Aplikasi Rekursif dalam Analisis Sintaks Program

Albertus Kelvin / 13514100
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13514100@std.stei.itb.ac.id

Abstract – Makalah ini akan membahas mengenai bagaimana menerapkan sifat rekursif untuk mengecek kebenaran sintaks program, dimana pengecekan ini merupakan salah satu tahap yang dilakukan oleh compiler. Sintaks program yang memiliki beberapa identifier dan karakter umum akan dijadikan sebagai token dan dimasukkan ke sebuah list. Pengecekan sintaks secara umum dapat dilakukan dengan metode Context Free Grammar (CFG). Metode CFG yang diimplementasikan secara rekursif tidak memerlukan proses parsing. Hal lainnya adalah bahasa pemrograman yang digunakan di makalah ini adalah bahasa pemrograman dasar yang memiliki kemiripan dengan Pascal.

Kata kunci – CFG, compiler, list, parsing, rekursif, sintaks.

I. PENDAHULUAN

Pernahkah Anda membayangkan bagaimana cara sebuah compiler bahasa pemrograman bekerja? Secara umum, compiler akan melakukan 7 tahap yaitu Lexical Analysis, Syntax Analysis, Semantic Analysis, IR Generation, IR Optimization, Code Generation, dan Optimization. Namun, pada makalah ini kita hanya akan membahas mengenai Syntax Analysis, dimana yang menjadi fokus dari analisis ini adalah kebenaran tata bahasa dari semua identifier maupun operasi perhitungan.

Untuk memecahkan masalah di atas, salah satu metode yang dapat digunakan adalah teori Context Free Grammar (salah satu bidang studi di Teori Bahasa dan Otomata), dimana bagian Left Hand Side (LHS) direpresentasikan sebagai nama fungsi/prosedur atau identifier, dan bagian Right Hand Side (RHS) direpresentasikan sebagai kemungkinan hasil yang dapat dicapai dari fungsi/prosedur atau identfier di LHS.

Berdasarkan teknik representasi LHS dan RHS yang dijelaskan di atas, maka dapat digunakan metode rekursif dimana sebuah fungsi memanggil dirinya sendiri dikarenakan adanya keterkaitan antara sebuah blok program dengan blok program lainnya.

II. TEORI DASAR

A. Teori Rekursif

Suatu kondisi dinamakan rekursif jika ada keadaan dimana kondisi itu memanggil dirinya sendiri untuk melanjutkan ke kondisi tahap berikutnya. Definisi ini juga berlaku untuk fungsi dan prosedur di bidang pemrograman.

Dalam teori rekursif, ada 2 bagian penting yang mendasari cara kerja teori ini, yaitu:

1. Bagian basis → bagian ini berperan untuk membuat proses rekursif berhenti agar tidak terjadi proses rekursif terus menerus.
2. Bagian rekursif → bagian ini berperan untuk memanggil fungsi/prosedur itu sendiri dengan parameter/kondisi yang mendekati basis sehingga proses rekursif diusahakan untuk berhenti.

Berikut salah satu contoh proses rekursif dalam menghitung nilai faktorial.

```
Long int faktorial (int N) {
        long int F;
        [1] if (N <= 1) { return 1; }
        [2] else { F = N * faktorial(N-1);
        [3]         return F; }
}
```

Dari petikan program di atas, yang menjadi bagian basis adalah baris [1], yaitu jika nilai N sudah kurang atau sama dengan 1, maka nilai faktorialnya adalah 1. Kemudian, yang menjadi bagian rekursifnya adalah baris [2], dimana nilai N masih lebih besar dari 1, sehingga disimpan ke variabel F yang bernilai N itu sendiri dikalikan dengan nilai faktorial berparameter N-1. Kita dapat melihat bahwa parameter fungsi faktorial semakin mengeceil mendekati 1 (menuju basis). Hasil akhir faktorial bilangan N akan dikembalikan di baris [3] setelah bagian basis mengembalikan nilai 1.

B. Teori Context Free Grammar (CFG)

Context Free Grammar (CFG) adalah sebuah proses menghasilkan string secara rekursif dengan memanggil produksi aturan yang bersangkutan. CFG ini dapat juga dikatakan sebagai proses interpretasi/prediksi string yang ingin dibentuk dari semua kemungkinan pola dan aturan-aturan.

Sebuah CFG mengandung beberapa komponen penting yang dapat dijelaskan sebagai berikut:

1. Komponen terminal → kumpulan karakter yang terdapat dalam string yang akan dibentuk dari CFG tersebut.
2. Komponen non-terminal → komponen yang sering

dikatakan sebagai variabel, dimana nilainya dapat diubah menjadi variabel lain atau bahkan komponen terminal dengan cara dipanggil oleh komponen non-terminal lainnya.

3. Aturan produksi → sebuah *rule* untuk mengganti komponen non-terminal dengan produksi yang dihasilkan komponen non-terminal tersebut. Produksi yang dihasilkan itu dapat berupa terminal ataupun non-terminal lain.

4. Simbol *start* → sebuah simbol khusus yang digunakan pertama kali sebagai *starting point* untuk menghasilkan string.

Berikut salah satu contoh CFG yang dapat menghasilkan string “main” yang terletak di antara string lain, dimana string lain itu bisa saja kosong atau tidak kosong.

```
S → FF1 main FF1                    [1]
FF1 → FF2 FF1 | epsilon             [2]
FF2 → A | B | … | Z | a | b | … | z  [3]
```

Dari contoh CFG di atas, S menjadi simbol *start*, FF1 dan FF2 menjadi komponen non-terminal, dan A, B, … Z, a, b, … z menjadi terminal.

Aturan produksi dari S adalah S menghasilkan FF1 main FF1, dimana FF1 bisa menghasilkan FF2 FF1 atau epsilon (string kosong). Begitu pula dengan FF2 yang menghasilkan komponen terminal.

Dari CFG di atas, kita dapat melihat proses rekursif di baris [2], dimana komponen non-terminal FF1 memanggil dirinya kembali di bagian FF2 FF1. Proses rekursif FF1 akan berhenti jika FF1 memanggil epsilon (string kosong).

Proses pembentukan string “body main void” dapat dirinci sebagai berikut.

```
S → FF1 main FF1
FF1(sebelum “main”) → b FF1
        FF1 → o FF1
        FF1 → d FF1
        FF1 → y FF1
        FF1 → epsilon
FF1(setelah “main”) → v FF1
        FF1 → o FF1
        FF1 → i FF1
        FF1 → d FF1
        FF1 → epsilon
```

Maka, setelah kedua non-terminal FF1 itu dikembalikan lagi ke non-terminal S, hasilnya adalah “body main void”.

Jadi, proses CFG dilakukan dengan cara *guessing*, dimana diusahakan agar string yang diinginkan dapat terbentuk dari pola grammar yang ada.

C. Teori Sintaks

Definisi sintaks menurut situs http://literarydevices.net/syntax/ adalah kumpulan aturan yang mendikte bagaimana kata-kata dalam sebuah bahasa saling berhubungan dengan cara memperhatikan urutan sehingga bahasa tersebut dapat dikomunikasikan.

Berikut adalah contoh sintaks yang digunakan di bahasa pemrograman *Pascal-like*.

```
Begin                   [1]
    a=1+2               [2]
    if (a = 3) then     [3]
    begin               [4]
        a=a+1           [5]
    end                 [6]
    else                [7]
    begin               [8]
        a=a-1           [9]
    end                 [10]
end                     [11]
```

Dapat dilihat bahwa sintaks *Pascal-like* tersebut valid karena memenuhi aturan grammar *Pascal-like*.

Salah satu aturan sintaks di *Pascal-like* adalah setelah identifier “begin” harus diakhiri dengan identifier “end” yang dipenuhi pada baris [1]-[11], [4]-[6], dan [8]-[10].

Jenis aturan sintaks yang lain adalah yang berkaitan dengan validitas operator, yang dalam hal ini ditunjukkan pada baris [2], [5], dan [9]. Operasi matematika di Pascal dikatakan valid jika sebelum dan sesudah tanda operator (+, -, *, /, =) haruslah berupa variabel atau konastanta atau ekspresi lain yang valid.

Aturan sintaks berikutnya adalah mengenai if-else, dimana kondisi if-else yang valid adalah sebelum identifier “else” harus ada identifier “if” dan “else” hanya boleh muncul sekali setelah “if”. Aturan lainnya adalah “if” boleh muncul tanpa diakhiri “else”. Ternyata, aturan if-else tidak hanya sampai di sana, melainkan harus ada pengecekan terhadap pernyataan kondisi setelah “if”, yaitu bagian (a=3) di baris [3]. Pernyataan kondisi “(a=3)” ini adalah bagian dari aturan sintaks operasi matematika yang sudah dijelaskan sebelumnya. Selain operasi matematika yang valid, aturan tanda kurung yang seimbang harus juga terpenuhi. Kemudian, setelah pernyataan kondisi harus dicek juga apakah ada identifier “then”, yang dilanjutkan dengan pengecekan setelah “then”, yaitu harus ada identifier “begin”. Pengecekan validitas “begin” ini dilakukan serupa dengan aturan yang sudah dijelaskan sebelumnya, yaitu harus diakhiri dengan “end”, dimana dalam kasus ini hal tersebut terpenuhi dari baris [4] sampai [6], begitu pula dengan “else” yang dilanjutkan dengan “begin” - “end”.

Berdasarkan penjelasan di atas, dapat dilihat bahwa pengecekan validitas sintaks serupa dengan proses rekursif, dimana sebuah identifier memiliki kondisi valid sama seperti identifier lain.

D. Teori Token dan List

Token adalah sebuah bagian dari string yang dipecah menjadi beberapa kata. Contoh: sebuah string berisi “Aplikasi rekursif untuk analisis sintaks” dapat menghasilkan token yang berisi “aplikasi”, “rekursif”, “untuk”, “analisis”, dan “sintaks”. Tidak hanya itu, token dapat dibentuk dengan memanfaatkan sebuah karakter yang dinamanakan delimiter. Delimiter ini nantinya akan berfungsi sebagai karakter batasan yang menjadi penanda

bagi pembentukan token. Contoh pada kasus string sebelumnya, jika delimiter yang digunakan adalah karakter “s”, maka token yang dihasilkan adalah “aplika”, “i rekur”, “if untuk anali”, “i”, “ “, dan “intak”. Jadi, dapat dilihat bahwa pada pembentukan token pertama kali, delimiter yang digunakan adalah karakter “ “ (spasi).

List merupakan sebuah kumpulan elemen yang dapat bernilai sama. Komponen yang dimiliki List dapat berupa isi dari list itu dan alamat selanjutnya dari sebuah elemen list. Komponen list yang berupa isi tersebut juga dapat dipecah menjadi beberapa komponen seperti nama elemen, id elemen, dll.

III. IMPLEMENTASI

A. Pengenalan Sintaks

Dalam makalah ini, jenis bahasa pemrograman yang akan dijadikan bahan untuk implementasi adalah bahasa *Pascal-like*, yang mana artinya bahasa ini memiliki kemiripan sintaks dengan bahasa pemrograman Pascal. Pembatasan lainnya adalah bahwa kita hanya akan melakukan pengecekan validitas sintaks pada algoritma utama, tanpa melihat kebenaran sintaks pada bagian kamus atau header program lainnya.

Berikut adalah definisi sintaks yang dinyatakan valid bagi bahasa *Pascal-like.*

1) Awalan algoritma utama
   → begin ... end
   Bagian algoritma utama harus diawali identifier begin dan diakhiri identifier end, dimana di antara kedua identifier tersebut berisi sebuah blok program atau mungkin juga kosong.
2) Sintaks operasi matematika dan relasi
   → <expression> <operator> <expression>
   Bagian operasi matematika ditandai oleh identifier seperti +, -, *, dan = (operator khusus bilangan bulat). Bagian relasi dinyatakan oleh identifier <, >, <>, <=, dan >=. Sintaks yang valid untuk masing-masing operator tersebut (baik operasi matematika maupun relasi) adalah pada kedua sisi kiri dan kanannya harus berisi ekspresi yang bernilai valid juga.
3) Sintaks IF-THEN
   → if (kondisi) then
   → begin
   → …
   → end
   Rincian sintaks yang valid untuk IF-THEN adalah bahwa setelah ditemukan identifier IF, maka selanjutnya dilakukan pengecekan terhadap kondisi yang diinginkan. Misalkan: if (a=1) then, maka kondisi yang diinginkan tersebut adalah (a=1) yang mana termasuk ke dalam bagian sintaks operasi matematika. Untuk melakukan pengecekan sintaks kondisi itu, kita dapat menggunakan prosedur yang sama saat mengecek validitas sintaks operasi matematika pada poin 2. Setelah sintaks kondisi dinyatakan valid, maka selanjutnya dilakukan pengecekan terhadap ada tidaknya identifier THEN. Selanjutnya dilakukan pengecekkan apakah setelah THEN terdapat identifier begin yang diakhiri dengan end, dimana di antara begin dan end itu terdapat blok program yang mana menjadi bagian yang akan dieksekusi oleh statement IF-THEN. Melakukan pengecekan terhadap validitas begin dan end pada IF-THEN ini dapat juga dilakukan dengan prosedur yang sama seperti poin 1.
4) Sintaks IF-THEN-ELSE
   → if (kondisi) then
   → begin
   → …
   → end
   → else
   → begin
   → …
   → end
   Rincian sintaks yang valid untuk IF-THEN-ELSE sebenarnya sama seperti IF-THEN pada poin 3, hanya saja setelah menyatakan kalau blok IF-THEN adalah valid, maka dilakukan pengecekan terhadap kasus ELSE. Kasus ELSE dinyatakan valid jika setelah identifier else ditemukan identifier begin yang diakhiri oleh end. Di antara begin dan end tersebut adalah blok program yang akan dieksekusi oleh kasus ELSE tersebut. Pengecekan tambahan bagi ELSE yaitu sebelum ELSE harus ada kondisi IF-THEN, dan ELSE hanya boleh muncul sekali setelah IF-THEN.
5) Sintaks DO-WHILE
   → do
   → …
   → while (kondisi)
   Bagian DO-WHILE diproses validitasnya dengan cara setelah program menemukan identifier do, maka program akan mencari identifier while sebagai penanda akhir dari blok DO-WHILE. Di antara do dan while tersebut menjadi blok program yang akan dijalankan oleh algoritma DO-WHILE tersebut. Setelah ditemukan identifier while, maka dilakukan pengecekan terhadap sintaks kondisi while, misalnya terdapat kode “while (a <= 9)”, maka sintaks kondisi yang harus dicek adalah “(a <= 9)”. Pengecekan sintaks kondisi ini sekali lagi dapat memanfaatkan prosedur yang sama seperti poin 2.
6) Sintaks FOR

→ for (var = const to const) do
→ begin
→ …
→ end

Bagian FOR diproses validitasnya dengan cara setelah program menemukan identifier for, maka akan dilakukan pengecekan terhadap kondisi perulangan FOR. Misalkan terdapat kode "for (a = 1 to 2) do", maka sintaks kondisi perulangan yang harus dicek adalah bagian "(a = 1 to 2)", dimana perlu diperhatikan ada identifier khusus di antara sintaks operasi for tersebut, yaitu identifier to. Pengecekan sintaks kondisi for ini dapat menggunakan prosedur sama seperti poin 2 dengan modifikasi sedikit bagi identifier to. Setelah itu, program akan melihat apakah ada identifier do setelah kondisi for sebelumnya. Jika ada, maka akan dilakukan pengecekan validitas terhadap identifier begin yang harus diakhiri dengan end. Pengecekan begin dan end ini memanfaatkan prosedur sama seperti poin 1.

7) Sintaks INPUT dan OUTPUT
→ input (kondisi)
→ output (kondisi)

Bagian input dan output diproses validitasnya dengan cara setelah program menemukan identifier input atau output, maka akan dilakukan pengecekan terhadap sintaks kondisi setelahnya. Pengecekan sintaks kondisi dilakukan dengan prosedur poin 2.

B. Pembentukan CFG

Setelah mempelajari jenis sintaks yang digunakan pada bahasa *Pascal-like*, kini saatnya kita membentuk Context Free Grammar (CFG) yang sesuai dan merepresentasikan alur kerja setiap identifier.

1) CFG untuk awalan algoritma utama
→ sintaks: begin <blok> end

Cara kerja dari sintaks di atas adalah setelah menemukan identifier begin, maka akan dilakukan pengecekan terhadap isi <blok> dimana <blok> merupakan sebuah fungsi yang akan menghasilkan 1 jika bernilai true, dan 0 jika false. Jika nilai validitas <blok> bernilai true, maka dicek apakah ada identifier end setelah <blok>. Jika ada, maka keseluruhan sintaks awalan algoritma utama bernilai true. Jika ada salah satu dari bagian sintaks tersebut (begin, <blok>, atau end) bernilai false, maka keseluruhan sintaks bernilai false. Karena awalan algoritma utama ini memiliki sintaks begin dan end, maka cocok untuk digunakan sebagai *starting point* dalam pembuatan CFG secara keseluruhan dan dalam hal ini kita beri simbol START.

CFG yang dihasilkan:
START → begin BLOK end BLOK

2) CFG untuk BLOK

BLOK di sini merupakan sebuah body program, dimana isinya berupa algoritma dari identifier lain seperti if, do-while, for, input, dan output. BLOK berfungsi sebagai pencatat kemunculan identifier, dimana jika ditemukan sebuah identifier, misalnya IF, maka BLOK akan memanggil fungsi untuk melakukan pengecekan validitas terhadap IF. Setelah pengecekan IF selesai, maka BLOK akan memanggil fungsi BLOK kembali karena ada kemungkinan setelah IF masih ada algoritma dari identifier lain.

Contoh implementasi BLOK:

```
begin
        if (a <= 5) then    [1]
        begin
                a=1
        end
        if (a <= 9) then    [2]
        begin
                a=2
        end
end
```

Dapat dilihat bahwa setelah statement IF pada baris [1], terdapat statement IF lagi pada baris [2]. Saat BLOK selesai mengecek validitas IF pada baris [1], maka saat BLOK memanggil dirinya kembali (rekursif), maka ia akan melihat statement IF lain pada baris [2] dan melakukan prosedur pengecekan yang sama seperti pada IF baris [1].

Berdasarkan penjelasan di atas, tentunya setiap statement identifier memiliki fungsi yang berbeda dalam proses melakukan pengecekan validitas, sehingga perlu dibuat fungsi untuk masing-masing identifier.

CFG yang dihasilkan:
BLOK → IF BLOK | DO BLOK | FOR BLOK | INPUT BLOK | OUTPUT BLOK | EXP BLOK | NIL

3) CFG untuk IF
→ sintaks: if (kondisi) then
→ begin
→ <BLOK 1>
→ end
→ <BLOK 2>

IF di sini merupakan fungsi yang mengembalikan integer bernilai 1 (true) atau 0 (false). Fungsi IF akan melakukan pengecekan terhadap sintaks statement IF dalam 1 baris yaitu pada bagian "if (kondisi) then". Untuk melakukan pengecekan terhadap bagian "(kondisi)", maka fungsi IF akan memanggil fungsi EXP yang berperan untuk

pengecekan kondisi operasi matematika dan relasi. Jika nilai EXP adalah 1, maka dilanjutkan ke pengecekan “then”, dan jika bernilai 1 pula maka fungsi IF memanggil fungsi START untuk melakukan pengecekan “begin … end” setelah identifier “then”. Dapat dilihat bahwa pada fungsi START, akan dipanggil fungsi BLOK kembali. Fungsi BLOK ini akan dipanggil untuk mengecek validitas body program padabagian <BLOK 1>. Jika nilai <BLOK 1> adalah 1, maka fungsi BLOK yang pertama kali dipanggil akan memanggil BLOK lagi sebagai pengecek bagian <BLOK 2>.

CFG yang dihasilkan:
IF → if (EXP) then START | if (EXP) then START else START

4) CFG untuk DO
→ sintaks: do
→ <BLOK 1>
→ while (kondisi)
→ <BLOK 2>
Konsep rekursif untuk DO-WHILE sama dengan yang digunakan oleh IF sebelumnya.

CFG yang dihasilkan:
DO → do BLOK while (EXP) BLOK

5) CFG untuk FOR
→ sintaks: for (var = const to const) do
→ begin
→ <BLOK 1>
→ end
→ <BLOK 2>
Konsep rekursif untuk FOR sama dengan yang digunakan oleh IF sebelumnya.

CFG yang dihasilkan:
FOR → for (EXP) do START BLOK

6) CFG untuk INPUT – OUTPUT
→ sintaks: input (EXP) <BLOK>
→ output (EXP) <BLOK>
Konsep rekursif untuk INPUT – OUTPUT sama dengan yang digunakan oleh IF sebelumnya.

CFG yang dihasilkan:
INPUT → input (EXP) BLOK
OUTPUT → output (EXP) BLOK

7) CFG untuk EXP
→ sintaks: <exp> <operator> <exp>
Salah satu metode yang digunakan untuk mengecek validitas ekspresi adalah dengan terlebih dahulu mengecek apakah tanda kurung yang ada sudah seimbang atau belum. Jika sudah seimbang, maka lanjut ke tahap berikutnya dan jika belum seimbang maka dinyatakan bahwa ekspresi itu tidak valid.

Untuk mengecek keseimbangan tanda kurung, maka dapat memanfaatkan struktur data STACK, dimana jika program melihat ada identifier '(', maka program akan memanggil prosedur PUSH pada STACK untuk menambah sebuah elemen. Jika program melihat identifier ')', maka program akan memanggil prosedur POP pada STACK untuk menghapus sebuah elemen. Jika tanda kurung seimbang, maka setelah melakukan proses PUSH dan POP isi STACK akan menjadi kosong. Jika tidak seimbang, maka isi STACK tidak kosong.

Untuk pengecekan tahap selanjutnya, maka akan dicek apakah di sisi kiri dan kanan sebuah operator terdapat ekspresi yang valid. Misalnya ekspresi yang ada sebagai berikut: (a) + (b), maka karakter yang ada di sebelah kiri operator + adalah), dan di sebelah kanan operator + adalah (. Kita tidak perlu mengecek apakah) dan (merupakan ekspresi yang valid, karena jika tanda kurung sudah dinyatakan seimbang, maka pastilah masing-masing karakter (dan) memiliki pasangan sehingga dapat dinyatakan sebagai ekspresi yang valid.

Walaupun tanda kurung sudah seimbang, tetapi masih ada kemungkinan operasi di dalamnya tidak valid. Misalnya operasi (a +). Dalam kasus ini tanda kurung sudah seimbang, tetapi karena di samping kanan operator + bernilai) yang mana adalah bukan karakter yang valid untuk dimasukkan ke operasi penjumlahan, maka operasi itu dinyatakan tidak valid. Hal yang sama berlaku pula untuk operator lain seperti -, *, <, >, <>, <=, >=, dan =.

CFG yang dihasilkan:
EXP → PLUS | MINUS | TIMES | M | L | ME | LE | E
PLUS → char + char | char + (|) + char |) + (
MINUS → char – char | char - (|) - char |) - (
TIMES → char * char | char * (|) * char |) * (
M → char > char | char > (|) > char |) > (
L → char < char | char < (|) < char |) < (
ME → char >= char | char >= (|) >= char |) >= (
LE → char <= char | char <= (|) <= char |) <= (
E → char = char | char = (|) = char |) = (
char → A | B | … | Z | a | b | … | z

C. Pembentukan Token dan List

Konsep awal dari analisis sintaks ini adalah membentuk token yang dihasilkan dari kode sumber. Token yang dibentuk pun memiliki karakteristik yang dibedakan antara identifier dan non-identifier. Untuk karakter yang jika dibaca menghasilkan identifier, maka karakter gabungan itu dijadikan 1 token. Untuk yang non-identifier juga

dilakukan prosedur yang sama untuk membuatnya menjadi sebuah token non-identifier.

```
temp1 = strtok(strcat(str, " "),"    \n");
if(temp1 == NULL) {
    InsertLast(&LT, Alokasi("NIL", baris));
}
else {
    strcpy(ptr, temp1);
    do
    {
        if(checkRule(ptr)) {
            InsertLast(&LT, Alokasi(ptr, baris));
        }
        else {
            i = 0;
            while(ptr[i] != '\0')
            {
            temp2[0] = ptr[i];
            temp2[1] = '\0';
            InsertLast(&LT, Alokasi(temp2, baris));
            i++;
            }
        }
        temp1 = strtok(NULL,"    \n");
        if (temp1 != NULL) {
            strcpy(ptr, temp1);
        }

    }while(temp1!=NULL);
```

Gambar C.1. Pembentukan token dan Insert List
Sumber: dokumen pribadi

Gambar di atas merupakan potongan kode program untuk membentuk token dan memasukkannya ke dalam sebuah list dalam bahasa C.

Dari potongan kode tersebut dapat dilihat bahwa kita menggunakan fungsi “strtok” yang berfungsi untuk memecah string menjadi sub-string (token) dengan delimiter berupa spasi, tabulasi, dan new line.

Baris program lainnya adalah yang berhubungan dengan pengecekan apakah token yang dibentuk merupakan identifier atau bukan, dimana proses ini dilakukan dengan memanggil fungsi checkRule.

D. Fungsi START

```
1064 int START(Address *P, char s[]) {
1065     //puts("start\n");
1066     Address temp, temp1, temp2;
1067     if (strcmp(InfoToken(*P).namaToken, "begin") == 0) {
1068         //printf("FOUND begin at row %d\n", InfoToken(*P).barisToken);
1069         temp = *P;
1070         //*P = NextToken(*P);
1071         if (*P != Nil) {
1072
1073             if (BLOCK(P, "begin")) {
1074                 //printf("VALID BLOCK after BEGIN from row %d\n", InfoTo
1075                 *P = NextToken(*P);
1076                 if (*P != Nil) {
1077                     if (strcmp(InfoToken(*P).namaToken, "end") == 0) {
```

Gambar D.1. Fungsi START
Sumber: dokumen pribadi

Dari potongan kode fungsi START tersebut, kita dapat melihat pada baris 1067 bahwa ia mengecek apakah ditemukan identifier “begin”. Jika “begin” ditemukan, maka fungsi START akan memanggil fungsi BLOCK pada baris 1073 dengan parameter “P” yang merupakan alamat ditemukannya identifier begin pada list token. Parameter ke-2 yaitu string “begin” hanya menjadi penanda kalau sebelum blok program yang akan dicek nanti ada sebuah identifier bernama “begin”.

```
if (strcmp(InfoToken(NextToken(*P)).namaToken, "else") == 0) {
    if (strcmp(s, "if")==0) {
        return 1;
    }
    else {
        printf("INVALID ELSE at row %d, expected IF before ELSE\n",
        return 0;
    }
}
else {
    if (BLOCK(P, s) == 1) {
        printf("VALID BLOCK from row %d till %d\n", InfoToken(temp1)
        return 1;
    }
```

Gambar D.2. Pengecekan BLOK program selanjutnya
Sumber: dokumen pribadi

Potongan kode di atas adalah bagian kode setelah baris 1077 (kode di dalam pengecekan kondisi if (namaToken = “end”)). Dapat dilihat ada pengecekan kondisi untuk kasus namaToken sama dengan “else”. Maksudnya, jika setelah identifier “end” ditemukan (yang berarti blok program begin ... end sudah selesai dicek) program menemukan identfier “else”, maka akan ada 2 kemungkinan untuk menghasilkan nilai validitas suatu blok program. Kemungkinan pertama yaitu jika sebelum “else” itu terdapat blok “if” (ditandai oleh baris if (strcmp(s, “if”)==0)), maka fungsi START mengembalikan nilai 1 (true). Sebaliknya, fungsi mengembalikan nilai 0 (false). Pengecekan ini dilakukan karena kondisi valid untuk if-then-else memiliki keunikan tersendiri.
Baris kode lainnya yang dijalankan jika tidak ditemukan identifier “else” adalah bagian pemanggilan fungsi BLOCK dengan parameter “P” yang menjadi alamat dari token sekarang di dalam list.

Prosedur ini sesuai dengan CFG yang dirancang, yaitu:
START -> begin BLOK end BLOK

E. Fungsi BLOCK

```
738 int BLOCK(Address *P, char s[]) {
739     Address temp, temp1, temp2;
740
741     temp1 = *P;
742     if (strcmp(InfoToken(*P).namaToken, "if") == 0) {
743         *P = NextToken(*P);
744         if (*P != Nil) {
745             temp = *P;
746             //printf("found if %d\n", InfoToken(*P).ba
747             if (IF(P, s) == 1) {
```

Gambar E.1. Fungsi BLOCK
Sumber: dokumen pribadi

Seperti yang sudah dijelaskan di poin B bahwa fungsi BLOCK berfungsi sebagai media pencatat kemunculan identifier dimana kemudian fungsi BLOCK akan

memanggil fungsi yang sesuai dengan identifier yang ditemukan.

Pada potongan kode fungsi BLOCK di atas, kita dapat melihat saat fungsi BLOCK menemukan identifier “IF” di baris 742, maka akan dipanggil fungsi IF pada baris 747 dengan parameter P yang adalah alamat token setelah identifier “IF” ditemukan.

```
if (BLOCK(P, s) == 1) {
    //printf("VALID BLOCK after IF row %d AND END row %d\n",
    return 1;
}
else {
    printf("INVALID BLOCK after IF row %d AND END row %d\n",
    return 0;
}
```

Gambar E.2. Pemanggilan fungsi BLOK setelah validasi identifier
Sumber: dokumen pribadi

Potongan kode di atas adalah bagian kode di dalam pengecekan kondisi “if (IF (P, s) == 1)” pada baris 747 sebelumnya. Setelah ekspresi yang dimiliki statement IF dinilai valid (bernilai 1), maka akan dicek kembali kondisi yang dimiliki oleh blok program setelahnya.

Prosedur ini memiliki kesamaan dengan CFG BLOK untuk IF, yaitu BLOK -> IF BLOK

Metode yang dijalankan oleh fungsi BLOK ini digunakan juga untuk identifier lain seperti FOR, DO, INPUT, dan OUTPUT. Berikut beberapa contoh potongan kode programnya nya di dalam fungsi BLOCK.

```
else if (strcmp(InfoToken(*P).namaToken, "input") == 0) {
    Address temp, temp1, temp2;

    temp = *P;
    *P = NextToken(*P);
    if (*P != Nil) {
        temp1 = *P;
        if (INPUT(P, s) == 1) {
            printf("VALID INPUT from row %d\n", InfoToken(
            temp2 = *P;
```

Gambar E.3. Pemanggilan fungsi INPUT
Sumber: dokumen pribadi

Potongan kode di atas akan dijalankan jika program menemukan identifier “input” dan kemudian fungsi BLOCK akan memanggil fungsi INPUT dengan parameter P yang merupakan alamat token setelah token “input”. Setelah ekspresi yang dimiliki oleh INPUT dinilai valid, maka akan dilakukan prosedur yang sama seperti gambar E.2. untuk memanggil fungsi BLOCK kembali.

```
else if (strcmp(InfoToken(*P).namaToken, "do") == 0) {
    Address temp;
    temp = *P;
    printf("FOUND DO at row %d\n", InfoToken(*P).barisToken);
    if (DO(P, "do") == 1) {
        //*P = NextToken(*P);
        if (BLOCK(P, "do") == 1) {
            printf("VALID!!!\n");
            return 1;
        }
        else {
            printf("INVALID BLOCK after DO from row %d\n", In
            return 0;
        }
```

Gambar E.4. Pemanggilan fungsi DO
Sumber: dokumen pribadi

Potongan kode di atas untuk menjalankan pengecekan validitas ekspresi DO-WHILE dengan parameter berupa alamat ditemukannya identifier “do” tersebut. Setelah dinyatakan valid, akan dilakukan prosedur seperti gambar E.2. untuk mengecek validitas blok program selanjutnya.

Metode yang sama juga berlaku untuk pemanggilan fungsi FOR dan OUTPUT.

Lalu, bagaimana jika token yang ditemukan bukan berupa identifier? Jika demikian, dapat dipastikan kalau token itu berupa karakter yang tergolong ke dalam pengecekan kondisi ekspresi.

F. Fungsi IF – DO – FOR – EXP - INPUT – OUTPUT

```
316     if (strcmp(InfoToken(*P).namaToken, "(") == 0) {
317         *P = NextToken(*P);
318         if (*P != Nil) {
319         if (EXP(P, "if") == 1) {
320             if (strcmp(InfoToken(*P).namaToken, ")") == 0) {
321                 *P = NextToken(*P);
322                 if (strcmp(InfoToken(*P).namaToken, "then") == 0) {
323                     //printf("VALID whole (EXP) after IF from row %
324                     temp=*P;
325                     *P = NextToken(*P);
326                     //printf("finding %s %d\n", InfoToken(*P).namaT
327                     if (*P != Nil) {
328                         if (START(P, "if")) {
```

Gambar F.1. Fungsi IF
Sumber: dokumen pribadi

Potongan kode program di atas adalah salah satu bagian di dalam fungsi IF. Yang dilakukan pertama kali adalah mengecek apakah terdapat identifier ‘(‘ setelah identifier “if” sebelumnya (baris 316). Jika ditemukan token ‘(‘, maka selanjutnya dilakukan pengecekan terhadap ekspresi kondisi miliki IF tersebut lewat fungsi “EXP (P, “if”)”. Jika ekspresi kondisi tersebut valid, maka selanjutnya dilakukan pengecekan apakah terdapat token ‘)’ sebagai akhir dari kondisi ekspresi IF. Jika ya, maka dilanjutkan mengecek apakah terdapat identifier “then”. Hal ini dilakukan mengingat sintaks bagi IF adalah “if (EXP) then”. Karena setelah identifier haruslah terdapat identifier “begin” yang nantinya diakhiri oleh “end”, maka dipanggilah fungsi START pada baris 328 yang akan mengecek validitas blok

program di antara “begin” dan “end” (kita tahu bahwa fungsi START akan kembali memanggil fungsi BLOK).

```
if (strcmp(InfoToken(NextToken(*P)).namaToken, "else") == 0) {
    *P = NextToken(*P);
    printf("find else at row %d next %s\n", InfoToken(*P).barisT
    temp3 = *P;
    //*P = NextToken(*P);
    //if (*P != Nil) {
    if (NextToken(*P) != Nil) {
        *P = NextToken(*P);
        printf("NOW SYM %s %d\n", InfoToken(*P).namaToken, InfoT
        if (START(P, "if") == 1) {
            printf("VALID STARTPOINT after ELSE from row %d\n",
            return 1;
        }
```

Gambar F.2. Pengecekan identifier ELSE
Sumber: dokumen pribadi

Masih di dalam fungsi IF, setelah semua blok program IF-THEN dinyatakan valid, maka selanjutnya dicek apakah terdapat identifier ELSE setelah nya. Jika ya, maka dipanggil kembali fungsi START untuk mengecek blok program di antara “begin” dan “end” untuk ELSE. Jika tidak ada ELSE, maka dilakukan prosedur yang sama seperti gambar E.2.

Yang menarik adalah terdapat algoritma fungsi yang mirip antara beberapa jenis identifier seperti antara IF dan FOR, dimana identifier “then” pada IF disamakan dengan identifier “do” pada FOR. Setelah itu kedua identifier memiliki bagian “begin” dan “end”.

Contoh lain adalah INPUT dan OUTPUT yang memiliki kemiripan dengan IF, namun yang membedakan hanyalah ada tidaknya identifier “begin” dan “end” setelah INPUT, OUTPUT, dan IF. Berikut contoh potongan kode programnya.

```
if (strcmp(InfoToken(*P).namaToken, "(") == 0) {
    *P = NextToken(*P);
    if (*P != Nil) {
        if (EXP(P, "input") == 1) {
            if (strcmp(InfoToken(*P).namaToken, ")") == 0) {
                //printf("VALID (EXP) for INPUT from row %d\n", In
                temp1 = *P;
                //*P = NextToken(*P);
                //if (*P != Nil) {
                if (NextToken(*P) != Nil) {
                    if (strcmp(InfoToken(NextToken(*P)).namaToken,
                        *P = NextToken(*P);
                    }
                    if (BLOCK(P, s) == 1) {
```

Gambar F.3. Fungsi INPUT
Sumber: dokumen pribadi

Dapat dilihat bahwa fungsi INPUT dan IF sama-sama memanggil fungsi EXP untuk mengecek validitas ekspresi kondisi mereka. Perbedaannya hanyalah jika fungsi IF memanggil fungsi START kembali, maka fungsi INPUT langsung melakukan prosedur seperti gambar E.2.

```
if (BLOCK(P, "do") == 1) {
    *P = NextToken(*P);
    if (*P != Nil) {
        temp3 = *P;
        if (strcmp(InfoToken(*P).namaToken, "while") == 0) {
            *P = NextToken(*P);
            if (*P != Nil) {
                if (strcmp(InfoToken(*P).namaToken, "(") == 0) {
                    *P = NextToken(*P);
                    if (*P != Nil) {
                        if (EXP(P, "while") == 1) {
                            if (strcmp(InfoToken(*P).namaToken, ")") == 0) {
```

Gambar F.4. Fungsi DO
Sumber: dokumen pribadi

Potongan kode untuk fungsi DO di atas melakukan pengecekan terhadap isi blok program setelah identifier “do”. Cara kerjanya sedikit berbeda karena setelah identifier “do” tidak ada identifier “begin”. Setelah isi blok program setelah “do” dinilai valid, maka dilakukan pengecekan terhadap ekspresi kondisi yang dimiliki oleh identifier “while” dimana bentuk sintaks ekspresi kondisinya sama seperti IF, INPUT, OUTPUT, dan FOR. Setelah itu dilakukan prosedur yang sama seperti gambar E.2.

Berikut adalah potongan kode program untuk fungsi EXP yang memanfaatkan struktur data STACK.

```
/** Mengecek validitas tanda () **/
while (InfoToken(PP).barisToken == InfoToken(*P).barisToken &&
    if (strcmp(InfoToken(PP).namaToken, "(")==0) {
        Push(&ST, "(");
    }
    else if (strcmp(InfoToken(PP).namaToken, ")")==0) {
        Pop (&ST);
    }
    PP = NextToken(PP);
}
```

Gambar F.5. Cek keseimbangan tanda kurung
Sumber: dokumen pribadi

Kode di atas melakukan pengecekan terhadap keseimbangan tanda kurung di dalam suatu ekspresi. Jika ekspresi itu tidak memiliki tanda kurung, maka tentu STACK akan kosong.

```
/** Mengecek ada tidaknya TO sebagai identifier untuk FOR **/
if (strcmp(s, "for") == 0) {
    PP = GetBefore(LT, *P);
    while (InfoToken(PP).barisToken == InfoToken(*P).barisToken &&
        if (strcmp(InfoToken(PP).namaToken, "to")==0) {
            foundTO++;
        }
        PP = NextToken(PP);
    }
    if (foundTO > 1) {
        printf("INVALID identifier TO in EXP after FOR from row %d,
        errorTO = 1;
    }
    else if (foundTO == 0) {
        printf("CANNOT FIND identifier TO in EXP after FOR from row
        errorTO = 1;
```

Gambar F.6. Cek ada tidaknya identifier TO jika fungsi EXP dipanggil oleh fungsi FOR.
Sumber: dokumen pribadi

Kode di atas melakukan pengecekan terhadap ada tidaknya identifier TO jika fungsi EXP dipanggil oleh fungsi FOR. Hal ini dilakukan karena sintaks ekspresi kondisi yang dimiliki oleh FOR adalah “for (var = const to const) do”. Ada tidaknya identifier “to” mempengaruhi validitas ekspresi kondisi FOR.

Tahap yang dilakukan selanjutnya adalah mengecek validitas operasi matematika atau relasi. Hal ini dilakukan dengan meng-implementasikan CFG untuk fungsi EXP yang sudah dijelaskan pada poin B.

IV. KESIMPULAN

Context Free Grammar (CFG) untuk menganalisis sintaks program dapat diimplementasikan sebagai fungsi rekursif dimana di dalam sebuah blok program bisa terdapat blok yang menjadi sub-program dari program yang lebih besar.

V. REFERENSI

[1] http://www.programcreek.com/2011/02/how-compiler-works/
Akses: Sabtu 5 Desember 2015 jam 22.09

[2] https://polidamar.wordpress.com/bahasa-c/menghitung-faktorial-dengan-bahasa-c/
Akses: Minggu, 6 Desember 2015 jam 11.35

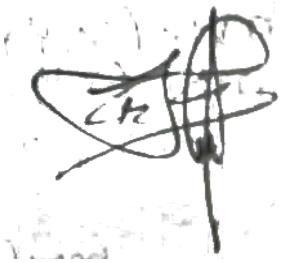
[3]https://www.cs.rochester.edu/~nelson/courses/csc_173/grammars/cfg.html
Akses: Minggu, 6 Desember 2015 jam 11.40

[4] Dokumen pribadi

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 8 Desember 2015



Albertus Kelvin / 13514100