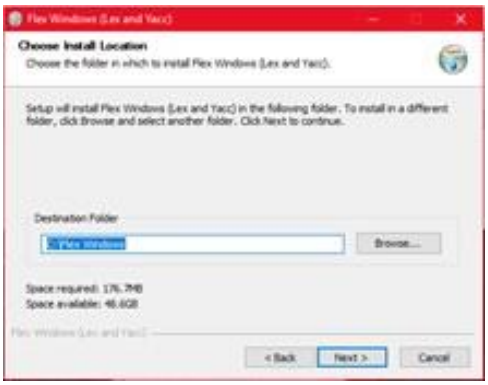
Gambar 1.15: Form Informasi

Tekan saja **Next>** untuk lanjut ke menu persetujuan. Tekan saja Agree 1.16.



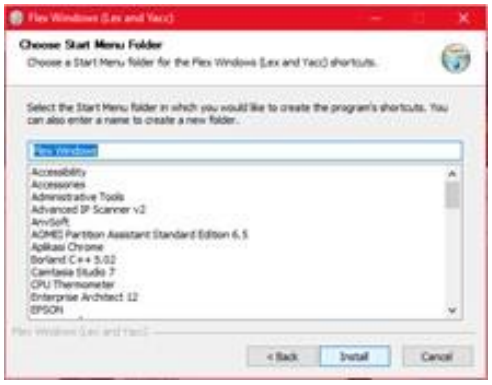
Gambar 1.16: Persetujuan Lisensi

Berikutnya, Lex and Yacc akan meminta folder tempat program diinstal. Arahkan sesuai dengan keinginan, atau biarkan secara default dengan menekan **Next>** 1.17.



Gambar 1.17: Pilih lokasi install

Berikutnya instalasi menanyakan lokasi menu folder Lex and Yacc nantinya. Biarkan secara default saja 1.18.



Gambar 1.18: Lokasi menu foldel Lex dan Yacc

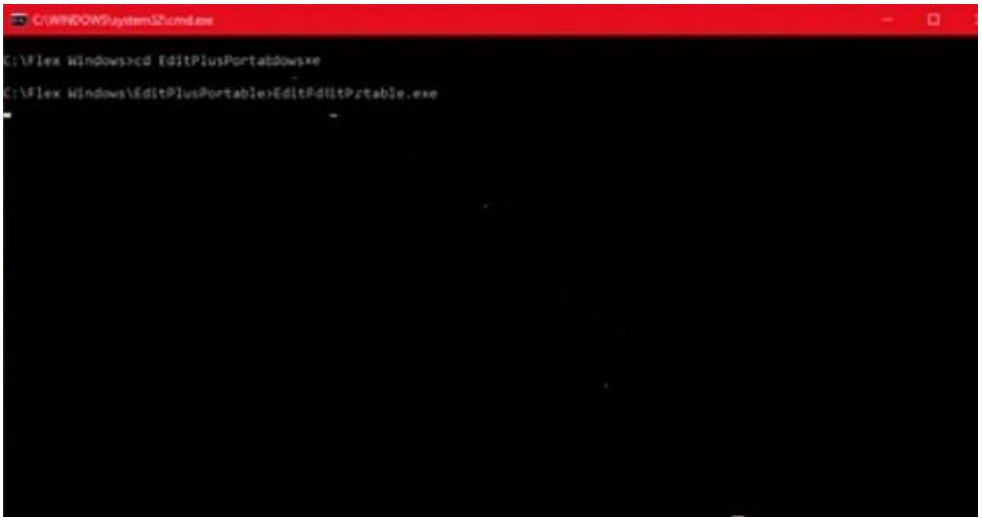
Akhirnya setelah menekan tombol **Install** maka proses instalasi akan berjalan hingga selesai. Hanya butuh satu hingga beberapa menit 1.19.

Gambar 1.19: Proses installasi

Pastikan instalasi lengkap dan tombol Finish muncul. Centang jika ingin langsung menjalankan aplikasi ini 1.20.

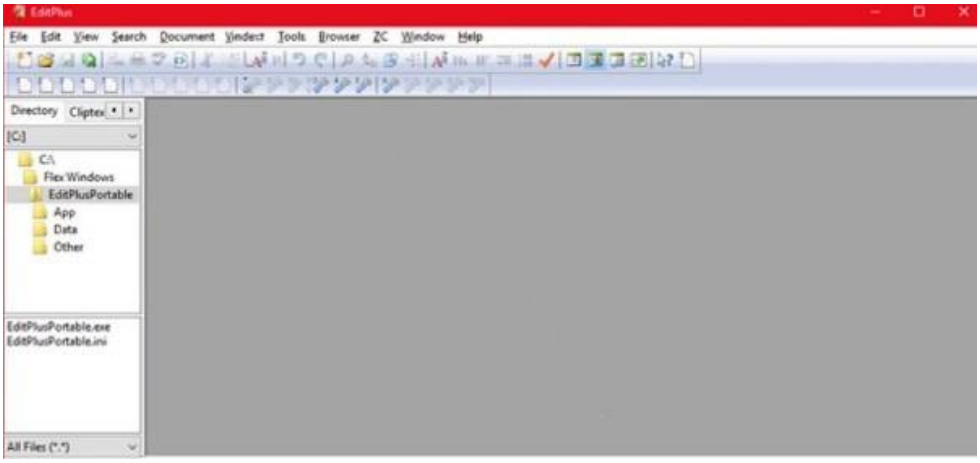
Gambar 1.20: Installasi Lengkap

Akan muncul konsol dos (CMD) ketika aplikasi ini berjalan. Tunggu sesaat 1.21.



Gambar 1.21: Form konsol dos

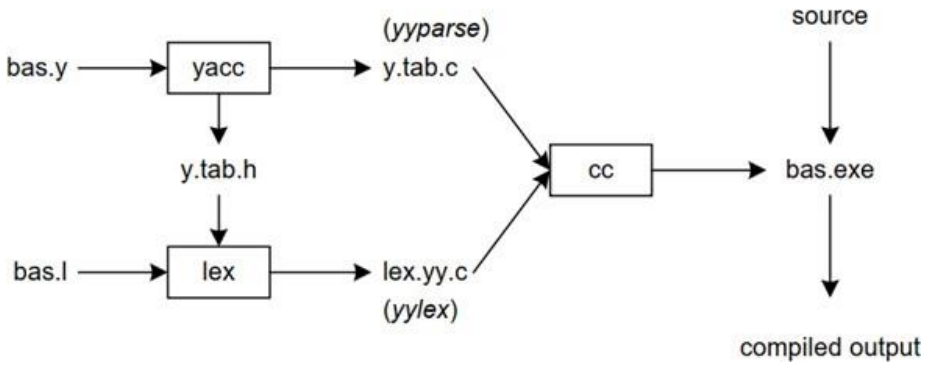
Jika sudah muncul tampilan seperti gambar di bawah berarti Lex and Yacc sudah siap untuk digunakan. Silahkan cari tatacara penggunaannya, bahkan ada juga yang menyediakan sampel programnya 1.22.



Gambar 1.22: Form Lex dan Yacc

1.8.3 Aplikasi Pembuat Bahasa

Memang untuk bisa menggunakan aplikasi-aplikasi pembuat bahasa pemrograman diperlukan teori khusus yaitu teori otomata dan bahasa (grammar). Tanpa hal itu dijamin kebingungan menggunakannya. Biasanya aplikasi yang digunakan untuk membuat bahasa adalah aplikasi yang berbasis C++ atau Java. Salah satu yang terkenal dan banyak dijadikan bahan praktek mata kuliah teknik kompilasi adalah Lex and Yacc. Bagan dibawah adalah proses pembuatan bahasa pemrograman dengan mengkonversi Lex-file dan Yacc-file menjadi executable 1.23. Di sini dicontohkan dua buah kode sumber (bas.y dan bas.l) yang masing-masing berfungsi sebagai grammar (yacc) dan scanner (lex). Setelah di-build/compile dengan cc, diperoleh file bas.exe yang siap digunakan.



Gambar 1.23: Proses Pembuatan Bahasa

1.8.4 Implementasi Pemrosesan bahasa

Lex/FLEX (*fast lexical analyzer generator*) adalah program komputer untuk membuat (*generating*) penganalisa leksikal atau scanner. Ditulis oleh Vern Paxson in C around 1987. porgram tersebut digunakan bersamaan dengan GNU Bison parser generator. Scanner generator menerima imputan berupa aliran karakter-karakter dari file code sumber kemudian menghasilkan aliran token seperti yang terlihat pada Gambar Struktur program lex terdiri dari tiga bagian:

5. **Bagian definisi** : Pada bagian ini berisi deklarasi variabel, definisi regular, manifest constants. Dalam bagian definisi ini, text ditutup denga "%{}%". Apapun yang ditulis dalam tanda kurung kurawal ini adalah hasil copy yang bersesuaian terhadap file lex.yy.c.

   Penulisan syntax :

   ```
   %{
     Definisi
   %}
   ```

6. **Bagian aturan**: bagian aturan ini berisi rangkaian dari aturan dalam bentuk: pola tindakan *pettern action*.

Penulisan syntax :

```
%{
   pettern action
%}
```

berikut adalah contoh pola lex generator:

7. **Bagian user code**: Pada bagian ini berisi statement code *C* dan fungsi tambahan. Kita juga dapat mengcompile fungsi tersebut secara terpisah dan load dengan pengalisa leksikal.

Basic struktur program :

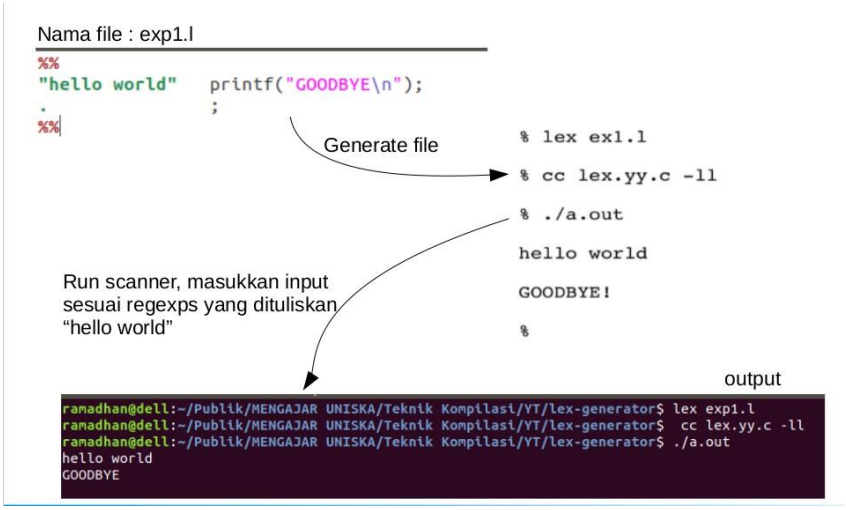
```
%{
   Definisi
% %
```

Pola	Match
[0-9]	semua dijit antara 0 dan 9
[0 + 9]	0, + atau 9
[0, 9]	0, ‘, ‘ atau 9
[09]	0, ‘ ‘ or 9
[_09]	-, 0 or 9
[—0 – 9]	– atau semua dijit antara 0 dan 9
[0 – 9]+	satu dijit atau lebih antara 0 dan 9
[^*a*]	semua karakter lainnya kecuali a
[^*A* − *Z*]	semua karakter lainnya kecuali huru besar
a{2, 4}	aa, aaa atau aaaa
.	karater apapun kecuali *newline*
a^*	0 atau lebih kemunculan a
a^+	1 atau lebih terhadap kemunculan a
[*a* – *z*]	huruf kecil alfabet
[*a* — z*A* – *Z*]	huruf alfabetik apapun
w(x | y)z	wxz atau wyz

rule
%

bagian code user

Untuk menjalankan program lex, pertamakali harus membuat file denga extensi **.l** atau **.lex** seperti yang ditunjukkan pada Gambar 1.24.



Gambar 1.24: RUN program lex

1.8.5 Implementasi Pemrosesan bahasa pada Windows OS

Dengan Lex and Yacc akan dicoba membuat kompiler yang bisa melakukan perhitungan seperti ini:

```
user:  3 * (4 + 5)
calc:  27
user:  x = 3 * (4 + 5)
user:  y = 5
user:  x
calc:  27
user:  y
calc:  5
user:  x + 2*y
calc:  37
```

Lex and Yacc memiliki dua mesin :

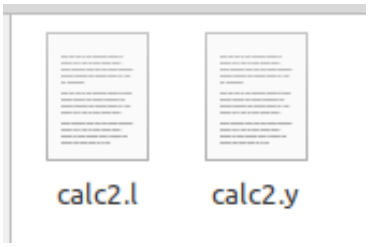
- Scanning kata per kata dengan mesin Lex (singkatan dari lexical generator), dan
- Grammar check dengan mesin Yacc (singkatan dari yet another compilers compiler)

Makanya software tersebut dinamakan Lex and Yacc. Jika software sudah diinstal, silahkan pelajari dari situs resminya, yang juga dilengkapi dengan kode sumber. Ada dua file yang diperlukan untuk menghasilkan satu kompiler yaitu file lex (berekstensi *.l) dan yacc (berekstensi *.y). Sebagai contoh dari kedua file tersebut bernama calc2.l dan calc2.y. Gambar 1.25 menunjukkan sebuah file calc2.l

```
    /* calculator #1 */
%{
    #include "y.tab.h"
    #include <stdlib.h>
    void yyerror(char *);
%}

%%

[a-z]       {
                yylval = *yytext - 'a';
                return VARIABLE;
                }

[0-9]+      {
                yylval = atoi(yytext);
                return INTEGER;
            }

[-+()=/*\n]     { return *yytext; }

[ \t]   ;       /* skip whitespace */

.               yyerror("Unknown character");

%%

int yywrap(void) {
    return 1;
}
```

Gambar 1.25: calc2.l

dan Gambar 1.27 menunjukkan sebuah file calc2.y yang sebelumnya telah tersimpan pada folder **latihan1.**

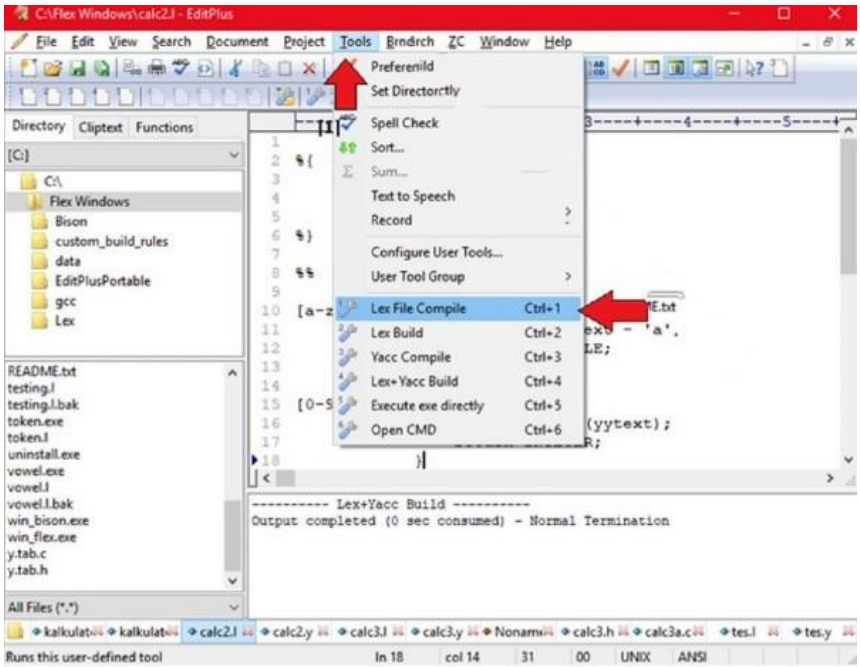


Gambar 1.26: file-file pada folder latihan1

```
%{
    #include <stdio.h>
    void yyerror(char *);
    int yylex(void);

    int sym[26];
%}

%token INTEGER VARIABLE
%left '+' '-'
%left '*' '/'

%%

program:
        program statement '\n'
        | /* NULL */
        ;

statement:
        expression                      { printf("%d\n", $1); }
        | VARIABLE '=' expression       { sym[$1] = $3; }
        ;

expression:
        INTEGER
        | VARIABLE                      { $$ = sym[$1]; }
        | expression '+' expression     { $$ = $1 + $3; }
        | expression '-' expression     { $$ = $1 - $3; }
        | expression '*' expression     { $$ = $1 * $3; }
        | expression '/' expression     { $$ = $1 / $3; }
        | '(' expression ')'            { $$ = $2; }
        ;

%%

void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}

int main(void) {
    yyparse();
}
```

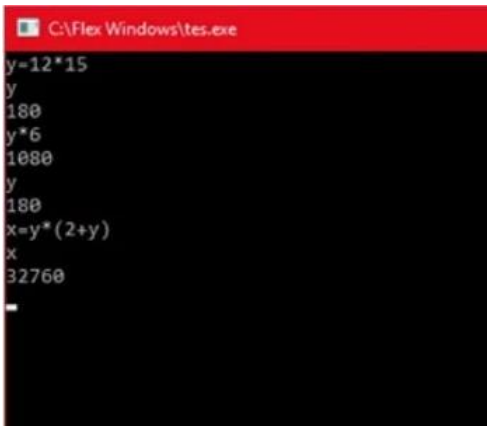
Gambar 1.27: calc2.y

Kopi-kan saja file-file tersebut ke Flex Windows (software Lex and Yacc versi windows) yang sudah terinstall sebelumnya. Masuk ke menu **Tools – Lex Compile**. Lanjutkan dengan masuk ke menu yang sama: **Tools – Lex Build**. Maka akan dihasilkan satu file baru lex.yy.c yang merupakan hasil generate ke bahasa c.



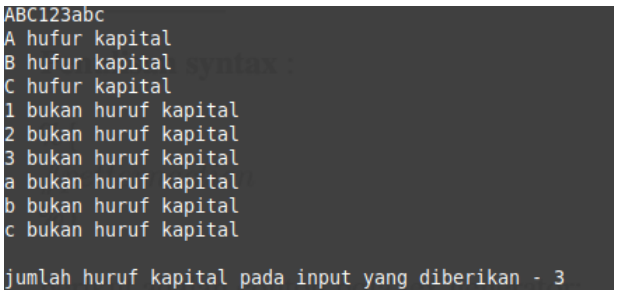
Jika Lex bertanggung jawab mengecek kata/word yang terdaftar di bahasa pemrograman yang dirancang, Yacc bertanggung jawab terhadap grammatical-nya. Buat kode dengan nama yang sama dengan file Lex, hanya saja untuk Yacc harus berekstensi *.y. Lakukan proses yang sama dengan Lex di menu **Tools – Yacc Compile** dan dilanjutkan dengan membuild beserta Lex-nya lewat menu yang sama **Tools – Lex+Yacc Build**. Hasil dari proses kompilasi adalah dua buah file y.tab.c dan y.tab.h yang satu untuk header (di bahasa c dengan kode #*include*) yaitu y.tab.h dan satu lagi y.tab.c digunakan untuk build Lex+Yacc. Di bagian indikator bawah pastikan tidak ada masalah dan cek di file lokasi penyimpanan (di folder yang sama dengan lex dan yacc) apakah file berekstensi exe dengan nama yang sama dengan lex and yacc ada, misalnya yang digunakan dalam contoh ini adalah calc2.exe.

Ada dua cara menjalankan bahasa pemrograman yang baru dibentuk yaitu dengan mengklik file calc2.exe atau nama lain yang Anda buat, atau dengan menekan **Tools – Execute Exe Directly**. Hasilnya akan memunculkan jendela di bawah ini, coba dengan melakukan operasi matematis tertentu dengan variabel.



Latihan !

- Buatlat program lex yang dapat menghitung jumlah karakter string, misal inputan string ″ABC123abc″. Hint: gunakan yytex sebagai pointer karakter input yang *match* dengan pola pada bagian aturan. Contoh output atau hasil program seperti berikut:



Gambar 1.28: Output

BAB 2
PENGANALISA SINTAK (SYNTAX ANALYSIS)

2.1 Penganalisa Sintak

Dimana Penganalisa leksikal (*lexical analyisis*) membagi sebuah source program kedalam token, tujuan dari penganalisa sintak (atau disebut juga parsing) adalah untuk menyatukan kembali token tersebut, namun tidak kembali menjadi sebuah list karakter, tetapi kedalam sesuatu yang mencerminkan struktur dari source program itu sendiri. Yang dimaksud sesuatunya disini, secara tipikal berupa struktur data yang biasa disebut dengan pohon sintak (*syntax tree*) dari sebuah source program. Dari namanya mengindikasi, ini adalah stuktur pohon (*structure tree*). Daun-daun dari pohon ini adalah token-token yang diperoleh dari proses leksikal analisis (*scanner*), dan jika daun- daun pada struktur pohon ini dilakukan pelacakan dari kiri ke kanan, rangkaian atau urutan akan sama sebagai souce program inputan.

Sebagai tambahan untuk menemukan stuktur dari teks inputan, penganalisa sintak harus juga melakukan reject tehadap source program yang tidak valid dengan memberikan pesan kesalahan sintak yang tepat. Sebagai contoh, pada source program Java berikut ini :

```
public class  Bug{

   public static void main(String[] args){
      System.out.println("hello")
   }
}
```

tidak terdapatnya tanda titik koma pada statement println adalah bagian source program yang tidak valid (kesalahan sintak atau *syntax error*).

Untuk mendeteksi setiap kemungkinan kesalahan sintak, compiler dengan jelas harus memiliki pengetahuan yang komplit tehadap sintak dari source language itu sendiri. Pengetahuan ini harus ditanamkan kedalam compiler dalam bentuk sebagai berikut :

1. **Ringkas**, jika hal ini diabaikan kompiler akan menjadi sangat besar.
2. **Presisi**, jika hal ini diabaikan compiler tidak dapat melakukan pengecekan kesalahan secara akurat.
3. **powerfull**, cukup kuat untuk dapat mendeskripsikan sintaks
4. **Algoritma**, pemilihan algoritma *syntax checking* yang *suitable* untuk efesiensi.

Mari kita pertimbangkan seberapa bagus representasi sintak terhadap (misal) sebuah buku

pemrograman dipertemuakan dengan 4 syarat diatas:

1. Ringkas ? :, misalkan kita memiliki sebuah buku pemrograman dengan 1000 halaman, setengah dari halamannya berisi sintak program , dengan 2000 karakter per halaman. dalam permasalahan ini, textbooks menggunakan total dari 1 milion karakter untu spesifikasi sintak. **Sangat sulit untuk diringkas**.

2. Presisi ? : tidak ada jalan. Deskripsi bahasa inggris pada sintak selalu meninggakan banyak point terbuka yang bagus untuk pertanyaan.

3. Cukup kuat (*powerful*) ?:, kebanyakan buku bahasa pemrograman tidak benar-banar full mendeskripsikan sintak dari sebuah bahasa yang mereka sajikan. Sebagai contoh seberapa banyak buku permrograman Java yang kamu ketahui jika sebuah bidang dan sebuah metode dalam sebuah class dapat memili nama class yang sama. Normalnya , kamu selalu akan menggunakan nama yang berbeda, jadi detail dari sintak java ini bukan merupakan hal yang penting dari perspektif pemrograman. Namu bagaimanapun juga , hal detail tersebut adalah esensial untuk diketahui oleh compiler.

4. Apakah ada algoritme yang dapat membaca dan menginterpretasi secara tepat sebuah textbooks ? Mungkin kita akan mengembangkan algoritme tersebut ditahun 2100. tetapi bahkan jika kita memilika algoritme tersebut, requirments 2 dan 3 akan tetap menjadi masalah.

2.2 Tata bahasa bebas konteks (Context-Free Grammars)

Seperti halnya regular expression, *Context-Free Grammars* (CFG) juga mendeskripsikan himpunan-himpunan string, yaitu bahasa (*language*). CFG juga memberi struktur definisi pada string dalam bahasa yang didefinisikan. Sebuah bahasa menetapkan lebih dari beberapa alphabet, sebagai contoh : himpunan dari token-token yang dihasilkan oleh lexer atau himpunan karakter dari *alphanumeric*. Simbol-simbol dalam alfabet disebut dengan *terminals*.

CFG secara rekursive memberi definisi beberapa *sets* string. Masing-masing *set* disimbolkan dengan sebuah nama yang disebut dengan *nonterminals*. *Set* dari *nonterminal* adalah *disjoint* dari *set* terminal. Satu dari nonterminal dipilih untuk menyimbolkan bahasa utama yang dideskripsikan oleh grammar. Nonterminal ini disebut dengan simbol permulaan (*start symbol*) dari grammar dan memainkan peranan mirip dengan state awal (*start state*) pada sebuah finite automata. *Sets* dijelaskan oleh sejumlah *production*. Tiap-tiap *production* menjelaskan beberapa dari kemungkinan string yang termuat dalam himpunan, disimbolkan dengan sebuah nonterminal.

Tiap tiap production tediri dari dua string yang dipisahkan dengan simbol →. Simbol → artinya "dapat digantikan dengan" contoh bentuk dari production: $N \rightarrow X_1 ... X_N$.

dimana N adalah nonterminal dan $X_1 ... X_N$ adalah zero atau simbol-simbol lainnya. Seperti halnya regular expression a^+ ekivalen denga bentuk *production* $A \rightarrow aA$ (himpunan dengan simbol A berisi semua string yang dibentuk dengan meletakkan a pada bagian depan dari string yang diambil dari himpunan simbol A) dan a^* ekivalen terhadap production :

$$B \rightarrow$$
$$B \rightarrow aB$$

contoh lainn :

$$T \rightarrow R$$
$$T \rightarrow aTaR \rightarrow b$$
$$R \rightarrow bR$$

dapat ditulis sebagai,

$$T \rightarrow R|aTaR \rightarrow b|bR$$

dapat ditulis sebagai,

$$T \rightarrow b^{+}|aTa$$

yang merupakan *Extended Backus–Naur Form* (EBNF) dari regular expression (?, + dan $*$).

Contoh :

- Definisikan grammar dari bahasa non regular ini : $\{a^n b^n | n \geq 0\}$

Solusi **:**

$$S \rightarrow$$
$$S \rightarrow aSb$$

penjelasan : production yang kedua yaitu $S\ aSb$ memastikan bahwa a dan b dipasangkan secara simetris karena S pada production pertama merupakan string kosong dan S berada disekitar tengah string, dengan demikian ab akan muncul dengan jumlah yang sama.

Latihan !

- Grammar memberi definisi sebagai berikut :

$$A \rightarrow aAa$$
$$A \rightarrow bAb$$
$$A \rightarrow$$

tulis kedalam pernyataan bahasa non regular (*language*) dari definisi Grammar tersebut.

2.2.1 Cara menulis Context-Free Grammars

Regular expression secara sistematikal dapat dituliskan ulang ke sebuah CFG yang ekivalen menggunakan nonterminal untuk setiap *subexpression* pada regular expression, dan menggunkan satu atau dua production untu masing-masing nonterminal. Susunuan atau bentuk dapat dilihat pada Tabel 2.1 dengan demikian, jika kita dapat menyatakan sebuah bahasa sebagai regular expression, hal itu akan mudah untuk membuat grammar tersebut. Kita juga akan menggunakan CFG untuk mendeskripsikan bahasa non-regular. Contoh bahasa non-regular adalah jenis pengoperasian aritmatika yang ada pada kebanyakan bahasa pemrograman yang terdiri dari bilangan-bilangan, operator dan tanda (). Jika pernyataan aritmatika tidak memiliki tanda kurung,

bahasa tersebut dapat dinyatakan dengan sebuah regular expression, misalnya seperti:

$$num((+|-|*|/)num)^*$$

num menyatakan sembarang bilangan konstan dan penryataan aritmatika dengan tanda kurung dapat dinyatakan dengan CFG sederhana seperti berikut:

$$\begin{aligned}
Exp &\rightarrow Exp + Exp \\
Exp &\rightarrow Exp - Exp \\
Exp &\rightarrow Exp * Exp \\
Exp &\rightarrow Exp\ Exp \\
Exp &\rightarrow num \\
Exp &\rightarrow (Exp)
\end{aligned}$$

namun CFG aritmatika sederhana diatas tidak dapat memebedakan operator yang mana lebih dulu dikerjakan. Dan untuk pernyataan *statement* grammar sederhana adalah seperti berikut:

$$\begin{aligned}
Stat &\rightarrow id := Exp \\
Stat &\rightarrow Stat; Stat \\
Stat &\rightarrow \text{if } Exp \text{ then } Stat \text{ else } Stat \\
Stat &\rightarrow \text{if } Exp \text{ then } Stat
\end{aligned}$$

pada grammar tersebut *Exp* adalah nonterminal dari CFG sederhana dan terminal *id* menyatakan nama variabel.

Bentuk s_i	**Production untuk** N_i
ε	$N_i \rightarrow$
a	$N_i \rightarrow a$
$s_j s_k$	$N_i \rightarrow N_j N_k$
$s_j \mid s_k$	$N_i \rightarrow N_j$ $N_i \rightarrow N_k$
s_j^*	$N_i \rightarrow N_j N_i$ $N_i \rightarrow$
s_j^+	$N_i \rightarrow N_j N_i$ $N_i \rightarrow N_j$
$s_j^?$	$N_i \rightarrow N_j$ $N_i \rightarrow$

Tabel 2.1: Dari REGEXPS ke CFG

2.3 Derivasi

Derivasi atau penurunan adalah cara formal untuk menentukan himpunan string yang dijelaskan dengan grammar. Ide dasar dari derivasi adalah untuk memperhitungkan production sebagai aturan yang ditulis ulang. Bilamana saja terdapat sebuah nonterminal, dimana nonterminal tersebut muncul pada sisi kiri maka dapat diganti dengan apapun production tunggal pada sisi kanan. Hal ini dapat dilakukan kapanpun dalam urutan simbol (nonterminal dan terminal) dan dapat diulang sampai yang tersisa hanya terminal. Urutan terminal yang dihasilkan pada proses ini adalah bahasa string terdefinisi oleh grammar. Secara formal hubungan derivasi dapat didefinisikan dengan tiga aturan berikut:

5. $\alpha N \beta \Rightarrow \alpha\gamma\beta$ jika terdapat production $N \rightarrow \gamma\alpha \Rightarrow \alpha$

6. $\alpha \Rightarrow \gamma$ jika terdapat β yang seperti $\alpha \Rightarrow \beta$ dan $\beta \Rightarrow \gamma$

dimana α, β dan γ adalah urutan (mungkin kosong) simbol-simbol grammar (noterminal dan terminal). Pada aturan state pertama adalah tahap derivasi (dalam urutan simbol grammar) menggunakan production sebagai sebuah aturan yang ditulis kembali. Pada state kedua adalah hubungan derivasi refleksip, yaitu urutan α itu sendiri. Aturan ketiga menyatakan transitip, yaitu urutan pada derivasi tersebut merupakan derivasi itu sendiri. Kita dapat menggunakan derivasi untuk menyatakan secara formal bahasa yang dibuat (*generates*). Misal diberikan sebuah CFG G dengan *start* simbol S, simbol terminal T dan production P, sebuah bahasa dinyatakan dengan $L(G)$ bahwa G menghasilkan simbol terminal didefinisikan sebagai himpunan string yang dapat diperoleh dengan derivasi dari S menggunakan production P, yaitu sebuah himpunan $\{w \in T^* | S \Rightarrow w\}$.

Sebagai contoh, Grammar di bawah ini membuat string aabbbcc:

$$T \rightarrow R$$
$$T \rightarrow aTc$$
$$R \rightarrow$$
$$R \rightarrow RbR$$

dengan derivasi atau penurunan terkanan (*rightmost derivation*) sebagai berikut:

$$\underline{T}$$
$$\Rightarrow a\underline{T}c$$
$$\Rightarrow aa\underline{T}cc$$
$$\Rightarrow aa\underline{R}cc$$
$$\Rightarrow aaRb\underline{R}cc$$
$$\Rightarrow aa\underline{R}bcc$$
$$\Rightarrow aaRb\underline{R}bcc$$
$$\Rightarrow aaRb\underline{R}bRbcc$$
$$\Rightarrow aa\underline{R}bbRbcc$$
$$\Rightarrow aabb\underline{R}bcc$$
$$\Rightarrow aabbbcc$$

dengan derivasi atau penurunan terkiri (*leftmost derivation*) sebagai berikut:

$$\underline{T}$$
$$\Rightarrow a\underline{T}c$$
$$\Rightarrow aa\underline{T}cc$$
$$\Rightarrow aa\underline{R}cc$$
$$\Rightarrow aa\underline{R}bRcc$$
$$\Rightarrow aa\underline{R}bRbRcc$$
$$\Rightarrow aab\underline{R}bRcc$$
$$\Rightarrow aab\underline{R}bRbRcc$$
$$\Rightarrow aabb\underline{R}bRcc$$
$$\Rightarrow aabbb\underline{R}cc$$
$$\Rightarrow aabbbcc$$

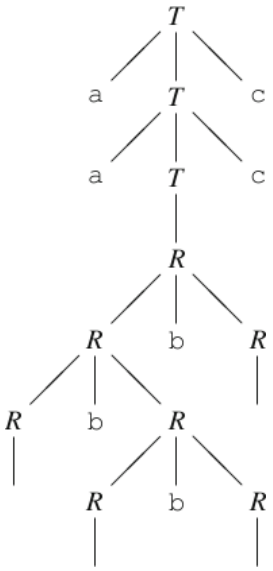
Penurunan yang ditulis ulang paling kiri nonterminal disebut dengan penurunan terkiri *leftmost*

derivasi dan yang selalu ditulis ulang paling kanan disebut dengan penurunan terkanan*rightmost derivasi*.

2.3.1 Pohon Sintak dan Kedwiartian

Pohon sintak gunanya untuk menggambarkan bagaimana memperoleh suatu string dengan cara menurunkan simbol-simbol variabel menjadi simbol-simbol terminal. Akar dari pohon adalah simbol mulai (*start symbol*) pada grammar. Setiap kali nonterminal ditulis ulang, kita tambahkan nonterminal tersebut sebagai simbol anak pada bagian kanan dari production yang sebelumnya digunakan. Daun dari pohon adalah terminal yang saat dibaca dari kiri ke kanan membentuk turunan string. Jika nonterminal ditulis ulang menggunakan production kosong (*empty production*), bagian node anak ditunjukkan sebagai sebuah node daun kosong, yang mana diabaikan saat penelusuran string dari daun terhadap pohon tersebut. Node daun kosong pada pohon sintak ditunjukkan dengan ε.

Sebagai contoh, derivasi atau penurunan terkiri dan derivasi atau penurunan terkanan di atas menghasilkan pohon sintak yang sama. Pohon sintak dapat dilihat pada Gambar 2.1.



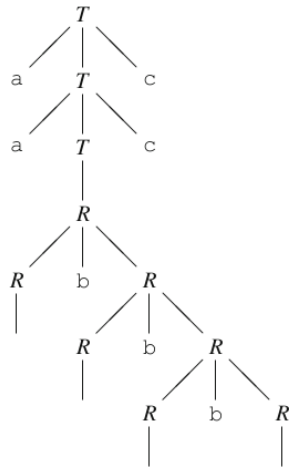
Gambar 2.1: Pohon sintak untuk string aabbbcc menggunakan grammar sebelumnya

Meskipun urutan derivasi tidak dipermasalahkan saat membangun pohon sintak, pemilihan produksi yang berbeda terhadap nonterminal akan membawa pada string yang berbeda saat diturunkan, bisa juga terjadi pada string yang sama dengan pohon sintak yang berbeda. Sebagai contoh, alternatif pohon sintak untuk string aabbbcc dapat dilihat pada Gambar 2.2. Saat beberapa pohon sintak dapat dibangun dengan grammar yang sama terhadap suatu string, maka grammar ini disebut kedwiartian atau memilik sama arti. Jika grammar hanya digunakan untuk menyatakan himpunan string kedwiartian bukanlah sebuah masalah. Akan tetapi, saat ingin menentukan **struktur pada string** lebih baik hidari masalah kedwiartian. Biasanya dalam permasalahan kedwiartian pada grammar dapat ditulis ulang ke grammar yang tidak memiliki sama arti dan tetap menghasilkan himpunan string yang sama. Sebagai contoh dapat dilihat pada 2.3 Kita dapat mengatahui kapan grammar tersebut dikatakan memiliki kedwiartian dengan

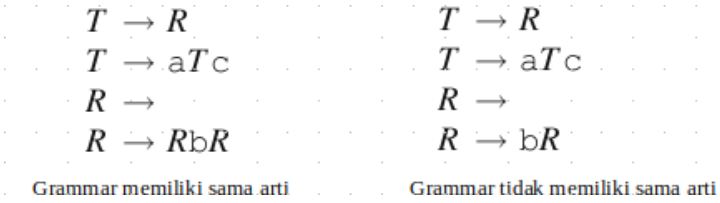
cara jika terdapat dua alternatif pohon sintak pada suatu string maka grammar tersebut memiliki sama arti. Sebagai contoh, jika grammar mempunyai bentuk production berikut:

$$N \rightarrow N\alpha N$$
$$N \rightarrow \beta$$

dimana α dan β merupakan rangkaian sembarang simbol grammar, grammar tersebut memiliki sama arti.

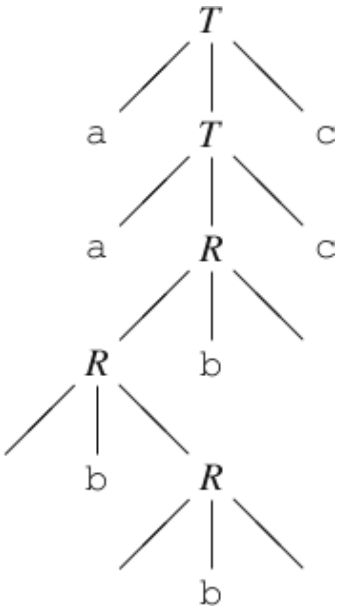


Gambar 2.2: Pohon sintak untuk string aabbbcc menggunakan grammar sebelumnya



Gambar 2.3: Grammar dwiarti dan non-dwiarti

Saat dua pohon sintak yang berbeda menyatakan struktur yang sama maka kita dapat mendefinisikan ulang dengan cara mereduksi pohon sintak tersebut. Jika sebuah node pada pohon sintak hanya memiliki satu node anak, ganti node tersebut dengan node anak(mungkin node jadi kosong). Sebuah pohon sintak yang tidak memilik node dengan hanya satu node anak adalah reduksi penuh (*fully reduced*). Kita menganggap dua pohon sintak untuk menyatakan struktur yang sama jika pohon *fully reduced* adalah indentik kecuali untuk nama nonterminal yang menyatakan kategori sintak yang sama. perlu diperhatikan bahwa pohon yang direduksi tidak selalu berupa pohon sintak yang tepat. *Edge* tidak menyatakan langkah derivasi tunggal, tetapi rangkain langkah derivasi. Gambar 2.4 menunjukkan bentuk sintak tree fully reduced terhadap pohon sintak 2.1. Pada bagian selanjutnya, kita akan melihat cara menghilangkan kedwiartian pada grammar dengan menulis ulang grammar tersebut. Seperti dengan menggunakan transformasi, kita dapat membuat himpunan grammar skala besar. Sayangnya, seprti kdwiartian, ekivalensi pada CFG tidak dapat dipastikan. Terkadang ekivalansi bisa dibuktikan dengan melakukan induksi kesuluruh himpunan string yang dihasilkan grammar.



Gambar 2.4: Pohon sintak *fully reduced*

Latihan!

- Buatlah pohon sintak tree untuk string aabbbcc menggunakan grammar berikut:

$$T \rightarrow R$$
$$T \rightarrow aTc$$
$$R \rightarrow$$
$$R \rightarrow bR$$

serta tunjukkan derivasi sebelah kiri atau derivasi sebelah kanan (pilih salah satu) yang bersesuaian dengan pohon sintak tersebut.

2.4 Operator Prioritas

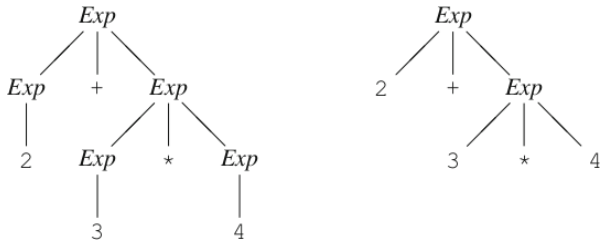
Pernyataan aritmatika tradisional dapat dideskripsikan dengan Grammar berikut:

$$Exp \rightarrow Exp + Exp$$
$$Exp \rightarrow Exp - Exp$$
$$Exp \rightarrow Exp * Exp$$
$$Exp \rightarrow Exp\ Exp$$
$$Exp \rightarrow num$$
$$Exp \rightarrow (Exp)$$

num adalah terminal yang menyatakan integer konstan, simbol tanda kurung adalah terminal(tidak seperti pada Regexps, dimana tanda kurung digunakan untuk menetapkan struktur pada regexps tersebut).

Kebanyakan bahasa pemrograman menggunakan konvensi yang sama dengan kalkulator ilmiah. Dengan demikian kita akan membuat hal ini secara eksplisit dalam sebuah grammar.Idealnya kita ingin menyatakan sebuah ekspresi 2 + 3 ∗ 4 kedalam bentuk pohon sintak yang

ditunjukkanseperti pada Gambar 2.5, yang mencerminkan operator prioritas dengan mengelompokkan subexpression. Subexpression dinyatakan dengan subtree pada pohon sintak yang telah dievaluasi sebelum operator paling atas diterapkan.



Gambar 2.5: Pohon sintak *Reduksi penuh*

2.4.1 Menulis Kembali Kedwiartian Grammar

Terdapat sebuah pendekatan yang mungkin dilakukan untuk mengatasi kedwiartian selama penganalisaan sintak. Banyak parser generator memperbolehkan pendekatan ini. Pertama kali kita akan mendefinisikan beberapa konsep terkait operator infix:

1. **Asosiatif terkiri**. Sebuah operator $\oplus$ adalah Asosiatif terkiri jika ekspresi $a \oplus b \oplus c$ harus dievaluasi dari kiri ke kanan, misalnya : $(a \oplus b) \oplus c$
2. **Asosiatif terkanan**. Sebuah operator $\oplus$ adalah Asosiatif terkanan jika ekspresi $a \oplus b \oplus c$ harus dievaluasi dari kanan ke kiri, misalnya : $a \oplus (b \oplus c)$
3. **Non-Asosiatif**. Sebuah operator $\oplus$ adalah Non-asosiatif jika ekspresi terhadap bentuk $a \oplus b \oplus c$ adalah tidak sah.

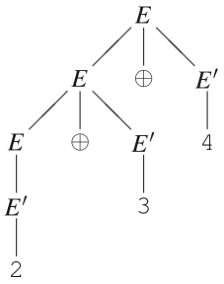
Dengan kesepakatan biasa, dan / adalah asosiasi-terkiri, sebagai contoh : 2 3 4 dioperasikan dengan (2 3) 4. + dan merupakan asosiatif secara istilah matematika. Diberikan sebuah grammar ambigu sebagai berikut:

$$E \rightarrow E \oplus E$$
$$E \rightarrow num$$

Kita dapat menulis ulang grammar diatas sehingga tidak menjadi ambigu dan menghasilkan struktur yang benar. Kita akan menggunakan aturan penulisan ulang yang berbeda untuk asosiatif yang berbeda. Jika $\oplus$ adalah asosiatif-terkiri, kita buat grammar rekursif-terkiri dengan hanya memiliki referensi kekiri pada simbol operator dan mengganti referensi rekursif terkanan dengan sebuah referensi pada nonterminal baru yang menyatakan permasalahan non-rekursif.

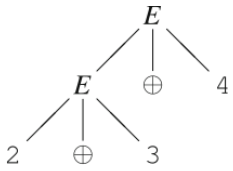
$$E \rightarrow E \oplus E^{\text{I}}$$
$$E \rightarrow E^{\text{I}}$$
$$E^{\text{I}} \rightarrow num$$

Sekarang, ekspresi $2 \oplus 3 \oplus 4$ hanya dapat diparse sebagai berikut 2.6 :



Gambar 2.6: Pohon sintak $2 \oplus 3 \oplus 4$

Pohon sintaks di atas sedikit lebih kompleks dari idealnya, untuk menghindari dua kepemilikan nonterminal yang berbeda dan juga menghidari derivasi $E \rightarrow E^{\text{I}}$ tetapi pohon tersebut mesti mencerminkan lebih kepada penerapan struktur terkiri dari $\oplus$ dibandingkan dengan penerapan terkanan dari $\oplus$. pohon sintaks reduksi penuh yang bersesuaian membuat hal ini lebih jelas 2.7:



Gambar 2.7: Pohon sintak $2 \oplus 3 \oplus 4$ reduksi penuh

Kita menangani asosiatif-terkanan dalam sebuah cara yang sama dengan rekursif-terkanan:

$$E \rightarrow E^{\text{I}} \oplus E$$
$$E \rightarrow E^{\text{I}}$$
$$E \rightarrow num$$

Operator Non-asosiatif yang ditangani dengan produksi non-rekursif :

$$E \rightarrow E^{\text{I}} \oplus E$$
$$E \rightarrow E^{\text{I}}$$
$$E \rightarrow num$$

Perlu diperhatikan bahwa akhir transformasi sebenarnya merubah bahasa yang dihasilkan grammar, seperti membuat bentuk $num \oplus num \oplus num$ tidak valid.

Dengan demikian , kita telah menangani hanya kasus dimana sebuah operator berinteraksi dengan operator itu sendiri. Hal ini dengan mudah diperluas ke suatu masalah dimana beberapa operator dengan prioritas sama secara asosiatif beriteraksi dengan satu sama lainny, sebagai contoh + dan $-$:

$$E \rightarrow E + E^{\text{I}}$$
$$E \rightarrow E - E^{\text{I}}$$
$$E \rightarrow E^{\text{I}}$$
$$E^{\text{I}} \rightarrow num$$

Operator dengan prioritas yang sama harus mempunyai kesamaan secara asosiatif untuk dapat diproses, sebagai campuran produksi rekursif-terkiri dan rekursif-terkanan untuk nonterminal

yang sama membuat grammar menjadi ambigu. Sebagai contoh, grammar berikut:

$$E \rightarrow E + E^{\text{I}}$$
$$E \rightarrow E^{\text{I}} \oplus E$$
$$E \rightarrow E^{\text{I}}$$
$$E^{\text{I}} \rightarrow num$$

Terlihat jelas seperti prinsip generalisasi yang digunakan diatas, memberikan prioritas sama terhadap + dan $\oplus$ dan secara asosiatif berbeda. Tetapi tidak hanya memberikan grammar ambigu, bahkan itu tidak menerima bahasa yang dimaksud. Sebagai contoh, string $num + num \oplus num$ tidak dapat diturunkan dengan grammar ini.

Secara umum tidak ada cara yang jelas untuk menyelesaikan kedwiartian terhadap expression seperti $1 + 2 \oplus 3$, dimana + adalah asosiatif-terkiri dan $\oplus$ adalah asosiatif-terkanan (atau sebaliknya). Oleh sebab itu, kebanyakan bahasa pemrograman (dan parser generator) memerlukan operator-operator dengan prioritas sama tingkatan untuk memiliki asosiatif yang identik. Kita juga perlu untuk menangani operator-operator dengan prioritas berbeda. Ini dapat dilakukan dengan menggunakan sebuah nonterminal untuk masing-masing tingkatan prioritas. Idenya adalah jika sebuah ekspresi menggunakan sebuah tingkatan operator tertentu, maka subekspresinya tidak dapat menggunakan prioritas operator lebih rendah(kecuali menggunakantanda kurung). Oleh sebab itu, production untuk sebuah nonterminal sesuai terhadap tingkatan prioritas utama hanya mengacu pada nonterminal yang sesuai dengan sama tingkatan prioritas atau lebih tinggi, kecuali tanda kurung atau yang serupa menyusun disambiguasi yang digunakan untuk hal ini. Grammar 2.8a menunjukkan bagaimana aturan ini digunakan untuk membuat sebuah Grammar tidak ambigu dari grammar yang ada pada Gambar 2.8b

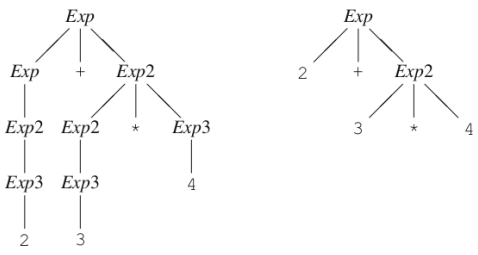
$$\begin{aligned}
Exp &\rightarrow Exp + Exp2 \\
Exp &\rightarrow Exp - Exp2 \\
Exp &\rightarrow Exp2 \\
Exp2 &\rightarrow Exp2 * Exp3 \\
Exp2 &\rightarrow Exp2 \,/\, Exp3 \\
Exp2 &\rightarrow Exp3 \\
Exp3 &\rightarrow \textbf{num} \\
Exp3 &\rightarrow (\, Exp\,)
\end{aligned}$$

(a) Ekspresi Grammar tidak ambigu

$$\begin{aligned}
Exp &\rightarrow Exp + Exp \\
Exp &\rightarrow Exp - Exp \\
Exp &\rightarrow Exp * Exp \\
Exp &\rightarrow Exp / Exp \\
Exp &\rightarrow \textbf{num} \\
Exp &\rightarrow (\, Exp\,)
\end{aligned}$$

(b) Ekspresi Grammar Sederhana

Gambar 2.9 berikut merupakan pohon sintak dari grammar 2.8a untuk 2+3+4 dan telah direduksi secara bersesuaian. Perlu diperhatikan bahwa nonterminal $Exp, Exp2$ dan $Exp3$ semua sama



Gambar 2.9: Pohon sintak *Reduksi*

menyatakan kategori(ekspresi) sintaktik, Jadi saat poho direduksi kita dapat menyamakan Exp dan Exp2, membuat hal tersebut ekivalen untuk pohon yang direduksi.

Latihan !

Tunjukkan bahwa grammar berikut :

$$A \rightarrow -A$$
$$A \rightarrow A - id$$
$$A \rightarrow id$$

adalah ambigu dengan menemenukan sebuah string yang memiliki dua pohon sintak berbeda serta buat dua grammar berbeda yang tidak ambigu untuk bahasa yang sama. dengan menggunakan grammar baru tunjukkan juga pohon sintak tree reduksi penuh.

2.5 Top-Down Parsing

Parsing secara istilah umum diartikan menganalisa sebuah kalimat yang terdapat pada struktur sintaksis itu sendiri. Dari sudut pandang teoritis, penganalisa leksikal juga adalah parsing.

Top-Down parsing dapat dilihat sebagai suatu permasalahan terhadap sebuah rancangan *parse tree* dalam preorder (*depth-first*). Secara ekivalensi, top-down parsing dapat dilihat sebagai proses pencarian penurunan terkiri untuk inputan string. Gambar 2.11 adalah contoh dari Top-down parsing yang dibangun dari grammar berikut:

$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \mid \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \mid \epsilon \\ F & \rightarrow & (\ E\ ) \mid \mathbf{id} \end{array}$$

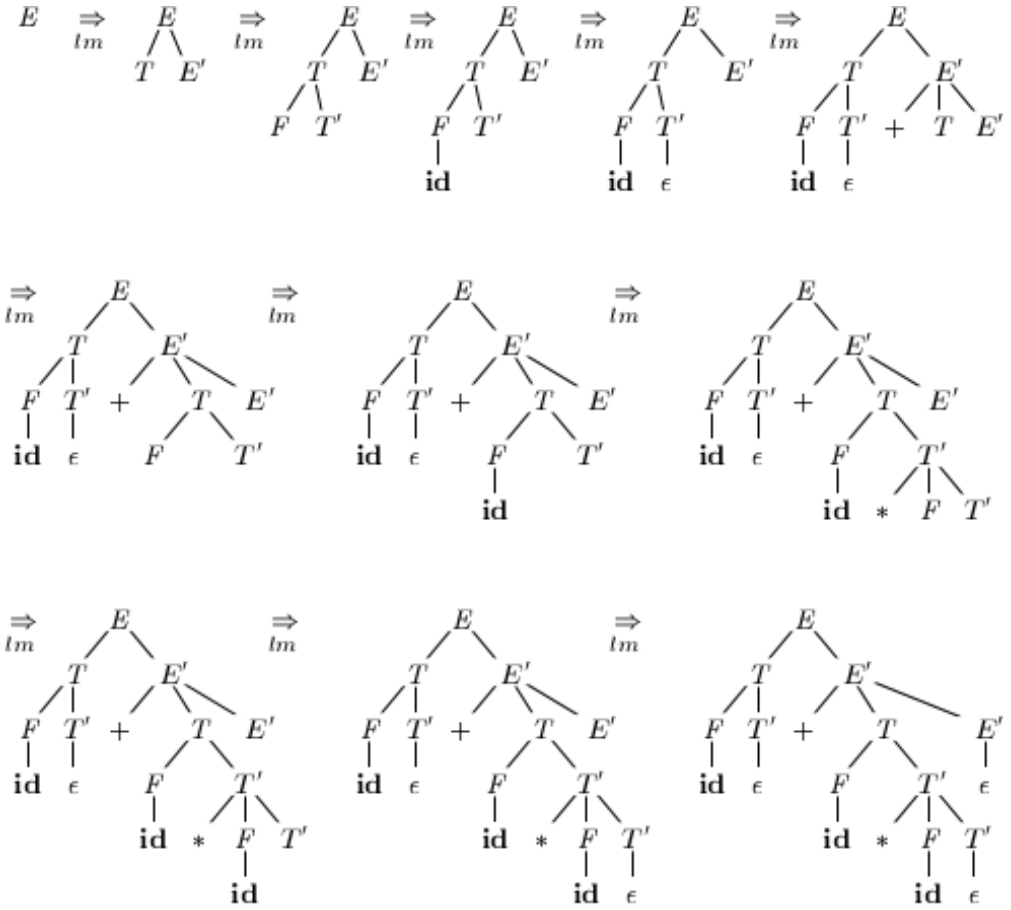
Pada tiap-tiap tahapan dari top-down parse, masalah utamanya adalah menentukan produksi untuk diterapkan pada sebuah nonterminal. Sebagai contoh top-down parse pada Gambar 2.11 yang merancang sebuah pohon dengan dua node dilabelkan E'. Pada E' pertama node dalam preorder, produksi $E' \rightarrow +TE'$ dipilih; pada node E' kedua, produksi $E' \rightarrow E$ dipilih. Sebuah predictive parser dapat memilih antara E'-produksi dengan melihat pada simbol inputan selanjutnya. Bentuk umu dari top-down parsing disebut rekursif descent parsing. Kasus khusus dari rekursif-descentparsing, dimana tidak diperlukan runut mundur. Berikut adalah tipikal prosedur recursif-descent parsing untuk nonterminal dalam sebuah top-down parser:

```
    void A() {
1)      Choose an A-production, A → X1 X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

Gambar 2.10: Jenis prosedur untuk nonterminal pada *top-down parser*

Perlu diperhatikan bahwa pseudocode diatas adalah nondeterministic karena dimulai dengan memilih A-production untuk diterapkan dengan cara tidak ditentukan. Rekursif descentmungkin memerlukan backtracking; dalam hal ini memerlukan pemindaian secera berulang pada keseluruhan input, namun proses backtracking ini jarang diperlukan untuk bahasa pemrogramanparsing yang dibangun. Jadi proses backtracking ini jarang ditemui. Bahkan untuk situasi seperti pasrsing bahasa alami backtracking sangat tidak efisien.

Untuk dapat melakukan backtracking, prosedur diatas memerlukan perubahan, yang pertama kita tidak dapat memilih *A-production* unik pada line (1), jadi kita harus mencoba tiap-tiap dari beberapa produksi dalam beberapa urutan. Kemudian error pada line (7) bukan error terakhir, tetapi hanya sebuah pesan untuk kembali ke line (1) dan kemudian mencoba *A-production* lainnya. Hanya jika tidak terdapat lebih *A-production* maka pesan error dapat ditampilkan. Untuk mencoba *A-production* lainnya, kita perlu melakukan reset pointer inputan dimana sebelumnya saat pertama kali line (1) terpenuhi. Dengan demikian sebuah variabel lokal diperlukan untuk menyimpan pointer inputan untuk digunakan pada proses berikutnya.



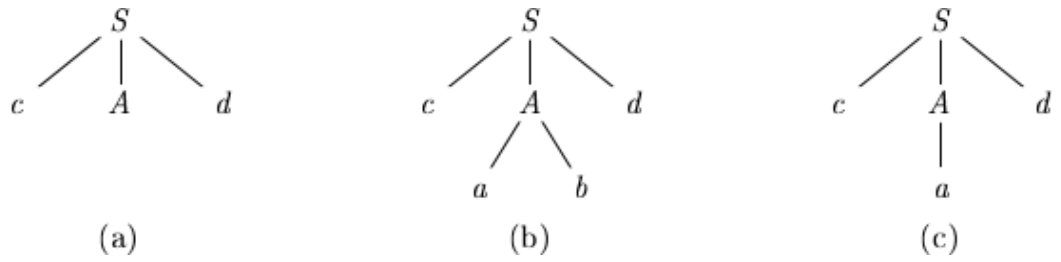
Gambar 2.11: Pohon sintak Top-down untuk *id + id * id*

Sebagai contoh rancangan pohon parse *top-down* untuk inputan string $w = cad$ menggunakan grammar berikut:

$$S \rightarrow cAd$$
$$A \rightarrow ab|a$$

dimulai dengan sebuah pohon terdiri dari satu node yang dilabelkan dengan S dan pointer inputan menunjuk ke *c*, merupakan simbil pertama untuk w. S hanya memiliki 1 produksi, jadi

kita gunakan itu untuk mereganggkan S dan diperoleh pohon seperti yang ditunjukkan pada Gambar 2.12(a)



Gambar 2.12: Tahapan parsing *top-down*

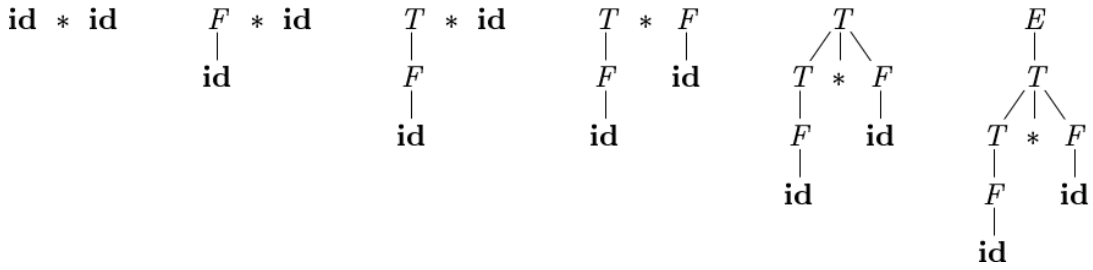
Sekarang, kita akan perluas A menggunakan Alaternatif pertama $A \rightarrow a\ b$ untuk mendapatkan pohon seperti yang ditunjukkan pada Gambar 2.12(b). Kita memiliki simbol input kedua, a, yang bersesuaian, jadi kita majukan pointer inputan ke d, inputan simbol ke 3, dan berlawanan membandingkan d pada daun selanjutnya, dilabelkan dengan b. Karena b tidak sesuai dengan d, kita laporkan kegagalan dan kembali ke A untuk melihat apakah ada alternatif lain yang belum pernah dicoba, akan tetapi hal itu menghasilkan kecocokan.

Dalam proses kembali ke A, kita mesti melakukan reset pointer inputan ke posisi 2, posisi itu saat pertama kali mengunjungi A, yang artinya bahwa prosedur untuk A harus menyimpan pointer inputan dalam sebuah lokal variabel.

Alternatif kedua untuk A menghasilkan pohon yang dapat dilihat pada Gambar 2.12(c). Daun a cocok sesuai dengan simbol kedua dari inputan string w dan daun d sesuai dengan simbol ke 3. Karena kita telah menghasilkan *parse tree* untuk inputan string w, proses dihentikan dan memberitahukan proses parsing berhasil dilakukan.

2.6 Bottom-Up Parsing

Parse *bottom-up* dimulai pada daun paling bawah dan dilanjutkan dengan menuju daun paling atas (root). Sebai ilustrasi pada Gambar 2.13 merupakan pohon parse *bottom-up*.



Gambar 2.13: bottom-up parse untuk *id * id*

Gambar 2.13 merupakan tahapan parse terhadap aliran token $id * id$ terhadapa ekspresi grammar berikut:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

2.7 Strategi Pemulihan Kesalahan

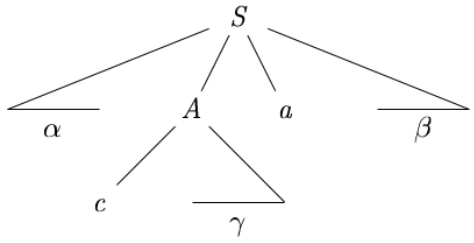
Secara umum kesalahan yang terdeteksi pada saat program dijalankan berbeda tingkatan mulai dari kesalahan *lexical* seperti penulisan identifier, keywords, atau opertor contohnya penggunaan identifier ellipseSize namu ditulis dengan ellipseSize. *Sintaksis* termasuk lupa menambahkan tanda titik koma (;) atau lupa menambahkan kurung kurawal "{" atau"}". Sebagai contoh lainnya misalkan pada bahasa pemrograman C/C++ atau Java dengan statement case tanpa ditutup dengan switch. *Semantic* ketidak sesuaian penggunaan operator dan operand, misalnya return nilai pada method Java dengan tetapi dengan tipe void. *Logical* misalkan ingin membandingakan suatu nilai dengan operator == tetapi menggunakan operator =.

Sekali sebuah kesalahan terdeteksi, bagaiaman cara parser memulihkannya? Meskipun tidak ada strategi yang menjamin secara universal, beberapa metode telah memiliki penerapan secara luas. Pendekatan paling sederhana dengan sedikit pesan error informatif yang lakukan oleh parser dapat mengembalikan state tersebut dimana pemrosesan dari inputan dapat dilanjutkan dengan pesan kesalahan yang dapat diterima untuk dilakukan pemrosesan selanjutnya dengan memberikan informasi diagnostik. Salah satu strategi yang dapat digunakan dalam hal ini yaitu saat parser menemukan kesalahan apapun pada statement, parser akan mengabaikan sisa statement dengan tidak memproses pembatas inputan seperti titik koma. pemulihan error seperti ini disebut dengan pemulihan $Panic - Mode$.

2.8 FIRST dan FOLLOW

Rancangan dari kedua parser top-down dan bottom-up dibantu dengan dua fungsi, $FIRST$ dan $FOLLOW$, yang menghubungkan dengan grammar G. Selama proses parsing top-down, fungsi $FIRST$ dan $FOLLOW$ memperbolehkan kita untuk memilih *production* yang mana yang ingin diterapkan, berdasarkan pada simbol inputan berikutnya. Selama pemulihan *Panic Mode*, himpunan-himpunan dari token yang dihasilkan oleh $FOLLOW$ dapat digunakan sebagai sinkronisasi token-token.

Diberikan definisi $FIRST(\alpha)$, dimana α adalah sembarang string dari simbol grammar, menjadi himpunan terminal yang memulai string diturunkan dari α. Jika $\alpha \stackrel{*}{\Rightarrow} E$, maka E juga adalah $FIRST(\alpha)$. Sebagai contoh pada Gambar 2.14, $A \stackrel{*}{\Rightarrow} c\gamma$, jadi c adalah pada $FIRST(A)$.



Gambar 2.14: Terminal c adalah pada $FIRST(A)$ dan a adalah pada $FOLLOW(A)$

Untuk meninjau kembali bagaimana $FIRST$ dapat digunakan selama prediktif parsing, pertimbangkan 2 A-production $A \rightarrow \alpha \mid \beta$, dimana $FIRST(A)$ dan $FIRST(\beta)$ adalah himpunan disjoint. Kemudian kita dapat memilih antara A-production ini dengan melihat pada simbol inputan a selanjutnya, karena a dapat berada pada paling banyak 1 dari fungsi $FIRST(A)$ dan $FIRST(\beta)$. Untuk instansiasi, jika a berada pada $FIRST(\beta)$ pilih produksi

$A \rightarrow \beta$. Ide ini kan di ulas lebih lanjut pada pembahasan selanjutnya yaitu grammar LL(1) .

Diberikan definisi *FOLLOW* (*A*) , untuk nonterminal *A*, menjadi himpunan terhadap terminal a yang dapat muncul pada bagian terkanan dari beberapa bentuk sentensial; hal ini, himpunan dari terminal a sedemikian rupa muncul dalam bentuk turunan $S \overset{*}{\Rightarrow} \alpha A a \beta$, untuk beberapa α dan β seperti yang ditunjukkan pada Gambar 2.14. Perlu diperhatikan bahwa disana mungkin terdapa simbol antara *A* dan a pada suatu waktu selama proses derivasi. tetap jika demikian, mereka diturunkan dari *E* dan dianggap string kosong. Sebagai tambahan , jika *A* menjadi simbol terkanan dalam beberapa bentuk sentensial, maka $ ada dalam fungsi *FOLLOW* (*A*) ; dimana $ adalah simbol karakter penanda khusus *"endmarker"* yang mengasumsikan tidak menjadi simbol terhadap grammar apapun.

Untuk mengoperasikan fungsi *FIRST* (*X*) untuk seluruh simbol grammar *X*, terapakanaturan-aturan berikut selama tidak ada terminal atau *E* dapat ditambahkan ke fungsi himpunan*FIRST* apapun.

7. Jika *X* adalah sebuah terminal, maka *FIRST* (*X*) = *X* .

8. Jika *X* adalah nonterminal dan $X \rightarrow Y_1 Y_2 ... Y_k$ adalah produksi untuk beberapa $k \geq 1$, maka tempatkan a pada *FIRST* (*X*) jika untuk beberapa i, a berada pada *FIRST* (Y_i) , dan *E* berada pada semua fungsi *FIRST* (Y_1) , ..., *FIRST* (Y_{i-1}) ; dalam hal ini, $Y_1 ... Y_{i-1} \overset{*}{\Rightarrow} E$. Jika *E* ada pada fungsi *FIRST* (Y_j) untuk semua $j = 1, 2, ... , k,$ maka

   tambahkan *E* ke *FIRST* (*X*) . Sebagai contoh, semua yang ada pada fungsi *FIRST* (Y_1) pasti berada pada fungsi *FIRST* (*X*) . Jika Y_1 tidak menurunkan *E,* maka tidak ada yang ditambakan ke *FIRST* (*X*) , tetapi jika $Y_1 \overset{*}{=} E$ maka tambahkan *FIRST* (Y_2) dan seterusnya.

9. Jika $X \rightarrow E$ adalah produksi, maka tambahkan *E* ke *FIRST* (*X*)

Sekarang kita dapat mengoperasikan fungsi *FIRST* untuk string apapun $X_1 X_2 ... X_n$ sebagai berikut. Tambahkan ke *FIRST* ($X_1 X_2 ... X_n$) semua yang bukan simbol *E* dari *FIRST* (X_1) . Juga tambahkan yang bukan simbol *E* dari *FIRST* (X_2) jika *E* berada di *FIRST* (X_1) ; yang bukan simbol *E* dari *FIRST* (X_3) jika *E* berada di *FIRST* (X_1) dan *FIRST* (X_2) ; dan seterusnya. Terakhir; tambahkan *E* ke *FIRST* ($X_1 X_2 ... X_n$) jika, untuk semua i, *E* berada di *FIRST* (X_i) .

Untuk komputasi *FOLLOW* (*A*) untuk semua nonterminal *A*, terapkan atura-aturan berikut selama tidak ada yang dapat ditambahkan ke himpunan *FOLLOW* apapun.

1. Letakkan $ pada *FOLLOW* (*S*) , dimana *S* adalah simbol mulai (*start*) , dan $ adalah kanan input *endmaker* .

2. Jika terdapat produksi $A \rightarrow \alpha B \beta$, maka segala pada *FIRST* (β) kecuali *E* berada di *FOLLOW* (*B*)

3. Jika terdapat produksi $A \rightarrow \alpha B$, atau produksi $A \rightarrow \alpha B \beta$, dimana *FIRST* (β) berisi *E* maka segala pada *FOLLOW* (*A*) ada di *FOLLOW* (*B*) .

Contoh : Diberikan grammar non-left-recursive sebagai berikut :

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \textbf{id}
\end{array}
$$

Maka :

1. *FIRST* (*F*) = *FIRST* (*T*) = *FIRST* (*E*) = {(, **id**}. Perlu diperhatikan bahwa dua produksi untuk *F* memiliki tubuh yang dimulai dengan dua terminal simbol, **id** dan buka kurung. *T* hanya memiliki 1 produksi, dan tubuhnya dimulai dengan *F* . Karena *F* tidak menurunkan *E*, *FIRST* (*T*) harus sama sebagai *FIRST* (*F*) . Argument yang sama mencakup *FIRST* (*E*) .

2. *FIRST* (E') = {+ , *E* }. Alasannya adalah bahwa satu dari dua produksi untukE' mempunyai tubuh yang dimulai dengan terminal + , dan tubuh lainnya adalah *E* .nonterminal apapun yang menurunkan *E*, kita tempatkan *E* pada *FIRST* untuk nonterminal tersebut.

3. *FIRST* (T') = {* , *E* } . Alasannya adalah dianalogikan dengan *FIRST* (E') .

4. *FOLLOW* (*E*) = *FOLLOW* (E') = {) , $} . Karena *E* adalah simbol mulai, *FOLLOW* (*E*) harus berisi $. tubuh produksi (*E*) menjelaskan mengapa tutup kurung berada pada *FOLLOW* (*E*) .Untuk E', Perlu diperhatikan bahwa nonterminal ini kemunculannya hanya pada akhir dari tubuh produksi *E* (*E Productions*) . Dengan demikian, *FOLLOW* (E') harus menjadi sama sebagai *FOLLOW* (*E*) .

5. *FOLLOW* (*T*) = *FOLLOW* (T') = { + ,) , $}. Perhatikan bahwa *T* muncul diikuti dengan E'. Dengan demikan, semua yang ada pada *FIRST* (E') harus berada pada *FOLLOW*(*T*) kecuali *E* ; yang menjelaskan simbol +. Bagaimanapun juga, karena *FIRST*(E') berisi *E* (contoh:$E' \stackrel{*}{\Rightarrow} E$) , dan E' adalah string keseluruhan mengikuti *T* pada tubuh dari *E*-produksi, semua pada *FOLLOW* (*E*) juga harus berada pada *FOLLOW*(*T*)Ýang menjelaskan keberadaan simbol $ dan tutup kurung. Untuk sebagai T', karena itu hanya muncul pada akhir dari *T*-produksi, itu harus menjadi *FOLLOW* (T') = *FOLLOW* (*T*) .

6. *FOLLOW* (*F*) = { + , *,) , $}. Alasanya adalah dianalogikan bahwa untuk *T* pada point 5.

2.9 Tata bahasa (Grammmar) LL(1)

Grammar LL(i̇) merupakan kelas grammar yang dapat digunakan untuk merancang salah satu bahasa pemrograman, meskipun diperlukan kehati-hatian dalam menulis tata bahasa yang bersesuaian untuk sebuah *source language*. Huruf "L" pertama pada LL(1) memperbolehkan pemindaian input dari kiri ke kanan, "L" yang kedua untuk menghasilkan sebuah derivasi terkiri, dan angka "1" digunakan untuk satu simbol input dari *lookahead* pada tiap-tiap tahapan untuk membuat aksi keputusan parsing.

Grammar G adalah LL(1) jika dan hanya jika saat $A\ \alpha|\beta$ adalah 2 produksi **distinct** dari *G*, berlaku kondisi berikut:

1. Yang bukan terminal a lakukan derivasi pada α dan β dimulai dengan string a
2. Setidaknya satu dari α dan β dapat menderivasikan string kosong.
3. Jika $\beta \stackrel{*}{\Rightarrow} E$, makan α tidak menurunkan string permulaan apapun dengan sebuah terminal pada $FOLLOW(A)$.Demikian pula, jika $\alpha \stackrel{*}{\Rightarrow} E$, maka β tidak menurunkan string permulaan apapun dengan sebuah terminal pada $FOLLOW(A)$.

Dua kondisi pertama adalah ekivalen terhadap statement $FIRST(\alpha)$ dan $FIRST(\beta)$ tersebut yang merupakan himpunan saling lepas (*disjoint sets*). Kondisi ketiga ekivalen dengan menyatakan bahwa jika E ada pada $FIRST(\beta)$,maka $FIRST(\alpha)$ dan $FOLLOW(A)$ adalah himpunan saling lepas, dan demikian juga jika E ada pada $FIRST(\alpha)$ .//[2mm] Algoritme selanjutnya adalah mengumpulkan informasi dari himpunan-himpunan $FIRST$ dan $FOLLOW$ kedalam sebuah tabel parsing prediktif $M[A, a]$, sebuah array dua dimensi, dimana A adalah suatu nonterminal, dan a adalah suatu terminal atau simbol $, input *endmarker*. Algoritme ini berdasarkan pada ide berikut : produksi $A \rightarrow \alpha$ terpilih jika simbol input selanjutnya a ada pada $FIRST(\alpha)$.Komplikasi hanya terjadi saat $\alpha = E$ atau, lebih secara umumnya, $\alpha \Rightarrow E$. Dalam masalah ini, kita harus memilih kembali $A \rightarrow \alpha$, jika simbol input saat ini berada pada $FOLLOW(A)$, atau jika $ berada pada input yang telah tercapai dan $ ada pada $FOLLOW(A)$.

Algorithm 2 Rancang tabel parsing prediktif

INPUT: Grammar G
OUTPUT: Tabel Parsing M

METHOD: Untuk tiap-tiap produksi $A \rightarrow \alpha$ dari grammar, lakukan tahapan berikut:

1. Untuk tiap-tiap terminal a pada $FIRST(\alpha)$,tambahkan $A \rightarrow \alpha$ ke $M[A, a]$.
2. Jika E berada pada $FIRST(\alpha)$, maka untuk tiap-tiap terminal b pada $FOLLOW(A)$, tambahkan $A \rightarrow \alpha$ ke $M[A, b]$. Jika E berada pada $FIRST(\alpha)$ dan $ berada pada $FOLLOW(A)$, tambahkan $A \rightarrow \alpha$ ke $M[A, \$]$ demikian juga seterusnya.

Jika, seletelah melakukan tahapan di atas, tidak terdapat produksi pada semua dalam $M[A, a]$, maka set $M[A, a]$ ke **error** (biasanya secara normal didalam tabel dinyatakan dengan inputan kosong).

Tahapan yang diperlukan dalam membuat tabel parsing antara lain :

1. Menghilangkan langsung rekursif kiri dengan menulis kembali produksi tersebut sebagai contoh : produksi $A \rightarrow A\alpha|\beta$ dapat ditulis ulang dengan aturan produksi *non-left-recursive* :

$$A \rightarrow \beta A^{I}$$
$$A^{I} \rightarrow \alpha A|E$$

   tanpa merubah string yang dapat diturunkan dari A.
2. Tentukan $FIRST$ dan $FOLLOW$ dari tiap-tiap produksi
3. Buat tabel parsing dengan menggunakan Algoritma rancangan tabel parsing prediktif

Contoh : dinyatakan grammar sebagai berikut:

$$E \rightarrow E + T|T$$
$$T \rightarrow T * F|F$$

$$F \rightarrow (E)|id$$

Menggunakan aturan produksi *non-left-recursive* maka dapat ditulis kembali menjadi grammar berikut:

$$E \rightarrow TE'$$
$$E' \rightarrow +TE'|E$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT'|E$$
$$F \rightarrow (E)|id$$

Kita akan menggunakan *non-left-recursive* grammar untuk mencari input simbol dengan fungsi $FIRST$ dan $FOLLOW$ sehingga diperoleh seperi berikut :

Produksi	FIRST	FOLLOW
E → TEI	{(, id }	{$,) }
EI →	{+ , E}	{$,) }
T → FT	{(, id}	{+,$,) }
TI → ∗FT	{* , E}	{+, $,)
F → (E)|id	{(, id}	{*, +, $,)

CONTOH : Penerapan Algoritma Rancangan tabel parsing prediktif terhadap *non-left-recursive* Grammar di atas menghasilkan tabel parsing berikut :

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow$ **id**			$F \rightarrow (E)$		

Gambar 2.15: Tabel Parsing M

Mengingan produksi $E \rightarrow TE'$, karena $FIRST(E)=FIRST(T)=FIRST(F)=\{(, id\}$ produksi ini ditambahkan ke $M[E, (]$ dan $M[E, id]$. Produksi $E' \rightarrow +TE'$ ditambahkan ke $M[E', +]$ karena $FIRST(+TE')=\{+\}$. Karena $FOLLOW(E')=\{\$,)\}$, produksi $E' \rightarrow E'$ ditambahkan ke $M[E',)]$ dan $M[E', \$]$.

Algoritma Rancang Tabel Prediktif dapat diterapkan pada tata bahasa G apapun untuk menghasilkan tabel parsing M.

2.10 Shift and Reduce Parsing

Sesuai dengan namanya metode ini mengkombinasikan antara penggeseran dan pengurangan parsing. Contoh sebelumnya jika kita kerjakan maka otak kita lebih mudah mengerjakan lewat mekanisme 'top down' sambil melihat grammarnya. Dalam contoh di atas, ada penjumlahan (x+y) dan perkalian (dengan z). Dengan menggunakan left hand first diperoleh urutan berikut ini:

```
E -> E * E          (r2)
  -> E * z          (r3)
  -> E + E * z      (r1)
  -> E + y * z      (r3)
  -> x + y * z      (r3)
```

Gambar 2.16: Urutan grammar

Perhatikan aturan grammar yang diterapkan di dalam kurung di samping kanan proses top-down. Pertama-tama digunakan aturan perkalian yang menghasilkan dua non-terminal E. E yang kanan dengan menerapkan rule no.3 diperoleh konversi dari E ke id (dalam hal ini z). Langkah berikutnya dengan menggunakan aturan no.1 dimana E menurunkan E+E (simbol -> diistilahkan turunan (derrive). Teruskan dengan menggunakan rule no. 3 akan merubah non-terminal menjadi terminal berturut-turut x dan y. Karena hasil akhir sama dengan soal, maka dapat dikatakan instruksi x+y*z dapat diterima parser.

Kita mungkin mudah mengerjakan dengan top-down method ini. Bagaimana dengan komputer? Tentu saja berbeda dengan otak kita. Untuk itulah metode shift and reduce layak diperhitungkan. Cara kerja shift and reduce adalah sebagai berikut:

- Tulis operasi yang akan dicek grammar-nya.
- Lakukan operasi shift untuk memisahkan non-terminal/variabel yang akan diisikan id.
- Lakukan operasi reduce untuk mengganti variabel menjadi terminal.

```
 1    . x + y * z     shift
 2    x . + y * z     reduce(r3)
 3    E . + y * z     shift
 4    E + . y * z     shift
 5    E + y . * z     reduce(r3)
 6    E + E . * z     shift
 7    E + E * . z     shift
 8    E + E * z .     reduce(r3)
 9    E + E * E .     reduce(r2)        emit multiply
10    E + E .         reduce(r1)        emit add
11    E .             accept
```

Perhatikan 11 langkah di atas, yang merupakan tipikal operasi *shift and reduce*. Pada langkah pertama, instuksi akan siap-siap melakukan proses "shift" dimana x bergeser ke kiri. Selanjutnya, x yang telah berada di kiri dikonversi menjadi terminal/variabel E. Berikutnya langkah ke3 dan 4 dua buah proses shift yaitu untuk plus dan y dilanjutkan dengan reduce y menjadi terminal E. Langkah 8, 9, dan 10 bermaksud mereduksi instruksi mengikut grammar sehingga dihasilkan E yang artinya instruksi x+y*z dapat diterima (accepted). Berikutnya akan dicoba lewat perangkat lunak Lex and Yacc, khususnya bagian Yacc yang bertanggung jawab mengurusi Grammar.

2.11 Parsing LR Sederhana

Kebanyakan parser yang sering ditemui saat ini adalah tipe parser *bottom-up* yang merupakah konsep dari parsing LR(k); L diartikan sebagai proses pemindaian dari kiri kekanan terhadap inputan, R untuk membuat sebuah derivasi terkiri secara berlawanan dari sebelumnya (*reverse*), dan k untuk memandang kemuka jumlah dari simbol-simbol input yang digunakan dalam membuat keputusan parsing. Dalam hal ini $k = 0$ atau $k = 1$ diterapkan secara peraktiknya, dan sebagai pertimbagangan parser dengan $k \trianglelefteq 1$. Saat (k) dihilangkan , maka diasumsikan menjadi1.

Pada pembahasan ini akan memperkenalkan konsep dasar dari parsing **LR** dan metode termudah untuk membuat *shift-reduce parser*, yang disebut dengan **LR sederhana** atau disingkat dengan **SLR**.

2.11.1 Mengapa LR Parser?

LR parser meruapakan *table-driven*. Secara intuitif, agar grammar menjadi LR yang bersesuaian dengan parser shift-reduce dari kiri kekanan parser dapat mengenali bentuk kalimat terkanan saat muncul teratas pada tumpukan. Terdapat beberapa alasan menarik pada LR parsing yaitu:

- LR parser yang telah dibuat secara virtual dapat mengenali semua rancangan bahasa pemrograman yang dapat dituliskan dengan context-free grammar.
- Metode LR parser diketahui meruapakan *nonbacktracking shift-reduce parsing*, dengan demikian dapat diimplementasikan dengan efisien.
- LR parser dapat mendeteksi kesalahan sintak sesegera mungkin dengan melakukan pemindaian inputan dari kiri ke kanan.
- Kelas dari grammar yang dapat diparse menggunakan metode LR adalah superset dari grammar yang dapat diparse dengan prediktif atau dengan metode LL. dengan grammar menjadi LR(k), kita mesti dapat mengenali terjadinya bentuk sentensial kanan dari sisi kanan sebuah produksi, dengan memandang kemuka inputan simbol k. Perbandingan dengan LL(k) adalah dimana kita mesti dapat mengenali penggunaan produksi yang hanya melihat simbol pertama k dari apa yang diturunkan pada sisi kanan. Dengan demikian , bukan hal yang mengejutkan LR grammar dapat menyatakan banyak bahasa ketimbang LL grammar.

2.11.2 Items dan LR(0) Automaton

Bagaimana cara kerja *shift-reduce* parser tau kapan untuk melakukan proses *shift* dan kapan melakukan *reduce*? sebagai contoh, dengan *stack content* $\$T$ dan simbol input selanjutnya $*$ pada Gambar 2.17

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	shift
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	reduce by $F \to \mathbf{id}$
$ F	$* \mathbf{id}_2$ $	reduce by $T \to F$
$ T	$* \mathbf{id}_2$ $	shift
$ $T *$	$\mathbf{id}_2$ $	shift
$ $T * \mathbf{id}_2$	$	reduce by $F \to \mathbf{id}$
$ $T * F$	$	reduce by $T \to T * F$
$ T	$	reduce by $E \to T$
$ E	$	accept

Gambar 2.17: Konfigurasi shift-reduce parser

bagaimana parser mengetahui tidak ada penanganan pada T yang berada di atas dari stack, sehingga aksi yang tepat dilakukan adalah melakukan shift bukan reduce T ke E?

LR parser membuat keputusan shift-reduce dengan memperbaiki *states* untuk tetap menelsuri dimana kita berada dalam sebuah parse. *States* menyatakan himpunan-himpunan dari "Items". Sebuah $LR(0)$*item* dari grammar G adalah produksi dari G dengan sebuah tanda titik pada beberapa posisi dari tubuh grammar. Dengan demikian , produksi $A \to XYZ$ menghasilkan empat items berikut:

$$A \to \cdot XYZ$$
$$A \to X \cdot YZ$$
$$A \to XY \cdot Z$$
$$A \to XYZ \cdot$$

Produksi $A \to E$ hanya menghasilkan satu item, $A \to \cdot$. Secara intuitif, sebuah item mengindikasi berapa banyak produksi yang telah kita lihat memberikan point dalam proses parsing. Sebagai contoh, item $A \to \cdot XYZ$ mengindikasi bahwa kita berharap untuk melihat string dapat diturunkan dari XYZ selanjutnya pada input. Item $A \to X \cdot YZ$ mengindikasi bahwa kita hanya telah melihat pada sebuah input string dapat diturunkan dari X dan kita berharap selanjutnya untuk melihat sebuah string dapat diturunkan dari YZ. Item $A \to XYZ \cdot$ menindikasi bahwa kita telah melihat tubuh XYZ dan hal itu mungkin waktunya untuk melakukan reduce XYZ ke A.

Representing Items Sets

Sebuah parser generator yang menghasilkan parser bottom-up mungkin memerlukan pernyataan items dan sets pada item secara benar. Perlu diperhatikan sebuah item dapat dinyatakan dengan pasangan integer, yang pertama adalah satu dari produksi grammar yang mendasarinya, dan yang kedua adalah posisi dari tanda titik. Himpunan dari itemsdapat dinyatakan dengan sebuah list dari pasangan tersebut. Bagaimanapun juga sepertiyang kita lihat, himpunan dari item yang diperlukan sering termuat items "closure", dimana tanda titik pada permulaan tubuh produksi. Ini selalu dapat menjadi perancanganulang dari item lainya dalam hipunan, dan kita tidak dapat memuat mereka dalam list.

Satu *Collection* pada himpunan-himpunan *items* **LR**(0), disebut dengan *Collection canonical*, penyedia dasar rancangan *deterministic finite automaton* yang digunakan untuk membuat

keputusan parsing. Seperti sebuah automaton disebut dengan **LR**(0) *automaton*. Secara khusus, tiap-tiap state dari **LR**(0) automaton menyatakan sebuah himpunan dari items dalam canonical collection **LR**(0). Automaton untuk ekspresi grammar E berikut:

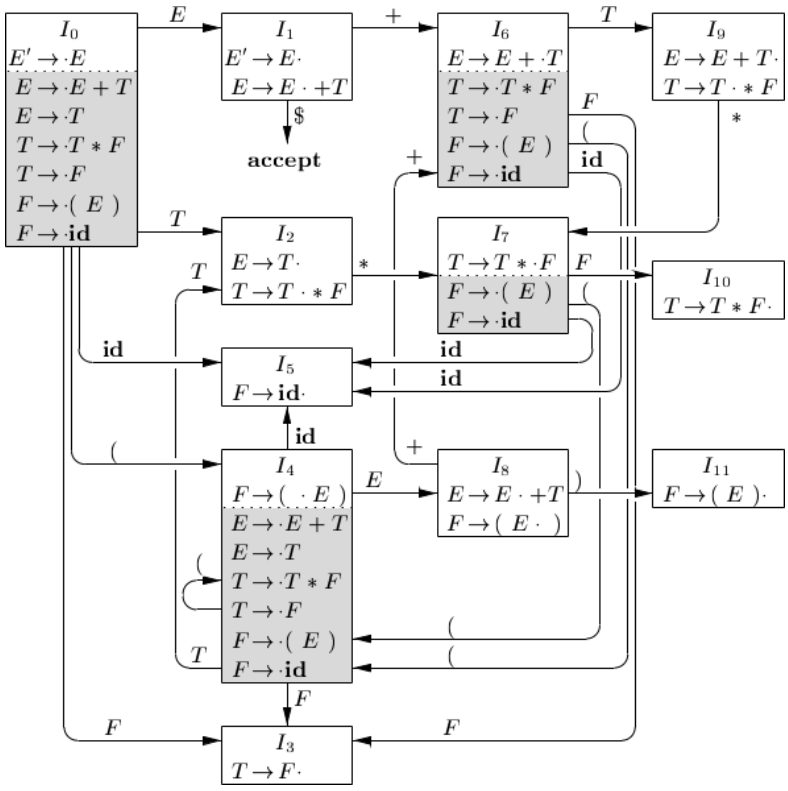
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

ditunjukkan pada Gambar 2.18, berfungsi sebagai contoh berjalanya untuk memberikan informasi collection **LR**(0) untuk sebuah grammar.

Untuk membuat collection canonical **LR**(0) terhadap sebuah grammar, kita tentukan sebuah *augmented* grammar (Augmented grammar adalah grammar start (posisi teratas), lalu berikan tanda ', Contohnya : grammar $A\ BCD$ maka augmented nya $A^{I}\ A$) dan dua fungsi, *CLOSURE* dan *GOTO*. Jika G adalah sebuah grammar dengan simbol awalan S, maka G^{I} , *augmented grammar* untuk G, G adalah dengan simbol awalan baru S^{I} dan produksi $S^{I}\ S$. Tujuan dari produksi awal ini adalah untuk mengindikasi parser saat parsing harus berhenti danmemberikan pemberitahuan kepada *acceptanceinput*. *Accaptence* terjadi saat dan hanya saat
paser akan direduksi dengan $S^{I} \rightarrow S$.

Closure dari Himpunan Item : jika I adalah suatu himpunan dari item-item untuk sebuah grammar G, maka $CLOSURE(I)$ adalah himpunan dari item-item yang telah dibangun dari I dengan dua aturan berikut:

1. Inisialisasi, tambahkan tiap-tiap item pada I terhadap $CLOSURE(I)$.
2. Jika $A\ \alpha B\beta$ berada di $CLOSURE(I)$ dan $B\ \gamma$ adalah sebuah produksi, maka tambahkan item $B\ \gamma$ ke $CLOSURE(I)$, jika sebelumnya tidak ada. Terapkan aturan ini selama tidak ada lagi item-item baru dapat ditambahkan ke $CLOSURE(I)$



Gambar 2.18: LR(0) automaton untuk ekspresi grammar E

Secara intuitif, $A \rightarrow \alpha \cdot B\beta$ pada $CLOSURE(I)$ mengindikasi bahwa pada sebagian point dalam proses parsing, kita berpikir mungkin selanjutnya melihat sebuah substring dapat diturunkan dari $B\beta$ sebagai input. Substring dapat diturunkan dari $B\beta$ akan memiliki prefix yang dapat diturunkan dari B dengan menerapkan satu dari $B-$produksi. Untuk alasan itu kita tambahkan items untuk semua $B-$produksi; Jika $B \rightarrow \gamma$ adalah sebuah produksi, kita juga memasukkan $B \rightarrow \cdot\gamma$ dalam $CLOSURE(I)$.

Contoh : diberikan ekspresi augmented grammar berikut:

$$
\begin{aligned}
E' &\rightarrow E \\
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid id
\end{aligned}
$$

Jika I adalah himpunan dari satu item $\{[E' \rightarrow \cdot E]\}$, maka $CLOSURE(I)$ berisikan himpunan dari item I_0 pada Gambar 2.18. Untuk melihat bagaimana closure ditentukan, $E' \rightarrow \cdot E$ dimasukkan kedalam $CLOSURE(I)$ dengan aturan (1). Karena terdapat sebuah E disebelah kanan tanda titik, kita tambahkan $E-$produksi dengan tanda titik pada akhir sebelah kiri:$E \rightarrow \cdot E + T$ dan $E \rightarrow \cdot T$. Sekarang tedapat T secara langsung disebelah kanan titik, maka kita tambahkan $T \rightarrow \cdot T * F$ dan $T \rightarrow \cdot F$. Selanjutnya, F disebelah kanan tanda titik mememaksa kita untuk menambahkan $F \rightarrow \cdot(E)$ dan $F \rightarrow \cdot id$, tetapi tidak ada items lain yang perlu ditambahkan.

Cara mudah untuk implementasi fungsi *closure* adalah membiarkan boolean array ditambahkan (*added*), diindeks dengan nonterminals G, yang berarti $added[B]$ manjadi **true** jika dan saat kita tambahkan item $B \rightarrow \cdot\gamma$ untuk tiap-tiap B-produksi $B \rightarrow \gamma\cdot$.

```
SetOfItems CLOSURE(I) {
        J = I;
        repeat
                for ( each item A → α·Bβ in J )
                        for ( each production B → γ of G )
                                if ( B → ·γ is not in J )
                                        add B → ·γ to J;
        until no more items are added to J on one round;
        return J;
}
```

Gambar 2.19: Komputasi CLOSURE

Perlu diperhatikan bahwa jika satu B produksi ditambahkan ke closure terhadap I dengan tanda titik pada akhir kiri, maka semua B produksi akan ditabahkan dengan cara yang sama keclosure. Oleh karena itu dalam keadaan tertentu tidak perlu membuat list item $B\gamma$ yang ditambahkan ke I oleh $CLOSURE$. Kita bagi semua himpunan dari item yang penting kedalamdua kelas:

1. *Kernel Items*: inisialisasi item $S' \rightarrow .S$, dan semua item yang terdapat tanda titik tetapi bukan pada bagian akhir sebelah kiri.

2. *Nonkernel Items*: semua item dengan tanda titiknya pada bagian akhir kiri, kecuali untuk $S' \rightarrow \cdot S$

Sebagai tambahannya, tiap-tiap himpunan untuk kepentingan items dibentuk dengan mengambil closure dari himpinan kernel items, artinya items yang ditambahkan dalam closure tidak pernah dapat menejadi items kernel. Dengan dmikian, kita dapat menyatakan himpunan-himpunan items pada penyimapanan yang sangat kecil dengan kita membuang semua items nonkernel, diketahui bahwa mereka dapat dibuat ulang dengan proses closure. Pada Gambar 2.18, bagian yang disamarkan pada kotak adalah item nonkernel untuk sebuah state.

FUNGSI GOTO

Fungsi yang berguna kedua adalah GOTO(I,X) dimana I adalah himpunan dari items dan X adalah simbol grammar. GOTO(I,X) didefinisikan menjadi closure terhadap himpanan dari seluruh items $[A \rightarrow \alpha X \cdot \beta]$ dengan demikian $[A \rightarrow \alpha \cdot X\beta]$ berada pada I. Secara intuitif, fungsi GOTO digunakan untuk mendefinisikan transisi pada **LR**(0) automaton untuk sebuah grammar. State dari automaton bersesuaian terhadap himpunan dari item-item, dan GOTO(I,X) menentukan transisi dari state untuk I dibawah input X.

CONTOH: Jika I adalah himpunan dari dua items $\{[E' \rightarrow E \cdot], [E \rightarrow E \cdot + T]\}$, maka GOTO($I$,+) mengandung items berikut:

$$
\begin{aligned}
E &\rightarrow E + \cdot T \\
T &\rightarrow \cdot T * F \\
T &\rightarrow \cdot F \\
F &\rightarrow \cdot (E) \\
F &\rightarrow \cdot \mathbf{id}
\end{aligned}
$$

Untuk menentukan GOTO (I,+) dengan memeriksa I untuk items + berada tepat pada bagian terkanan dari tanda titi. $E' \rightarrow E \cdot$ tidak seperti sebuah item, tetapi $E \rightarrow E \cdot + T$ adalah item. Kita pindahkan tanda titik didepan simbol + untuk memperoleh $E \rightarrow E + \cdot T$ dan kemedian mengambil closure dari singleton set ini.

CONTOH: Canonical collection dari sets **LR**(0) items untuk grammar;

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid id
\end{aligned}
$$

dan fungsi GOTO ditunjukkan pada Gambar 2.18. GOTO di encode dengan transisi pada gambar tersebut.

Menggunakan LR(0) Automaton

Pusat ide dibalik "LR Sederhana", atau SLR, adalah rancangan parsing dari grammar **LR**(0) automaton. state-state dari automaton ini adalah sets item dari canonical **LR**(0) collection, dan diberikan transisi dengan fungsi GOTO.

State awal dari **LR**(0) automaton adalah CLOSURE($\{[S' \rightarrow \cdot S]\}$), dimana S' adalah simbol awal dari augmented grammar. Semua state adalah *accepting* state. Kita katakan "*state j*" merujuk

pada state yang bersesuaian dengan set item I_j.

Bagaimana keterkaitan **LR**(0) automaton dengan *shift-reduce* dalam membuat keputusan? Keputusan *shift-reduce* dapat dibuat misalkan dengan asumsi bahwa string γ dari simbol grammar mengambil **LR**(0) automaton dari state awal 0 ke beberapa state j. Maka, lakukan shift pada simbol input a berikutnya jika state j memiliki sebuah transisi pada a. Sebaliknya, kita pilih reduce; items dalam state j akan memberitahu kita produksi yang mana untuk digunakan.

CONTOH: Tabel berikut mengilustrasikan aksi dari sebuah shift-reduce parser pada **id * id**, menggunakan **LR**(0) automaton terhadap Gambar 2.18 di atas. Kita gunakan stack untuk menahan states; untuk kejelasan, simbol grammar bersamaan dengan states pada stack muncul pada kolom SYMBOLS. Pada baris pertama (1), stack menahan state awal 0 dari automaton; simbol yang sesuai adalah ditandai dengan $ *bottom-of-stack*. Simbol input berikutnya adalah **id** dan state 0 memiliki sebuah transisi pada **id** ke state 5. Oleh karena itu kita lakukan shift. Pada bari (2), state 5 (simbol **id**) telah di push kedalam stack. Tidak ada transisi dari state 5 pada input *, dengan demikian kita lakukan reduce. Dari item [$F \rightarrow$ **id**] pada state 5, reduksi dialakukan dengan produksi $F \rightarrow$ **id**. Dengan simbol-simbol, sebuah reduksi diimplementasikan dengan melakukan popping pada tubuh produksi dari stack (baris (2), tubuh produksi **id**) dan melakukan push pada head dari produksi (dalam kasus ini, F). Dengan state-state yang ada, kita lakukan pop pada state 5 untuk simbol **id**, yang membawa pada state 0 ke posisi atas dan melihat untuk transisi F, sebagai head dari produksi.

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	$	**id** $*$ **id** $	shift to 5
(2)	0 5	$ **id**	$*$ **id** $	reduce by $F \rightarrow$ **id**
(3)	0 3	$ F	$*$ **id** $	reduce by $T \rightarrow F$
(4)	0 2	$ T	$*$ **id** $	shift to 7
(5)	0 2 7	$ $T*$	**id** $	shift to 5
(6)	0 2 7 5	$ $T*$ **id**	$	reduce by $F \rightarrow$ **id**
(7)	0 2 7 10	$ $T*F$	$	reduce by $T \rightarrow T*F$
(8)	0 2	$ T	$	reduce by $E \rightarrow T$
(9)	0 1	$ E	$	accept

Gambar 2.20: Tabel parse **id * id**

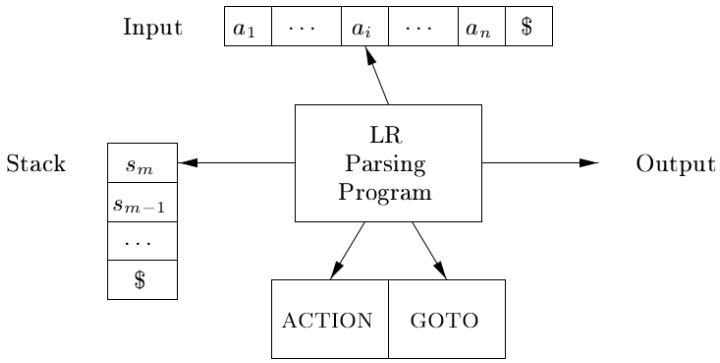
Pada Gambar 2.18, state 0 memiliki transisi pada F ke state 3, jadi kita push state 3, bersama dengan simbol F; lihat baris (3).

Contoh lainnya, pada line (5), dengan state 7 (simbol *) pada bagian atas stack. State ini memiliki sebuah transisi ke state 5 dengan input **id**, jadi kita push state 5 (simbol **id**). State 5 tidak memiliki transisi, jadi kita reduce dengan $F \rightarrow$ **id**. Saat kita pop state 5 untuk tubuh **id**, state 7 datang ke bagian atas dari stack. karena state 7 tidak memiliki transisi pada F ke state 10, kita push state 10 (simbol F).

2.11.3 Algoritme LR-Parsing

Skematik dari sebuah LR parser ditunjukkan pada Gambar 2.21 Itu terdiri dari sebuah input dan output, sebuah stack, driver program, dan sebuah tabel parsing yang memiliki 2 bagian (ACTION dan GOTO). Driver program adalah sama untuk semua LR parsers; hanya merubah tabel parsing daru satu parser ke parser lainnya. Program parser membaca karakterk dari sebuah penyangga

input satu persatu. Dimana shift-reduce parser adalah sebuah shift simbol, LR parser shift sebuah *state*. Tiap-tiap state berisi kesimpulan informasi pada stack dibawahnya.



Gambar 2.21: Model LR Parser

stack menahan sebuah rentetan dari states, $s_0 s_1 ... s_m$, dimana s_m diletakkan paling atas stack. Pada metode SLR, stack menahan states dari **LR**(0) automaton; mirip dengan metode canonical LR dan LALR. Dengan merancang, tiap-tiap state memiliki kesesuaian simbol grammar. Mengingat bahwa state-state sesuai pada set dari item-item, dan disana terdapat sebuah transisi dari state i ke state j jika GOTO(I_i, X) = I_j. Semua transisi ke state j harus sama untuk simbol grammar X. Dengan demikian, tiap-tiap state, kecuali stat awal 0, memiliki simbol grammar yang unik terasosiasi dengannya.

Struktur dari Tabel Parsing LR

Tabel parsing terdiri dari dua bagian: sebuah fungsi ACTION dan GOTO.

1. Fungsi ACTION bertindak sebagai argumen state i dan terminal a atau $. Nilai dari ACTION[$i$, a] dapat mempunyai satu dari 4 bentuk berikut:

   (a) Shift j. Dimana j adalah sebuah state. Parser secara efektif melakukan shift input a ke stack dengan menggunakan state j untuk menyatakan a.

   (b) Reduce $A \rightarrow \beta$. Dengan efektik parser melakukan reduce β pada bagian atas stack ke head A.

   (c) Accept. Parser melakukan accept terhadap input dan menyelesaikan proses parsing.

   (d) Error. Parser mengetahui sebuah kesalahan (Error) pada inputan dan melakukan koreksi.

2. Melakukan perluasan terhadap fungsi GOTO, ditentukan pada himpunan items, ke state: jika GOTO[I_i, A] = I_j, maka GOTO juga memetakan sebuah state i dan sebuah nonterminal A ke state j.

Konfigurasi LR-Parser

Untuk menggambarkan prilaku dari sebuah LR parser dibantu dengan notasi yang menyatakan state secara utuh dari parser yaitu stack dan sisa input yang telah digunakan (*remaining input*). *Konfigurasi* parser adalah pasangan dari komponen ($s_o s_1 ... s_m$, $a_i a_{i+1} ... a_n$\$) dimana komponen pertama adalah konten stack (bagian atas terkanan), dan komponen kedua adalah *remaining input*. Konfigurasi ini menyatakan bentuk kalimat terkanan (right-sentential) $X_1 X_2 ... X_m a_i a_{i+1} ... a_n$ secara esensialnya cara yang sama sebagai parser shift-reduce; satu-

satunya perbedaan adalah sebagai pengganti simbol grammar, stack menahanstate dari grammar yang simbolnya dapat dipulihkan. Dalam hal ini, X_i adalah simbol grammaryang dinyatakan dengan state s_i. Perhatikan bahwa s_o, state awal dari parser, tidak menyatakansebuah simbol grammar, dan kegunaannya sebagai penanda stack dengan posisi di bawah.

Prilaku LR Parser

Perpindahan parser berikutnya dari konfigursi diatas adalah dengan ditentukannya pembacaan a_i, simbol input saat ini, dan s_m, state teratas dari stack, dan kemudian berkaitan dengan entri ACTION[s_m, a_i] dalam tindakan tabel parsing. Konfigurasi dihasilkan setelah tiap-tiap dari empat tipe perpindahan sebagai berikut:

1. Jika ACTION[s_m, a_i] = shift s, parser menjalankan perpindahan shift; shift tersebut adalah next state a kedalam stack, masukkan konfigurasi ($s_o s_1 \ldots s_m$, $a_i a_{i+1} \ldots a_n$\$) simbol a_i tidak perlu menahan stack, karena dia dapat dipulihkan dari s, jika diperlukan (secara prakteknya tidak pernah dilakukan). Simbol inputan saat ini adalah a_{i+1}.

2. Jika ACTION[s_m, a_i] = reduce $A \rightarrow \beta$, kemudian parser menjalankan pepindahan reduce, masukkan konfigurasi berikut ($s_o s_1 \ldots s_{m-r} s$, $a_i a_{i+1} \ldots a_n$\$) dimana r adalah panjang dari β, dan s = GOTO[s_{m-r}, A]. Disini parser pertama kali melakukan pop simbol state r terhadap stack, mengekspos state s_{m-r}. Kemudian parser melakukan push s, masukkan GOTO[s_{m-r}, A] kedalam stack. Simbol inputan saat ini tida berubahah dalam perpindahan reduce. Untuk parser LR kita mesti merancang, $X_{m-r+1} \ldots X_m$, rentetan dari simbol grammar bersesuaian dengan state yang telah dilakukan pop pada stack, akan selalu cocok dengan β, sisi kanan dari proses reduksi sebuah produksi.

   Output dari LR parser diciptakan setelah perpindahan reduce dengan menjalankan semantic action diasosiasikan dengan mereduksi produksi. Untuk saat ini, kita berasumsi output hanya terdiri dari printing reduksi sebuah produksi.

3. Jika ACTION[s_m, a_i] = accept, pasing telah selesai.

4. Jika ACTION[s_m, a_i] = error, parser telah menemukan sebuah kesalahan dan memanggil *error recovery routine*.

Algoritma LR-Parsing dirangkumkan dibawah ini; satu-satunya perbedaan antara satu LR parser dan yang lainnya adalah informasi yang termuat pada ACTION dan GOTO terhadap tabel parsing.

Algoritma LR-Parsing.

INPUT: Sebuah string input w dan tabel LR-parsing dengan fungsi ACTION dan GOTO untuk grammar G.

OUTPUT: Jika w berada dalam $L(G)$, tahapan reduksi terhadap parser bottom-up untuk w; sebaliknya, mengindikasi sebuah kesalahan.

METHOD: Inisialisasi, parser memiliki s_0 pada stacknya, dimana s_o adalah inisial state, dan w\$ dalam input buffer. Kemudian parser menjalankan program 2.22

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

Gambar 2.22: Program LR-Parsing

Contoh: Gambar 2.23 menunjukkan fungsi ACTION dan GOTO dari sebuah tabel LR-Parsing untuk grammar berikut:

(1)	$E \rightarrow E + T$	(4)	$T \rightarrow F$
(2)	$E \rightarrow T$	(5)	$F \rightarrow (E)$
(3)	$T \rightarrow T * F$	(6)	$F \rightarrow \mathbf{id}$

Keterangan dari kode-kode nya adalah :

1. si artinya shift dan stack state i,
2. rj artinya reduce dengan sejumlah produksi j.
3. acc artinya accept,
4. blank artinya error

Perlu diperhatikan nilai dari GOTO[s, a] untuk terminal a ditemukan pada field ACTION yang terhubung dengan shift pada input a untuk state s. field GOTO memberikan GOTO[s, A] untuk nonterminal A. Meskipun belum ada penjelasan bagaimana melakukan entri untuk tabel yang dintujukkan pada Gambar 2.23

STATE	ACTION						GOTO		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Gambar 2.23: Parsing tabel grammar

Pada input **id * id + id**, rentetan stack dan konten input ditunjukkan pada Gambar 2.24 pada gambar tersebut juga menunjukkan rentetan simbol grammar yang bersesuaian terhadap state yang menahan stack. Sebagai contoh, pada line (1) LR parser adalah state 0, inisial state dengan tidak terdapatnya simbol grammar, dan dengan **id** sebagai simbol input awal. Sebuah aksi pada baris 0 dan kolom **id** dari action field pada Gambar 2.23 adalah s5, yang maksudnya adalah shift dengan melakukan push pada state 5. Begitu juga apa yang terjadi pada line (2): state simbol 5 telah melakukan push kedalam stack, dan **id** telah dihapus dari input.

Kemudian, * menjadi simbol input saat ini, aksi terhadap state 5 pada input * direduksi dengan $F \rightarrow$ **id**. Satu simbol state dihilangkan dari stack. Kemudian state 0 di ekspos. Karena goto daristate 0 pada *F* adalah 3, dilakukan push pada state 3 ke stack. Sekarang kita memiliki konfigurasi pada line (3). Tiap-tiap sisa perpindahan yang belum dijalankan ditentukan dengan cara yang sama.

2.11.4 Merancang Tabel SLR-Parsing

Merancang tabel parser dengan metode SLR adalah titik awal yang bagus untuk mempelajari LR parsing. Kita mesti merujuk pada tabel parsing yang telah dirancang dengan metode ini sebagai sebuah tabel SLR, dan untuk sebuah LR parser menggunakan tabel SLR-parsing sebagai sebuah SLR parser.

Metode SLR dimulai dengan item LR(0) dan automata LR(0), yang telah di singgung pada pembahasan sebelumnya. Yaitu, diberikan sebuah grammar, *G*, augment *G* untuk menghasilkan G^{I}, dengan simbol awal baru S^{I}. Dari G^{I}, kita merancang *C*, canonical collection dari himpunan item untuk G^{I} bersamaan dengan fungsi GOTO.

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		**id** $*$ **id** $+$ **id** \$	shift
(2)	0 5	**id**	$*$ **id** $+$ **id** \$	reduce by $F \rightarrow$ **id**
(3)	0 3	F	$*$ **id** $+$ **id** \$	reduce by $T \rightarrow F$
(4)	0 2	T	$*$ **id** $+$ **id** \$	shift
(5)	0 2 7	$T*$	**id** $+$ **id** \$	shift
(6)	0 2 7 5	$T*$ **id**	$+$ **id** \$	reduce by $F \rightarrow$ **id**
(7)	0 2 7 10	$T*F$	$+$ **id** \$	reduce by $T \rightarrow T*F$
(8)	0 2	T	$+$ **id** \$	reduce by $E \rightarrow T$
(9)	0 1	E	$+$ **id** \$	shift
(10)	0 1 6	$E+$	**id** \$	shift
(11)	0 1 6 5	$E+$ **id**	\$	reduce by $F \rightarrow$ **id**
(12)	0 1 6 3	$E+F$	\$	reduce by $T \rightarrow F$
(13)	0 1 6 9	$E+T$	\$	reduce by $E \rightarrow E+T$
(14)	0 1	E	\$	accept

Gambar 2.24: Perpindahan LR parser pada **id * id + id**

ACTION dan GOTO masuk kedalam tabel parsing kemudian dirancang menggunakan algoritma berikut. Itu mengharuskan kita untuk mengetahui FOLLOW(A) untuk tiap-tiap nonterminal A dari sebuah grammar.

Tabel parsing terdiri dari fungsi ACTION dan GOTO yang ditentukan dengan algoritma 3 disebut degan *tabel untuk G SLR*(1). LR parser yang menggunakan SLR(1) untuk tabel G disebut dengan parser SLR(1) untuk G, dan grammar yang memiliki sebuah parsing tabel SLR(1) dikatakan sebagai SLR(1). Kita biasanya mengabaikan aturan "(1)" setelah "SLR", karena tidak terdapat kesepakatan dengan parser yang memiliki lebih dari satu simbol pada lookahead.

CONTOH : Diberikan augmented grammar seperti di bawah ini untuk membuat tabel SLR

$$E \rightarrow \cdot E$$
$$E \rightarrow \cdot E + T$$
$$E \rightarrow \cdot T$$
$$T \rightarrow \cdot T * F$$
$$T \rightarrow \cdot F$$
$$F \rightarrow \cdot (E)$$
$$F \rightarrow \cdot \textbf{id}$$

canonical collection sets items LR(0) untuk grammar tersebut sebelumnya dintunjukkan pada Gambar 2.18. Pertamakali pertimbangkan set pada items I_0: Item $F \rightarrow \cdot(E)$ membakitkan entri ACTION[0, (] = shift 4, dan item F **id** ke entri ACTION[0, **id**] = shift 5, item lainnya dalam I_0 tidak menimbulkan aksi. Sekarang pertimbangkan I_1:

$$E \rightarrow \cdot E$$
$$E \rightarrow \cdot E + T$$

item pertama menimbulkan ACTION[1, \$] = accept, dan yang kedua menimbulkan ACTION[1, +] = shift 6. Selanjutnya pertimbangkan I_2:

$$E \rightarrow \cdot T$$
$$T \rightarrow \cdot T * F$$

Karena FOLLOW(E) = {$, +,)}, adalah item pertama membuat ACTION[2, $] = ACTION[2, +] = ACTION[2,)] = reduce $E \rightarrow T$

item kedua membuat ACTION[2, *] = shift 7.

Algorithm 3 Merancang parsing tabel SLR

INPUT: Augmented grammar G'

OUTPUT: Tabel parsing SLR fungsi ACTION dan GOTO untuk G'

METHOD: :

1. Konstruk $C = \{I_0, I_1, ..., I_n\}$, collection terhadap himpunan item LR(0) untuk G'
2. State i dibangun dari I_i. Aksi parsing untuk state i ditentukan dengan cara berikut:
   (a) Jika $[A \rightarrow \alpha . a\beta]$ ada pada I_i dan GOTO(I_i, a)= I_j, maka set ACTION[i, a] ke "shift j". a mesti jadi sebuah terminal.
   (b) JIka $[A \rightarrow \alpha .]$ ada pada I_i, maka set ACTION[i, a] ke "reduce $A \rightarrow \alpha$" untuk semua a dalam FOLLOW(A); A mungkin tidak menjadi S'.
   (c) Jika $[S' \rightarrow S\cdot]$ ada pada I_i, maka set ACTION[$i,$ $] ke "accept."

   Jika penerapan aturan di atas terdapat kesalahan-kesalahan, dapat kita katakan bahwa grammar bukanlah SLR(1). Dalam masalah ini algoritma gagal dalam menghasilkan sebuah parse.
3. Transisi goto untuk state i telah dirancang untuk semua nonterminals A menggunakan aturan: Jika GOTO(I_i, A) = I_j, maka GOTO[i, A] = j.
4. Semua entri yang tidak didefinisikan dengan aturan (2) dan (3) akan menyebabkan "error".
5. Inisial state pada parser adalah satu yang dirancang dari set items berisikan $[S' \rightarrow \cdot S]$

LATIHAN : Diberikan context-free grammar sebagai berikut:

$$S \rightarrow SS+ | SS* | a$$

dan string $aa+a*$.

- Buatlah derivasi terkiri untuk string tersebut.
- Buatlah derivasi terkanab untuk string tersebut.
- Buatlah parse tree untuk string tersebut.
- Apakah grammar tersebut Ambigu atau Tidak? Jelaskan!
- Deksripsikan bahasa yang dihasilkan oleh grammar tersebut.

BAB 3
INTERPRETASI

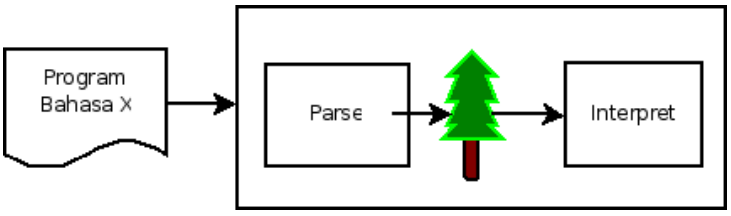
Setelah lexing dan parsing terdapat juga *abstract syntax tree* (ast) pada sebuah program sebagai struktur data dalam memory. Tetapi sebuah program perlu untuk dijalankan. Cara yang paling sederhana untuk menjalankan sebuah program adalah *interpretasi*. Interpretasi dilakukan dengan sebuah program yang disebut dengan *interpreter*, yang mengambil ast pada sebuah program dan menjalankannya dengan cara melakukan inspeksi pada pohon sintak untuk mengetahui instruksi apa saja yang dilakukan. Mirip dengan cara manusia melakukan evaluasi pada sebuah ekspresi matematika: Kita memasukkan nilai pada variabel-variabel yang tersedia dan mengevaluasinya bit per bit, dimulai dengan tada kurung terdalam dan berpindah keluar sampai kita mempunyai hasil dari ekspresi tersebut. Kita kemudian dapat mengulang proses ini dengan memasukkan nilai yang lain terhadap variabel.

3.1 Proses Interpreter

Interpreter membaca source dalam bahasa X (misalnya file.php, program.py, dsb). Interpreter akan menjalankan program input tersebut, dan menghasilkan output. Kira-kira seperti ini diagramnya 3.1

Gambar 3.1: Diagram Interpreter

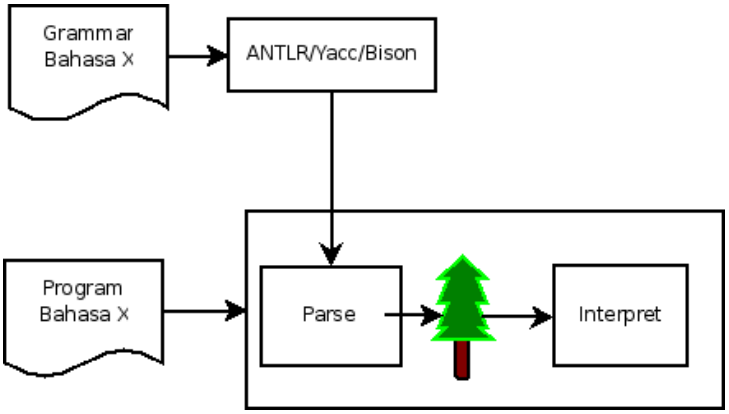
Sebuah interpreter terdiri atas bagian parser, dan interpreter. Parser menghasilkan sebuah tree yang disebut dengan parse tree. Jadi, isi sebuah interpreter bisa digambarkan seperti ini 3.2:



Gambar 3.2: Isi sebuah interpreter

Bagian yang umumnya memakan banyak waktu adalah menuliskan bagian parser, oleh karena itu banyak program parser generator yang dikembangkan (beberapa di antaranya YACC, Bison,dan

ANTLR). Sebuah parser generator hanya perlu diberikan grammar sebuah bahasa, lalu akan menghasilkan kode program untuk parsing seperti yang ditunjukkan pada Gambar 3.3. Kode program yang dihasilkan bisa dikompilasi bersama-sama dengan kode program kita yang lain.



Gambar 3.3: Proses ANTLR/Yacc/Bison

Dengan adanya parser generator, kita hanya perlu berkonsentrasi pada dua hal: seperti apa syntax bahasa kita, dan bagaimana mengeksekusi tree yang dihasilkan oleh parser untuk bahasa tersebut.

Bahasa paling sederhana: KALKULATOR

Kita akan mengimplementasikan bahasa yang sangat sederhana, yang hanya berfungsi sebagai kalkulator. Pertama kita akan membuat versi interpreter, lalu membuat versi compilernya. Grammar kalkulator ini sangat sederhana, hanya sebuah ekspresi '+', '-' dan '*'. Saya sengaja tidak membuatkan '/' untuk bahan latihan. Ketika dijalankan program akan mencetak setiap ekspresi yang dievaluasi. Jadi program:

$$1 + 2$$
$$2 * 3$$
$$(8 - 2) * (7 - 2)$$

akan menghasilkan: 3, 6, dan 40

ANTLR

Untuk memudahkan, dalam implementasi ini kita gunakan tools ANTLR http://www.antlr.org/ yang berbasis GUI yaitu Antlrworks. Tools ini sangat mudah dipakai dan sudah cukup matang. Versi command line ANTLR sudah dirilis sejak 1992 dan GUI modern dikembangkan sejak 2005. Sebelumnya sebenarnya sudah ada versi GUI, tapi masih kurang matang, sehingga dibuatlah versi GUI yang modern.

Bagi Anda yang masih kurang lancar dalam memahami grammar, ANTLR sangat bisa membantu, Anda bisa mencoba langsung grammar Anda sebelum mulai membuat program satu baris pun. Jika ada fitur yang saya jelaskan tapi Anda belum paham, Anda bisa mencoba-coba mengubah grammar dan langsung mencoba melihat visualisasinya.

> ANTLR dan ANTLRWorks hanyalah salah satu tools yang tersedia. Jika Anda sudah mahir, tools apapun akan sama saja. Programmer yang baik tidak akan dibatasi oleh

tools.
Cara menjalankan ANTLRWorks tergantung pada OS yang Anda gunakan. Di Mac OS X/Windows, jika sudah diset dengan benar, Anda bisa mengklik file antlrworks-1.2.3.jar, dan GUI akan muncul. Jika cara tersebut gagal, mungkin Anda perlu menjalankan dari command line, caranya:

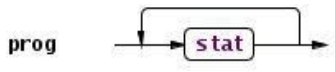
```
java -jar /path/to/antlrworks-1.2.3.jar
```

Berikut ini grammar yang akan kita pakai (ini versi 1, lihat file Expr_1.g) sebagai dasar bagi interpreter dan compiler kita (catatan, baris yang diawali // adalah komentar):

grammar Expr;

```
// START:stat
prog:  stat+ ;
stat:    expr NEWLINE
    |    NEWLINE
    ;
// END:stat
// START:expr
expr:   multExpr (('+'|'-') multExpr)*
    ;
multExpr
    :    atom ('*' atom)*
    ;
atom:    INT
    |    '(' expr ')'
    ;
// END:expr
// START:tokens
INT :  '0'..'9'+ ;
NEWLINE:'\r'? '\n' ;
WS  :  (' '|'\t'|'\n'|'\r')+ {skip();} ;
// END:tokens
```

Mari kita bahas grammarnya. Sebuah program <prog> terdiri atas banyak pernyataan <stat>+ (simbol plus artinya satu atau lebih), sebuah pernyataan boleh sebuah ekpresi <expr> atau sebuah baris kosong (NEWLINE). Anda juga bisa melihat Syntax Diagram dari sebuah rule, misalnya prog akan tampak seperti ini:

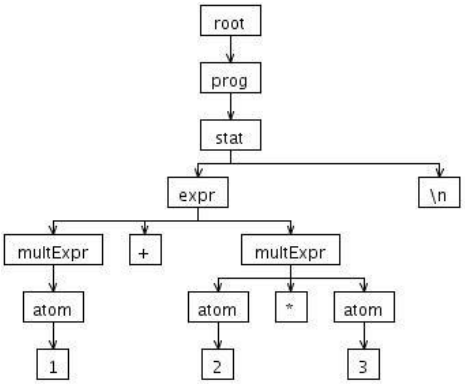


Karena Anda bisa melihat sendiri syntax diagram-nya di ANTLRWorks, saya tidak akan menampilkan sisanya.

Sebuah ekspresi terdiri dari pernyataan perkalian <multExpr> yang diikuti oleh plus/minus ekpresi yang lain. Tapi plus dan minus itu tidak wajib, jadi kita tambahkan * yang artinya nol atau lebih. Pernyataan perkalian sendiri terdiri atas <atom> yang (mungkin) dikalikan dengan atom lain, karena tidak harus dikalikan atom lain, maka kita tambahkan juga *. Aturan terakhir adalah <atom> yang bisa

berupa sebuah integer, atau ekspresi lain dalam tanya kurung.
Berikutnya kita perlu mendefinisikan token. Dalam kasus ini yang menjadi token adalah INT (0-9),NEWLINE (boleh $\backslash r\ \backslash n$ yang merupakan versi DOS atau $\backslash n$ saja yang merupakan versi UNIX). Kita juga mengijinkan spasi ada di antara ekspresi, jadi 1+2 sama dengan 1 + 2, untuk itu kita perlu mendefinisikan karakter apa saja yang perlu dilewatkan (skip), dalam kasus ini kita mengabaikan spasi, tab, dan karakter baris baru. Kita bisa langsung mencoba grammar ANTLR ini, dengan menggunakan ANTLRWorks. Coba pilih menu Debugger, lalu pilih Debug. Masukkan teks, misalnya 1+2. Perhatikan bahwa Anda harus mengakhiri sebuah ekspresi dengan karakter baris baru (enter) setelah ekspresi. Anda bisa menjalankan grammar langkah per langkah, atau langsung saja klik pada tombol END. Hasilnya sebuah pohon akan ditampilkan, pohon ini dinamakan Pohon Parsing (Parsing Tree). Silakan Anda mencoba-coba aneka ekspresi lain, termasuk ekspresi multi baris, supaya bisamelihat bagaimana pohon untuk setiap ekspresi.

Berikut ini adalah gambar pohon yang dihasilkan oleh 1 + 2 * 3. Gambar 3.4 pohon ini dihasilkan langsung oleh ANTLRWorks (saya tidak menggambarnya manual).



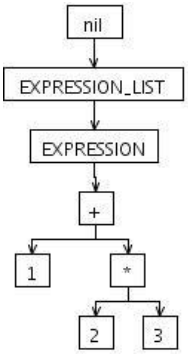
Gambar 3.4: Pohon parsing **1 + 2 * 3**

3.1.1 Abstract Syntax Tree (AST)

Jika diperhatikan karakter yang tidak penting juga masuk dalam pohon ini, yaitu karakter '$\backslash n$'. Ada juga node yang tidak penting, yaitu atom. Jika Anda membuat bahasa yang lebih rumit, misalnya bahasa C, maka karakter seperti '{ ' , '} ', ';' yang tidak penting juga akan masuk dalam parse tree. Kita mengatakan karakter itu tidak penting karena gunanya hanya untuk memisahkan blok, dan dalam bentuk pohon, sudah jelas bahwa blok-blok tersebut terpisah.

Sebelum masuk tahap pemrosesan, kita perlu mengubah pohon tersebut ke bentuk AST (abstract syntax tree) dengan membuang hal-hal yang tidak perlu, dan mungkin mengorganisasi tree menjadi bentuk yang lebih baik (lebih mudah diproses, misalnya menukar node kiri dan kanan, dsb). Jika kita menggunakan tools tertentu (misalnya Bison) kita menulis kode untuk mengubah parse tree menjadi AST, tapi untungnya ANTLR sudah cukup canggih, sehingga kita cukup menambahkan aturan untuk membuat pohon.

AST yang saya inginkan untuk 1 + 2 * 3 adalah:

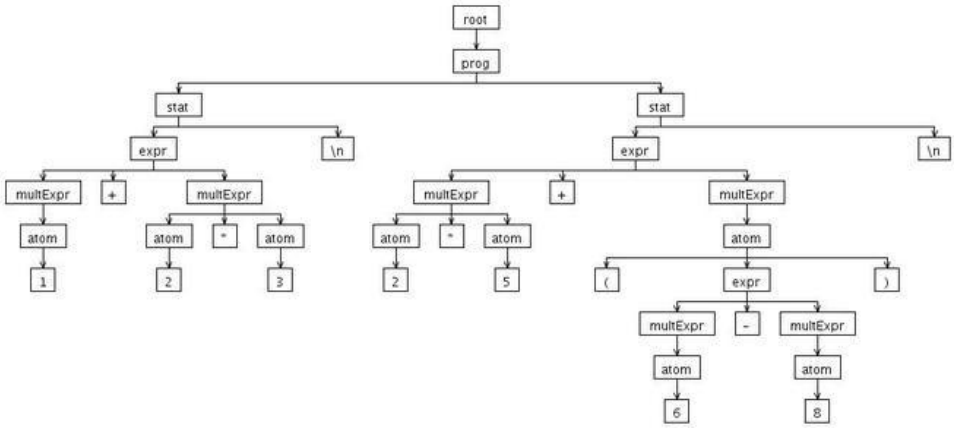


Gambar 3.5: AST untuk **1 + 2 * 3**

Dan jika ada dua baris (saya menambahkan satu baris baru: 2 * 5 + (6 - 8)) :

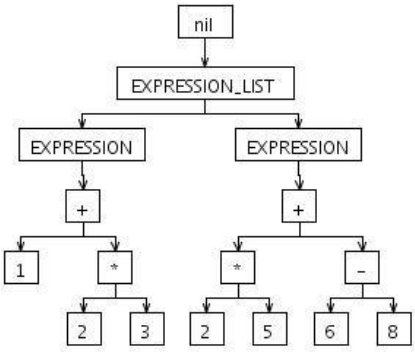
$$1 + 2 * 3$$
$$2 * 5 + (6 - 8)$$

Saat ini parse tree sudah terlalu kompleks:



Gambar 3.6: Pohon parse untuk **1 + 2 * 3 dan 2 * 5 + (6 - 8)**

Sedangkan AST yang diharapkan adalah seperti ini:



Gambar 3.7: AST **1 + 2 * 3 dan 2 * 5 + (6 - 8)**

Perhatikan bahwa tanda kurung juga tidak ada lagi (tidak lagi penting, karena dalam bentuk tree sudah jelas presedensinya). Ada beberapa perubahan yang diperlukan untuk menghasilkan AST. Pertama di bagian options, kita tambahkan opsi output = AST, dan ASTLabelType = CommonTree. Ini artinya kita meminta ANTLR menghasilkan AST, dengan tipe node AST-nya adalah CommonTree. Jika kita mau, kita juga bisa membuat sendiri jenis node untuk tree kita sendiri, tapi saat ini hal tersebut tidak diperlukan.

Berikutnya, kita perlu menentukan seperti apa bentuk tree kita. Dalam kasus ini, karena ada banyak ekspresi, saya ingin di bagian akar (root) adalah EXPRESSION_LIST, dan terdiri atas banyak EXPRESSION. Jika dilihat kembali, tidak ada rule yang bernama EXPRESSION ataupun EXPRESSION_LIST, jadi kita perlu mendefinisikan kedua nama tersebut di bagian tokens. Kita juga ingin agar INT menjadi nama node untuk literal integer. Setiap nama yang didefinisikan di bagian tokens akan memiliki konstanta bertipe integer di file parser yang dihasilkan ANTLR.

```
options {
 output = AST;
 ASTLabelType =CommonTree;
}
tokens {
        EXPRESSION_LIST;
        EXPRESSION;
        INT;
}
```

Kita perlu mengubah pohon stat menjadi pohon EXPRESSION_LIST, yang terdiri atas banyak EXPRESSION. Caranya kita gunakan operator -> milik ANTLR. Operator ini digunakan setelah sebuah rule, untuk menentukan bentuk tree untuk rule tersebut. Umumnya bentuknya adalah^(ROOT rules), yang artinya, jadikan ROOT sebagai akar dan rules sebagai anak-anaknya. Contohnya seperti ini:

```
// START:stat
prog:    stat+ -> ^(EXPRESSION_LIST stat+);
stat:    expr NEWLINE -> ^(EXPRESSION expr)
    |    NEWLINE
    ;
// END:stat
```

Bagian pertama stat+ ->^(EXPRESSION_LIST stat+); artinya, node-node yang berupa stat, dikumpulkan dibawah node yang bernama EXPRESSION_LIST. Bagian kedua expr NEWLINE ->^(EXPRESSION expr) artinya Node expr ditaruh dibawah node EXPRESSION, dan kita mengabaikan karakter NEWLINE.

Kita juga ingin agar '+' dan '-' memiliki nodenya sendiri. Jadi jika ada 11+12, kita ingin agar punya Node '+' yang anak kirinya adalah node 11 dan anak kanannya adalah node 12. Untuk hal ini, ANTLR memiliki shortcut. Agar '+' dan '-' menjadi node, cukup tambahkan karakter ^di bagian grammar yang ingin dijadikan root node.

```
// START:expr
```

```
expr:   multExpr (('+'|'-')^ multExpr)*;
```

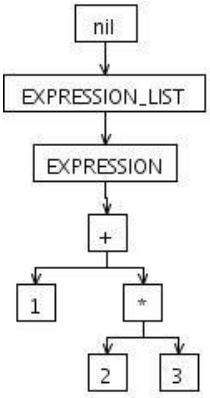
Sama halnya dengan '*', kita juga ingin agar * memiliki nodenya sendiri

```
multExpr
    :   atom ('*'^ atom)*
    ;
```

Dan terakhir, kita ingin membuang kurung buka dan kurung tutup, karena urutan evaluasi sudah jelas dalam tree. Untuk membuangnya, kita nyatakan bahwa '(' expr ')' -> expr, yang artinya: jika ada kurung buka, lalu expr, lalu kurung tutup, cukup hasilkan expr saja (dengan kata lain buang kurung buka dan tutupnya).

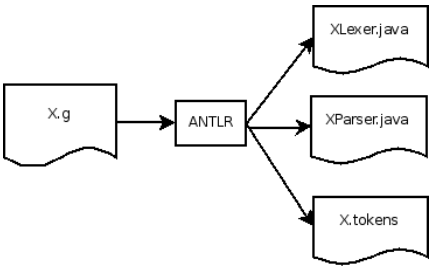
```
atom:   INT
    |   '(' expr ')' -> expr
    ;
// END:expr
```

Sekarang kita bisa mengklik tab AST di ANTLRWorks, dan hasil AST-nya adalah seperti ini.



Gambar 3.8: AST **1 + 2 * 3**

Nah sekarang kita sudah punya tree yang bagus. Berikutnya adalah bagaimana mengeksekusi tree tersebut? Ada dua cara: pertama adalah interpretasi, dan kedua adalah kompilasi. Namun kita perlu menghasilkan dulu source code parsernya, caranya cukup klik menu Generate lalu pilih Generate Code. ANTLR akan membuat tiga buah file, yaitu file Lexer, file Parser, dan file Tokens.



Gambar 3.9: Proses Generate ANTLR

Catatan

Java hanyalah salah satu bahasa yang didukung ANTLR. ANTLR juga bisa membuat parser dalam bahasa C, C#, Python, JavaScript, dan ActionScript.

Menulis Kode

Nah sekarang kita perlu menuliskan kode dalam bahasa Java. Kode-kode berikut ini ada pada file ExprLang.java. Setiap parser pasti punya kode inisialisasi, kode inisialisasi ini akan sama untuk aneka jenis bahasa, sehingga tidak akan dijelaskan lagi di bagian berikutnya.

```
public static void main(String argv[]) throws Exception {
        ExprLexer lex = new ExprLexer(new ANTLRFileStream(argv[0]));
        CommonTokenStream tokens = new CommonTokenStream(lex);
        ExprParser g = new ExprParser(tokens);
        ExprParser.prog_return r = g.prog();
        CommonTree ct = (CommonTree)r.getTree();
    }
```

Baris pertama dalam sebuah fungsi main membuat sebuah lexer (dalamkasus ini ExprLexer), yang fungsinya memecah string menjadi bagian-bagiannya (menjadi INT, '*', '+', '-', dsb). Baris kedua membuat object CommonTokenStream yang diberikan ke parser (ini adalah sebuah adapter, Anda tidak perlu mengetahui internalnya kecuali ingin mengubah kode ANTLR). Baris ketiga adalah bagian untuk mengkonstruksi parser, dan baris ke empat adalah baris yang penting, baris di mana proses parsing itu sendiri dipanggil:

```
ExprParser.prog_return r = g.prog();
```

Kita meminta nilai kembalian parser dimulai dari aturan prog. Setelah itu kita bisa mendapatkan Tree (AST) dengan menggunakan r.getTree(). Tree yang kita pakai adalah Tree standar bawaan ANTLR, jadi kita memakai CommonTree. Setelah memiliki root dari tree, kita bisa mengevaluasi ekspresi dengan mudah. Saya tidak akan menjelaskan semua method milik CommonTree, penjelasan lengkap ada di dokumentasinya di: http://www.antlr.org/api/Javaclassorg_1_1antlr_1_1runtime_1_1tree_1_1_common_tree.html.

Method-method yang akan saya pakai adalah: getChildren, getChilCount, getChild, getType, dan getText. Berikut ini penjelasan singkatnya:

1. Method getChildren untuk mendapatkan List of children yang bisa diiterasi menggunakan format loop Java (for (Tree x: e.getChildren()) {}). Sebagai catatan, Anda akan melihat banyak casting tipe Object ke CommonTree, ketika ANTLR ditulis, Java 1.5 belum dirilis, sehingga fitur Generic milik Java belum dipakai. Mereka saat ini sudah mulai beralih ke JDK 1.5.
2. Method getChildCount digunakan untuk mendapatkan jumlah anak. Berguna untuk menentukan apakah misalnya pernyataan if memiliki else atau tidak.
3. Method getChild digunakan untuk mendapatkan anak ke-n.

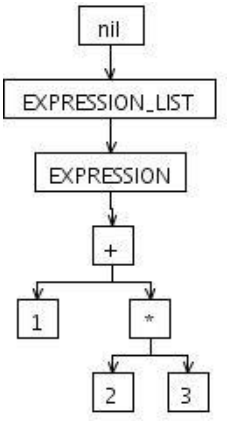
4. Method getType digunakan untuk mendapatkan konstanta integer tipe node. Nilainya terdefinisi (tidak nol) jika node tersebut diberi nama dibagian tokens. Hasil kembalian method ini bisa dibandingkan dengan konstanta integer yang dihasilkan ANTLR dalam format NamaGrammarLexer.KONSTANTA (dalam contoh ini misalnya ExprLexer.INT)

5. Method getText digunakan untuk mendapatkan teks node (misalnya node + akan memiliki teks +). Ketika nanti node variabel diperkenalkan, getText bisa digunakan untuk mendapatkan nama variabel.

Mari kita mulai memproses tree. Kita memiliki banyak ekspresi dalam satu file, maka kita buat method evaluateExprList, method ini hanya akan memanggil evaluateExpression yang tugasnya adalah mengevaluasi ekspresi itu sendiri.

```
void evaluateExprList(CommonTree exprlist) {
        for (Object e: exprlist.getChildren()) {
            System.out.println("Result: " +
            evaluateExpression((CommonTree)e));
        }
    }
```

Kalau kita lihat dari gambar AST yang dihasilkan oleh Antlr, misalnya pohon ini:



Gambar 3.10: Hasil dari ANTLR

kita melihat bahwa ada suatu node dengan nama EXPRESSION yang tidak terlalu berguna untuk saat ini. Gunanya hanyalah agar terlihat bahwa node di bawahnya adalah sebuah ekspresi. Saya sengaja membuat node ini untuk pengembangan versi berikutnya, di mana setiap baris belum tentu berisi ekspresi. Kita hanya perlu melewati node itu dengan mengambil anaknya yang pertama.

```
int evaluateExpression(CommonTree expr) {
        debug("Evaluate Expression "+expr.getText());
        return evaluateExpr((CommonTree)expr.getChild(0)) ;
    }
```

Fungsi evaluateExpr adalah fungsi yang melakukan komputasi. Ini sangat mudah, karena pada dasarnya hanya ada dua kasus: INTEGER, dan OPERATOR.

Pertama, jika isi node adalah integer, maka hasilnya adalah nilai integer itu sendiri (di Java kita memakai Integer.parseInt untuk mengubah string menjadi integer)

```
if (expr.getType()==ExprLexer.INT) {
        return Integer.parseInt(expr.getText());
    }
```

Kedua, jika isi node adalah operator ('+' atau '-' atau '*') berarti nilai ekspresi adalah nilai anak pertama dioperasikan (ditambah/dikurang/dikali) dengan anak kedua:

```
if (expr.getText().equals("+")) {
    return evaluateExpr((CommonTree)expr.getChild(0)) +
         evaluateExpr((CommonTree)expr.getChild(1));
}
if (expr.getText().equals("-")) {
    return evaluateExpr((CommonTree)expr.getChild(0)) -
        evaluateExpr((CommonTree)expr.getChild(1));
}

if (expr.getText().equals("*")) {
    return evaluateExpr((CommonTree)expr.getChild(0)) *
        evaluateExpr((CommonTree)expr.getChild(1));
}
```

Cara Berpikir

Perhatikan bahwa algoritma ini sangat sederhana, Anda hanya perlu berpikir: di Node ini saya harus melakukan apa? Anda jangan melihat kompleksitas semu. Jika Anda berpikir bahwa untuk membuat kalkulator seperti ini Anda harus memikirkan aneka kombinasi yang ada, seperti 4*7, 4+7, 4+(7), (4)+7, dst, maka cara berpikir Anda masih salah. Ingatlah bahwa proses parsing menghasilkan pohon, sehingga Anda harus berpikir bagaimana melakukan aksi dalam suatu node di pohon tersebut.

Mengkompilasi

Untuk mengkompilasi interpreter ini, Anda perlu file antlr-runtime-3.1.3.jar (tergantung versi terbaru saat ini). Anda perlu memberikan path ke file tersebut, di Linux/Mac/BSD, kira-kira seperti ini:

```
javac -classpath /path/to/antlr-runtime-3.1.3.jar:. ExprLang.java
```

di Windows, kira-kira seperti ini:

```
javac -classpath c:\path\to\antlr-runtime-3.1.3.jar;. ExprLang.java
```

Tentu saja kita juga bisa menggunakan Netbeans, atau Eclipse atau IDEapapun. Jangan lupa menambahkan antlr-runtime-3.1.3.jar ke bagian Library/Classpath.

Untuk menjalankan programnya dari command line:

```
java -classpath /path/to/antlr-runtime-3.1.3.jar:. ExprLang test.e
```

Di mana test.e adalah file teks yang berisi baris-baris ekspresi.

Selesai sudah interpreter yang sangat sederhana. Saya sengaja menyertakan pernyataan debug agar pembaca dapat memahami alur eksekusi. Total baris, tanpa debug hanyalah 60 baris, ditambah dengan file grammar Expr.g totalnya hanya 100 baris. Mudah bukan?

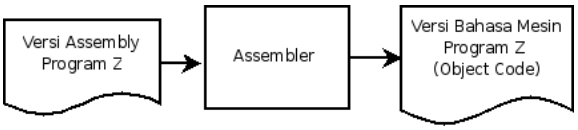
3.1.2 Membuat Compiler

Sebelumnya kita sudah mendefinisikan dan membuat interpreter untuk kalkulator. Sekarang bagaimana kalau kita ingin membuat compiler? Interpreter hanya mengeksekusi program, dan tidak menghasilkan output. Compiler perlu mengoutputkan kode dalam bahasa assembly, yang kemudian akan dikompilasi oleh assembler. Mempelajari assembly yang lengkap butuh waktu, apalagi jika kita ingin menargetkan berbagai prosessor, sehingga membuat compiler yang mengoutputkan ke assembly langsung tidaklah mudah. Karena sulitnya bahasa assembly, dalam tahap-tahap berikutnya, kita tidak akan mengoutputkan bahasa assembly langsung, tapi akan menggunakan bantuan LLVM (Low Level Virtual Machine). Tapi untuk tahap-tahap awal, saya bisa menunjukkan bagaimana output assembly langsung bisa dibuat, apalagi dalam bahasa yang sangat sederhana seperti ini.

Pertama akan dibahas dulu apa bedanya sebuah compiler dengan interpreter. Sebuah compiler menghasilkan kode assembly dari sebuah program. Compiler tidak menghasilkan file executable. Diberikan program Z dalam bahasa X, compiler membuat versi assembly dari program Z tersebut. Jadi compiler hanya menerjemahkan sebuah bahasa ke ekivalennya dalam bahasa assembly.

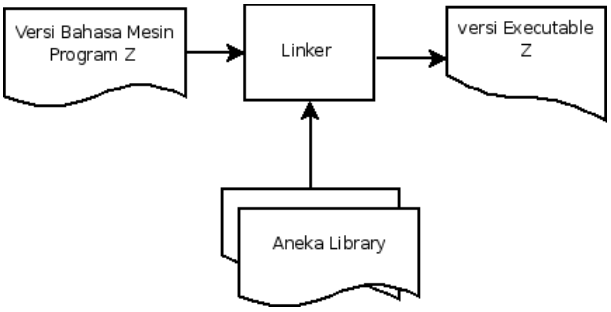
Gambar 3.11: Compiler menghasilkan kode assembly

Tugas mengubah ke bahasa mesin dilakukan oleh assembler. Assembler mengubah instruksi assembly menjadi bahasa mesin. File yang dihasilkan compiler ini disebut sebagai object code, file ini sudah dalam bahasa mesin, tapi belum bisa dieksekusi.



Gambar 3.12: Compiler menghasilkan object code

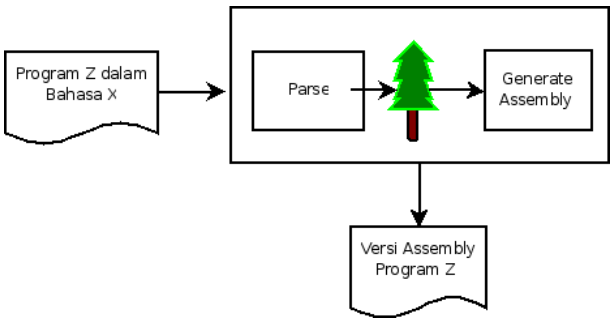
Mengapa belum bisa? karena masih ada fungsi-fungsi yang belum diketahui definisinya. Misalny, ketika Anda membuat program dalam C, apakah Anda mengimplementasikan sendiri fungsi printf? biasanya tidak, karena fungsi itu sudah ada di Library. Proses berikutnya adalah menggabungkan library dengan



Gambar 3.13: Program linker

object code untuk membentuk executable. Proses ini dilakukan oleh program linker.

Yang paling penting bagi kita bukanlah urutan proses tersebut, tapi apa yang ada dalam sebuah compiler yang membedakannya dari interpreter. Sebagian komponen compiler sama dengan interpreter. Bagian parser sama persis, sehingga kita bisa memakai parser dari tutorial bagian sebelumnya.



Gambar 3.14: Program linker

Algoritma

Algoritma dasar untuk compiler masih sama dengan interpreter. Pada versi interpreter, kita langsung menjalankan program, nah di versi compiler ini, kita menghasilkan teks di setiap langkah (teks bahasa assembly). Kode assembly tidak langsung dituliskan ke file, tapi ditampung dulu dalam sebuah StringBuffer.Untuk memudahkan, saya akan menggunakan pendekatan berbasis stack (stack based) dan bukan register based (akan saya jelaskan nanti alasannya). Jika Anda tertarik, Anda bisa membaca Wikipedia mengenai Stack machine dan Register machine.

Di awal kita perlu membuat sebuah template, yang merupakan kerangka program. Berikutnya, di setiap langkah kita menghasilkan instruksi yang sesuai, misalnya ketika menemui sebuah integer, kita menghasilkan instuksi untuk menaruh integer tersebut di stack. Ketika menemui instruksi untuk menambah dua ekspresi, kita panggil secara rekursif kode program kita untuk menghasilkan assembly bagi operand kiri dan operand kanan, lalu kita hasilkan instruksi untuk menjumlahkan kedua hasilnya.

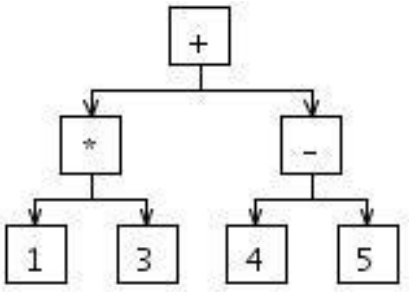
Memakai Assembly

Kita akan membatasi bahasan kita untuk Intel x86 32 bit saja. Untuk mengimplementasikan compiler dalam bahasa sederhana tersebut, kita hanya perlu 6 instruksi dan 2 register. Instruksi pertama adalah push <reg> atau push <nilai> untuk menaruh nilai ke stack. Lalu pasangannya adalah pop <reg> untuk menaruh isi stack ke register. Berikutnya kita perlu instruksi add untuk menjumlah, sub performing a corresponding memory-to-register operation on the same variable.untuk mengurangi, dan imull untuk mengalikan integer. Kita juga perlu instruksi call untuk memanggil fungsi printf milik library C, serta ret untuk kembali dari fungsi utama ke sistem operasi.

Kita akan menggunakan syntax assembly AT&T, dan saya hanya mengetes ini di Linux. Ada beberapa tutorial assembly untuk Linux, misalnya http://asm.sourceforge.net/howto/Assembly-HOWTO.html, Anda bisa membaca aneka tutorial jika Anda benar-benar blank mengenai assembly. Setiap file output pasti punya kerangka dasar seperti ini:

```
.section    .rodata
.mytext:
.string "Result %d\n"
.text
.globl  main
.type    main, @function
main:
/*aneka macam perintah akan diletakkan di sini*/
ret
.size    main, .-main
```

Untuk mengevaluasi ekspresi, kita selalu menggunakan stack. Contohnya begini, Jika kita memiliki (1 * 3) + (4 - 5), kita akan memiliki tree, dengan + sebagai akar (root), anak pertama akan mengandung subtree dengan * di root serta 1 dan 3 di anak, sedangkan anak kedua memiliki subtree dengan - di root serta 4 dan 5 sebagai anak.



Gambar 3.15: Pohon parse untuk **(1 * 3) + (4 - 5)**

Ketika menemui '+', kita ingin agar ekpresi di sebelah kiri (1 * 3) dievaluasi dulu, lalu sebelah kanan di evaluasi (4-5), dan hasil evaluasi keduanya kita jumlahkan. Pertama, kita evaluasi 1 * 2. Ketika menemui *, kita ingin agar bagian kiri (1) dan (3) dievaluasi, lalu hasilnya baru dikalikan. Ketika mengevaluasi 1, maka hasilnya

adalah instuksi assembly untuk menaruh 1 di stack, ketika menemui 3, maka hasilnya juga instruksi assembly untuk menaruh hasilnya di stack. Perhatikan awalan $ di syntax AT&T artinya angka 1 dan 3 merupakan literal:

```
pushl  $1
ISI STACK:
1
pushl  $3
ISI STACK:
1  3
```

Ketika menemui *, kita pop 2 angka dari stack, lalu kita kalikan kedua angka tersebut, lalu taruh hasilnya di stack:

```
popl e%ax
ISI STACK:
1
Isi EAX = 3

popl %ebx
ISI STACK:
kosong
Isi EBX = 1

imull %ebx, %eax ; artinya EAX = EAX * EBX
ISI STACK:
kosong
Isi EAX = 3

pushl %eax

ISI STACK:
3
```

Sekarang kita evaluasi (4-5), langkahnya sama dengan di atas (jika tidak yakin, Anda bisa menjalankan compilernya), di akhir, kita akan mendapati isi stack seperti ini:

```
ISI STACK:
3  -1
```

Lalu operasi penjumlahan dilakukan

```
popl %eax -->  ambil dari stack (-1)
popl %ebx -->  ambil dari stack (3)
addl %ebx, %eax --> EAX = EAX + EBX
pushl %eax  -->  masukkan  hasilnya  ke  stack
```

Di akhir, kita ingin mencetak hasilnya. Untuk mudahnya, kita akan menggunakan library C. Anda juga bisa menggunakan cara khusus sebuah OS, misalnya di DOS Anda bisa menggunakan INT 21 Fungsi 9 dan di Linux Anda bisa memanggil syscall write. Tapi cara-cara tersebut tidak portabel. Library C sudah tersedia di aneka OS

berbasis UNIX, jadi demi kesederhanaan artikel, saya akan memakai library C. Di C, mencetak sebuah integer mudah sekali, cukup printf("Result: %d\n", result). Di assembly ini juga tidak sulit, cukup perlu:

```
pushl %eax
push  $.mytext
call  printf
popl  %ebx
popl  %ebx
```

Passing parameter dalam assembly dapat dilakukan via register atau stack (tergantung calling convention, dan jumlah parameternya, tapi itu tidak penting sekarang). Dalam kasus printf kita perlu menggunakan stack, parameter untuk printf dalam kasus ini ada dua, yang pertama adalah format string "Result: %d n", yang saya letakkan di label .mytext, serta nilai integer yang akan kita cetak. Passing dilakukan terbalik, parameter terakhir dipush pertama.

Karena pushl %eax sudah dilakukan di akhir setiap ekspresi, maka kita tidak perlu mengulanginya. Seperti yang Anda lihat di bagian template, isi $.mytext adalah "Result: %d n". Instruksi call digunakan untuk memanggil printf. Fungsi printf di C merupakan fungsi khusus (jumlah parameternya bisa banyak), sehingga kita perlu membuang lagi nilai yang dipush dengan popl ke sembarang register (dalam hal ini saya pilih saja %ebx).

3.1.3 Menjalankan compiler

Anda bisa menjalankan compiler ini seperti menjalankan interpreter. Output compiler ini ada dua, yang pertama adalah file assembly .s (misalnya input adalah test.e, maka outputnya adalah test.e.s), dan file executable (file test.e.exe). Jika compiler gcc tidak tersedia di sistem, maka hanya satu saja outputnya (.s). Anda bisa melakukan assembling dan linking dengan:

```
gcc -m32 namafile.s -o namafile.exe
```

Secara otomatis program Java akan mencoba menjalankan perintah itu, tapi tidak akan berhasil jika gcc tidak ada di path. Parameter -m32 memaksakan agarkita menggunakan mode 32 bit meski di OS 64 bit (OS yang saya pakai 64 bit,tapi sebagian besar orang masih memakai 32 bit).performing a corresponding memory-to-register operation on the same variable.

Anda bisa mempelajari dan membandingkan dengan output sebuah program dalam bahasa C. Menggunakan compiler GCC, Anda bisa mengoutputkan kode assembly untuk sebuah program dalam bahasa C seperti ini:

```
gcc -S namafile.c
```

hasilnya adalah namafile.s. Sebenarnya gcc selalu menghasilkan file assembly .s, tapi file ini dibuat di direktori sementara. Dengan opsi -S kita meminta agar membuat

file .s di direktori saat ini dan meminta gcc berhenti setelah membuat file .s (tidak meneruskan tahap assembler dan linker).

Kode Program

Source code compiler lebih panjang dari source code interpreter (83 baris vs 60 baris). Kode-kode berikut ini ada pada file ExprComp.java. Di bagian main kita buat dulu template dasar file assembly yang akan dihasilkan:

```
StringBuffer result = new StringBuffer();
result.append(".section .rodata\n");
result.append(".mytext:\n");
result.append(".string \"Result %d\\n\"\n");
result.append(".text\n");
result.append(".globl main\n");
result.append(".type main, @function\n");
result.append("main:\n");
el.compile(result);
result.append("ret\n");
result.append(".size main, .-main\n");
```

Lalu dibagian evaluasi (method compileExpr), kita perlu menghasilkan assembly yang sesuai, untuk integer:

```
if (expr.getType()==ExprLexer.INT) {
        result.append("pushl $"+expr.getText()+"\n");
    return;
}
```

Untuk +,-,* semua perlu dua operand, jadi instruksi awalnya pasti sama, yaitu: hasilkan instruksi untuk kiri dan kanan, lalu pop dua operand ke eax dan ebx:

```
if (expr.getText().equals("+") || expr.getText().equals("-")
    || expr.getText().equals("*")) {
    compileExpr(result, (CommonTree)expr.getChild(0));
    compileExpr(result, (CommonTree)expr.getChild(1));
    result.append("popl %eax\n");
    result.append("popl %ebx\n");
}
```

Berikutnya mudah, kalau + hasilkan addl, kalau - hasilkan subl, dan kalau * hasilkan imull:

```
if (expr.getText().equals("+")) {
    result.append("addl %ebx, %eax\n");
}
if (expr.getText().equals("-")) {
    result.append("subl %eax, %ebx\n");
}
if (expr.getText().equals("*")) {
    result.append("imull %ebx, %eax\n");
}
```

Hasilnya kita kembalikan ke stack:

```
result.append("pushl %eax\n");performing a corresponding memory-to-register operation on the same variable.
```

Setelah sebuah ekspresi selesai, kita perlu mencetaknya

```
void compileExpression(StringBuffer result, CommonTree expr) {
    compileExpr(result, (CommonTree)expr.getChild(0));
    result.append("push $.mytext\n");
    result.append("call printf\n");
    result.append("popl %ebx\n");
    result.append("popl %ebx\n");
}
```

Jadi instruksi yang kita pakai benar-benar amat sedikit.

Program Java akan menyimpan hasil akhir ke file:

```
String asmname = argv[0]+".s";
FileWriter fw = new FileWriter(asmname);
PrintWriter pw = new PrintWriter(fw);
pw.println(result.toString());
fw.close();
```

Lalu hasilnya dikompilasi:

```
String command[] = {"gcc", "-m32", asmname, "-o", argv[0]+".exe"};
Process p = Runtime.getRuntime().exec(command);
p.waitFor();
```

Perhatikan bahwa jika kompilasi gagal atau berhasil, tidak akan ada pesan apapun. Anda hanya akan tahu bahwa kompilasi berhasil atau tidak dengan melihat apakah file .exe tercipta atau tidak. Silahkan tambahkan sendiri kode untuk melakukan pemeriksaan tersebut.

Instruksi yang dihasilkan oleh compiler ini sangat sederhana, namun sangat tidak efisien. Karena semua hanyalah konstanta (kita belum bisa menerima input dari user), maka sebenarnya yang penting hanyalah hasil akhir saja. Meski demikian, latihan ini penting sebelum masuk ke bahasa yang bisa memiliki variabel/identifier.

Ketika kita sudah bisa menerima input dari user, eksekusi berbasis stack kurang efisien. Sebuah prosessor memiliki beberapa register, dan penggunaan stack lebih lambat dari register. Misalnya kita punya 3 variabel, kita bisa meletakkan 3 variabel tersebut di register, tanpa perlu menyentuh stack sama sekali. Tapi masalah muncul ketika jumlah variabel semakin banyak. Jumlah register di prosessor terbatas (biasanya 16-32 register), jadi kita tetap perlu memakai stack ketika jumlah variabel semakin banyak. Kita harus dengan pintar mengatur, variabel apa yang masuk register dan apa yang masuk stack. Masalah ini dinamakan register allocation problem. Anda bisa membaca aneka buku dan paper untuk memahami masalah tersebut.

Kita juga harus memiliki pengetahuan assembly aneka prosessor untuk bisa membuat kode assembly yang baik. Masalahnya terutama adalah masalah optimasi, ada banyak cara untuk melakukan suatu hal (misalnya membagi dua bisa dilakukan dengan shift right satu bit), sebuah compiler yang baik harus bisa memilih instruksi terbaik untuk menghasilkan kode tercepat.Pada pembahasan berikutnya, kita akan mencoba bagaimana menggunakan LLVM, yang akan bisa mengoutputkan kode bahasa mesin, tapi kita sendiri tidak perlu memahami aneka prosessor. LLVM merupakan proyek yang sudah ada sejak 9 tahun yang lalu (tahun 2000), dan sudah didukung oleh banyak perusahaan besar (Adobe, Apple, dsb).

Tambahkan operator /, tangani kasus pembagian dengan 0.
Tambahkan unary operator + dan -

3.1.4 Input, Output, dan Variabel

Bahasa yang kita buat sebelumnya sangat sederhana, sampai-sampai tidak berguna secara praktis. Kita bisa menambah sedikit kegunaan dengan mengijinkan adanya VARIABEL dan instruksi untuk membaca INPUT, serta memberikan OUTPUT. Tujuannya adalah, kita bisa membuat interpreter dan compiler untuk bahasa seperti ini:

```
var panjang,lebar,luas,keliling
print 'input panjang'
input panjang
print 'input lebar'
input lebar
luas=panjang*lebar
print 'luas adalah '
print luas
keliling=(2*panjang)+(2*lebar)
print 'Keliling adalah '
print keliling
```

Perhatikan bahwa dalam bagian ini, kita akan memakai evaluator ekspresi yang sudah dibahas pada bagian interpretasi, jadi bagian sebelumnya berguna untuk membentuk sebuah compiler, meski bahasa yang dihasilkan tidak berguna secara praktis. Kita juga akan menambahkan Unary Operator (+ dan -) sehingga lebih berguna (sekaligus sebagai jawaban untuk latihan di bagian sebelumnya). Operator pembagian silahkan ditambahkan sendiri (ini sangat mudah). Saat ini kita hanya menangani integer, jadi bahasa ini akan kita namakan BULAT. Grammarnya sekarang menjadi 79 baris. Anda dapat melihat sendiri grammar lengkap di source code, kita hanya akan membahas yang berubah saja.

Sekarang ada lebih banyak token.

```
tokens {
```

```
    STATEMENT_LIST;
    DECLARATION;
    ASSIGNMENT;
    INPUT;
    PRINT;INT;
    STRING;
    ID;
}
```

Token yang penting adalah DECLARATION untuk deklarasi variabel (var panjang, lebar, luas, keliling), ASSIGNMENT untuk operasi assignment (luas = panjang*lebar), PRINT untuk mencetak string dan variabel (print luas dan print print 'luas adalah'), serta INPUT untuk meminta masukan (input panjang). Karena kita sekarang memiliki variabel, kita memperkenalkan token ID, dan karena print bisa digunakan untuk mencetak selain variabel (print 'hello'), kita definisikan juga tipe STRING.

Sebuah program terdiri atas deklarasi (satu baris), diikuti oleh satu atau lebih statement

```
prog:  declaration stat -> ^(STATEMENT_LIST declaration stat);
```

Sebuah statement bisa berupa assignment, input, atau print, dan boleh juga baris kosong

```
stat:    assignment NEWLINE -> assignment
    | input NEWLINE -> input
    | print NEWLINE ->print
    | NEWLINE
    ;
```

Sebuah deklarasi diawali dengan var dan diikuti dengan daftar variabel yang dipisahkan dengan koma

```
declaration
    :
    'var' ID ( ',' ID)* NEWLINE -> ^(DECLARATION ID)
```

Untuk meminta input dari pengguna, kita menggunakan input diikuti nama variabel:

```
input :
    'input' ID -> ^(INPUT ID)
    ;
```

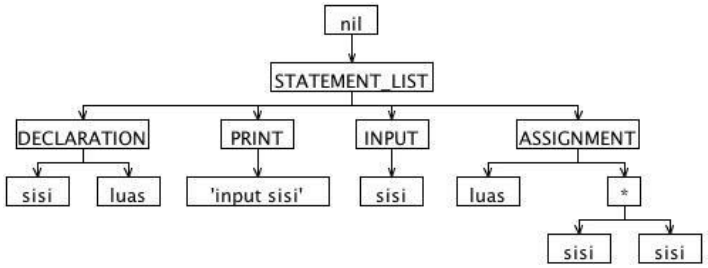
Instruksi print boleh diikuti oleh sebuah variabel atau sebuah string:

```
print :
    'print' ID -> ^(PRINT ID)
    | 'print' STRING -> ^(PRINT STRING)
    ;
```

Assignment dilakukan dalam bentuk namavar = ekspresi. Ekspresi ini sangat mirip dengan di pembahasan sebelumnya.

```
assignment
    :
    ID '=' expr -> ^(ASSIGNMENT ID expr ;
```

Secara visual, contoh tree adalah seperti ini:



Gambar 3.16: Pohon sintak

Satu-satunya perbedaan di dalam ekspresi adalah dukungan terhadap unary operator

```
multExpr
      :    unaryExpr ('*'^ unaryExpr)*
      ;

 unaryExpr
     : unary_op atom
     | atom
     ;

 unary_op:
     PLUS
     |MINUS
     ;

 PLUS    :    '+';
 MINUS :    '-';
```

Sebenarnya bisa saja plus/minus digunakan langsung seperti ini

```
unaryExpr
     : ('+'|'-')^ atom
     | atom
     ;
```

Tapi kita menggunakan cara sebelumnya untuk menunjukkan cara penulisan grammar yang baik (karena mungkin suatu hari ingin menambahkan operator unary yang lain). Untuk variabel, hanya karakter a-z saja yang diijinkan:

```
ID:    'a'..'z'+ ;
```

Untuk string, apapun yang ada antara tanda kutip diijinkan (kita tidak mendukung escape character apapun, seperti \n atau \t)

```
STRING :  '\''( ~('\'') )*'\'' ;
```

Menyimpan Variabel

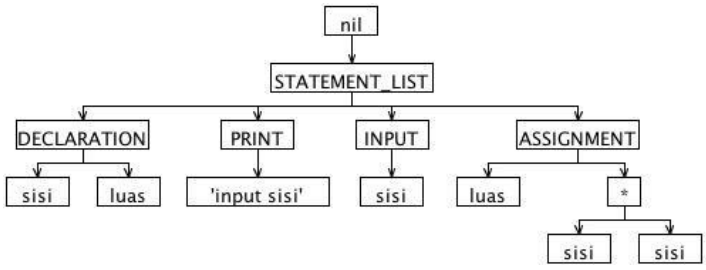
Karena semua tipe data hanyalah integer, kita bisa menggunakan hashtable seperti ini:

```
Hashtable<String, Integer> variables;
```

Jika ingin mendukung aneka tipe data, bisa menggantikan Integer dengan kelas Anda sendiri (misalnya kelas Variable), dan Anda perlu menyimpan tipe variabel serta nilainya.

Eksekusi

Kita review lagi perubahan grammar: baris-baris program tidak lagi berupa ekspresi saja. Di bagian awal harus ada deklarasi (lihat bagian declaration). Tepatnya, sebuah program adalah deklarasi yang diikuti oleh banyak statement. Sebuah statement bisa berupa: assignment, yaitu memberi nilai pada suatu variabel, input, yaitu meminta input dari user, dan print yaitu mencetak sebuah string, atau sebuah variabel. Untuk lebih jelasnya, Anda bisa melihat Syntax Diagram menggunakan ANTLRWorks. Supaya tidak perlu scroll ke atas, saya tampilkan lagi gambar sebelumnya:



Gambar 3.17: Diagram pohon sintak

Catatan

Kode-kode berikut ini ada pada file BulatLang.java

Mengevaluasi daftar statement sangat mudah, Anda cukup meloop semua statement, dan mengevaluasi masing-masing statement:

```
void evaluateStatementList(CommonTree exprlist) throws Exception{
        for (Object e: exprlist.getChildren()) {
            evaluateStatement((CommonTree)e);
```

```
        }
    }
```

Tapi ada banyak statement, jadi kita perlu mendelegasikan ke method yang sesuai. Kalau program sudah semakin rumit, delegasi bisa dilakukan ke kelas lain.

```
void evaluateStatement(CommonTree expr) throws Exception{
        switch (expr.getType()) {
        case BulatLexer.DECLARATION:
        evaluateDeclaration(expr);break;
        case BulatLexer.INPUT:
            evaluateInput((CommonTree)expr.getChild(0));
            break;
        case BulatLexer.PRINT:
            evaluatePrint((CommonTree)expr.getChild(0));
            break;
        case BulatLexer.ASSIGNMENT:
            evaluateAssignment(expr);
            break;
        }
    }
```

Pertama kita bahas dulu evaluasi deklarasi. Supaya terdeklarasi, setiap variabel dimasukkan ke hash table dan diberi nilai nol.

```
void evaluateDeclaration(CommonTree decl) {
        for (Object e: decl.getChildren()) {
            CommonTree et = (CommonTree)e;
            variables.put(et.getText(), 0);
        }
    }
```

Berikutnya, evaluasi input. Pertama kita cek dulu apakah variabelnya sudah dideklarasikan atau belum (jika belum, exception akan dilemparkan, dan program akan berhenti):

```
void evaluateInput(CommonTree expr) throws Exception {
        String var = expr.getText();
        checkVar(var);
        int i = scanner.nextInt();
        variables.put(var, i);
    }
```

Kita akan memakai fitur Java 5, yaitu kelas Scanner untuk membaca input dari keyboard. Memakai kelas ini cukup mudah dibanding pendekatan Java versi sebelumnya yang memerlukan BufferedReader. Contoh memakai kelas scanner seperti ini:

```
Scanner scanner;
scanner = new Scanner(System.in);

int i = scanner.nextInt();
```

Supaya tidak penasaran, pemeriksaan apakah variabel sudah dideklarasikan dilakukan dengan memanggil method containsKey milik Hashtable.

```
void checkVar(String var) throws Exception {
    if (!variables.containsKey(var)) {
        throw new Exception("Variable '"+var+"' not declared");
    }
}
```

Setelah itu kita bahas bagaimana mengevaluasi print. Ada dua bentuk, yang mencetak variabel dan yang mencetak string. Sebenarnya ini bisa dibuat lebih generik, tapi versi ini demi kesederhanaan tutorial. Jika tipenya adalah string, makakita perlu menghapus tanda petik di awal dan di akhir string. Mudahnya, gunakan regex di Java. Setelah itu cetak stringnya dengan System.out.println. Jika yang dicetak akan variabel, kita cek dulu apakah variabelnya ada atau tidak. Jika ada, kita ambil nilainya dengan get, dan kita cetak ke layar.

```
void evaluatePrint(CommonTree expr) throws Exception {
    if (expr.getType()==BulatLexer.STRING) {
        String s = expr.getText();
        /*remove first ' and last ' from string*/
        s = s.replaceAll("^'", "");
        s = s.replaceAll("'$", "");
        System.out.println(s);
    } else {
        String var = expr.getText();
        checkVar(var);
        System.out.println(variables.get(var));
    }
}
```

Dan yang terakhir adalah assignment. Anak pertama adalah variabel dan anak kedua adalah ekspresi. Kita sudah punya kode untuk evaluasi ekspresi dari tutorial sebelumnya, jadi yang diperlukan adalah: cek apakah variabel sudah dideklarasikan, evaluasi ekspresi, dan masukkan hasilnya dengan put.

```
void evaluateAssignment(CommonTree astmt) throws Exception {
    CommonTree var = (CommonTree)astmt.getChild(0);
    checkVar(var.getText());
    CommonTree expr = (CommonTree)astmt.getChild(1);
    int res = evaluateExpr(expr);
    variables.put(var.getText(), res);
}
```

Karena sekarang kita mendukung variabel dan unary operator, maka evaluasi ekspresi sedikit berubah. Jika kita bertemu dengan variabel, kita cek apakah variabel tersebut ada, lalu kita kembalikan nilainya:

```
if (expr.getType()==BulatLexer.ID) {
        checkVar(expr.getText());
        return variables.get(expr.getText());
}
```

Untuk unary plus dan minus juga tidak sulit. Kita tambahkan satu buah `if` untuk memeriksa jumlah anak. Jika jumlahnya satu, maka itu unary, selain itu pasti punya dua anak (binary).

```
    if (expr.getText().equals("+")) {
            if (expr.getChildCount()==1) {
                return evaluateExpr((CommonTree)expr.getChild(0));
            } else {return
evaluateExpr((CommonTree)expr.getChild(0))
+evaluateExpr((CommonTree)expr.getChild(
                1));
            }
    }
    if (expr.getText().equals("-")) {
            if (expr.getChildCount()==1) {
                return -evaluateExpr((CommonTree)expr.getChild(0));
            } else {
                return evaluateExpr((CommonTree)expr.getChild(0)) -
                 evaluateExpr((CommonTree)expr.getChild(1));
            }
    }
```

Anda bisa melihat sendiri source code lengkapnya di file zip yang tersedia (kira-kira 133 baris kode). Jika Anda mengikuti tutorial ini dari awal, tentunya tidak sulit mengikuti perubahan source codenya.

3.1.5 Compiler Untuk Bulat

Sebelumnya kita sudah memiliki grammar dan interpreter untuk Bahasa Bulat. Berikutnya kita akan membuat compilernya. Tidak semua detail dibahas, karena sudah dibahas di bagian pembuatan compiler ketika membuat compiler untuk ekspresi sederhana.

Kita sudah merencanakan untuk memakai LLVM di langkah-langkah berikutnya, tapi compiler untu bulat ini masih terlalu sederhana, jadi kita masih bisa memakai assembly langsung. Di sini Anda akan mulai melihat bahwa memakai assembly langsung akan semakin rumit. Compiler bulat ini sudah 198 baris. Sebagai pengingat, Assembly yang digunakan hanya ditargetkan untuk Linux.

Catatan

Kode-kode dalam bagian ini ada pada file BulatComp.java

Penyimpanan variabel

Karena bahasa ini masih sederhana, semua masih bisa disimpan sebagai global. Jika kita mengenal scoping, maka kita perlu memakai stack (sebenarnya ini juga tidak terlalu sulit, tapi tidak diperlukan sekarang). Sebuah variabel global integer bernama myvar bisa dideklarasikan di assembly seperti ini:

```
.global myvar
 .size myvar, 4
 myvar:
 .zero 4
```

Nah di Java, ini dihasilkan di bagian compileDeclaration dengan kode berikut:

```
        for (Object e: decl.getChildren()) {
            CommonTree et = (CommonTree)e;
            String var = et.getText();if
            (!variables.containsKey(var))
            { result.append(".globl "+
            var+"\n"); result.append(".size
            "+var+", 4\n");
            result.append(var+":\n");
            result.append(".zero 4\n");
            }
            variables.put(var, 0);
        }
```

Perhatikan bahwa di bahasa Bulat, kita mengijinkan var a,b,a, tapi a hanya akan didefinisikan sekali saja. Untuk memastikan itu, kita hanya akan menghasilkan deklarasi variabel, jika variabel belum ada di daftar variabel.

Fungsi main

Semua deklarasi dilakukan sebelum fungsi utama, setelah itu kita perlu memberikan template fungsi utama. Kita menambahkan beberapa string, "%s\n" digunakan untuk memformat string dalam instruksi print, string "%d\n" untuk mencetak integer, dan "%d" (tanpa \n) untuk membaca input dengan fungsi library C scanf.

```
.section    .rodata
.print_str_format:
.string     "%s\n"
.print_int_format:
.string     "%d\n"
.scan_int_format:
.string     "%d"
.text
.globl  main
.type main, @function
main:
```

Dengan kata lain di Java, kita melakukan ini:

```
        result.append(".section .rodata\n");
        result.append(".print_str_format:\n");
        result.append(".string \"%s\\n\"\n");
        result.append(".print_int_format:\n");
        result.append(".string \"%d\\n\"\n");
        result.append(".scan_int_format:\n");
```

```
        result.append(".string \"%d\"\n");
        result.append(".text\n");
        result.append(".globl main\n");
        result.append(".type main, @function\n");
        result.append("main:\n");
```

Perhatikan betapa membosankannya baris-baris tersebut. Anda bisa memakai library stringtemplate untuk memudahkannya, tapi kita masih bisa bertahan untuk saat ini.

Kode penting berikutnya adalah membaca input. Di C ada fungsi scanf, yang bisa digunakan untuk membaca aneka data (karakter, string, integer, double). Untukmembaca sebuah integer kita bisa memanggil scanf dengan parameter "%d" dan alamat variabel, misalnya scanf("%d", &sisi). Pemanggilan scanf ini mirip dengan printf (sudah dibahas di bagian membuat compiler).

Instruksi pushl $namavar akan mempush alamat variabel ke stack, dan kemudian kita mempush alamat string "%d" (yang sudah didefinisikan di atas dengan nama .scan_int_format), lalu memanggil scanf, dan membersihkan stack dengan instruksi popl.

```
void compileInput( CommonTree expr) throws Exception {
    String var = expr.getText();
    checkVar(var);
    result.append("pushl $"+var+"\n");
    result.append("push $.scan_int_format\n");
    result.append("call scanf\n");
    result.append("popl %ebx\n");
    result.append("popl %ebx\n");
}
```

Mencetak String

Mencetak sebuah integer sangat mudah, tapi mencetak string agak sulit. Kita perlu meletakkan sebuah string di alamat memori, lalu memberikan alamat memori itu ke fungsi printf. Kita bisa meletakkan string-string tersebut di mana saja, tapi untuk mudahnya, semua string diletakkan di bagian akhir assembly setelah kode program terakhir. Kita perlu mendeklarasikan sebuah string untuk penyimpanan sementara, nanti setelah semua kode program dihasilkan, kita tempelkan string ini di akhir kode program.

```
StringBuffer strings = new StringBuffer();
```

Di assembly semua string harus diberi nama (tepatnya diberi label alamat), jadi kita perlu memberikan nama untuk merujuk pada string tersebut. Supaya tidak bentrok, kita gunakan saja nama dengan format ._str_XX dengan xx adalah nomor string. Setiap string kita beri nomor menaik. Kita gunakan counter global agar semua nomor sifatnya unik:

```
int strcounter = 0;
```

Sekarang perhatikan kode berikut untuk menghasilkan instruksi print. Bagian ini merupakan bagian yang agak rumit dibanding bagian yang lain.

```
void compilePrint( CommonTree expr) throws Exception {
        if (expr.getType()==BulatLexer.STRING) {
            String s = expr.getText();
            /*remove first ' and last ' from string*/
            s = s.replaceAll("^'", "");
            s = s.replaceAll("'$", "");
            strcounter++;
            strings.append("._str_"+strcounter+":\n");
            strings.append(".string
            \""+s+"\"\n");result.append("push
            $._str_"+strcounter+"\n");
            result.append("push
            $.print_str_format\n");
            result.append("call printf\n");
            result.append("popl %ebx\n");
            result.append("popl %ebx\n");
        } else {
            String var = expr.getText();
            checkVar(var);
            result.append("push "+var+"\n");
            result.append("push $.print_int_format\n");
            result.append("call printf\n");
            result.append("popl %ebx\n");
            result.append("popl %ebx\n");
        }
 }
```

Bagian pertama adalah penanganan untuk print 'string', di bagian kedua dalah untuk print var. Untuk setiap string, kita hasilkan ID baru, lalu simpan di variabel strings:

```
strcounter++;
strings.append("._str_"+strcounter+":\n");
strings.append(".string \""+s+"\"\n");
```

Kita panggil printf seperti pada tutorial sebelumnya, tapi parameter kedua (yang dipush pertama, karena urutannya terbalik) adalah string yang baru saja kita buat tadi:

```
result.append("push $._str_"+strcounter+"\n");
result.append("push $.print_str_format\n");
result.append("call printf\n");
result.append("popl %ebx\n");
result.append("popl %ebx\n");
```

Untuk bagian print dengan variabel, kita menggunakan push var, tanpa $ karena kita ingin nilainya yang dicetak, bukan alamatnya:

```
result.append("push "+var+"\n");
```

```
  result.append("push $.print_int_format\n");
  result.append("call printf\n");
  result.append("popl %ebx\n");
  result.append("popl %ebx\n");
```

Variabel dan Unary minus

Dibandingkan dengan pembuatan compiler sebelumnya, dalam compileExpr, kita perlu menambahkan kasus untuk menangani identifier/variable. Ini relatif mudah:

```
  if (expr.getType()==BulatLexer.ID) {
       checkVar(expr.getText());
       result.append("movl "+expr.getText()+", %eax\n");
       result.append("pushl %eax\n");return;
  }
```

Dan kasus untuk Unary:

```
if (expr.getChildCount()==1) {
   if (expr.getText().equals("+")) {
        /*nothing changed*/
          compileExpr((CommonTree)expr.getChild(0));
          return;
    }
    if (expr.getText().equals("-")) {
        compileExpr((CommonTree)expr.getChild(0));
        result.append("popl %eax\n");
        result.append("negl %eax\n");
        result.append("pushl %eax\n");
        return;
     }
}
```

Untuk unary '+' tidak ada instruksi khusus yang dihasilkan. Untuk unary minus, instruksi assembly negl digunakan, instruksi ini gunanya untuk menegatifkan suatu register.

3.2 Pemrograman Karya Indonesia

Implementasi interpreter dan compiler bukanlah hal yang sulit, ini terbukti dari sudah adanya implementasi interpreter, compiler, dan translator program di berbagai tugas akhir mahasiswa sejak tahun 1981 (bisa di cek di perpustakaan Informatika ITB, dan saya yakin Ilmu Komputer UI/UGM/IPB juga memiliki banyak tugas akhir semacam ini). Buku mengenai pembuatan interpreter berbahasa Indonesia juga sudah ditulis, sejak 1984 (misalnya KILANG 002: BASIC dalam Bahasa Indonesia, Jakarta: Kesaint Blanc, 1984), dan satu lagi yang diterbitkan di tahun 1995 (Tuntunan Praktis Pemrograman Merekayasa Interpreter: Sebuah Penerapan

Teknik Kompilasi, Sukamdi, 1995 ISBN :979-637-744-6), dalam buku yang ditulis Sukamdi ditunjukkan mengenai pembuatan bahasa ALIN untuk memproses ekspresi aljabar linier.

Bahasa KILANG dibuat oleh Prof.Dr.Ir.Dali Santun Naga KILANG merupakan singkatan dari Kaidah Informasi Lambang Aneka Nalar dan Guna, di tahun 1984 bahasa ini sudah mencapai versi 2. Bahasa KILANG ini pernah dibahas di Seminar Komputer dan Diskusi Ilmiah Dies Natalis III Himpunan Mahasiswa Informatika ITB, Bandung, 19 Maret 1986 "Dua Tahun KILANG 002: Kisah Singkat tentang Pendomestikasian Bahasa Komputer". Sayangnya saat ini belum terdengar lagi kelanjutannya.

Dua bahasa pemrograman yang saat ini masih kembangkan oleh putra Indonesia secara open source adalah bahasa Qu dan bahasa BAIK. Bahasa Qu sudah bisa dipakai sejak 2002, definisi bisa dipakai adalah interpreternya sudah berjalan. Di tahun 2007 Qu sudah memiliki banyak fitur dan didokumentasikan dengan baik. Bahasa Qu ditujukan sebagai bahasa umum (general purpose language). BahasaQu yang bersifat open source ini sudah dipakai untuk membuat aplikasi web dan aplikasi komersial.

Sementara bahasa BAIK yang baru dikembangkan sejak tahun 2008, sudah memiliki interpreter di tahun yang sama, dan di tahun 2009-2010 sudah ada 3 naskah ilmiah (paper) nasional dan internasional yang membahas bahasa BAIK. Bahasa BAIK ditujukan untuk pembelajaran pemrograman. Source code baik juga terbuka untuk umum. Sebagai tambahan informasi, bahasa baik berusaha menggunakan bahasa Indonesia, mirip seperti yang dilakukan bahasa KILANG.

Bahasa Qu

Lisensi: GPL
Pengembang: Marc Krisnanto
Rilis awal: 2002
Rilis terbaru: 2009
Naskah Ilmiah: -
Platform: Unix based (kemungkinan bisa diperforming a corresponding memory-to-registeroperation on the same variable.jalankan di Windows dengan Cygwin)

Website: http://www.qu-lang.org

Implementasi awal bahasa ini hanya dikembangkan dalam beberapa minggu saja (referensi: http://www.qu-lang.org/about.htm), dan masih terus dikembangkan dengan rilis terakhir tahun 2009. Bahasa ini cukup sederhana (seperti python) jadi bisa dipelajari dengan mudah oleh pemula, tapi juga memiliki berbagaifitur "advanced" (coroutine, list comprehension, function and method currying, dsb), sehingga bisa digunakan juga oleh programer tingkat lanjut.

Salah satu kelebihan yang saya lihat dari bahasa ini dibanding sebagian besar bahasa lain adalah: bahasa ini memiliki strong semi-dynamic typing/optional static typing, artinya sebagian variabel bisa diberi tipe, sebagian lagi tidak. Ini merupakan kompromi dari kemudahan berbagai bahasa terinterpretasi seperti PHP yang sama sekali tidak mengenal tipe (sehingga tidak bisa mengoptimasi pengunaan memori),

dengan bahasa yang sangat strict (seperti Java).

Interpreter bahasa ini diimplementasikan dalam C dengan parser yang ditulis tangan (tidak dihasilkan oleh parser generator). Source code yang diberikan mudah dikompilasi, dan sudah menyertakan skrip "configure" (sehingga bisa menyesuaikan diri dengan berbagai platform UNIX yang ada). Implementasi interpreter bahasa ini sudah menyertakan banyak modul built-in (seperti Big Integer, List, Tree), dan tersedia juga package untuk parsing XML (dengan Expat), membuat aplikasi GUI (dengan GTK), dan bahkan melakukan koneksi dengan database (Mysql). Jika Anda masih butuh yang lebih, Anda bisa mengimplementasikan fungsi dalam C dan memanggilnya dari Qu, atau sebaliknya Anda bisa mengembed bahasa Qu ke program C.

Meskipun tidak/belum ada paper ilmiah mengenai bahasa ini, dokumentasi bahasa ini sangat lengkap, mulai dari syntax/grammar sampai dokumentasi fungsi dan kelas yang tersedia. Selain dalam bentuk referensi, dokumentasi berupa tutorial juga tersedia. Tapi tentu saja jangan dibandingkan dengan bahasa yang sudah matang seperti python yang punya tutorial sangat detail, tutorial yang diberikan pada dokumentasi Qu cukup singkat. Untuk orang yang baru belajar pemrograman mungkin tidak cocok, tapi sangat cocok untuk orang yang sudah mengenal berbagai bahasa lain seperti Python dan Perl.

Penulis bahasa ini tidak membuat bahasa untuk tujuan pengajaran, tapi karena dia frustasi dengan bahasa yang sudah ada. Jadi bahasa ini tidak memiliki tujuan yang spesifik. Tapi ini bukanlah sebuah kekurangan, bahasa-bahasa open source seperti PHP, Perl, Python dan Ruby juga tidak memiliki tujuan khusus di awal, hanya untuk keperluan penciptanya saja. Mungkin satu-satunya kritik saya adalah: saat ini tidak ada versi yang bisa diakses dengan versioning management (misalnya CVS atau git), padahal saya ingin melihat perkembangan bahasa ini. Sementara kode program juga tidak mencantumkan tanggal, dan versi lama tidak bisa didownload lagi.

Saya sudah mengkonfirmasi dengan penulis mengenai keluhan terakhir, dan masalah utama yang dihadapi adalah kesibukannya dan koneksi internet di Indonesia yang sering kurang stabil. Namun di masa depan kemungkinan akan disetup sebuah sistem (via sourceforge misalnya) agar lebih mudah bagi pihak yang ingin berkontribusi.

Secara singkat, sisi negatif dan positif bahasa Qu adalah sebagai berikut (perhatikan bahasa sisi negatif ini hanya untuk versi saat ini, masih bisa terus diperbaiki):

Positif:

- Memiliki fitur advanced
- Memiliki dokumentasi yang lengkap
- Memiliki banyak package
- Struktur program sangat baik (bisa digunakan oleh orang yang ingin belajar pembuatan interpreter)
- Sudah dipakai di dunia nyata (situs http://www.ina.travel/ dan program accounting http://www.gate17.net/)

Negatif:

- Kurang dipromosikan sehingga kurang dikenal
- Versi lama tidak bisa didownload
- Belum ada naskah ilmiah yang membahasnya (namun tujuan bahasa ini bukan untuk pengajaran jadi sepertinya memang tidak diperlukan, sementara itu dokumentasi sudah cukup lengkap)

Bahasa BAIK (Bahasa Anak Indonesia untuk Komputer)

Lisensi: source terbuka tapi mungkin
tidak memenuhi syarat open source menurut OSI
Pengembang: Haris Hasanudin
Rilis awal: 2008
Rilis terbaru: 2010
Naskah Ilmiah: 3 paper
Platform: UNIX based (Linux, Solaris), Windows

Website: http://sourceforge.net/projects/baik/

Bahasa ini dikembangkan oleh Haris Hasanudin sejak tahun 2008. Tujuan bahasa ini adalah untuk pengajaran pemrograman dengan menggunakan Bahasa Indonesia. Dokumentasi bahasa ini masih minimal, tapi naskah ilmiah tentang BAIK telah dipublikasikan di dua konferensi internasional dan jurnal nasional sebagai berikut:

- Basic Design of BAIK = Scripting Language with Indonesian Lexical Parser for Internet-based Software Development, Proc. of Int. conf. on Advance Computer Science and Information System - ICACSIS 2009.
- BAIK (Bahasa Anak Indonesia untuk Komputer) = Programming Language based on Indonesian Lexical Parsing for Multi-Tier Web Development, Journal on Computer Science and Information (JIKI) June 2010, Universitas Indonesia.
- BAIK Language for Visual Programming with Indonesian Natural Language, Int. MALINDO Workshop 2010.

Bahasa ini dapat digunakan untuk mengajarkan konsep prosedural dan object oriented. Bahasa ini tidak mengandung aneka fitur "advanced" seperti Qu, tapi menurut saya itulah kelebihan bahasa ini yaitu kesederhanaannya. Tidak seperti Qu yang tidak menyertakan dukungan GUI secara default, bahasa ini mendukung GUI sebagai bagian dari bahasanya. Anda juga bisa melakukan koneksi SQL ke Postgres dan Oracle, serta melakukan koneksi TCP/IP. Pemrograman web dengan CGI pun bisa dilakukan.

Dari segi pengguna BAIK, bahasa ini sepertinya cocok untuk pemula (sesuai tujuannya), tapi kurang cocok untuk pengembangan aplikasi yang besar. Tidak ada pemisahan modul, dan semua statement pakai_xx (pakai_layar, dsb) merupakan sesuatu yang dihardcode (bandingkan dengan Qu). Mungkin pemisahan modul bisa menjadi fitur bahasa ini di masa depan, karena itu akan mengajari pemakai bahasa

BAIK mengenai code reuse.

Bahasa ini diimplementasikan dalam C, dan masih banyak hal yang bisa diperbaiki dari implementasi interpreternya. Misalnya semua nama fungsi di library seperti "gambarpoligon" di-hardcode di bagian Lexer. Masih ada memory error dan memory leak ketika diperiksa dengan valgrind, dsb. Saya menghubungi langsung pembuat bahasa ini untuk perbaikan beberapa hal tersebut, dan respon pembuat bahasa ini menurut saya sangat baik. Saat ini implementasi bahasa BAIK masih kurang cocok dipelajari untuk Anda yang ingin belajar pembuatan interpreter (designnya monolitik). Mungkin di masa depan implementasinya juga bisa ditingkatkan agar lebih mudah dipelajari.

Secara singkat, sisi negatif dan positif bahasa BAIK adalah sebagai berikut (perhatikan bahwa sisi negatif ini hanya untuk versi saat ini, masih bisa terus diperbaiki):

Positif:

- Cocok untuk pemula
- Banyak contoh program disertakan
- Sudah ada beberapa naskah ilmiah ditulis mengenai bahasa ini
- Pemrograman grafik, database, dan web mudah dilakukan
- Buku panduan bahasa baik tersedia secara gratis

Negatif:

- Bahasa BAIK belum mendukung pemrograman modular
- Implementasi dalam C kurang modular

BAB 4
TYPE CHECKING

4.1 Static VS Dynamic Cheking

Type checking adalah salah satu aspek semantik yang sangat penting dalam kompilasi. Type Checker memeriksa apakah tipe dari suatu konstruksi cocok dengan konteksnya. Misalnya, built-in operator arithmetic mod dalam Pascal memer-lukan operand integer, sehingga type checker harus memeriksa bahwa ope-rand dari mod mempunyai type integer. Intinya, type checking :

1. Memungkinkan programmer untuk membatasi penulisan yang bisa digunakan dalam circumtances tertentu
2. Menugaskan type untuk menilai (values)
3. Menentukan apakah value digunakan dengan cara yang sesuai

Terlepas dari membuktikan apakah kode yang dipakai itu sudah benar, seperti memeriksa bahwa function calls telah benar jumlah dan tipe parameternya, type checking juga membantu dalam memutuskan kode yang mana yang akan digeneralkan seperti dalam kasus arithmetic expressions. Pada bab ini, kita akan membicarakan tentang bagaimana type checking dapat dimasukkan ke dalam bahasa pemrograman untuk membuktikan kegunaannya dalam program. Ada banyak variasi type checking. Dalam situasi yang sederhana, type checking digunakan untuk memeriksa tipe objek dan melaporkan type-error. Di sisi lain, penulisan yang tidak benar bisa dibetulkan (dengan coercing, yang akn dibahas nanti). Definisi dari type sangat sulit untuk dijelaskan, seperti yang akan kita lihat pada bab ini, memeriksa tipe ekuivalen dari dua objek merupakan masalah penting yang harus dipecahkan.

Dalam berbagai bahasa modern, type checking dilakukan pada saat meng-compile. Hal ini juga dikenal sebagai static checking, karena pada tipe ini, properties akan diperiksa sebelum running program. Sedangkan tipe dynamic checking atau runtime checking dilakukan selama eksekusi program.

Manfaat dari compile static checking di antaranya :

1. Dapat menunjukkan banyak kesalahan yang mungkin terjadiStatic checking lebih dibutuhkan ketika speed (kecepatan running program) memegang peranan penting, karena dapat menghasilkan kode yang lebih cepat yang tidak bisa dihasilkan oleh type checking yang dilakukan ketika eksekusiprogram

contoh static cheking adalah :

- Type checks
- Flow-of-control checks
- Uniqueness checks
- Name-related checks

Sedangkan dynamic checking memiliki kelebihan sebagai berikut :

1. Biasanya memungkinkan programmer untuk tidak terlalu memperhatikan penulisan. Dengan demikian, hal ini memberi kelonggaran pada programmer
2. Biasanya diperlukan dalam beberapa kasus seperti array bounds check, yang hanya bisa ilakukan selama eksekusi program
3. Dapat menghasilkan kode yang lebih jelas

Beberapa kompiler PASCAL mengga-bungkan static-checking dan interme-diate code generation dengan parsing. Untuk bahasa yang konstruksinya kompleks, seperti Ada, type-checking biasanya dilakukan terpisah, antara parsing dan intermediate code generation. Bahasa ADA secara eksplisit meng-ijinkan programmer untuk overload operator dimana sangat banyak bahasa lainnya yang overload operator aritmatika dengan type numerik yang berbeda (integer dan real). Constraint overload operator dicek dalam one-pass parser tidak meng-ijinkan terlalu banyak jenis untuk forward referensi simbol. Pada Bahasa C dan Fortran dengan kemudahan yang dimilikinya, bahwa type checking tidak mungkin dalam one-pass parser.

4.1.1 Tipe Ekspresi

Type expressions digunakan untuk merepresentasikan tipe konstruksi bahasa. Type expression juga dapat menjadi basic type, type name, atau type constructor yang diterpkan dalam daftar type expressions. Berikut adalah definisi dari tipe ekspresi:

1. Basic type : boolean, char, integer, real dan Basic type khusus type_error (beri signal selama type checking)
2. Type Name
3. Type Constructor, yaitu : Array, Product, Record, Pointer, Function.
4. Type Variable

4.1.2 Type Conversion

Secara eksplisit bahwa representasi, instruksi mesin dan operasi dari integer dan real berbeda oleh karena itu compiler harus mengkonversi salah satu dari type operand tersebut untuk menjamin bahwa kedua operand bertipe sama.

Type checking dalam kompiler dapat dipakai untuk menyisipkan operasi konversi kedalam intermediate repre-sentation dari source program. Contoh :

Suatu ekspresi adalah x + **i**,
dimana : x bertipe real dan
i bertipe integer.
Postfix notation untuk x + **i** adalah :
x **i** inttoreal real+
dimana operator inttoreal meng-konversikan
i dari integer ke real
kemudian real+ melakukan operasi penjumlahan.

OVERLOADING FUNCTION DAN OPERATOR

Overloaded Symbol adalah simbol yang mempunyai arti lain dan tergantung daripada konteksnya. Dalam matematika, misal : A+B yang mana operator penjumlahan (+) adalah overloaded, sebab berbeda arti dengan A dan B. Pada bahasa ADA, tanda kurung () adalah overloaded.

Set Kemungkinan Tipe

Dalam ADA, standar interpretasi operator * adalah function dari sepasang integer. Kadang-kadang sub-expression mempunyai beberapa kemungkinan tipe. Dalam ADA, konteks harus memberi informasi yang cukup untuk mempersempit pilihan menjadi tipe tunggal. Operator * dapat di-overload dengan menambah deklarasi sebagai berikut :

```
function "*" (i,j:integer) return complex;
function "*" (x,y:complex) return complex;
```

Setelah deklarasi diatas kemungkinan tipe untuk * adalah :

---

integer x integer x integer
integer x integer x complex
complex x complex x complex

Misalkan bilangan 2, 3, 5 tipenya mungkin hanya integer, subekspresi dari 3*5 dapat berupa integer atau kompleks, tergantung kepada konteksnya. Jadi untuk ekspresi 2*(3*5), maka 3*5 harus bertipe integer karena * meng-ambil sepasang integer atau sepasang bilangan kompleks sebagai argumen. Sebaliknya, 3*5 harus bertipe

kompleks jika ekspresi lengkapnya adalah (3*5)*Z dan Z dideklarasikan sebagai kompleks.

4.1.3 Fungsi Polymorphic

Suatu prosedur biasanya membolehkan statement dalam body-nya dieksekusi dengan argumen bertipe tetap. Suatu polymorphic procedure dapat dipanggil (statement dalam body-nya dapat dieksekusi) dengan argumen yang berbeda tipenya. Istilah "polymorphic" juga dapat dite-rapkan untuk sepenggal code yang dapat dieksekusi dengan argument yang berbeda tipenya.

Contoh : C reference manual menyatakan ten-tang pointer operator & : "Jika tipe operand adalah x, maka hasilnya akan bertipe pointer to x". Karena beberapa tipe data bisa menggantikan x, maka operator & dalam C bersifat polymorphic. Didalam bahasa Ada, "generic" function bersifat polymorphic. Namun polimorfis dalam ADA terbatas. Karena istilah "generic" juga digunakan untuk overloaded function dan coercion dari argument suatu function.

Polymorphic Function sangat menarik karena memberi fasilitas untuk meng-implementasikan algoritma yang me-manipulasi struktur data, tanpa mem-perhatikan tipe elemen dalam struktur data tersebut. Contoh dalam Pascal, program untuk menentukan panjang dari list berelemen integer adalah sebagai berikut :

```
type link = ^cell;
   cell = record
     info : integer;
     next : link;
   end;

function length(lptr : link) : integer;
   var len : integer;
   begin
     len := 0;
     while lptr <> nil do
     begin
       len := len + 1;
       lptr := lptr.next;
     end;
       length := len;
    end;
```

Dalam bahasa ML :

```
fun length(lptr) =
    if null(lptr) then 0
    else length(tl(lptr)) + 1;
maka :
    length(["sun","mon","tue"]);
    length([10,9,8]);
adalah 3
```

BAB 5
INTERMEDIATE-CODE GENERATION

\

Tujuan akhir dari compiler adalah untuk memperoleh program yang dituliskan dalam bahasa tingkat tinggi untuk berjalan pada sebuah komputer. Ini artinya bahwa, akhirnya, prgoram harus telah diekspresikan sebagai code mesin yang dapat berjalan pada komputer. Bukan berarti bahwa kita perlu menterjemahkan secara langsung dari abstrak sintaks tingkat tinggi ke code mesin. Banyak compiler menggunakan sebuah bahasa tingkat medium sebagai sebuah batu loncatan antara bahasa tingkat tinggi dan kode mesin tingkat terendah. Yang dimaksud dengan bahasa batu loncatan disini adalah *intermediate code*. Selain daripada mengorganisir compiler kedalam pekerjaan yang kecil, menggunakan bahasa intermediate memiliki beberapa kelebihan sebagai berikut:

- Jika compiler perlu untuk menciptakan code untuk beberapa arsitektur mesin yang berbeda, intermediate code hanya memerlukan satu penerjemah. Hanya penerjemah dari intermediate code ke bahasa mesin (yaitu *back end*) perlu dituliskan kebeberapa versi.
- Jika beberapa bahasa tingkat tinggi perlu di compile, hanya satu penerjemah yang diperlukan intermediate code yang dituliskan untuk tiap-tiap bahasa. Mereka semua dapat berbagi back-end, yaitu penerjemah dari intermediate code ke code mesin.
- Sebagai penggati penerjemah intermediate code ke kode mesin, itu dapat diinterpretasikan dengan program kecil dituliskan dalam mesin code atau dalam sebuah bahasa yang sudah ada compiler atau interpreternya.

Kelebihan yang paling terasa saat menggunakan bahasa intermediate saat banyak bahasa dicompile kebanyak mesin. Jika penerjemah dilakukan secara langsung, jumlah dari compiler-compiler sama dengan produk dari jumlah pada bahasa dan jumlah mesin. Jika intermediate code digunakan secara umum, satu front-end (yaitu, compiler ke intermediate code) dibutuhkan untuk tiap-tiap bahasa dan satu back-end (interpreter atau code generator) diperlukan untuk masing-masing mesin, yang membuat jumlah total pada front end dan back-end sama dengan total dari jumlah bahasa dan jumlah dari mesin.

Jika sebuah interpreter untuk bahasa intermediate dituliskan dalam bahasa yang mana telah diimplementasikan untuk target mesin, interpreter yang sama dapat juga diinterpretasikan atau dicompile untuk tiap-tiap mesin. Cara ini, tidak perlu menulis back-end secara terpisah untuk tiap-tiap mesin. Kelebiahan menggunakan pendekatan ini adalah:

- Secara aktual tidak diperlukan back-end untuk dituliskan pada tiap-tiap masin baru, sejauh mesin diperlengkapi dengan sebuah interpreter atau compiler untuk implementasi bahasa terhadap interpreter untuk bahasa intermediate.

- Program yang dicompile dapat didistribusikan dalam bentuk intermediate tunggal untuk semua mesin. berlawanan dengan pengiriman binary secara terpisah untuk tiap-tiap mesin.

- Bentuk intermediate mungkin lebih compact dari pada kode mesin. Ini akan menghemat ruang memory dalam pendistribusian dan mesin yang menjalankan program.

Kelemahannya adalah pada kecepatan: Menginterpretasikan bentuk intermediate dalam banyak kasus akan sedikit lebih lambat dari pada menjalankan penerjemah kode secara langsung. Namun demikian, pendeketan seperti ini telihat berhasil, contohnya yang telah diterapkan pada bahasa pemrograman Java. Penalti pada kecepatan dapat dieliminasi dengan segera menerjemahkan intermediate code ke mesin code sebelum atau selama program dieksekusi. Bentuk hybird seperti ini disebut dengan *just-in-time compilation* dan sering digunakan untuk implementasi modern pada Java untuk menjalankan code intermediate (Java Virtual Machine).

5.1 Varian pada Pohon Sintak

Node dalam sebuah pohon sintak menyatakan susunan-sunanan dalam source program; node anak (*children*) menyatakan komponen penting pada susunan-susunan tersebut. *Directed acyclic graph* atau disingkat dengan DAG untuk mengindentifikasi ekspresi subekspresi (*subexpressions*) yang muncul lebih dari satu kali pada ekspresi. DAG dapat disusun dengan menggunakan teknik yang sama seperti membuat pohon sintak.

5.1.1 DAG untuk Ekspresi

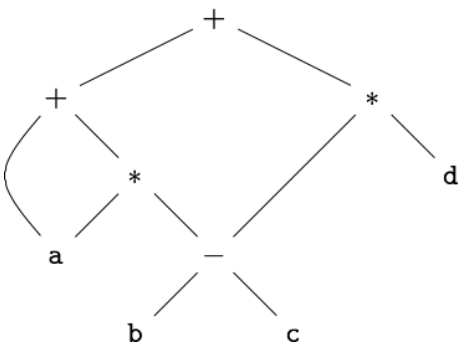
DAG memiliki daun-daun sesuai terhadap atomic operand dan node interior yang sesuai terhadap operator. Perbedaannya adalah sebuah node *N* dalam DAG mempunyai lebih dari satu node orang tua (*parent*) jika *N* menyatakan subexpression yang bersamaan; dalam pohoh sintak, pohon untuk subexpression yang bersamaan subexpression akan di replika sebanyak subexpression yang muncul dalam ekspressi aslinya. Dalam artian, DAG tidak hanya menyatakan ekspresi secara ringkas tetapi juga memberikan clue penting terhadap compiler tentang efesiensi code generasi untuk evaluasi sebuah ekspresi.

Contoh : Gambar 5.1 menunjukkan DAG untuk ekspresi

```
a + a * (b - c) + (b - c) * d
```

a memiliki dua daun parent, karena a dalam ekspresi muncul berpasangan. Lebih

menariknya lagi, dua subexpression b-c terjadi besamaan dinyatakan dengan satu node, node dilabelkan dengan . Node tersebut memiliki dua parent, yang menyatakan duan tersebut digunakan dalam subexpression a*(b-c) dan (b-c)*d. Meskipun b dan c muncul berpasangan dalam ekspresi lengkap, node-node mereka masing-masing memiliki satu parent, karena keduanya digunakan bersamaan dalam subexpression b-c.



Gambar 5.1: Dag untuk ekspresi a + a * (b - c) + (b - c) * d

Contoh : Susunan tahapan-tahapan untuk membuat DAG pada Gambar 5.1 adalah sebagai berikut:

$$
\begin{aligned}
1)\quad & p_1 = \mathit{Leaf}(\mathbf{id}, \mathit{entry\text{-}a}) \\
2)\quad & p_2 = \mathit{Leaf}(\mathbf{id}, \mathit{entry\text{-}a}) = p_1 \\
3)\quad & p_3 = \mathit{Leaf}(\mathbf{id}, \mathit{entry\text{-}b}) \\
4)\quad & p_4 = \mathit{Leaf}(\mathbf{id}, \mathit{entry\text{-}c}) \\
5)\quad & p_5 = \mathit{Node}('-', p_3, p_4) \\
6)\quad & p_6 = \mathit{Node}('*', p_1, p_5) \\
7)\quad & p_7 = \mathit{Node}('+', p_1, p_6) \\
8)\quad & p_8 = \mathit{Leaf}(\mathbf{id}, \mathit{entry\text{-}b}) = p_3 \\
9)\quad & p_9 = \mathit{Leaf}(\mathbf{id}, \mathit{entry\text{-}c}) = p_4 \\
10)\quad & p_{10} = \mathit{Node}('-', p_3, p_4) = p_5 \\
11)\quad & p_{11} = \mathit{Leaf}(\mathbf{id}, \mathit{entry\text{-}d}) \\
12)\quad & p_{12} = \mathit{Node}('*', p_5, p_{11}) \\
13)\quad & p_{13} = \mathit{Node}('+', p_7, p_{12})
\end{aligned}
$$

Kita asumsikan bahwa point *entry a* ke entri tabel simbol untuk a, dan dengan cara serupa untuk identifier lainnya.

Saat *Leaf* (**id**, *entry a*) dipanggil akan terjadi perulangan pada step 2, node dibuat oleh pemanggilan yang telah dikambalikan sebelumnya, jadi *p*2=*p*1. Dengan cara yang serupa, node-node dikembalikan pada step 8 dan 9 sama seperti yang dikembalikan pada step 3 dan 4 (yaitu *p*8 = *p*3 dan *p*9=*p*4). Demikian juga node yang dikembalikan pada step 10 harus sama pada step 5 yaitu; *p*10 = *p*5

5.2 Three-Address Code

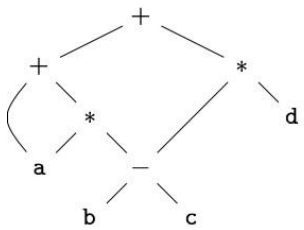
Dalam three-address code, terdapat satu operator pada sisi kanan dari instruksi;

artinya tidak diperbolehkan menambah ekspresi aritmatika. Dengan demikianekspresi source-language seperti x+y*z akan diterjemahkan kedalam susunaninstruksi three-address code berikut

```
t 1 = y * z
t 2 = x + t 1
```

dimana t1 dan t2 dihasilkan oleh compiler sebagai nama variabel sementara. Memisahkan ekspresi aritmatika multi-operator dan perulangan flow-of-control statament diperlukan sekali untuk target-code generation dan optimisasi.

Contoh : Three-address code adalah sebuah pernyataan secara linear terhadap sebuah pohon sintak atau DAG dimana nama variabel sementara dari three-address code secara eksplisit bersesuaian dengan interior node dari graph.



(a) DAG

$t_1 = b - c$
$t_2 = a * t_1$
$t_3 = a + t_2$
$t_4 = t_1 * d$
$t_5 = t_3 + t_4$

(b) Three-address code

Gambar 5.2: DAG dan three andress codenya

5.2.1 Quadruples

Quadruple atau "*quad*" memiliki empat field yaitu *op*, arg_1, arg_2, dan *result*. Field *op* berisikan sebuah code internal untuk operator. Misalnya instruksi three-address seperti $x = y + z$ dinyatakan dengan menempatkan + dalam *op*, y dalam arg_1, z dalam arg_2, dan x dalam *result*. Berikut ini beberapa aturan yang digunakan:

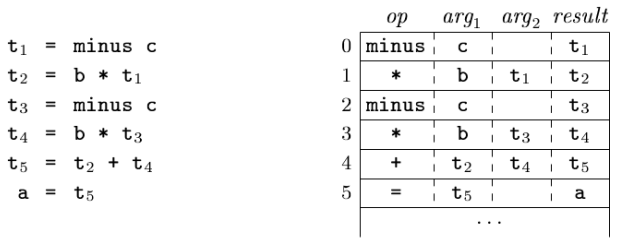
1. Intruksi dengan operator unary seperti $x =$ minus y atau $x = y$ tidak menggunakan arg_2.Perhatikan bahwa untuk stament seperti $x = y$, *op* adalah =, sedangkan untuk kebenyakan operasi lainnya, operator assigment secara tersirat.

2. Operator seperti param menggunakan baik itu arg_2 atau *result*.

3. Loncatan Conditional dan unconditional menempatkan targel label dalam *result*.

Contoh : Three-adress code untuk assigment = a = b * -c + b * - c ; muncul dalam Gambar 5.3(a). Operator khusus minus digunakan untuk membedakan operator minus unary, seperti -c, dari operator minus binary, seperti b - c. Perhatikan bahwa statement unary-minus "three-address" hanya memiliki dua pengalamatan, seperti halnya statement a = t_5.

Quadruples pada Gambar 5.3(b) implementasi dari three-address code bagian (a).



(a) Three-address code (b) Quadruples

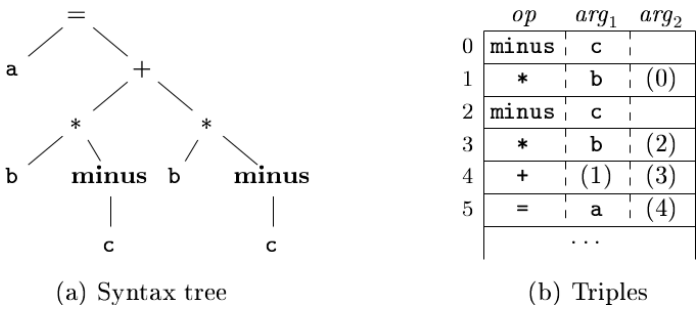
Gambar 5.3: Three-address code dan representasi quadruplenya.

Untuk kenyamanan pembaca, maka digunakan identifier aktual seperti a, b, dan c dalam field arg_1, arg_2, dan *result* pada Gambar 5.3(b). Variabel sementara dapat dimasukkan kedalam tabel simbol seperti halnya programmer mendefinisikan sebuah nama variabel, atau juga dapat diimplementasikan sebagai objek dari kelas *Temp* dengan method yang tedapat pada class tersebut.

5.2.2 Triples

Triple hanya memiliki tiga field yaitu *op*, arg_1 dan arg_2. Perhatikan bahwa field *result* pada Gambar 5.3(b) digunakan untuk nama variabel sementara. Menggunakan triple, kita merujuk pada hasil dari sebuah pengoperasian x *op* y berdasarkan posisinya, ketimbang dengan nama variabel sementara yang eksplisit. Jadi, sebagai pengganti variabel sementara t_1 dalam Gambar 5.3(b), representasi triple akan merujuk ke posisi (0). Angka dalam kurung menyatakan pointer kedalam struktur triple itu sendiri.

Contoh: Pohon sintak dan triple pada dalam Gambar 5.4 dapat disamakan dengan three-address code dan quadruple dalam Gambar 5.3. Dalam pernyataan triple pada Gambar 5.4(b), statement copy a = t_5 adalah encode dalam pernyataan triple dengan meletakkan a pada field arg_1 dan (4) pada field arg_2.



(a) Syntax tree (b) Triples

Gambar 5.4: Representasi dari a = b*-c+b*-c;

Operasi ternary seperti x[i] = y memerlukan dua entri dalam struktur triple; sebagai contoh, kita dapat menaruh x dan i dalam satu triple dan selanjutnya y. Demikian pula, x = y[i] dapat diimplementasikan dengan penanganan serupa dengan dua instruksi t = y[i] dan x=t, dimana t adalah nilai sementara yang dihasilkan oleh compiler. Perlu diperhatikan t secara aktual tidak muncul dalam triple, karena

bersifat nilai sementara yang dirujuk ke posisinya sendiri dalam struktur triple.

LATIHAN

1. Nyatakan ekspresi berikut kedalam three-address code serta buatlahDAG nya !.

x = a / b + (c + d) * e

Terjemahkan ekspresi aritmatik a + -(b + c) kedalam
Pohon sintak
Three-address code
Quadruple
Triple

BAB 6
REGISTER ALLOCATION

Register allocation adalah proses pengalokasian atau memeberikan (*assign*) variabel program ke register. Ide umum dibalik register allocation adalah instruksi mesin yang beroperasi pada variabel program dalam register biasanya lebih cepat dari pada pengoperasian didalam memory. Karena secara alami instruksi set arsitektur intel, satu operand harus berada dalam register. Operand yang kedua dapatberada dimemory atau register. Terdapat perbedaan waktu yang diperlukan untuk menjalankan sebuah instruksi dengan dua operand dalam register dibandingkan dengan haya satu operand yang ada diregister.

6.1 Partisi Register Allocation

Register allocation dapat dilakukan dengan dua cara yaitu secara lokal dan secara global. Lokal register allocation berkenaan dengan penugasan (assign) register kedalam set basic block atau kedalam sebuah prosedur program. Secara umum, untuk lokal register allocation, nilai-nilai didalam register harus dituliskan kedalam memory pada akhir tiap-tiap basic block. Hal ini diperlukan karena kontrol mungkin mencapai block dari banyak block lainnya, dan tidak dapat diasumsikan secara langsung bahwa variabel yang digunakan oleh block successor akan selalu muncul dalam sama register yang sama. Dalam beberapa kasus kinerjanya dapat seperti operasi register-to-memory dan kemudian dalam block succesive, melakukan operasi memory-to-register pada variabel yang sama. Global register allocation berkenaan dengan penugasan atau menggunakan register dibeberapa basic block atau prosedur. Subset pada register dipesan untuk penugasan lokal dan sisanya ditugaskan untuk global.

Sebagai tambahan, beberapa register dipesan oleh bagian back-end compiler untuk keperluan (pencatatan harian) bookkeeping. Tergantung pada arsitekturnya, termasuk juga register sebagai stack pointer, base pointer, frame pointer, dan/atau instruksi pointer. Umumnya register ini ditentukan oleh hardware, yang dikhususkan ke fungsi respektifnya. Untuk arsitektur intel 80x86, register dikhususkan ini termasuk, segment, stack, base pointer, dan instruksi register pointer.

6.2 Single-Use Register Allocation

Bentuk sederhana dan paling dasar dari pengalosasian register sepanjang waktu pembuatan code adalah *single-use* register allocation. Single-user register allocation maksudnya adalah nilai atau nilai-nilai yang ditempatkan dalam register diperlukan sebagai instruksi dan disimpan kembali ke memory setelah instruksi.

Back-end, menggunakan single-user register allocation akan mengubah ekspresi:

```
a = b + c
```

kedalam kode berikut:

```
mov ax, word ptr ss:[bp-4]]
add ax, word ptr ss:[bp-6]]
mov word ptr ss: [bp-2 ], axx
```

kelebihan dari pendekatan ini adalah sangat kecil akan terjadinya waktu compile overhead. Juga, rancangan back-end menjadi lebih sederhana dan berukuran kecil. Kerurangan utama nya adalah code yang dihasilkan sangat tidak efisien. Kebutuhan yang besar dan kemungkinan terjadinya pengeoperasian yang overload pada memory-to-register dan register-to-meory.

6.3 Algoritma Register Allocation

Ada tiga algoritma populer dalam register allocation yaitu:

1. Naive Register Allocation
2. Linear Scan Algorithm
3. Chaitin's Algorithm

Naive Register Allocation

- Naive Register Allocation berdasarkan pada asumsi bahwa variabel-variabel disimpan didalam memory utama (*Main Memory*)
- Performa pengopreasian tidak bisa dilakukan secara langsung terhadap variabel yang tersimpan dalam memory utama
- Variabel dipindahkan ke register sehingga memungkinkan pengoperasian diselesaikan menggunakan ALU

- Alu berisikan sebuah register temporary dimana variabel dipindahkan sebelum pengoperasian aritmatik dan logic
- Sekali pengoperasian dinyatakan selesai kita perlu menyimpan hasilnya kembali ke memory utamaMengirimkan variabel-variabel ke dan dari memory utama akan mengurangikecepatan eksekusi.

Contoh pengoperasian:

```
a = b + c
d = a
c = a + d
```

Variabel-variabel yang disimpan didalam memory utama:

```
a       b       c       d

2 fp      4 fp      6 fp      8 fp
```

Tingkatan instruksi mesin:

```
LOAD R1, _4fp
LOAD R2, _6fp
ADD   R1, R2
STORE R1, _2fp
LOAD R1, _2fp
STORE R1, _8fp
LOAD R1, _2fp
LOAD R2, _8fp
ADD   R1, R2
STORE R1, _6fp
```

Kelebihan dari dari algoritma ini adalah:

- Pengoperasianya mudah untuk dipahami dan aliran variabel-variabel dari memory utama ke register dan sebaliknya.
- Cukup hanya dengan dua register untuk melakukan pengoperasian
- Kompleksitas rancangan lebih sedikit

Keurangan :

- Waktu kompleksitas mengalami peningkatan saat variabel-variabel dipindahkan ke register dari memori utama
- Terlalu banyak instruksi LOAD dan STORE

- Untuk mengakses sebuah variabel kedua kalinya perlu untuk menyimpannya (STORE) ke memory utama untuk merekam setiap perubahan yang dibuat dan kemudian di (LOAD) kembali

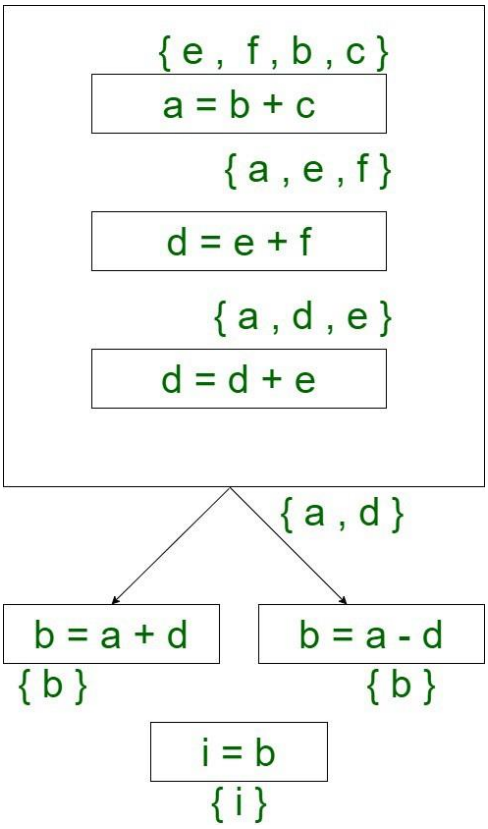
- Tidak cocok untuk compiler modern

Algoritma Linear Scan

- Meknisme algoritma Linear scan adalah global register allocation
- Pendekatan bottom up
- Jika n variabel berada pada point tertentu maka hanya memerlukan n register
- Dalam algoritma ini variabel discan secara linear untuk menentukan rentang pada variabel berdasarkan pengalokasian registernya
- Ide utama dibalik algoritma ini adalah minimum jumlah pengalokasian terhadap register dengan demikian register dapat digunakan kembali dan seluruhnya tergantung pada rentang variabel-variabel
- Algoritma ini memerlukan implementasi analisis variabel pada optimisasi code.

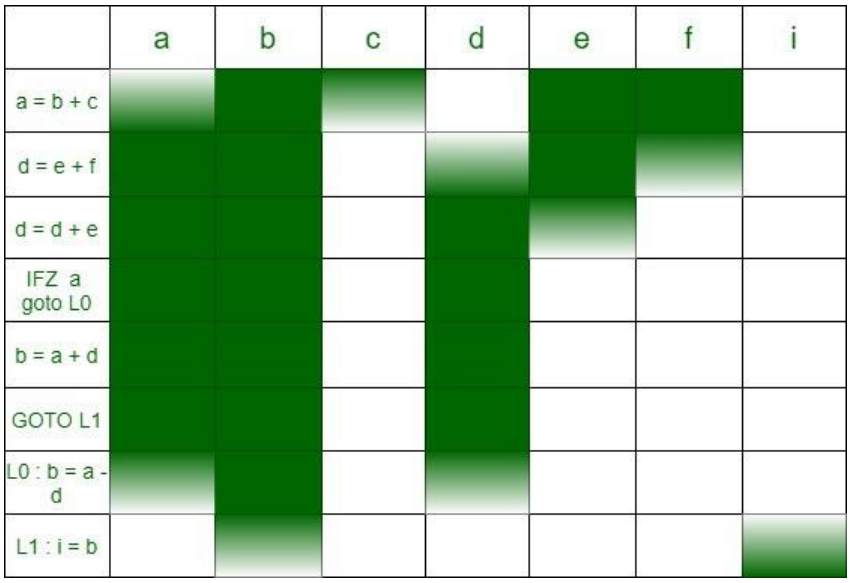
```
a = b + cd
= e + fd =
d + e
IFZ a goto L0
b = a + d
goto L1
L0 : b = a – d
L1 : i = b
```

Kontrol aliran graph:



Gambar 6.1: Graph Kontrol

- Pada titik waktu apapun dalam contoh ini jumlah maksimum variabel-variabel adalah empat. Karenanya kita memerlukan empat register untuk register allocation maksimum.



Gambar 6.2: Diagram Register Allocation

Jika kita gambar garis horizontal pada point tertentu dalam digram di atas kita dapat melihat bahwa kita memerlukan empat register untuk melakukan kinerja

pengoperasian dalam program.

Splitting

- Terkadang jumlah register yang diperlukan tidak tersedia. Dalam masalah seperti ini kita perlu memindahkan beberapa variabel dari dan ke RAM. Proses seperti ini dikenal dengan spilling.
- Spilling dapat dilakukan secara efektif dengan memindahkan sejumlah variabel yang jarang digunakan dalam program.

6.4 Graph Coloring(Chaitin's Algoritma)

- Register allocation dinterpretasikan sebagai sebuah permasalahan pewarnaan graph
- Node-node menyatakan rentang dari variabel
- Edges menyatakan hubungan antara dua rentang
- pemberian warna ke node-node tidak ada dua node-node yang berdekatan memiliki warna yang sama
- Jumlah warna menyatakan jumlah minimu yang diperlukan register

k-coloring pada graph dipetakan ke k register. Langkah-langkah:

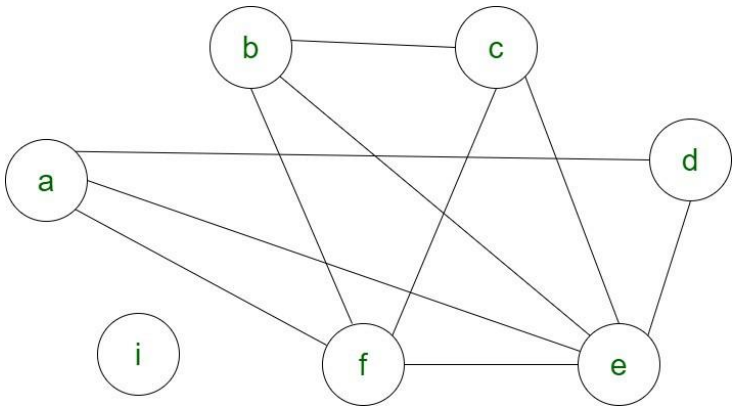
1. Pilih sebuah sembarang node pada tingkatan lebih kecil dari k.
2. Lakukan Push pada node kedalam stack dan hapus semua edge pada node yang keluar
3. Cek apakah sisa edge memiliki tingkatan kurang dari k, jika YES lanjut ke 5, jika tidak lanjut ke #
4. Jika tingkatan sisa dari vertex apapun lebih kecil dari k maka lakukan push kedalam stack
5. Jika tidak ada lagi edge yang tersedia untuk dilakukan push dan jika semua edge sudah berada di stack, lakukan POP tiap-tiap node dan berikan warna sehingga tidak ada dua node yang berdekatan memeilik sama warna.
6. Jumlah warnah yang diterapkan ke node-node adalah jumlah minimum dari register yang diperlukan.

spill beberapa node berdasarkan rentang keberadaannya dan kemudian coba lagi dengan nilai k yang sama. Jika masalah berslanjut berarti nilai k yang diasumsikan tidak dapat menjadi jumlah nilai minimum pada register. Coba untuk menaikkan

nilai k dengan menambahkan 1 dan coba kembali keselruhan prosedur.

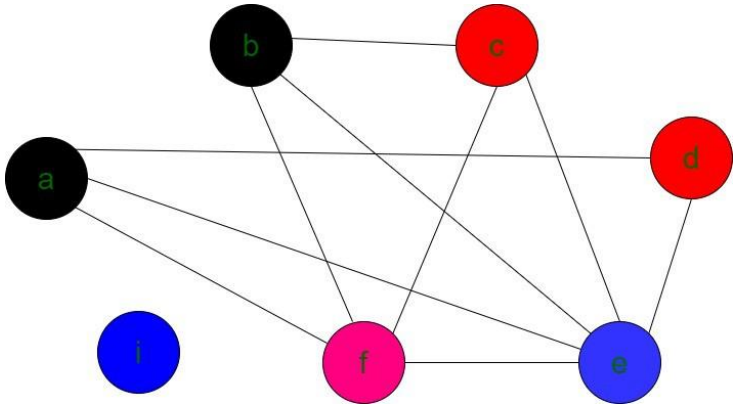
Untuk instruksi yang sama seperti yang sudah disebutkan di atas pewarnaan graph akan nampak seperti berikut:

Asumsi k = 4:



Gambar 6.3: Graph sebelum pewarnaan

Setelah pewarnaan graph dilakukan, diperolah akhir graph seperti berikut:



Gambar 6.4: Graph sesudah pewarnaan

Warna apapun(register) dapat diberikan ke ’i’ karena tidak memiliki edge ke node lainnya.

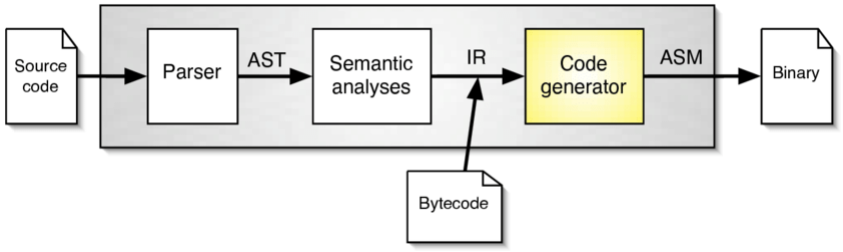
LATIHAN

Berapa jumlah minimum register yang diperlukan untuk menghindari spilling?

BAB 7
PEMBANGKIT CODE (CODE GENERATION)

Code generation adalah proses dimana pembangkit code compiler mengkonversi beberapa representasi intermediate kode sumber (source code) kedalam bentuk kode mesin yang dengan mudah dapat dijalankan oleh mesin. Code generation merupakan tahapan terakhir pada proses kompilasi.

Input pada generator code biasanya terdiri dari pohon parse *(parse tree)* atau *abstract syntax tree*. Pohon (tree) dikonversikan kedalam rentetan instruksi linear, biasanya dalam sebuah bahasa intermediate seperti three-address code.



Gambar 7.1: Skematik Code Generator

Compiler tersebut perlu menghasilkan program target secara efisien, termasuk tahapan optimisasi yang menjadi prior dalam pembangkit kode ini. Pemetaan optimisasi IR kadalam IR dapat menghasilkan kode yang efisien. Pada umumnya, Optimisasi kode dan pembangkit code adalah tahapan sebuah compiler, sering disebut sebagai tahapan pada sisi back-end.

7.1 Persoalan-persoalan dalam merancang codegenerator

Persoalan-persoalan Seperti memory management, instruction selection, register allocation dan evaluation order adalah masalah-masalah yang ada dihampir semua code generation. Pada bab ini, kita akan menjelaskan permasalahan yg umum dlm merancang pembangkitan kode. Input ke code generator. Input ke pembangkit code terdiri dari representasi source program lanjutan yang menghasilkan ujung akhir, bersamaan dengan informasi di tabel symbol yg di gunakan untuk menentukan

alamat run-time dari data objek yang ditandai dgn nama dan representasi lanjutan. Target program Output dari pembangkitan code adalah target program. Output ini mungkin memiliki bentuk yang berbeda: bahasa mesin absolut, bahasa mesin relocatable, atau bahasa assembly. Membuat program bahasa mesin sebagai output memiliki pengembangan yang ditempatkan di lokasi memori yg tetap dan dieksekusi segera. Program yg kecil dapat di-compile dan dieksekusi dengan cepat.

7.1.1 Penyeleksian Instruksi

Code generator harus memetakan program IR kedalam rentetan code yang dapat dijalankan oleh mesin target. Kompleksitas performa pemetaan ini ditentukan oleh faktor berikut:

- Level IR
- Set-instruksi alami aksitektur
- Kode yang dihasilkan berkualitas

Jika IR adalah bahasa pemrograman tingkat tinggi, code generator akan menterjemahkan tiap-tiap statement IR kedalam rentetan instruksi mesin menggunakan code templates. Seperti code generation statement-by-statement, bagaimanapun, code yang dihasilkan dengan kualitas rendah memerlukan optimisasi. Jika IR mencerminkan beberapa detail yang mendasari low-level mesin, maka code generator dapat menggunakan informasi ini untuk menghasilkan rentetan code yang lebih efisien.

Instruksi set alami pada mesin target mempunyai dampak kekuatan pada sulitnya Penyeleksian instruksi. Sebagai contoh, keseragaman dan kelengkapan set instruksi merupakan faktor penting. Jika mesin target tidak mendukung tiap-tiap tipe data dalam artian cara yang seragam, maka tiap-tiap exception dengan aturan umum memerlukan penangan khusus. Pada bebera mesin, misalnya, operasi floating-point diselesaikan menggunakan register terpisah.

Untuk tiap-tiap tipe dari statement three-address, kita dapat merancang sebuah skeleton code yang mendefinisikan code target yang dihasilkan untuk rancangan tersebut. Sebagai contoh, Masing-masing tipe statement three-address dengan bentuk x = y + z, dimana x, y dan z dialokasikan secara statik, dapat diterjemahkan kedalam rentetan code

```
LD R0, y        // R0 = y        (load y kedalam register R0)
ADD R0, R0, z   // R0 = R0 + z   (add z ke R0)
ST x, R0        // x = R0        (store R0 kedalam x)
```

Strategi ini sering menghasilkan redudansi load dan store. Sebagai contoh, rentetan dari statement three-address

```
a = b + c
d = a + e
```

akan diterjemahkan kedalam

```
LD R0, b    //R0 = b
ADD  R0, R0, c  //R0 = R0 + c
ST a, R0    //a = R0
LD R0, a    //R0 = a
ADD  R0, R0, e  //R0 = R0 + e
ST d, R0    //d = R0
```

Disini, statement ke empat adalah redudansi karena nilai yang barus disimpan di load kembali, dan begitu juga statement ke 3 jika a sesudah itu tidak digunakan.

Kualitas paa code yang dihasilkan bisanya ditentukan oleh ukuran dan kecepatannya. pada kebanyakan mesin, program IR yang diberikan dapat diimplementasikan dengan banyak rentetan code berbeda, dengan perbedaan yang signifikan antara biaya dan implementasi. Penerjemah naive pada code intermediate akan membuat code yang benar, tetapi code target menjadi sangat tidak efisien.

sebagai contoh, jika mesin target memiliki sebuah instruks "increment" (INC), maka statement three-address a = a + 1 mungkin lebih efisien jika diimplementasikan dengan instruksi tunggal INC a, dari pada dengan rentetan load a kedalam sebuah register, add satu ke register, dan kemudian simpan kembali hasilnya kedalam a:

```
LD R0, a    //R0 = a
ADD  R0, R0, #1 //R0 = R0 + 1
ST a, R0    //a = R0
```

7.1.2 Model Mesin Target Sederhana

Model komputer target mesin three-address dengan operasi load dan store, operasi komputasi, operasi jump, dan operasi kondisional jump. Yang mendasari computer adalah sebuah mesin pengalamatan byte dengan tujuan umum register n, RO, R1, ... Rn - 1. Kebanyakan instruksi terdiri dari sebuah operator, diikuti dengan target, diikuti dengan list pada sumber operand. Asumsi dari sebuah instruksi yang tersedia:

- Operasi *Load*: Instruksi LD *dst*, *addr* load nilai dalam lokasi *addr* kedalam lokasi *dst*. Instruksi ini menunjukkan penugasan *dst* = *addr*. Bentuk paling umum pada instruksi ini adalah LD *r*, *x* yang meload nilai dalam lokasi *x* kedalam register *r*. Sebuah buntuk instruksi pada LD r_1, r_2 adalah sebuah copy *register_to_register* dimana konten pada register r_2 dicopy kedalam register r_1.

- Operasi *store*: Instruksi ST *x*, *r* penyimpan nilai pada register *r* kedalam lokasi *x*. Instruksi ini disimbolkan dengan *assignment* *x* = *r*.

- Operasi *Computation*: Dalam bentuk *OP dst*, src_1, src_2, dimana *OP* adalah operator seperti ADD atu SUB, dan *dst*, src_1, src_2 menyatakan lokasi. Efek

dari instruksi mesin ini adalah untuk menerapkan operasi yang dinyatakan dengan *OP* pada nilai-nilai dalam lokasi src_1 dan src_2, dan meletakkan hasil dari operasi ini kelokasi *dst*. Sebagai contoh, SUB r_1, r_2, r_3 hitung $r_1 = r_2 - r_3$. Nilai apapun yang sebelumnya disimpan dalam r_1 akan hilang, tetapi jika $r_1$adalah r_2 atau r_3 , nilai yang lama dibaca pertama kali. Operator Unary yanghanya mengambil satu operand tidak memiliki src_2.

- *Unconditional jumps*: Instruksi BR *L* menyebabkan percabangan kontrol ke instruksi mesin dengan label *L* (BR artinya cabang)
- *Conditional Jumps* pada bentuk B*cond r*, *L*, dimana *r* adalah sebuah register, *L* adalah sebuah lebel, dan *cond* adalah salah satu test pada nilai-nilai dalam register *r*. Sebagai contoh, BLTZ*r*, *L* menyebabkan sebuah loncatan (jump) ke label *L* jika nilai dalam register *r* lebih kecil dari 0, dan memperbolehkan control untuk melewatkan instruksi ke mesin selanjutnya.

7.1.3 Program dan Biaya Instruksi

Kita sering mengasosiasikan sebuah program dengan biaya (cost) compiling dan running. Tergantung dari aspek program apa yang ingin dioptimimalkan, beberapa berkaitan dengan pengukuran panjang waktu durasi proses compile, ukuran dan konsumsi listrik yang digunakan.

Menentukan cost compiling dan program yang berjalan secara aktual adalah permasalahan yang kompleks.Secara umum menemukan sebuah program target yang optimal untuk sebuah source program yang diberikan adalah masalah yang tidak dapat dipecahkan, dan banyak sub-masalah lain terlibat adalah NP-hard. Seperti yang telah kita indikasi, dalam pembangkit code kita sering mengandalkan teknik heuristik yang dapat menghasilkan target program yang baik, tetapi terkadang belum tentu target program tersebut optimal.

Dalam pembahasan ini, kita berasumsi bahwa tiap-tiap instruksi bahasa target terasosiasi dengan cost. Sederhananya, kita ambil cost pada sebuah instruksi menjadi satu penambahan cost yang diasosiasikan dengan mode pengalamatan pada operand. Cost disini berkenaan dengan panjang kata dari instruksi. Mode pengalamatan yang melibatkan register memiliki 0 tambahan cost, ketika melibatkan lokasi memory atau nilai constans memiliki satu tambahan cost, karena operand harus disimpan didalam kata-kata yang mengikuti instruksi. Beberapa contoh:

- Instruksi LD R0, R1 menyalin content pada register R1 kedalam register RO. Instruksi ini memilik satu cost karena tidak ada tambahan memory word yang diperlukan.
- Instruksi LD R0, M load content pada lokasi memory M kedalam register R0. Proses ini cost nya adalah 2 karena alamat lokasi memory M ada di word yang mengikuti instruksi.
- Instruksi LD R1, *100(R2) load kedalam register R1 nilai diberikan oleh *contents*(*contents*(100 + *contents*(R2))). Cost nya adalah dua karena nilai

constans 100 disimpan didalam word mengikuti instruksi.

Algoritma pembangkit kode yang bagus akan berusahan meminimalkan jumlah cost pada instruksi yang dijalankan oleh target program yang dihasilkan berdasarkan tipe inputan. Dapat kita ketahui bahwa pada beberapa situasi, secara aktual kita dapat membangkitkan code optimal untuk ekspresi pada pengkelasan tertentu dari mesin register.

7.2 Basic Blocks

Basic block merupakan rentetan code dimana instruksi dijalankan secara berurutan, tanpa ada looping dan testing *(straight-line)*, tidak memiliki cabang masuk dan cabang keluar kecuali untuk entri dan akhir dari code. Basic block adalah sebuah set statement-stament yang dieksekusi satu-persatu, secara berurutan.

Pengerjaan pertama adalah melakukan partisi pada rentetan three-address code kedalam basic block. Basic block yang baru dimulai dengan instruksi pertama dan instruksi akan ditambahkan sampai label atau *jump* terpenuhi. Dengan tidak adanya labels atau jump, Secara berurutan meneruskan kontrol dari satu instruksi ke berikutnya. Ide ini diformalkan dalam algoritma berikut:

Algorithm 4 Instruksi partisi three-addres kedalam basic blocks

---

INPUT: Rentetan Instruksi pada three-address

OUTPUT: List pada basic block untuk urutannya dimana tiap-tiap instruksi ditugaskan (assignment) ke satu basic block secara tepat

METHOD: Pertama, kita tentukan instruksi tersebut kedalam code intermediate yaitu *leaders*, yaitu, instruksi pertama didalam beberapa basic block. Instruksi hanya melewati akhir program intermediate program, bukan termasuk sebagai leader. Aturan untuk menemukan leaders adalah:

1. Instruksi pertama three-address didalam kode intermediate adalah leader
2. Instruksi apapun yang menjadi target oleh sebuah *conditional* atau *uncoditional* jump/stament goto adalah leader
3. Instruksi apapun yang secara langsung diikuti *conditional* atau *uncoditional* jump/statement goto adalah leader

---

Kemudian, untuk tiap-tiap leader ini ditentukan basic blocknya berisikan basic block itu sendiri dan semua instruksi hingga tidak termasuk leader berikutnya. Contoh matrik code intermediate ditetapkan 10*10 ke sebuah matrik identitas:

```
1)  i=1          //Leader 1 (Statement pertama)
2)  j=1               //Leader 2 (Target terhadap statement ke 11)
3)  t1 = 10 * i     //Leader 3 (Target terhadap statement ke 9)
4)  t2 = t1 + j
5)  t3 = 8 * t2
```

```
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9)  if j <= goto (3)
10) i = i + 1  //Leader 4 (Secara langsung setelah statement Conditional
11) 11) if i <= 10 goto (2)
12) i = 1    //Leader 5 (Secara langsung setelah statement Conditional
13) t5 = i – 1  //Leader 6 (Target terhadap statement ke 17)
14) 14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Algoritma digunakan untuk konversi sebuah matrix kedalam matrix identitas yaitu matrix yang semua element diagonal 0 dan semua element lainnya sebagai 1. Step (3)-(6) digunakan untuk membuat element 0 dan step (14) digunakan untuk membuat element 1. Langkah-langkah ini digunakan secara rekursiv dengan statement goto.

Terdapat 6 Basic block pada code diatas:

B1) Statement 1
B2) Statement 2
B3) Statement 3-9
B4) Statement 10-11
B5) Statement 12
B6) Statement 13-17

Contoh pseudocede untuk code intermediate di atas:

```
for i from 1 to 10 do
   for j from 1 to 10 do
      a[i,j] = 0.0;
for i from 1 to 10 do
   a[i,i ] = 1.0;
```

Pertama, instruksi 1 adalah leader dengan aturan (1) pada algoritma Instruksi partisi three-addres di atas. Untuk mencari leader lainnya, pertama kita perlu mencari *jumps*. Dalam contoh ini, terdapat tiga *jumps*, semua conditional, pada instruksi 9, 11, dan 17. Dengan aturan (2), target pada *jumps* ini adalah leaders; mereka adalah instruksi 3, 2, dan 13, berdasarkan urutan. Kemudian, dengan aturan (3), tiap-tiap instruksi setelah *jump* adalah leader, itu adalah instruksi 11 dan 12. Perhatikan bahwa tidak ada instruksi lagi setelah 17 di dalam kode tersebut, tetapi jika terdapat code setelahnya , maka instruksi ke 18 itu juga jadi leader. Kita menyimpulkan bahwa leaders adalah instruksi 1, 2, 3, 10, 12, dan 13.

7.3 Optimasi Code

Optimasi code pada tahapan sintesis adalah sebuah teknik transformasi program, yang mencoba melakukan improvisasi terhadap intermediate code dengan tujuan agar konsumsi resource (CPU, Memory) lebih sedikit sehingga menghasilkan kode mesin yang berjalan cepat.

Terdapat tiga teknik dalam melakukan optimasi code yaitu :

1. Dependensi Optimasi
2. Optimasi Lokal
3. Optimasi Global

7.3.1 Dependensi Optimasi

Bertujuan untuk menghasilkan kode program yang berukuran lebih kecil dan lebih cepat. Berdasarkan ketergantungannya pada mesin, optimasi dibagi menjadi: Machine Dependent Optimizer: Kode dioptimasi sehingga lebih efisien pada mesin tertentu

- Machine Independent Optimizer: Strategi optimasi yang bisa diaplikasikan tanpa tergantung pada mesin tujuan tempat kode yang dihasilkan akan dieksekusi nantinya. Optimasi yang dilakukan meliputi optimasi lokal dan optimasi global.

7.3.2 Optimasi Lokal

Optimasi Lokal adalah optimasi yang dilakukan hanya pada suatu blok dari source code. Caranya sebagai berikut:

- Folding
- Redudant-Subexpression Elimination
- Optimasi dalam sebuah iterasi
- Strength Reduction

Folding

Mengganti konstanta atau ekspresi yang bisa dievaluasi pada saat compile time dengan nilai komputasinya. Contoh:

```
A := 2 + 3 + B
```

bisa diganti menjadi

```
A := 5 + B
```

dimana 5 menggantikan ekspresi 2 + 3

Redundant-Subexpression Elimination

Menggunakan hasil komputasi terdahulu daripada melakukan komputasi ulang. Misalnya terdapat urutan instruksi :

```
A := B + C
X := Y + B + C
```

Kemunculan B + C yang bersifat redundan, bisa memanfaatkan hasil komputasi sebelumnya selama tidak ada perubahan nilai variabel.

Optimasi dalam sebuah iterasi

Terdiri dari 2 macam optimasi yaitu: *Loop Unrolling*: Menggantikan suatu loop dengan menulis statement dalamloop beberapa kali. Hal ini karena sebuah iterasi pada implementasi level rendah akan memerlukan operasi :

- Inisialisasi/pemberian nilai awal pada variabel loop. Dilakukan sekali pada saat permulaan eksekusi loop
- Pengetesan, apakah variabel loop telah mencapai kondisi terminasi.
- Adjustment, yaitu penambahan atau pengurangan nilai pada variabel loop dengan jumlah tertentu.
- Operasi yang terjadi pada tubuh perulangan (loop body)

Contoh instruksi:

```
FOR I := 1 to 2 DO
  A [I] := 0 ;
```

dioptimasi menjadi

```
A [1] := 0 ;
A [2] := 0 ;
```

Sehingga memerlukan 2 instruksi assignment saja

Terdapat instruksi untuk inisialisasi I menjadi 1. Serta operasi penambahan nilai/increment 1 dan pengecekan nilai variabel I pada setiap perulangan. Sehingga untuk perulangan saja memerlukan 5 instruksi, ditambah dengan instruksi assignment pada tubuh perulangan menjadi 7 instruksi.

2. *Frequency Reduction*: Pemindahan statement ke tempat yang lebih jarang dieksekusi. Contohnya :

```
FOR I := 1 TO 10 DO
BEGIN
 X := 5 ;
 A := A + I ;
END ;
```

Variabel X dapat dikeluarkan dari iterasi menjadi

```
X := 5 ;
 FOR I := 1 TO 10 DO
 BEGIN
 A := A + I ;
END ;
```

Strength Reduction

Mengganti suatu operasi dengan jenis operasi lain yang lebih cepat dieksekusi. Misalnya, pada beberapa computer operasi perkalian memerlukan waktu lebih banyak dari pada operasi penjumlahan, oleh karena itu bias dilakukan pengehematanwaktu dengan mengganti operasi perkalian tertentu dengan penjumlahan. Contohlainnya:

```
A := A + 1

dapat diganti dengan INC (A) ;
```

7.3.3 Optimasi Global

Optimasi global biasanya dilakukan dengan analisis flow, yaitu suatu graph berarah yang menunjukkan jalur yang mungkin selam eksekusi program. Ada 2 kegunaan optimasi global, yaitu bagi programmer dan bagi kompilator itu sendiri.

1. Bagi programmer: *Unreachable atau dead code* adalah kode yang tidak akan pernah dieksekusi misalnya:

```
X := 5
IF X = 0 THEN
 A := A + 1
```

Instruksi A := A + 1 tidak akan pernah dieksekusi

2. Bagi programmer: *Unused parameter pada prosedur* adalah Parameter yang tidak pernah digunakan di dalam prosedur,misalnya:

```
Procedure Jumlah (a,b,c : integer)
var x : integer
begin
 x := a + b
end ;
```

Parameter c tidak pernah digunakan di dalam prosedur, sehingga seharusnya tidak perlu diikutsertakan.

3. Bagi programmer: *Unused variabel* adalah Variabel yang tidak pernah dipakai di dalam program, misalnya:

```
Program pendek;
var a, b : integer;
begin
 a := 5;
 end ;
```

Variabel b tidak pernah dipergunakan di dalam program, sehingga bias dihilangkan.

4. Bagi programmer: *First Value Unused* adalah Variabel yang dipakai tanpa nilai awal, misalnya:

```
Program awal;
var a, b : integer;
begin
a := 5;
 a := a + b;
 end ;
```

Variabel b digunakan tanpa memiliki nilai awal atau belum di-assign

Bagi compiler optimasi global dapat meningkatkan efisiensi eksekusi program dan menghilangkan useless code atau kode yang tidak terpakai.

LATIHAN

2. Lakukanlah optimalisasi lokal untuk potongan program berikut ini:

```
A:= B + 10 * 4;C:=  B  +  D;
F:=  B  +  D  -  G;
for I:=1 to 100 do begin
X:= X + I; A:= A + X; B:= 7;
end;
```

Daftar Pustaka

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[2] Yohanes Nugroho. Membuat interpreter/compiler itu mudah, 2019. https://yohan.es/compiler/, Last accessed on 2021-11-30.

[3] Torben Ægidius Mogensen. *Introduction to Compiler Design, Second Edition*. Undergraduate Topics in Computer Science. Springer, 2017.

Penerbit :
Universitas Islam Kalimantan
Muhammad Arsyad Al -Banjary

Alamat :
Gedung A UPT Publikasi dan Pengelolaan Jurnal
Universitas Islam Kalimantan
Muhammad Arsyad Al-Banjary

Jl. Adhyaksa No. 2 Kayutangi
Banjarmasin, Kalimantan Selatan
Telepon : 0511 - 3304352
FAX : 0511

ISBN 978-623-7583-83-7