SCK イントロダクション

未踏成果報告書を修正したもの

2007 年 4 月 10 日

目 次

1　要約

本プロジェクトではマルチソース、マルチターゲットのコンパイラ作成支援環境（コンパイラキット）を作成した。現在はソース言語として C 言語、ターゲットマシンとして x86 及び SPARC をサポートしているが、以下に述べる特徴により、他のソース及びターゲットのコンパイラ作成にも容易に適用可能である。まず第一に、提供されるソフトウエァ部品のモジュラリティーが高いこと、第二に現代的なパーサジェネレータ及びリターゲッタブルコード生成系を備えていること、そして第三にそれらが Emacs Lisp という強力な言語で実装されていることである。

2　背景及び目的

コンパイラキット (インフラストラクチャ) はユーザにとっての使いやすさが重要である。コンパイラの扱うデータは複雑であるから、ソースが公開されていて、ドキュメントが整備されているだけでは十分とは言えない。

例えば、現在最も広く使われているリターゲッタブルコンパイラである GCC は C 言語で実装された優れたオープンソフトであり、中間言語やリターゲッタブルコード生成 系のドキュメントも整っている強力なコンパイラであるが、いざ内部をいじろうとすると、複雑なデータ構造、グローバル変数の使用、メモリー管理の危険性などが障害となり、それは容易ではない。本来、GCC は強力で実用的なフリーのリターゲッタブルコンパイラを作成するのが目的であったから、コンパイラキットとしての視点、すなわち各部品のモジュラリティーの良さや簡潔さは考慮されていない。

もっとコンパイラ研究者にとって使いやすいコンパイラインフラを作ろうという反省から、COINS (COmpiler INfraStructure) プロジェクトが発足した。開発代表者も COINS プロジェクトに参加していた。COINS コンパイラはその点実装言語が Java になったことから改善されている。C 言語で実装されている gcc における複雑で危険なメモリー管理からは開放され、C 言語特有のトリッキーなポインタ操作も Java のオブジェクトレベルでの安全な操作で可能になっている。

一方で、我々は依然としてこれらのコンパイラキット（インフラ）には大きな問題点があると考える。それは我々のプロジェクトのタイトルにもある「実装言語からの独立性」である。モジュラリティーはどのようなソフトウエアでも重要であるが、特にソフトウエァの個々の構成部品を部分的に利用したり、あるいは差し替えたりする必要性があるコンパイラキットでは、モジュラリティーこそ使いやすさの最重要ポイントである。

我々の目的は後に説明するようなアプローチによって、実装言語から独立なで使いやすいコンパイラキットを作成することである。そしてそれがコンパイラ研究者のみならず、広く教育者や学生にも楽しく使ってもらえることを目指す。

3　プロジェクト概要

目標については以下の通りである。まず、以上のような、実装言語独立でモジュラリティーの良いコンパイラキットを作成するために、我々はまずソース言語として C 言語、ターゲットマシンとして X86 及び SPARC を対象としたリターゲッタブルコンパイラを作成することにした。現在 C 言語は最も広く使われている実用的なプログラミング言語であり、また重要なコンパイラ技術の多くを適用出来る簡潔な言語であるから、最初に実装するソース言語としては適切な選択である。最適化に関しては、開発期間が短いこともあり、多くを取り入れることは不可能であるが、命令選択とレジスタアロケーションというリターゲッタブルコンパイラの核になる部分については、現代的なコンパイラ技術の水準を満たすものを目標にした。ターゲットマシンは「マシン記述」と呼ばれるファイルで「マシン記述言語」に

より記述され、それからターゲットマシン依存部が自動生成されるという現代的なリターゲッタブルコンパイラの仕組みを取り入れる。

実現手段については、以下の通りである。実装言語独立の意味については後に説明するが、いうまでもなくコンパイラを実装する実装言語は必要である。コンパイラの実装言語として、我々は GNU Emacs の核言語である Emacs Lisp を採用した。Emacs Lisp は柔軟で強力であり、またエディタのコマンドとしての意味もあり、これは Emacs 上での簡潔なインタラクションメカニズムを提供している。そして各ソフトウエア部品間のインタフェースとしては S-式ベースの簡潔で実装言語独立なインタフェースを採用した。

開発結果については、以下の通りである。まずソース言語としては ANSI-C が実装出来ている。ただし現在ビットフィールド及び構造体引数がサポートされていない。これらの実装はターゲットマシンに強く依存するが、高級言語をターゲット独立の Core-AST (後述) に変換するという我々のアプローでは扱いにくく後回しになった結果である。ターゲットマシンに関しては X86 の浮動小数点演算がサポートされていない。これは演算自体が問題ではなく、X86 の浮動小数点レジスタがスタック構造になっており、通常のレジスタアロケーションの枠組みでは対処出来ないからである。ただし X86 は現在広く使われているハードであるから、何らかの対処、例えば MMX 命令を利用するとか、浮動小数点数はレジスタに乗せない、といった対処が必要である。また SPARC に関しては COINS プロジェクトで実績のある SPARC のマシン記述 (開発代表者が記述したもの) を基にしたもので、現在テストの準備をしている段階である。

他に、当初の予定には無かったがプロジェクトで作成したものとして、以下がある。まず LALR(1) に基づく (いわゆる yacc, bison と同等の) パーサジエネレータ yayacc である。これは C 言語以外のソース言語に対応する場合の強力なツールとなる。他に ATT のグラフ可視化ツール graphviz への Emacs Lisp からのインタフェースがある。これはさまざまなグラフ構造を扱うコンパイラにおいて、それらを可視化することが可能になり、デバッグ目的のみならず教育的な意味でも魅力的な要素となっている。

4 開発内容

ここでは本プロジェクトの技術的な部分について解説する。本コンパイラキットの名称 SCK は (S-expression based Compiler Kit) の略である。また SCK コンパイラとは SCK を用いて作られたひとつのコンパイラを意味する。

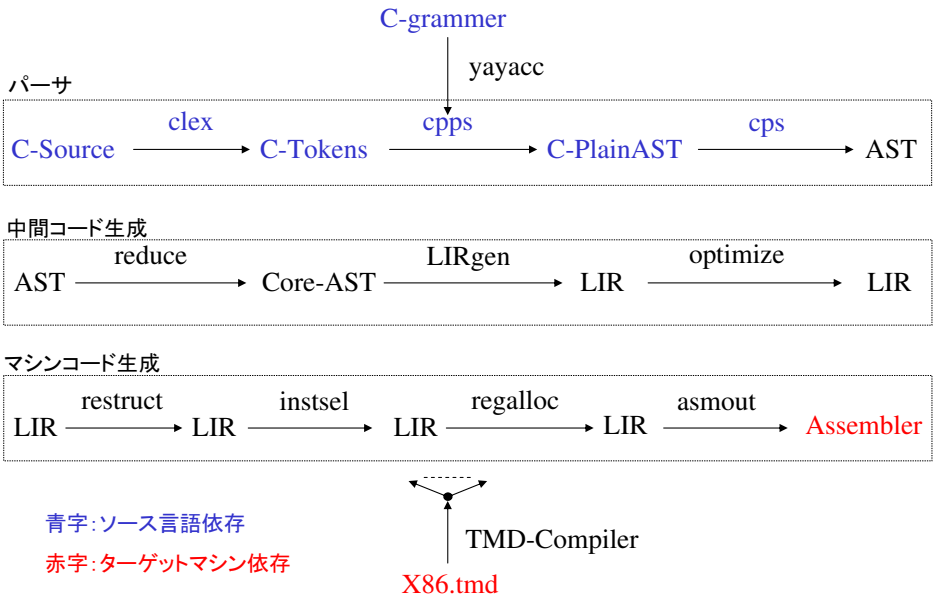
4.1 SCK コンパイラの全体構成

まず SCK コンパイラの全体構成を説明する。以下はソース言語が C 言語、ターゲットマシンが X86 の場合の例である。まずここでは全体について説明し、続くセクションで各部の詳細とモジュラリティーについて説明する。

一般的な用語で言えば、上段が「パーサ」、中段が「中間コード生成」、そして下段が「マシンコード生成」に対応する。まず C 言語のソース C-Source が clex により字句解析をされ、トークンのリスト C-Tokens に変換される。SCK の C コンパイラではプリプロセッサは提供せずに既成のものを利用するので、C-Source は正確にはプリプロセッサを通した C 言語のソースである。

トークンのリスト C-Tokens は cpps (C Plain Parser) により C-PlainAST に変換される。各部については後に詳しく説明するが、PlainAST は純粋に構文的な表現としての AST (Abstract Syntax Tree) であり、cpps は構文的なチェック以外、例えば変数が宣言されているとか、式の型が合っているとかいったチェックは一切行わない。現在は再帰的下向き構文解析法による手書きの cpps 及び、C-grammer から yayacc によって生成された cpps-yayacc の二つを利用することが出来る。yayacc については後に説明する。

SCK 構成図



C-PlainAST は C に依存した意味的なチェックを行う cps により、型の付いた AST (これを我々は単に AST と呼んでいる) に変換される。次に AST から型情報のみを残し、他の全てのシンタックスシュガー的な要素 (ネストした副作用、制御構造等) をより基本的な要素に置き換えたものが Core-AST である。

Core-AST はまだ高級言語としての型を保持しているが、それらをマシンレベルの型に落とすのが LIRgen である。LIR は後のフェーズで共通に使われる実装言語、ターゲットマシン独立の中間言語である。これは開発者代表が COINS の低水準中間言語として設計したものとほぼ同じものを利用している。

マシン独立の最適化、いわゆるコード最適化を行うのが optimize であり、これは LIR 上の変換として行われる。現在の SCK では分岐最適化と定数畳み込という基本的な最適化のみが実装されている。

下段はリターゲッタブルコンパイラの最重要部分、マシンコード生成部である。これらのフェーズの各部品は、部分的にマシン記述ファイル、この場合は X86.tmd から TMD コンパイラによって生成されたコードを取り入れることで、ターゲットマシンに特化された処理を行う。(より正確に言えば regalloc は X86.tmd からも、さらには LIR からも独立になっている (後述))。

ターゲットマシン独立の最適化が終了した LIR に対して行う最初のターゲットマシン依存の変換は restruct である。ここでは LIR のレベルで抽象的表現されてるが、実際のコード生成では具体化する必要のあるものをターゲットマシン依存のコードに変換する。具体的には関数の入口、出口、呼び出し、及びローカル変数のアクセスを具体的なフレームポインタ (X86 であれば ebp) によるアクセスに変換する処理である。次に命令選択 instsel がマシン記述ファイルで定義されている X86 に実在する命令のみでプログラムが実行出来るように LIR を変換する。

次にレジスタアロケータ regalloc により LIR の仮想レジスタが実レジスタに置き換えられる。この段階の LIR は事実上 X86 の命令と一対一に対応する命令の列に変換されているが、シンタックスは LIR のままである。そして最後に asmout が LIR をターゲットアセンブラの表記に変換して、コンパイルフェーズが終了する。SCK ではアセンブラ、リンカは提供せず、既存のものを流用する。実際には regalloc でレジスタ溢れ (spill) が起こる可能性もある。この場合はスピルの処理を行い、幾つかの仮想レジスタに対してはレジスタアロケーションをあきらめてフレームへ割り当て、命令選択部から繰り返

すことになる。この部分の詳細は構成図では簡略化されている。

4.2 パーサの詳細

パーサ (構成図上段) は C 言語を「構文解析」する部分である。しかし、一言で構文解析といっても、何を行うかはコンパイラの設計思想によって異なる。ここでは順を追って、SCK のデザインを解説していく。

まず、ソース言語はレキシカルアナライザ clex により字句解析される。SCK における C のレキシカルアナライザ clex は以下のような Emacs Lisp 関数である (名前の衝突を避けるため、SCK の関数には sck- というプリフィックスが付く)。少し冗長ではあるが、SCK の提供する実装言語独立なインタフェースの実例として手ごろな例でもあるので、全てを記載する。本来、S-式自体に型は無い。しかし SCK では以下のような BNF で記述された S-式の部分集合を型と考え、SCK が提供する関数も以下のように「型付け」されたものと考える。この型付けされた S-式の意味は Emacs Lisp のセマンティックスとは完全に独立な、非常に単純なものに限定する。複数の引数や返り値を必要とする関数が一般的であるが、SCK の部品として提供する関数は必ず一つの引数、一つの帰り値の関数とし、複数の情報はいわゆる A-Lisp の形でやりとりする。以下の例で言えば、clex は解析結果の他に、ソースポジション情報のために必要となるファイル名のリスト、さらにエラーメッセージのリストを返す必要があり、A-List の形のインタフェースとなっている。入力に関しても、項目は一つであるが拡張性を考慮して A-List の形としている。

```
sck-clex : <ClexInput> -> <ClexOutput>

<ClexInput>  ::= ((source <String>))
<ClexOutput> ::= ((tokens (<Token>..))
                  (file-names (<String>..))
                  (messages (<String>..)))

<Token> ::= <Keyword> | <Punctuation> | <Ident> | <Literal> | <PosInfo>
<Keyword> ::= <Symbol>
<Punctuation> ::= <Symbol>
<Ident> ::= (ident <String>)

<Literal> ::= <Iconst> | <Fconst> | <Cconst> | <Sconst>
<Iconst> ::= (iconst <Integer> <ISuffixes>)
<Fconst> ::= (fconst (<Ipart> <Fpart> <Epart>) <FSuffixes>)
<Cconst> ::= (cconst <CharCode> <CSuffixes>)
<Sconst> ::= (sconst (<CharCode>..) <SSuffixes>)
<{I,F,E}part> ::= <Integer>
<CharCode> ::= <Fixnum>
<ISuffixes> ::= [L | LL | U | UL | ULL]
<FSuffixes> ::= [F | L | LL]
<CSuffixes> ::= [L]
<SSuffixes> ::= [L]

<PosInfo> ::= ($ <FileNo> <LineNo>)
<FileNo> ::= <Fixnum> ; index of file-names
<LineNo> ::= <Fixnum>
```

通常、レキシカルアナライザのインタフェースは 1 トークンを読み込む関数と、その解析結果 (トークン種別、数や識別子ならその値や名前等) を受け取るためのグローバル変数の組みとして実装されるのが普通である。しかし我々の実装はソースを文字列としてもらい、結果を全て解析してリストとして返すという単純なものになっている。例えば以下のような (プリプロセス済みの) C-Source は

```
# 1 "poi.c"
unsigned long a=1ul;
char c='\z';
```

文字列として Lisp 関数に渡され、解析結果が関数の戻り値として返される。clex は Emacs Lisp の関数であるから、トップレベルループ (これは Emacs Lisp 付属のインタラクションモードではなく、SCK プロジェクト用に作成した機能強化版のトップレベルループであり、「->」は入力プロンプトである。このモードは SCK のライブラリで提供される) において、以下のように容易に動作確認が可能である。さらに、Emacs Lisp の利点を活かして、バッファのリージョンに対しても実行可能であるモジュールも幾つかある。これは我々の言う実装言語独立性とは無関係であるが、 Emacs で開発する上では非常に便利なインタフェースである。

```
-> (sck-clex '((source "# 1 \"poi.c\"\nunsigned long a=1ul;\nchar c='\\z';")))
((tokens
 (($ 0 1) unsigned long (ident "a") equ (iconst "1" UL) sem
  ($ 0 2) char (ident "c") equ (cconst 122) sem eof))
(file-names ("poi.c"))
("poi.c:2: unknown escape sequence '\\z'")))
```

ここで、`($ 0 1)` 等はファイルポジション情報である。また、`(messages (...))` は解析中のエラーメッセージである。パーサの各解析関数はエラーをこのように分離して次の解析関数へ渡し、最終的にマージされて一つのエラーメッセージになる。

このようにファイルの内容を一気に読み込んで、文字列として渡し、さらに解析結果も一つのリストとして一気に返すというインタフェースはたしかに実行効率やメモリー効率を犠牲にしているが、その犠牲はこのような単純で使いやすいインタフェースのためには、そしてさらに現在の CPU のパワーを考えると十分に割に合う犠牲であると考える。ファイル I/O は強く実装言語に依存するものであるが、コンパイラの処理においては本質的な部分ではい。

字句解析された結果である C-Tokens は純粋な構文解析系 cpps により C-PlainAST に変換される。ここで「純粋」という意味は、本来の構文解析以外の処理は一切行なわず、構文的な解析とチェックのみを行った結果を C-PlainAST として返す、という意味である。cpps は以下のような関数である。

```
sck-cpps : <ClexOutput> -> <CppsOutput>

<CppsOutput> ::= ((plain-ast <CPlainAst>)
                  (file-names (<String>..))
                  (messages (<String>..)))

<CPlainAST> ::= 略
```

以下に簡単な実行例を示す。実際には sck-clex の結果を合成するのであるが、同等の処理をより簡単に行なえる Emacs Lisp のコマンド版の `sck-cpps-{region,buffer}` の結果である。これを試すには、バッファに C のソースを書き込み、コマンドを実行すればよい。

```
int a = 1;
int f(int x)
{
  return a+x*3;
}
```

とバッファに書き込み `ESC x sck-cpps-buffer RET`

```
((plain-ast
  (module ($ 0 1)
   "stdin.c"
   (declaration ($ 0 1)
    ((keyword ($ 0 1) int))
    ((ident ($ 0 1) "a")
     (iconst ($ 0 1) "1")))
   (function ($ 0 2)
    (declaration ($ 0 2)
```

```
     ((keyword ($ 0 2) int))
     ((call ($ 0 2)
       (ident ($ 0 2) "f")
       (declaration ($ 0 2)
        ((keyword ($ 0 2) int))
        ((ident ($ 0 2) "x")
         nil)))
      nil))
    nil
    (block ($ 0 3)
     (return ($ 0 4)
      (add ($ 0 4)
       (ident ($ 0 4) "a")
       (mul ($ 0 4)
        (ident ($ 0 4) "x")
        (iconst ($ 0 4) "3"))))))))

(file-names ("stdin.c"))

(messages ()))
```

この cpps はこのように純粋に構文的な解析のみを行う。後に意味解析でのエラー処理のために、ソースポジション情報 (`$ ...`) が付加されているので一見複雑であるが、内容的には純粋に構文解析がやるべき仕事以外は一切行わない。意味解析、型解析等は後の cps によって行われる。cpps 及び C-PlainAST というレベルを設けることの利点は以下の通りである。

1. アクションの記述が単純になる
2. エラーリカバリ処理が容易になる
3. 他の言語で実装されたパーサとの接続が可能

非常に簡単なコンパイラは別として、実際のコンパイラで yacc を使うのは難しい。何が難しいかというと、それは本来 yacc が行ってくれる処理と、意味的な解析を同時に行うようなアクションを書かなければならないということ、及び意味的な処理のエラーリカバリと、構文的なエラーリカバリを同時に考慮した error アクションを記述しなければならないことの二点である。

我々のアプローチを取ることにより、これらの問題は解決されるのみならず、純粋なパーサ部分を他の実装と容易に置き換えることが可能となる。実際 C 言語で実装された cpps と接続することも容易に実現出来る。実際、最初に書かれた cpps は再帰的下向き方による手書きのものであったが、後に我々が開発したパーサジェネレータ yayacc により文法と、ごく単純なアクションの記述から自動生成された cpps と容易に置き換えることが出来た。SCK のパーサジェネレータ yayacc については、セクションを分けて説明する。

純粋に構文的な解析まで済んだ C-PlainAST は意味と型の解析を行う cps により高級言語としての型を持つ AST に変換される。cps は以下のような関数であり、これも Emacs のコマンド版が用意されている。

```
sck-pps : <CppsOutput> -> <CppOutput>

<CpsOutput> ::= ((ast <AST>)
                 (messages (<String>..)))
<AST> ::= 略
```

以下に実行例を示す。

```
struct data {
  int id;
  char *name;
  int flags[10];
```

```
};

struct node {
  struct node *next;
  int key;
  struct data data;
};

typedef struct node Node;

int flagok(int flags[10])
{
  int flag,i;
  for (i = 0, flag = 0; i < 10; i++)
    flag |= flags[i];
  return flag == 0xffffffff;
}

Node *search(Node *root, int key, int (*flagok)(int [10]))
{
  Node *p;
  for (p = root; p; p = p->next)
    if (key == p->key && flagok(p->data.flags))
      return p;
  return (Node*)0;
}
```

とバッファに書き込み ESC x sck-cps-buffer RET

```
(ast
  (module "stdin.c"
   (type "struct data" (struct ((var "id" int 0)
                                (var "name" (pointer char) 4)
                                (var "flags" (array 10 int) 8))
                         48 4))
   (type "struct node" (struct ((var "next" (pointer "struct node") 0)
                                (var "key" int 4)
                                (var "data" "struct data" 8))
                         56 4))
   (type "Node" "struct node")

   (var "flagok" (function ((pointer int)) int)
    (lambda (function ((pointer int)) int)
       ((var "flags" (pointer int)))
     (block ((var "flag" int)
             (var "i" int))
      (for "for0" ((comma int
                    (set int (vref int "i") (const int 0))
                    (set int (vref int "flag") (const int 0)))
                   (tstlt int (vref int "i") (const int 10))
                   (posf int
                    (add int (vref int "i") (const int 1))))
       (pref int
        (bor int
         (vref int "flag")
         (mem int
          (add (pointer int)
           (vref (pointer int) "flags")
           (vref int "i"))))))
      (return (tsteq int
                (conv uint (vref int "flag"))
                (const uint "4294967295")))))))

   (var "search" (function ((pointer "Node")
                            int
```

```
                          (pointer (function ((pointer int)) int)))
                 (pointer "Node"))
(lambda (function ((pointer "Node")
                   int
                   (pointer (function ((pointer int)) int)))
          (pointer "Node"))
   ((var "root" (pointer "Node"))
    (var "key" int)
    (var "flagok" (pointer (function ((pointer int)) int))))
 (block ((var "p" (pointer "Node")))
  (for "for1" ((set (pointer "Node")
                (vref (pointer "Node") "p")
                (vref (pointer "Node") "root"))
               (tstne int
                (vref (pointer "Node") "p")
                (conv (pointer "Node") (const int 0)))
               (set (pointer "Node")
                (vref (pointer "Node") "p")
                (member (pointer "struct node")
                 (mem "Node" (vref (pointer "Node") "p"))
                 (vref (pointer "struct node") "next"))))
   (if (land int
        (tsteq int
         (vref int "key")
         (member int
          (mem "Node" (vref (pointer "Node") "p"))
          (vref int "key")))
        (tstne int
         (call int
          (vref (pointer (function ((pointer int)) int))
           "flagok")
          (decay (pointer int)
           (member (array 10 int)
            (member "struct data"
             (mem "Node" (vref (pointer "Node") "p"))
             (vref "struct data" "data"))
            (vref (array 10 int) "flags"))))
         (const int 0)))
    (return (vref (pointer "Node") "p"))
    nil))
  (return (conv (pointer "Node") (const int 0))))))))
```

この AST は型解析が終了し、ある意味ソース言語を忠実に内部表現に変換したものであるが、GCC や COINS も含めて、その表現はコンパイラの実装言語である C (GCC) や Java (COINS) に依存しており、その実装言語のデータ構造でのみ定義されている。たとえ実装言語のみを使うことを想定したとしても、上記のような構造をもつデータを忠実に表現するヘッダファイルやクラスファイルは、この AST のプログラミング言語としての定義に比較してはるかに複雑であり、さらに他の言語とのインタフェースを取ることは困難である。さらに、そのような複雑なデータ構造に対しては分かりやすいデバッグダンプルーチンが用意されているのが普通である。

一方で我々の S-式ベースの表現はこの AST も含めて、全てがそれ自体真のデータであり、単なるデバッグダンプではなく (もちろん結果としてデバッグダンプとしても機能しているが) それはモジュール間のインタフェースに使われるデータ構造として正当なものなのである。もちろん、実装上は S-式として受け取ったデータを効率等の都合で別な型式 (e.g. Emacs Lisp の構造体) で表現し、内部で処理することは自由であり、さらにはこれらのインタフェースさえ合わせれば SCK の一部を別の言語 (e.g. Haskell) で記述することも容易であり、さらに別の言語から SCK のモジュールを利用することも可能である。

我々の言うところの「実装言語独立」の意味はほぼここまでの説明で理解していただけたと思う。我々はどのようなインタフェースでも受け付けるような柔軟なモジュールといった夢のようなことは目指してはいない。インタフェースは固定であり、厳密に定められていなければならないという点は他のコン

パイラインフラと同じである。ただし、インタフェースを言語として実装言語と独立に設計することで「モジュラリティーが良く使いやすい」コンパイラキットが作れるというのが我々の主張である。

4.3 中間コード生成の詳細

中間コード生成 (構成図中段) は、高級言語としての型から離れ、ターゲットマシンが直接扱えるような型及び演算を表現する中間言語、それも可能であればより効率的な中間言語に変換するという処理である。

構成図上段までで得られた AST は高級言語としての型を持ち、さらに C 言語に固有の制御構造 (e.g. `while`, `for` ,etc.) や特殊な式 (e.g. `x++` , `&&` , `||` , `x?y:z` ,etc.) の表現もそのまま残っている。

近年、レジスタプロモーションに代表される、高級言語の型が残っている表現を対象にすべき解析が重要になってきている。一方で、上記の AST を対象としてそのような解析をすることは困難である。高級な制御構造のままではフロー解析も出来ないし、また、式の形をしていても実際は制御構造であるような演算子、式の任意の場所で許される side-effect 等、特に C 言語は問題が多い。

しかしながら、そのような要素を取り除き (ただし重要な高級言語としての型は残す) より単純な形に変換することで、そのような解析は可能となる。このような理由から我々は AST から LIR へ直接変換を行うのではなく、間に Core-AST という表現を設けた。この Core-AST は AST から C 言語固有で解析の妨げになる以下のような要素を取り除いた AST のサブセットである。

1. `while`, `for` といった「高級な」制御構造
2. `&&`, `||`, `x?y:z` といった non-strict な演算
3. 式の副作用 (代入と関数呼び出し) は式のトップレベルで一回だけ起こる
4. ブロックはネストしない

AST は関数 reduce により、この Core-AST に変換される。先の AST を生成した C コードに対して生成される Core-AST は以下の通りである。

```
(ast
  (module "stdin.c"
   (type "struct data" (struct ((var "id" int 0)
                                (var "name" (pointer char) 4)
                                (var "flags" (array 10 int) 8))
                          48 4))
   (type "struct node" (struct ((var "next" (pointer "struct node") 0)
                                (var "key" int 4)
                                (var "data" "struct data" 8))
                          56 4))
   (type "Node" "struct node")

   (var "flagok" (function ((pointer int)) int)
    (lambda (function ((pointer int)) int)
       ((var "flags" (pointer int) nil register))
     (block ((var "_1" int nil register)
             (var "flag" int nil register)
             (var "i" int nil register)
             (var "_2" int nil register))

       (label "L1")
       (set int (vref int "i") (const int 0))
       (set int (vref int "flag") (const int 0))
       (goto "L2")

       (label "L2")
       (if (tstlt int (vref int "i") (const int 10))
```

```
        (goto "L3") (goto "L4"))

    (label "L3")
    (set int
     (vref int "flag")
     (bor int
      (vref int "flag")
      (mem int
       (add (pointer int)
        (vref (pointer int) "flags")
        (vref int "i")))))
    (set int
     (vref int "i")
     (add int (vref int "i") (const int 1)))
    (goto "L2")

    (label "L4")
    (if (tsteq int
         (conv uint (vref int "flag"))
         (const uint "4294967295"))
        (goto "L5") (goto "L6"))

    (label "L5")
    (set int (vref int "_2") (const int 1))
    (goto "L7")

    (label "L6")
    (set int (vref int "_2") (const int 0))
    (goto "L7")

    (label "L7")
    (set int (vref int "_1") (vref int "_2"))
    (return (vref int "_1")))))
```

以下略

Core-AST というレベルを設ける理由は以下の通りである。

1. 高級な型に依存する解析が容易に出来る
2. ソース言語からの独立性
3. LIR の生成が容易になる
4. LIR 以外の低水準中間言語の生成が容易になる

ここで、1. については既に説明した。また Core-AST のレベルでは、高級言語に必要とされるような型はあるが、一方で制御構造等のシンタックスシュガーと見なせる要素は排除されているので、ソース言語からの独立性は高いといえる。実際、Core-AST に新たな型システムを追加することで、広い範囲の高級言語に対応することが可能である。次に 3. であるが、経験上 AST から LIR のようなレベルに落とす過程で行われる処理は reduce の行う処理と高級言語の型をマシンで扱える型に「落とす」処理の二つから成り立っており、前者の処理がはるかに面倒である。よってこのレベルをモジュールとして分離することで、LIR の生成 (LIRgen) が容易になるばかりでなく、(4.) LIR 以外の低水準中間言語の生成も容易になる。先の例について LIRgen で生成された LIR は以下のようになる。

```
(module "stdin.c"
  (symtab ("flagok" static nil 4 "data" xdef)
          ("search" static nil 4 "data" xdef))

  (function "flagok"
     (symtab ("flags" reg i32 4 nil)
             ("_1" reg i32 4 nil)
```

```
        ("flag" reg i32 4 nil)
        ("i" reg i32 4 nil)
        ("_2" reg i32 4 nil))

(deflabel "L1")
(prologue (reg i32 "flags"))
(set i32 (reg i32 "i") (iconst i32 0))
(set i32 (reg i32 "flag") (iconst i32 0))
(jump (label i32 "L2"))

(deflabel "L2")
(jumpc (tstlts i32 (reg i32 "i") (iconst i32 10))
       (label i32 "L3") (label i32 "L4"))

(deflabel "L3")
(set i32
  (reg i32 "flag")
  (bor i32
    (reg i32 "flag")
    (mem i32
      (add i32
        (reg i32 "flags")
        (mul i32 (reg i32 "i") (iconst i32 4))))))
(set i32
  (reg i32 "i")
  (add i32 (reg i32 "i") (iconst i32 1)))
(jump (label i32 "L2"))

(deflabel "L4")
(jumpc (tsteq i32 (reg i32 "flag") (iconst i32 -1))
       (label i32 "L5") (label i32 "L6"))

(deflabel "L5")
(set i32 (reg i32 "_2") (iconst i32 1))
(jump (label i32 "L7"))

(deflabel "L6")
(set i32 (reg i32 "_2") (iconst i32 0))
(jump (label i32 "L7"))

(deflabel "L7")
(set i32 (reg i32 "_1") (reg i32 "_2"))
(epilogue (reg i32 "_1")))
```

以下略

先の Core-AST と比較して、基本的には型を落とす単純な処理で済んでいることが分かる。例えば、実行効率はあまり問題ではないような言語をスタックベースのバイトコードで実行させたいというような場合、レジスタベースの LIR はあまり適切な表現ではない。むしろそのような場合は Core-AST から直接バイトコードを生成する、というような使い方も可能であろう。

次に LIR 上の最適化変換 (SCK 構成図でいう optimize) が行われる。この部分に関しては、現在分岐最適化及び定数の畳み込み以外は実装されていないので、省略する。

4.4 マシンコード生成の詳細

マシンコード生成 (構成図下段) は、LIR からターゲットマシンのアセンブラコードを生成する処理である。ターゲットマシンはマシン記述言語により記述され (e.g. `x86.tmd`) TMD コンパイラにより生成された幾つかのコードが下段の各モジュールに取り込まれてターゲットマシンのコード生成が行われる。マシン記述は以下のように構成される。

1. 実レジスタ集合の定義

   実レジスタの名前や属性を定義する。一般に、命令によっては実レジスタの一部しかオペランドにとれないものがある。そのような制限されたレジスタ集合も全てここで定義しておく。

2. アドレッシングモードの定義

   命令の利用可能なアドレッシングモードを `defrule` 及び LIR のパターンとして記述する。`defrule` は一般に命令の一部としてのパターンに名前を付ける機能があるので、特定の範囲の定数なども必要ならここで定義する。

3. 命令の定義

   命令の定義を `defcode` 及び LIR のパターンにより記述する。`defrule` との違いは `defrule` で定義するものが、他の完結した命令の一部を記述しているのに対して、`defcode` は完結した命令を記述している点である。

4. restruct 関数

   マシンコード生成で最初に行われる処理である resutrct の処理を Emacs Lisp で記述する。ここの記述は実装言語、すなわち Emacs Lisp に依存している。

5. アセンブラコード出力関数

   ターゲットマシンに存在する命令単位で再構成され、実レジスタ割り当ても終了した LIR を、ターゲットアセンブラの表記に従って出力する関数を Emacs Lisp で記述する。ここの記述も実装言語である Emacs Lisp に依存している。

このように、SCK のリターゲッタブルシステム (コードジェネレータジェネレータ) は、書換え規則に基づいた典型的な構成であるが、記述上 `defrule` と `defcode` に分けて記述する点が異なっている。既存のリターゲッタブルシステムでは SCK における `defrule` でのみ全ての記述を行うが、我々は `defcode` から複数のパターンを自動生成することで、このようなシステムの記述に見られる冗長な記述を簡潔に記述することが可能となっている。詳細は付録の項目 LIR に上げた文献を参照されたい。最後の 2 つの記述において我々は Emacs Lisp の記述をそのまま利用しているため、その点でマシン記述は我々のいう実装言語からの独立性は保たれていない。ただしこの 2 つの箇所は非常に簡単な処理であり、現状ではモデル化、抽象化を試みるよりも強力な実装言語 Emacs Lisp による簡潔な記述を選択している。マシン記述が大体どのようなものなのかを概観してもらうため、X86 のマシン記述 (全体で約 700 行) から特徴的な部分を以下に抜粋しておく。

```
;;; 1. 実レジスタ集合の定義

(def *real-reg-symtab*
     ;; 実レジスタのシンボルテーブルエントリ。
     (symtab
      ;; general registers
      ;;
      ;; foreach は記述を簡潔にするためのマクロである。以下では
      ;; @g を eax ... edi のそれぞれに置き換えたもので本体
      ;; ("%@g" reg i32 4 nil) を置き換え
      ;;
      ;; ("%eax" reg i32 4 nil) ("%ecx" reg i32 4 nil) ...
      ;;
      ;; とマクロ展開する。
      ;;
      (foreach @g (eax ecx edx ebx esi edi)
        ("%@g" reg i32 4 nil))
      ;; special registers
      ("%ebp" reg i32 4 nil) ; frame pointer
```

```
      ("%esp" reg i32 4 nil) ; stack pointer
      ))

(def *reg-i32* ((foreach @g (eax ecx edx ebx esi edi)
                  (reg i32 "%@g"))))

(def *reg-i16* ((foreach @g (eax ecx edx ebx esi edi)
                  (subreg i16 (reg i32 "%@g") 0))))

(def *reg-i8* ((foreach @g (eax ecx edx ebx)
                 (subreg i8 (reg i32 "%@g") 0))))

(def *reg-eax-i32* ( (reg i32 "%eax") ))
(def *reg-ecx-i32* ( (reg i32 "%ecx") ))
(def *reg-edx-i32* ( (reg i32 "%edx") ))

...


;;; 2. アドレッシングモードの定義

;; register

(defrule reg reg (reg _ _)
  (asmarg (sck-tmd-x86-regname $0)))

(defrule reg-subreg reg (subreg _ _ _)
  (asmarg (sck-tmd-x86-regname $0)))

;; constant

(defrule con con (iconst _ _)
  (asmarg `(con ,(caddr $0))))

;; static

(defrule sta sta (static i32 _)
  (asmarg `(sta ,(caddr $0))))

;; constant or static

(defrule cs-con cs _con)
(defrule cs-sta cs _sta)

;; label

(defrule lab lab (label i32 _)
  (asmarg `(lab ,(caddr $0))))

;; base part of addr

(defrule base-cs base _cs
  ;; (base cs ())
  (asmarg `(base ,$1 ())))

(defrule base-reg base _reg
  ;; (base () reg)
  (asmarg `(base () ,$1)))

(defrule base-csreg base (add i32 _reg _cs)
  ;; (base cs reg)
  (asmarg `(base ,$2 ,$1)))

;; index part of addr

(defrule index-1 index _reg
  (asmarg `(index ,$1 1)))
```

```
(defrule index-2 index (mul i32 _reg (iconst i32 2))
  (asmarg `(index ,$1 2)))

...

;;; 3. 命令の定義

(defcode leal (set i32 _reg _addr)
  ;; アセンブラコード
  (asminst `(leal ,$1 ,$0))
  ;; 命令コスト
  (cost 1))

...

(foreach (@op @code) ((add addl) (sub subl) (band andl) (bor orl) (bxor xorl))
  (defcode @code (set i32 _reg (@op i32 _reg _mrc1))
    (regeq $0 $1)
    (asminst `(@code ,$2 ,$0))
    (cost 2)))
...

;;; 4. restruct 関数

(defun sck-tmd-restruct-prologue (exp lsymtab)
  ;; LIR の prologue (関数入口) 式の抽象的な表現を
  ;; X86 に依存した形で具体化する。
  (let ((org-args (cdr exp))
        (act-to-org-exps (make-queue))
        (act-offset (+ 16 4)))
    ;;
    (dolist (arg org-args)
      (let ((type (sck-lir-type arg)))
        (case (sck-lir-type-kind type)
          (integral
           (queue-append act-to-org-exps
                         (copy-tree
                          `(set ,type
                                ,arg
                                (mem ,type
                                     (add i32
                                          (reg i32 "%ebp")
                                          (iconst i32 ,act-offset)))))))
          (t
           (assert nil)))
        (incf act-offset (roundup (sck-lir-type-bytes type) 4))))
    (setq act-to-org-exps (queue-to-list act-to-org-exps))
    ;;
    (copy-tree
     `((prologue)
       ,@act-to-org-exps))))
...

;;; 5. アセンブラコード出力関数

(defun sck-tmd-asmout-emit-data (type data)
  ;; データの出力
  (ecase type
    ((i32) (insertf "   .long   %s\n" data))
    ((i16) (insertf "   .word   %s\n" data))
    ((i8)  (insertf "   .byte   %s\n" data))))

...
```

他のターゲットマシンについても、これと類似の記述を行い、TMC-Compiler でコンパイルし、システムに取り込むことにより、そのターゲットマシンのマシンコード (アセンブラ) を生成することが可能

である。

マシン記述に従った最初の変換は restruct である。これは LIR のレベルで抽象的表現されてるが、実際のコード生成では具体化する必要のあるものをターゲットマシン依存のコードに変換す処理であり、具体的には関数の入口、出口、呼び出し、及びローカル変数のアクセスを具体的なフレームポインタ (X86 であれば ebp) によるアクセスに変換する処理である。この処理はターゲット依存であり、特に難しい処理ではないが、一般的に定式化することが難しいので、先に説明したように Emacs Lisp で直接記述している (記述自体はマシン記述ファイル中に置かれる)。上記 X86 のマシン記述からの抜粋には prologue の restruct 関数がある。すなわち、現在の SCK では、このような処理を書くためには Emacs Lisp の知識を必要とする。

次に行われるマシン記述に従った変換は、リターゲッタブルコードジェネレータの核心部分である命令選択 instsel である。この部分についても、Emacs Lisp としてのコマンドが用意されており、どのような命令が選択されたのかを容易に確認することが可能である。以下に X86 のマシン記述を取り込んだ instsel の実行例を見てみよう。

```
int f(int *p, int a, int b, int c, int d)
{
  int x;

  x = p[a]*b+c*4+d;

  return x;
}
```

という C のコードの命令選択を実行してみる。
まず適当なバッファにこのプログラムを書き込み LIR に変換するために

```
ESC x sck-lirgen-buffer RET
```

とタイプすると、別のバッファに以下のような LIR が生成される。

```
(module "stdin.c"
  (symtab ("f" static nil 4 "data" xdef))

  (function "f"
     (symtab ("p" reg i32 4 nil)
             ("a" reg i32 4 nil)
             ("b" reg i32 4 nil)
             ("c" reg i32 4 nil)
             ("d" reg i32 4 nil)
             ("_1" reg i32 4 nil)
             ("x" reg i32 4 nil))

     (deflabel "L1")
     (prologue (reg i32 "p")
       (reg i32 "a")
       (reg i32 "b")
       (reg i32 "c")
       (reg i32 "d"))
     (set i32
       (reg i32 "x")
       (add i32
         (add i32
           (mul i32
             (mem i32
               (add i32
                 (reg i32 "p")
                 (mul i32 (reg i32 "a") (iconst i32 4))))
             (reg i32 "b"))
           (mul i32 (reg i32 "c") (iconst i32 4)))
         (reg i32 "d")))
     (set i32 (reg i32 "_1") (reg i32 "x"))
     (epilogue (reg i32 "_1"))))
```

ここでは restruct 変換は飛ばして instsel を実行してみよう。
instsel は LIR の関数 (function ...) 単位で行われるので、 (function ...) の先頭に行き

```
  ESC x sck-tmd-instsel RET
```

とタイプすると、それぞれの LIR の式についてマシン記述のパターンをダイナミックプログラミングにより最適にマッチングし、マシン命令に分割したものが表示される。以下はその出力の抜粋である。

;;; After tiling: (ダイナミックプログラミングによる最適なマッチング結果)

```
  1 set i32
  2 | * 17, stmt: (set i32 ($ 0 reg) ($ 1 addr)) ; leal/stmt
  3 +--reg i32 "x"
  4 |    * 0, reg: (reg ($ -1) ($ -1)) ; reg
  5 |      0, base: ($ 0 reg) ; base-reg
  6 |      0, addr: ($ 0 base) ; addr-base
  7 |      0, index: ($ 0 reg) ; index-1
  8 |      0, mr: ($ 0 reg) ; mr-reg
  9 |      0, rc: ($ 0 reg) ; rc-reg
 10 |      0, mrc0: ($ 0 rc) ; mrc0-rc
 11 |      0, mrc1: ($ 0 rc) ; mrc1-rc
 12 |      0, mrc3: ($ 0 rc) ; mrc3-rc
 13 |      0, callarg: ($ 0 reg) ; callarg-reg
 14 +--add i32
 15    | * 16, addr: (add i32 ($ 0 base) ($ 1 index)) ; addr-baseindex
 16    |   17, reg: ($ 1 addr) ; leal/reg
 17    |   17, base: ($ 0 reg) ; base-reg
 18    |   17, index: ($ 0 reg) ; index-1
 19    |   17, mr: ($ 0 reg) ; mr-reg
 20    |   17, rc: ($ 0 reg) ; rc-reg
 21    |   17, mrc0: ($ 0 rc) ; mrc0-rc
 22    |   17, mrc1: ($ 0 rc) ; mrc1-rc
 23    |   17, mrc3: ($ 0 rc) ; mrc3-rc
 24    |   17, callarg: ($ 0 reg) ; callarg-reg
 25    +--add i32
 26    |  | * 16, reg: ($ 1 addr) ; leal/reg
 27    |  | * 16, base: ($ 0 reg) ; base-reg
 28    |  | * 15, addr: (add i32 ($ 0 base) ($ 1 index)) ; addr-baseindex
 29    |  |   16, index: ($ 0 reg) ; index-1
 30    |  |   16, mr: ($ 0 reg) ; mr-reg
 31    |  |   16, rc: ($ 0 reg) ; rc-reg
 32    |  |   16, mrc0: ($ 0 rc) ; mrc0-rc
 33    |  |   16, mrc1: ($ 0 rc) ; mrc1-rc
 34    |  |   16, mrc3: ($ 0 rc) ; mrc3-rc
 35    |  |   16, callarg: ($ 0 reg) ; callarg-reg
 36    |  +--mul i32
 37    |  |  | * 15, reg: (mul i32 ($ 1 reg) ($ 2 mrc3)) ; mul-mrc3/reg
 38    |  |  | * 15, base: ($ 0 reg) ; base-reg

 ...............
```

;;; After relaxing: (X86 に実在する命令で分割した後)

```
(set i32
  (reg i32 "_2")
  (mem i32
    (add i32
      (reg i32 "p")
      (mul i32 (reg i32 "a") (iconst i32 4)))))
(set i32 (reg i32 "_5") (reg i32 "_2"))
(set i32
```

```
  (reg i32 "_5")
  (mul i32 (reg i32 "_5") (reg i32 "b")))
(set i32 (reg i32 "_3") (reg i32 "_5"))
(set i32
  (reg i32 "_4")
  (add i32
    (reg i32 "_3")
    (mul i32 (reg i32 "c") (iconst i32 4))))
(set i32
  (reg i32 "x")
  (add i32 (reg i32 "_4") (reg i32 "d")))


;;; Assembler instructions: (生成されるアセンブラ命令)

        movl    (p,a,4),_2
        movl    _2,_5
        imull   b,_5
        movl    _5,_3
        leal    (_3,c,4),_4
        leal    (_4,d),x
```

この最後のアセンブラ命令はまだレジスタアロケーションが終了していない段階でデバッグのためにマシン記述に定義されているアセンブラ出力関数を呼び出して表示したものである。実際にはこの instsel の後にレジスタアロケーションが行われることになる。

SCK のレジスタアロケータ regalloc のインタフェースはパーサの解説で説明したような S-式 (A-List) インタフェースであり、LIR からも完全に独立した部品となっている。レジスタアロケータの説明の前に、解説の流れ上上記の例の続きを完結しよう。この命令選択された LIR に対して regalloc を行い最終的にアセンブラ出力が得られるまでを以下に追加する。

実際の出力は (命令選択のダンプは非常にに冗長であるため、細部を省略し) SCK のコンパイル過程を一気に見ることの出来る Emacs Lisp コマンド `sck-cc-demo-{region,buffer}` の実行過程の抜粋である。

```
        int f(int *p, int a, int b, int c, int d)
        {
          int x;

          x = p[a]*b+c*4+d;

          return x;
        }
```

という `C` のコードに対して `SCK` の全フェーズの変換過程を見るためには、
適当なバッファにこのプログラムを書き込み

```
  ESC x sck-cc-demo-buffer RET
```

とタイプすればよい。以下は `regalloc` 以後のみの抜粋である。

```
;;; LIR after regalloc (with block re-ordering):

(module "stdin.c"
  (function "f"
     (symtab ("%esi" reg i32 4 nil &regset *reg-i32*) ; &regset ... は
             ("%eax" reg i32 4 nil &regset *reg-i32*) ; 命令選択で付加する
             ("%edx" reg i32 4 nil &regset *reg-i32*) ; レジスタアロケーション
             ("%ecx" reg i32 4 nil &regset *reg-i32*) ; で必要な情報
             ("%ebx" reg i32 4 nil &regset *reg-i32*))

     (deflabel "L1")
     (prologue)
     (set i32
```

```
        (reg i32 "%esi")
        (mem i32
          (add i32 (reg i32 "%ebp") (iconst i32 20))))

      ... 略 ...

      (epilogue (reg i32 "%eax"))))


;;; Assembler output:

# FUNCTION f
        .align  4
        .globl  _f
_f:
        pushl   %ebx
        pushl   %esi
        pushl   %edi
        pushl   %ebp
        movl    %esp,%ebp
        movl    20(%ebp),%esi
        movl    24(%ebp),%eax
        movl    28(%ebp),%edx
        movl    32(%ebp),%ecx
        movl    36(%ebp),%ebx
        movl    (%esi,%eax,4),%eax
        imull   %edx,%eax
        leal    (%eax,%ecx,4),%eax
        leal    (%eax,%ebx),%eax
        movl    %ebp,%esp
        popl    %ebp
        popl    %edi
        popl    %esi
        popl    %ebx
        ret
```

以上で SCK のコンパイル過程の詳細を説明したが、最後にレジスタアロケータのインタフェース及びコード生成について、少し補足する。レジスタアロケータ regalloc は以下のような関数である。ここで特筆すべき特徴は、このレジスタアロケータが完全に中間言語 LIR から独立に実装されている、という点である。

```
        sck-regalloc : <RegallocInput> -> <RegallocOutput>

        <RegallocInput> ::=
          ((interference-graph         ((<VirtualReg> <VirtualReg>)..))
           (coalesceable-edges         ((<VirtualReg> <VirtualReg>)..))
           (real-register-sets         ((<RealRegSet> (<RealReg>..))..))
           (real-register-constraints  ((<RealReg> <RealReg>)..))
           (possible-assignments       ((<VirtualReg> <RealRegSet>)..))
           (spill-priority             ((<VirtualReg> <Number>)..)))

          <VirtualReg> ::= <S-exp>
          <RealReg> ::= <String>
          <RealRegSet> ::= <Symbol>

        <RegallocOutput> ::=
          ((register-assignments ((<VirtualReg> <RealReg>)..)
           (spilled-registers    (<VirtualReg>..))))
```

SCK のレジスタアロケータは、グラフカラーリングベースで、逐次的に合併処理を行う (iterated register coalescing) 手法に基づいている。一般にレジスタアロケータは深く中間コードに依存したものになり、通常は生存区間解析、レジスタプライオリティティ計算 (どのレジスタを実レジスタに乗せるべきかの指標)、及びレジスタ溢れ (spill) 処理などと一体化されて実装される。

しかし SCK ではこれらは別々の部品として提供される。生存区間解析や、プライオリティ計算のために必要となるループ解析モジュール (これらは直接 SCK 構成図には現れていないが) も、全て上記ようなインタフェースにより LIR からの独立性を保っている。

そして、regalloc は本来レジスタアロケータが行うべき処理のみを行うように可能な限り分割されている。仮想レジスタ `<VritualReg>` は任意の S-式でよい。それらのペアとして表現された干渉グラフ `interference-graph` や合併可能な仮想レジスタペア `coalesceable-edges` ループ解析から推定されるプライオリティー `spill-priority` といった情報から、レジスタアロケーションを試み、結果を A-List として返す。

実際にはレジスタアロケーションが失敗 (溢れ) することもある。この場合は spiller が溢れ処理、すなわち溢れた仮想レジスタをフレームに再配置するという処理を行い、命令選択から繰り返すことになる。この繰り返しの部分は SCK 構成図では簡略化されている。

4.5 パーサジェネレータ yayacc

SCK の提供するパーサジェネレータ yayacc について、簡単に説明する。これは yacc あるいは bison といったパーサジェネレータと同様なパーサジェネレータの Emacs Lisp による実装である。

1. LALR(1) に基づく構文解析
2. アクションは合成属性で記述
3. 曖昧な文法も受け付ける (演算子優先順位)
4. エラーシンボルによるエラーリカバリ機構
5. C のフルスペックパーサ記述の実績 (cpps の yayacc 版)
6. オートマトン生成系と駆動系を完全に分離

ここで、SCK の設計思想的な意味での特徴として、6. の分離が上げられる。yacc や bison あるいは恐らく他の既存パーサジェネレータに共通する不満の一つとして、LALR(1) オートマトンを作成する部分が分離されていない、という点が上げられる。LALR(1) オートマトンの作成は複雑な処理であるが、その作成部のみ流用し、駆動系を独自に作りたいという要求に、既存のシステムは答えることが出来ない。例えば特殊なエラーリカバリ機構を備えた LALR(1) パーサを作りたい、あるいは研究したいという場合、オートマトンのテーブル作成自体に興味はないにもかかわらず、その部分のみを作成するモジュールは分離されていないので容易に流用することは出来ない (もちろん「分離作業」をすれば可能ではあるが)。同様に、yacc や bison は C 言語の駆動系を生成するが、他の言語 (e.g. Haskell) で駆動系を実装したいという要求に容易に答えることは出来ない。

yayacc は LALR(1) オートマトン作成部と、駆動系 (これは現在 Emacs Lisp のみ) が完全に、ここまで述べてきたような A-List インタフェースにより、分離されている。これにより駆動系の改良や研究が容易になり、また他の実装言語で記述された駆動系を利用することも容易となる。以下は yayacc で簡単な式の構文解析を行う文法定義の例である。

`yayacc` への入力

```
(defconst sck=yayacc-yacc-demo-input
  '((object-language "elisp")
    (associativity ((left ("+" "-"))
                    (left ("*" "/"))
                    (left (" NEG"))
                    ))
    (attribute-grammar
```

```
(((START E " EOF") $0)
 ((E E "+" E)      (list '+ $0 $2))
 ((E E "-" E)      (list '- $0 $2))
 ((E E "*" E)      (list '* $0 $2))
 ((E E "/" E)      (list '/ $0 $2))
 ((E "(" E ")")    $1)
 ((E "-" E)        (list '- 0 $1) " NEG")
 ((E "+" E)        (list '+ 0 $1) " NEG")
 ((E "number")     $0)))
(output-filename "yayacc-demo-output.el")))
```

生成されたパーサへの入力

```
(1+2)*3
```

構文解析結果

```
(* (+ 1 2) 3)
```

尚、この文法に対する LALR(1) オートマトンを graphviz インタフェースを利用して表示したものを付録に付けたので、参照されたい。このようなグラフが文法を書いてその場で可視化されるというツールは知る限り存在しない。LALR(1) 構文解析は実用的にも理論的にも重要な構文解析手法であり、このような可視化ツールはその理論を理解する上で大きな役割を果たすと思われる。

5 開発成果の特徴

SCK の特徴は大きく二つに分けられる。それはモジュラリティーの良さと開発環境の良さである。前者はコンパイラの部品を実装言語と独立な S-式ベースの単純なインタフェースを持つモジュールに分割することで、コンパイラ開発者、研究者にとって使いやすい部品を提供している。以下に主要なモジュールのモジュラリティー (独立性) をまとめておく。

	実装言語独立	C 言語独立	LIR 独立
yayacc-lir	Yes	No	Yes
cpps	Yes	No	Yes
cps	Yes	No	Yes
reduce	Yes	yes	Yes
tmd	No	Yes	No
loop	Yes	Yes	Yes
live	Yes	Yes	Yes
regalloc	Yes	Yes	Yes

一方、後者は Emacs Lisp という実装言語により SCK を実装したことにより、Lisp のもつ強力さ、ラピッドプロトタイピングのしやすさに加え、Emacs のインタラクティブな環境を利用出来るという点である。実際、今までの例で見てきたように Emacs のエディタコマンドとして手軽に個々の部品を実行し、テストすることが出来るという特徴は他に類を見ない。これは新たなコンパイラを開発する場合にも威力を発揮するし、また個々のコンパイルフェーズが行う処理を理解することも容易になるためコンパイラ教育の教材としても優れている。付属の　graphviz へのインタフェースによる可視化ツールは、デバッグのみならず、コンパイラの教材を作る上でも、また、コンパイラの授業をアクティブで楽しいものにする上でも、重要な価値がある。

6 今後の課題、展望

今後の課題、展望を重要と思われる順に述べる。

1. ドキュメント
   現在、ドキュメントに関しては各モジュールについて A-List インタフェースの定義等が散在している状態であり、早急にドキュメントの整備が必要である。我々は Emacs の texinfo によるドキュメント作成を予定している。texinfo でドキュメントを記述すれば、その一つのソースから、Emacs 上でオンラインで閲覧できるマニュアルと、出版物のソースとなる TeX の二つを生成することが可能である。

2. 完成度の向上
   デバッグ、X86 の不動少数点演算の不備の解消、SPARC 版のテストが必要である。これらについては、(実際の開発はまだ継続しているので)、あと 1 ヶ月ほどで完了出来る見込みである。コード生成部、特に命令選択部とレジスタアロケータは特に重要な部分であり、今後もプロジェクトが継続するかぎりテストと改良が最も必要な部分である。

3. コード最適化
   コード最適化については、分岐最適化と定数畳み込み以外実装出来ていない。今後、基盤部の完成度を上げることと同時に、上位レベルのコード最適化の実装が望まれる。ただし、我々の方針としては無節操にそれらを実装するのではなく、あくまでもモジュラリティーと簡潔さを考慮して慎重に実装して行きたいと考えている。

4. ターゲットマシン
   X86 のコードを生成する優れた処理系は多くあり、それらと競い合うことは目標としていない。また、SPARC についても、既に現在ではあまり使われていないので完成度を高める価値は無いと思われる。むしろまだコンパイラが存在していない、組み込み系の特殊な CPU を次のターゲットとして考えたい。

5. LIR インタプリタ
   現在、機能は制限されているが、コード最適化やレジスタアロケーションのテストには使える程度の LIR インタプリタが存在する。これを完全なものにして、メモリーアクセス等も完全にシミュレートすることが出来るならば、Emacs Lisp の上でいわばハードウエアのシミューレタが実装出来る事になる。なぜなら命令選択後の LIR はマシン記述に従ったターゲットマシンの命令を忠実に表現したものだからである。この延長線上には、ハードウエアのスペックを打ち込むだけで、そのマシンのオブジェクトコードが生成できるのみならず、実機が無くても Emacs 上で実機の動作を完全にシミュレート出来ることになる。

6. yayacc のエラーリカバリ
   yayacc はエラ発生を表す特殊な error トークンを文法中に記述することによるエラーリカバリ機構を備えている (yacc や bison と同様の手法)。しかし、このスタイルのエラー処理は非常に難しく、エラートークンの誤挿入により正しいプログラムがエラーになったり、あるいはその逆というケースすらしばしば発生する。我々のアプローチ、すなわち cpps と同様のインタフェースを取ることで、意味的な Undo 操作は不要になり、また入力トークンは任意個数先読みすることが可能であるから、yayacc にフルオートマティックなエラーリカバリ機構さえ導入出来れば、非常に強力なパーサジェネレータシステムを構築することが可能であると考えられる。

7. C 以外のソース言語
   マルチソースであることを示すという意味でも C 言語以外のソース言語に対応したい。構文解析

的な部分では先に述べた yayacc の拡張も関連する。 PL/0 のような型の単純なトイ言語に対してトイコンパイラを作ってみせるというのも、デモとしては重要であり、まずそのようなものを一つ作る必要がある。本質的な拡張としては、 Core-AST には新たな型システムを導入する必要があるが、まず最初にトライしたいのは、いわゆる Boxed 表現の導入である。これはタグオブジェクトを表現する上で重要であり、ML や Haskell といったソース言語に対応する際に必要となる。もし余裕があれば ML や Haskell の実装にチャレンジしたい。

7 実施計画書との相違点

当初、ターゲットマシンとして X86 及び SPARC のコード生成を予定していたが、開発期間中に SPARC はサポート出来なかった。また、X86 版も不動少数点演算が未サポートである。SPARC に関しては COINS プロジェクトで実績のある SPARC のマシン記述 (開発代表者が記述したもの) を基にしたもので、現在テストの準備をしている段階である。COINS プロジェクトとの互換性、インタフェースに関しても、達成出来ていない。コード最適化に関しては、当初予定していた共通部分式とループ不変式の最適化が達成出来ていない。

逆に、当初の予定には無かったがプロジェクトで作成した主な物としては、パーサジェネレータ yayacc、ATT のグラフ可視化ツール graphviz への Emacs Lisp からのインタフェース、Core-AST、LIR インタプリタ (機能制限版) がある。

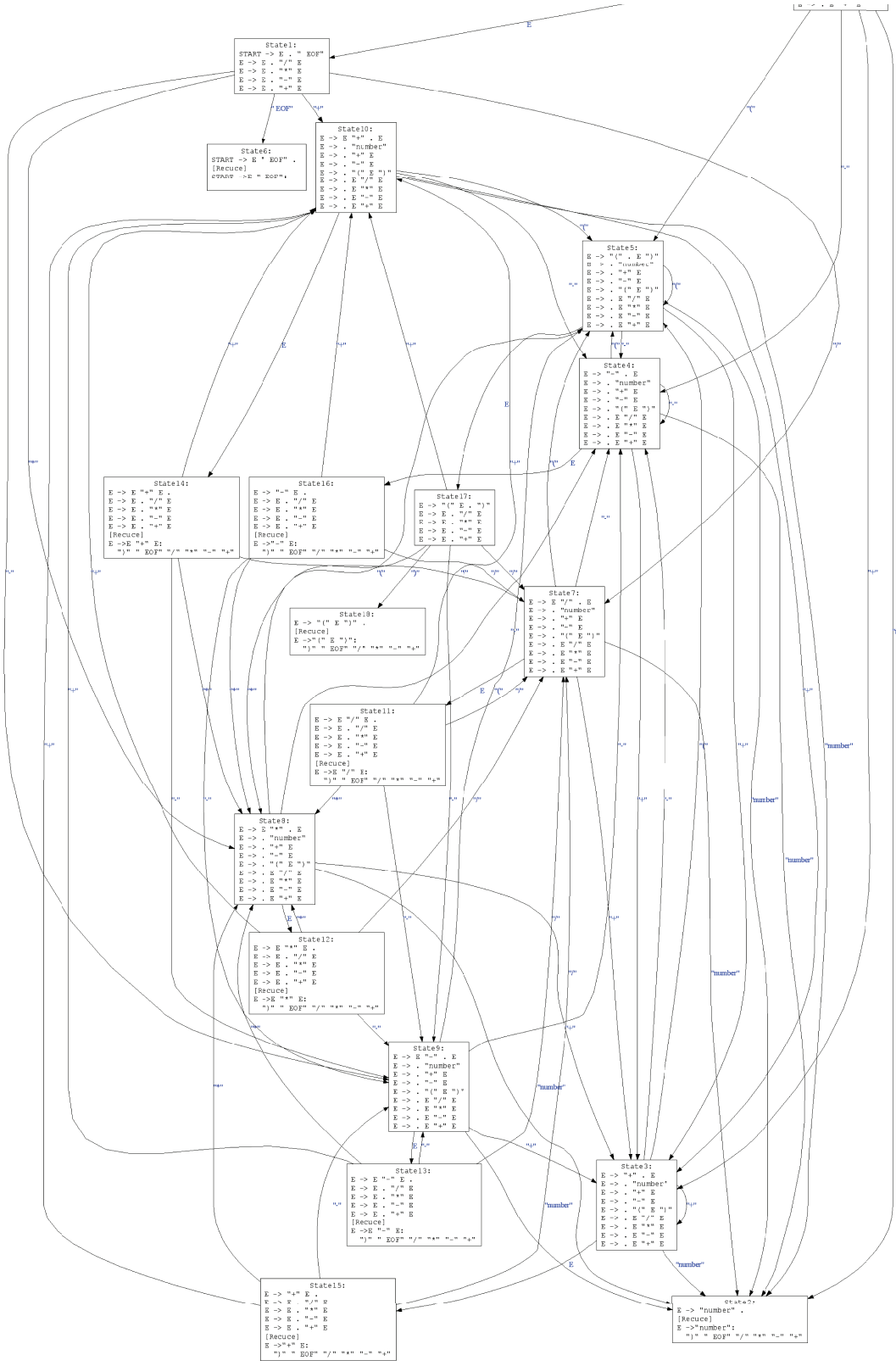
8 付録

1. SCK - S-expression based Compiler Kit.
   農工大並木研、東大萩谷研、の WeB ページにて公開予定

2. GNU Emacs - The extensible, customizable, self-documenting real-time display editor.
   `http://www.gnu.org/software/emacs/`

3. COINS - COmpiler INfraStructure
   COINS コンパイラプロジェクト
   `http://www.coins-project.org`

4. LIR - Lower level Intermediate Language
   COINS プロジェクトで使用されている低水準中間言語
   Seika Abe, Masami Hagiya and Ikuo Nakata: A Retargetable Code Generator for the Generic Intermediate Language in COINS, *IPSJ Transactions on Programming*, Vol.46, No.SIG14, 2005.



yayacc が生成した LALR(1) オートマトン