COINSを用いるためのコンパイラの自動生成の一方式

舞田純一[†] 中井央[†] 佐藤聡[†]

コンパイラバックエンドの実装は，最適化や様々なCPUへの対応などを考慮に入れると多大な労力がかかる．並列化コンパイラ向け共通インフラストラクチャCOINSは，容易にコンパイラを実装するための機能を提供している．COINSは手続き型言語を念頭において設計されており，手続き型言語の基本的な構造を採り入れた高レベルの中間表現HIRやよりマシンよりの低水準中間言語LIR，およびそれらを利用するためのAPIが用意されている．一方で，オブジェクト指向言語や関数型言語に採り入れられている機能，たとえば，クロージャ，クラスと継承など，モダンな言語が持つ機能のフロントエンドを作成し，COINSの提供する機能により，バックエンドを実装するのは容易ではない．我々は，COINSを用いてより容易にモダンな言語機能を実現するために，COINSの中間表現LIRをもとに，よりバックエンドの実装に適した機能を追加した新たな中間表現MIRを考案し，そのライブラリ・処理系を開発した．中間表現MIRは，マクロによる命令系の拡張が可能であり，この命令定義では対象言語に現れるオブジェクト構造の表現を支援するための機能を利用できるようにした．また，対象言語の構造を保ったまま，自然な形でスコープの管理を行うための検索スコープの指定記法を考案し，そのための記号表を実装した．以上の機能は，我々がこれまでに研究・開発した，入力記述を用途に合わせて拡張可能とする構文解析器生成系を用いてコンパイラを実装することを念頭として開発を行った．本論文では，これらのアイデアとその機能による効果について述べる．

A Compiler Generation Method for Using COINS

Jun'ichi Maita,[†] Hisashi Nakai[†] and Akira Sato[†]

It is not easy to implement a compiler backend, especially considering optimization and/or supporting various processors. COINS, A COmpiler INfra Structure, provides functions for implementing a compiler easily. COINS is basically designed for procedural languages. It provides high level intermediate language HIR, which can express the basic structure of procedural languages easily, and also provides APIs to deal with the HIR. However, it is not easy to implement the backend of a compiler of modern programming language that has ability of closure, class and inheritance, and/or etc with COINS. We have invented an intermediate representation MIR based on LIR, COINS's Low-level Intermediate Representation, for implementing modern language features more easily with COINS. And we have developed the library and the processor for MIR. MIR is extensible by macro definitions. The macro expression is designed so that the target language's object structure can be expressed easily. In addition, in order to management scope naturally, we have invented scope notation and have provided the symbol table library for using it. We have implemented the functions above for using with our extensible parser generator. It extends the input notation as usage. In this presentation, we describe these ideas and effects by these functions.

1. はじめに

コンパイラバックエンドの実装は，最適化や様々なCPUへの対応などを考慮に入れると多大な労力がかかる．一方で，コンパイラバックエンドの実現のためのフレームワークがすでに実用化されており，これを用いることでバックエンド実装に必要な労力を大幅に削減できる．

並列化コンパイラ向け共通インフラストラクチャCOINS[2)]はコンパイラバックエンドに必要な様々な機能をモジュールとして提供する．COINSは，低水準中間表現LIR（Low-level Intermediate Representation）および高水準中間表現HIR（High-level Intermediate Representation）を中核としている．コンパイラフロントエンドでは，入力プログラムに対応するLIRまたはHIRを構築することで，COINSの機能によってバックエンドを実現できる．

LIRはよりマシン語に近い表現形式であり，非常に単純で簡潔である．LIRには，関数定義・変数定義や

† 筑波大学
University of Tsukuba

算術式，変数のアドレス取得，メモリ参照・代入，ジャンプ・関数呼び出しなどの構文が存在する．詳しくは参考文献 3) を参照されたい．

HIR は C 言語相当の表現力を持つ高水準な中間表現である．COINS では HIR を構築するためのインタフェースを提供しており，これを用いて HIR を構築することができる．このインタフェースを用いる場合，COINS が提供する専用の記号表を利用する必要がある．HIR は最終的には LIR に変換される．

HIR は C 言語相当の機能を持つ高水準な中間表現であるが，一方で，より高度で複雑な言語機能を実現したい場合，HIR の制約のために実現が複雑になりやすい．このため，COINS を用いて実現されたコンパイラでは，対象言語ごとに HIR の上位の層となる構造を定義し，入力プログラムをその構造を用いて表現し，そこから HIR への変換を行う，といった手段がとられている．しかし，これを直接実装するには相応の労力がかかる．また，この変換の際 HIR 自身が独自の型と記号表を持つため，これを考慮に入れた変換をする必要がある．

本研究では，近年のオブジェクト指向言語や関数型言語の機能を持つ言語のコンパイラの作成を COINS を用いて行うことを想定し，生成系を用いてそれらの機能を実装するための記法を考案した．また，その実現としてコンパイラ生成システムとして実装を試みた．一般に生成系を用いたコンパイラの開発の場合，字句解析・構文解析の部分は生成系の機能によって実現できる．一方で，意味解析の部分は Yacc[8] のアクションなどに見られるように生成系が出力するプログラムを記述しているプログラミング言語を用いたコード断片を，コンパイラの作成者が記述する必要がある．本研究では，記号表の管理に関わるスコープの扱いについての記法を提案している．また，近年のオブジェクト指向言語や関数型言語の持つ機能に対応するコードを LIR として生成することは一般に複雑になるため，これを比較的容易に扱うための中間表現 MIR を考案した．ここではマクロ定義が利用可能であり，マクロを利用した MIR をフロントエンド作成者が作成することで，複雑な LIR の生成を比較的容易にすることが可能となる．そして，これらの機能を，我々がこれまでに開発を行ってきている拡張可能な構文解析器生成系[14),17)] を用いて実装した．

2. スコープの管理の記法

一般に，高度な言語機能を持つプログラミング言語では，スコープの種類が多くなり，その管理が煩雑になりやすい．言語におけるスコープの扱いとは，その言語のポリシによって様々であり，たとえば関数定義にしても，入れ子の定義を許すか，許したとして関数の外側の記号の参照を許すか，といった違いが出てくる．また，実装においては，たとえば，クラスや関数といったスコープの種類ごとにスタックを用意し，記号検索時にはそれぞれのスタックごとに条件を設けて，条件に適うスコープから記号を検索する，といった処理が必要になる．

現在のスコープの扱いの主流はレキシカルスコープであり，レキシカルスコープにおいては，スコープ全体を覆うルートが存在し，現在解析中のブロックからルートまでのブロックの入れ子を線形に見ることができる．そして，そのブロックの種類の並びには規則性がある．たとえば，ある言語の関数定義構文では，その内と外を隔てるマーカとなるブロック（`f` とタグ付けされる）が構築され，さらにその内部にローカル変数用のブロック（`l` とタグ付けされる）が構築されるとする．入れ子関数を許すなら，そのスコープの入れ子は `flfl`…と規則的に構築される．ここで，この言語のポリシにおいて，外側のローカル変数の参照を許すのならば，内側から `fl` の並びを繰り返し見ながら記号を検索していけばよく，外側のローカル変数の参照を許さないならば，最も内側の `l` だけから記号を検索すればよい．このようなポリシは，タグの並びに対する正規表現として表現できる．

我々はブロックに対しその種類を表すタグを付け，構築されたブロックの並びに対し，記号検索のポリシを正規表現を用いて表現することで，抽象的にスコープを管理する方法を提案する．スコープの管理では，ブロックの構築と記号が検索されるブロックの探索を行う必要があるが，今回は主に後者を対象とする．

2.1 スコープ規則の記述

本研究では，上述の考えから，スコープを管理するために以下の 3 つのスコープの概念を導入する．「物理スコープ」は構文上の要素と対応付けられる個々のブロックである．「論理スコープ」は，物理スコープの並びの規則から定義される抽象的なスコープである．前述したように，スコープの管理には言語ごとのポリシがあり，そのポリシを表現するのが論理スコープである．「特殊スコープ」は物理スコープと論理スコープの両方の性質を持つスコープである．現在解析中のスコープを表す `Current` とすべてのスコープのルートとなる `Root` がこれに相当する．

物理スコープや論理スコープには，言語実装者がその種類を表すタグを用意し，他と区別する．本論文で

は，物理スコープは小文字（例：physical），論理スコープは大文字（例：LOGICAL），特殊スコープは先頭のみ大文字（例：Special）として表記する．

論理スコープは，Current から Root までの物理スコープを線形に並べたときの Current α Root において，物理スコープの並び α の，ある部分の並びの規則から定義される．並びの規則は正規表現によって記述される．この正規表現の一覧を**図 1** に示す．ここでは，p は 1 つの物理スコープを，r，s は正規表現を表す．また，優先順位は ()，+ および +?，#，連接の順に高い．

α において，論理スコープ L が正規表現 r に該当する物理スコープの並びとして定義されるとき，L=r と記述する．たとえば，α の部分である p によって論理スコープ L が定義されるとき，L=p と記述する．同様に，r，s の並びとして定義されるとき L=rs，r の 1 つ以上の並びとして定義されるとき L=r+と記述する．+では，通常は最長一致をとるが，最短一致させる場合+?を用いる．具体的に p や q のように物理スコープを特定しない場合「.」を用いる．また，括弧を用いることで，算術式で使われるのと同様に，正規表現の結合の仕方を明示することができる．規則の出現を α の先頭に限定する場合，L=^r と記述する．規則中の一部に該当する物理スコープに対し記号の検索を行わない場合，その部分の直前に#を付与する．我々は，スコープは規則的に構築されるという考えから，0 回以上の並び，選択（r または s）といった表現は除外した．

言語のスコープ規則は，スコープ管理のポリシである論理スコープ（特殊スコープを含む）の集合として表現される．複数の論理スコープに対して記号の検索を行う場合，規則に該当した複数の物理スコープに対する記号の検索の順序は，構築された物理スコープの内側から外側の順にソートされたものとする．たとえば，物理スコープが内側から外側に fedcba と構築されていて，論理スコープ X，Y に対して記号を検索した場合に，X に c が，Y に ed が該当したとすると，記号の検索は，edc の順で行われる．また，1 つの物理スコープが複数の論理スコープに該当してもよく，上記例で X，Y がそれぞれ ed，edcb に該当したとすると，記号の検索は edcb に対して行われる．

2.2 例

例として，以下のような構文を持つ言語について考える．ここで DF は関数定義，S は文，e は式，x は変数，n は整数である．この言語では，関数定義，関数呼び出し，変数への代入が可能であり，関数定義は入れ子にすることができる．関数定義本体を 1 つの独立したローカル変数スコープとし，関数定義の内部から外部の変数が参照可能とする．このとき，同じ名前が複数あれば，より入れ子の内側に属するものを優先する．

p	物理スコープ p
.	任意の物理スコープ
rs	正規表現 r およびそれに隣接する正規表現 s の連接
(r)	結合の仕方を明示する
r+	正規表現 r の 1 つ以上の並び
r+?	最短一致する正規表現 r の 1 つ以上の並び
^r	Current に隣接する正規表現 r
#r	正規表現 r に該当する物理スコープを，記号の検索から除外する

図 1 正規表現の一覧
Fig. 1 Regular expressions for scoping rule.

```
DF   ::=  f(x) { {S} }
S    ::=  DF | x=e; | e;
e    ::=  x | f(x) | n
```

この言語の実装において，関数内部のローカル変数の扱いと関数外部の変数の扱いを区別したいとする．そのため，物理スコープとして，関数内部と外部を分けるためのマーカとなる defun，ローカル変数のための local を導入する．defun はマーカとしてのみ使われ，実際に記号が登録されるスコープではない．これらを用いて，関数定義 DF では以下のように，defun と仮引数のための local，関数定義本体内でのローカル変数のための local を構築する．これによって，以下のように物理スコープが構築される．

```
DF:
  defun
    local
      local
```

入力として，以下のプログラムについて考える．

```
f(x){
  g(y) {
    z = x + y;
  }
}
```

この場合，物理スコープは以下のように構築される．

```
Root
  defun
    local
      local
        defun
          local
            local=Current
```

Current から Root までを線形に並べると Current= local local defun local local defun Root となる．

論理スコープとして，Current を含まない関数内部の変数スコープ（INNER），Root を含まない関数外部の変数スコープ（OUTER）を定義する．これは，ローカル変数と関数外の変数参照についてのスコープ管理のポリシを表し，それぞれ以下のようになる．

```
INNER=^local+
OUTER=(defun local+)+
```

以上を用い，Current，INNER，OUTER，Root に対して記号を検索することで，この言語のスコープ規則を実現することができる．また，記号検索において，その記号がどの論理スコープから検索されたかを判別することでそれに応じた処理を行うことができる．

次に，この言語を以下のように変更したとする．

- それぞれの関数において局所的な定数を定義できる．
- 関数定義の内部からその関数内部および入れ子となっている外部の定数を参照可能とする．
- 関数定義の内部から外部の変数は参照不可能とする．

これを実現するために，定数のための物理スコープとして，const を追加する．関数定義 DF では，以下のように物理スコープが構築される．

```
DF:
  defun
    const
      local
        local
```

論理スコープとして，Current を含まない関数内部の変数スコープ（INNER），定数スコープ（CONST）を定義する．これは，それぞれ以下のようになる．

```
INNER=^local+
CONST=((#.+?) const)+
```

ここで，CONST は const 以外を除外し，const のみを繰り返し探索することを表している．これによって，定数に関するスコープ管理のポリシを表現している．

以上を用い，Current，INNER，CONST，Root に対して記号を検索することで，変更されたスコープ規則を実現することができる．

3. 中間表現 MIR

この章では，我々が LIR をもとに設計した中間表現 MIR について述べる．

3.1 開発の動機

COINS では，アセンブラに近い構造を持った LIR と，より高級言語の構造に近い HIR が用意されている．

コンパイラ作成者が実装する対象となる言語の構造が複雑な場合，LIR の構造と大きく離れているため，原始プログラムの LIR への変換は複雑となる．そのため，たとえば，コンパイラ作成者は，原始プログラムから LIR への変換のために，（複数の）独自の中間表現の実装とその中間表現または LIR との間の変換アルゴリズムを実装する必要がある．LIR と比較すると，HIR の構造はより高級言語の構造に近く，単純な手続き型言語であるならば，直接 HIR へ変換するアルゴリズムを容易に実装することができる．

しかし，実装する対象となる言語が，ある程度複雑さを増すと，その構造は HIR からもかけ離れてしまい，LIR の場合と同様に独自の中間表現と変換アルゴリズムの実装が必要となる．たとえば，COINS に標準で添付されている C，Fortran においても，原始プログラムから直接 HIR を構築せずに，まず HIR-C，FIR といった専用の中間表現に変換した後に，さらに HIR へと変換している．このように，ある程度複雑な構造を持つ言語のコンパイラを COINS を利用して実装する場合，作成するコンパイラのための，専用の中間表現および HIR または LIR への変換アルゴリズムの実装が必要となる．

我々はそのための実装の複雑さを解消するため，マクロによる中間表現の拡張を考えた．マクロとして新たな命令を定義することで，実装するコンパイラ用にもととなる中間表現の機能を拡張できる．このようにすることで，実装するコンパイラごとに独自の中間表現を用意し，そこから HIR や LIR へと変換する処理を直接実装する労力が削減できる．つまり，作成するコンパイラに必要な機能のみの実装に専念することができる．また，マクロとして定義された命令は，他のコンパイラの実装にも再利用できる可能性もあり，さらなる労力の軽減が期待できると考えた．

3.2 HIR，LIR に対するマクロシステムの考察

我々は，まず HIR，LIR に対する単純な置換処理をベースとしたマクロシステムの構築について考えた．

しかし，HIR，LIR に対するマクロシステムは，HIR，LIR の制限から，十分な効果が得がたいという結論に達した．

たとえば，次の処理を行うマクロを定義したい場合を考える．

- ヒープに領域を確保する．
- その領域に対し，必要なデータを書き込む．
- その領域へのポインタを返す．

このような複数の命令からなる処理は，HIR，LIR の制約から，命令の一部として用いることができない．たとえば，代入命令の代入元オペランドや，2 項演算命令のオペランドとして，このマクロを直接展開できない．

擬似コードとして表現すると以下のようになる．ここで tmp は一時変数であり，展開されるたびに別の名前が付けられるものとする．

```
M(x) = {
  tmp=malloc(4);
  *tmp=x;
  tmp;
};
```

これを以下のように式の途中で用いるとする．

```
x = M(M(1234));
```

マクロ M が展開されると以下のようになる．

```
x = {
  tmp0=malloc(4);
  *tmp0={
    tmp1=malloc(4);
    *tmp1=1234;
    tmp1;
  };
  tmp0;
};
```

しかし，前述のように代入命令の代入元オペランドとして複数の命令の並びを与えることはできため，これは妥当な LIR，HIR として表現できない．これを妥当でない LIR として直接表現すると，**図 2** のようになる．この例では，1 行目の代入命令（SET）に対し，第 2 オペランド（代入元オペランド）として 4 行目から 20 行目までの一連の命令列（4 行目 CALL，8 行目 SET，20 行目 MEM）を与えており，また，8 行目の SET に対し，代入元オペランドとして 11 行目から 19 行目までの一連の命令列（11 行目 CALL，15 行目 SET，19 行目 MEM）を与えている．前述の制約から，この LIR は妥当ではない．

```
 1 (SET I32
 2   (MEM I32 (FRAME I32 "x"))
 3   (; 以下は SET のオペランドとして妥当ではない
 4    (CALL
 5      (STATIC I32 "malloc")
 6       ((INTCONST I32 4))
 7       ((MEM I32 (FRAME I32 "tmp0"))))
 8    (SET I32
 9      (MEM I32
10        (MEM I32 (FRAME I32 "tmp0")))
11      ((CALL
12         (STATIC I32 "malloc")
13         ((INTCONST I32 4))
14         ((MEM I32 (FRAME I32 "tmp1"))))
15       (SET I32
16         (MEM I32
17           (MEM I32 (FRAME I32 "tmp1")))
18         (INTCONST I32 1234))
19       (MEM I32 (FRAME I32 "tmp1"))))
20    (MEM I32 (FRAME I32 "tmp0"))))
```

図 2 妥当でない LIR
Fig. 2 Invalid code of LIR.

そのため，以下のように命令の配置を調整する必要がある．

```
tmp0=malloc(4);
tmp1=malloc(4);
tmp1[0]=1234;
tmp0[0]=tmp1;
x = tmp0;
```

これは，まずこのマクロを含むステートメントの直前に，マクロ M の展開コードの，(1) ヒープに領域を確保し，(2) その領域に対し，必要なデータを書き込む命令を配置し，マクロ M 自体は，その領域へのポインタを示すコードとして展開している．実際の LIR は **図 3** のようになる．ここでは，命令のオペランドに複数の命令からなる命令列が含まれないため，この LIR は妥当となる．

これを実現するためには，マクロ定義者が，マクロが展開される位置の外側の命令並びを考慮に入れ，適切な位置に展開される命令が配置されるようにマクロを定義する必要があるが，これではマクロ定義者の負担が大きくなりすぎる．たとえば，上記のようなマクロを 1 つ追加するだけでも，マクロ定義者はそれを含みうるすべての命令について，展開された命令が正しく配置されるための実装を行う必要がある．

```
 1 (CALL
 2   (STATIC I32 "malloc")
 3   ((INTCONST I32 4))
 4   ((MEM I32 (FRAME I32 "tmp0"))))
 5 (CALL
 6   (STATIC I32 "malloc")
 7   ((INTCONST I32 4))
 8   ((MEM I32 (FRAME I32 "tmp1"))))
 9 (SET I32
10   (MEM I32
11     (MEM I32 (FRAME I32 "tmp1")))
12   (INTCONST I32 1234))
13 (SET I32
14   (MEM I32
15     (MEM I32 (FRAME I32 "tmp0")))
16   (MEM I32 (FRAME I32 "tmp1")))
17 (SET I32
18   (MEM I32 (FRAME I32 "x"))
19   (MEM I32 (FRAME I32 "tmp0")))
```

図 3 妥当な LIR
Fig. 3 Valid code of LIR.

このような問題を解決するため，我々はマクロ展開が行われることを考慮した，新たな中間表現 MIR を開発した．

3.3 MIR の設計方針

MIR は以下を目的として設計された．

- 独自の中間表現を実現するための基礎として十分であること
- COINS（LIR）の機能を十分に反映すること
- 単純にして十分な記述力があること
- マクロ定義での自由度が高いこと

前述のとおり，COINS で複雑な構造を持つ言語のコンパイラを実装する場合，コンパイラ作成者は専用の中間表現と，それから HIR または LIR への変換処理を実装する必要がある．MIR はこの実装の労力を削減することを目的としている．前節で述べたように，我々は独自の中間表現から LIR または HIR への変換処理において，妥当な命令配置となるように命令を並べ替えるための処理が特に実装を難しくしていると考え，MIR の処理系では，その労力を軽減するための実装を提供することとした．そのために，MIR には命令配置を明示するための `PROGN` 命令を追加し，MIR のすべての命令は，`PROGN` をオペランドとして用いることで，複雑な処理の流れを表現できるよう設計にした．

LIR は非常に単純であるため，新たな中間表現を開発し，それを LIR に変換することも HIR と比較して容易にできるようになっている．また，マクロがあれば，HIR に存在するような諸々の制御構造などは容易に実現できるため，我々の中間表現は LIR へと変換されることを前提に設計している．COINS では，LIR 段階において多くの最適化が行われる．MIR は LIR へと変換されるため，今回は MIR 段階でのフロー解析処理などの，最適化を補助するための実装は行わないこととした．一方で，マクロの定義では，利用者が自由にコードの展開のされ方を定義できるので，マクロの定義が複雑になるが，マクロ展開段階でのより効率的なコードへの展開は利用者が定義できる．

```
F ::= FUNCTION[f, [{D}], {s}]
D ::= SYM[x]
s ::= D | e   | NOP[]
    | DEFLABEL[l] | JUMP[l] | JUMPC[e, l, l]
e ::= OP[PureOp {,e}]
    | ADDR[x] | REF[x]
    | MEM[e] | SET[e, e]
    | CALL[{e}] | RET[{e}]
    | CONST[c] | PROGN[{s,} e]
```

図 4 MIR の構文
Fig. 4 Syntax of MIR.

型の処理は意味解析に任せ，MIR は LIR 同様マシンレベルの基本型のみを扱う．そのため，LIR の型を踏襲した．これは，今日において同じ手続き型言語に分類される言語においてすら，統一した型システムの提供は不可能であり，型の処理は完全に言語実装者に任せることが結果として最も良いと判断したためである．

3.4 MIR の構文

MIR は**図 4** の構文を持つ．ここで，`F` は関数定義，`D` は記号定義，`s` は文，`e` は式，`x` は変数，`l` はラベル，`c` は定数である．各命令は，暗黙に `I32` の型として扱われる．他の型として扱いたい場合，オペランドの最後に `LIR` の `Ltype` を明示する．プログラムは `F` または `D` の並びとなる．

- `FUNCTION` は関数定義命令である．
- `SYM` は記号定義命令である．トップレベルにおいては，その初期値を与えることができる．
- `NOP` は何も行わない命令である．
- `CONST` は定数即値を示す命令である．
- `ADDR`，`REF` は記号のアドレス取得・メモリ内容参照である．
- `MEM`，`SET` はメモリ参照・書き換え命令である．
- `OP` は算術命令であり，LIR の Pure 式に分類される諸命令を総称する．
- `CALL`，`RET` は関数呼び出し・復帰命令である．

- DEFLABEL，JUMP，JUMPC はラベル定義，無条件ジャンプ，条件付きジャンプである．
- PROGN は複数の命令を処理し，その最後の命令の結果を返す命令である．

LIR では，関数呼出が多値を返すことができ，そのため関数呼び出しを直接他の命令のオペランドにできない．MIR の関数呼び出し CALL 式では，多値返り値の先頭をその値とし，他の式のオペランドとして利用可能にした．また，LIR では返り値変数として指定された変数に値を代入することで，返り値を扱っていたが，これにについても RET 式を追加することで簡潔に扱えるようにした．MIR ではマクロ展開が行われることを想定しているが，マクロ定義での命令の実行順の制限を緩和するため，PROGN 式を導入した．PROGN は式であるため，式をオペランドとして許すすべての命令においてオペランドとして利用できる．PROGN に与えられた命令の並びはそれぞれ文として扱われ，適切な順序で処理される．

図 5 に MIR によるコード例を示す．これは図 3 と同様のプログラムである．PROGN を用いることで，命令のオペランドとして複数の命令からなる処理を与えることができるようになっている．ここでは，9 行目の SET に対し，代入元オペランドとして 11 行目の PROGN を用いることで，11 行目から 19 行目までの一連の命令列（11 行目 SET，16 行目 SET，19 行目 REF）を与えており，1 行目の SET に対し，代入元オペランドとして 4 行目の PROGN を用いることで，4 行目から 20 行目までの一連の命令列（4 行目 SET，9 行目 SET，20 行目 REF）を与えている．このように，MIR を用いることで，より高級な構造を容易に実現できる．

3.5 マ ク ロ

MIR では，その処理系でのマクロ機能を前提としている．マクロを用いることで，対象言語に必要な機能のための新たな要素（命令）を利用者が定義でき，利用できる．

処理系でのマクロ展開機能は，C 言語におけるプリプロセッサのような単純な文字列置換ではなく，任意のマクロを任意の MIR 要素またはマクロへと展開し，その過程において任意の処理を行えるものを想定している．たとえば，マクロ展開中に記号表を操作したり，与えられた情報をもとにマクロ展開時に動的にコードを構築し，大域関数や変数を定義したりできるものを想定している．

マクロを用いることで，図 5 はさらに以下のように構築できる．

```
 1 SET[
 2   REF['x'],
 3   ; 式のオペランドとして任意の式を利用できる.
 4   PROGN[SET[
 5           REF['tmp0'],
 6           CALL[
 7             ADDR['malloc'],
 8             [CONST[4]]]],
 9         SET[
10           MEM[REF['tmp0']],
11           PROGN[SET[
12                   REF['tmp1'],
13                   CALL[
14                     ADDR['malloc'],
15                     [CONST[4]]]],
16                 SET[
17                   MEM[REF['tmp1']],
18                   CONST[1234]],
19                 REF['tmp.2']]
20         REF['tmp.2']]]]]]]
```

図 5 MIR によるコード例
Fig. 5 An example of MIR.

```
SET[REF['x'], M[M[CONST[1234]]]]
```

このマクロ M の定義は，本システムでは以下のようになる．ここで use_local はこのマクロでのみ有効な一時命令を宣言するための補助関数である．

```
M[x]{
  tmp = use_local('tmp')
  PROGN[SET[
           REF[tmp],
           CALL[
             ADDR['malloc'],
             [CONST[4]]],
         SET[
           MEM[REF[tmp]],
           @x],
         REF[tmp]]
}
```

マクロの定義は，本システムを用いて行うことを前提としている．マクロの定義については後述する．

4. システムの概要

以上の考えをもとに，我々は COINS を用いるためのコンパイラ生成システムを開発した．

本システムは，我々が開発している自己拡張可能構文解析器生成系[14] を基礎としている．この生成系で

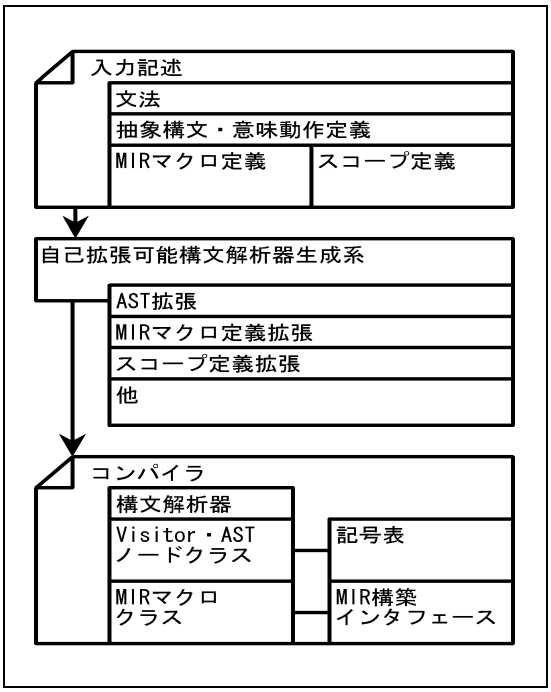

図 6 システムの全体像
Fig. 6 System overview.

は，生成系本体は純粋に構文解析器を生成する機能のみを持ち，「拡張」を用いることで，新たな生成系機能を追加できるようになっている．これによって，たとえば Yacc[8] のアクション機能のような拡張や，SableCC[4] の AST 構築機能のような拡張を用途に合わせて使い分けることができる．また，生成系自身を用いて拡張を独自に定義することも可能である．

本研究では，今回開発した機能を自己拡張可能構文解析器生成系の拡張として実装した．本システムは自己拡張可能構文解析器生成系の機能に加え，記号表，MIR 構築インタフェースおよび MIR マクロ定義の機能を提供する．本システムは，全体として，**図 6** のようになる．

以降は，これらについて解説する．

4.1 システムの全体像

本システムを用いた典型的なコンパイラの実装は，以下のようになる．

- 具象構文の定義及び抽象構文との対応付けの定義
- 記号表エントリの属性定義
- 抽象構文要素とその意味動作の定義および構築される MIR との対応付けの定義
- MIR マクロの定義

このようにして定義され，生成されたコンパイラでは，(1) まず入力を構文解析して AST を構築し，(2) 構築された AST を意味動作を行いながら走査して MIR を構築し，(3) 構築された MIR を LIR に変換し，(4) 最終的に COINS の機能によって LIR から実コードが生成される．

```
%class ACompiler
…ヘッダ部…
%%
%SYMENT{
  name,
  …記号表エントリの定義…
%}
%SCOPE{
  …スコープの定義…
%}
%MIR{
  …MIR マクロの定義…
%}
%AST{
  program(decls, body)
  => …MIR への対応付け…
  {
    …意味動作の定義…
  }
  …抽象構文の定義と MIR への対応付け…
%}
%%
program:
   decls body
   => program(decls, body)
…具象構文の定義…
%%
…プログラムメイン…
```

図 7 システムに与える記述
Fig. 7 The overview of the input for the system.

4.2 システムに与える記述

本システムに与える記述は以下のものがある．全体として，入力記述は**図 7** のようになる．

記号表エントリ定義

`%SYMENT{…%}`．記号表エントリが持つ属性の定義を記述する．

スコープ定義

`%SCOPE{…%}`．スコープの定義を記述する．

MIR マクロ定義

`%MIR{…%}`．MIR マクロを定義する．

AST 要素定義

`%AST{…%}`．AST の要素およびその意味動作を定義する．また，定義される AST 要素と変換される MIR への対応付けを定義する．

記号表エントリが持つ属性の定義では，属性名を列挙する．記号表エントリには，この記述をもとに各属

性が追加され，そのアクセスメソッドが定義される．意味動作の定義やマクロ定義など，記号表を用いる場面では，それぞれのエントリにおいて定義した属性を用いることができる．

スコープの定義は前述のスコープの説明どおりに記述できる．たとえば，以下は 2 章の説明と同様の意味である．

```
%SCOPE{
  INNER = ^local+
  OUTER = (defun local+)+
%}
```

具象構文の定義では，各構文に対し=>ASTElem(attrs, …) としてそれに対応させる AST 要素が記述できる．ここで，ASTElem は対応させる AST 要素名，attrs はその子要素であり，具象構文に現れる記号の持つ値，他の AST 要素，または文字列，数値などの即値を指定できる．たとえば，以下のように記述すると expr '+' term，expr '-' term という具象構文に対し，apply('+', expr, term)，apply('-', expr, term) という抽象構文を対応させる．

```
expr : expr '+' term
       => apply('+', expr, term)
     | expr '-' term
       => apply('-', expr, term) ;
```

個々の AST 要素の定義は，ASTElem(attrs, …) =>MIR[…] {…} と記述する．ASTElem，attrs は前述と同様である．=>MIR[…] はその抽象構文要素から構築される MIR を定義する．また，{…}にはその抽象構文要素における意味動作について，Ruby コードとして記述できる．

たとえば，以下のように記述すると AST 要素 apply を定義する．apply は関数名 f と関数呼び出しの引数となる args を子要素として持つ．意味動作として型検査などを実行し，最終的に MYCALL という MIR マクロに変換される．

```
apply(f, args)
=> MYCALL[f, args]
{  …  }
```

MIR マクロ定義では，MacroName[args, …] として個々のマクロを定義できる．マクロ定義本体は MIR 構築インタフェースを用いて Ruby コードとして記述する．たとえば，以下のように記述すると MIR マクロ MYCALL を定義する．MYCALL は関数名 f と関数呼び出しの引数となる args を引数にとる．ここでは関数名が+なら OP に，そうでないなら CALL に展開される．

```
MYCALL[f, args] {
  if f == '+'
    OP[:ADD, args]
  else
    CALL[f, args]
  end
}
```

4.3 記 号 表

AST 要素定義および MIR マクロ定義において，本システムが提供する記号表が利用できる．

記号表は以下のインタフェースを提供する．

- 物理スコープの構築（enter）
- 論理スコープを用いた記号の検索（lookup）
- 物理スコープを用いた記号の登録・更新・削除（install，update，delete）
- 論理スコープを用いた物理スコープの列挙（collect）
- あるスコープに属する記号の列挙（each）

記号の検索では，検索された記号はそれが検索された論理スコープについての情報を持ち，これを用いて処理が行えるようになっている．たとえば，**図 8** のように用い，記号を Current，INNER，Root スコープから検索し，記号が見つからなかった場合や Root スコープから検索された場合に，それぞれ特別な処理を行う，といったことができる．この例ではまた，検索処理の結果を変数 ent に代入している．

4.4 MIR 構築インタフェース

MIR マクロ定義において，その展開される MIR を表現するために，MIR 構築インタフェースを用いる．MIR 構築インタフェースは，MIRElem[args, …] のように用い MIR を構築する．たとえば，SET[REF[x], CONST[0]] のように用いる．また，定義したマクロの使用も同様に記述できる．MIR 構築インタフェース

```
ent = lookup(記号名, ：Current, :INNER, :Root){
  #見つからなかった場合の処理
}.if_in(:Root){
  #Root で検索された場合の処理
}.done
```

図 8 記号検索の例
Fig. 8 An example of lookup.

を用いることで，MIR を容易に構築できる．MIR 構築インタフェースでは，本システムの記号表を用いることを前提としており，関数定義要素 FUNCTION の構築では，記号表を受け取ることができる．

また，本システムはマクロ定義を補助する以下のインタフェースを提供する．

- ローカル変数・ラベルの定義（use_local，use_label）
- 大域変数・大域関数の定義（use_global）
- トップレベル情報の取得（toplevel）
- 処理中のスコープに関連する記号表の取得（current_symtab）
- 対象言語におけるオブジェクトの構築コードの生成（construct）

これらを用いることで，マクロが展開される文脈において，ローカルな一時変数を用いたり，記号表から記号を列挙し，関数やデータオブジェクトを構築したりするといったことが可能である．

4.5 簡単な使用例

簡単な使用例として，構造体をヒープに構築することを目的としたマクロの定義を**図 9** 示す．3 行目では，このマクロは構造体のサイズ size および各要素を初期化するための式の配列 inits を引数にとることを宣言している．4 行目ではこのマクロ展開のために，目的コードレベルのローカルな一時変数を使用することを宣言している．この一時変数は確保した領域へのポインタの保持のために用いる．また，5〜10 行目では確保した領域に対して inits の各式の値を書き込むコードを構築している．11〜15 行目がこのマクロによって展開される MIR を示しており，このマクロは malloc により領域を確保し，それに対して初期化を行い，確保した領域へのポインタを返す PROGN 式へと展開される．

点を表す構造体 PT，色付きの線を表す構造体 LINE を定義し，それを使って線オブジェクトを作成する以下のような原始プログラムを考える．

```
PT=struct{int x; int y;}
LINE=struct{int color; PT from; PT to;}
LINE(0, PT(15,20), PT(3,0))
```

これは上述のマクロを用いて以下のように表現できる．そして，その LIR としての出力は**図 10** となる．

```
ALLOCSTRUCT[12, CONST[0],
  ALLOCSTRUCT[8, CONST[15], CONST[20]],
  ALLOCSTRUCT[8, CONST[3], CONST[8]]]]
```

```
 1  %MIR{
 2    …
 3    ALLOCSTRUCT[size, inits] {
 4      t = use_local("t")
 5      code = []
 6      @inits.each_with_index{|i, x|
 7        code << SET[MEM[OP[:ADD,
 8                      REF[t], CONST[x*4]]],
 9                    i]]
10      }
11      PROGN[
12        SET[REF[t],
13            CALL['malloc', [CONST[@size]]]],
14        code,
15        REF[t]]
16    }
17    …
18  %}
```

図 9 マクロ定義の例
Fig. 9 An example of the input.

5. 本システムの実装

本システムは，前述の自己拡張可能構文解析器生成系の拡張として実装されており，実装言語として Ruby を用いている．本章では，本システムにおける実装について述べる．

5.1 AST の構築・走査

本システムにおける AST の構築・走査の実装は，自己拡張可能構文解析器生成系標準の AST 拡張をもとに，本システムにあわせて機能追加したものとなっている．

本システムにおける AST の構築・走査は，Visitor パターン[6] によって実現される．AST に関連する入力記述の処理では，それぞれの AST 要素に対応する AST ノードクラスと，それを走査するための Visitor クラスを生成する．AST に対する意味動作の記述は多パスに対応しており，多パスの場合はそれぞれのパスに対応する Visitor を生成する．AST ノードおよび Visitor においては，任意の属性を持たせることができ，意味動作の処理ではこれを用いることができる．

AST 要素の定義における，その AST 要素から構築される MIR についての記述からは，1 つのパスとして MIR の構築を行うための Visitor が生成される．その Visitor には MIR 構築インタフェースを用いた MIR の構築処理が意味動作として定義される．

5.2 記 号 表

本システムでは，2 章で述べたスコープ規則の記述に基づいて記号を扱うための独自の記号表を備えてい

```
(CALL (STATIC I32 "malloc")
  ((INTCONST I32 12))
  ((MEM I32 (FRAME I32 "functionvalue_1.5"))))
(SET I32
  (MEM I32 (FRAME I32 "t_0.4"))
  (MEM I32 (FRAME I32 "functionvalue_1.5")))
(SET I32
  (MEM I32 (ADD I32
      (MEM I32 (FRAME I32 "t_0.4"))
      (INTCONST I32 0)))
  (INTCONST I32 0))
(CALL (STATIC I32 "malloc")
  ((INTCONST I32 8))
  ((MEM I32   (FRAME I32 "functionvalue_3.7"))))
(SET I32
  (MEM I32 (FRAME I32 "t_2.6"))
  (MEM I32 (FRAME I32 "functionvalue_3.7")))
(SET I32
  (MEM I32 (ADD I32
      (MEM I32 (FRAME I32 "t_2.6"))
      (INTCONST I32 0)))
  (INTCONST I32 15))
(SET I32
  (MEM I32 (ADD I32
      (MEM I32 (FRAME I32 "t_2.6"))
      (INTCONST I32 4)))
  (INTCONST I32 20))
(SET I32
  (MEM I32 (ADD I32
      (MEM I32 (FRAME I32 "t_0.4"))
      (INTCONST I32 4)))
  (MEM I32 (FRAME I32 "t_2.6")))
(CALL (STATIC I32 "malloc")
  ((INTCONST I32 8))
  ((MEM I32 (FRAME I32 "functionvalue_5.9"))))
(SET I32
  (MEM I32 (FRAME I32 "t_4.8"))
  (MEM I32 (FRAME I32 "functionvalue_5.9")))
(SET I32
  (MEM I32 (ADD I32
      (MEM I32 (FRAME I32 "t_4.8"))
      (INTCONST I32 0)))
  (INTCONST I32 3))
(SET I32
  (MEM I32 (ADD I32
      (MEM I32 (FRAME I32 "t_4.8"))
      (INTCONST I32 4)))
  (INTCONST I32 8))
(SET I32
  (MEM I32 (ADD I32
      (MEM I32 (FRAME I32 "t_0.4"))
      (INTCONST I32 8)))
  (MEM I32 (FRAME I32 "t_4.8")))
```

図 10 出力例
Fig. 10 The example of the output.

る．また，この記号表に格納される記号表のエントリがどのような構造を持つかについては，利用者が定義できる．これによって，目的言語の実装に必要な属性を自由に持たせることができるようになっている．具体的には，名前を取得する name というインタフェースを備えるオブジェクトならば記号表のエントリとして利用でき，本システムへの入力記述 %SYMENT{…%} は，name インタフェースが定義されたクラス Entry を定義するコードへと変換される．

記号表の実装は，記号表クラスと記号表マネージャクラスから構成される．記号表クラスは記号の登録，検索などの記号管理のためのインタフェースを持つ．本システムでは，1 つの物理スコープ（ブロック）を 1 つの記号表クラスのインスタンスで扱う．あるブロックで宣言された記号は，その対応する記号表クラスのインスタンスへと登録される．ブロックは入れ子になりうるため，記号表クラスのインスタンスもそれに対応した構造を持つ．入れ子のブロック間の関係は，記号表クラスのインスタンス間の関係として実装されている．それぞれの記号表クラスのインスタンス間の関係は，全体としては木構造となるため，記号表クラスはその階層関係を示すためのインタフェースを持つ．これには，親ブロックに対応する記号表インスタンスの取得，子ブロックに対応する記号表インスタンスの取得などがある．

記号表マネージャクラスはすべての記号表クラスのインスタンスを管理するために用意されており，利用者はこれを用いて記号表を操作する．記号表マネージャクラスは記号表クラスと同様のインタフェースを持ち，特に指定されない限り，Current スコープに相当する記号表クラスのインスタンスへその処理を委譲する．記号表マネージャクラスには，利用者が物理スコープ，論理スコープを定義するためのインタフェースを用意している．論理スコープの定義では，与えられた規則に従い物理スコープを探索する新たなメソッドをリフレクションを用いて定義する．本システムへの入力記述 %SCOPE{…%} は，このメソッドの呼び出しを行うコードへと変換される．記号の検索ではこのように定義されたスコープを扱うことができるようになっている．

記号表に対する記号の検索には，lookup メソッドを用いる．lookup メソッドは，記号の名前と複数の検索対象となるスコープ，記号が見つからなかった場合の手続きを引数としてとる．検索した結果は LookupResult というクラスのインスタンスとして返される．LookupResult は，検索されたエントリ（記号が見つからなかった場合，手続きの評価結果）と，それが検索されたスコープ情報を持ち，if_in(

logical_scopes, &block), done() というインタフェースを提供する．if_in(logical_scopes, &block) は，検索結果が logical_scopes で与えられた複数の論理スコープの中から検索されたものであるなら，&block の手続きを評価する．その手続きの評価結果は LookupResult に保存される．if_in は，つねにそのレシーバである LookupResult のインスタンス自身を返す．done() は検索処理結果のエントリそのものを返し，分岐後のエントリの取得に用いる．これらを用いることで，図 8 のように，記号が検索されたスコープによる分岐処理と，検索処理の結果の取得を 1 度に記述するための構文を実現している．この例では，検索結果の LookupResult に対し，if_in により Root で検索された場合の処理を与えている．最後の done により，Root で検索されたならば，与えられた手続きの評価結果を，そうでないのならば，lookup により検索されたエントリを取得している．

5.3 MIR 構築インタフェースとマクロ

本システムを用いて生成されたコンパイラでは，AST を構築・走査し，MIR の構築を行う．本システムでは MIR の各命令に対応したクラスが用意されており，MIR の構築において，内部ではこれらのクラスのインスタンスが作成され，原始プログラムに対応する MIR の構造が形成される．

MIR の各命令に対応したクラスには，その命令を対応する LIR へと変換するための gen_code メソッドが定義されている．構築された MIR は木構造となっており，MIR から LIR への変換処理では，gen_code が木のルートから終端まで再帰的に適用される．gen_code は MIR 要素を返してよく，その場合 LIR に展開されるまで gen_code が再び適用される．

MIR から LIR への変換では，PROGN が持つ各オペランドに関して，それらの命令が LIR として妥当な実行順となるように命令の並べ替え（木の組替え）を行う必要がある．これは，PROGN をオペランドとして含みうるすべての命令において行う必要があり，命令の並べ替えは構築された MIR の木構造全体に波及する．そのため，完全な MIR が構築されるまでは，どのような命令の並びとなるか定まらないため，LIR へと変換することはできない．MIR の各命令に対応したすべてのクラスでは，この命令の並べ替えのための処理が gen_code メソッドに実装されている．

マクロは，利用者が MIR 構築インタフェースを直接用いて定義する．本システムでは，LIR レベルでの命令の並べ替えに関する実装はすべて提供されているので，システムの利用者は PROGN 命令を用いるだけで，命令の並べ替えに関する煩雑な実装を行うことなく，複雑な処理が記述できるようになっている．

マクロは MIR の各命令に対応するクラスと同様に gen_code が定義されているクラスであればよく，実装レベルでは両者はまったく同等である．本システムへの入力記述 %MIR{…%} に定義されたマクロは，gen_code メソッドが定義されたクラスへと変換される．

5.4 LIR への変換とコンパイル

構築された MIR から LIR への変換処理は，上述の gen_code メソッドを用いて行われる．このとき不要コードの除去も行われる．LIR のコンパイルには COINS が用意している API を用いている．COINS の API は Java で実装されているので，Ruby-Java ブリッヂ rjb [1] を利用して，これを用いている．gen_code によって構築された LIR はテキスト形式で一時ファイルに保存され，それを COINS に読み込ませ，COINS のバックエンド機能を用いて最適化，コード生成を行う．

6. 適 用 例

本章では，システムの適用例について述べる．

6.1 クロージャ

本システムによるクロージャの実装例として，以下のような構文を持つ手続き型言語の実装を行った．

s	::=	x=e;	代入文
	|	e;	式文
e	::=	x	変数参照
	|	e(e)	関数適用
	|	\(x){ {s} return e; }	関数抽象
	|	N	整数

s は文であり，e は式である．文は代入と式文からなり，式は変数参照，関数抽象，関数適用，整数からなる．プログラムは，文の並びとなる．この言語では，関数はファーストクラスとして扱われ，変数への代入や，関数の返り値として関数を返すなどができる．また，関数内からはすべての自由変数を参照できる．組み込みの関数として加算関数 add と出力関数 print を持つ．

実装に際し，スコープは 2 章の例と同様に取り扱った．また，関数抽象 ABS と関数適用 APP を MIR マクロとして定義した．

ABS は，以下を行うコードに展開される．

(1) 大域関数を定義する．この関数はもとの引数に加え，クロージャレコードを受け取るものとして定義する．

```
FUNCTION['main', [SYM['x']], PROGN[
  SYM['f'],
  SET[REF['f'],
      ABS[[SYM['x']], PROGN[
        RET[ABS[[SYM['y']], PROGN[
          RET[APP[REF['add'],
                  REF['x'], REF['y']]]]]]]],
  APP[APP[REF['f'], CONST[5]],
      CONST[10]]]]
```

図 11 構築される MIR の例
Fig. 11 An example of MIR.

(2) クロージャレコードを構築する．領域を確保し，それに (1) で定義した大域関数のアドレスと使用される外部変数へのポインタをコピーする．必要であれば外部変数のための領域を確保する．
(3) クロージャレコードのアドレスを返す

APP はクロージャレコードと引数をとり，クロージャレコードの最初の要素を関数として呼び出すコードに展開される．このとき，クロージャレコード自身も引数に追加される．

実装したコンパイラに対し以下の入力を与えると，**図 11** のようにマクロを含んだ MIR が構築される．ABS，APP のマクロ定義は合計 30 行程度であるが，この MIR は 1 命令 1 行として約 180 行の LIR へと変換される．

```
f=\(x){
  return \(y){ return add(x, y); };
};
f(5)(10);
```

6.2 クラスベースオブジェクト指向

本システムによるクラスベースオブジェクト指向言語の適用例として Java のサブセットである Featherweight Java[7] の実装を行った．Featherweight Java は以下の構文を持つ．CL はクラス定義，K はコンストラクタ定義，M はメソッド定義，e は式である．式には this の参照，変数参照，メソッド呼び出し，オブジェクトの作成，型変換が含まれる．詳しくは参考文献を参照されたい．

```
CL ::= class C extends C{{C f;} K {M}}
K  ::= C({C f}){ super({f}); {this.f=f;}}
M  ::= C m({C x}){return e;}
e  ::= x | this | e.f | e.m({e}) |
       new C({e}) | (C)e
```

また，基本構文を変更して this をともなわないメソッド呼び出し，フィールド参照およびローカル変数宣言・参照の機能を実装した．

実装に際し，スコープ管理のため，物理スコープとして class，fields，methods，method，local を導入した．物理スコープは以下のように構築される．

```
クラス定義 CL:
  class
    fields
      methods
メソッド定義 M:
  method
    local
      local
```

class，method は意味動作時の作業用情報が格納される．fields にはフィールドが，methods にはコンストラクタおよびメソッドが格納される．継承関係を表すために，これらでは，構築時に親クラスの持つ記号がコピーされる．メソッド定義での最初の local は引数のためのスコープ，その次の local はローカル変数用のスコープである．

論理スコープの定義は以下のようにし，変数の検索などに用いた．

```
THIS    = methods fields
LOCAL   = ^local+
```

また，クラス定義 CLASS，メソッド定義 METHOD，オブジェクト作成 NEW，メソッド呼び出し MCALL を MIR マクロとして定義した．本システムは，実マシン向けのコード生成を行うため，オブジェクトの構造やメソッド呼び出しなどの実装に関し，C++の手法[9),11)] を参考にした．

CLASS は，以下のように展開される．

(1) 記号表からスーパクラスの情報と，そのクラスに属するメソッドの情報を取得する．
(2) (1) で得た情報をもとに仮想関数テーブルを構築し，大域データとして定義する．
(3) このクラスのインスタンスの構築関数を定義する．この関数では，領域を確保し，その先頭に (2) の仮想関数テーブルへのポインタを書き込む．そして，その仮想関数テーブルを経由してコンストラクタを呼び出す．
(4) このマクロは上記を目的としており，式として扱われないので NOP を返す

METHOD は，メソッド名のマングリングを行い，そ

れを用いてメソッドを大域関数として定義する．式として扱われないので NOP に展開される．NEW はクラス名を受け取り，CLASS で定義されたインスタンスの構築関数を呼び出すコードに展開される．MCALL はレシーバ，メソッド名およびメソッドへの引数を受け取り，レシーバの仮想関数テーブルを経由し，メソッドを呼び出すコードに展開される．

実装したコンパイラに対し以下の入力を与えると，1 命令 1 行として 250 行程度の LIR へと変換される．上述のマクロの定義は合計で 50 行程度である．

```
class Id {
  Id() {}
  Object id(Object x) { return x; }
}
new Id().id(new Object());
```

このように，マクロ定義を用いることで複雑な LIR 構築を容易に記述することができた．

7. 関連研究

COINS は，高度な最適化を行い，またバックエンド機能を利用するためのインタフェースが簡潔であるため本研究では COINS を利用した．我々は以前，COINS の HIR をベースとしたコンパイラ生成システムの開発[16] を行った．しかし，HIR では高度な言語機能の実現が複雑になりやすく，これが MIR の開発の動機の 1 つとなった．本システムでは，高度な言語機能の実現のため，マクロ機能を実現したが，これは Lisp 系言語のマクロ機能を参考としている．また，MIR では複数命令を 1 つの式として扱うために，Lisp 系言語から PROGN を取り入れた．

既存のバックエンドインフラストラクチャ・フレームワークとして，COINS のほか，C-- [10]，SUIF [12]，CFL [15] などがあげられる．SUIF は主に研究分野において広く使われている．SUIF は独立した中間表現を持ち，中間表現の入出力インタフェースを備えている．SUIF では SUIF1 の開発が終了しており，現在はその後継となる SUIF2 の開発が行われている．C-- は，構文的に C 言語に似た，コード生成のための中間言語であり，様々な CPU と様々な言語のためのバックエンドとなるように設計されている．C-- には，例外処理や実行時インタフェースなど，バックエンド実現に利用できる様々な機能が備わっている．CFL は組み込み機器向けのコード生成のためのフレームワークとして設計されている．そのため，最終的な出力コードサイズが最小となるようにコードが生成される．また，GCC [5] が提供するコンパイラでは，内部で GENERIC/GIMPLE，RTL といった中間表現を用い，これらに対して諸々の最適化を行っている．これらは GCC から独立しておらず，GCC の機能をバックエンドに用いるには膨大なソースを理解する必要がある．

既存のシステムの応用例として，文献 13) では，外部から XML 形式で RTL を受け取れるように GCC を拡張し，GCC バックエンドを用いて最適化・コード生成を行う手法を提案している．しかし，システムの実現には至っておらず，その理由の 1 つとして GCC のドキュメントの不備をあげている．

8. おわりに

本研究では，COINS を用いて高度な言語機能を実現するための中間表現・マクロシステムおよび記号表の開発を行った．本システムを用いることで，複雑なコード変換をマクロとして定義することでき，より容易に COINS を利用することが可能となった．また，スコープ管理のための正規表現を用いることで，オブジェクト指向言語などの複雑なスコープ規則を持つ言語のスコープ管理を容易にした．

本システムは我々が開発している自己拡張可能構文解析器生成系をベースにその拡張として実装した．拡張として実装することで，構文解析器の生成などの基本的な機能を実装せずに，目的とする機能のみの実装に専念することができた．

今後の課題について以下に述べる．

MIR 自体に関しては，COINS を対象としている限り，基本的にこれ以上の拡張は必要でないと考えている．これは，MIR が独自の中間表現実現のための基底部分であり，現状で LIR の機能はカバーしているためである．その代わりとして，高級言語相当の機能を持つマクロセットを用意することを考えている．これは利用者の利便性のためであり，たとえば HIR 相当のマクロセットがあれば，利用者はこれを用いることで HIR 相当の制御構造を簡単に実現できる．

スコープ規則に関しては，今回はスコープポリシの集合としてのスコープ規則の定義というアイデアの提案のために，検索機能に重点をあて，記述を定義したが，さらにスコープの構築に関しても記述できるように記述を拡張することがあげられる．たとえば，オブジェクト指向言語の継承を実現するには，基底クラスの記号を継承クラスから参照できる必要がある．今回 Featherweight Java の実装ではこれを直接コードを記述することで実装したが，これには記号表クラ

スのインスタンスが持つ記号を別の記号表クラスのインスタンスへとコピーする方法をとった．このような実装は難しいものではないが，システムをより簡便に利用できるように，このような機能をシステムが直接サポートできるようにすることを考えている．これを実現するためにスコープ構築時のブロック間の関係を記述できるようにする必要がある．またこれにより，今回はサポートしなかった，C++などにおける `Namespace::Member` のような名前空間によるスコープ管理についてもサポートすることを考えている．

今後の展開としては，本システムを Java バイトコード，.NET の CIL などのコード生成に利用できるように本システムを発展させることがあげられる．そのためには，これらが持っている機能を利用するための命令の追加が必要となる．これには，たとえば，メソッド呼び出しのようなネイティブのオブジェクト指向機能などがある．本システムは，COINS を用いることを前提としているが，その内部実装は COINS から独立しているため，これは比較的容易に実現できると考えている．

参 考 文 献

1) arton: RubyJavaBridge. http://arton.no-ip.info/collabo/backyard/
2) coins 開発グループ：COINS — 並列化コンパイラ向け共通インフラストラクチャ. http://www.coins-project.org/
3) coins 開発グループ：COINS プロジェクト LIR 仕様書. http://www.coins-project.org/spec/lir.pdf
4) Gagnon, É.: SABLECC, AN OBJECT-ORIENTED COMPILER FRAMEWORK, Master's thesis, School of Computer Science, McGill University (1998).
5) Free Software Foundation: GCC, the GNU Compiler Collection. http://www.gnu.org/software/gcc/gcc.html
6) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Massachusetts (1994).
7) Igarashi, A., Pierce, B. and Wadler, P.: Featherweight Java: A Minimal Core Calculus for Java and GJ, *Proc. 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications* (*OOPSLA'99*) N.Y., Meissner, L. (Ed.), Vol.34, No.10, pp.132–146 (1999).
8) Johnson, S.C.: Yacc: Yet Another Compiler Compiler, *UNIX Programmer's Manual*, Holt, Rinehart, and Winston, New York, NY, USA, Vol.2, pp.353–387 (1979). AT&T Bell Laboratories Technical Report July 31, 1978.
9) Lippman, S. B.: *Inside the C++ object model*, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1996). 三橋二彩子ほか（訳）：C++オブジェクトモデル内部メカニズムの詳細，トッパン (1997).
10) Ramsey, N. and Jones, S.P.: A single intermediate language that supports multiple implementations of exceptions, *SIGPLAN Not.*, Vol.35, No.5, pp.285–298 (2000).
11) Stroustrup, B.: *The design and evolution of C++*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1994). 岩谷　宏（訳）：C++の設計と進化，ソフトバンククリエイティブ (2005).
12) The SUIF Group: The SUIF Compiler System. http://suif.stanford.edu/
13) 斉木晃治，権藤克彦：最適化器の生産性向上を目的としたコンパイラフレームワークの設計と実現，第 6 回プログラミングおよび応用のシステムに関するワークショップ (2003).
14) 中井研究室：拡張可能構文解析器生成系. http://nakai2.slis.tsukuba.ac.jp/hiki/
15) 中西恒夫，福田　晃，平尾智也：組込みシステムのためのコンパイラフレームワークライブラリ，平成 12 年度未踏ソフトウエア創造事業，情報処理振興事業協会 IPA (2001). http://www.ipa.go.jp/SPC/report/00fy-pro/projects/explorer/067/067.pdf
16) 舞田純一，佐藤　聡，中井　央：Ruby と拡張可能構文解析器生成系による COINS を用いたコンパイラの自動生成，第 5 回情報科学技術フォーラム (2006).
17) 舞田純一，佐藤　聡，中井　央：機能拡張可能なコンパイラ生成系，情報処理学会論文誌：プログラミング，Vol.47, No.SIG.16 (PRO 31), pp.1–9 (2006).

(平成 18 年 12 月 15 日受付)
(平成 19 年 3 月 23 日採録)

舞田　純一（学生会員）

1982 年生．2006 年筑波大学図書館情報専門学群卒業．筑波大学大学院図書館情報メディア研究科在学中（2006 年現在）．

中井 央（正会員）

1968 年生．筑波大学第三学群情報学類卒業，筑波大学大学院工学研究科修了（博士（工学））．1997 年 10 月図書館情報大学助手，2001 年 8 月同総合情報処理センター講師，2002 年 8 月同助教授，2002 年 10 月の筑波大学との統合により，筑波大学図書館情報メディア研究科助教授（学術情報メディアセンター勤務）．日本ソフトウェア科学会，ACM，ACM-SIGMOD-JAPAN 各会員．

佐藤 聡（正会員）

1991 年筑波大学第三学群情報学類卒業．1996 年筑波大学大学院工学研究科単位取得退学．同年広島市立大学情報科学部助手．2001 年筑波大学システム情報工学研究科講師．現在，同大学学術情報メディアセンター勤務．キャンパスネットワークの企画管理運用，ネットワーク，データベース，言語処理等の研究に従事．電子情報通信学会，ACM-SIGMOD-JAPAN 各会員．