コンパイラのテストデータ網羅性判定ツール C-GRAM†

上原憲二†† 堀川博史†† 大川勉††
高野彰†† 春原猛††

コンパイラのテストデータが十分であるかどうかを言語仕様に基づいて判定するために，言語の文脈自由文法に基づくテストデータ網羅性判定手法とそれを支援するツールC-GRAMを提案する．C-GRAMは，構文上正しいテストデータを書くのに，BNFで定義された言語の構文規則の各部分が何度適用されたかを測定する．測定は段階的に詳細化する形で行われ，その測定結果によりテストデータの網羅性を判定する．実際に開発されたコンパイラのテストデータを用いてC-GRAMの有効性を評価した．その結果，不注意によるテストデータ漏れの発見を容易にする，テストデータのレビューを容易にする等の効果が確認された．C-GRAMはコンパイラのテストデータ網羅性に対する定量的な確認手段を与えるものとして有効である．

1. まえがき

ソフトウェア開発において，プログラムの高信頼性を保証するテストを効率よく行うことは，重要な課題の一つである．これを左右する一つの大きな要因はテストデータの適切な選択[1]である．すなわち，妥当で信頼できる選択基準を過不足なく満足するテストデータを選択することである．このとき，テストデータの選択基準を満たす度合をテストデータの網羅性という．テストデータの網羅性を保証するアプローチとしては次の2通りある．一つは，網羅性を保つようにテストデータの自動生成を行うことである．他方は，人手で作られたテストデータが網羅性を保っているか検査することである．

本論文ではコンパイラの言語仕様に基づくテストデータの網羅性判定手法を示す．コンパイラは重要な基本ソフトウェアであり，その開発に際しては高信頼性を保証するテストを行わねばならない．また，複雑なプログラムでもあり，そのようなテストを行うためにテストデータの選択は非常に重要である．このため，従来コンパイラのテストではテストデータの選択に多大な労力が払われ，大量のテストデータが作られてきた．しかし，人手作業のためテストデータの重複や漏れなどの問題がある．さらに，これらの問題を発見するためのレビューも人手作業であり困難なものである．

これに対処するため，種々のテストデータ自動生成の手法[2),3)]が研究開発されてきた．これらの手法は，コンパイラが対象とする言語の形式的に定義された文法に基づいてテストデータを生成する．そこでは，文法に含まれる構文規則と一部の意味規則のおのおのを少なくとも1回適用し，かつテストデータ量を最小にすることを基本としている．これらの手法の問題は，生成されたテストデータが必ずしも実行可能なプログラムでないことである．また，生成されたテストデータは特定の意味をもっていないためテスト実行結果（コンパイル結果）の検査が困難である．

上で述べたように，コンパイラのテストデータ自動生成はまだ実用化の域に達しておらず，テストデータの作成は現在人手で行うのが通常である．そこで，人手で作成されたテストデータの網羅性判定を行う必要がある．テストデータの網羅性判定手法は，プログラムの構造に基づく手法とプログラムの機能仕様に基づく手法に分けられる．前者に属するもので最近広く使われている手法の一つにMiller[4)]の提案によるものがある．これは，与えられたテストデータを用いてプログラムを実行したとき，プログラムの制御のすべての分岐のうち少なくとも1回通過された分岐の割合でそのテストデータの網羅性を表す．しかし，プログラムの構造に基づく手法では与えられたテストデータがプログラムの機能仕様全体をテストしているか判定することはできない．

そこでわれわれは，文脈自由文法においてBNFで定義された言語の構文規則に基づくコンパイラのテストデータの網羅性（構文的網羅性と呼ぶ）判定の手法

† C-GRAM: A Coverage Checker for Compiler Test Data by KENJI UEHARA, HIROSHI HORIKAWA, TSUTOMU OHKAWA, AKIRA TAKANO and TAKESHI SUNOHARA (Information Systems & Electronics Development Laboratory, Mitsubishi Electric Corporation).

†† 三菱電機(株)情報電子研究所

とそれを支援するツールC-GRAM (Coverage checker based on GRAMmar) を提案する．この手法では，BNF で記述された構文規則に従ってテストデータを直構文解析し，その際の構文規則の各部分の適用回数を測定する．この適用回数の測定は段階的に詳細化する形で行われる．したがって，ユーザはこれらの適用回数をもとにテストデータの網羅性判定を段階的に行うことができる．さらに，C-GRAM は構文規則を BNF で定義し字句解析プログラムを作れば任意のプログラム言語に適用することができる．

2章で構文的網羅性の判定方法について述べる．3章では効果的な網羅性判定を行うための手順を示し，4章でその手順を支援するツール C-GRAM の構成を示す．5章では C-GRAM の評価について述べる．

2. 網羅性判定方法

コンパイラに入力される原始プログラムを書くのに使用する言語の構文規則は**図1**に示す BNF により定義する．$(\cdots)^+$ は1回以上の繰返しを，$(\cdots)^*$ は0回以上の繰返しを意味する．以下，図1の BNF を標準形と呼ぶ．一般に，BNF は**図2**(a)に示す簡便記法を含むが，その展開形（図2(b)）の形（組合せ）で適用回数を見たほうが網羅性判定がより的確になる．

構文的網羅性判定のため，構文規則の各部分に対して，すべての構文上正しいテストデータ（原始プログラムコード）を書くのにその部分が何度使用されたかを示す適用回数の尺度を定義し，それによる網羅性判定方法について述べる．この尺度は，段階的な網羅性判定のため，生成規則の詳細化の程度に対応して定義される．

生成規則の詳細化とは，その右辺に現れる各非終端記号をそれに対する生成規則（もとの生成規則の下位の生成規則と呼ぶ）の右辺で置き換え，そしてそれを標準形にすることを意味する（**図3**参照）．

詳細化の程度を表すために生成規則の階数 k を次のように定義する．初めに定義された生成規則の階数を0とする．$k-1$ 階の生成規則について，0階の下位の生成規則の右辺で置き換えたものを k 階の生成規則とする．また，k 階の生成規則に含まれる式，項および因子の階数も k と定義する．

生成規則，項および繰返し部（これらを総称してセグメントと呼ぶ）に対し，適用回数の尺度 N_x^k を次のように定義する．繰返し部とは繰返しを表す因子のことである．

```
構文規則   ::= (生成規則)+.
生成規則   ::= 非終端記号 ::= 式.
式         ::= 項 ( | 項)*.
項         ::= (因子)+.
因子       ::= 非終端記号 | 終端記号 |
               (式)+ | (式)*.
非終端記号 ::= 識別子.
終端記号   ::= "文字列".
```

図1 BNF の構文
Fig. 1 Syntax of BNF.

```
A ::= {U|V}B[W].
```
(a) 簡便記法

```
A ::= UBW|UB|VBW|VB.
```
(b) 展開形

図2 簡便記法の展開
Fig. 2 Expansion of concise notations.

```
A ::= XY.
X ::= X1|X2.
Y ::= Y1|Y2.
```
(a) 0階の生成規則

```
A1 ::= X1Y1|X1Y2|X2Y1|X2Y2.
```
(b) Aの1階の生成規則

図3 生成規則の詳細化
Fig. 3 Detailed production rule.

N_p^k: k 階の生成規則 p の適用回数

N_t^k: k 階の項 t の適用回数

$N_{\alpha+1}^k, N_{\alpha+2}^k$: k 階の繰返し部 $(\alpha)^+$ のそれぞれ1回または2回以上の繰返しによる適用回数

$N_{\alpha*0}^k, N_{\alpha*1}^k, N_{\alpha*2}^k$: k 階の繰返し部 $(\alpha)^*$ のそれぞれ0回，1回または2回以上の繰返しによる適用回数

この尺度に基づく構文的網羅性判定方法を次に示す．

(1) 未テスト部分の判定

0階の尺度 N_x^0 ($x=p, t, \alpha+1, \alpha+2, \alpha*0, \alpha*1$ または $\alpha*2$) の値が0ならば，対応するセグメントは未テストである．0階のすべてのセグメントについてそれを適用するテストデータが存在することは，適切なテストデータ選択の最低条件である．同様に，N_x^k ($k \geqq 1$) の値が0ならば，対応するセグメントは未テストである．たとえば図3(a)で示されるAの0階の生成規則のすべてのセグメントに対して $N_x^0 \neq 0$ であっても図3(b)の1階の生成規則では $N_x^1 = 0$ であるようなセグメントが存在しうる．

(2) 意味規則を考慮した判定

N_x^k の値が0でない場合，その値が妥当なものかを判定するのに各生成規則に対応する意味規則による場

合分け数を考慮する．たとえば，GO TO 文の生成規則が

GOTO 文 ::= "GO" "TO" 識別子 ";".

の場合に，意味規則を考慮すると識別子が同一ブロック内の名札のとき，異なるブロックの名札のとき，異なる手続きの名札のときのおのおので意味が異なってくる．すなわち，2番目の場合はたんに名札で示される場所に実行制御が移るのみならず，ブロックの終了が伴うし，同様に3番目の場合は手続きの終了が伴う．したがって上記生成規則の尺度 N_p^0 の値は少なくとも3以上でなければならない．また，名札変数が許されている場合にはもっと場合分け数が増える．このような場合分け数より対応する尺度の値が小さいときはテストが不足している．ただし，尺度 N_x^k は構文規則だけを考慮した適用回数であり，上でたとえ $N_p^0=3$ であっても三つのテストデータがすべて同一ブロック内の名札への GO TO 文であるかもしれないので，完全に網羅性を保証することはできない．

3. 網羅性判定手順

構文規則の各セグメントの複雑度 C_x^k（付録参照）および尺度 N_x^k をツールを用いて測定し，前章で述べた方法に従ってテストデータの構文的網羅性を判定する．より効果的な判定を行うため次の手順に従う．

[手順 1] 構文規則の調整

まず，対象言語の構文規則を定義しなければならない．一般に言語仕様書には種々の形式の BNF を用いて構文規則が定義されており，それを2章で述べた標準形に変換する．ただし，構文規則の入力を容易にするために簡便記法を標準形に展開する支援があることが望ましい．また，言語仕様書での構文規則の定義は，読者が理解しやすいような形式で定義されており，構文的網羅性判定のためには必ずしも適当ではない．そのため以下のような構文規則の調整を行う．

（1） 生成規則の複雑度の低減

生成規則 p によってはその複雑度 C_p^k の値が k に関し指数関数的に増大する．C_p^k の値は p の k 階の項の数を示している．C_p^k の値が大きくなると，p の k 階の項に対する尺度 N_t^k の測定費用が大きくなり，場合によっては測定不可能になる．この場合，中間的な生成規則の導入により C_p^k の値を小さくする．図4の例では，X, Y, U と V の生成規則の導入により C_{pA}^1 の値は400から4に減少している．

（2） 生成規則の厳密化

言語仕様書では構文上許されない事項を BNF でなく文章で記述している場合があり，生成規則の詳細化により構文上許されない項が作られる場合がある．たとえば，GO TO 文の生成規則が

GOTO 文 ::= "GO" "TO" 参照 ";".
参照 ::= 識別子 [添字指定].

で，GO TO 文の参照は名札定数か名札変数参照であり，名札変数は配列宣言できないことが文章で示されていたとする．その場合，生成規則の詳細化によって作られる項

"GO" "TO" 識別子 添字指定 ";"

は許されない項である．このような項はできるだけ取り除いた生成規則を定義する必要がある．なぜならば，その項に対するテストデータは不要であるにもかかわらず，未テストセグメントであるという誤った判

```
A ::= BC.
B ::= X1|X2|X3|…|X10|
      Y1|Y2|Y3|…|Y10.
C ::= U1|U2|U3|…|U10|
      V1|V2|V3|…|V10.
```
（a） 調整前

```
A ::= BC.
B ::= X | Y .
C ::= U | V .
X ::= X1|X2|X3|…|X10.
Y ::= Y1|Y2|Y3|…|Y10.
U ::= U1|U2|U3|…|U10.
V ::= V1|V2|V3|…|V10.
```
（b） 調整後

図 4 中間的生成規則の導入
Fig. 4 Introduction of intermediate production rules.

```
代入文     ::= 変数 "=" 式 ";".
           ⋮
配列要素   ::= 変数名 "(" 式 ")".
           ⋮
式         ::= 項 (加減演算子 項)*.
項         ::= 因子 (乗除演算子 因子)*.
因子       ::= 変数 | 定数 | "(" 式 ")".
```
（a） 調整前

```
代入文     ::= 変数 "=" 代入式 ";".
代入式     ::= 代入項 (加減演算子 代入項)*.
代入項     ::= 代入因子 (乗除演算子 代入因子)*.
代入因子   ::= 変数 | 定数 | "(" 代入式 ")".
           ⋮
配列要素   ::= 変数名 "(" 添字式 ")".
添字式     ::= 添字項 (加減演算子 添字項)*.
添字項     ::= 添字因子 (乗除演算子 添字因子)*.
添字因子   ::= 変数 | 定数 | "(" 添字式 ")".
           ⋮
```
（b） 調整後

図 5 生成規則の複製
Fig. 5 Copies of production rules.

定をされてしまうからである．

（3） 生成規則の複製

一般に非終端記号はいくつかの生成規則の右辺で参照される．それら参照間にはその文脈に依存して意味規則の違いがある場合がある．意味規則の違いがある場合は，その非終端記号の生成規則および下位の生成規則に対する尺度は，各参照ごとに測定されることが望ましい．このため，それら生成規則を複製することによって，各参照ごとに専用のものを作る．たとえば，**図5**は，代入文の右辺の式，配列の添字式に対し別々に尺度を測定するために，専用の式，項と因子の生成規則を複製した場合を示している．

［手順 2］ 導出木の生成

手順1で作成された構文規則に従ってすべての構文上正しいテストデータを直構文解析し，それらの導出木を生成する．

［手順 3］ 尺度の測定と網羅性判定

まず，すべての導出木を走査し，0階の尺度 N_x^0 をすべての生成規則，項および繰返し部について測定する．そして，2章で述べた判定方法によって未テスト部分があるかどうか，またテストが不足しているかどうか判定する．さらに，生成規則の複雑度が階数 k とともに大きくなる場合は，k を段階的に上げながら k 階の項の尺度 N_x^k $(k \geqq 1)$ を測定し判定する．これは，N_x^k の測定費用が許される範囲で行う．

4. C-GRAM の構成

コンパイラのテストデータ網羅性判定ツール C-GRAM の構成は，**図6**に示すように，3章で述べた各手順を支援するようになっている．

図6において，モジュール①は手順1を支援する．ユーザは①の出力する非終端記号と終端記号の相互参照リスト，生成規則の複雑度 C_p^k の情報を見ながら構文規則の調整を行い，構文規則の変更を①に入力する．

モジュール②～④は手順2を支援する．テーブル生成モジュールが作るテーブル類は直構文解析のための

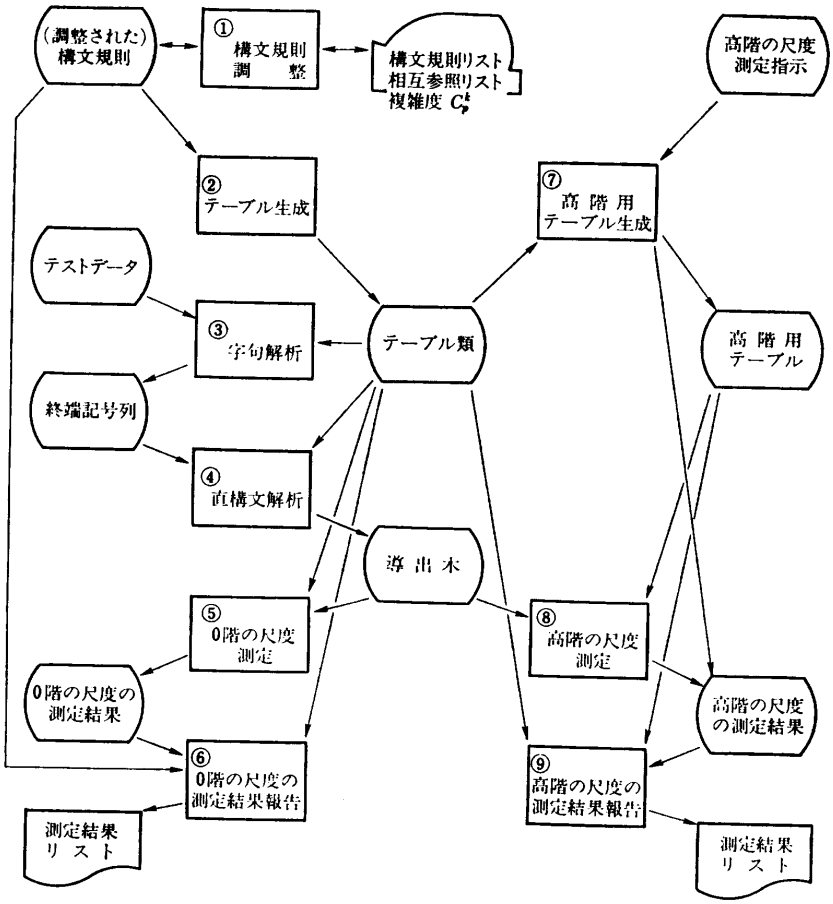

図 6 C-GRAM の構成
Fig. 6 Structure of C-GRAM.

構文規則の内部表現である．

モジュール⑤～⑨は手順3を支援する．⑤と⑥は0階の尺度 N_x^0 を測定する．⑦～⑨は，ユーザの指定した生成規則および階数に対して尺度 N_i^k ($k \geqq 1$) を測定する．

この構成からわかるように，C-GRAM は構文規則を標準形の BNF で記述し，字句解析プログラム（モジュール③）を作りさえすれば，任意のプログラム言語に適用することが可能である．

5. 評　　価

網羅性判定手法と支援ツール C-GRAM を実際のコンパイラ開発の総合テスト段階で使用されたテストデータに適用して，その有効性の評価を行った．

コンパイラは PL/I 型の言語に対するものである．テストデータ（原始プログラムコード）は，初版のコンパイラに対するもので，言語仕様書の各章各節ごとにそこに記述されている言語機能を網羅するよう作られた．作成されたテスト用の原始プログラムの総行数（注釈行を含む）は51737行である．

構文規則の調整では，生成規則の複雑度の低減，生成規則の複製によって新しい生成規則がそれぞれ34個，36個追加された．生成規則の厳密化により，生成規則の詳細化によって作られる構文上許されない項が7個削除された．全体として生成規則の数は言語仕様書で定義された101個から170個に増えた．

C-GRAM がテーブル生成に費した CPU 時間は MELCOM-COSMO 900 II 計算機で1分41秒であった．すべてのテストデータの字句解析および直構文解析による導出木生成には15分28秒費した．今回採用した直構文解析アルゴリズムは最も単純なものであり，より効率的なアルゴリズムを用いれば導出木生成にかかる時間を短縮できる可能性がある．導出木走査による0階の尺度 N_x^0 ($x=p, t, \alpha+1, \alpha+2, \alpha*0, \alpha*1$ および $\alpha*2$) の測定（**図7**）には1分5秒費した．また，複雑度が適当な大きさである二つの生成規則を選びそれらに対する1階の尺度 N_i^1 を測定した（**図8**）．それぞれの1階の生成規則の複雑度 C_p^1 は600，2310である．したがって，あわせて2910個の N_i^1 を測定したことになる．この測定時間は3分23秒であった．

測定された尺度のデータによる上記テストデータの構文的網羅性と C-GRAM の有効性について以下に述べる．

```
DECLARE文 ::= "DECLARE" ﾚﾍﾞﾙ 宣言
3451          513
                    ( "," ﾚﾍﾞﾙ 宣言 | "," 宣言 )* ";" |
                      3787            27
                                              0-0
                                              1-1
                                              2-512
              "DECLARE" 宣言
              2938
                    ( "," ﾚﾍﾞﾙ 宣言 | "," 宣言 )* ";" .
                      306             3397
                                              0-2272
                                              1-213
                                              2-453
宣言          ::= 識別子 次元 ( 属性 )*                    |
17399             1867        4172
                                  0-61
                                  1-433
                                  2-1373
                  識別子 ( 属性 )*                         |
                  14076   18256
                            0-4475
                            1-4394
                            2-5207
                  "a" 次元 ( 属性 )*                       |
                  4           3
                                  0-1
                                  1-3
                                  2-0
                  "a" ( 属性 )*                            |
                  100   117
                            0-16
                            1-52
                            2-32
                  "(" 宣言 ( "," 宣言 )* ")" ( 属性 )* .
                  1352       5079             3225
                                  0-0              0-2
                                  1-218            1-259
                                  2-1134           2-1091
ﾚﾍﾞﾙ          ::= 算術定数 .
4606              4606
```

(注)　生成規則の左辺下の数字は N_p^0 を，'0－', '1－' および '2－' が付けられた数字はそれぞれ $N_{\alpha*0}^0$, $N_{\alpha*1}^0$ および $N_{\alpha*2}^0$ を，他のものは N_i^0 を表す．

図7　0階の尺度 N_x^0 の出力例
Fig. 7　Sample output of N_x^0.

(1)　$N_p^0=0$ である生成規則が5個，$N_x^0=0$ ($x=t, \alpha+1, \alpha+2, \alpha*0, \alpha*1$ または $\alpha*2$) であるセグメントを含む生成規則が54個存在した．

それら未テストセグメントのうち3個は不注意によるテストデータ漏れ（テスト項目としてあげられていたが，テストデータ作成のとき欠落した）であった（たとえば図7の5行目）．それに対しテストデータを追加しテストを行った結果，1個の不具合が発見された．

未テストセグメントのうち22個のものは，構文規則の調整における生成規則の複製によって検出されたものである．今回の評価では，代入文の右辺の式，配列要素の参照における添字式など4種類の式についておのおの専用の式の生成規則を複製し測定した（**図9**）．その結果，代入文の右辺の式以外の式において未テストセグメントが存在し，各尺度の値も代入文の右辺の式に対するそれの1/2以下であった．このように，式に対するテストデータは代入文の右辺のものが多く作られており，他の三つの式のテストデータが少ないことが明らかになった．4種類の式では必ずしも意味規則は同じではないので，コンパイラの内部処理をブラックボックスとしてテストを行う立場からは，この結果はテストデータの不足を示しており，より適切なテストデータの選択を行わせるきっかけを与えて

```
  2  DECLARE ( 宣言 , 宣言 ) ;
     DECLARE ( 宣言 , 宣言 ) , 算術定数 識別子 次元 ;
     DECLARE ( 宣言 , 宣言 ) , 算術定数 識別子 次元 属性 ;
     DECLARE ( 宣言 , 宣言 ) , 算術定数 識別子 ;
     DECLARE ( 宣言 , 宣言 ) , 算術定数 識別子 属性 ;
     DECLARE ( 宣言 , 宣言 ) , 算術定数 * 次元 ;
     DECLARE ( 宣言 , 宣言 ) , 算術定数 * 次元 属性 ;
     DECLARE ( 宣言 , 宣言 ) , 算術定数 * ;
     DECLARE ( 宣言 , 宣言 ) , 算術定数 * 属性 ;
     DECLARE ( 宣言 , 宣言 ) , 算術定数 ( 宣言 ) ;
     DECLARE ( 宣言 , 宣言 ) , 算術定数 ( 宣言 ) 属性 ;
     DECLARE ( 宣言 , 宣言 ) , 算術定数 ( 宣言 , 宣言 ) ;
     DECLARE ( 宣言 , 宣言 ) , 算術定数 ( 宣言 , 宣言 ) 属性 ;
     DECLARE ( 宣言 , 宣言 ) , 識別子 次元 ;
     DECLARE ( 宣言 , 宣言 ) , 識別子 次元 属性 ;
     DECLARE ( 宣言 , 宣言 ) , 識別子 ;
     DECLARE ( 宣言 , 宣言 ) , 識別子 属性 ;
     DECLARE ( 宣言 , 宣言 ) , * 次元 ;
     DECLARE ( 宣言 , 宣言 ) , * 次元 属性 ;
     DECLARE ( 宣言 , 宣言 ) , * ;
     DECLARE ( 宣言 , 宣言 ) , * 属性 ;
     DECLARE ( 宣言 , 宣言 ) , ( 宣言 ) ;
     DECLARE ( 宣言 , 宣言 ) , ( 宣言 ) 属性 ;
     DECLARE ( 宣言 , 宣言 ) , ( 宣言 , 宣言 ) ;
     DECLARE ( 宣言 , 宣言 ) , ( 宣言 , 宣言 ) 属性 ;
939  DECLARE ( 宣言 , 宣言 ) 属性 ;
     DECLARE ( 宣言 , 宣言 ) 属性 , 算術定数 識別子 次元 ;
     DECLARE ( 宣言 , 宣言 ) 属性 , 算術定数 識別子 次元 属性 ;
  1  DECLARE ( 宣言 , 宣言 ) 属性 , 算術定数 識別子 ;
     DECLARE ( 宣言 , 宣言 ) 属性 , 算術定数 識別子 属性 ;
     DECLARE ( 宣言 , 宣言 ) 属性 , 算術定数 * 次元 ;
     DECLARE ( 宣言 , 宣言 ) 属性 , 算術定数 * 次元 属性 ;
     DECLARE ( 宣言 , 宣言 ) 属性 , 算術定数 * ;
     DECLARE ( 宣言 , 宣言 ) 属性 , 算術定数 * 属性 ;
     DECLARE ( 宣言 , 宣言 ) 属性 , 算術定数 ( 宣言 ) ;
     DECLARE ( 宣言 , 宣言 ) 属性 , 算術定数 ( 宣言 ) 属性 ;
     DECLARE ( 宣言 , 宣言 ) 属性 , 算術定数 ( 宣言 , 宣言 ) ;
     DECLARE ( 宣言 , 宣言 ) 属性 , 算術定数 ( 宣言 , 宣言 ) 属性 ;
     DECLARE ( 宣言 , 宣言 ) 属性 , 識別子 次元 ;
 14  DECLARE ( 宣言 , 宣言 ) 属性 , 識別子 次元 属性 ;
     DECLARE ( 宣言 , 宣言 ) 属性 , 識別子 ;
 18  DECLARE ( 宣言 , 宣言 ) 属性 , 識別子 属性 ;
     DECLARE ( 宣言 , 宣言 ) 属性 , * 次元 ;
     DECLARE ( 宣言 , 宣言 ) 属性 , * 次元 属性 ;
     DECLARE ( 宣言 , 宣言 ) 属性 , * ;
     DECLARE ( 宣言 , 宣言 ) 属性 , * 属性 ;
     DECLARE ( 宣言 , 宣言 ) 属性 , ( 宣言 ) ;
     DECLARE ( 宣言 , 宣言 ) 属性 , ( 宣言 ) 属性 ;
     DECLARE ( 宣言 , 宣言 ) 属性 , ( 宣言 , 宣言 ) ;
 56  DECLARE ( 宣言 , 宣言 ) 属性 , ( 宣言 , 宣言 ) 属性 ;
```

(注) 各行は DECLARE 文 (図7参照) の1階の生成規則の一つの項を表している．左の数字は N_i^1 を表している (空白は0を示す)．

図 8 1階の尺度 N_i^1 の出力例
Fig. 8 Sample output of N_i^1.

```
代入式 ::= 代入式4 ( "&&" 代入式4 )* .
12609      12609    116
                                  0-12495
                                  1-112
                                  2-2
代入式4 ::= 代入式3 ( "|" 代入式3 )* .
12725       12725    210
                                  0-12530
                                  1-181
                                  2-14
代入式3 ::= 代入式2 ( "&" 代入式2 )* .
12935       12935    255
                                  0-12681
                                  1-253
                                  2-1
代入式2 ::= 代入式1 ( ( "+" | "-" ) 代入式1 )* .
13190       13190    807   755
                                            0-11846
                                            1-1159
                                            2-185
代入式1 ::= 代入式素 ( ( "*" | "/" | "//" ) 代入式素 )* .
14752       14752    409   388   157
                                                      0-13819
                                                      1-913
                                                      2-20
```

(a) 代入文の右辺の式

```
添字式 ::= 添字式4 ( "&&" 添字式4 )* .
4849       4849     0
                                  0-4849
                                  1-0
                                  2-0
添字式4 ::= 添字式3 ( "|" 添字式3 )* .
4849        4849     0
                                  0-4849
                                  1-0
                                  2-0
添字式3 ::= 添字式2 ( "&" 添字式2 )* .
4849        4849     0
                                  0-4849
                                  1-0
                                  2-0
添字式2 ::= 添字式1 ( ( "+" | "-" ) 添字式1 )* .
4849        4849     190   108
                                            0-4567
                                            1-266
                                            2-16
添字式1 ::= 添字式素 ( ( "*" | "/" | "//" ) 添字式素 )* .
5147        5147     33    35    0
                                                      0-5079
                                                      1-68
                                                      2-0
```

(b) 添字式

図 9 複製された生成規則に対する0階の尺度
Fig. 9 N_x^0 for production rules.

いる.

他の34個の未テストセグメントは，テスト対象のコンパイラが言語仕様の一部を実現していない初版のものであったため，テストデータが意図的に省略された箇所であった.

(2) 高階の項の尺度 N_i^k $(k \geqq 1)$ は項ごとに分割されたテストデータに対する適用回数を示すので，たとえば次に示すような考察のきっかけを与え，テストデータのレビューを行いやすくする.

図8は図7に示されている DECLARE 文の生成規則における1階の項の尺度 N_i^1 の測定結果の一部を示している．これによればほとんどの項が未テストであり，コンパイラの内部処理をブラックボックスとする立場からはテストデータの不足を示している．しかし，これらの項の意味規則は似ており対応するコンパイル処理は共通なので，一部の項のテストで他の項のテストは代替できるというテストデータ選択基準に従ったと推定される．また，項は最初の右括弧の次の「属性」の有無により二つに分けられる．「属性」のないほうのテストデータが少ないのも同様に代替できるからと推定される.

6. むすび

言語の文脈自由文法に基づくコンパイラのテストデータ網羅性判定手法とそれを支援するツール C-GRAM について述べ，実際のコンパイラのテストデータに適用した評価により C-GRAM の有効性を示した.

C-GRAM は，コンパイラのテストデータ網羅性判定に対して，次のような効果があることが明らかになった.

(1) 不注意による未テストセグメントの発見ができ，また意味規則を考慮するとある程度テストの不足を判定できる.

(2) 詳細化された生成規則に対する適用回数表示により，テストデータのレビューを容易にする.

また，C-GRAM は，構文規則を BNF で定義し字句解析プログラムを作れば，任意のプログラム言語に適用することができる.

C-GRAM は，構文規則に基づいたツールであり，意味規則に対しては十分な判定はできない．しかし，従来人手によるしかなかった大量のテストデータの網羅性判定に，計算機を利用した実用的な確認手段を追加することによって，判定をより的確にすることを可能とした.

今後の課題としては，構文上正しくないテストデータに対する網羅性判定手法，意味規則に基づく網羅性判定手法があげられる．とくに後者は重要であり，テストデータ網羅性判定に効果的な意味規則の定義法を検討する必要がある[5].

謝辞 本研究に対して励ましや助言をいただいた当社情報電子研究所情報処理開発部長首藤勝博士および当社計算機製作所汎用開発グループ山口和彦氏に深謝いたします.

参考文献

1) Goodenough, J. B. and Gerhart, S. L.: Toward a Theory of Test Data Selection, *IEEE Trans. Softw. Eng.*, Vol. SE-1, No. 2, pp. 156-173 (1975).
2) Bazzichi, F. and Spadafora, I.: An Automatic Generation for Compiler Testing, *IEEE Trans. Softw. Eng.*, Vol. SE-8, No. 4, pp. 343-353 (1982).
3) Celentano, A. et al.: Compiler Testing Using a Sentence Generator, *Softw. Pract. Exper.*, Vol. 10, No. 11, pp. 897-918 (1980).
4) Miller, E. F.: Program Testing: Art Meets Theory, *Computer*, Vol. 10, No. 7, pp. 42-51 (1977).
5) 上原，堀川，大川，高野，春原：コンパイラのテストプログラム網羅性判定について，情報処理学会第24回全国大会予稿集，pp. 373-374 (1982).

付録 構文規則の複雑度 C_x^k

(i) 生成規則 p_A: $A=t_1|t_2|\cdots|t_l$. に対して，

$$C_{p_A}^k=\sum_{i=1}^{l} C_{t_i}^k \quad (k=0,1,\cdots).$$

(ii) 項 t: $f_1 f_2 \cdots f_m$ に対して，

$$C_t^k=\prod_{i=1}^{m} C_{f_i}^k \quad (k=0,1,\cdots).$$

(iii) 因子 f に対しては四つの場合がある.

f が終端記号の場合

$$C_f^k=1 \quad (k=0,1,\cdots),$$

f が非終端記号 B の場合

$$C_f^k=\begin{cases}1 & (k=0)\\ C_{p_B}^{k-1} & (k=1,2,\cdots),\end{cases}$$

f が繰返し部 $(t_1|t_2|\cdots|t_m)^+$ の場合

$$C_f^k=\sum_{i=1}^{n} C_{t_i}^k \quad (k=0,1,\cdots),$$

f が繰返し部 $(t_1|t_2|\cdots|t_m)^*$ の場合

$$C_f^k=1+\sum_{i=1}^{n} C_{t_i}^k \quad (k=0,1,\cdots).$$

(昭和58年10月14日受付)
(昭和59年 9 月20日採録