PERTEMUAN 13

ANALISIS SEMANTIK DAN PEMBENTUKAN KODE (1)

A. TUJUAN PEMBELAJARAN

Pada pertemuan ini akan dijelaskan mengenai pengertian Analisis Semantik, Ide Pokok Analisis Semantik, Struktur Konkret dan Sintaks Abtrak serta Aturan Validasi dan Visibilitas. Setelah menyelesaikan materi pada pertemuan ini, mahasiswa mampu:

1. Menjelaskan pengertian Analisis Semantik.
2. Menjelaskan ide dan gagasan pokok Analisis Semantik.
3. Menjelaskan Strukture Konkret dan sintaks Abstrak.
4. Menjelaskan aturan Validasi dan Visibilitas.

B. URAIAN MATERI

1. Analisis Semantik

Analisis Semantik adalah bagian dari kompilator yang menghasilkan kode perantara dan berfungsi untuk menemukan kesalahan non-sintaksis dengan memeriksa jenis dan deklarasi pengenal.

Beberapa properti program yang diperlukan tidak dapat dijelaskan dengan tata bahasa bebas konteks. Properti ini dijelaskan oleh kondisi konteks. Dasar untuk persyaratan ini adalah aturan bahasa pemrograman untuk validitas dan visibilitas pengenalan. Autran tentang deklarasi menentukan apakah deklarasi eksplisit harus diberikan untuk sebuah pengenal, dimana ia harus ditempatkan, dan apakah beberapa deklarasi pengenal diperbolehkan.

Aturan validitas menentukan untuk pengidentifikasi yang dideklarasikan dalam program apa ruang lingup deklarasi, yaitu, pada bagian mana dari program deklarasi dapat berpengaruh. Aturan visibilitas menentukan dimana dalam cakupannya pengenal dapat dilihat atau disembunyikan. Tujuan dari pembatasan tersebut adalah untuk menegakkan konsistensi tipe dan inisialisai variabel atau atribut tertentu. Inisialisasi diterapkan untuk mencegah akses baca ke variabel yang tidak diinisialisasi yang hasilnya tidak ditentukan.

Konsistensi tipe, sebaliknya, diharapkan untuk menjamin bahwa pada saat run-time tidak ada operasi yang akan mengakses operan yang tidak kompatibel dengan tipe argumennya.

2. Ide Pokok Analisis Semantik

Pengenal adalah simbol yang dapat digunakan dalam program untuk memberi nama elemen program. Elemen program dalam bahasa imperatif yang dapat dinamai adalah modul, fungsi ata prosedur, label pernyataan, konstanta, variabel dan parameter serta tipenya. Dalam bahasa pemrograman berorientasi objek seperti JAVA, kelas dan antarmuka, bersama dengan atibut dan metodenya, juga dapat diberi nama. Dalam bahasa fungsional sepert OCAML, variabel dan fungsi sedikit berbeda secara semantik dari konsep terkait dalam bahasa imperatif, tetapi dapat dinamai dengan pengenal juga. Kelas penting dari struktur data dapat dibangun dengan menggunakan konstruktor yang pengenalnya diperkenalkan bersama dengan deklarasi tipe. Dalam bahasa logika seperti PROLOG, pengidentifikasi dapat mengacu pada predikat, atom, konstruktot data dan variabel.

Beberapa pengenal diperkenalkan dalam deklarasi eksplisit. Terjadinya pengenal dalam deklarasinya adalah kejadian yang menentukan pengenal; semua kejadian lain adalah kejadian terapan. Dalam bahasa pemrograman imperatif seperti C dan bahasa berorientasi objek seperti JAVA, semua pengenal perlu diperkenalkan secara eksplisit. Pada dasarnya, ini juga berlaku untuk bahasa fungsional seperti OCALM. Dalam PROLOG, bagaimanapun, baik konstruktor dan atom, maupun variabel lokal dalam klausa secara eksplisit diperkenalkan. Senaliknya, mereka diperkenalkan oleh kemunculan pertama secara sintasksis dalam program atau klausal. Untuk membedakan antara variabel dan atom, pengidentifikasi masing-masing diambil dari ruang nama yang berbeda. Variabel dimulai dengan huruf kapital atau garis bawah, sedangkan konstruktor dan atom diidentifikasi dengan huruf kecil di depannya. Jadi, istilah f (X, a) merupakan aplikasi dari konstruktor biner f = 2 pada variabel atom X dan atom a.

Setiap bahasa pemrograman memiliki konstruksi perlingkupan yang memperkenalkan batas-batas dimana pengenal dapat digunakan. Bahasa imperatif menawarkan paket, modul, deklarasi fungsi dan prosedur serta blok

yang dapat meringkas beberapa pernyataan dan deklarasi. Bahasa berorientasi objek seperti JAVA juga menyediakan kelas dan antarmuka yang dapat diatr dalam hierarki. Bahasa fungsional seperti OCAML juga menawarkan modul untuk mengumpulkan set deklarasi. Explisit let dan let-rec-construct memungkinkan kita membatasi deklarasi variabel dan fungsi ke bagian tertentu dari program. Diluar klausa, dialeg modern prolog juga menyediakan sistem modul.

Types adalah bentuk spesifikasi yang lebih sederhana. Jika elemen pemrograman adalah modul, *types* menentukan operasi, struktur data dan elemen program lebih lanjut mana yang diekspor. Untuk fungsi atau metode, ini menentukan jenis argumen serta jenis hasil. Jika elemen pemrograman adalah variabel program dari bahasa imperatif atau berorientasi objek, *types* membatasi nilai mana yang dapat disimpan dalam variabel.

Dalam bahasa fungsional murni, nilai tidak dapat secara eksplisit ditetapkan ke variabel, yaitu disimpan di area penyimpanan yang sesuai dengan variabel ini. Variabel disini tidak mengidentifikasi area penyimpanan, tetapi nilai itu sendiri. Jenis ariabel karenanya juga cocok dengan jenis dari semua nilai yang mungkin dilambangkan, jika elemen pemrograman adalah nilai, maka jenisnya juga menentkan berapa banyak ruang yang harus dialokasikan oleh sistem run-time untuk menyimpan representasi internalnya. Nilai bertipe *int* bahsa pemrograman JAVA, misalnya, saat ini membutuhkan 32 bit atau 4 byte, sedangkan nilai tipe double membutuhkan 64 bit atau 8 byte. Jenis ini juga menentukan representasi internal mana yang akan digunakan dan operasi mana yang akan diterapkan pada nilai serta semantiknya. Sebuah *int-value* di JAVA, misalnya, harus direpresentasikan dalam dua komplemen dan dapat dikombinasikan dengan nilai lain dari tipe int melalui operasi aritmatika untuk menghitung nilai baru dari tippe int.

Overflow diperbolehkan secara eksplisit. Untuk int-value bertanda, misalnya, dapat diwakilkan oleh satu dan juga oleh pelegkap keduanya – tergantung pada apa yang disediakan oleh arsitektur target, karenanya, efek overflow dibiarkan tidak menentu.

Oleh karena itu fragmen:

If (MX_INT + 1 = MIN-INT) printf(“Hello\n”);

Tidak selalu menampilkan Hello. Sebaliknya, kompilator diizinkn untuk mengoptimalkannya menjadi pernyataan kosong. Untuk unsigned int, dijamin untuk membungkus overflow, sehingga:

If (MAX_UINT + 1 = 0) printf(“Hello\n”);

Outputnya akan selalu Hello.

3. Struktur Konkret dan Sintaks Abstrak

Secara konseptual, input pada analisis semantik merupakan representasi dari srtuktut program yang dihasilkan oleh analisis sintaks. Representasi paling populer dari struktut ini adalah pohon parse menurut konteks bahasa bebas. Pohon ini disebut sintaks konkret dari pprogram. Tata bahasa bebas konteks untuk bahasa pemrograman berisi informasi yang penting untuk analisis sintaks, tetapi tidak relevan t\untuk fase kompilator selanjutnya. Diantaranya adalah simbol terminal yang diperlukan untuk analisis sintaksis dan penguraian, tetapi tidak memiliki nilai semantiknya sendiri seperti kunci if, else atau while. Dalam banyak tata bahasa, prioritas operator memperkenalnkan rangkaian node nonterminal yang dalam pohon parse, biasanya satu nonterminal perkedalaman prioritas. Hal-hal nonterminal ini bersama-sama dengan produksi yang bersesuaian seringkali tidak memiliki arti penting lagi setelah struktur sintakssnya diidentifikasi. Oleh karena itu penyusun sering menggunakan represenntasi sederhana dari struktut sintaksis, yang oleh karena itu disebut sintaksis abstak. Ini hanya mewakili konstruksi yang terjadi dalam program dan cakupannya.

Contoh penggunaan sintaks yang konkret dalam fragmen program:

Int a, b;

a ← 42;

b ← a ∗ a − 7;

Tata bahasa bebas pada konteks membedakan antara tingkat prioritas untuk tugas, perbandingan, penjumlahan dan operator perkalian. Yang menonjol adalah rantai panjang produksi yaitu penggantian satu nonterminal dengan yang lain, yang telah diperkenalkan untuk menjebatani perbedaan prioritas. Representasi yang lebih abstrak diperoleh jika pada langkah pertama, aplikasi roduksi rantai dihapus dari pohon sintaks dan kemudian simbol terminal yang berlebihan dihilangkan.

Grammar G dapat ditulis ulang secar sistematis menjadi:

Untuk setiap CFG G dan CFG G' tanpa dasar produksi dapat dibuat dengan mengikuti aturan sebagai berikut:

1. L(G) = L(G')
2. Jika G adalah LL(k)-grammar, maka G'.
3. Jika G adalah LR(k)-grammar, maka G'.

Bukti bahwa CFG G diberikan oleh G=(V_N, V_T, P, S).

CFG G' dengan bilangan G' = (V_N, V_T, P', S) dengan nilai V_N dan V_T yang sama.

Dimana dimulai dengan simbol S dimana P' dan G' mereupakan himpunan $A \rightarrow \beta$ dimana

$A_0 \underset{G}{\Rightarrow} A_1 \underset{G}{\Rightarrow} \ldots \underset{G}{\Rightarrow} A_n \underset{G}{\Rightarrow} \beta$ *dimana* $A = A0, \beta \notin V_N$.

Untuk beberapa kasus $n \geq 0$. Jumlah #P' pada P' bisa saja lebih besar dari nilai #P pada grammar G, tetapi tetap dalam batasan #N - #P.

Pada $R \subseteq V_N \times V_N$ menunjukkan hubungan dengan $(A,B) \in R$ *if* $A \rightarrow B \in P$. Kemudian cukup untuk menghitung refleksi dan transisi dari closure R* pada relasi ini untuk menentukan semua himpunan (A,B) dimana B dapat diturunkankan dari A dengan urutan rantai transisi yang berubah-ubah.

Contoh grammar G_1:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid Ide$

Hubungan R dan R* diberikan dengan:

R	*E*	*T*	*F*
E	*0*	*1*	*0*
T	*0*	*0*	*1*
F	*0*	*0*	*0*

*R**	*E*	*T*	*F*
E	*1*	*1*	*1*
T	*0*	*1*	*1*
F	*0*	*0*	*1*

Dengan menghapus rangkaian produksi, akan didapatkan rangkaian grammar:

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

$T \rightarrow T * F \mid (E) \mid Id$

$F \rightarrow (E) \mid Id$

Catatan:

Hasil dari gramar bukan lagi SLR(1)-grammar, tetapi tetap LR(1)-grammar.

Kesimpulan:

Duplikasi sisi kanan, yang diperkenalkan untuk dihilangkan, tidak dihitung selama proses spesifikasi grammar. Disisi lain, untuk penerapan pohon parse, aturan rantai harus dihindari. Oleh karena itu, dengan mengizinkan aturan rantau dalam grammar, tetapi mendelegasikan penghapusannya ke generator parser. Terkadang kita bisa menggunakan sintaks konkret atau sintaks abstrak tergantung mana yang lebih cocok.

4. Aturan Validasi Dan Visibilitas

Bahasa pemrograman sering memungkinkan kita menggunakan pengenal yang sama untuk beberapa elemen program dan dengan demikian memiliki beberapa deklarasi. Dengan demikian, strategi umum diperlukan untuk

menentukan kejadian yang diterapkan dari pengidentifikasi, kejain yang menentukan. Strategi ini ditentukan melalui aturan untuk visibilitas validitas. Ruang lingkup (kisaran validitas) dari kejadian yang menentukan dari pengenal X adalah bagian dari program dimana kejadian X yang diterapkan dapat merujuk pada kejadian yang menentukan. Dalam bahasa pemrograman dengan blok bersarang, validitas kejadian menentukan pengenal membentang diatas blok yang berisi deklarasi. Bahasa seperti itu seringkali mengharuskan hanya ada satu deklarasi pengenal dalam sebuah blok.

Semua kejadian yang diterapkan dari pengenal mengacu pada kejadian yang menentukan tunggal ini. Blok, dapat bersarang dan blok bersarang dapat berisi deklarasi baru dari pengidentifikasi yang telah dideklarasikan di blok luar. Meskipun deklarasi di blok luar juga valid di blok bersarag, deklarasi itu mungkin tidak lagi terlihat. Itu mungkin disembunyikan oleh deklarasi pengenal yang sama di blok dalam. Kisaran visibilitas kejadian menentukan pengenal adalah teks rogram dimana pengenal itu valid dan terlihat. JAVA, misalnya, tidak mengizinkan kita menyembunyikan pengenal lokal karena ini adalah sumber kesalahan pemrograman yang umum. Contoh program

```
for (int i ← 0; i < n; i++) {
        for (int i ← 0; i < m; i++) {
        ....
        }
}
```

Tidak mungkin di JAWAB, sebab dalam pendeklarasian *i* akan menimpa deklarasi dibagian luar sarang. Tetapi, dengan metode class, JAVA tidak seketat itu:

```
Class Test {
        Int x;
        void foo (int x) {
                x ← 5;
        }
}
```

Method foo tidak akan mengubah nilai x dari penerima, melainkan memodifikasi sebagai nilai parameter x.

Beberapa bahasa pemrograman seperti JAVA dan C++, mengizinkan penempatan deklarasi variabel pada blok pertama sebuah program.

```
{
        int y;
        ...
        int x ← 2;
        ...
        y ← x + 1;
}
```

Variabel x valid diseluruh blok, tetapi hanya terlihat setelah deklarasinya. Properti pertama mencegah deklarasi lebih lanjut dari x dalam blok yang sama. Proses identifikasi mengidentifikasi untuk setiap kejadian yang menentukan yang termasuk kejadian yang diterapkan sesuai dengan aturan validitas dan aturan visibilitas bahasa pemrograman. Validitas dan aturan visibilitas bahasa pemrograman sangat terkait dengan jenis penumpukan cakupan yang diizinkan oleh bahasa tersebut.

COBOL tidak memiliki cakupan blok sama sekali, semua pengenal valid dan terlihat dimana-mana. FORTRAN77 hanya mengizinkan satu tingkatan blok, yaitu tanpa cakupan bersarang. Prosedur dan fungsi semuanya ditentukan dalam program utama. Pengenal yang ditenntukan dalam blok hanya terlihat didalam blok itu. Pengenal yang dideklarasikan di program utama dimulai dengan deklarasinya, tetapi tersembunyi didalam deklarasi prosedur yang berisi deklarasi pengenal baru.

Bahasa imperatif dan berorientasi objek modern seperti PASCAL, ADA, C, C++ dan JAWA serta bahasa pemrograman fungsional memungkinkan penumpukan blok. Rentang validitas dan visibilitas penentuan kejadian pengenal ditetapkan oleh aturan tambahan.

let $x = e_1$ *in* e_0

pada OCAML pengenal x hanya valid dalam body e_0 dari let-construct. Kemunculan x yang diterapkan dalam ekspresi e_1 merujuk pada kmunculan x yang menentukan dalam blok yang melingkupi. Ruang lingkup pengidentifikasian x_1, ..., ..., x_n pada let-rec construct.

let rec $x_1 = e_1$ *and ... and* $x_n = e_n$ *in* e_0

terdiri dari semua ekspresi e0, e1, ..., en secara bersamaan. Aturan ini menyulitkan untuk menerjemahkan program dalam sekali jalan, ketika menerjemahkan sebuah deklarasi, *compiler* mungkin membutuhkan informasi tentang sebuah identifier yang deklarasinya belum diproses. PASCAL, ADA, dan C oleh karena itu memberikan deklarasi maju untuk menghindari masalah ini.

PROLOG memiliki beberapa kelas pengenal, yang dicirikan oleh posisi sintaksisnya. Karena tidak ada persembunyian, validitas dan visibilitas setuju. Pengenal dari kelas berbeda memiliki cakupan sebagai berikut:

a. Predikat, atom dan konstruktor memiliki visibilitas global, mereka valid diseluruh program PROLOG dan di query terkait.

b. Pengidentifikasi variabel klausa hanya valid di klausa tempat variabel tersebut muncul. Deklarasi eksplisit hanya ada untuk predikat: ini ditentukan daftar alternatifnya.

Konsep penting untuk membuat pengenal terlihat dalam konteks tertentu adalah kualifikasinya. Perhatikan ekspresi x.a dalam bahasa pemrograman C. Deklarasi tipe yang dirujuk komponen a bergantung pada variabel x. Variabel x, lebih tepatnya jenis x, berfungsi sebagai kualifikasi komponen a. Kualifikasi juga digunakan dalam bahasa pemrograman dengan konsep modul seperti MODULA dan OCALM untuk membuat pengenal publik dari modul terlihat diluar modul pendefinisian. Misalkan A menjadi modul OCAML *A.f* memanggil fungsi f yang dideklarasikan di A. Demikian pula use-directive di ADA mencantumkan pengenal cakupan sekitarnya, sehingga deklarasinya terlihat. Visibilitas pengenal ini membentang dari akhir petunjuk penggunaan hingga akhir unit program penutup. Konsep serupa untuk kualifikasi ada dalam bahasa pemrograman berorientasi objek sepertin JAVA. Pertimbangkan nama x di JAVA yang didefinisikan dikelas C sebagai publik. Dalam kelas A yang berbeda dari C dan bukan kelas manapun dalam subkelas C, pengidentifikasi x kelas C

masih valid. Pertama, pertimbangkan kasus dimana x dinyatakan sebagai statis. Kemudian x hanya ada sekali untuk seluruh kelas C. Jika kelas C milik paket berbeda dari kelas A maka identifikasi x tidak hanya membutuhkan kelas C, tetapi juga nama paket ini. Penggilan metode statis *newInstance*(), dari kelas *DocumentBuilder-Factory* dari paket *javax.xml.parsers* memiliki bentuk:

javax.xml.parsers.DocumentBuilderFactory .newInstance()

Kualifikasi yang panjang membutuhkan banyak penulisan kode. Oleh karena itu JAVA memberikan fasilitas import.

import javax.xml.parsers.

diawal file membuat semua kelas publik dari paket *javax.xml.parsers* terlihat disemua kelas file saat ini tanpa kualifikasi lebih lanjut. Panggilan dari metode statis newInstance(), maka DocumentBuilderFactory bisa disingkat menjadi:

DocumentBuilderFactory.newInstance()

Perintah:

static ***import*** *javax.xml.parsers.DocumentBuilderFactory.**

pada awal file, tidak hanya membuat *DocumentBuilderFactory* class terlihat, tetapi jugga membuat semua atribut publik statis dan metode kelas *DocumentBuilderFactory.*

Perintah serupa membuat pengenal yang valid tetapi tidak terlihat secara langsung terlihat dalam konteks sebenarnya ada di banyak bahasa pemrograman. Di OCAML, A terbuka dimodul B membuat semua variabel dan jenis modul A terlihat di modul B yang bersifat publik. Hal-hal berbeda jika bidang atau metode x dalam program JAVA tidak statis.

Kelas tempat instance pengenal x berada ditentukan dengan mempertimbangkan tipe statis dari ekspresi yang nilai run-time nya mengacu pada objek yang dipilih oleh x.

Contoh pendeklarasian class:

```
class A {                    class B extends A{
int a 1;                         int b 2;
}                                int foo() {
                                         A o new B();
                                         return 0.a;
                                 }
                             }
```

Jenis statis dari atribut o adalah A. Pada waktu berjalan, atibut o memiliki nilai objek subkelas B dari A. Terlihat dari objek yang dievaluasi hanya terlihat atribut, metode dan kelas dalam superclass A.

Kesimpulan:

Tidak semua tempat dalam lingkup kemunculan menentukan x merujuk pada kejadian yang menetukan x. Jika kejadian yang menentukan bersifat global ke blok sebenarnya, itu mungkin disembunyikan oleh deklarasi ulang class x. Maka tidak langsung terlihat. Ada beberapa kemungkinan dalam ruang lingkupnya, meskipun untuk membuat kejadian yang menentukan dari pengenal x yang tidak terlihat secraa eksplisit. Untuk itu, banyak bahasa pemrograman yang memberikan kualifikasi eksplisit pada pengguna tertentu atau arahan umum untuk menghilangkan kualifikasi eksplisit dalam konteks tertentu.

5. Tipe Inferensi

Bahasa imperatif biasanya mengharuskan kita menyediakan tipe untuk pengenal. Ini digunakan untuk mendapatkan jenis ekspresi. Dalam bahasa pemrograman fungsional modern, bagaimanapun, tidak hanya jenis ekspresi, tetapi juga jenis pengenal secara otomatis disimpulkan. Oleh karena itu pengenal baru diperkenalkan dalam program (kebanyakan) tanpa mengaitkannya dengan tipe.

Let rec fac = fun x → if $x \leq 0$ then 1

Else $x \cdot$ fac (x-1)

Operasi aritmatika untuk bilangan integral diterapkan ke argumen x dari fungsi fac. Oleh karena itu tipe argumen fac harus int. Nilai kembaliannya adalah 1 atau dihitung menggunakan operator · untuk bilangan integral. Karenanya, jenis nilai yang dikembalikan harus lagi int. Kompiler OCAML menyimpulkan bahwa fungsi fac memiliki tipe int → int, artinya fac adalah fungsi yang mengambil nilai-int sebagai argumen dan mengembalikan nilai-int sebagai hasil.

Ide untuk menyimpulkan tipe secara otomatis kembali ke J.R. Hindley dan R. Milner. Kami mengikutinya dan mengkarakterisasi himpunan tipe potensial dari ekspresi dengan memperkenalkan aksioma dan aturan inferensi, yang menghubungkan tipe ekspresi dengan tipe subekspresinya. Untuk kesederhanaan kami hanya mempertimbangkan bahasa inti fungsional, yang berasal dari OCAML. Bahasa inti fungsional serupa juga dipertimbangkan dalam volume Desain Kompilator: Mesin Virtual. Program dalam bahasa pemrograman ini adalah ekspresi tanpa variabel bebas, di mana ekspresi e dibuat menurut tata bahasa berikut:

$e ::= b \mid x \mid (x_1\ e) \mid (e_1\ x_2\ e_2)$

$\mid$ *(if* e_0 *then* e_1 *else* e_2*)*

$\mid (e_1,...,e_k) \mid [\,] \mid (e_1 :: e_2)$

$\mid$ *(match* e_0 *with* $[\,] \mid h :: t \rightarrow e_2$*)*

$\mid$ *(match* e_0 *with* $(x_1,....,x_k) \rightarrow e_1$*)*

$\mid (e_1\ e_2) \mid$ *(fun* $x \rightarrow e$*)*

$\mid$ *(let* $x_1 = e_1$ *in* e_0*)*

$\mid$ *(let rec* $x_1 = e_1$ *and ... and* $x_n = e_n$ $i_n\ e_0$*)*

b adalah nilai dasar, *x* adalah variabel, dan X_i *(i = 1, 2)* adalah operator *i*-menempatkan operator pada nilai dasar. Untuk kesederhanaan, kami menganggap sebagai tipe data terstruktur hanya tuple dan daftar. Pencocokan pola dapat digunakan untuk menguraikan data terstruktur. Sebagai pola dekomposisi, kami hanya menerima pola dengan satu konstruktor. Kami menggunakan aturan prioritas dan asosiasi biasa untuk menyisihkan tanda kurung.

C. SOAL LATIHAN/TUGAS

1. Jelaskan yang dimaksud dengan Analisis Semantik?
2. Jelaskan bagaimana cara pengecekan duplikasi pada pendefinisian nama variabel dalam suatu blok?
3. Bagaimana cara untuk mengetahui apakah nama variabel dalam suatu program sudah terdefiniskan atau belum?
4. Bagaimana memeriksa tipe sebuah variabel?
5. Jelaskan pengecekan terhadap kode

   A:= (A+B)*(C+D)

D. REFERENSI

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

Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J.H. Jacobs, Koen Langendoen. (2012). "*Modern Compiler Design"*. British Library Catalogue In Publication Data.

Kenneth C. Louden. Kenneth A. Lambert. (2011). "*Programming Languages Principles and Practices"*, Third Edition. USA.

Pattrick D. Terry (2000). "*Compilers and Compiler Generators an Introduction with C++".* Rhodes University.

Reinhard Wilhelm, Helmut Seidl, Sebastian Hack. (2013). "*Compilers Design "Syntatic and Semantic Analysis".* Sprinder-Verlag Berlin Heidelberg.

GLOSARIUM

Semantic Analysis adalah tugas untuk memastikan bahwa deklarasi dan pernyataan suatu program benar secara semantik, yaitu maknanya jelas dan konsisten dengan cara di mana struktur kontrol dan tipe data seharusnya digunakan.

Type Inference adalah pengecekan yang mengacu pada deteksi otomatis jenis `ekspresi dalam bahasa formal. Ini termasuk bahasa pemrograman, sistem tipe matematika, tetapi juga bahasa alami di beberapa cabang ilmu komputer dan linguistik.

Concrete Syntax adalah bahasa pemrograman yang ditentukan oleh tata bahasa bebas konteks. Ini terdiri dari seperangkat aturan (produksi) yang menentukan tampilan program bagi programmer.

Abstract Syntax adalah bagian dari sebuah implementasi pada sekumpulan pohon yang digunakan untuk merepresentasikan program dalam implementasi. Sintaksis abstrak mendefinisikan tampilan program bagi evaluator / *compiler*.