ソフトウェア工学　52－22
（１９８７・２・１４）

言語設計システム「つくばね」の ユーザーインターフェース記述言語SDRLの言語仕様とその実現法

戸村　哲，保科　剛*，大蒔　和仁，二木　厚吉
電子技術総合研究所　日本ユニバック㈱*

言語設計システム「つくばね」のユーザーインターフェース記述言語SDRLとその実現法について述べる．SDRLはコマンド言語の構文規則と意味規則とを定義するものである．構文規則はコマンド・サブコマンドの階層構造を表現可能な拡張BNFに基づいた記法で記述する．意味規則はその構文規則に動作規則を付加することで記述する．またユーザーインターフェース記述言語固有の機能としてコマンド実行の中断・再開点の指定や援助機能の指定を構文規則に付加することができる．SDRLの処理系はSDRLの記述をLispのプログラムにコンパイルする方式を採っている．コンパイルされたプログラムはコマンド言語を解釈実行するユーザーインターフェースとなる．

USER INTERFACE SPECIFICATION LANGUAGE "SDRL" AND ITS IMPLEMENTATION (in Japanese)

Satoru Tomura, Tsuyoshi Hoshina*, Kazuhito Ohmaki and Kokichi Futatsugi
Electrotechnical Laboratory
1-1-4 Umezono, Sakura-mura, Niihari-gun, Ibaraki, 305, Japan.
Nippon UNIVAC Kaisha*
2-17-51 Akasaka, Minato-ku, Tokyo, 107, Japan.

We have designed and implemented a user interface specification language "SDRL". SDRL is one component of the language design system "Tsukubane". SDRL defines a command language. We can describe a hierarchy of commands in SDRL as syntax rules by an extended Backus Normal Form. Semantics of commands are embedded in syntax rules. We have introduced arguments and local variables in SDRL to write this semantics. To describe the interactional behaviour such as introduced interruption, restart, and help functions, we introduced attributes of syntactic symbols in SDRL. A program of SDRL is compiled into a Lisp program. This compiler is an incremental one in the sence that we can develop SDRL programs interactively.

1. はじめに

言語設計システム「つくばね」は，プログラミング言語の設計過程を支援することを目標としたシステムである．とくに言語設計における実現過程を支援する機能として，ユーザーインターフェースの作成支援系IMF(Interface Management Facility)を提供する．

IMFでは主な入力手段によってユーザーインターフェースを

①キーボード入力によるもの，

②マウス入力によるもの，

とに分類している．そしてキーボート入力主体であるコマンド言語などによるインターフェースを操作中心型(verb oriteted)と，またマウス入力主体で操作対象を視覚的に表示したインターフェースを対象中心型(object oriented)と分類している．

IMFではこの分類に対応して，操作中心型のインターフェース作成支援系としてSDR(Syntax Directed Reader)を，また対象中心型のインターフェース作成支援系としてOMS(Object Manupilating System)をそれぞれ提供する予定である．

本論文では，SDRの定義言語であるSDRL(Syntax Directed Reader Language)の言語仕様を述べ，SDLRコンパイラで用いた遅延コンパイル技法(deffered compilation)について述べる．

2. SDRの機能

SDRは入力構文の文法を与えると，その文法に従って構文解析を行う構文駆動型の入力系(またはその生成系)である．コマンド言語の構文文法を与えることによってSDRを操作中心型のユーザーインターフェースとして使用することができる．この入力構文の文法を記述する言語がSDRLである．

SDRLでは，入力の構文を拡張BNFに基づく文脈自由文法で表現し，意味規則の記述を非終端記号の引数や動作規則などを用いて行う．この引数は属性文法の入力属性に相当するものである．SDRLでは属性文法の出力属性に相当するものは陽には宣言しない．しかし非終端記号は構文解析の結果を多値として返すことができ，また呼び出し側はその値を受け取ることができるので，出力属性があるのと同等である．これらの属性値の計算は左から右に行うので，SDRLをL属性文法を拡張したものとみることもできる．

コマンド言語を定義する他の方法として，各コマンドの構文規則の右辺を字句単位列に制限し，コマンドモードはいくつかのコマンドの繰り返しであり，その中の一つのコマンドによってモードがサブコマンドモードに遷移すると言う方法もある[1]．この方式はモードを持つコマンド体系のコマンドとサブコマンドの階層構造を表現するのに適している．しかし一方では，コマンドの構文規則の右辺に非終端記号が書けないので，予め用意されている字句単位の種類が目的に十分でない場合に，コマンド言語を記述できない欠点がある．そのため拡張性を重視して，SDRLでは一般の文脈自由文法でコマンド言語を記述する．

これらSDRLの機能は一般のコンパイラ生成系[2]の機能と同じである．しかし言語処理系をユーザーインターフェースとして使用するには，従来の言語処理系にはない機能が必要となる．そうした機能に，

1.入力編集機能
2.COMPLETION機能
3.HELP機能
4.QUIT機能

がある．そこでSDRではこれらの機能を呼び出すための特殊キーとしてRUBOUT，ABORT，COMPLETION，HELP，QUITという仮想キーを想定している．

ユーザーが入力したものを再利用するのが入力編集機能である．入力編集機能としてはキーボード入力をEmacsのコマンドを用いて編集する入力編集系(input editor)[1]等がある．しかしこれらの機能は操作システムと深く関係しており，画面制御などが必要となる．そこでSDRでは最低限の機能として，

1)入力文字の一文字削除(RUBOUT)機能と

2)コマンドレベルの入力全体の削除(ABORT)機能

を提供している．

複数の入力が可能なときに，入力の途中でCOMPLETIONキーを押すと，そこまでの入力で残りの入力が決定できる場合に残りをシステムが補う機能がCOMPLETION機能である．またHELPキーを押すと入力可能なものの一覧とその説明を表示する機能がHELP機能である．

コマンドとサブコマンドといったコマンドモードを持つコマンド言語の場合，サブコマンドの入力中に一段上の親のコマンドモードに戻りたいことがある．このためにはサブコマンドで親のモードに戻るコマンドを用意すればよいが，モード毎にそうしたコマンドを用意すると記述が煩雑になりやすい．そこでSDRでは標準のものとしてQUIT機能を用意している．つまりQUITキーを押すことによってサブコマンドから親のコマンドへ戻ることができる．

この他に利用者の入力を容易にするために，ユーザーインターフェースの状態などを表示する入力促進機能を提供している．

これらの機能はユーザーインターフェースに特有な機能であり，SDRLでは構文規則中に属性として記述する．

3. SDRLの言語仕様

SDRLの言語仕様を拡張BNFを用いて説明する．ここで用いる拡張BNFの記法を説明する．拡張BNFのメタ記号として::=, {, }, [,], *, +, |, <, >, "を用いる．::=は規則の定義を表し，{}はアイ味さを除去するための括弧を表す．X，Yが拡張BNFの形式を表すとすると，X*はXの0回以上の繰り返しを，X+はXの1回以上の繰り返しを，X|Y はXまた

はYの選択を，そして[X]はXまたは空列(empty)をそれぞれ表す．<と>とで囲んだ文字列で規則の左辺に出現するものを非終端記号(nonterminal symbol)と呼び，規則の左辺に出現しないものは字句単位(lexical unit)と呼ぶ．メタ記号以外の特殊記号，空白以外の文字列，"と"とで囲まれた文字列は字句定数を表すものとする．とくに字句単位と非終端記号とを合わせて文法記号と呼び，字句単位と字句定数を併せて終端記号(terminal symbol)と呼ぶことにする．

3.1. ユーザーインターフェース定義

<userInterface定義> ::=
　　{ <defkey> | <deflex> | <defrule> }*

ユーザーインターフェースの定義は，メタキーの定義を行う<defkey>，字句単位記号を定義する<deflex>，非終端記号の構文規則を定義する<defrule>からなる．

3.2. メタキー定義

<defkey> ::= (defkey <keyname> <character>)
<keyname> ::=
　　:help | :abort | :quit | :completion

<defkey>は仮想キーを実際のキーに割当を行う．<character>はCommon lisp[3]の文字定数である．例えば，
(defkey :abort #/control-c)
と定義すると，control-Cを入力するとABORTキーが入力されたことになる．

3.3. 字句単位定義

<deflex> ::=
　　(deflex (<lexName> <arg>*) <type>)

<deflex>は字句単位記号<lexName>を型<type>を持つ字句単位記号として定義する．<arg>は字句単位記号の仮引数である．字句単位記号を呼び出す側の実引数の評価値を仮引数の値として字句解析を行う．<type>は字句単位記号の型を表す式であり，Common Lispの型仕様(type specifier)である．例えば字句単位記号nameをCommon Lispのsymbol型として定義するには，
　(deflex (name) symbol)
とする．

3.4. 非終端記号定義

<defrule> ::=
　(defrule (<ruleName> <arg>*) <形式>*)
<形式> ::= <構文形式> | <意味形式>

<defrule>は非終端記号<ruleName>の構文規則を定義する．<arg>は非終端記号の仮引数である．これは属性文法の入力属性に相当する．この非終端記号を構文解析する時には，実引数の評価値を仮引数の値として構文規則の右辺<形式>*を構文解析する．

3.5. 構文規則の記述

<構文形式>は，拡張BNFのメタ記号に対応するものであり，<or>は選択に，<seq>は連接に，<loop>は0回以上の繰り返しに，<optional>は0または1回の出現に，<空>は空列に，<終端記号>は終端記号に，<非終端記号>は非終端記号にそれぞれ対応する．

<構文形式> ::=
　<seq> | <or> | <loop> | <optional> |
　<empty> | <終端記号> | <非終端記号>
<seq>　::= (seq　<基本属性>* <形式>*)
<or>　::= (or　<基本属性>* <形式>*)
<loop> ::= (loop <基本属性>* <形式>*)
<optional> ::=
　　(optional <基本属性>* <形式>*)
<空> ::= (empty)
<終端記号> ::=
　<string> | (<string> <終端属性>*) |
　(<lexName> <arg>* <終端属性>*)
<非終端記号> ::=
　(<ruleName> <arg>* <非終端属性>*)
<arg> ::= <LispForm>

3.6. 意味規則の記述

<意味形式> ::= <let> | <action> | <return>

<意味形式>は意味規則に関する記述であり，局所変数を宣言する<let>，動作規則を記述する<action>，非終端記号の値を返す<return>がある．

<let> ::= (let (<var>*) <形式>*)
<var> ::= <identifier>

<var>*を局所変数として宣言する．変数の有効範囲は<形式>*の中である．変数の初期値はnilである．

<action> ::= (lisp <LispForm>+)

<LispForm>はLispのプログラムであり，順に評価実行する．プログラム<LispForm>の中では，<action>の属する非終端記号の仮引数および有効な局所変数をLispの変数として参照できる．

<return> ::= (return <LispForm>+)

<LispForm>+の値を<return>の属する非終端記号の値として返す．複数の値を返すことができる．この値は後述する<result属性>によって受け取ることができる．

3.7. 属性の記述

ユーザーインターフェースに関する記述などを<構文形式>に付加するのが属性である．

<終端属性> ::=
　<help属性> | <基本属性> | <result属性>
<非終端属性> ::= <基本属性> | <result属性>

<help属性> ::= :help <string>
　HELPキーを入力すると，入力可能な終端記号を列挙する．列挙する終端記号に<help属性>が付いていると文字列<string>をその説明として表示する．

<result属性> ::= :result <var>*
　文法記号が返した多値を対応する<var>に格納する．終端記号の場合は，入力された字句単位を<var>に格納する．

<基本属性> ::=
　<command属性> | <undo属性> |
　<prompt属性> | <条件属性>

<command属性> ::= :command t
　<command属性>はABORTキーやQUITキーが入力されたときに，どこに制御を移すかを指定する．ABORTキーが入力されると実行を中断し，直前の<command属性>のついた<形式>まで戻り，そこから再開する．またQUITキーが入力されると<command属性>のついた<形式>を二段戻り，そこから再開する．例えば，
(seq :command t "a"(seq :command t "b" "c"))
で，"c"を入力する時に，ABORTキーを入力すると(seq :command t "b" "c")から再開し，QUITキーを入力すると(seq :command t "a" ...)から再開する．

<undo属性> ::= :undo <LispForm>
　ABORTキーやQUITキーが入力されると，システムの状態を対応する<command属性>のついた<形式>の実行前の状態に戻すために必要な動作を指定する．実行を再開する<形式>に戻るまでに出現する<形式>の<undo属性>の<LispForm>を順に評価実行する．

<prompt属性> ::=
　:prompt (<controlString> <argument>*)
　入力を行うときの入力促進文字列を指定する．Common Lispの，
(format nil <controlString> <argument>*)を実行して得られる文字列を入力促進文字列とする．入力にしたがって，辿った<形式>の<prompt属性>に記述された入力促進文字列を順に表示する．

<条件属性> ::= :condition <LispForm>
　<条件属性>の属する<形式>の構文解析を終了するときに満足すべき条件を記述する．<LispForm>の評価値がtになるまでその<形式>の構文解析を繰り返す．

　<構文形式>の間には，αを<属性>*，βを<形式>*としたときに，次の等式が成立する．

(or α β)≡(seq α (or β))
(optional α β)≡(or α <空> β)
(loop α β)≡(or α <空> (seq β(loop β))
　また，<意味形式>の構文規則としての意味は次のとおりである．
(let (<var>*) β) --> β
(lisp <LispForm>*) --> <空>
(return <LispForm>*) --> <空>

3. 例題

　簡単なエディタをSDRLで書いた例を示す．このエディタにはchangeコマンドとsaveコマンドがあり，それらのコマンドの構文規則は拡張BNFで以下のように与えられるものとする．

```
<edit> ::=
 "edit" <file> { <change> | <save> }* "exit"
<change> ::=
 "change" <text> <text> { "yes" | "no" }*
<save> ::= "save" [ "file" <file> ]
```

　<edit>は「エディタを起動し，ファイルを編集し，終了する」までのコマンド列を定義する非終端記号である．"edit"の入力によりエディタを起動し，次に編集するファイル名<file>が続き，<change>サブコマンドまたは<save>サブコマンドの繰り返しの後で，"exit"により編集を終了するコマンドである．
　<change>は文字列を置き換えるコマンドである．最初の<text>を次の<text>で置き換える．"yes"または"no"の0回以上の繰り返しは，見つけた文字列を置き換えるか否かの問い合わせに対する利用者の応答を表す．
　<save>は編集したテキストをファイルに格納するコマンドである．<file>はなくてもよい．ない場合は"edit"の直後に指定したファイルに格納する．<file>，<text>は字句単位である．
　この拡張BNFでの定義をそのままSDRLに変換すると次のようになる．

```
(defrule (edit)
   "edit" (file)
   (loop (or (change) (save)))
   "exit")
(defrule (change)
  "change" (text) (text)
  (loop (or "yes" "no")))
(defrule (save)
  "save" (optional (seq "file" (file)))
(deflex  (file) pathname)
(deflex  (text) string)
```

　このSDRLの記述に意味規則・ABORT・QUIT・HELP・入力促進機能等に関する記述を加えると次のようになる．詳しい説明は省略する．

```
(defrule (edit)
  (let (file)
    ("edit" :help "edit file")
    (file :prompt ("file") :result file)
    (lisp (init-edit file))
    (loop :command t
     (or (change) (save file)))
    "exit"))
(defrule (change)
  (let (from to)
    ("change" :help "change string")
    (seq :command t
     (text :result from) (text :result to))
    (lisp (locate from)))
    (loop
      (or
       (seq "yes"
        (lisp(change from to)(locate from)))
       (seq "no" (lisp (locate from)))))))
(defrule (save file)
  ("save" :help "save file")
  (optional(seq "file" (file :result file)))
  (lisp (save file)))
(deflex (file) pathname)
(deflex (text) string)
```

4. SDRL処理系

SDRはSDRLの記述に基づいて下降型のLL(1)構文解析を行う．実現法は解釈実行方式ではなく，SDRLのプログラムをLL(1)構文解析を行うLispのプログラムに変換するコンパイラ方式を用いている．まずSDRL処理系の概要について述べ，次にSDRLで採用した遅延コンパイル技法について述べる．この技法はSDRLプログラムを対話的に開発することを可能とするものである．

4.1. 構文解析系の動作環境

SDRの構文解析系は，下降型のLL(1)構文解析系を行うが，この構文解析系が動作する環境について説明する．

4.1.1. メタキー入力

メタキーの処理のうち，COMPLETION，HELP，RUBOUTの処理は字句解析系が行う．この処理は構文解析系とは独立である．残りのメタキーABORT，QUITが入力されるとシグナルをthrowする．具体的には

ABORT : (throw 'meta :abort)
QUIT : (throw 'meta :quit)

を実行する．

4.1.2. 字句解析系

構文解析系と字句解析系とのインターフェースは，次の3つの関数による．

input inputlist Function

inputは終端記号の入力を行う．引数inputlistはその時点で入力可能な終端記号のリスト(これを入力リストと呼ぶ)である．入力リストにはHELP機能の情報や終端記号の型情報などを含み，COMPLETION機能，HELP機能はこれに基づいて行う．字句解析系は内部変数に直前の入力を保持する．inputは内部変数が空でない時実際の入力をし，入力されたものを内部変数に保持する．inputは関数の値として，内部変数の値を返す．

memToken inputlist Function

memTokenは引数inputlistとして入力リストをとり，直前に入力された字句単位がこの中に含まれているかどうかを返す．

remToken Function

remTokenは直前に入力された字句単位を保持している内部変数を空にする．

ただし簡単のため以下では入力リストは終端記号のリストとして説明する．

4.1.3. LL(1)性の検査

与えられた文法がLL(1)文法であるかどうかの検査は関数checkを用いる．LL(1)性の検査は文法の選択肢に対して行う．

check first follow Function

関数checkは，引数として選択肢のFIRSTのリストとFOLLOWを取る．ただしFIRST(X)は記号列Xから導出される終端記号列の先頭の記号からなる集合である．またここでのFOLLOWとは，開始記号から構文解析を行ったときに選択肢の後ろに続く文形式のFIRSTである．[註1]

選択肢がLL(1)性を満足するとは，各選択肢のFIRSTが互いに素であり，もしある選択肢がεを生成する場合は，FOLLOWとも素であることである．

FIRSTが互いに素であるかどうかを判定するために，対象とするFIRSTに含まれる字句単位記号は高々一つでなければならない．これは，二つ以上の字句単位記号がある場合，それらが互いに素であるかどうかが判定できないからである．一方，ひとつ含む場合は，字句単位記号の型仕様を用いてほかの字句定数と重ならないかどうかを判定できる．

[註1] ここでのFOLLOWは開始記号と構文解析経過に依存するのでLL(1)文法で用いられているFOLLOWとは少し異なる．SDRではすべての文法記号が開始記号と成りうるので，動的に計算している．

4.2 SDRLからLispへのプログラム変換

プログラム変換を関数ϕ[P,F]として説明する．ここでPはSDRLの記述，FはPのFOLLOWである．

4.2.1. 定義のプログラム変換

非終端記号ntの定義は関数ntの定義に変換する．非終端記号ntの引数はそのまま関数ntの引数にする．関数ntはkeyword引数としてfollowを持つ．これは先に述べたFOLLOWを計算するためものである．非終端記号ntを開始記号として構文解析するにはfollowを空(nil)として関数ntを実行する．

```
φ[(defrule(<name> <args>) P),nil] ->
(defun <name> (<args> &key follow)
  φ[P,follow])
```

字句単位記号gの定義も関数gに変換する．字句単位記号gの引数はそのまま関数gの引数とする．関数gを評価すると，字句単位記号定義で指定された型の字句要素を入力し，入力された字句要素を値として返す．

```
φ[(deflex (<name>) <type>),nil] ->
(defun <name> ()
  (progl (input '(<type>)) (remToken)))
```

4.2.2. 構文規則のプログラム変換

構文規則のプログラム変換法を示す．ただし関数firstSetは引数の構文規則からFIRSTを計算する関数である．

```
φ["a",follow] ->
  (progl (input '("a")) (remToken))
φ[(A),follow] -> (A :follow follow)
φ[(seq X Y),follow] ->
  (progn
    φ[X,(firstSet Y follow)] φ[Y,follow])
φ[(or X Y),follow] ->
  (progn
    (check (list (firstSet X)
                 (firstSet Y) follow))
    (input (firstSet '(or X Y) follow))
    (cond ((memToken (firstSet X follow))
           φ[X,follow])
          ((memToken (firstSet Y follow))
           φ[Y,follow])))
φ[(optional X),follow] ->
  (progn
    (check '((empty) (firstSet X)) follow)
    (input (first X follow))
    (if (memToken (firstSet X))
        φ[X,follow])))
φ[(loop X),follow] ->
  (progn
    (check '((empty) (firstSet X)) follow)
    (loop
      (input (firstSet X follow))
      (if (memToken (firstSet X))
          φ[X,(firstSet X follow)]
          (return nil))))
φ[(empty),follow] -> nil
φ[(let (<vars>) X),follow] ->
  (let (<vars>) φ[X,follow])
φ[(return <forms>),follow] ->
  (return-from <rulename> <forms>)
```

但し<rulename>は現在処理している非終端記号名である．

```
φ[(lisp <forms>),follow] -> (progn <forms>)
```

4.2.3. 属性部分のプログラム変換

属性部分のプログラム変換法を示す．

```
φ[(X :result <vars>),follow] ->
  (multiple-value-setq(<vars>) φ[X,follow])
φ[(X :condition <form>),follow] ->
  (loop φ[X,follow] (if <form>(return nil))
φ[(X :command t),follow] ->
  (loop
     (case (catch 'meta φ[X,follow] nil)
        ((:quit) (throw 'meta :abort))
        ((:abort) nil)
        (otherwise (return nil)))))
φ[(X :undo <form>),follow] ->
  (let ((undo t))
    (unwind-protect
      (progn φ[X,follow] (setq undo nil))
      (if undo <form>)))
φ[(X :prompt (<prompt>)),follow] -->
  (progn (format t <prompt>) φ[X,follow])
```

4.3. コンパイル技法

SDRLは言語設計の支援を目標としており，SDRLの処理系はincrementalなプログラミングができる対話的処理系でなければならない．この「対話的」という意味は，プログラムの開発途中で，

①プログラムの部分的試験を行うことができる．試験に必要でない定義が未定義であっても実行できること，

②プログラムの再定義をいつ行っても常に①の条件が満足されていること，

である．この条件を満たす言語処理系の実現法として解釈実行系(interpreter)がある．解釈実行系は，プログラムの実行時に最新のプログラムの必要な部分だけ参照し実行するので，「対話的」処理系を実現する．

しかし解釈実行系は，実行効率がコンパイラに劣る．そこでSDRLの処理系では，コンパイラを用いながら「対話的」条件を満足させる方法として，必要でない部分のコンパイルをできるだけ遅らせる「遅延コンパイル方式」を採用した．この方式はコンパイラを用いながら「対話的」処理系を実現する一技法である．

4.3.1. 局所コンパイルと大域コンパイル

コンパイルには，そのコンパイル単位中にある情報だけでコンパイルする方法（局所コンパイル）と，他のコンパイル単位の情報も含めてコンパイルする方法（大域コンパイル）がある．

局所コンパイルしたコードは，他のコンパイル単位の定義から独立であり，自分自身の定義を変更しない限り常に実行可能である．また，未定義なコンパイル単位があってもよい．しかし，他のコンパイル単位の情報を利用しないので，それを用いた最適化の可能性が残っている．

大域コンパイルは他のコンパイル単位の情報を利用して最適化を行うコンパイルである．しかし，利用したコンパイル単位の定義が変わればコンパイルしたコードは正しくなくなることがある．そこで最適化するときに利用したコンパイル単位の定義が変更された場合，現在のコードが正しいコードであるかどうかを判断し，正しくない場合は，再コンパイルする必要がある．

SDRLでの局所コンパイルと大域コンパイルの例を示す．

例 1.

(defrule (A) (or (B) "+") "-")

をコンパイルする．先に述べた定義にしたがった最も単純なコンパイル・コードを示す．

```
(defun A (&key follow)
  (check (list (firstSet '(B))
               (firstSet '("+")))
         (firstSet '("-") follow))
  (input (firstSet '(or (B) "+")
                   (firstSet '("-") follow))
  (cond
    ((memToken (firstSet '(B) follow))
     (B :follow (firstSet '("-") follow)))
    ((memToken (firstSet '("+") follow))
     (remToken)))
  (input (firstSet '("-") follow))
  (remToken))
```

下線の部分は簡単に部分計算できるのでプログラム変換すると，

```
(defun A (&key follow)
  (check (list (firstSet '(B)) '("+"))
         '("-")))
  (input (firstSet '(or (B) "+") '("-")))
  (cond((memToken (firstSet '(B) '("-")))
        (B :follow '("-"))
       ((memToken '("+"))
        (remToken)))
  (input '("-")) (remToken))
```

となる．これ以上はBの定義がないと変換できないので，これが局所コンパイルしたコードである．

Aの大域コンパイルにはFIRST(B)が必要である．そこで，FIRST(B) = ("*" empty)とすると，下線部のfirstSetの計算と，checkの計算が可能になり，次のコードを得る．

```
(defun A (&key follow)
  (input '("*" "+" "-"))
  (cond
    ((memToken '("*" "-"))(B :follow '("-"))
    ((memToken '("+"))(remToken)))
  (input '("-")) (remToken))
```

このように大域コンパイルしたコードは，局所コンパイルしたコードに比べると，下線部のfirstSetやcheckの計算を実行時に行わない分だけ，実行効率がよい．その代わりFIRST(B)の値が変化した場合には，Aの再コンパイルが必要である．

4.3.2. 遅延コンパイル

コンパイルの時期を遅延させる方式は，データ駆動型と要求駆動型に分類できる．

データ駆動型コンパイルでは「プログラム単位を定義する」ことを「データの到着」とみなす．つまり定義が追加される毎にコンパイルできる部分をコンパイルするものである．この方式では実行に不要なものまでコンパイルを行う．

要求駆動型コンパイルは「プログラム単位の実行」を「要求」とみなし，実行する時点で必要があればコンパイルを行う．したがって，実行に不要なコンパイルは行わない．

例 2.

```
(defrule (A) (or "a" "aa"))
(defrule (B) (or "b" "bb"))
```

AとBが定義されると，データ駆動型コンパイルでは，AもBもコンパイルが完了し（他の定義なしにコンパイルできるため），要求駆動型コンパイルでは，コンパイルはなにも行われない．

次に，Aの実行が行われたとする．データ駆動型コンパイルでは，コンパイルが完了しているので，ただちにAを実行する．しかし，要求駆動型コンパイルでは，まずAをコンパイルし，それから実行する．この時，Bのコンパイルは行われない．

データ駆動型のコンパイルは，実行時の手間が少ない反面，不要なコンパイルが行われる．要求駆動型コンパイルは，実行に必要なコンパイルだけ行うので無駄が少なく，対話的環境で用いるのに適している．

4.3.3. 依存関係グラフ DG

要求駆動型コンパイルを実現するには，実行をする前にそのコンパイル単位が再コンパイルを必要とするかを判定する必要がある．この判定をするためにコンパイルの依存関係グラフDGを用いる．この方法は，

①コンパイルが必要となるのは，前回コンパイルしたときに用いた他のコンパイル単位の定義が変わったときである．

②一度コンパイルすれば，そのコンパイルに使った他のコンパイル単位の定義が変わらない限り再コンパイルする必要はない．

の二つの規則に基づき，コンパイル単位間の依存関係から，コンパイルの必要性を判定するもので

ある．

例 3.

```
(defrule (A) (or (B) (C))
(defrule (B) "b")
(defrule (C) "c")
```

により構文規則を定義し，コンパイルが完了している時に，

```
(defrule (B) "bb")
```

により，Bを再定義したとする．

ここで，Aを実行すると，Aのコンパイルに用いたBの定義が変わっているので，①の条件からAを再コンパイルする．しかし，Cを実行すると②の条件から再コンパイルは行わない．

4.3.4. データ駆動型DGと要求駆動型DG

データ駆動型DGは，定義時にコンパイルの必要なものを計算し，各コンパイル単位のコンパイルの必要性を管理する．要求駆動型DGは，実行時にコンパイルの必要性を計算する．

例 4.

```
(defrule (A) (B))
(defrule (B) "b")
```

により構文規則が定義され，コンパイルが完了しているときに，

```
(defrule (B) "c")
```

によりBが再定義されたとする．

データ駆動型DGでは，Bが再定義された時点でAのコンパイルが必要と判定される．これにより，Aが実行されると，直ちにAのコンパイルが行われる．要求駆動型DGでは，Aが実行された時点で，Aのコンパイルの必要性が計算され，コンパイルが行われる．

また，データ駆動型DGでは，Aのコンパイルが行われると，Bが再定義されない限りAのコンパイルを必要とはしないため，Aを実行するときに手間がかからない．しかし，要求駆動型では，実行する度に，コンパイルの必要性を計算しなければならない．

こうしたコンパイル単位間の依存関係を管理するシステムとしてUnixのmake等がある．makeでは単位間の依存関係を単位の内容とは別に記述し，両者は独立している．そしてDGの管理を要求駆動型で行い，同時にコンパイルを行っている．

これに対し遅延コンパイル技法では，コンパイル単位間の依存関係をコンパイル単位の意味から管理している．DGの管理をデータ駆動型で行い，コンパイルについてはDGに基づいて要求駆動型で行っている．

5. おわりに

現在SDRはSymbolics3640上にCommon Lispを用いて実装している．使用実験として代数的プログラミング言語OBJ2[4]のユーザーインターフェースをSDRLで記述した．記述にはシステムのコマンドの構文からプログラムの構文まですべてを含めた．この経験については稿を改めて報告する予定である．またその経験では遅延コンパイルの機能は期待するものが得られたと考えている．同様な技法はCommon lispのmacroやinline宣言などの処理に十分適用できる方法である．

次に検討課題について述べる．まず，構文解析系はLL(1)構文解析系を用いている．このため，入力可能な終端記号が複数ある時は，それらに共通なものがあってはならない．次のshowコマンドはorの選択肢に共通して"show"を持つのでLL(1)ではない．

```
(defrule (show) (or (showf) (showd)))
(defrule (showf)
   "show" "file" (filename))
(defrule (showd)
   "show" "directory" (directoryname))
```

この例は文法規則を，

```
(defrule (show)
  "show"
  (or (seq "file" (filename))
      (seq "directory" (directoryname)))
```

と変更すれば，LL(1)文法となるが，LR(1)等の上昇型構文解析を用いて解析できる．そこで下降型で解析不能となるときに，局所的に上昇型構文解析を適用して解析を続行する方法などの採用を検討している．

つぎに複数の選択肢でプロンプトを指定しているとキーボード入力方式ではよい解決策がない．

```
(defrule (foo)
   (or ("a" :prompt ("a>"))
       ("b" :prompt ("b>"))))
```

このような場合マウスを使って直接選択肢を指示する方法などを検討する必要がある．

現在は入力編集機能は最低限のものしか提供していないが，これをどう拡充するのか，そうした機能を操作システムとどこで分担するのかなどを検討する必要がある．

参考文献

[1]Symbolics Inc. 1985. Programming the User Interface.

[2]S.C.Johson. 1975. Yacc: Yet Another Compiler-Compiler. Computing Science Technical Report, No.32, Bell Laboratories.

[3]G.Steele. 1984. Common Lisp The Language. Digital Press.

[4]K.Futatsugi, J.A.Goguen, J.P.Jouannaund and J.Meseguer. 1985. Principles of OBJ2. Proc. of 12th ACM Sympo.on POPL. 52-66.

本研究の一部は電子技術総合研究所が日本ユニバック㈱に対して行った技術指導（61電総研670号）として行われたものである．