B3　コンパイラ自動作成の1実験

藤野喜一・下村達之（日本電気中央研究所）

1. はじめに

コンパイラの作成において，Syntax処理とSemantics処理の両部分共，source programの言語文法及びオブジェクトマシンの性質にできるだけ依存しないような方式の開発は大切な問題である．このことは，コンパイラ自動作成の面に於ても中心的な問題の一つである．この報告では，ソースプログラムのparsingの手法が，ソース言語の定義によって変らないようにする為，Backus Normal Formで示されるSyntaxのProduction RulesをSyntax Graphによって表現する方式を採用した．さらにSyntax Analysisの結果得られるparsing treeとSemanti Actionsとの関係づけの方式についてのべる．実験例として小型のアルゴルをとり，この方式によるコンパイラの自動作成の結果をのべている．この実験は次の3つの過程にわけられる．

(1)　言語LのSyntax Sx(L)からSyntax Graph SG(L)を作る．この処理を行なうプログラムをP_1とかく．

(2)　Syntax Graph SG(L)を用いて，言語Lでかかれたソースプログラムpの構文解析を行ない，目的プログラム作成のためのparsing treeに変換する．これを行なうプログラムをP_2とする．

(3)　言語LのSemanticsをSyntax Graphの適当なノードを対応させた表をつくる．この表をSemantics tableという．これを参照して(2)で作ったparsing treeを目的プログラムに変換して出力する．このプログラムをP_3と呼ぶ．

以上(1)(2)(3)をまとめて図示すれば次のようになる（第1図）．

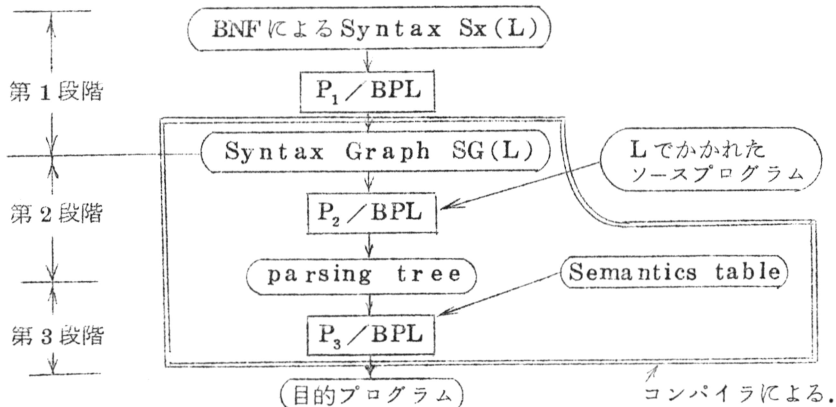


図1　全体図

2. Syntax Graphについて

2.1 Syntax Graphの作成

RNFで表わされたproduction rule

<FCT> :=<PRM> | <FCT>↑<PRM>がある.

これをグラフでは次の様に表現する(図2).

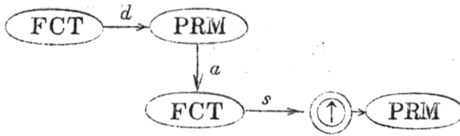


図2 Syntax Graphの表現

ここで◯はnon-terminal，◎はterminalであるsyntax elementを示す．またⓐ $\xrightarrow{d}$ ⓑはaがbではじまるグラフで定義されることを示し，ⓒ $\xrightarrow{a}$ ⓓはc及びdではじまるグラフがalternativeの関係にあることを示す．又ⓔ $\xrightarrow{s}$ ⓕならばfがeのsuccessorであることを表している．この様な意味をもつグラフのノードとアークを表示する為，次の様なデータ構造を考える．即ちノードのもつ情報を表現する為，ノードのパターンを

(Value, Definition, Alternative, Successor)

の様に定めると，第2図のグラフ表現は図3のようになる.

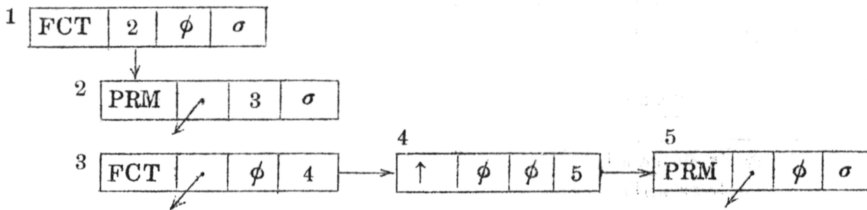


図 3

① box 2又は5におけるSuccessorの値はσになっている．これは2及び5のSuccessorがないことを示す.

② box 1, 3, 4及び5のAlternativeの値φはそれに対するAlternativeなものがないことを示す.

③ box 4のdefinitionのφは↑が他で定義されない，即ちterminalであることを示す.

2.2 グラフの簡単化

SGをできるだけ簡単にする為以下の処理がある.

(i) 共通要素のくくり出し

A → BCD | BED なる A → B(C | E)D

のように変形できるから，これをグラフ上で行なう.

例

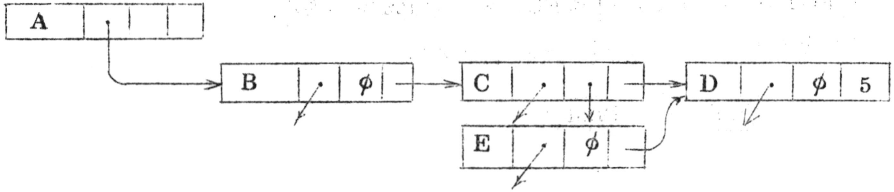


図 4

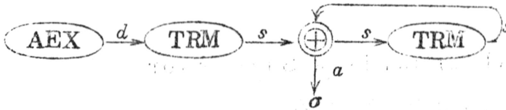
(ii) terminal のみで定義されているとき.

その nonterminal を省きその代りに，それを定義している terminal のグラフをいれる.

(iii) 再帰的な部分の簡単化

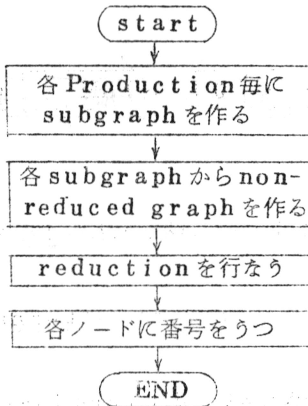
例えば AEX := TRM | AEX + TRM ならば，

AEX := TRM (+ TRM)* の様に表し，グラフでは次の様にする.



(iv) 同一内容のノードを SG からとり除く.

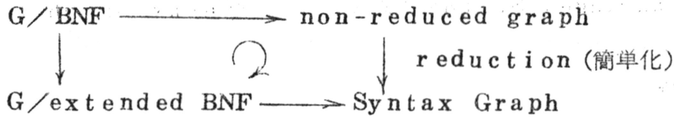
以上によって，プログラム P_1 は次の様になる.



注意 (1) グラフの簡単化はグラフの意味を変えない範囲，即ち，もとの BNF を再び構成できる範囲で行なわなければならない. その為 P_1 では (i)〜(iv) の順序で簡単化を行ない，又 (i) については前からの簡単化を優先する等の条件をつけている.

(2) BNF に記号 { }, ・・・ を導入することがある(extended BNF)があるが，この BNF

の拡張とグラフの簡単化を対応させることができる．但しextended BNFへの変換，グラフの簡単化は共に一意的なものではないが，グラフの簡単化のアルゴリズムに対応してextended BNFへの変換のアルゴリズムを作ることができる．



この図型は可換

次にこの実験で使用した言語（Small Language）のSyntaxをあげる．

<program>:=<block>

<block>:=<head> END

<head>:=BEGIN | BEGIN <dcl> | <head>; <statement>

<statement>:=<uncond-st> | <cond-st> | <for-st>

<dcl>:=<type-dcl> | <array-dcl> | <dcl>; <type-dcl> |
<dcl>; <array→dcl>

<type-dcl>:=<type><type-list>

<type>:=REAL | INTEGER | BOOLEAN

<type-list>:=<ID> | <type-list>, <ID>

<array-dcl>:=<type> ARRAY <array-list>

<array-list>:=<array-seg> | <array-list>, <array-seg>

<array-seg>:= ID〔<positive integer>〕| ID〔<positive integer>, <positive integer>〕
| ID, <array-seg>

<uncond-st>:=<ass-st> | <go-to-st> | <block>

<go-to-st>:=GO TO <ID>

<ass-st>:=<var>←<AEX> | <ID>←<B>

<AEX>:=<TRM> | +<TRM> | −<TRM> | <AEX>±<TRM>

<TRM>:=<FCT> | <TRM> */ <FCT>

<FCT>:=<PRM> | <FCT>↑<PRM>

<PRM>:=<var> | (<AEX>)

<var>:=<ID> | <ID>〔<AEX>〕| <ID>〔<AEX>, <AEX>〕

<REL>:= = | ≠ | > | <

<s-B>:=<var> | <AEX><REL><AEX>

<B>:=<s-B> | <if-cl><s-B> ELSE <B>

<if-cl>:= IF <B> THEN

<if-st> := <if-cl> <uncond-st>

<cond-st> := <if-st> | <if-st> ELSE <statement>

<for-st> := <for-cl> <statement>

<for-cl> := FOR <ID> = <AEX> STEP <AEX> UNTIL <AEX>

以上がSmall LanguageのSyntaxである．これのSyntax Graphの一部は次節2.3 図5，図6を参照．

2.2 構文解析

L(G) でかかれたソースプログラムを解析し，出力として，ノード番号とソースプログラムのword（又は空白）との対を出す過程である．構文解析のアルゴリズムはtop-down方式でBacktrackingできるようになっている．

出力中のノード番号はparsing treeに出るterminal又はnonterminalを示すが，object code生成（2.3参照）の過程では，その全てを必要とする訳ではない．従って次のコード生成の手順の為に使用するノード番号即ち2.3のSemantic tableにのっているものだけを出力すれば十分である．

ソースプログラムとそのtreeの例を次にあげる．
次節2.3の図5とSemantics tableを参照のこと．

ソースプログラム	tree	必要なノードだけとり出したtree
A+B*C $\underset{P_2}{\Longrightarrow}$	(115, A)	(115, A)
	(112,)	(101, +)
	(108,)	(115, B)
	(100,)	(109, *)
	(101, +)	(115, C)
	(115, B)	(111,)
	(112,)	(103,)
	(108,)	
	(109, *)	
	(115, C)	
	(112,)	
	(111,)	
	(103,)	

2.3 Semantics tableとCode生成

2.2で作られたソースプログラムのtreeの情報を目的プログラムに変換する過程である．はじめに言語の意味をノードに割りつけた表を作る．これをSemantics tableという．この作成はもとのBNFとSGを参照しながら作らねばならない．プログラムP_3はデータとし

てのtreeのノード列の順に従ってSemantics tableできめられている処理を行ないcodeを生成する為のものである.

Small LanguageのSemantics tableを例として以下にのせる.

Semanticsは

SMT = { (Node No, Action by P_3, Output Code) }

以下，図5，図6に示したArithmetic expression及びIf statement のSGに対応するSemantics tableの部分を例にして説明する. 2つ以上のノードNoに対して，同じP_3のActionと同じOutput Codeが一致する場合は，Node Noがまとめてかかれている. 尚 Output Codeは1アドレス方式のアセンブラ形式の言語で かかれている.

Semantics tableの例		
ノード番号	Action	Code
53	定数か変数かをみて，変数なら DCL tableを参照	
101, 102, 109, 110, 113	演算子をstack Sopにいれる. 即ち $j=j+1$ Sop(j)=operator	
103, 111, 114	codeを作る際 S(i)とS($i-1$)が 共にTの要素 $k=k-1$ 一方がTの要素 $k=k$ いずれもTの要素でない $k=k+1$	[LOD S($i-1$) OP S(i) STO T(k)]
54, 55	DCL-tableをみる	
115	operandをstack Sにいれる $i=i+1$, S(i)=operand	
117	$k=k+1$, $i=i+1$, S(i)=T(k)	[STO T(k)]
107	$k=k+1$, S(i)=T(k)	[LOD S(i) CIA T STO T(R)]
89	Lをstack IFSTにいれる $l=l+1$, IFST(l)=L	[LOD VB SAZN BRU L]
90	codeを作った後 $l=l-1$	[BRU L1 L NOP]
92		[L1 NOP]
83		[LOD VB SAZN BRU L1]
80	treeで次のノードが81かどうかみる 81なら	[BRU L2 L1 NOP]

82	81でないなら	〔L1　NOP　〕 〔L2　NOP　〕

但し，ノード番号は図5，6 参照

Codeの説明

LOD	Accumulatorにoperandを入れる.
OP	＋，－，×，÷，べき
STO	メモリーにいれる.
SAZN	Accの内容が0でないなら命令を1つとびこす.
BRU	実行順序を指定された命令に変更する
CIA	Accの内容の符号をかえる.
NOP	無機能

Sop, S, Tはstack, VBは真か偽かを示す変数

注意　DCL tableとはBlock毎に作られる変数の表である.

変数名，その属性，配列かどうか，再定義かどうか（より大きいBlockで既に宣言されているかどうか）等を記録している．このtableをソースプログラムのBlock構造に従って結びつけ，この表の並びを参照することにより，プログラム内の変数が正しく使用されているかどうかを判別する.

例　Small Languageでかかれたソースプログラムからcode生成した例

①

		tree		code	
A＋B＊C	$\xrightarrow[P_2]{}$	(115, A)	$\xrightarrow[P_3]{}$	LOD	B
		(101, ＋)		MULT	C
		(115, B)		STO	T1
		(109, ＊)		LOD	A
		(115, C)		ADD	T1
		(111,)		STO	T1
		(103,)			

② IF-statementの例

ソースプログラム		tree			code	
IF B THEN st1	$\xrightarrow[P_2]{}$	(85, IF)		$\xrightarrow[P_3]{}$	⋮	}Bのcode
		⋮	}Boolean Bのtree			
		(86,)			〔 STO VB 〕	
		(87, THEN)			〔 LOD VB SAZN BRU L 〕	
		⋮	}st1のtree			
		(84,)			⋮	}st1のcode
					〔L NOP 〕	

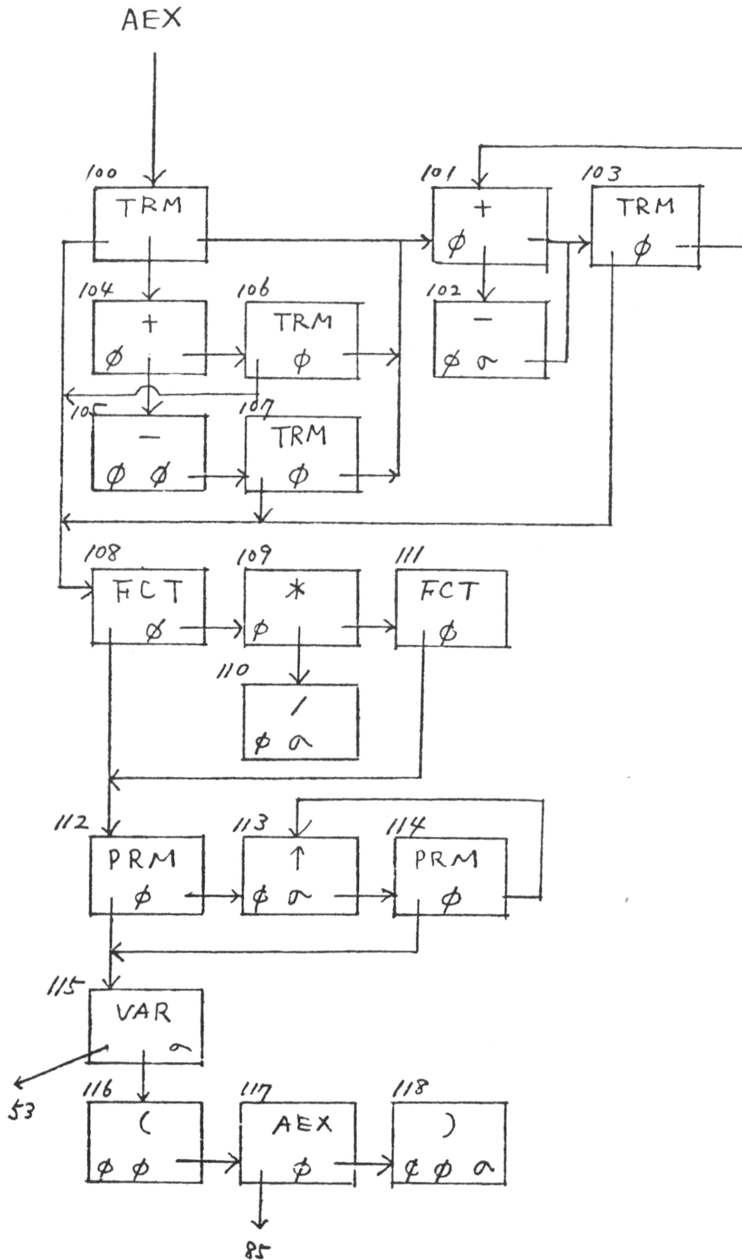


図5　Arithmetic Expression (AEX) の Syntax Graph

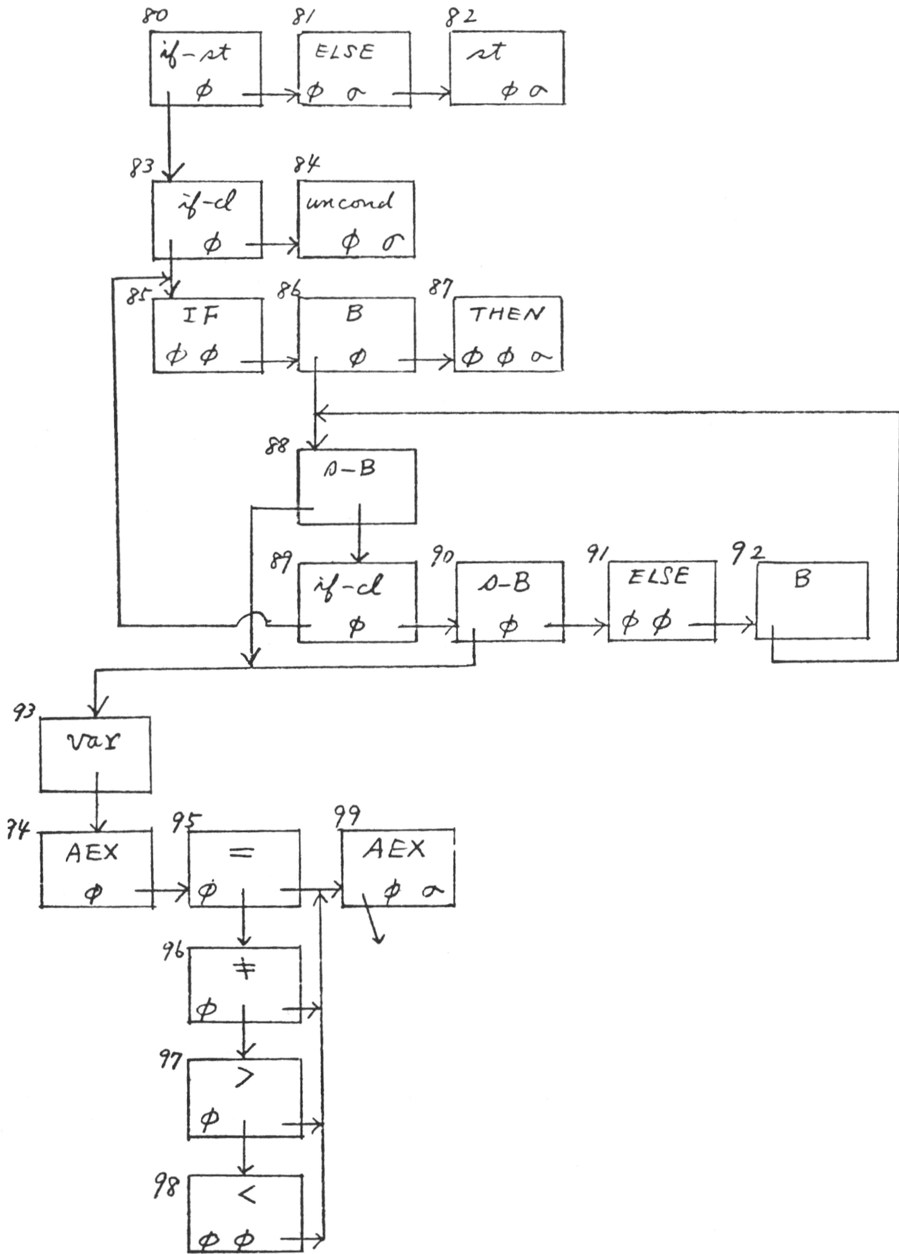


図6　If-statement (If-st) の Syntax Graph

3. あとがき

本実験では最終のOutputの目的プログラムの言語をアセンブラー形式の1アドレス言語を用いたが，よりmachine independentなもの，例えばtriple記法を使用し，目的codeはそのtripleから変換出力させる方式も考えられる．この場合2.3の表(Semantics table)もより簡単なものになると思われる．

文法のBNF記法をSyntax Graphの形に直すことの利点は構文解析のアルゴリズムのflowが簡単になること，又BNFで各production ruleに対応する意味をグラフ上では，BNFのproduction rule内のterminal，non-terminalに対応するノード毎に割りつけることができることである．ここではそのことを使っている．しかしながら，数式処理等にみられるBNFの冗長性はグラフにも残っている．従って数式処理法としてはoperator precedence法等の方がすぐれている様に思われる．おわりに，常に暖く御指導いただく日本電気中央研究所，渡部コンピュータサイエンス研究部長に感謝致します．

参考文献

A List Structure Form of Grammars for Syntactic Analysis
D.J.Cohen and C.C.Gotlieb
(Computing Surveys Vol 2. NO.1, March, 1970)

A Formal Semmantics for Computer Languages and its Application In a Compiler-Compiler
J.A.Feldman
(CACM Vol 9, NO.1, Jan. 1966)

Translator Wrighting Systems
J.A.Feldman and David Gries
(CACM Vol 11, NO.2, Feb.1968)

Compiling Techniques
Hopgood
(MACDONALD)

本 PDF ファイルは 1971 年発行の「第 12 回プログラミング―シンポジウム報告集」をスキャンし、項目ごとに整理して、情報処理学会電子図書館「情報学広場」に掲載するものです。

この出版物は情報処理学会への著作権譲渡がなされていませんが、情報処理学会公式 Web サイトの https://www.ipsj.or.jp/topics/Past_reports.html に下記「過去のプログラミング・シンポジウム報告集の利用許諾について」を掲載して、権利者の捜索をおこないました。そのうえで同意をいただいたもの、お申し出のなかったものを掲載しています。

過去のプログラミング・シンポジウム報告集の利用許諾について

情報処理学会発行の出版物著作権は平成 12 年から情報処理学会著作権規程に従い、学会に帰属することになっています。

プログラミング・シンポジウムの報告集は、情報処理学会と設立の事情が異なるため、この改訂がシンポジウム内部で徹底しておらず、情報処理学会の他の出版物が情報学広場 (=情報処理学会電子図書館) で公開されているにも拘らず、古い報告集には公開されていないものが少からずありました。

プログラミング・シンポジウムは昭和 59 年に情報処理学会の一部門になりましたが、それ以前の報告集も含め、この度学会の他の出版物と同様の扱いにしたいと考えます。過去のすべての報告集の論文について、著作権者（論文を執筆された故人の相続人）を探し出して利用許諾に関する同意を頂くことは困難ですので、一定期間の権利者捜索の努力をしたうえで、著作権者が見つからない場合も論文を情報学広場に掲載させていただきたいと思います。その後、著作権者が発見され、情報学広場への掲載の継続に同意が得られなかった場合には、当該論文については、掲載を停止致します。

この措置にご意見のある方は、プログラミング・シンポジウムの辻尚史運営委員長 (tsuji@math.s.chiba-u.ac.jp) までお申し出ください。

加えて、著作権者について情報をお持ちの方は事務局まで情報をお寄せくださいますようお願い申し上げます。

期間：2020 年 12 月 18 日 ～ 2021 年 3 月 19 日
掲載日：2020 年 12 月 18 日

プログラミング・シンポジウム委員会

情報処理学会著作権規程
https://www.ipsj.or.jp/copyright/ronbun/copyright.html