Majalah Ilmiah Matematika dan Komputer, April 2006
ISSN 0216-4728

PEMBENTUKAN AUTOMATA HINGGA DETERMISTIK MENGGUNAKAN POHON SINTAKS

Maukar

Jurusan Teknik Informatika, Fakultas Teknologi Industri
Universitas Gunadarma
maukar@staff.gunadarma.ac.id

ABSTRAK

Dalam proses kompilasi untuk bahasa sumber, tahap analisis leksikal diantaranya berperan sebagai pengenal suatu token yang ada dalam bahasa sumber tersebut. Automata Hingga Deterministik merupakan salah satu alat yang cukup efektif dijadikan sebagai pengenal. Banyak teknik yang dapat digunakan untuk membentuk Automata Hingga Determinsitik, diantaranya teknik yang menggunakan pohon sintaks. Dengan teknik ini ekspresi beraturan yang dipakai sebagai pola pembentukan suatu token, akan diubah menjadi Automata Hingga Determinsitik, antara lain mengunakan fungsi *nullabel, firstpos, lastpos,* dan *followpos.* Tulisan ini mencoba untuk mendeskripsikan proses pembentukan Automata Hingga Deterministik tesebut secara lugas dengan metodologi yang cukup efektif, artinya proses ini akan menghasilkan keluaran Automata Hingga Determinstik dengan jumlah *state* minimal

Kata Kunci: Analisis Leksikal, Automata Hingga Determinsitik, Pohon Sintaks.

PENDAHULUAN

Proses penterjemahan dari suatu bahasa sumber menjadi bahasa target (proses kompilasi) dilakukan dengan melalui beberapa tahap yang satu sama lain saling terkait erat, bahkan sulit untuk dipisahkan secara tegas (terkadang ada pekerjaan yang tumpang tindih antara satu tahap dengan tahap lainnya).

Analisis Leksikal merupakan tahap awal dalam prose kompilasi, dengan tugas utamanya antara lain: mengidentifikasi token yang terdapat dalam program sumber. Tentu saja, agar sebuah token dapat diidentifikasi, token tersebut harus dideskripsikan menggunakan suatu ekspresi (biasa dinamakan sebagai ekspresi beraturan (EB)). Dengan demikian, setiap token yang ada dalam suatu bahasa pemrograman akan memiliki satu ekspresi beraturan yang unik.

Untuk menjalankan fungsinya sebagai pengenal token, tahap analisis leksikal akan menggunakan suatu alat bantu. Alat bantu tersebut adalah Automata Hingga. Automata Hingga dapat dibedakan menjadi Automata Hingga Nondeterministik (AHN) dan Automata Hingga Deterministik (AHD).

AHN dalam proses pengenalan token memiliki kelemahan dari segi waktu yang digunakan relatif lebih lama dibandingkan dengan AHD, tetapi jumlah *state* yang terdapat dalam suatu AHN secara umum relatif lebih sedikit dibandingkan dengan AHD (untuk mengenal token yang sama).

Tulisan ini mencoba untuk mendeskripsiakn proses pembentukan AHD secara lugas dan terperinci menggunakan teknik pohon sintaks.

TINJAUAN PUSTAKA

Sebagai suatu model matematik AHD memiliki komponen sebagai berikut: himpunan *state* (S), himpunan simbol input (Σ), sebuah *state* awal (S_0, $S_0 \in S$), himpunan *state* penerima (F, $F \subseteq S$), dan fungsi transisi (δ, $\delta(s_i, a_i) \rightarrow s_j$, dimana $s_i, s_j \in S$ dan $a_i \in \Sigma$).

AHD merupakan Automata Hingga yang mempunyai properti khusus, yaitu tidak diperbolehkan adanya transisi atas input string kosong (ε-transisi) dan setiap transisi yang keluar dari suatu state selalu mempunyai label yang unik atau dengan kata lain tidak boleh ada transisi yang keluar dari suatu *state* ke *state* lain dengan input simbol sama.

State yang terdapat dalam sebuah AHD dapat diminimalisasi. Dengan demikian, untuk suatu AHD yang menerima bahasa yang dideskripsikan oleh suatu EB dapat mempunyai jumlah *state* minimal. Tidak semua teknik pembentukan AHD dari EB akan menghasilkan jumlah *state* minimal.

Salah satu cara untuk meminimalisasi *state* di dalam AHD adalah menggunakan teknik partisi. Teknik ini diawali dengan mempartisi AHD menjadi dua kelompok, yaitu kelompok *state* penerima dan kelompok *state* bukan penerima. Setiap kelompok yang mempunyai jumlah *state* ≥ 2 akan diuji apakah masih harus dipartisi atau tidak. Kelompok yang mempunyai minimal satu *state* melakukan transisi ke *state* dalam kelompok yang berbeda akan menyebabkan kelompok tersebut harus dipartisi. Proses ini dilakukan terus, hingga tidak ada lagi kelompok yang harus dipartisi. Pada akhir proses, setiap kelompok akan diwakili dengan satu *state* baru pada AHD.

Pohon sintaks yang digunakan dalam teknik pembentukan AHD ini adalah, pohon yang dibentuk berdasarkan ekspresi beraturan yang mendeskripsikan suatu token. Pohon sintaks tersebut dibentuk dengan memperhatikan operator presedensi dan assosiativitas yang terdapat dalam ekspresi beraturan tersebut. Untuk setiap ekspresi beraturan dipastikan akan mempunyai sebuah pohon sintaks yang unik, sehingga AHD yang terbentuk juga akan mempunyai sifat unik tersebut.

METODE PENELITIAN

Teknik pembentukan AHD untuk suatu ekspresi beraturan yung mendeskripsikan sebuah token melalui beberapa tahap sebagai berikut,

Diawali dengan pembentukan ekspresi beraturan *Augmented* (EBA), yaitu dengan menambahkan symbol # di akhir EB sebagai tanda akhir dari suatu input string.

Berdasarkan EBA tersebut dibuatlah pohon sintaks dengan memperhatikan presedensi dan assosiativitas dari operator yang ada dalam ekspresi beraturan. Setiap *node* daun yang terbentuk diberi nomor sesuai urutan, dan dimulai dari kiri ke kanan.

Untuk setiap *node* yang terdapat pada pohon sintaks, ditentukan nilai *nullable*, *firstpos*, dan *lastpos* dengan aturan seperti pada tabel 1 (lihat halaman 72).

Langkah selanjutnya menentukan nilai fungsi *followpos*(*i*) untuk setiap *node* yang berlabel operator *concatenation* dan *star*, dimana *i* adalah nomor posisi yang ada pada pohon sintaks. Penentuan *followpos* ini didasarkan pada hasil dari *nullable*, *firstpos*, dan *lastpos* menggunakan aturan sebagai berikut: (1). Jika n adalah *node* yang berlabel *concatenation* dengan anak kiri c_1 dan anak kanan c_2, dan *i* adalah sebuah posisi dalam *lastpos*(c_1), maka semua posisi dalam *firstpos*(c_2) adalah merupakan *followpos*(*i*); (2). Jika *n adalah*

Nilai *followpos* untuk setiap nomor posisi pada pohon sintaks, akan dijadikan sebagai acuan pembentukan AHD (fungsi transisi dalam AHD). Dengan terlebih dahulu menentukan *state* awal, yaitu nilai *firstpos*(akar pohon sintaks).

PEMBAHASAN

Dalam pembahasan ini, akan dipaparkan sebuah contoh kasus pembentukan AHD dari suatu ekspresi beraturan. Untuk bagian ini, diawali dengan mengambil sebuah EB: $a\ (b \mid a)^*\ \varepsilon$.

Dengan menambahkan symbol # di akhir EB tersebut, akan terbentuk sebuah EBA berikut: $a\ (b \mid a)^*\ \varepsilon\ \#$. Pohon sintaks yang dihasilkan berdasarkan EBA tersebut adalah seperti pada gambar 1 (lihat halaman 70). Setiap *node* daun diberi nomor posisi terurut mulai dari kiri ke kanan, termasuk *node* yang berlabel ε.

Berdasarkan pohon sintaks yang terbentuk dan menggunakan aturan pada tabel 1, akan diperoleh nilai *nullable*, *firstpos*, dan *lastpos* untuk setiap *node*. Hasilnya digambarkan oleh pohon sintaks pada gambar 2 (lihat halaman 71).

Untuk setiap *node* yang berlabel operator *concatenation* dan *star* dijadikan sebagai dasar untuk menentukan fungsi *followpos*(*i*). Dengan menerapkan aturan yang ada, diperoleh nilai *followpos* dari setiap nomor posisi sebagi berikut:

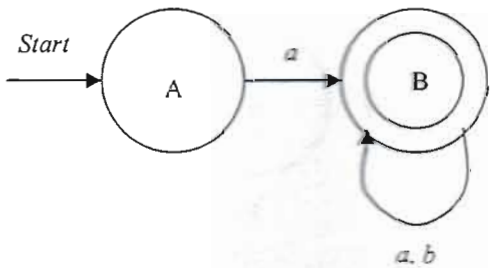
Tabel 2. Nilai *followpos*(i)

i	*Follopos*
1	{2,3,5}
2	{2,3,5}
3	{2,3,5}
4	{ }
5	{ }

Selanjutnya berdasarkan tabel 2 akan dibentuk funsgi transisi untuk AHD, dengan terlebih dahulu membentuk *state* awal, yaitu *fisrstpos*(akar pohon sintaks) dalam hal ini adalah *state* $A = \{1\}$. Fungsi transisi untuk *state A* adalah, $\delta(A, a) = followpos(1) = \{2,3,5\}$, dinamakan *state B*, yang juga merupakan *state* penerima. $\delta(A, b) = \{\ \}$. Fungsi transisi untuk *state B* adalah, $\delta(B, a) = followpos(3) = \{2,3,5\} = B$. $\delta(B, b) = followpos(2) = \{2,3,5\} = B$. Karena tidak ada lagi *state* baru yang terbentuk, maka proses dihentikan. Selengkapnya berdasarkan proses ini dihasilkan AHD seperti pada tebel 3 dan gambar 3 berikut.

Tabel 3. Tabel Transisi untuk AHD

State	Simbol Input	
	a	*b*
A	*B*	-
B	*B*	*B*



Gambar 3. AHD untuk $a\,(b \mid a)^*\,\varepsilon$

Melalui uji minimisasi AHD dapat dibuktikan bahwa setiap AHD yang terbentuk dengan teknik ini mempunyai jumlah *state* minimal. Teknik minimisasi yang digunakan adalah dengan melakukan partisi.

PENUTUP

Pembentukan AHD sebagai alat pengenal untuk identifikasi token dalam tahap analisis leksikal pada proses kompilasi menjadi hal penting, sehingga perlu digunakan suatu AHD yang jumlah *state*nya minimal untuk mengurangi sisi kelemahan AHD dalam hal jumlah *state*. Penggunaan teknik pohon sintaks untuk membentuk AHD dari sebuah ED merupakan salah satu cara yang dapat digunakan untuk memenuhi tujuan tersebut.

Dari hasil pembahasan dapat dilihat bahwa AHD yang terbentuk menggunakan teknik pohon sintaks terbukti memiliki keunggulan yaitu mempunyai jumah *state* minimal dengan uji partisi minimalisasi AHD.

Ke depan akan lebih memudahkan bagi pengembang kompiler untuk menggunakan teknik ini secara praktis, yaitu dengan membuat program yang menerima masukan berupa EB dan keluarannya adalah AHD yang dapat mengenali token yang dideskripsikan oleh EB tersebut dengan jumlah *state* minimal.

DAFTAR PUSTAKA

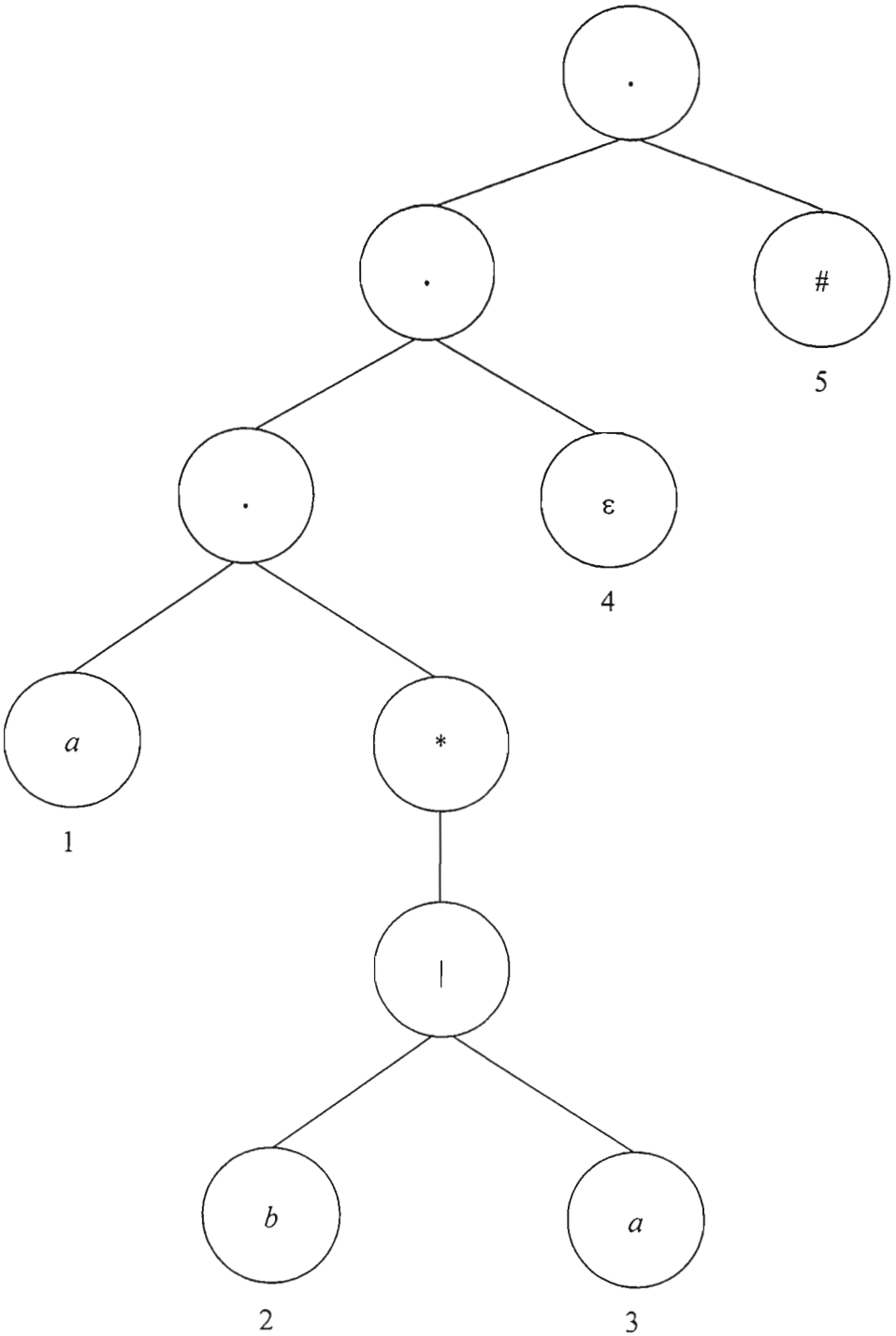
Aho, A.V., Rave S., and Jeffrey D.U. **Compilers: Principles, Techniques, and Tools**. Addison-Wesley. Canada. 1996.

Bennet, J.P. **Introduction to compiling techniques: a first course using ANSI C, LEX and YACC.** 2nd ed. McGraw-Hill. Cambridge. 1996

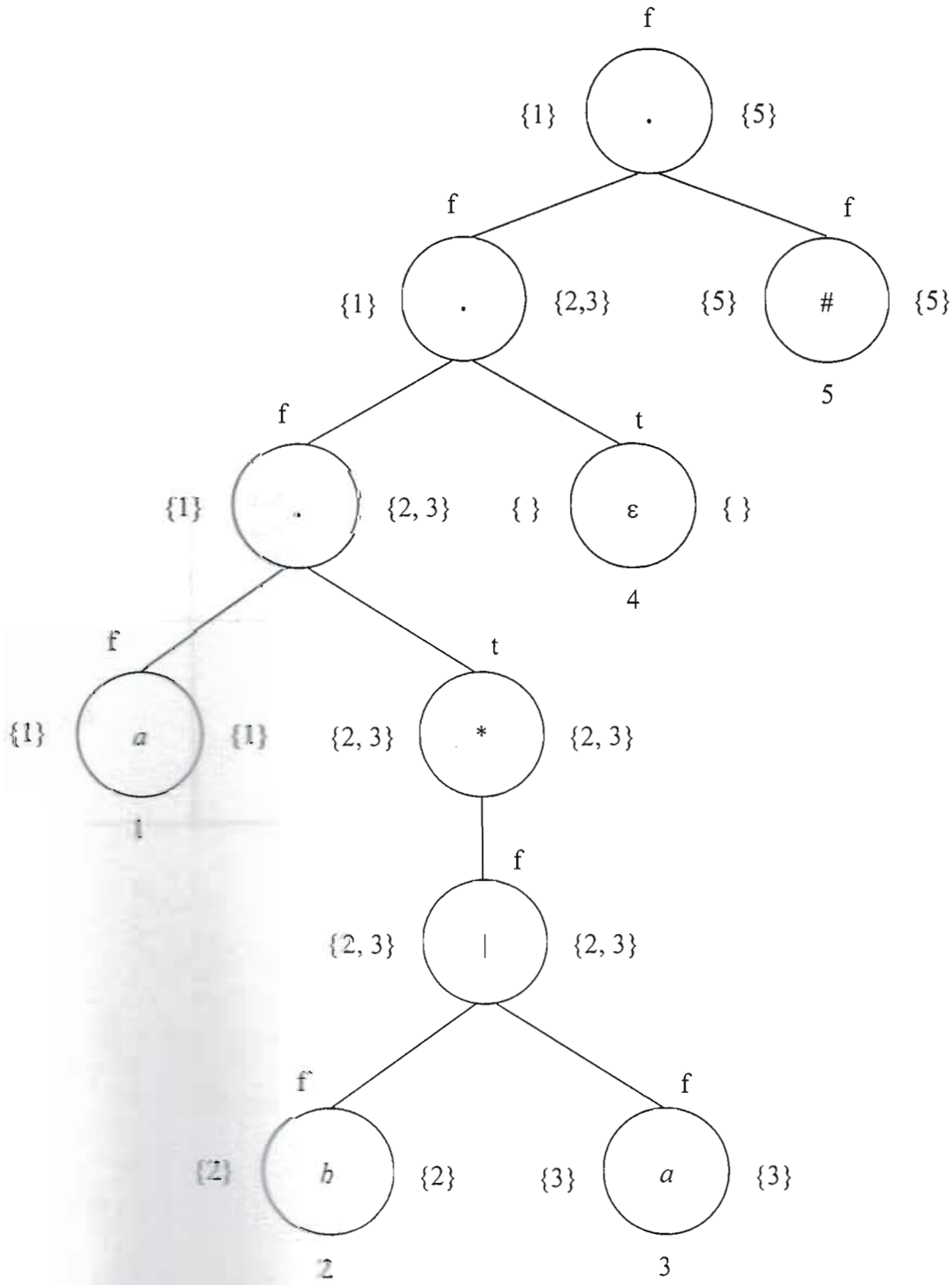
Fraser, C., and David H. **A Retargetable C Compiler: Design and Implementation**. Addison-Wesley. California. 1998.

Lewis, H.R., and Christos H.P. **Elements of The Theory of Computation**. 2th ed. Prentice-Hall. New Jersey. 1998.

Wilhelm, R., and Dieter M. **Compiler Design**. Addison-Wesley. CornWall. 1995.

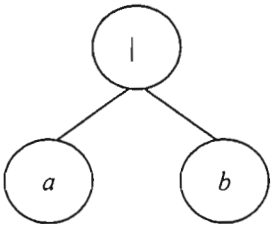
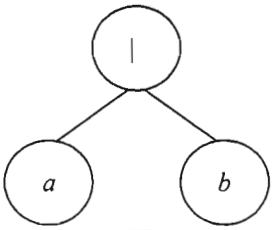
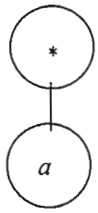


Gambar 1. Pohon Sintak Untuk $a\ (b \mid a)^*\ \varepsilon\ \#$



Gambar 2. Pohon Sintaks dengan *nullable, firstpos,* dan *lastpos*

Tabel 1. Aturan Untuk Menghitung Fungsi *nullable, Firstpos,* dan *lastpos*

Node n	*nullable*(*n*)	*firstpos*(*n*)	*lastpos*(*n*)
n adalah sebuah *node* daun yang berlabel ε	**true**	{ }	{ }
n adalah sebuah *node* daun yang berlabel selain ε dan bernomor posisi *i*	**false**	{*i*}	{*i*}
	*nullable*(*a*) **or** *nullable*(*b*)	*firstpos*(*a*) ∪ *firstpos*(*b*)	*lastpos*(*a*) ∪ *lastpos*(*b*)
	*nullable*(*a*) **and** *nullable*(*b*)	**If** *nullable*(*a*) **then** *firstpos*(*a*) ∪ *firstpos*(*b*) **else** *firstpos*(*a*)	**If** *nullable*(*b* **then** *lastpos*(*a*) ∪ *lastpos*(*b*) **else** *lastpos*(*b*)
	**true**	*firstpos*(*a*)	*lastpos*(*b*)