特集● PPL2005

既存Cヘッダファイルの構文の異なる言語での有効利用

平石 拓　八杉 昌宏　湯淺 太一

我々は，コード変形による言語拡張を支援する機構としてS式ベースC言語（SC言語）とその処理系を開発している．既存のCヘッダファイルに相当するSCヘッダファイルを利用したいとき，人手でCのヘッダファイルをSCに書き直すと開発コストがかかるので，本研究ではC言語からSC言語への変換器を実装した．このとき，構文の違いなどにより変換が困難な場合がある．特に，Cプリプロセッサ用の `#define` によるマクロ定義をSCプリプロセッサ用の `%defmacro`，`%defconstant` に置き換えることは完全には不可能である．本論文では，それらの限界と現実的な対応策について議論する．なお，この対応策の考え方の一部は他の言語間においても適用可能である．

We are developing S-expression based C languages, SC languages, and their language system, which supports transformation-based language extension. When we need SC header files corresponding to existing C header files, translating header files into SC by hand will need undesirable implementation cost. So in this research we have implemented a C-to-SC translator. In some cases such a translation is not obvious. In particular, it is sometimes impossible to translate `#define` macro definitions for the C preprocessor into `%defmacro` or `%defconstant` constructs for the SC preprocessor mainly because of their syntactical difference. This paper discusses the limitations of the translation and our pragmatic and reasonable solutions to them. Some of the ideas of our solutions are applicable to translation between other languages.

1 はじめに

我々は，コード変形による言語拡張を支援する機構としてS式ベースC言語（SC言語）とその処理系[2]を開発している．SC言語は，C言語の意味とS式ベースのLisp風の構文を持つ言語である．S式とは

$\langle$S式$\rangle$ ::= $\langle$原子式$\rangle$ | `(`$\langle$S式$\rangle$ `.` $\langle$S式$\rangle$`)`

`(`$\langle$S式$\rangle_1\langle$S式$\rangle_2\cdots\langle$S式$\rangle_n$`)`

$\equiv$ `(`$\langle$S式$\rangle_1$ `.` `(`$\langle$S式$\rangle_2$ `.` `(`$\cdots$ `(`$\langle$S式$\rangle_n$ `. nil))`$\cdots$`)`

で定義される，Lisp系の言語で利用されている文法単位であり，Lisp処理系を用いた変形が容易である．SC言語処理系ではこれを利用し，S式ベースの拡張言語のコードの変形によって言語拡張を実現する．

拡張言語を抽象構文木（AST）に変換したのち，必要な解析・変形を行ってC言語などの言語に変換するような言語処理系[1]や言語拡張機構[7]は従来から存在するが，ASTそのものでプログラミングを行おうとするのが本言語の特徴の1つである．

SC言語でプログラムを記述する際には当然，`printf` 関数などのC言語のライブラリの機能や，`putchar` や `NULL` などのマクロを利用したいという需要がある．そこで本研究では従来の処理系に，ソースコードのプリプロセス時にC言語のソースコードをインクルードする機能を追加した．

Cヘッダファイルを利用する場合，それに相当する

Effective Utilization of Existing C Header Files in other languages with different syntaxes.

Tasuku HIRAISHI, Masahiro YASUGI, Taiichi YUASA, 京都大学情報学研究科通信情報システム専攻, Dept. of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University.

SC のヘッダファイルを書けばよいが，それを人手で行うと開発コストがかかるので，本研究では C 言語から SC 言語への変換器（CtoSC コンパイラ）を実装した．

このような変換は，簡単には行えない場合がある．特に，C プリプロセッサ用の`#define`によるマクロ定義を SC プリプロセッサ用のマクロ定義である`%defmacro`や`%defconstant`に置き換えることは，その定義内容によっては不可能である．これはおもに，SC プリプロセッサは 1 つの構文要素（S 式）を生成するのに対し，C プリプロセッサのマクロは複数の異なる構文要素として解釈され得る（構文要素にならない可能性もある）トークン列を生成するためである．

本研究ではこの問題に対し，本質的に変換ができないマクロは変換を諦め，複数の変換の候補がある場合はユーザに問い合わせるという対応策をとった．実際この方法により，利用にいくつかの制限を生じさせるだけで，実用上ほぼ問題ないような変換を行うことが可能である．本論文ではこの対応策の説明を中心に，CtoSC コンパイラの実装について述べる．

この考え方は，SC 以外の言語から C の関数を呼び出すインタフェースを実装する際にも有用である．

2 SC 言語および SC 言語処理系

まず，本論文で扱う SC 言語とその処理系について簡単に説明しておく（詳細は文献[2] を参照していただきたい）．

2.1 処理系の概要

SC 言語処理系は，以下のモジュールから構成される，Common Lisp[9] 処理系上で動作するシステムである．

- **SC プリプロセッサ**
  C 言語のプリプロセッサと同様の処理を行う．すなわち，`%defmacro`，`%defconstant`で指定されたマクロの展開，`%include`によるファイルの挿入などを行う．2.4 節で詳細を述べる．
- **SC 変換器**
  変換元の S 式（プログラム）と変形規則を入力として受け取り，S 式を与えられた規則にしたがって変形する．2.3 節で説明する（本処理系の中心となる部分だが，本論文の議論にはあまり関連しない）．
- **SC コンパイラ**
  SC-0 言語のコードを C 言語のコードに変換する．SC-0 言語は，本処理系の核となる S 式ベースの言語である．2.2 節で説明する．

```
(def (sum a n) (fn int (ptr int) int)
  (def s int 0)
  (def i int 0)
  (do-while 1
    (if (>= i n) (break))
    (+= s (aref a (inc i))))
  (return s))
```

図 2 SC-0 のプログラム例

本処理系によるコードの変換の流れを図 1 に示す．S 式ベースの拡張言語で記述されたプログラムは，SC 変換器を通して SC-0 言語のコードに変形される（SC 変換器の各適用の前に SC プリプロセッサによる前処理が実行される）．最終的に出力された SC-0 言語のコードはさらに SC コンパイラ，C コンパイラを通して実行形式に変換される．

拡張言語の実装は，SC 変換器に与える変形規則をプラグインとして記述することによって行う．図 1 に示したように複数の規則を用意し，それらを順に適用していくこともできる．

2.2 SC-0 言語

SC-0 言語は，以下の特徴を持つ，SC 変換器の最終的な変形先となる言語である．

- S 式ベースの，Lisp 言語風の構文を持つ．
- ただし意味的には C 言語とほぼ同等である．C 言語で記述可能なコードは，ほぼ 1 対 1 の対応で SC-0 言語でも記述できる．
- 人手によるプログラミングに用いる言語として十分実用的である．

つまり，C 言語の構文を素直に S 式ベースに置き換えたものが SC-0 言語だと考えることができる．

図 2 に SC-0 言語のコードの例を，それに対応する C 言語のコードを図 3 に示す．

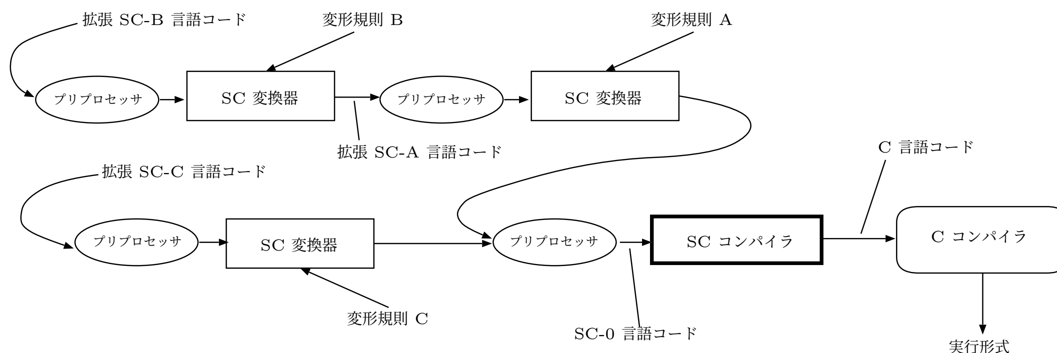

図 1 SC 言語処理系における変換の流れ

```
int sum( int* a, int n ){
  int s=0;
  int i=0;
  do{
   if ( i >= n ) break;
   s += a[i++];
  } while(1);
  return s;
}
```

図 3 図 2 に相当する C プログラム

2.3 SC 変換器・変形規則

SC 変換器に与える変形規則は，以下のような形式の関数として記述する．

(⟨関数名⟩ ⟨パターン⟩ ⟨引数⟩$_1$ ··· ⟨引数⟩$_n$)
-> ⟨式⟩

定義された関数は，通常の Lisp 関数として呼び出すことができる．呼び出しが起こると，まず⟨パターン⟩と第 1 引数のパターンマッチングを行う．マッチした場合，⟨式⟩が Common Lisp 処理系によって評価され，その評価値が関数の返り値となる．⟨引数⟩$_2$以下を指定した場合，これらは通常の引数と同様に扱われる[†1]．

また，同名の関数を複数定義することができる．この場合，各関数定義に指定されているパターンを定義順に調べていき，最初にマッチした関数定義で指定されている⟨式⟩の評価値を返り値とする．

⟨パターン⟩は以下の要素からなる S 式で表現する．

1. ⟨シンボル⟩
   指定したシンボルと同じシンボルにマッチ．
2. ,⟨シンボル⟩
   任意の 1 要素にマッチする．
3. ,@⟨シンボル⟩
   0 個以上の任意の要素列にマッチ．
4. ,⟨シンボル⟩[⟨関数名⟩]
   ⟨関数名⟩で指定された関数に要素を渡し，返り値が真であった要素にマッチする．
5. ,@⟨シンボル⟩[⟨関数名⟩]
   0 個以上の任意の要素列に対し，各要素を関数名で指定された関数に順に渡し，その全ての返り値が真であった場合，その要素リストにマッチする．

[⟨関数名⟩] で用いる関数は，変形規則とは別に，Common Lisp の関数定義と同様に定義しておくことができる（処理系の組み込み関数も指定できる）．また，マッチした S 式全体に x という特別な変数が束縛されるほか，2 〜 5 では実際にマッチした部分の S 式に⟨シンボル⟩の名前の変数が束縛され，->の右辺の式で使用することができる．

関数の定義例を以下に示す

```
(EX (,a[numberp] ,b[numberp]))
-> `(,a ,b ,(+ a b))
(EX (,a ,b))
-> `(,a ,b ,a ,b)
(EX (,a ,b ,@rem))
-> rem
```

†1 パターン以外のデータの受渡しは Lisp の動的変数を利用するのが便利なので，実際に第 2 引数以下を利用することは少ない．

```
(EX ,otherwise)
-> '(error)
```

この関数 EX の適用は以下のようになる.

```
(EX '(3 8))    → (3 8 11)
(EX '(x 8))    → (x 8 x 8)
(EX 8)         → (error)
(EX '(3))      → (error)
(EX '(x y z)) → (z)
```

変形規則は，変形に必要な（一般には複数の）関数を以上のように定義したものである．このうちの関数の 1 つ（最初に呼び出す関数）を，変形したいコード（S 式）に適用することで変形結果が得られる．

簡単な例として，SC-0 言語で Java 言語のラベル付きの break 文，continue 文を利用できるようにする言語拡張のための変形規則を図 4 に示す.

2.4 SC プリプロセッサ

SC プリプロセッサは，プログラム中に以下の形の S 式があると，それぞれに対応した処理を実行する．これらの処理は，2.1 節でも示したように，SC 変換器や SC コンパイラの処理に先立って実行される.

- (%include ⟨ファイル名⟩)
  この S 式の位置に，⟨ファイル名⟩で指定されたファイルの中身を挿入する．C プリプロセッサの #include 命令に相当する.
- (%defmacro ⟨マクロ名⟩ ⟨ラムダリスト⟩ ⟨S 式⟩$_1$···⟨S 式⟩$_n$)
  この S 式を Common Lisp の defmacro 式として評価しマクロを定義する．これにより，定義より後にあるプログラム中の全ての (⟨マクロ名⟩ ···) の形の S 式（リスト）が，同じく Common Lisp の macroexpand-1 関数によるマクロ展開の結果に置き換えられるようになる．置き換えた結果にさらにマクロが含まれていた場合の展開方法は，展開中のマクロと同名のマクロの再帰的な展開は行わないなど，標準の C プリプロセッサの仕様[4] にできるだけ従う.
- (%defconstant ⟨マクロ名⟩ ⟨S 式⟩)
  %defmacro と同様にマクロ定義を行うが，こちらは定義より後にある全ての⟨マクロ名⟩と同じ名前のシンボルが⟨S 式⟩ に置き換えられる.
- (%undef ⟨マクロ名⟩)
  %defmacro , %defconstant によるマクロ定義を取り消す.
- (%ifdef ⟨シンボル⟩ ⟨リスト⟩$_1$ ⟨リスト⟩$_2$)
  (%ifndef ⟨シンボル⟩ ⟨リスト⟩$_1$ ⟨リスト⟩$_2$)
  ⟨シンボル⟩で指定された名前のマクロが定義済みであれば（なければ）⟨リスト⟩$_1$，定義済みでなければ（あれば）⟨リスト⟩$_2$の内容をこの位置に挿入する.
- (%if ⟨S 式⟩ ⟨リスト⟩$_1$ ⟨リスト⟩$_2$)

```
(LBC0 (,@declaration-list))
-> (progn
     (defvar *used-id*)
     (defvar *label-list*)
     (let ((*used-id* (get-all-identifier x))
           (*label-list* nil))
       (LBC1 declaration-list)))

(LBC1 (label ,id[ID] (do-while ,exp ,@body)))
(LBC1 (label ,id[ID] (switch ,exp ,@body)))
-> (let ((*label-list*
     (cons
      `(,id
        ,(generate-id
          (par-identifier id *used-id*))
        ,(generate-id
          (par-identifier id *used-id*)))
      *label-list*)))
     `(begin (label ,id
              (,(car (third x)) ,exp
               ,@(LBC1 body)
               (label ,(second
                        (first *label-list*))
                      nil)))
       (label ,(third (first *label-list*))
              nil)))
(LBC1 (continue ,id[ID]))
(LBC1 (break ,id[ID]))
-> (let ((label-tuple
         (car (member (par-identifier id)
                 *label-list*
                 :key #'(lambda (x)
                          (par-identifier (first x)))
                 :test #'string=))))
     (unless label-tuple
       (error "label ~s is undefined." id))
     `(goto ,(funcall
              (case (car x)
                ((continue) #'second)
                ((break) #'third))
              label-tuple)))
(LBC1 (,@list))
-> (mapcar #'LBC1 list)
(LBC1 ,otherwise)
-> otherwise
```

図 4 ラベル付き break, continue を実現する変形規則

⟨S 式⟩をマクロ展開し，Common Lisp による評価を行う．その結果が，nil または 0 なら⟨リスト⟩$_2$，それ以外なら⟨リスト⟩$_1$の内容をこの位置に挿入する．

- (%error ⟨文字列⟩)
  ⟨文字列⟩をエラーメッセージとして出力し，コンパイルの実行を中断する．
- (%cinclude ⟨ファイル名⟩)
  指定された C 言語のソースファイルを SC に変換し，この位置に挿入する．本論文の主題なので，これより後で詳述する．

3 CtoSC コンパイラ

本研究では，C 言語から SC-0 言語へのコンパイラ（CtoSC コンパイラ）を実装した．これは，C 言語から同じ意味の SC-0 言語への変換，つまり SC コンパイラの逆変換を行うものである．通常は，SC プリプロセッサの%cinclude 命令を通して呼び出され，変換した SC コードが呼び出し元の SC コードに取り込まれるという利用方法を想定している．

この章ではまず，このような変換器を実装する必要性を述べ，その後，実装についての詳細を説明する．

3.1 変換器の必要性

SC 言語は，他のコード変形による言語拡張機構で用いられる AST とは異なり，この言語自体が人間によるプログラミングに利用されることを想定している．2.2 節でも述べたように SC の言語の意味は C 言語と同じなので，C で定義された既存の関数などは，その宣言を SC 言語に置き換えれば SC でもそのまま使用することができる．しかし，両者の文法などは異なるので，C のヘッダファイルなどを直接インクルードして利用できるわけではない．このためには，C のコードを（意味を変えずに）SC の構文に変換する必要がある．

ただし，単に SC-0 言語で書いたプログラムを C 言語に変換し，それを C コンパイラでコンパイルするだけの場合は，そのような変換は必要ではない．これは，SC コンパイラは単に構文解析を行い，対応する C のコードを出力するだけで，意味的な解析は全く行わないからである．例えば，次の SC-0 言語の文

```
(begin (def x double 3.14159)
       (def p (ptr int))
       (* p (sin x)))
```

では，ポインタと浮動小数点の乗算をしているため，意味解析をすれば型エラーになることがわかるが，SC コンパイラはそのような解析は行わず，機械的に

```
{ double x=3.14159;
  int *p;
  p * sin(x); }
```

という C コードを出力するだけである．そのような変換をするだけならば，「sin は double 型の引数を受け取り，double 型の値を返す関数である」という情報は必要ではなく，出力する C コードに

```
#include <math.h>
```

という 1 文を追加し，C プリプロセッサが sin 関数の宣言を取り込めるようにしておけば十分である．

ところが，言語拡張のための変形を行う際にはこれでは不十分な場合がある．例えば，

```
(= y (f (sin x) (cos x)))
→
(= tmp1 (sin x))
(= tmp2 (cos x))
(= y (f tmp1 tmp2))
```

のように，「一時変数を利用して，関数呼び出しが部分式に現れないようにする」変換を行いたいときがある．この変換の際には，tmp1，tmp2 の宣言をどこかに追加しておく必要があり，それを行うためには sin や cos の返り値の型が double 型であるという情報を SC 変換器による変形中に知っておかなければならない．このような場合に対応するためには，上の方法では不十分である．

変換器を実装するもう 1 つの目的は，ヘッダファイル中で定義されたマクロを利用することである．C 言語のプログラミングでよく利用される，NULL や stdin，stdout などの定数，putchar や getchar などの機能は#define によるマクロとして定義されていることが多い．これらを SC から利用するためには，C から SC への変換器を実装する必要がある．

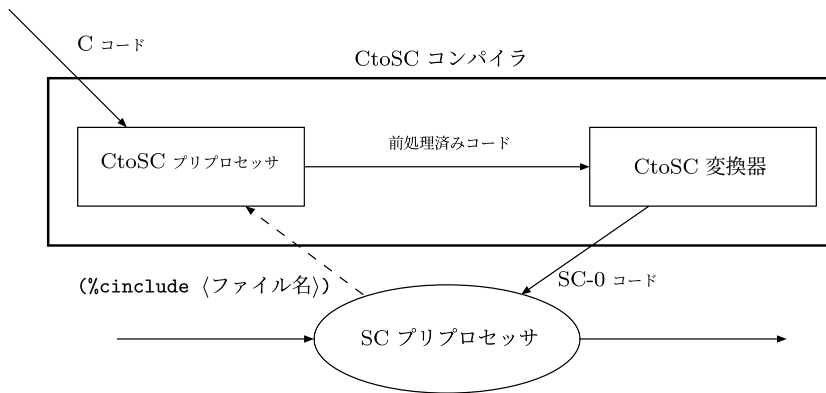

図 5 CtoSC コンパイラにおける変換の流れ

3.2 実装

3.2.1 概要

図 5 に CtoSC コンパイラにおける変換の流れを示す．SC プリプロセッサの前後の流れは，図 1 に相当する．SC プリプロセッサは`%cinclude`命令を読み込むと，C コードのファイル名をパラメータとして CtoSC コンパイラを呼び出す．このコンパイラは，与えられたファイル名の C コードを SC-0 のコードに変換して，結果を SC プリプロセッサに返す（この時点で処理中のコードは，一般には SC-0 のみではなくその拡張言語のものである可能性もあるが，SC-0 言語を含んだ形の拡張であれば問題ない）．CtoSC コンパイラの処理はさらに，CtoSC プリプロセッサと CtoSC 変換器の処理に分かれる．

3.2.2 CtoSC プリプロセッサ

CtoSC プリプロセッサは，既存の C プリプロセッサ（MCPP [6]）を改造して実装したもので，与えられた C コードに対して以下の処理を実行する．

1. まず，C プリプロセッサに相当する処理を実行する．すなわち，
   - `#include` による C ソースコードの挿入
   - `#define`（`#undef`）にしたがうマクロの展開
   - `#ifdef`，`#ifndef`，`#if` にによるソースコードの一部無効化

   などの処理を行い，処理結果を出力していく[†2]．ただし，ソース中の`#define`，`#undef` の各命令に対しては，マクロ定義の処理に加え，

   `#define` *identifier repl-list*
   → (`%defconstant-cexp` *identifier* ⟨文字列⟩)
   `#define` *identifier*(*identifier-list*) *repl-list*
   → (`%defmacro-cexp` *identifier*
   (*identifier-list*) ⟨文字列⟩)
   `#undef` *identifier*
   → (`%undef` *identifier*)

   と S 式への変換を行う（⟨文字列⟩は *repl-list* に相当する Common Lisp の文字列表現である）．変換結果はこの時点では出力せず，2. の処理ために記憶しておく．
2. `#define`，`#undef` を変換した S 式を，1. の出力に続けてまとめて出力する．ただしこのとき，S 式中の⟨文字列⟩は，入力ソースを全て処理したときのマクロの定義状態にもとづいて展開しておく．

2. の処理で S 式を末尾にまとめて出力しているのは，すでにマクロ展開された C のコードが SC プリプロセッサによって再び展開処理されないようにするためである．また，⟨文字列⟩（マクロの定義内容）を

†2 環境ごとの predefined macro などに依存する処理もこの時点で完了するので，たとえば SC に変換済みのヘッダファイルを別の環境で利用することは一般にはできない．

前もって展開する処理は，C のプリプロセッサの標準の仕様[4] を破っているが，これが必要な理由とその影響については 4.2 節で議論する．

3.2.3 CtoSC 変換器

CtoSC 変換器は CtoSC プリプロセッサの出力を受け取り，対応する SC-0 言語のコードを出力する．実装は Common Lisp で行った．

受け取ったコードのうち，C コードの部分に関しては，通常の C コンパイラと同様に字句解析・構文解析を行った後，対応する SC-0 コードに変換する．

`%defconstant-cexp` と `%defmacro-cexp` に対しては，まず C コードの断片である⟨文字列⟩を，**可能であれば** SC-0 言語のコードの断片に変換した後，SC プリプロセッサの `%defconstant`，`%defmacro` 命令を出力する．変換が不可能な場合は，

(`%defmacro` *identifier* (*identifier-list*)
　　`‘(c-exp` ⟨文字列⟩ `,@`*identifier-list*))
(`%defconstant` *identifier* (`c-exp` ⟨文字列⟩))

を出力する．ここで `c-exp` は SC-0 言語のプリミティブで，SC コンパイラによる変換時に⟨文字列⟩をそのまま変換結果として出力する．第 2 引数以降は，Common Lisp の `format` 関数を利用して⟨文字列⟩中に埋め込まれる．このように出力されたマクロも一応利用が可能だが，3.1 節で述べたような，コード変形のための解析を行うことはできなくなる．

`%undef` に対しては，変形は特に行わず，そのまま SC プリプロセッサの `%undef` 命令として扱う．

3.3 C のマクロから SC のマクロへの変換

前述の通り，SC-0 は C と同じ意味を持ち，構文だけが異なる言語である．したがって，C の完全なコード（コンパイル可能な単位）が与えられれば，これを SC-0 に変換するのは容易である．

しかし，`#define` で定義されたマクロを SC プリプロセッサ用の `%defmacro` や `%defconstant` に置き換えるのは，完全には不可能である．これはおもに，`#define` で指定される *repl-list* は C のトークン列であり，構文単位ではないということに起因する．このトークン列の情報だけでは，構文木が作れない場合や，使用される文脈や環境により構文木の作り方が一意に定まらない場合がある．SC プリプロセッサ用のマクロは，構文木とほぼ等価な S 式で定義するため，上記の場合は置き換えができないことになる．引数を受け取るマクロでは，引数はいかなるトークン列にも置き換わる可能性があるため，さらに複雑になる．

[例 1]
```
#define apply_f(a) f(a)
```
[例 2]
```
#define BEGIN {
#define END }
#define NOTOKEN
```
[例 3]
```
#define concat_token(a,b) a ## b
```
[例 4]
```
#define tp_cast(a) ((tp)(a))
```

図 6 SC への変換が困難な`#define` の例

図 6 に，変換が困難なマクロ定義の具体例を示す．

まず，例 1 のような単純なマクロにおいてさえ，適用される場所によって

`y = apply_f(x);` → `y = f(x);`
`long apply_f(int x);` → `long f(int x);`

のような複数の異なる解釈 がある．これらの SC-0 での表現はそれぞれ，`(= y (f x))`，`(decl (f x) (fn long int))` であり，いずれにも対応できるような SC のマクロ定義は不可能である．さらに，後者の解釈の場合，そのような展開が行われるような S 式のマクロを定義すること自体が難しい．

例 2 は構文として不完全なマクロの例である．`BEGIN` や `END` は，単独では構文木を生成できず，したがって SC には変換できない．`NOTOKEN` のような，トークンがまったく含まれないマクロも，構文木の生成ができないという点では上の 2 つと同質である．

例 3 はトークン連結演算子 “`##`” を使用したマクロだが，そもそもトークンベースの言語におけるトークン連結を，S 式ベースの言語に置き換えることは明らかに不可能である．

例 4 は環境によって異なる構文木が生成される例である．このマクロの適用は，

`tp_cast(x);` → `((tp)(x));`

のようになる．この式は，識別子 `tp` が `typedef` で型名として定義されている型環境においては型キャス

ト（SC-0 言語では (cast tp x) ）と解釈されるが，そうでない環境では関数呼び出し（ (tp x) ）と解釈されてしまう．SC のプリプロセッサはこのマクロを，2 つのうちのどちらかとして展開しなければならないが，tp が型であるかどうかはプリプロセス時にはわからないので，どちらが正しいか決定できない．

3.4 対応策

前節で確認したように，C のマクロから SC のマクロへの完全な変換は不可能である．だが，図 6 の例については，次のように捉えることができる．

- 例 2 のような構文木にできないマクロは，S 式ベースの言語である SC ではそもそも利用価値のないマクロである．
- 例 1 も，関数宣言の目的で定義されたとすれば，それは S 式ベースの文法では利用できないマクロである．
- 例 3 のようなトークン連結を行うマクロは，識別子の生成に使われることが一般的であり，使用をその目的に限定すれば利用価値はある．
- 例 4 の例などは明らかに型キャストを想定したマクロ定義であると考えられる．少なくとも，型キャストと関数呼び出しの両方の用途で用いられることを想定したマクロであるとは考えにくい（もしそのような定義を含んでいたとしたら，それはプログラム自体に問題がある）．

つまり，マクロの適用方法の可能性のうち，いくつかの可能性を切り捨てれば多くのマクロは変換可能であり，それでも変換できないマクロは変換する必要性はないとみなすことにする．

本研究では，この考え方に基づいて次のような C コードの断片から S 式への変換アルゴリズムを実装した．

ステップ 1　C 言語の構文を定義する BNF[4] の非終端記号のうち，*translation-unit*，*struct-declaration-list*，*type-name*，*statement*，*expression* のそれぞれの位置からの構文解析を試みる．

ステップ 2　ステップ 1 の構文解析のすべてに失敗した場合は，変換を諦める（このコードで定義された C マクロは，c-exp を利用した SC マクロに置き換えられる）．

ステップ 3　ステップ 1 の構文解析のうち 1 つだけに成功した場合，または同じ構文木（S 式）が得られた場合は，それを変換結果とする．

ステップ 4　複数の候補が得られた場合は，どれを変換結果とするかをユーザに問い合わせ，ユーザからの入力にしたがって変換結果を決定する．

上のステップ 1 に関して，引数を受け取るマクロに対する構文解析においては，**マクロの引数として与えられるトークン列は *identifier* であると仮定する**．この仮定によって，たとえば

```
#define X_OR_Y(x,y) ((x)||(y))
```

と定義されたマクロは，x，y を *identifier* とみなしても，*expression* として構文解析できるので，このアルゴリズムによって

```
(%defmacro X_OR_Y (x y) `(or ,x ,y))
```

と変換できるが，

```
#define X_OP_Y(op) (x op y)
```

のような op に演算子が与えられることを期待したマクロの場合は，op を *identifier* とすると，ステップ 1 の 5 つのどの構文単位としても構文木を構成できないため，変換を諦めることになる．

マクロ定義中の##演算子に対しては，連結されたトークンを *identifier* とみなし，構文解析に成功した場合は，SC プリプロセッサの処理中にシンボルを結合する Lisp 関数が適用されるように変換する．たとえば図 6 の例 3 のマクロは

```
(%defmacro concat_token (a b)
  (concat-symbol a b)
```

と変換される（ concat-symbol はシンボルを結合する Lisp 関数とする）．#演算子の場合も同様に，演算子の適用部分を文字列定数として扱い，SC プリプロセッサの処理中に引数の文字列化を行う関数が適用されるように変換する．

さらに図 6 の例 4 のようなマクロに対応するため，**識別子（上のマクロの引数で *identifier* と仮定したものも含む）が型名かどうかで得られる結果が変わる場合は，両方の構文解析を行い．ステップ 4 の問い合わ**

せの候補に含める†3．ただし，これに起因する問い合わせをなるべく少なくするために，CtoSC コンパイラが処理できる以下のプラグマを定義する．

- `#pragma c2sc_typename` *identifier*
- `#pragma c2sc_not_typename` *identifier*
- `#pragma c2sc_query_typename` *identifier*

`c2sc_typename` で指定された *identifier* は，それ以後のマクロ変換内の構文解析において，型名として扱われる．同様に `c2sc_not_typename` で定義された *identifier* は型名でないとして扱われる．例えば，図 6 の例 4 のマクロ定義を変換する際，通常は，

```
1:‘(cast tp ,a)
2:‘(tp ,a)
>>
```

と問い合わせが発生するが，`#define` の前に，

```
#pragma c2sc_typename tp
```

を記述しておくと，自動的に 1 の結果が適用される．

`c2sc_query_typename` は，上の二つのプラグマによる型名の指定を取り消す†4．

ところで，アルゴリズムのステップ 1 で挙げられている非終端記号の候補の選択は恣意的なものである．候補の数が少ないと変換の失敗が多くなるが，多すぎると変換結果の候補が増え，結果としてユーザへの問い合わせを増やしてしまう．ここでは一つの案として，変換可能なマクロを，宣言，型名，文，式のいずれかとして構文解析可能なものに限定し，それらに対応するために上の 5 つの非終端記号を候補として選んだ．

translation-unit と *struct-declaration-list* はそれぞれ，*external-declaration*（トップレベルの宣言），*struct-declaration*（構造体や共用体のメンバ宣言）のリストとして定義されている非終端記号であり，この両者を候補に含めることで，置き換えた結果が宣言となることを許す．関数定義や関数宣言は *external-declaration*，ビットフィールドを含む宣言は *struct-declaration* でしか対応できないため，この両方を候補に含めた．

type-name は宣言の構文中ではなく，`sizeof` 演算子や型キャストの型指定の構文に現れる非終端記号である．これを候補に入れることによって，`int` や，`struct s *`などと定義されたマクロに対応できる．

statement と *expression* はそれぞれ `while(...){...}`，`y=f(x)` などに置き換えられる文，式のマクロに対応する．

†3 1 つの *identifier* という簡単なマクロの場合でも型名または変数名という二通りが考えられるが，どちらの場合でも得られる S 式は同じなので問い合わせは起こらず，特に問題にならない．

†4 これらの機能を利用するためには C のヘッダファイルを書き換える必要があるが，それを避けたい場合のために，`%cinclude` 命令のオプションでこれらのプラグマと同等の機能を指定できるようにもしている．

4 評価・議論

4.1 POSIX 標準ヘッダファイルへの適用結果

実装した CtoSC コンパイラの実用性を検証するため，FreeBSD5.3-STABLE 上の GCC3.4.2 に含まれるヘッダファイルのうち，POSIX [3] で規定されているもの（一覧を図 7 に示す）に対して変換を試みた．これらのヘッダから `#include` がネストして使用されている場合もあるが，それによってインクルードされるヘッダファイルも検証の範囲に含まれる．ただし，`#ifdef` などの条件判定によって無効化されたコードは含まれない．また，今回は `__attribute__`，`__extension__` などの GCC の拡張文法を用いたコードは原則として検証の範囲外とした．変換結果の確認は人の目で行った．

結果は次のとおりである．まず，これまで述べてきたように，マクロ定義以外の部分に関しては完全に正しい変換結果が得られた．マクロ定義も，大部分は正しい変換結果が得られたが，以下に示す通り，変換が不可能であったもの，変換結果が一意に決まらず，ユーザへの問い合わせが発生したものがあった．

- 変換に失敗したもの

  ```
  #define _STDIO_H
  ```

  などのトークンを含まない（基本的に `#ifdef`，`#ifndef` の条件判定にのみ用いられる）マクロは，前述のアルゴリズムでは変換は失敗となるが，これは特に問題ではない．それ以外で変換に失敗したマクロを図 8 に示す（ここで列挙したマクロの定義内容は，CtoSC プリプロセッサによ

```
<aio.h>          <arpa/inet.h>      <assert.h>
<complex.h>      <cpio.h>           <ctype.h>
<dirent.h>       <dlfcn.h>          <errno.h>
<fcntl.h>        <fenv.h>           <float.h>
<fmtmsg.h>       <fnmatch.h>        <ftw.h>
<glob.h>         <grp.h>            <iconv.h>
<inttypes.h>     <iso646.h>         <langinfo.h>
<libgen.h>       <limits.h>         <locale.h>
<math.h>         <monetary.h>       <mqueue.h>
<ndbm.h>         <net/if.h>         <netdb.h>
<netinet/in.h>   <netinet/tcp.h>    <nl_types.h>
<poll.h>         <pthread.h>        <pwd.h>
<regex.h>        <sched.h>          <search.h>
<semaphore.h>    <setjmp.h>         <signal.h>
<spawn.h>        <stdarg.h>         <stdbool.h>
<stddef.h>       <stdint.h>         <stdio.h>
<stdlib.h>       <string.h>         <strings.h>
<stropts.h>      <sys/ipc.h>        <sys/mman.h>
<sys/msg.h>      <sys/resource.h>   <sys/select.h>
<sys/sem.h>      <sys/shm.h>        <sys/socket.h>
<sys/stat.h>     <sys/statvfs.h>    <sys/time.h>
<sys/timeb.h>    <sys/times.h>      <sys/types.h>
<sys/uio.h>      <sys/un.h>         <sys/utsname.h>
<sys/wait.h>     <syslog.h>         <tar.h>
<termios.h>      <tgmath.h>         <time.h>
<trace.h>        <ucontext.h>       <ulimit.h>
<unistd.h>       <utime.h>          <utmpx.h>
<wchar.h>        <wctype.h>         <wordexp.h>
```

図 7　検証に用いたヘッダファイル

ってマクロ展開済みのものである．図 9 も同様）．

（1）は演算子のみからなるトークンで構文木を生成できないため,（2） は連続する文字列定数のトークンを結合しようとしているため（本来, 連続する文字列定数のトークンは C のプリプロセス時に結合されて 1 つの文字列定数トークンになる）それぞれ変換に失敗している．

（3）は，構造体の初期化子で定義したマクロで,

```
pthread_once_t once_control =
                        PTHREAD_ONCE_INIT;
```

のように利用するものである．しかし，これは前章のアルゴリズム中の 5 つの構文単位のいずれにも含まれないため，変換に失敗する．

（4）は，3.4 節であげた `X_OP_Y` の形のマクロが実際に現れたものである．マクロの引数の `cmp` に演算子を与えることを期待しているため，今回のアルゴリズムでは対応できない．

- ユーザへの問い合わせが発生したもの

52 個のマクロの変換に対して問い合わせが発生した．図 9 はその一部だが，問い合わせが起こった原因としてはここにあげたパターンで全てで

```
（1）
<iso646.h>
 #define and    &&
 #define and_eq &=
 #define bitand &
 #define bitor  |
 #define compl  ~
 #define not    !
 #define not_eq !=
 #define or     ||
 #define or_eq  |=
 #define xor    ^
 #define xor_eq ^=

（2）
<sys/cdefs.h>
 #define __COPYRIGHT(s) \
                    __asm__(".ident\t\"" s "\"")
 ほか，これと同種のマクロが計 8 個

（3）
<pthread.h>
 #define PTHREAD_ONCE_INIT {0, ((void *)0)}
ほか，これと同種のマクロが
<netinet/in.h>に 2 個
<netinet6/in6.h>に 6 個
<socket.h>に 2 個

（4）
<sys/time.h>
 #define timercmp(tvp,uvp,cmp) \
  (((tvp) -> tv_sec == (uvp) -> tv_sec) ? \
   ((tvp) -> tv_usec cmp (uvp) -> tv_usec) : \
   ((tvp) -> tv_sec  cmp (uvp) -> tv_sec))
```

図 8　C から SC の変換ができなかったマクロ

ある．

（1）は，マクロの引数 `c`，`wc` が型名かそうでないかで解釈が変わる．`isascii` を例にとると,

- `‘(== (cast ,c (ptr (bit-not 0x7F))) 0)`
- `‘(== (bit-and ,c (bit-not 0x7F)) 0)`

の 2 通りの解釈が可能である．正しい解釈はもちろん後者（`c` は型名ではない）である.（2）の `WEOF` も同様に

- `’(cast wint_t (- 1))`
- `’(- wint_t 1)`

の 2 通りに解釈できるが，こちらは前者の型キャストの式が正しい解釈である．

（3）（4）は図 6 の例 4 の具体例になっている．例えば `__offsetof` は,

- `‘(cast size_t (ptr (fref (mref (cast (ptr ,type) 0)) ,field)))`
- `‘(size_t (ptr (fref (mref`

```
（1）
<sys/_sigset.h>
 #define _SIG_IDX(sig)  ((sig)-1)
 #define _SIG_WORD(sig) (((sig)-1)>>5)
 #define _SIG_BIT(sig)  (1<<(((sig)-1)&31))
<sys/singal.h>
 #define sigmask(m) (1<<((m)-1))
<sys/select.h>
 #define _howmany(x,y) (((x)+((y)-1))/(y))
<sys/types.h>
 #define minor(x) ((int)((x)&0xffff00ff))
<ctype.h>
 #define isascii(c) (((c)&~0x7F)==0)
 #define toascii(c) ((c)&0x7F)

（2）
<wctype.h>
 #define WEOF ((wint_t)-1)

（3）
<sys/cdefs.h>
 #define __offsetof(type,field) \
                   ((size_t)(&((type*)0)->field))
 #define __rangeof(type,start,end)  \
   (((size_t)(&((type*)0)->end)) -  \
    ((size_t)(&((type*)0)->start)))

<netinet/in.h>
 #define IN_CLASSA(i) \
                 (((u_int32_t)(i)&0x80000000)==0)
 #define IN_CLASSB(i) \
        (((u_int32_t)(i)&0xc0000000)==0x80000000)


（4）
<stdio.h>
 #define feof(p)      \
  (!__isthreaded?    \
   (((p)->_flags&0x0020)!=0):(feof)(p))
 #define ferror(p)    \
  (!__isthreaded?    \
   (((p)->_flags&0x0040)!=0):(ferror)(p))

（5）
<sys/select.h>
 #define FD_ZERO(p) do {                          \
   fd_set *_p;                                    \
   __size_t _n;                                   \
   _p = (p);                                      \
   _n = (((1024U)+(((sizeof(__fd_mask)*8))-1)) \
         /((sizeof(__fd_mask)*8)));               \
   while (_n > 0)                                 \
     _p->__fds_bits[--_n] = 0;                    \
 } while (0)
```

図 9　C から SC への変換が一意に決まらなかったマクロ

```
(cast (ptr ,type) 0)) ,field)))
```

の 2 通りの解釈が可能である.（3）のマクロは型キャスト,（4）のマクロは関数呼び出しと解釈するのが正しい.

最後に（5）の FD_ZERO は，do ブロックの先頭の fd_set *_p; が問い合わせの原因である. fd_set が型名の場合は，変数宣言

```
(decl _p (ptr fd_set))
```

だが，型名でない場合は乗算の式（式文）

```
(* fd_set _p)
```

と解釈される．前者が正しい解釈である.

これらは，いずれも 3.4 節で説明したプラグマをコード中に挿入すれば，問い合わせの発生を回避することができる．また（1）の一部では,「整数定数に&演算子は適用できない」という知識を使えば，変換を一意に決めることができる.

以上より，図 8 に示したマクロ以外に関しては CtoSC コンパイラを通して SC から利用はできることになり，これは十分実用的な範囲である．また，図 8 中でも（1）（3）（4）については，アルゴリズム中で構文解析を開始する非終端記号など，変換時に置く仮定を検討しなおす（または#pragma c2sc_typename などと同様にユーザが指定できるようにする）ことで利用可能にすることができると考えられる.

4.2　変換の安全性

前章で述べたマクロの変換方法は，大胆かつ恣意的な仮定に基づいているので，それらが正しく変換されるという保証はできない．しかし，マクロが正しく変換できなかった場合も，そのマクロを SC 側から利用しない限りは，プログラムが本来と違う意味に解釈されることはない（C コード中でそのマクロが使用されていても，そのマクロはすでに CtoSC プリプロセッサによって展開済みなので問題にならない）.

したがって，まず，変換を諦めたマクロに対しては，そのことがプログラマに伝わる仕組みを作っておけば，当該のマクロが SC で利用できなくなる以上の問題は生じない．また，これらは c-exp を利用したマクロとしては定義されているので，プログラマがそのマクロに利用価値を見出せば，それを利用してもよい．%ifdef , %ifndef , %if における条件判定に利用することも可能である.

問題になるのは，前述のアルゴリズムで得られた変換結果が本来の意図とは異なるものであった場合である．たとえば,

```
#define EQ_AB a=3, b=4
```

と定義されたマクロは，CtoSC 変換器によって,

```
（1）
#define EXIT_IF_A(x)  \
  do {  \
    enum { EQ_AB, c } _t=(x);  \
    switch(_t) {  \
      case a:  exit(1); }  \
} while(0)

（2）
(%defmacro EXIT_IF_A (x)
  `(do-while 0
    (def _t (enum EQ_AB c) ,x)
    (switch _t
      (case a)  (exit 1))))

（3）
#define EXIT_IF_A(x)  \
  do {  \
    enum { a=3, b=4, c } _t=(x);  \
    switch(_t) {  \
      case a:  exit(1); }  \
} while(0)

（4）
(%defmacro EXIT_IF_A (x)
  `(do-while 0
    (def _t (enum (a 3) (b 4) c) ,x)
    (switch _t
      (case a)  (exit 1))))
```

図 10　再帰的なマクロ適用の例

```
(%defconstant EQ_AB (exps (= a 3) (= b 4))
```

と一見何の問題もなく変換されるが，実は，このマクロは列挙型の定義内で

```
enum eabc { EQ_AB, c };
→enum eabc { a=3, b=4, c };
```

と利用するためのものかもしれない．これを SC 側で

```
(def (enum eabc) EQ_AB c)
→(def (enum eabc) (exps (= a 3) (= b 4)) c)
```

と適用しても，これは正しい SC のコードではない（正しくは (def (enum eabc) (a 3) (b 4) c) ）.

人の目で変換結果を確認すればこのような問題は起こらないが，プログラマに逐一そのような確認をさせるのは現実的ではない．そこで，3.4 節のアルゴリズムのステップ 1 で挙げた非終端記号（宣言の列，型名，式，文のいずれか）に対応するマクロは必ず正しく変換されるので，それらに対応するマクロであることを前もって知っていれば，そのマクロは安全に利用できると保証することができると考えることができる．

ところがマクロ定義の *repl-list* の中に入れ子のマクロ適用が含まれている場合，それだけでは安全とはいえない．たとえば，図 10 の（1）のマクロ定義は CtoSC 変換器によって（2）のマクロ定義に変換されるが，このマクロの展開結果中の EQ_AB などがさらにマクロとして展開されるかどうか，展開される場合どのように展開されるかは（C プリプロセッサのマクロ展開の仕様[4] にしたがえば）SC プリプロセッサによる走査時のマクロ定義の状態に依存する．よって，このとき EQ_AB として前述の正しく変換できないマクロ定義がされていた場合などは，EXIT_IF_A も（ EXIT_IF_A 自体は正しく変換されたように見えるのにもかかわらず）安全に利用することはできない．

3.2.2 節で述べた CtoSC プリプロセッサの処理で SC への変換前に *repl-list* の内容もあらかじめ展開しておくようにしたのは，この問題を解決するためである．この実装によって,（ EQ_AB のマクロが C ヘッダ中に定義されていた場合）（1）のマクロ定義は（3）に展開後 SC に変換されるようになり，最終的に（4）の正しい SC のマクロを得られる．実際，4.1 節の検証中のヘッダファイル中にも

```
#define  _IOC(inout,group,num,len) \
  ((unsigned long) \
   (inout \
    | ((len & IOCPARM_MASK) << 16) \
    | ((group) << 8) | (num)))
#define  _IOW(g,n,t) \
   _IOC(IOC_IN, (g), (n), sizeof(t))
#define TIOCFLUSH _IOW('t', 16, int)
```

というマクロ定義が含まれているが，この TIOCFLUSH も_IOW と_IOC の展開を先に実行しておくことにより SC に変換することができるようになる（_IOW の第 3 引数が型名 int なので，そのままでは関数呼び出し式として変換できないことに注意）.

ただし前述の通り，この処理は C プリプロセッサの標準仕様を破っている．だがこれによる影響は「C のマクロ定義の展開方法を，SC の%defmacro , %defconstant , %undef で変更することができなくなる」ということだけであり，あまり重大ではないと考えられる．

本処理系はマクロ以外の部分については完全な変

換ができるので，ヘッダファイル中の関数の型情報，さらには関数定義自体が含まれていても[†5] 変換結果は安全に利用することができる．

4.3 複数の変換候補への対応

本研究の実装では，変換結果が一意に定まらなかった場合は，どの変換にするかをユーザに問い合わせるという方法をとった．これ以外の方法としては，一旦すべての変換結果を保存しておき，あとでマクロの適用場所に応じて正しい変換を自動的に選択するという方法が考えられる．そのようにすれば，プログラミング言語のコンパイラとしては不自然な「ユーザへの問い合わせ」をなくすことができる．

ただし SC 言語処理系にこの方法を適用する場合，どの変換が正しいかという選択は拡張言語の仕様に依存するので，選択の方針は拡張言語の実装者（変形規則の記述者）に任せる必要がある．そのために拡張の実装を複雑にしてしまうと，言語拡張を簡単に実装するという本処理系の特徴が失われてしまうので，実装の際にはインタフェースを工夫する必要がある．

5 関連研究

5.1 他言語からの C の関数の呼び出し

C のコードから生成されるオブジェクトファイルを取り込んで，他の言語から C の関数を呼び出す仕組みは，多くの言語で存在する．たとえば GCC [8] では，一定の命名規則にしたがってコーディングしておけば，Fortran と C のコードをそれぞれコンパイルして生成されたオブジェクトファイルをリンクすることで，Fortran から C の関数を呼び出すことができる．また，Common Lisp 処理系の 1 つである CMU Common Lisp [5] では，C 言語で

```
void foo( int x ){ ... }
```

と定義した関数をコンパイルしてできたオブジェクトファイルを読み込んでおくと，

```
(alien-funcall
  (extern-alien "foo" (function void int))
  10)
```

という手続きによってそれを呼び出すことができる機能を提供している．

ただし，オブジェクトファイルでは型や構造体のフィールドなどの情報はすでに失われているため，上の使用例のように，使用する関数の型などは Lisp 側で改めて再定義する必要がある（この定義を C の元の定義と一致させることはプログラマの責任である）．

本研究の目的は，C のコードからこのような型情報などを自動的に取り出すことなので，単に C の関数を呼び出すことを目的としているこのような手法を利用することはできない．

逆に，本研究の考え方を利用すれば，C のソースから型情報を自動的に取り込み，上のような型の再定義を不要にすることも可能である（扱う言語が C とある程度似ている場合，その言語から C のマクロ定義も取り込めるようにすることも考えられる）．

5.2 C++における C ヘッダファイルの利用

C++のプリプロセッサの`#include`命令では，C 言語のヘッダファイルに記述された関数の型情報やマクロ定義を取り込むことができる．C++言語のコンパイラは，ほぼすべての C のコードをそのままコンパイルできるため，SC の場合のような特別な手続きはほとんど不要である（オーバーロードのための関数名の置き換えを許さないために，`extern "C" { ... }` で C のコード全体を囲んでおく必要がある程度）．

しかし，C のヘッダで定義された関数の型やマクロ定義を，他の言語から利用しようとしている点では，本研究で実現を目指していることは，この C++の機構とほぼ同じだといえる．

6 おわりに

本論文では，SC 言語から C 言語のヘッダファイルを自動的に取り込む機構を実装する際の限界について議論し，それに対する現実的な対応策を示した．また，その機構をいくつかのヘッダファイルに対して実際に適用し，さらにこの対応策の妥当性を議論した．

C 言語のマクロはトークン単位で与えられているため，構文単位でマクロ定義を行う SC 言語への変換を完全に行うのは不可能だが，いくつかの制限を加え

†5 実際の C ヘッダファイルでも，inline 展開される関数などでは，定義の本体が記述されていることがある．

れば，十分実用的な範囲で変換可能である．

本論文で述べた考え方は，他の言語間のインタフェースを実装する際にも応用できる．

なお，CtoSC コンパイラは，当初は既存の C ヘッダファイルを SC から利用する際にヘッダファイルを人手で書き直す手間を省くために実装したものだが，それ以外にも，C で記述された一般のプログラムに対して CtoSC コンパイラを適用し，SC 変換器で変形規則による変形を行う（元のプログラムに何らかの変換を施す）など，ほかの利用方法も考えられる．

参 考 文 献

[1] Frigo, M., Leiserson, C. E., and Randall, L.: The Implementation of the Cilk-5 Multithreaded Language, *ACM SIGPLAN Notices*, Vol. 33,No. 5(1998), pp. 212–223.

[2] 平石拓, 李暁璐, 八杉昌宏, 馬谷誠二, 湯淺太一: S 式ベース C 言語における変形規則による言語拡張機構, **情報処理学会論文誌：プログラミング**, Vol. 46,No. SIG1(PRO 24)(2005), pp. 40–56.

[3] IEEE: *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 6*, 2001. Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992) Open Group Technical Standard Base Specifications, Issue 6.

[4] ISO/IEC: ISO/IEC 9899:1990(E) Programming Languages — C, (1990).

[5] MacLachlan, R. A.: CMUCL User's Manual, http://www.cons.org/cmucl/, (2004).

[6] 松井潔: MCPP — a portable C preprocessor with Validation Suite, http://www.m17n.org/mcpp/, (2004).

[7] Roudier, Y. and Ichisugi, Y.: Java Data-parallel Programming using an Extensible Java Preprocessor, *Swopp'97*, (1997).

[8] Stallman, R. M.: Using and Porting GNU Compiler Collection, (1999).

[9] Steele Jr., G. L.: *Common Lisp: The Language, Second Edition*, Digital Press, 1990.