PERTEMUAN 8

PENANGANAN KESALAHAN

A. TUJUAN PEMBELAJARAN

Setelah mengikuti materi pada pertemuan ini, Mahasiswa diharap mampu menjelaskan konsep penguraian dan penyusunan suatu kalimat atau statemen program.

B. URAIAN MATERI

Sederhananya seorang kompilator hanya memproses program yang benar, desain yang bagus, dan implementasi. Walaupun programer sudah membuat sebuah program sebaik mungkin, namun kompilator juga diharapkan dapat membantu programer dalam melacak dan menemukan kesalahan yang terdapat pada sebuah program. Menariknya, beberapa bahasa pemrograman telah dirancang untuk bisa menemukan kesalahan-kesalahan pada bahasa pemrograman tersebut dengan memberi tanda seperti perubahan warna. Hal tersebut dapat mempermudah programer atau kompilator dalam menemukan kesalahan dan segera untuk memperbaikinya.

1. Kesalahan Program

Umumnya kesalahan pemrograman dapat terjadi pada berbagai kategori, yaitu:

a. Kesalahan Leksikal

Kesalahan ini mencakup kesalahan dalam ejaan/penulisan pengenal, kata kunci, atau operator. Contohnya seperti penggunaan pengenal ellipseSize akan tetapi diketik elipseSize atau ellipsSize. Contoh yang lain adalah dalam pengetikan then tetapi diketik thn atau the.

b. Kesalahan Sintaks

Kesalahan sintaks sering terjadi pada operasi aritmatika seperti tanda kurung buka/tutup yang lebih atau kurang dan termasuk titik koma (;) yang salah tempat.

Contoh:

K:= X + (L * (M+N)

Operasi aritmatika di atas terjadi kesalahan yaitu kurangnya tanda kurung tutup.

c. Kesalahan Semantik

Kesalahan semantik termasuk ketidakcocokan tipe antara operator dan operan.

1) Tipe data yang salah

Contohnya:

int a;

a = 4.5 * 0.11

Kesalahan contoh di atas yaitu pada deklarasi menggunakan tipe data integer, akan tetapi pada operasi menggunakan real/desimal. Hal tersebut akan membuat error pada program.

2) Variabel belum didefinisikan

Contohnya:

C := C + 2

Variabel C belum di definisikan pada deklarasi, maka akan terjadi error pada program.

d. Kesalahan logika

Kesalahan logika dapat berupa apa saja mulai dari penalaran yang salah oleh programer hingga penggunaan dalam program C dari operator penugasan = yang seharusnya operator pembanding ==. Program yang berisi = mungkin benar, namun mungkin tidak mencerminkan maksud pemrogram.

2. Penanganan Kesalahan

Hal-hal yang perlu dilakukan dalam penanganan kesalahan adalah:

a. Mendeteksi Kesalahan

tata bahasa yang mendefinisikan sintaks bahasa pemrograman biasanya bukan tata bahasa bebas konteks. Mereka sering ditentukan dalam dua bagian: tata bahasa bebas konteks yang ketat mendefinisikan superset bahasa, dan serangkaian pembatasan sensitif konteks pada sintaks bebas

konteks. Sebagai konsekuensi dari ini, kesalahan terdeteksi dengan dua cara. Kesalahan dalam sintaks bebas konteks dari program sumber biasanya mudah dideteksi karena spesifikasi yang tepat dari sintaks bebas konteks. Kesalahan juga dapat dideteksi oleh pemeriksaan pada sintaks program sumber yang sensitif terhadap konteks. Kesalahan kelas ini, termasuk kesalahan dalam jenis variabel dan ekspresi, seringkali lebih sulit dideteksi secara efektif, karena ketidakjelasan dan ketidaksadatan deskripsi prosa yang mendefinisikan.

Kesalahan bebas konteks paling efektif terdeteksi oleh algoritma penguraian sesuai dengan sintaks. Jenis parser ini terdiri dari algoritma umum yang digerakkan oleh tabel dan serangkaian tabel yang dihasilkan dari tata bahasa bebas konteks dari bahasa pemrograman. Karena algoritma penguraian dan pembuatan tabel algoritma ditentukan dengan tepat, dapat dibuktikan bahwa parser menerima dengan tepat bahasa yang ditentukan oleh tata bahasa tertentu. Kesalahan bebas konteks dalam program sumber terdeteksi ketika algoritma penguraian menemukan kesalahan dalam menginput pada tabel. Ketika ini terjadi, parser memanggil beberapa pemulihan atau perbaikan rutin untuk menangani kesalahan.

Sangat disayangkan, dalam hal keandalan kompiler, bahwa beberapa kompiler yang banyak digunakan saat ini menggunakan metode ad hoc analisis sintaks (yang kami maksud dengan ad hoc adalah metode apa pun yang tidak diformalkan). Metode ad hoc populer karena memungkinkan antarmuka yang efisien, langsung, dan sederhana secara konseptual antara analisis sintaksis dan pembuatan kode. Algoritma yang menghasilkan kode untuk konstruksi tertentu dapat dimasukkan ke dalam analisis sintaksis untuk konstruksi itu. Pendekatan ad hoc pada analisis sintaksis terdiri dari penulisan program kasus khusus yang hanya menerima string dari bahasa yang dianalisis dan pada saat yang sama menghasilkan kode objek atau beberapa bentuk perantara yang diproses oleh penyusun lain. Pernyataan ini hanyalah definisi longgar dari penganalisis sintaksis.

Metode formal dapat dibuktikan benar; kebenaran pengurai ad hoc bergantung sepenuhnya pada perawatan yang dilakukan oleh perancang dan pemrogramnya untuk memastikan kebenarannya. Desainer parser ad hoc harus memiliki pemahaman yang luar biasa tentang detail bahasa yang

diimplementasikan dan harus sangat teliti dalam memeriksa ulang kode mereka dengan spesifikasi bahasa.

Berikutnya kompiler ad hoc biasanya terdiri dari sejumlah besar pemeriksaan simbol input untuk melihat bahwa itu legal dalam konteks simbol. Jadi kesalahan terdeteksi ketika salah satu dari tes ini gagal secara tidak terduga. Jika kompilator membuat dengan benar, kegagalan ini menyebabkan beberapa mekanisme penanganan kesalahan dipanggil, mungkin melalui pemanggilan prosedur.

Kekurangan metode analisis sintaksis ad hoc yang disebutkan di atas harus diperhatikan oleh semua programer. Tidak peduli seberapa bagus metode analisis sintaksis bebas konteks mereka, hampir semua kompiler menggunakan metode ad hoc untuk memeriksa sintaks sensitif konteks. Pemeriksaan tersebut mencakup memastikan bahwa variabel, konstanta, label, dan nama prosedur ditentukan dan memiliki atribut yang benar untuk konteks kemunculannya. Dalam mendesain bahasa pemrograman kami berusaha untuk menjaga sintaks operasi serupa untuk semua tipe data. Karena itu, kesalahan dalam atribut operan biasanya memiliki lebih banyak efek terlokalisasi daripada kesalahan bebas konteks, yang seringkali membuat struktur sebagian besar program sumber tidak valid. Oleh karena itu, seringkali lebih mudah untuk memulihkan dari kesalahan sensitif konteks daripada dari kesalahan bebas konteks.

Kami berikan contoh untuk dapat mendeteksi kesalahan input string dan hal apa yang dilakukan setelah kesalahan tersebut terdeteksi:

```
K = L + M + N + O + P + Q + R

THEN A = A*2;
ELSE A = B/2;
```

pada contoh di atas terlihat jelas bagi seorang programer letak kesalahannya adalah tidak adanya token IF. Lokasi THEN adalah awal mula kesalahan terdeteksi dalam penguraian dari kiri ke kanan. Ketika pemeriksaan sampai THEN, programer dengan cepat melihat langkah sebelumnya untuk menemukan titik kesalahan sebenarnya.

Pada contoh di atas, parser dapat melakukan langkah mundur dalam mendeteksi kesalahan, tetapi proses tersebut terlalu memakan waktu yang lama. Lyon (1974) memberikan algoritme yang memperbaiki program yang tidak valid dengan mengubahnya menjadi mendekati program yang valid,

dalam arti meminimalisir penyisipan dan penghapusan simbol yang diperlukan untuk melakukan transformasi. Algoritma mengambil sejumlah langkah sebanding dengan n^3, di mana n adalah jumlah simbol dalam program sumber. Algoritma ini terlalu lambat untuk mencocokkan terjemahan bahasa pemrograman. Barnard (1976) menunjukkan bahwa perbaikan kesalahan jarak minimum ini mungkin lebih disukai daripada metode yang memakan waktu lebih sedikit tetapi tentu saja tidak menjamin koreksi kesalahan sintaksis.

Meskipun token THEN dalam contoh sebelumnya adalah titik paling awal di mana kesalahan sintaksis dapat dideteksi, itu pasti bukan yang terbaru. Beberapa metode penguraian formal, termasuk pengurai prioritas, mungkin tidak mendeteksi kesalahan sintaksis sampai beberapa simbol telah diproses. Metode parsing LL dan LR mendeteksi kesalahan ketika token THEN diproses. Teknik parsing ini memiliki apa yang disebut properti valid prefix, yang berarti jika parser tidak mendeteksi kesalahan di bagian pertama X dari program XY, maka harus ada beberapa string simbol W sehingga XW adalah program yang valid.

Perlu dicatat bahwa kompilator biasanya tidak dapat menentukan penyebab kesalahan, bahkan jika kesalahan segera terdeteksi. Kesalahan tersebut mungkin disebabkan oleh kesalahan pengetikan, kekeliruan programmer, atau kesalahpahaman yang serius tentang bahasa pemrograman. Semakin cepat kesalahan terdeteksi, semakin besar kemungkinan kompilator dapat menebak penyebabnya dengan benar dan mengambil tindakan yang sesuai. Tindakan ini harus selalu menyertakan deskripsi kesalahan yang tepat dan lengkap.

b. Melaporkan Kesalahan

Agar deteksi kesalahan menjadi nilai yang efektif bagi pengguna kompilator, setiap kesalahan setelah terdeteksi harus dilaporkan dengan jelas dan tepat. Bagi mereka yang belum pernah terlibat dalam proyek perangkat lunak besar, mungkin tidak jelas bahwa pelaporan kesalahan memerlukan pertimbangan khusus. Program aplikasi kecil dapat menghasilkan satu dari selusin pesan pada kesempatan yang jarang terjadi ketika terjadi kesalahan. Program semacam itu dapat menggunakan rutinitas keluaran standar atau pernyataan PRINT dari bahasa yang diprogram untuk

mencetak pesan singkat kepada pengguna. Bagaimanapun, bahwa respon *compiler* terhadap kesalahan harus dilakukan dengan sangat baik. Jika kompiler digunakan terutama untuk pengembangan program, kompilator mungkin lebih sering dipanggil untuk menghasilkan pesan kesalahan yang baik daripada menghasilkan kode objek yang baik.

Pembuatan kode penuh kesalahan sebagai pengganti pesan tidak lagi dapat dibenarkan, kecuali pada mikrokomputer terkecil. Salah satu pendekatan untuk menghasilkan pesan bahasa alami adalah dengan menghasilkan kode kesalahan dalam kompiler, menyimpannya dalam file sementara, dan mengandalkan pass akhir terpisah untuk mencetak daftar sumber dan mengganti kode kesalahan dengan teks lengkap dari pesan kesalahan. Pendekatan ini pada dasarnya adalah otomatisasi dari metode buku kesalahan dan mungkin standar minimum pelaporan kesalahan yang dapat diterima saat ini. Ini menghilangkan masalah ketidaktersediaan dan ketidakakuratan buku kesalahan, tetapi tidak melakukan apa pun untuk membuat deskripsi kesalahan lebih dapat dipahami.

Pesan kesalahan yang telah didiskusikan tidak berisi referensi ke objek dalam program sumber yang merupakan penyebab kesalahan, dan seringkali tidak berisi informasi tentang lokasi kesalahan selain jumlah baris yang dideteksi. Ada beberapa cara untuk memperbaiki pesan kesalahan tersebut. Pesan kesalahan harus secara khusus menyebutkan informasi apa pun yang diketahui kompilator yang mungkin berguna bagi pengguna dalam membedakan penyebab sebenarnya dari kesalahan. Informasi tersebut dapat mencakup penunjuk yang terlihat ke kolom tempat kesalahan terdeteksi, nama dan atribut variabel dan konstanta yang terkait dengan kesalahan, tipe yang terlibat dalam kesalahan pencocokan tipe, dan sebagainya. Semua informasi ini harus diungkapkan dalam bahasa sumber dan harus berorientasi pada pengguna. Dalam beberapa kasus mungkin sulit bagi kompilator untuk menyediakan informasi ini, tetapi biasanya lebih mudah bagi kompilator daripada bagi pengguna. Penulis kompilator harus bersusah payah untuk memastikan bahwa pesan tertentu dicetak untuk menggambarkan kesalahan yang berbeda, daripada mengandalkan pengguna umum yang mungkin salah informasi untuk menafsirkan satu pesan samar yang digunakan untuk beberapa kesalahan berbeda. Untuk

digunakan secara ekstensif pesan seperti "Sintaks pernyataan buruk" sama sekali tidak dapat diterima untuk semua kompilator.

3. Pemulihan Kesalahan

Kompilator yang berguna harus dapat menyesuaikan statusnya setelah menemui kesalahan sehingga kompilasi dapat dilanjutkan dengan cara yang wajar melalui program lainnya. tidak ada upaya untuk mengubah program yang salah menjadi program yang valid secara sintaksis.

Saat ini sebagian besar menggunakan metode analisis sintaksis yang berbeda. Oleh karena itu, sulit untuk memberikan komentar umum tentang proses pemulihan kesalahan. Dalam diskusi berikut kami mengembangkan model penganalisis sintaksis dan membahas strategi pemulihan kesalahan untuk model ini.

a. Mekanisme Pemulihan Ad Hoc

Pendekatan yang diambil untuk pemulihan kesalahan di sebagian besar penyusun industri berkinerja tinggi adalah pendekatan informal. Bagian dari kompilator yang mendeteksi kesalahan memanggil beberapa rutinitas pemulihan kesalahan dengan tujuan khusus yang mampu menangani kesalahan atau kelas kesalahan tertentu. Tindakan yang diambil oleh rutinitas ditentukan oleh penulis kompilator saat mereka mengkodekan rutinitas. Dengan cara ini penulis kompilator dapat mengantisipasi kesalahan sintaksis tertentu dengan membuat kategori/kelas dan memulihkannya dengan cara apa pun yang mereka anggap paling masuk akal dan efektif.

Skema pemulihan ad hoc bekerja dengan baik selama kesalahan sintaksis yang ditemukan termasuk dalam salah satu kelas kesalahan yang diantisipasi oleh penulis kompilator. Sayangnya, sangat sulit untuk mengantisipasi semua kesalahan sintaksis. Metode pemulihan ad hoc memiliki kelemahan yaitu cenderung bereaksi buruk terhadap situasi yang tidak terduga dan dapat menghasilkan rentetan pesan kesalahan. Hal ini terkait dengan masalah bahwa kompilator harus memiliki reaksi yang benar, terbatas, dan stabil terhadap kesalahan. Kekurangan lainnya dari metode pemulihan kesalahan ad hoc adalah banyaknya upaya yang harus dilakukan

dalam penulisan rutinitas pemulihan kesalahan dan antisipasi yang cermat dari semua kemungkinan kesalahan sintaksis.

Dimungkinkan untuk menggunakan teknik pemulihan kesalahan ad hoc dalam parser yang diarahkan pada sintaks atau berbasis tabel. Kesalahan biasanya terdeteksi dalam pengurai berbasis tabel ketika tindakan penguraian yang ditunjukkan dalam tabel penguraian. Untuk menambahkan pemulihan kesalahan ad hoc ke pengurai semacam itu, Anda hanya perlu mengubah tindakan kesalahan dalam tabel menjadi panggilan ke rutinitas pemulihan kesalahan yang dikodekan secara manual. Jadi kesalahan tertentu atau sekelompok kesalahan menyebabkan rutinitas tertentu dipanggil. Rutin ini kemudian memodifikasi tumpukan, tabel simbol, dan / atau string input untuk menempatkan parser ke dalam kondisi stabil lagi.

Keuntungan utama dari metode pemulihan ad hoc adalah bahwa metode ini memungkinkan penulis kompilator untuk memasukkan intuisi mereka tentang kesalahan sintaksis umum ke dalam skema pemulihan kesalahan. Alih-alih mengkhususkan algoritma pemulihan, penulis kompilator dapat menambahkan informasi ini ke tata bahasa dalam bentuk produksi kesalahan. Produksi kesalahan adalah produksi khusus yang hanya berlaku jika kesalahan sintaksis ditemukan. Sistem parser dan pemulihan kesalahan dapat tetap umum, sementara informasi khusus bahasa digabungkan melalui produksi kesalahan. Wirth (1968) menggunakan tabel produksi kesalahan dalam parser yang diutamakan sederhananya untuk PL360 untuk memilih pesan kesalahan yang sesuai ketika terjadi kesalahan, tabel dicari untuk sisi kanan yang cocok dengan elemen teratas tumpukan, dan pesan yang dilampirkan di sisi kanan ini dicetak. Frase yang salah yang dicocokkan dengan produksi kesalahan kemudian dibuang. Aho dan Johnson (1974) menggunakan token kesalahan khusus dalam tata bahasa LALR (l) mereka untuk menunjukkan kepada parser cara memulihkan dari kesalahan sintaksis. Ketika kesalahan terdeteksi, pengurai mencari melalui string input sampai menemukan beberapa simbol yang secara legal dapat mengikuti kesalahan dalam tata bahasa. Kemudian menyesuaikan tumpukan dan melanjutkan seolah-olah telah membaca kesalahan token dalam string input.

Kerugian dari kesalahan produksi sebagai cara pemulihan mirip dengan metode lain yang bergantung pada bahasa. Banyak waktu harus dihabiskan untuk memasukkan kesalahan dan memeriksa apakah mereka bekerja dengan benar. Selalu ada bahaya bahwa beberapa situasi kesalahan akan terlewatkan dan pengurai akan bereaksi buruk dalam situasi yang tidak terduga. Bergantung pada bentuk tabel yang digunakan untuk tata bahasa, ada kemungkinan bahwa produksi kesalahan yang ditambahkan dapat membuat tata bahasa sangat besar dan tidak mungkin untuk disimpan.

b. Pemulihan Sintaks

Mengingat kemungkinan kelalaian dan kesalahan dalam pengkodean mekanisme pemulihan kesalahan ad hoc, tampaknya harus memiliki beberapa pendekatan umum formal untuk menghasilkan mekanisme pemulihan kesalahan. Banyak penelitian telah dikhususkan untuk pengembangan algoritma yang beroperasi pada struktur data parser bebas konteks dari berbagai jenis sehingga dapat menyesuaikannya untuk melanjutkan setelah kesalahan sintaks bebas konteks. Bagian ini menjelaskan pendekatan umum untuk pemulihan kesalahan yang diarahkan pada sintaks yang dapat diterapkan pada parser dengan arahan sintaks bottom-up.

Strategi pemulihan kesalahan biasanya dapat menggunakan hanya sebagian kecil dari konteks kesalahan. Pemulihan kesalahan terjadi dalam dua tahap. Fase pertama, yang disebut fase kondensasi, mencoba mengurai bentuk sentensial pada tumpukan dan di bagian string input yang belum dibaca, sehingga sebanyak mungkin konteks kesalahan dapat digunakan dalam fase kedua pemulihan . Pertama, coba langkah mundur, ini terdiri dari upaya untuk melakukan pengurangan pada formulir sentensial di tumpukan parser. Pengurangan terdiri dari penggantian sisi kanan beberapa produksi tata bahasa dengan sisi kiri produksi itu. Ketika semua kemungkinan pengurangan di atas tumpukan telah dilakukan, mekanisme pemulihan mencoba gerakan maju setelah kesalahan, yang terdiri dari penguraian bagian dari string input. Gerakan maju berhenti ketika menemukan kesalahan kedua dalam string input atau ketika tidak dapat melakukan tindakan penguraian lagi tanpa menggunakan konteks sebelum titik

kesalahan. Jumlah string masukan yang belum dibaca dapat digunakan oleh gerakan maju, tetapi karena semua masukan ini diurai dan diperiksa untuk kesalahan sintaksis, tidak ada teks sumber yang benar-benar diabaikan.

Untuk mengilustrasikan gerakan maju dan mundur, berikut contoh dari tata bahasa untuk pertimbangkan produksi:

<pernyataan> ::= <pernyataan> ; < pernyataan >

| < pernyataan >

dan segmen program ALGOL yang salah:

```
BEGIN  R:= S - 2
       S:= R*A;
       B:= S + 2
END;
```

dalam contoh di atas, ada titik koma yang hilang setelah pernyataan "R: = S - 2." Mari kita asumsikan pengurai bottom-up tanpa mengkhawatirkan detail strategi penguraian. Ketika titik koma yang hilang terdeteksi, tumpukan mungkin

...BEGIN <id>:= <id> - <integer>

dengan S sebagai token masukan berikutnya. Langkah mundur akan mengurangi tumpukan menjadi

... BEGIN <pernyataan>

titik kesalahan diwakili oleh ? digeser ke tumpukan, dan penguraian berlanjut dengan gerakan maju hingga titik tersebut

... BEGIN <pernyataan> ? <pernyataan> END

tercapai. Pada titik ini, diperlukan pengurangan titik kesalahan dan isi tumpukan diteruskan ke fase kedua. Singkatnya, langkah mundur dan maju berusaha menangkap konteks di sekitar tempat kesalahan terdeteksi.

Fase kedua dari pemulihan kesalahan melakukan operasi pencocokan pola pada elemen atas tumpukan parsing dalam upaya untuk menemukan produksi dalam tata bahasa yang sisi kanannya mirip dengan simbol pada tumpukan. Bagian atas tumpukan kemudian disesuaikan agar sesuai dengan

sisi kanan produksi atau awal produksi. Simbol masukan berikutnya juga dipertimbangkan dalam pilihan perubahan yang dilakukan pada tumpukan.

Dalam contoh sebelumnya, penyisipan titik koma pada titik kesalahan memungkinkan pengurangan

<pernyataan> ? <pernyataan>

menjadi

<pernyataan>.

bagaimanapun koreksi tidak selalu sesederhana itu. Pertimbangkan segmen program yang salah

R:= S - 2;

R = 2 THEN R:= S*T;

dalam program ini token "IF" tidak ada. Dengan asumsi bahwa kesalahan terdeteksi saat "=" dibaca, tumpukan pada titik ini akan berisi

... <pernyataan> ; <id>

dalam hal ini langkah mundur tidak dimungkinkan. Gerakan maju menghasilkan konten tumpukan berikut:

... <pernyataan> ; <id> ? = <ekspresi>

dengan token masukan berikutnya THEN. Langkah maju tidak dapat dilanjutkan, dan pengurangan titik kesalahan mungkin diperlukan. Satu kemungkinan adalah mengganti "=" dengan ":="; namun, token input "THEN", dalam kasus ini, tidak mengikuti. Kemungkinan lain adalah memasukkan token "IF" sebelum simbol <id>, yang memungkinkan pengurangan menjadi

... pernyataan; IF ekspresi

Contoh ini menggambarkan bahwa tampaknya ada lebih dari satu perubahan yang memungkinkan untuk pemulihan dari kesalahan. Dalam kasus seperti itu, pilihan dapat dibuat berdasarkan kemungkinan token mana yang terlibat dalam kesalahan sintaks.

Masalah utama dengan mekanisme pemulihan yang diarahkan pada sintaks adalah bahwa mekanisme tersebut tidak mencerminkan

pengetahuan penulis kompilator tentang kemungkinan masuk akal dan penyebab dari kesalahan. Metode Graham-Rhodes mengurangi hal ini dengan menggunakan kriteria minimum pertimbangan dalam memilih di antara beberapa perubahan tumpukan yang mungkin dilakukan dalam beberapa situasi pemulihan. Penulis penyusun menggunakan pengetahuan mereka tentang bahasa pemrograman untuk menyiapkan dua vektor, yang berisi biaya untuk menghapus atau menyisipkan setiap simbol tata bahasa. Biaya untuk membuat perubahan menunjukkan ketidakmampuan untuk membuat perubahan. Rutinitas pemulihan mencoba meminimalkan biaya total dari semua perubahan yang dibuatnya pada tumpukan. Dengan demikian, secara umum penulis kompilator dapat menyesuaikan sistem pemulihan agar pulih lebih baik dalam situasi kesalahan.

c. Pemulihan Kesalahan Sekunder (Mode Panik)

Beberapa strategi pemulihan kesalahan, terutama strategi ad hoc, tidak selalu berhasil dalam melokalisir dan memulihkan dari kesalahan. Strategi tersebut mungkin rentan terhadap perulangan tak terbatas dan mungkin harus dibatasi untuk mencegahnya menghabiskan semua waktu yang tersedia atau semua teks sumber. Dalam kasus seperti itu, perlu memiliki skema pemulihan sekunder untuk digunakan kembali saat sistem biasa gagal. Skema pemulihan seperti itu harus dijamin pulih dan harus yakin untuk melokalisasi kondisi kesalahan. Metode yang biasa dilakukan untuk melakukan pemulihan sekunder adalah mode panik (Graham dan Rhodes, 1975) dan penghapusan unit.

Mode panik adalah proses membuang teks sumber sampai sejumlah pembatas tegas ditemukan. Contoh pembatas tegas adalah ';' di PL / I. Ketika pembatas tersebut ditemukan, tumpukan akan muncul ke titik di mana konteks kiri yang diwakilinya dapat diikuti oleh struktur sintaksis yang mengikuti pembatas. Dalam beberapa parser sederhana, mode panik digunakan sebagai satu-satunya bentuk pemulihan kesalahan. Ini pasti berhasil melokalisir kesalahan sintaksis, tetapi dapat membuang jumlah teks sumber yang berubah-ubah, yang tidak pernah diperiksa validitas lokal. Ketika kesalahan ditemukan dalam pernyataan PL / I, mode panik membuang simbol sumber sampai menemukan ';', membersihkan tumpukan

sehingga parser siap menerima pernyataan, dan melanjutkan penguraian setelah titik koma.

Penghapusan unit mengacu pada penghapusan seluruh struktur sintaksis dari teks sumber. Efeknya sama dengan mode panik, tetapi karena seluruh unit sintaksis (bukan hanya ujungnya) dihapus, penghapusan unit mempertahankan kebenaran sintaksis dari program sumber dan sesuai untuk kompiler perbaikan kesalahan. Cara ini lebih sulit dicapai daripada mode panik, karena bentuk terjemahan dari seluruh unit harus disimpan dalam parser sehingga dapat dengan mudah dihapus.

Perlu dicatat bahwa beberapa metode formal untuk memulihkan kesalahan dalam parser yang diarahkan sintaks dapat memulihkan kesalahan apa pun, dan pemulihan sekunder tidak diperlukan.

d. Pemulihan Sensitif Konteks

Bagian a dan b berfokus terutama pada pemulihan dari kesalahan bebas konteks. Pemulihan tanpa konteks telah dipelajari secara lebih menyeluruh daripada pemulihan yang sensitif konteks karena metode penguraian bebas konteks telah diformalkan dengan lebih memadai. Pemeriksaan pembatasan sensitif konteks biasanya dilakukan secara ad hoc, dan oleh karena itu, pemulihan sensitif konteks juga ad hoc. Jadi agak sulit untuk merumuskan teori umum tentang pemulihan yang sensitif konteks.

Bentuk paling efisien dari pemulihan sensitif konteks sering kali hanya melaporkan kesalahan, mengabaikannya, dan menonaktifkan bagian pembuatan kode dari kompilator (karena program tidak lagi valid secara sintaksis, terjemahan mungkin tidak mungkin). Sintaks sensitif konteks biasanya diekspresikan sebagai sekumpulan batasan pada bahasa bebas konteks yang lebih besar. Pengurai bebas untuk menerima anggota bahasa bebas konteks, memberikan pesan dan menyembunyikan terjemahan jika program tidak termasuk dalam bahasa yang lebih kecil yang ditentukan oleh batasan sensitif konteks. Error yang peka konteks biasanya memiliki efek yang sangat lokal, dan sedikit yang perlu dilakukan untuk memulihkannya. Jika kesalahan melibatkan pengenal yang tidak ditentukan, mungkin diinginkan untuk memasukkan pengidentifikasi ke dalam tabel simbol, membuat asumsi tentang jenisnya berdasarkan tempat terjadinya.

4. Perbaikan Kesalahan

Perbaikan kesalahan adalah bentuk pemulihan kesalahan yang menjamin validitas sintaksis dari bentuk perantara program yang diteruskan ke bagian lain atau lewat dari kompilator. Dengan demikian kompilator dapat melakukan pemeriksaan lengkap dan terjemahan bagian yang benar dari program asli.

Harus ditekankan bahwa algoritma mahal pun tidak selalu dapat memperbaiki kesalahan. Kompilator tidak mempunyai cukup informasi untuk menentukan apa yang benar, dalam kaitannya dengan apa yang diinginkan pemrogram, dan tidak ada kompilator masa kini yang cukup canggih untuk menggunakan informasi ini jika tersedia.

Garis besar umum pemulihan kesalahan memungkinkan modifikasi pada tumpukan parse, tabel simbol, dan string input yang belum dibaca selama proses pemulihan. Jika diperlukan perbaikan, modifikasi pada konteks kiri menjadi sangat tidak diinginkan. Konteks kiri mewakili informasi yang terakumulasi mengenai teks sumber yang telah diterjemahkan dan tidak lagi tersedia. Jika modifikasi stack dan tabel akan dilakukan, sistem perbaikan harus dapat menarik banyak pemrosesan yang sudah dilakukan pada bagian program yang salah. Ini berarti menambah kompleksitas dan ruang untuk menyimpan bagian-bagian sumber dan terjemahannya.

Untungnya, algoritme parsing yang memiliki properti prefiks yang valid tidak perlu mengubah tumpukan parse untuk memulihkan atau memperbaiki. LR dan sebagian besar pengurai ad hoc top-down, tanpa cadangan memiliki properti ini dan dapat memulihkan atau memperbaiki dengan memasukkan atau menghapus teks sumber.

a. Perbaikan Ad Hoc

Semua penjelasan tentang pemulihan ad hoc juga berlaku untuk perbaikan ad hoc, karena perbaikan kesalahan adalah cara canggih untuk pemulihan kesalahan. Metode yang tersedia untuk menjalankan rutinitas penanganan kesalahan tidak berubah. Rutinitas penanganan error sekarang harus dikodekan sedemikian rupa untuk menghasilkan keluaran yang valid secara sintaksis untuk kompiler lainnya. Keuntungan dan kerugian dari metode perbaikan ad hoc paralel dengan keuntungan dan kerugian dari pemulihan kesalahan ad hoc.

b. Perbaikan Sintaks

Mekanisme perbaikan yang diarahkan pada sintaks adalah mekanisme yang umum untuk kelas bahasa dan dapat dihasilkan oleh algoritme formal. Bagian ini menguraikan strategi perbaikan kesalahan yang diarahkan pada sintaks untuk parser top-down, tanpa cadangan. Desain diadaptasi dari Holt dan Barnard (1976) dan menggunakan vektor biaya untuk memungkinkan penulis kompilator menyetel perbaikan kesalahan.

Parser top-down mencoba untuk membangun pohon sintaks yang menjelaskan program sumber. Ini dilakukan dengan berulang kali mengganti simbol nonterminal paling kiri di pohon dengan sisi kanan salah satu produksinya. Kesalahan terdeteksi ketika pertumbuhan ke bawah dari pohon sintaks memerlukan simbol terminal tertentu untuk muncul dalam string input, dan simbol terminal itu tidak ada. Parser ini memiliki properti awalan yang valid, karena ia mendeteksi kesalahan sintaksis saat simbol kesalahan pertama ditemukan.

Strategi perbaikan kesalahan adalah strategi yang relatif sederhana. Ketika pohon sintaks menuntut agar terminal tertentu terjadi dalam string input, dan terminal tidak muncul, algoritma perbaikan memasukkan terminal yang diperlukan ke dalam teks sumber (string input). Dalam beberapa kasus, pohon sintaks mungkin memerlukan salah satu dari sejumlah simbol terminal, tidak ada yang muncul dalam string input. Dalam hal ini, vektor biaya penyisipan, yang menentukan biaya untuk memasukkan setiap simbol tata bahasa, digunakan untuk menentukan terminal mana yang harus disisipkan. Algoritma perbaikan mencoba meminimalkan biaya penyisipan. Simbol masukan yang menyebabkan kesalahan dibuang kecuali jika simbol terminal yang dimasukkan adalah simbol khusus (bukan pengenal atau konstanta) dan simbol masukan bukan simbol khusus.

Perhatikan bahwa tindakan perbaikan yang baru saja dijelaskan tidak perlu benar-benar menghilangkan kesalahan. Jika simbol input dipertahankan, mungkin simbol tersebut tidak mengikuti simbol yang disisipkan secara valid. Dalam kasus seperti itu, sistem perbaikan harus melakukan penyisipan lain. Perbaikan harus memeriksa kondisi ini sebelum kembali ke parser untuk mencegah parser melaporkan kesalahan lagi. Selain itu, sistem perbaikan harus mengandung batasan yang mencegahnya

memasuki urutan penyisipan tak terbatas yang tidak pernah memperbaiki kesalahan (apakah ini mungkin atau tidak tergantung pada tata bahasa).

Karena strategi perbaikan sederhana ini mungkin gagal untuk memperbaiki kesalahan sintaksis, Holt dan Barnard menyediakan sistem perbaikan tiga tingkat. Jika pernyataan atau batas garis ditemui di pohon sintaks atau dalam string input selama perbaikan, strategi perbaikan tingkat-garis (mirip dengan penghapusan unit Sec. 5-4.3) dipanggil untuk menghasilkan atau menghapus baris sumber teks. Jika strategi ini juga gagal, dan pembatas akhir program ditemukan, tingkat perbaikan tertinggi mengambil alih dan menghasilkan sisa program atau menghapus sisa string input.

Ketika simbol terminal yang disisipkan adalah pengenal atau konstanta, ada beberapa kemungkinan pilihan pengidentifikasi atau konstanta untuk disisipkan. Berdasarkan informasi dalam tabel simbol mengenai ruang lingkup dan jenis pengenal, dimungkinkan untuk memasukkan pengenal yang dapat muncul secara valid dalam konteks ini. Dalam beberapa kasus, mungkin hanya ada satu pengenal tersebut. Jika ada beberapa pengenal yang dapat muncul pada titik tertentu dalam teks sumber, pilihannya harus sewenang-wenang. Pilihan konstanta untuk disisipkan hampir selalu sewenang-wenang, kecuali konteksnya membatasi rentang nilai yang mungkin muncul (dalam subskrip array, misalnya). Holt dan Barnard (1976) selalu memasukkan 1 untuk konstanta numerik, "?" untuk konstanta string, dan $ NIL untuk pengenal. Pilihan sewenang-wenang dari salah satu kemungkinan pengenal terkadang menghasilkan koreksi kesalahan; $ NIL tidak pernah merupakan koreksi, hanya perbaikan. Untuk mencegah kesalahan sensitif konteks karena atribut $ NIL, harus ada atribut tipe universal yang memungkinkan pengenal seperti $ NIL menjadi operan operator mana pun dalam bahasa tersebut. Kehati-hatian kemudian harus diambil oleh pembuat kode untuk menghasilkan semacam terjemahan yang masuk akal untuk pengenal yang dimasukkan ini.

c. Perbaikan Sensitif Konteks

Tidak seperti kompilator pemulihan sederhana, kompilator perbaikan kesalahan tidak dapat mengabaikan kesalahan sensitif konteks. Bentuk perantara dari program yang diteruskan ke bagian lain dari kompilator harus

berisi operan dengan atribut yang benar di setiap ekspresi. Jika atribut dari beberapa operan tidak cocok dengan yang dibutuhkan oleh operator tertentu, operan dari tipe yang diperlukan harus diganti dengan operan yang tidak valid. Komentar dari paragraf sebelumnya berlaku di sini. Pengenal tiruan seperti $ NIL dengan tipe atribut universal dapat menggantikan operan yang tidak valid. Konstanta dan ekspresi dengan tipe atau nilai yang salah harus diganti dengan konstanta dan ekspresi yang dihasilkan kompiler dengan atribut yang benar. Cara paling efektif untuk melakukannya biasanya jelas dari karakteristik rutinitas deteksi kesalahan itu sendiri. Karena batasan sensitif konteks biasanya diuji secara ad hoc, strategi perbaikan untuk setiap kasus dapat disesuaikan dan disematkan dalam rutinitas deteksi kesalahan.

d. Perbaikan Ejaan

Pengenal alfanumerik memainkan peran yang sangat besar di sebagian besar bahasa pemrograman. Pengidentifikasi digunakan untuk menunjukkan variabel, konstanta, operator, label, dan tipe data. Banyak bahasa mengandung kelas pengenal khusus yang disebut kata kunci, yang digunakan sebagai pembatas sintaksis. Kesalahan pengetikan dan kelupaan sering menyebabkan pengenal salah eja. Dalam kasus seperti itu kesalahan sintaksis sering kali dapat diperbaiki dengan menggunakan algoritma.

Morgan (1970) menyajikan suatu algoritma yang mencoba untuk memilih pengenal legal yang dekat dengan pengidentifikasi ilegal yang terjadi di program sumber. Jika algoritme berhasil, perbaikan terdiri dari mengganti pengenal legal dengan yang ilegal.

Fungsi SPELLING_REPAIR (SUBJECT). Mengingat pengenal yang salah SUBJECT, fungsi ini memanggil fungsi COMPARE untuk menentukan apakah dua pengenal cukup mirip sehingga salah satunya bisa salah eja.

C. SOAL LATIHAN/TUGAS

1. Pelajari kinerja penanganan kesalahan dari salah satu kompiler yang Anda ketahui. Silahkan jawab pertanyaan-pertanyaan berikut ini:
   a. Apakah kompilator selalu mendeteksi kesalahan sintaks?
   b. Seberapa memadai pesan kesalahan yang dihasilkannya? Apakah mereka tepat atau tidak jelas? Apakah kemungkinan penyebab kesalahan disebutkan dengan jelas dalam istilah bahasa sumber? Seberapa baik pesan menunjukkan lokasi kesalahan?
   c. Seberapa baik kompiler pulih dari kesalahan? Apakah satu kesalahan menyebabkan banyaknya pesan?
   d. Apakah perbaikan sintaks dilakukan? Kira-kira seberapa sering perbaikan dilakukan?
2. Buatlah kelompok lalu diskusikan seberapa pentingnya pembuatan bahasa pemrograman untuk mendeteksi program yang salah pada waktu kompilasi!
3. Manakah yang lebih menguntungkan antara perbaikan kesalahan dibandingkan pemulihan kesalahan yang menurut Anda anggap paling penting:
   a. Untuk Anda secara pribadi
   b. Untuk programmer industri berpengalaman
   c. Untuk mahasiswa pemrograman pemula

   Jelaskan mengapa !

D. REFERENSI

Alfred V. Aho, Monica S., Ravi Sethi, Jeffrey D. (2007). *Compilers "Priciples, Techniques, & Tools"*, 2nd Edition. Boston, San Francisco, New York.

Jean-Paul Tremblay, Paul G. Sorenson. (1995). *"The Theory and Practice of Compiler Writing"*, United State of America.

Kenneth C. Louden. (1997). "*Compiler Construction: Principles and Practice"*.

Thomas W. Parsons. (1992). *"Introduction to Compiler Construction"*. New York: Computer Science Press.

Hari Soetanto. (2004). "*Teknik Kompilasi"*. Fakultas Teknologi Informasi Universitas Budi Luhur.

GLOSARIUM

Ad hoc adalah menerangkan suatu panitia/organisasi yang dibentuk untuk jangka waktu tertentu dalam rangka menjalankan atau melaksanakan program khusus.

Case Sensitive merupakan kasus dimana huruf besar dan huruf kecil diartikan berbeda.