Model Pemroses Bahasa Pemrograman Dengan tools berbasis Java

OLEH

Heru Suhartanto
heru@cs.ui.ac.id

Fakultas Ilmu Komputer
Universitas Indonesia
(Fasilkom UI)

Depok, 16424
Indonesia

Abstrak

Buku petunjuk ini menjelaskan secara singkat teori dasar pengembangan suatu kompilator, dan penjelasan pengembangan model terhadulu yang berbasis Pascal ke basis Java. Tujuan utama pengalihan basis bahasa pemrograman adalah penyediaan keluwesan bagi pemakai untuk tidak tergantung pada kompilator (pascal) yang komersial dan lebih memudahkan pemakai untuk mengembangkan model kompilator tersebut bahasa Java yang gratis (public domain softwares)..

Dalam model terbaru ini seluruh komponent dikembangkan dengan bahasa Java dan menghasilkan kompilator dalam bahasa Java. Perbedaan dengan modul Pascal adalah dalam hal penganalisis leksikal dan sintak atau parsing. Pada model terdahulu, perancang suatu kompilator harus memahami teori dasar bahasa pemrograman (terutama leksikal dan sintak) dan mengimplementasikan teori tersebut langsung. Walaupun ini sangat bermanfaat dalam memahami bagaiman penerapan teori bahasa pemrograman, namun ia kurang efisien untuk pengembangan suatu pemroses bahasa dengan lebih efisien dan cepat. Pada model terhadulu, jika terjadi penambahan atau perubahan spesifikasi kompilator tersebut, perlu dilakukan perubahan yang banyak pada modul terkait yakni pada modul penganalisis leksikal dan sintak. Dengan pendekatan baru ini, hanya sedikit bagian yang perlu direvisi pada modul leksikal dan sintak, sehingga pemakai bisa berkonsentrasi untuk pengembangan model lebih lanjut.

Dalam pengembangan ini, JLEX atau JavaLex dipakai sebagai pembentuk penganalisis leksikal dan Cup atau JavaCup dipakai sebagai pembentuk penganalisis sintak. Perubahan pada analisis leksikal cukup dilakukan perubahan pada spesifikasi bagaimana pola suatu token dibentuk, dan perubahan tata bahasa (grammar) dalam suatu parsing cukup hanya dilakukan perubahan susunan atau penambahan fitur-fitur tatabahasa. Perancang pemroses/kompilator tidak perlu terlalu dalam mengiimplementasikan teori bahasa. Sedangkan modul-modul lainnya disesuaikan dengan bahasa pemrograman Java.

Sama seperti model kompilator sebelumnya, model ini belum mengimplementasikan metoda atau prosedur atau function, array multi dimensi, teknik object oriented programming, kelas atau modul. Fitur-fitur ini dan fitur fitur tambahan lainnya, dapat dengan sangat mudah ditambahkan dengan memperkaya penganalisis leksikal (scanner), penganalisis sintak (parser), dan modul atau kelas kelas pendukung lainnya.

Acknowledgment

Penulis mengucapkan banyak terima kasih kepada

- Fasilkom UI yang telah memberikan kesempatan kepada penulis memberikan kuliah Teknik Kompilator sejak 1992 hingga saat penulisan dokumen ini.
- Mahasiswa-mahasiswa S1 Fasilkom UI yang pada awal awalnya telah berkontribusi dalam konversi model dari Pascal ke Java dan penyempurnaan pada proses proses berikutnya, dan testing-testing dengan variasi input sehingga mengkonfirmasikan berfungsinya model pemroses bahasa MINUI ini. Mereka adalah A Sasmito Adibowo (1999), Bayu Adianto Prabowo (2000), Dalton E Pelawi (1999), Ilham W. Kurniawan (2002), Jimmy (1999), Ryan Loanda (2002) .
- Dan para asisten yang membantu perkuliahan tersebut antara Hanna, Adila, Jimmy, Carroline, serta semua pihak yang tidak dapat disebutkan satu persatu.

May the Almighty God rewards you all with plenty good rewards, Amin.

BAB 1 PENDAHULUAN

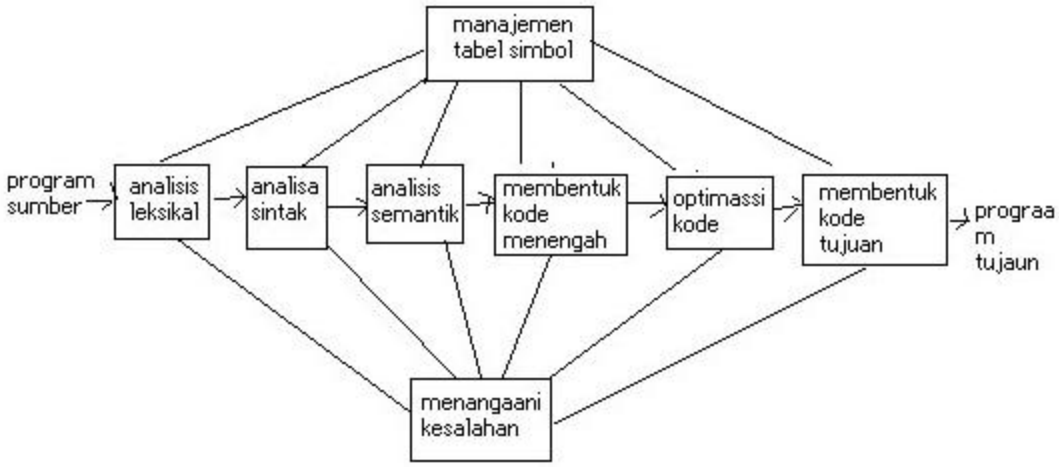
Bagian ini menjelaskan secara ringkas beberapa prinsip dasar teknik-teknik dasar pengembangan sutu kompilator atau pemroses bahasa pemrograman. Informasi lebih rinci dapat dilihat di dokumen yang terdapat pada Daftar Pustaka.

1.1 Kompilator

Kompilator adalah suatu program yang membaca program yang ditulis dengan bahasa tertentu (bahasa sumber) dan menerjemahkannya ke bahasa lain (bahasa tujuan). Contoh bahasa-bahasa sumber adalah bahasa-bahasa pemrograman tradisional seperti Fortran, Pascal, dan Cobol, sedangkan bahasa tujuan pada umumnya merupakan bahasa mesin atau bahasa assembly yang tergantung pada jenis mesinnya.

Proses kompilasi dapat dibagi menjadi dua bagian, yaitu bagian analisis dan sintesis. Dalam proses analisis, program sumber diuraikan menjadi program dalam bahasa tingkat menengah. Proses sintesa akan mengubah representasi menengah tersebut menjadi program dalam bahasa tujuan.

Bagian analisis dan sintesa dibagi-bagi lagi menjadi beberapa tahap. Salah satu bentuk umum dari pentahapan proses kompilasi dapat dilihat pada gambar 1.1.



Gambar 1.1. Tahap-tahap kompilasi

Mulai dari tahap analisis leksikal sampai pembentukan kode menengah adalah bagian analisis. Sedangkan tahap optimisasi kode dan pembentukan kode tujuan adalah bagian sintesa. Dalam implementasinya, tahap-tahap tersebut dapat dikerjakan secara berkelompok. Misalnya analisis leksikal, analisis sintak, analisis semantik, dan pembentukan kode menengah dikerjakan dalam suatu kelompok (pada pass pertama). Sedangkan mulai dari optimisasi kode dapat saja dihilangkan jika ingin membuat suatu kompilator yang relatif sederhana.

Ada dua tahap penting lain yang bekerja secara interaktif dangan ketujuh tahap yang termasuk dalam bagian analisis dan sintesa, yaitu pengaturan tabel simbol dan penanganan kesalahan.

1.2 Analisis Leksikal

Peranan utama dari analisis leksikal adalah membaca karakter masukan dari program sumber dan mengelopokkannya menjadi token-token yang akan dipergunakan untuk proses berikutnya, yaitu analisis sintak. Analisis leksikal juga berperan dalam membuang karakter-karakter tertentu, seperti spasi, tabulasi dan komentar.

Dalam penanganan kesalahan pada saat kompilasi, analisis leksikal digunakan untuk menghubungkan pesan kesalahan dengan program sumbernya. Misalnya, nomor baris letak kesalahan dapat ditampilkan, karenan dalam analisis leksikal, nomor baris ini tetap disimpan.

Ada beberapa istilah penting penting dalam masalah analisis leksikal, seperti "token", "leksim" dan "pola". Suatu himpunan karakter akan membentuk suatu jenis token. Misalnya deretan karakter 3.14 akan membentuk token **bilangan**, sedangkan deretan karakter JUMLAH akan membentuk token *identifier*. Sedangkan **bilangan** 3.14 dan *identifier* JUMLAH pada contoh di atas adalah suatu contoh leksim. Deretan karakter suatu leksim sesuai dengan pola suatu token. Token **bilangan** memiliki pola: deretan karakter yang merupakan gabungan angka dengan karakter 8xyz yang tidak dapat digolongkan dalam token *identifier* maupun token bilangan.

1.3 Analisis Sintak

Analisis sintak lebih sering disebut penguraian (*parsing*). Tujuan utama dari analisis sintak adalah memeriksa apakah urutan token-token yang dihasilkan sesuai dengan tata bahasa dari bahasa yang bersangkutan. Misalnya bahasa C mengenal kalimat: jumlah++; yang berarti menaikkan harga variabel jumlah dengan angka satu. Tetapi kalimat di atas akan salah jika dikompilasi dengan kompilator bahasa Pascal, karena tidak sesuai dengan tata bahasa Pascal.

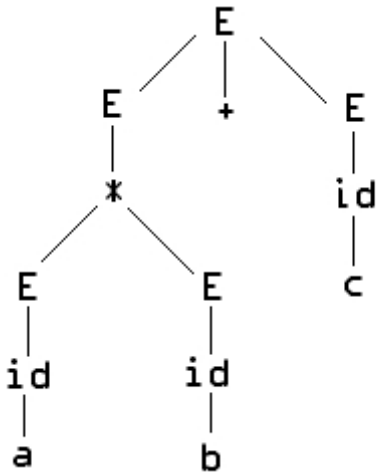
Dalam analisis sintak, tata bahasa yang digunakan untuk mendefinisikan aturan sintak suatu bahasa disebut tata bahasa bebas konteks (*Context Free Grammar*). Tata bahasa ini memiliki empat komponen penting yaitu himpunan simbol terminal, himpunan non-terminal, himpunan produksi dan simbol awal. Dalam bahasa pemrograman, yang disebut terminal adalah token. Contoh terminal adalah token. Contoh token misalnya kata kunci (keyword) **if**, **while**, dan *identifier* serta bilangan. Sedangkan non-terminal merupakan variabel-variabel sintak yang menyatakan himpunan terminal maupun non-terminal. Dalam proses parsing terjadi proses penggantian suatu non terminal dengan sederetan himpunan non terminal dan terminal yang berada dalam sisikanan produksnya. Proses ini disebut sebagai *derivasi.* Contohnya non-terminal *if_stmt* merupakan himpunan terminal **if, then, else**, dan non-terminal *expr* dan *stmt*, yang membentuk aturan produksi : *if_stmt* → **if** *expr* **then** *stmt* **else** *stmt*. Dari semua simbol non-terminal yang digunakan, ada satu simbol yang bertindak sebagai simbol awal, yaitu simbol yang pertama kali diderivasi. Aturan produksi menggambarkan bagaimana kombinasi non-terminal dan terminal yang benar menurut tata bahasa yanbg bersangkutan.

Dalam proses penguraian, token-token yang dihasilkan dalam analisis leksikal dibentuk menjadi pohon urai (*parse tree*). Dalam implementasi kompilator, pohon urai ini belum tentu berbentuk pohon yang sebenarnya, misalnya dengan menggunakan fasilitas *pointer*.
Pohon urai merupakan hasil derivasi dari aturan –aturan produksi. Contohnya tata bahasa berikut ini :

E → E+E | E*E | (E) | -E | **id** (1.1)

Dengan masukan **a*b+c** dapat dibentuk satu contoh pohon urai seperti gambar 1.2. Pohon urai tersebut didapat dengan menderivasi aturan produksi di atas sebagai berikut :

E ➔ E+E ➔ E*E+E ➔ id*E+E ➔ id*id+E ➔ id*id+id



Gambar 1.2. Pohon Urai untuk ekspresi a*b+c

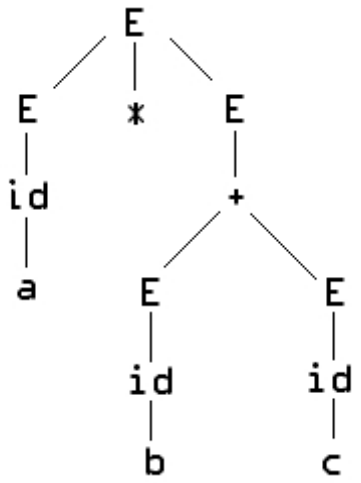
Ada dua jenis derivasi, yaitu derivasi terkiri (Left Most Derivation) dan derivasi terkanan (Right Most Derivation). Derivasi terkiri akan menderivasi suatu aturan produksi mulai dari non-terminal yang paling kiri. Sedangkan derivasi terkanan akan menderivasi suatu aturan produksi mulai dari non-terminal yang paling kanan. Contoh derivasi **a*b+c** di atas adalah derivasi terkiri. Derivasi terkanan dapat dilakukan sebagai berikut :

E ➔ E+E ➔ E+id ➔ E*E+id ➔ E*id+id ➔ id*id+id

Jika proses derivasi aturan-aturan produksi suatu tata bahasa terhadap suatu masukan menghasilkan lebih dari satu pohon urai maka tata bahasa tersebut dikatakan rancu (*ambiguous*).

Tata bahasa (1.1) di atas termasuk rancu, karena selain dapat membentuk pohon urai pada gambar 1.2 juga adapat membentuk pohon urai lain yang dapat dilihat pada gambar 1.3. Pohon urai pada gambar 1.3 berasal dari derivasi sebagai berikut :

E ➔ E*E ➔ id*E ➔ id*E+E ➔ id*id+E ➔ id*id+id



Gambar 1.3. Pohon urai kedua untuk ekspresi

Tata bahasa yang rancu dapat diperbaiki dengan cara menghilangkan rekursi kiri dan melakukan faktorisasi kiri. Rekursi kiri adalah aturan produksi dengan simbol non-terminal pada sisi kiri produksi sama dengan simbol awal sisi kanan produksi, contoh E ➔ E+E. Rekursi kiri dapat menyebabkan pengulangan tak hingga. Faktorisasi kiri adalah pengubahan aturan-aturan produksi yang memiliki simbol awal sisi kanan produksi yang sama.

Contohnya E ➔ E+E dan E ➔E*E.

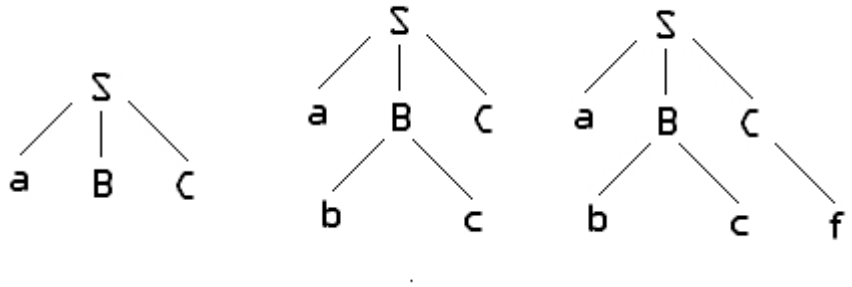
Faktorisasi kiri perlu dilakukan untuk memastikan pilihan aturan produksi mana yang akan dipakai.

Tidak rancu dan tidak memiliki rekursi kiri merupakan ciri-ciri dari tata bahasa yang disebut tata bahasa LL(1). Kedua huruf L berasal dari left (kiri). Huruf L pertama berarti tiap karakter masukan diproses satu per satu mulai dari kiri ke kanan. Huruf L kedua berarti pemakain derivasi derivasi terkiri (left most derivation). Sedangkan angka 1 berarti pada setiap langkah penguraian hanya membutuhkan satu token.

Penguraian tata bahasa LL(1) menggunakan teknik top-down. Dalam teknik tersebut dilakukan proses derivasi terkiri terhadap suatu masukan . Dalam proses derivasi tersebut dilakukan pembentukan pohon urai yang dimulai dari akar dan membentuk simpul-simpul baru dengan urutan preorder. Contohnya tata bahasa berikut

S ➔ aBC
B ➔bc | d
C ➔ e | f

dengan masukan abcf. Proses penguraian top-down dapat dilihat pada gambar 1.4.



Gambar 1.4. Tahap-tahap pengurai top-down

Selain pengurai top-down, ada satu jenis pengurai yang disebut pengurai bottom-up, yaitu proses pembentukan pohonnya merupakan kebalikan dari pengurai top-down. Pengurai top-down lebih mudah digunakan untuk pembentukan pengurai yang efisien. Pengurai bottom-up dapat menangani tata bahasa yang lebih besar, tetapi implementasi pengurai ini lebih sukar dibandingkan pengurai top-down.

1.4 Analisis Semantik

Analisis semantik berperan dalam memeriksa kesalahan-kesalahan yang bersifat semantik. Salah satu peranan analisis semantik yang penting adalah pemeriksaan tipe variabel. Contohnya operator * hanya digunakan untuk operand dengan tipe integer ataupun real. Sedangkan operator and, or, digunakan hanya untuk operand dengan dengan tipe boolean.

Peranan lain dari analisis semantik adalah memeriksa keunikan suatu nama. Misalnya dalam Pascal, nama variabel global tidak boleh sama dengan prosedur atau nama fungsi. Dalam bahasa C, jika suatu nama konstanta didefinisikan lebih dari satu kali, maka akan diperiksa kesamaan nilai kedua konstanta.

Analisis semantik dapat dilakukan dengan menggunakan salah satu dari dua bentuk notasi, yaitu Definisi Berdasarkan Sintak (DBS) dan Skema Translasi.

Definisi Berdasarkan Sintak (DBS) merupakan gabungan tata bahasa dengan himpunan aturan semantik yang akan menentukan struktur sintak dari suatu masukan. Aturan semantik digunakan untuk menghitung atribut, misalnya tipe atau nilai konstanta, yang berkaitan dengan simbol dalam aturan produksi. Contohnya, untuk mengubah ekspresi matematika menjadi bentuk prefiks dapat dilihat pada DBS yang digambarkan pada gambar.5.

Produksi			Aturan Semantik
Expr	➔	expr + term	expr.t = + | expr.t | term.t
expr	➔	expr - term	expr.t = - | expr.t | term.t
expr	➔	term	expr.t = term.t
term	➔	0	term.t = 0
term	➔	1	term.t = 1
Dst			dst

Gambar 1.5. DBS untuk mengubah ekspresi ke dalam bentuk prefik

Evaluasi aturan semantik dilakukan dengan pelacakan terdalam (depth first traversal). Skema Translasi adalah suatu tata bahasa bebas konteks dengan penyisipan suatu bagian program, yang disebut aksi semantik, pada sisi kanan produksi. Secara prinsip, skema translasi hampir sama dengan DBS. Perbedaannya terletak pada urutan evaluasi aturan semantik yang ada pada skema translasi dinyatakan dengan jelas, sedangkan pada DBS tidak. Contoh bentuk skema translasi dapat dilihat pada gambar 1.6.

expr	➔	{print ('+')} expr1 + term
expr	➔	{print ('-')} expr1 - term
expr	➔	term
term	➔	0 {print ('0')}
term	➔	1 {print ('1')}
dst.		

Gambar 1.6. Skema Translasi untuk mengubah ekspresi ke dalam bentuk prefik

Pada gambar 1.6. terlihat bahwa posisi aksi-aksi semantik diketahui dengan jelas.

1.5 Pembentukan Kode Menengah

Pembentukan kode menengah merupakan tahap lanjutan setelah analisis semantik. Hasil pembentukan kode menengah dapat dianggap sebagai program dengan instruksi-instruksi bahasa mesin abstrak. Bentuk representasi kode menengah harus mudah pembuatannya dan mudah diterjemahkan dalam bahasa tujuan. Salah satu bentuk representasi kode menengah adalah kode tiga alamat. Misalnya, suatu kalimat matematik a := b * c + d memiliki bentuk kode tiga alamat sebagai berikut :

t1 := b * c t2 := t1 + d a := t2

Representasi kode tiga alamat memiliki bentuk yang menyerupai kode dalam bahasa Assembly, sehingga memudahkan proses penterjemahannya, jika bahasa tujuan adalah bahasa Assembly. Bentuk kode tiga alamat di atas memiliki karakteristik: mengandung paling banyak

tiga operand dan dua operator, serta memiliki variabel sementara. Bentuk lain dari representasi kode menengah adalah dalam bentuk representasi grafik, seperti pohon maupun graf.
Salah satu manfaat pembentukan kode menengah adalah ia berfungsi sebagai input untuk proses optimisasi. Salah satu contoh adalah jika terdapat sub ekspresi yang sama muncul dalam program pemakai, maka kompilator dengan fasilitas optimisasi tidak akan mengeksekusi ekspresi itu berulang kali, tapi cukup sekali. Namun karena model saat ini belum menerapkan proses optimisasi, maka proses pembentukan kode menengah belum diterapkan. Ia akan diterapkan pada saat kami mengembangkan modul optimisasi.

1.6 Optimisasi Kode

Tujuan dari optimisasi kode adalah meningkatkan performansi dari program tujuan, seperti kecepatan eksekusi program dan efisiensi pemakaian memori. Optimisasi kode dapat dilakukan pada instruksi berikut : x := a * b - c + a * b. Perhitungan a*b dapat dilakukan satu kali saja sehingga hasilnya adalah dua instruksi :

```
t := a * b          x := t - c + t
```

Optimisasi kode dapat dilakukan terhadap kode menengah pada saat pembentukan kode menengah dan juga terhadap kode tujuan pada saat pembentukan kode tujuan. Berbeda dengan optimisasi pada kode menengah, optimisasi yang dilakukan pada kode tujuan tergantung pada mesin tujuan yang digunakan.

1.7 Pembentukan Kode Tujuan

Dalam tahap ini, input bisa kode menengah atau terjemahan langsung dari tata bahasa dengan fasilitas aksi semantik. Bagian dari tatabahasa langsung diterjemahkan kedalam kode sasaran yang bisa berupa instruksi bahasa mesin atau bahasa assembly. Pada tahap ini, setiap variabel telah ditentukan lokasi memorinya. Kemudian, setiap instruksi yang dibuat dalam kode menengah diterjemahkan satu per satu menjadi deretan instruksi mesin atau assembly.

Contohnya kalimat berikut : c := a * b + 5

Melalui tahap pembentukan kode tujuan, akan dihasilkan kode umum sebagai berikut :

```
MOV R1,a
MOV R2,b
MUL R1,R2
ADD R1,#5
MOV c,R1
```

Namun untuk dapat disimulasikan, kami mendefinisikan kode sasaran simulasi seperti di lampiran. Simulasi dilakukan dengan mengeksekusi barisan perintah dalam Java yang sesuai dengan maksud yang didefinisikan dalam kode sasaran.

1.8 Pembentukan Tabel Simbol

Tabel simbol merupakan suatu struktur data yang menyimpan informasi atau attribute nama-nama simbol, seperti nama variabel, konstanta, prosedur, fungsi, dan informasi atribut-atribut tiap simbol, seperti tipe, parameter, nilai konstanta, dan lokasi memorinya. Seluruh informasi ini disimpan dalam satu unit yang disebut sebagai entri tabel symbol atau sering disebut dalam buku referensi sebagai bucket.

Informasi-informasi yang terdapat dalam tabel simbol dapat digunakan untuk pemeriksaan kesalahan pada saat tahap analisis, seperti pemeriksaan nama dan tipe variabel. Selain itu, informasi lokasi memori dipergunakan dalam pembentukan kode tujuan.

Struktur data tabel simbol dapat berbentuk list linier maupun tabel hash. Bentuk list linier mudah untuk diimplementasikan namun ia tak efisien untuk mencapai kinerja yang baik. Sebagai alternative, banyak para pengembang yang memakai tabel Hash. Ciri utama tabel ini adalah ia mempunyai satu list atau array utama yang isinya menunjuk (melink) ke enti tabel symbol. Dan setiap enti tabel symbol dapat menunjuk ke entri lainnya. Penempatan di index ke berapa array itu suatu entri akan berada, ditentukan oleh fungsi Hash.

1.9 Penanganan Kesalahan

Penanganan kesalahan mencakup pendeteksian kesalahan, pemberian pesan kesalahan dan perbaikan kesalahan. Prosedur untuk menangani kesalahan dibuat dalam setiap tahap kompilasi yang membutuhkannya. Misalnya dalam analisis leksikal, dibuat prosedur untuk menangani kesalahan-kesalahan leksikal. Demikian pula untuk analisis sintak dan analisis semantik. Sedangkan kesalahan pada waktu program berjalan (run time error) juga harus dapat ditangani dengan memasukkan prosedur penanganan kesalahan dalam program tujuan.

BAB 2 Perancangan Pemroses Bahasa MINUI

2.1 Pendahuluan

Model Bahasa mini diperkenalkan pada mata kuliah Teknik Kompilator di Program Studi Ilmu Komputer Universitas Indonesia (Prosilkom UI) sejak tahun 1990. Sebagian besar bahan diperoleh dari hasil project penulis sendiri saat mengambil perkuliahan Language Processor di University of Toronto, Canada tahun 1988. Di UI, pertama kali bahasa ini diberikan nama IKI336 yang merupakan kode mata kuliah tersebut pada saat itu. Kemudian nama bahasa itu berubah sesuai dengan perbahan nama kode mata kuliah tersebut yakni IKI40800. Bahasa ini dipakai juga dalam Model Kompilator berbasis Pascal yang sudah terdaftar sebelumnya di Dirjen HAKI RI. Agar lebih mudah diingat dan tak tergantung dengan nama koda mata kuliah, mulai saat ini bahasa tersebut diberi nama MINUI. Nama ini mengandung makna MINI yang menggambarkan bahwa fitur bahasa tersebut kecil dan dikembangkan di perkuliahan Fasilkom UI.

Karakteristik dari tata bahasa MINUI (tata bahasa secara lengkap dapat dilihat pada lampiran 1) dapat dijabarkan sebagai berikut :

- merupakan tata bahasa bebas kontek dengan simbol awal program, memiliki himpunan terminal yaitu identifier dan semua yang berada dalam tanda kutip, dan simbol-simbol lain yang merupakan himpunan non terminal, kecuali yang diawali 'C' atau 'R' dan diikuti angka, serta memiliki 85 aturan produksi.
- termasuk dalam bahasa LL(1), dimana karakteristik dari tata bahasa ini adalah tidak bersifat rancu dan tidak memiliki rekursi kiri.
- berbentuk skema translasi, karena di dalam penulisan tata bahasanya disisipkan bagian-bagian program berupa aksi-aksi semantik.

Aksi-aksi semantik ini adalah simbol-simbol yang diawali dengan huruf 'C' atau 'R' diikuti oleh angka. Yang diawali huruf 'C' adalah aksi semantik untuk melakukan analisis semantik, sedangkan yang diawali huruf 'R' digunakan untuk membentuk kode tujuan.

Kata-kata kunci (keywords)

Beberapa kata-kata kunci yang dikenal dalam tata bahasa MINUI adalah **if, then, else, end if, repeat, until, loop. end loop, exit, var, proc, func, put, get, skip**. Tipe variabel yang dikenal ada dua, yaitu **integer** dan **boolean**. Nilai **integer** memiliki interval di antara -32767 sampai +32767, sedangkan nilai **boolean** berkisar antara **true** atau **false**. Tanda **;** digunakan untuk menyatakan akhir deklarasi variable. Untuk menyatakan awal dan akhir dari suatu scope digunakan tanda **{** dan **}**.

Operator Aritmetika

Operator -operator aritmetika dan presedensinya dapat dilihat pada tabel 2.1

Operator	Presedensi	Kategori
+, -, ~(NOT)	1 (tertinggi)	unari
*, /, &(AND)	2	Pengali
+, -, |(OR)	3	Penambah
=, #, <, , <=, =	4	Relasi

Tabel 2.1 Presedensi Operator

Contoh program dalam bahasa MINUI dapat dilihat pada gambar 2.1 berikut.

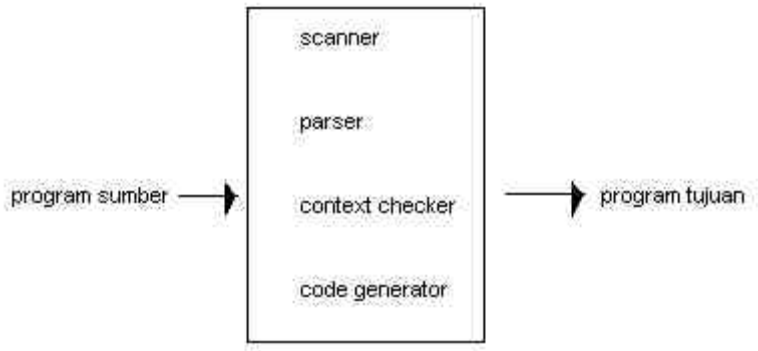
```
{
   var A     : integer
   var B[10] : integer;


  A := 0
  repeat
     B[A] := A*2
     A := A+1
   until A = 10
   put B[A]
}
```

Gambar 2.1. Program dalam bahasa MINUI

2.2 Struktur Kompilator berbasis Java

Tahap-tahap kompilasi yang dilakukan oleh kompilator MINUI dimulai dari analisis leksikal (scanner), analisis sintaks (parser), analisis semantik (context checker) kemudian langsung dibentuk kode tujuan. Keseluruhan proses kompilasi tersebut dilakukan pada satu pass.

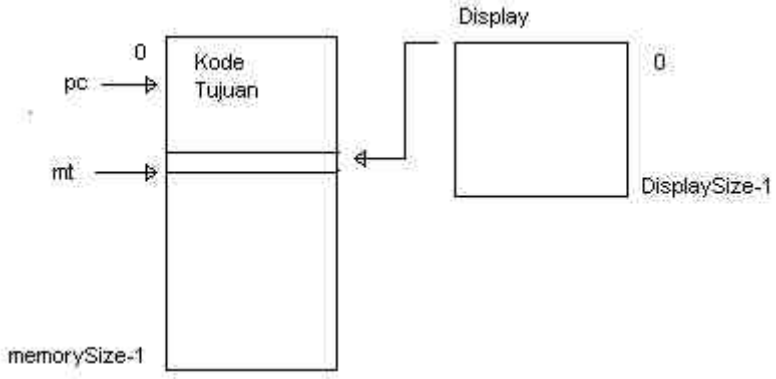


Gambar 3.1 Struktur kompilator MINUI

Model kompilator bahasa MINUI ini merupakan suatu simulasi. Program sumber merupakan program dalam bahasa MINUI yang dibuat dalam bentuk file teks. Sedangkan program tujuan berada dalam suatu variable "memori" berbentuk array satu dimensi. Memori tersebut digunakan untuk menyimpan kode tujuan, alokasi variabel, dan memori *run time*. Sekilas variable Memori ini menggambarkan bentuk memori sebenarnya suatu komputer.

Kode tujuan yang berada dalam memori dapat dieksekusi. Instruksi-instruksi kode tujuan dilakukan terhadap memori, dengan menggunakan instruksi dalam bahasa pemrograman Java. Deskripsi mesin dan instruksi-instruksi kode tujuan MINUI secara lengkap dapat dilihat pada lampiran.

Selain memori, terdapat "register" display yang merupakan pointer ke lokasi variabel untuk setiap tingkat leksikal (lexic level). "Register" display ini juga merupakan array satu dimensi. Untuk lebih jelasnya, bentuk memori dan register display pada saat run time dapat dilihat pada gambar 3.2



Gambar 3.2 Deskripsi mesin

2.3 Scanner (Lexical analyzer) dengan Java Lex

Untuk melakukan analisis leksikal, di dalam program kompilator MINUI terdapat pengelompokan jenis-jenis karakter dan token. Untuk karakter, bentuk pengelompokan karakter adalah sebagai berikut:

- karakter yang diabaikan (ignore), yaitu spasi, tabulasi
- karakter ilegal
- karakter komentar yaitu karakter prosentase yakni %
- huruf dan digit
- awal teks, yaitu "
- karakter spesial, yaitu karakter-karakter yang dipakai dalam tata bahasa selain huruf dan digit.
- karakter akhir baris

- karakter akhir file

Karakter-karakter dikelompokkan menjadi token gabungan/komposit, token spesial, dan token keyword. Token gabungan misalnya Tidentifier, Ttext, untuk nama identifier dan teks. Token spesial misalnya Tcolon untuk tanda ';', Tcomma untuk ',' dst. Token keyword misalnya Tif untuk 'if', Tvar untuk 'var', dst.

Yang dilakukan oleh program scanner adalah:

- Melakukan pembacaan karakter per karakter pada berkas masukan.
- Mengklasifikasikan karakter, apakah spesial, huruf, atau lainnya. Jika karakter tersebut tidak termasuk dalam kelompok manapun, maka prosese kompilasi akan berhenti dan memberi pesan kesalahan dan baris posisi kesalahan.
- Mengklasifikasikan karakter atau beberapa karakter menjadi suatu token.
- Token yang didapat akan diproses pada tahap berikutnya, yaitu analisis sintaks oleh program parser.

Program scanner tersebut akan dipanggil oleh parser setiap kali suatu token yang dihasilkannya telah dproses oleh parser.

Berbeda dengan model kompilator berbasis Pascal, dalam model kali ini pendefinisian token yang akan diambil dan diserahkan ke penganalisis sintak lebih mudah. Kita tak perlu secara rinci membaca input satu karakter per karakter dan mencoba mengembangkan sendiri pengenalan pola token. Namun pola-pola pembentukan token dituliskan dalam notasi yang sederhana dan mirip dengan yang berlaku dalam teori bahasa yakni ekspresi regular.

Gambar berikut memuat contoh bagaimana pengenalan token identifier dapat dituliskan dalam notasi JavaLex. Pertama kumpulan karakter huruf dikumpulkan dalam symbol ALPHA, dan angka-angka dengan symbol DIGIT. Masing-masing menggunakan representasi ekspresi regular. Kemudian token IDENT diberikan ke parser.

```
%state COMMENT

ALPHA=[A-Za-z]
DIGIT=[0-9]
NONNEWLINE_WHITE_SPACE_CHAR=[\ \t\b\012]
WHITE_SPACE_CHAR=[\n\ \t\b\012]
STRING_TEXT=(\\\"|[^\n\"]|\\{WHITE_SPACE_CHAR}+\\)*
COMMENT_TEXT=.*
EOL=\r\n

%%

<YYINITIAL> ({ALPHA}|_)({ALPHA}|{DIGIT}|_)*
{
return (new Yytoken(sym.IDENT,yytext(),yyline,yychar,yychar + yytext().length()));
}
```

Rincian definisi penganalisis leksikal model ini dapat dilihat pada file scanner.lex. sedangkan linformasi lengkap bagaimana pemakaian JavaLex dapat dilihat pada dokumen yang diacu di daftar pustaka.

2.4 Parser (sintax analyzer) dengan Java Cup

Sebelumnya telah dijelaskan bahwa tatabahasa MINUI adalah tata bahasa LL(1). Cara penguraian yang dipakai adalah *top-down*, karena itu teknik derivasi yang dilakukan adalah derivasi terkiri.

Berbeda dengan Model berbasis Pascal, dalam model yang baru ini pembentukan table pengurai/parsing yang melibatkan pembentukan himpunan First dan Follow tidak perlu dilakukan oleh perancang kompilator. Pendefinisian cukup diatur di Javalex dan dikomunikasikan dengan JavaCup. Perancang cukup memfokuskan penulisan tatabasa yang dipakai dengan notasi tatabahasa dalam JavaCup.

Gambar berikut menjelaskan bagaimana sebagian dari bagian tatabahasa tersebut dinotasikan dalam JavaCup. Rincian seluruh pendefinisian seluruh komponen parser dapat dilihat di file parser.cup

Di sini gambar contoh deklarasi variable dalam java Cup.

```
import java_cup.runtime.*;
/*
*code initialiasi di sini
*
*/

terminal TYPE
BOOL, INT ;
terminal TOKN
NUMCONST, STRCONST, IDENT, STRUNCLD ;

nonterminal
declarations, moreDeclarations, declaration,  type, optArrayBound,;

declarations      ::= declaration moreDeclarations
                |  error moreDeclarations;
moreDeclarations  ::=
                |  declaration moreDeclarations ;

declaration       ::= VAR _current IDENT C3 C4 optArrayBound AS type C5 C11 C7

type              ::= INT C9
                |  BOOL C10;
optArrayBound     ::= C18 R37
                  |  LBRACKET simpleExpression C12 C11 R39 RBRACKET C19;

C3                ::= {: parser.context.C(3); :} ;
C5                ::= {: parser.context.C(5); :} ;
R37               ::= {: parser.generate.R(37); :} ;
```

2.5 Contect checker (semantic analyzer)

Analisis semantik dilakukan dengan menyisipkan aksi-aksi semantik pada tata bahasa dalam notasi JavaCup. Program contect checker akan dipanggil oleh parser pada saat parser menemukan token berawalan dalam bentuk 'Cx' (x = bilangan). Fungsi-fungsi yang dilakukan oleh program contect checker antara lain adalah :

- Manajemen tabel simbol (akan dijelaskan pada sub bab berikut).
- Pemeriksaan deklarasi variabel ganda.
- Pemeriksaan pemakaian variabel, meliputi :
    - pemeriksaan apakah variabel telah dideklarasikan
    - pemeriksaan kesesuaian tipe variabel yang digunakan, baik pada kalimat assignment maupun kalimat perbandingan.
    - pemeriksaan jenis variabel.
- Pemeriksaan pemakaian fungsi dan prosedur, termasuk kesesuaian jumlah dan tipe parameter.

Jika suatu kesalahan ditemukan, maka pesan kesalahan akan ditampilkan dengan nomor barisnya, dan proses kompilasi akan berhenti.

2.6 Manajemen table simbol

Struktur data tabel simbol yang dipergunakan dalam kompilasi ini adalah dalam bentuk tabel hash. Tabel hash ini berbentuk array berukuran hash_size, dimana hash_size merupakan bilangan prima. Tiap entri dari array menunjuk ke suatu linked-list linier, dimana setiap node dari linked-list adalah suatu record yang menyimpan informasi satu identifier.
Setiap record dari suatu identifier berisi informasi:

- nama identifier
- order number (nomor urutan identifier pada suatu scope)
- lexic level (tingkat leksikal)
- tipe identifier (integer, boolean)
- jenis identifier (skalar, array, prosedur, atau fungsi)
- pointer ke node berikutnya.
- jenis identifier (skalar, array, prosedur, atau fungsi)
- pointer ke node berikutnya

Ada beberapa prosedur utama yang digunakan untuk manajemen tabel simbol :

Menambah entri

Posisi suatu identifier di dalam tabel simbol berada pada indeks array yang dihitung dengan cara:

(nilai ASCII nama identifier) *mod* hash_size

Sehingga entri identifier a berada pada indeks 65 (nilai ASCII a = 65), jika nilai hash_size = 211. Jika pada indeks ke-I, sudah ada list yang terbentuk ,maka perlu diperiksa apakah ada nama identifier yang sama pada tingkat leksikal yang sama. Jika tidak ada, entri baru sebaiknya disisipkan dikepala list. Alasannya, nama identifier yang terakhir dideklarasi akan lebih dulu ditemukan. Jadi, jika ada dua nama identier yang sama dalam dua tingkat leksikal, l1 dan l2, dimana l1l2, maka pada l2, identifier yang dipergunakan adalah identifier yang dideklarasi di l2 (yang pertama ditemukan pada waktu pencarian entri pada tabel simbol).

Menghapus entri

Entri suatu identifier pada suatu tingkat leksikal dihapus pada saat keluar dari scope pada tingkat leksikal tersebut.
Penghapusan dilakukan dengan menelusuri setiap linked list yang ditunjuk oleh pointer pada tiap tabel simbol dan menghapus node yang memiliki tingkat leksikal tersebut.

Mencari entri

Prosedur ini dipakai pada saat menambah entri, pemeriksaan nama, tipe dan jenis suatu identifier, pemeriksaan nama prosedur dan fungsi, pemeriksaan nama dan jumlah parameter pada prosedur dan fungsi. Dalam pencarian entri, terlebih dahulu dihitung indeks tabel simbol untuk nama identifier yang dicari. Kemudian dilakukan pencarian pada linked list yang ditunjuk oleh pointer pada indeks tersebut.

Mencetak tabel simbol

Fasilitas ini merupakan pilihan bagi pemakai kompilator. Isi tabel simbol dapat dicetak pada saat deklarasi variabel pada suatu tingkat leksikal selesai.

2.7 Pembentukan kode tujuan (code generator)

Telah dijelaskan pada suatu Bab sebelumnya , bahwa kode tujuan yang dihasilkan oleh kompilator ini adalah kode mesin MINUI yang dapat dilihat pada lampiran. Kode tujuan dimasukkan ke dalam "memory" yang diinisialisasi terlebih dahulu sebelum proses kompilasi dimulai.

Program code generator ini dipanggil oleh program parser pada saat ditemukan aksi semantic yang disisipkan pada parser di JavaCup. Dalam contoh di atas, perhatikan nonterminal R37

dan R39 merupakan aksi semantic yang akan membentuk kode sasaran berkaian dengan jenis identifier apakah ia berbentuk array atau bukan. Jenis-jenis instruksi yang dibentuk oleh code generator untuk masing-masing x dapat dilihat pada lampiran.

Beberapa hal penting yang dibutuhkan dalam program code generator ini adalah :

- Stack untuk menyimpan informasi jumlah variabel dalam setiap tingkat leksikal, dibutuhkan untuk melakukan dealokasi variabel pada saat keluar dari suatu scope.
- Stack untuk menyimpan alamat lompatan bersyarat (untuk if). Lompatan tidak bersyarat (biasanya dipakai sebelum deklarasi suatu prosedur/fungsi), dan lompatan mundur untuk loop. Stack diperlukan karena if dan loop bisa bersarang (nested).
- Prosedur untuk cetak tekas yang hanya disisipkan dalam kode tujuan jika ada kalimat put text.
- Prosedur untuk pemeriksaan pembagian dengan nol yang hanya disisipkan dalam kode tujuan jika ada operasi aritmatika yang menggunakan '/'.

Kode tujuan yang dihasilkan dapat dieksekusi oleh suatu program tambahan yang akan menjalankan kode tujuan dalam instruksi Java. Rincian kode kode ini dan bagaimana kaitannya dengan memori tempat eksekusi akan dijelaskan pada bagian berikut.

2.8 Definisi kode sasaran (mesin)

Mesin yang dipakai mempunyai komponen-komponen yakni memori utama dan register display yang didefinsikan sebagai array dalam bahasa Java. Kemudian variable pc yang berfungsi sebagai program counter dan variable mt (memory top) yang berfungsi sebagai puncak stack memori yang kosong berikutnya untuk dipakai sebagai tempat sementara proses komputasi.

Memory utama yang terdiri dari word sebanyak memorySize dengan indeks dari 0 sampai memorySize-1. Suatu bilangan yang mewakili suatu word di dalam memory disebut dengan address. Masing-masing word berukuran 16 bit (karena integer biasa). Bentuk bit tersebut di dalam memory dapat diartikan sebagai berikut:

- Suatu nilai integer antara -32767 sampai +32767.
- Suatu nilai boolean. False direpresentasikan oleh integer 0, sedangkan true direpresentasikan oleh integer 1.
- Suatu nilai karakter (satu karakter per satu word).
- Nilai data tak terdefinisi direpresentasikan oleh 1000000000000000.
- Kode operasi. Operasi dan kodenya akan diberikan setelah ini.
- Suatu address.

pc (program counter) memuat address dari pada instruksi berikutnya yang akan dieksekusi
.
mt (memory top) memuat address dari pada word berikutnya yang kosong.

Register display yang berindek dari 0 sampai displaySize-1. Register ini dipakai agar variabel yang berada pada level lexic tertentu dapat ditemukan addressnya.

Pada awal eksekusi, hal-hal berikut akan terjadi

- Instruksi dari suatu program akan menempati memory yang bersambungan, dimulai pada word 0.
- Register mt memuat address dari pada word pertama tepat setelah word milik instruksi-2.
- Sisa memory memuat sampah.
- Display memuat sampah.
- pc memuat address dari pada instruksi pertama yang akan dieksekusi.

Pada tempat setelah instruksi-instruksi, space dialokasikan untuk variabel-variabel dan dapat diakses melalui display. Space yang dibutuhkan untuk hasil sementara evaluasi suatu ekspresi berada pada puncak memory yang sudah dipakai (dinamakan Stack Evaluasi).

Suatu instruksi dapat menempati satu, dua, atau tiga word memory. Word pertama memuat kode operasi yang menentukan operasi apa yang akan dilakukan. Word kedua dan ketiga (jika ada) memuat operand-operandnya. Pada penjelasan berikut, push, pop, dan top mengacu pada memory yang dianggap sebagai satu stack yang pointer topnya adalah mt. Variabel a, v, n, x, dan y adalah bersifat lokal.

0	NAME LL ON	Push(display[LL]+ON)

1	LOAD	a:=top; pop;
		jika memory[a]=undefined maka error;
		push(memory[a]);

2	STORE	v:=top; pop;
		a:=top; pop;
		memory[a]:=v

3	PUSH V	Push(v)
4	PUSHMT	Push(mt)
5	SETD LL	Display[LL]:=top; pop;
6	POP	n:=top; pop; popn
7	DUP	n:=top; pop; v:=top; pop; pushn(v)
8	BR	a:=top; pop; goto a
9	BF	a:=top; pop; v:=top; pop; jika Bv maka goto a
10	ADD	Untuk operasi 10-16
11	SUB	y:=top; pop;
12	MUL	x:=top; pop;

13	DIVI	push(x op y)
14	EQ	jika overflow atau division by zero maka error
15	LT	
16	ORI	
17	FLIP	x:=top; pop; y:=top; pop; push(x); push(y);
18	READC	Satu karakter input dibaca dan dipush
19	PRINTC	Catak top sebagai karakter; pop;
20	READI	Baca integer n; push(n);
21	PRINTI	Catak top sebagai integer; pop;
22	HALT	Halt (berhenti)

Rincian bagaimana mesin simulasi ini dituliskan dalam Java dapat kita lihat di file Hmachine.java dan machine.java

BAB 3 Struktur Program Pemroses

Model pemroses bahasa ini terdiri dari sepuluh file, delapan file harus dibentuk oleh pengembang yakni enam ditulis dalam Java satu ditulis dalam notasi penganalisis leksikal JavaLex dan satu ditulis dalam notasi tatabahasa JavaCup. Sedangkan dua lagi yakni JavaLex dan JavaCup sudah tersedia bebas di internet. File-file ini adalah

- Scanner.lex. File ini mendefinisikan aturan pembentukan suatu token dan memberikan token tersebut ke parser yang didefinisikan di file Parser.Cup. Secara otomatis karakter-karakter yang tak terpakai akan dibuang/dilewatkan.

- Parser.cup. File ini berisi beberapa metoda pendukung dan tatabahasa ini model. Penulisan tatabahasa memakai notasi yang merepresentasikan produksi yang dapat diganti oleh kumpulan string terminal dan non terminal. Presedensi operator dapat didefinisikan di sini sesuai dengan kehendak pengembang.

- Bucket.java dan Hash.java. Bucke.java berisi definisi kelas entry suatu table symbol dan metoda metoda (prosedur) yang terkait dengan operasi pengisan, perubahan dan pengambilan status/attribute suatu entry. Attribute ini berguna untuk pemeriksaan kebenaran pemakaian identifier yang dilakukan oleh penganalisis semantic (semantic analysis atau context checker).

- Hash.java mendefinisikan kelas yang memuat metoda metoda pendefinisian fungsi Hash dan pengelolaan suatu entry table symbol. Pengelolaan ini terdiri antara lain mencari, memasukkan, menghapus entry table symbol.

- Context.java. File ini berupa kelas yang berfungsi sebagai aksi semantic. Beberapa aksi yang ditangani antara lain pemeriksaan apakah suatu identifier sudah berada dalam table symbol, jika belum ia akan memasukkannya ke dalam table. Pemeriksaan type identifier agar sinkron sesuai dengan operator yang terlibat dalam suatu ekspresi. Dalam melakukan aksi semantic ini, context checker akan berinteraksi dengan kelas penanganan table symbol yakni kelas yang terdefinisi dalam Bucket.java dan Hash.java

- Generate.java. File ini berupa kelas yang berfungsi sebagai pengelola pembentukan kode akhir (Code Generator). Ia akan berinteraksi dengan table symbol untuk memperoleh address suatu identifier. Informasi yang terkait dengan identifier ini berguna untuk membantu pengalokasian space memori model kita. Selain itu juga dipakai untuk membentuk instruksi yang terkait, misalnya instruksi-instruksi yang terkait dengan ekspresi. Kelas ini juga akan berinteraksi dengan File Hmachine dan Machinve.java sewaktu akan mengisikan memori dengan kode kode mesin simulasi.

- Hmachine.java dan Machine.java. Kedua file ini berisi kelas yang mendefinisikan mesin kompilator. Di sini instruksi-instruksi mesin didefinisikan dalam bahasa Java. Di

sini juga terdapat metoda untuk mengeksekusi instruksi yang sudah berada dalam memori.

- File pembentuk penganalisis leksikal, yakni JavaLex. File ini secara lengkap berisi definisi-definisi pengenalan pola token. Di sini juga didefinisikan karakter-karakter apa saja yang tidak dianggap penting dalam program dan karakter ini akan dibiarkan tidak diproses untuk pembentukan token. Identifikasi pengenalan baris program input dilakukan disini dan disimpan dalam variable yang dapat dipakai oleh kelas kelas lain guna melokalisasi tempat dimana terjadi kesalahan pada program input.

- File pembentuk parser, yakni JavaCup. File ini berisi definisi tatabahasa model yang dipakai. Di sini didefinisikan symbol-simbol tatabahasa yang termasuk terminal dan non terminal. Simbol symbol yang mengkaitkan dengan aksi semantic dan pembentukan code (code generator) dianggap sebagai non terminal. Ia kemudian dapat diganti dengan aksi semantic yang terdefinisi di kelas aksi semantic atau pembentukan kode.

Seluruh file-file tersebut membentuk kesatuan model pemroses, dan cara persiapan dalam komputer anda dan cara pemakaiannya akan dijelaskan pada bab berikut.

BAB 4 CARA PEMAKAIAAN

4.1 Persiapan

Perangkat keras dapat dipakai Personal Computer atau Laptop baik yang beroperating system Windows maupun Linux. Sebelumnya pastikan bahwa kompilator Java (teruji untuk java 1.4) sudah terpasang dalam komputer. Kemudian pastikan bahwa pustaka/libraries sudah tersedia Jlex dan JavaCup, dan libraries ini bisa diakses dalam environment variable CLASSPATH. Kedua libraries ini harus dikompilasi dengan kompilator yava yakni javac.

Setelah libraries itu sudah siap, langkah selanjutnya adalah mengkompilasi source code model dengan cara sebagai berikut, yakni ketikkan setiap langkah berikut:

a. Jika memakai Windows. If you have the 'compile.bat' file and are using Windows/MS-DOS ketikkan
    a. "compile" sudah cukup, namun kalau ingin dilakukan satu persatu ketikan berikut
    b. java JLex.Main Scanner.lex
    c. java java_cup.Main Parser.cup
    d. ren Scanner.lex.java Yylex.java
    e. javac *.java
b. Jika anda memakai Linux, ketikkan
    a. java JLex.Main Scanner.lex
    b. java java_cup.Main Parser.cup
    c. mv Scanner.lex.java Yylex.java
    d. javac *.java

Pada keadaan ini, kita sudah siap untuk mencoba model kompilator kita.

4.2 Cara menggunakan kompilator

Dengan mengasumsikan bahwa instalasi telah dilakukan dengan benar, maka cara mengkompilasi program yang ditulis berdasarkan tatabahasa MINUI adalah dengan memberikan perintah baris (command line) sebagai berikut

> java parser < [input_file]

dimana, input-file adalah file yang berisi program berdasarkan tatabahasa yang dipakai. Jika tidak terjadi kesalahan, maka kita akan memperoleh bahasa sasaran sebagai output, namun jika terjadi kesalahan pada input-file maka kita akan pesan kesalahan akan ditampilkan di layar komputer.

Bahasa sasaran dapat disimpan dalam suatu file dengan memberikan perintah

> java parser < [input_file] > [target_file]

Jika kita sudah yakin tidak ada kesalahan, maka bahasa sasaran itu dapat dieksekusi dengan memberikan perintah berikut:

- java Machine [target_file]

perintah ini akan mensimulasikan bahasa sasaran dalam file target-file dengan instruksi-instruksi Java sehingga dapat berjalan seperti layaknya suatu kompilator lainnya.

4.3 Contoh masukan dan keluaran

Berikut adalah suatu input yang dapat diujicobakan, simpan program berikut dalam suatu file dan proses menurut petunjuk di atas.

```
{ var a : integer
var b : integer
  var c : boolean;

  get a
  b := 0
  if a > b then
     c := true
  else
     c := false
  end if
  { var a : integer;
    a := 5
    if c then
       repeat
          put b
          b := b+1
       until b >= a
    end if
  }
}
```

Dalam contoh ini tidak ada kesalahan, sehingga akan menghasilkan output kode sasaran sebagai berikut:

```
0   NAME 0 0
3   PUSHMT
4   SETD 0
6   PUSH -32768
8   PUSH -32768
10   PUSH -32768
12   NAME 0 0
15   READI
16   STORE
17   NAME 0 1
20   PUSH 0
22   STORE
```

```
23  NAME 0 0
26  LOAD
27  NAME 0 1
30  LOAD
31  FLIP
32  LT
33  PUSH 45
35  BF
36  NAME 0 2
39  PUSH 1
41  STORE
42  PUSH 51
44  BR
45  NAME 0 2
48  PUSH 0
50  STORE
51  NAME 1 0
54  PUSHMT
55  SETD 1
57  PUSH -32768
59  NAME 1 0
62  PUSH 5
64  STORE
65  NAME 0 2
68  LOAD
69  PUSH 115
71  BF
72  NAME 0 1
75  LOAD
76  PRINTI
77  PUSH 10
79  PUSH 13
81  PRINTC
82  PRINTC
83  NAME 0 1
86  NAME 0 1
89  LOAD
90  PUSH 1
92  ADD
93  STORE
94  NAME 0 1
97  LOAD
98  NAME 1 0
101 LOAD
102 LT
103 PUSH 0
```

```
105   EQ
106   PUSH 0
108   EQ
109   PUSH 115
111   BF
112   PUSH 72
114   BR
115   PUSHMT
116   NAME 1 0
119   SUB
120   POP
121   SETD 1
123   PUSHMT
124   NAME 0 0
127   SUB
128   POP
129   SETD 0
131   HALT
```

Dan jika dieksekusi dan diberikan input yang sesuai, ia akan berjalan sesuai dengan yang kita harapkan. Misalkan kita masukkan angka 3 sebagai input untuk variable a, maka variable c akan berisi nilai Boolean true, dan kemudian isi variable b dari 0 sampai dengan 5 akan dicetak di layar komputer.

Demikianlah satu contoh ringkas masukan dan keluaran, rincian uji coba dengan bermacam variasi input dapat ditemukan dalam disk beserta dokumentasi ini.

Daftar Referensi

- Aho, A. V., Sethi, R., Ullman, J. D., ***Compilers: Principles, Tehchniques, and Tools***, Addison Wesley 1986
- Berk, Elliot, JLex: A Lexical Analyzer Generator for Java(TM) http://www.cs.princeton.edu/~appel/modern/java/JLex/
- Hudson, Scott , Cup v0.10k release 1999, http://www.cs.princeton.edu/~appel/modern/java/CUP/
- Heru Suhartanto, Teknik Kompilator (slides materi kuliah), Fasilkom UI, 2005.
- Heru Suhartanto, Model Kompilator berbasis Pascal, terdaftar di Dirjen HAKI, Departemen Hukum dan Hak Asasi Manusia, NO: 029005, tahun 2006

Lampiran A: Grammar Model

```
program:    scope R38 R0

scope:      '{' C0 R1 declarations C1 R3 ';' statements C2 R5 '}'

statements:   ,
              statement  statements

statement:    identifier C6 assignOrCall,
              'if' expression C13 C11 R8 'then' statements optElse 'end' 'if',
              'repeat' R11 statements 'until' expression C13 C11 R18 R8 R12 R10,
              'loop' R11 R53 statements R12 'end' 'loop' R51,
              'exit' R52,
              'put' outputs,
              'get' inputs,
              scope

optElse:      R10,
              'else' R7 R10 statements R9

assignOrCall:   C28 C29 R45 R44 C7,
                '(' R45 C28 C30 arguments ')' C32 R44 C7,
                ':' '=' C20 R31 assignExpression,
                '[' C21 R40 subscript ']' ':' '=' assignExpression

assignExpression:   expression C16 C11 R33 C7

subscript:      simpleExpression C12 C11 R41

expression:       simpleExpression optRelation

optRelation:      ,
                  '=' simpleExpression C14 C11 C11 C10 R21,
                  '#' simpleExpression C14 C11 C11 C10 R22,
                  '<'  lessOrLessEq,
                  '>' greaterOrGreaterEq

lessOrLessEq:           simpleExpression C15 C11 C11 C10 R23,
                    '=' simpleExpression C15 C11 C11 C10 R24,

greaterOrGreaterEq:   simpleExpression C15 C11 C11 C10 R25,
                      '=' simpleExpression C15 C11 C11 C10 R26
```

```
simpleExpression:    term moreTerms

moreTerms:           ,
                     '+' C12 C11 term C12 R14 moreTerms,
                     '-' C12 C11 term C12 R15 moreTerms,
                     '|'  C13 C11 term C13 R20 moreTerms

term: factor moreFactors

moreFactors: ,
                          '*' C12 C11 factor C12 R16 moreFactors,
                          '/' C12 C11 factor C12 R17 moreFactors,
                          '&' C13 C11 factor C13 R19 moreFactors

factor: primary,
        '+' factor C12,
        '-' factor C12 R13,
        '~' factor C13 R18

primary: integer C9 R36,
        'true' C10 R35,
        'false' C10 R34,
        '(' expression ')',
        '{' C0 R2 declarations C1 R3 ';' statements ';' expression C2 R6 '}',
        identifier C6 subsOrCall

subsOrCall: C37 R49 C29 C8 R50 C7,
        '(' C33 R46 C30 arguments ')' C32 C8 R47 C7,
        '[' C21 R40 subscript ']' C8 R32 C7

arguments: expression C34 C31 C11 R48 moreArguments

moreArguments: ,
        ',' expression C34 C31 C11 R48 moreArguments

declarations: declaration moreDeclarations
        moreDeclarations: ,
        declaration moreDeclarations

declaration: 'var' identifier C3 C4 optArrayBound ':' type C5 C11 C7,
        type 'func' identifier C3 C26 R7 C22 C23 C5 C0 R2 funcBody,
        'proc' identifier C3 C24 R7 C22 C0 R1 procBody

funcBody: '=' expression C36 C11 C11 C2 R6 R43 R9 C7,
        '(' C30 parameters C35 C1 ')' '=' expression C36 C11 C11 C27 R6 R43 R9 C7
```

```
procBody: scope C2 R5 R42 R9 C7,
        '(' C30 parameters C35 C1 ')' scope C27 R5 R42 R9 C7

type: 'integer' C9,
        'boolean' C10

optArrayBound: C18 R37,
        '[' simpleExpression C12 C11 R39 ']' C19

parameters: identifier C3 C4 C25 ':' type C5 C11 C7 C34 moreParameters

moreParameters: ,
        ',' identifier C3 C4 C25 ':' type C5 C11 C7 C34 moreParameters

Outputs: output moreOutput R30

output: expression C12 C11 R28,
                Text R29,
                'skip' R30

moreOutput: ,
                ',' output moreOutput

inputs: input moreInputs

moreInputs: ,
                ',' input moreInputs

input: identifier C6 optSubscript C17 R27 C7
                optSubscript: C20 R31,
                '[' C21 R40 subscript ']'
```

Lampiran B: Aturan Aksi Semantik dan Pembentukan Kode

Context Checking Rules

- C0 masuk ke suatu scope.
- C1 option mencetak tabel simbol scope yang bersangkutan.
- C2 keluar dari suatu scope.
- C3 periksa bahwa identifier belum dideklarasi di dalam scope ini, masukkan identifier ke dalam tabel simbol, Push index dari tabel simbol.
- C4 masukkan lexic level dan order number ke dalam tabel simbol.
- C5 masukkan type ke dalam tabel simbol.
- C6 periksa bahwa identifier sudah dideklarasi. Push indek tabel simbolnya.
- C7 Pop indek tabel simbol.
- C8 Push type.
- C9 Push type Int.
- C10 Push type bool.
- C11 Pop type.
- C12 Periksa bahwa type adalah int.
- C13 Periksa bahwa type adalah bool.
- C14 Periksa type untuk perbandingan kesamaan (equality comparison).
- C15 Periksa type untuk perbandingan urutan (order comparison).
- C16 Periksa type untuk assignment.
- C17 Periksa bahwa type variable adalah integer.

Array.

- C18 beri tanda, entry tabel simbol sebagai skalar.
- C19 beri tanda, entry tabel simbol sebagai variable array.
- C20 Periksa bahwa identifier adalah nama variable skalar.
- C21 Periksa bahwa identifier adalah nama variable array.

Fungsi dan prosedur.

- C22 masukkan lexic level dan order number ke dalam tabel simbol
- C23 masukkan type int atau bool.
- C36 periksa bahwa type dari ekspresi sama dengan type dari fungsi.

Parameter.

- C24 masukkan prosedur ke dalam tabel simbol.
- C25 masukkan parameter ke dalam tabel simbol.

- C26 masukkan fungsi ke dalam tabel simbol.
- C27 keluar dari scope yang mengandung parameter.
- C28 periksa bahwa identifier merupakan nama prosedure.
- C29 periksa bahwa fungsi atau prosedur tak punya parameter.
- C30 Push jumlah argument = 0
- C31 periksa argument terhadap parameter.
- C32 periksa bahwa semua argumen sudah dilihat. Pop jumlah argument.
- C33 periksa bahwa identifier adalah nama fungsi.
- C34 tambah nilai jumlah argument.
- C35 masukkan jumlah argument (parameter) ke dalam tabel simbol. Pop jumlah argument.
- C37 if identifier suatu fungsi maka C33 else C20

Code Generation Rules

- R0 tentukan pc dan mt
- R1 periksa bahwa lexic level <displaySize. Buat instruksi untuk memasuki scope statement.
- R2 periksa bahwa lexic level <displaySize. Buat instruksi untuk memasuki scope ekspresi.
- R3 buat instruksi untuk mengalokasi variable.
- R4 buat instruksi untuk mendapatkan address untuk hasil perhitungan.
- R5 buat instruksi untuk menghapus variable dan keluar dari scope statement.
- R6 buat instruksi untuk memindahkan hasil perhitungan, hapus variable, dan keluar dari scope ekspresi.
- R7 buat instruksi untuk forward branch (BR).
- R8 buat instruksi untuk forward bersyarat (BF).
- R9 buat instruksi untuk membetulkan address dari forward branc-2 (forward branch-2 bersyarat).
- R10 buat instruksi untuk membetulkan address dari forward branch bersyarat.
- R11 simpan address untuk backward branch.
- R12 buat instruksi untuk backward branch.
- R13-R26 buat instruksi yang sesuai dengan operator.
- R27 buat instruksi untuk membaca dan menyimpan integer.
- R28 buat instruksi untuk mencetak integer.
- R29 buat instruksi untuk mencetak teks.
- R30 buat instruksi untuk memindahkan kursor ke baris baru.
- R31 buat instruksi untuk mendapatkan address variable.
- R32 buat instruksi untuk memperoleh nilai dari suatu variable.
- R33 STORE.
- R34 PUSH 0.
- R35 PUSH 1.
- R36 PUSH nilai integer.
- R37 alokasi space untuk variabel
- R38 HALT

Array.

- R39 buat instruksi untuk memeriksa bahwa batas array adalah non-negatif. buat instruksi untuk mengalokasikan array.
- R40 buat instruksi untuk memperoleh address array.
- R41 buat instruksi untuk memeriksa bahwa indek array berada pada batasnya.

Fungsi, prosedur dan parameter.

- R42 buat instruksi untuk kembali dari prosedure.
- R43 buat instruksi untuk kembali dari fungsi.
- R44 buat instruksi untuk memanggil suatu prosedur.
- R45 buat instruksi untuk membentuk tanda block untuk pemanggilan prosedur.
- R46 buat instruksi untuk membentuk tanda block untuk pemanggilan fungsi.
- R47 buat instruksi untuk memanggil fungsi.
- R48 buat instruksi untuk menyimpan argument untuk pemanggilan prosedu/fungsi.
- R49 if identifier suatu fungsi maka R46 else R31
- R50 if identifier suatu fungsi maka R47 else R32