軽量下流CASEツール構築のための ソースプログラム表現形式の提案

吉 田 敦†

本論文では，汎用性や再利用性は考慮せず，作業対象のソースプログラムに特化した特定の作業のみを行う CASE ツールを軽量下流 CASE ツールと呼び，軽量下流 CASE ツールを構築するための開発環境の基盤として StreamCode を提案する．StreamCode は，プログラムの抽象構文木の 1 つの表現形式であり，スタック型仮想計算機に対する仮想命令の系列として表現する．CASE ツールは仮想命令系列に対する操作として実現し，具体的なツールの構築例を通して，軽量下流 CASE ツールが簡単なパターン置換で実装できることや，UNIX 系のツールとの親和性が高く，それらのツールとの組合せで実現できることを示す．

An Internal Expression of Source Program for Construting Lightweight Lower CASE Tools

Atsushi Yoshida†

In this paper, I propose StreamCode, which is an internal expression of source program for constructing lightweight lower CASE tools. A Lightweight lower CASE tool is a tool for a specific manipulation of target source programs, even though the tool has no generality and usability. StreamCode is a virtual code of a stack-based virtual machine, generated from an abstract syntax tree. CASE tools are described as manipulation of virtual code. I also show examples of lightweight lower CASE tool for explaining how briefly we can construct them by using pattern matching and by combination with UNIX tools.

1. は じ め に

ソースプログラムに対する編集を支援する下流 CASE ツールには，UNIX 系システムで用いられるエディタや indent などの整形ツール，ctags や GLOBAL[1] などのクロスリファレンス支援ツール，Eclipse[2] に代表される統合支援環境などが用いられている．これらは，様々なソフトウェアシステムの開発に共通する定型作業をツール化することで，開発作業の一部を自動化し，開発の効率や正確性を高める効果がある．

一般的に CASE ツールが支援しない作業においても，定型的な作業を繰り返すことがあり，ツール化ができれば作業効率の向上が期待できる．また，作業が機械的になるため，人為的なミスの混入を防止する効果もある．しかし，そのような作業は，開発対象のソフトウェアシステムに特化しており，ツール化しても再利用性が低く，ツール化するより手作業で行う方が短時間かつ低コストになることが多い．そのため，従来は，汎用的な作業のみがツール化の対象となっている．

ここで，再利用性は無視し，特定の作業を繰り返す必要があるときに，その作業専用のツールを構築することを考える．この場合，手作業とほぼ同等かより少ないコストでツールが構築できるのであれば，ツール化することに価値がある．また，多少，手作業より時間やコストがかかったとしても，人為的なミスを減らす効果を考えれば，ツール化することは望ましい．

そのためには，特定の作業に特化したツールを短時間に構築できる下流 CASE ツール開発環境が必要である．本論文では，そのような汎用性や再利用性は問わず，目的の作業に特化し，かつ，短時間に構築できる下流 CASE ツールを軽量下流 CASE ツールと呼び，その構築環境の基盤として，StreamCode を提案する．StreamCode は，CASE ツール内において作業対象となるソースプログラムの内部表現形式であり，ソースプログラムの書き換えに主眼を置いて設計された形式

† 和歌山大学システム情報学センター
Center for Information Science, Wakayama University

である．軽量下流 CASE ツールは，StreamCode 化されたソースプログラムを操作することで，目的の作業を達成する．

以下では，2 章で軽量下流 CASE ツールを定義し 3 章で StreamCode を提案，4 章で C 言語用の StreamCode の詳細について説明する．5 章では，軽量下流 CASE ツールの実装例を示し，6 章では実装例に基づいて StreamCode の有効性や問題点を議論する．

2. 軽量下流 CASE ツール

2.1 軽量下流 CASE ツールとは

ソースプログラムに対する編集作業の中心は，知的かつ創造的な作業であり，ツール化することは難しい．しかし，ソースプログラムの整形や，クロスリファレンス，デバッグ支援など，多くのソフトウェアシステムの開発に共通な作業は存在し，ツール化されている．また，プログラムの可読性や再利用性の向上など，ユーザに提供する機能の向上とは無関係に，プログラマの作業効率を高めるための編集作業としてリファクタリング[3) が存在し，様々な研究が行われている[4)．リファクタリングには，プログラムの意味を理解しなければ自動化できない作業も含まれるが，機械的な作業も多く，全体または部分的なツール化が可能である．実際，Eclipse の Java 開発環境には，一般的によく行われるであろうリファクタリングのツールがプラグインとして含まれている．

一方，個々のソフトウェアシステムの開発に着目すると，同じような作業を何度も繰り返すことがある．たとえば，関数のインタフェースの変更や，データ構造の修正は，修正するべき箇所がソースプログラム全体に分散しており，すべての修正箇所に対して特定の変更作業を繰り返す必要がある．このような作業を汎用化してツール化することは，プログラムの意味を理解する必要があるなどの困難な問題が多く，難しい．しかし，開発しているソフトウェアシステムに限定したツールであれば，プログラムの意味を知識としてツールに埋め込むことで，ツール化は可能である．ただし，このようなツールは汎用性がなく，構築の手間と効果を考えると，結局，手作業で修正が行われ，バグの混入など，問題の原因となりやすい．

そこで，下流 CASE ツールをきわめて小さな手間で開発できるようにすれば，たとえ汎用性がないとしても CASE ツールを構築することで開発効率を高めることができる．また，ツール化することで，作業の正確さも高めることができ，人為的なミスの混入を防止できる．さらに，多少の汎用性があって何度か利用されるツールであったとしても，短期間に継続的に利用するのでなければ，必要になるたびに構築する方が，ツールの管理の手間がかからない分，開発現場全体の運用コストは低くなる可能性がある．本論文の手法で構築する下流 CASE ツールは，このような汎用性を無視した，その場限りの簡易なツールであり，軽量下流 CASE ツールと呼ぶ．

2.2 CASE ツールの使い捨て

軽量下流 CASE ツールは，極端ないい方をすれば，使い捨てを前提とした CASE ツールである．この考え方自体は新しいことではなく，UNIX 系のシステムでは古くからシェルスクリプトとして実現されている．すなわち，コマンドライン上で，パイプなどを使って複数のツールを組み合わせてデータの加工を行ったり，複数のファイルに対して特定の作業を繰返し行った場合，そのときに記述したものは無名のツールであり，作業が終わればそのまま廃棄されるツールである．また，AWK[5) などのインタプリタ言語のプログラミングで使われる「一行野郎」のように，数行の記述で実現するツールも，再利用可能な形で管理することは求めておらず，むしろ必要になるたびに書くことが前提となっている．

しかし，CASE ツールについては，構文に基づいて作業を行う必要があるため，ツールの内部で抽象構文木を構築したり，外部のリポジトリに格納した抽象構文木の上を探索したりしながら作業を達成する必要がある．変数名を変更するだけのツールであっても，抽象構文木や記号表への参照を記述する必要があり，記述量も多く，簡単には記述できない．そのため，これまでは使い捨てを前提とした開発は行われていない．

「使い捨て」は，一見するとソフトウェア工学の基本的な概念である「再利用」と反するようであるが，使い捨てられるほど軽量化された CASE ツールの背後には再利用性の高い基盤の存在が前提となる．すなわち，UNIX 系の OS の場合には，sort や uniq などの汎用性の高い単機能なツール群が提供されている．また，それらのツールの入出力データは，基本的には，行を単位とした文字列であり，また，各行は空白やカンマなどの特定の記号で区切られた文字列のリストで構成されるなど，変更操作や参照が容易な汎用的な書式が用いられており，ツール間の連携が容易になっている．

よって，CASE ツールの「使い捨て」を実現するためには，抽象構文木の生成などの基本ツール群と，CASE ツールでよく使われるであろう単機能ツール群

を提供でき，また，そのツール間の連携が可能となるよう，汎用性のある入出力データの書式を用いることが必要である．本論文では，作業対象のソースプログラムの表現形式を工夫し，UNIX 系のツールと親和性が高い書式を用いることで，独自の基本ツール群や単機能ツール群だけでなく，UNIX 系のツール群とも組合せ可能にした，再利用性の高い基盤を構築することを目指している．

3. ソースプログラムの表現形式の提案

3.1 開発基盤としての要件

軽量下流 CASE ツールの開発基盤に用いる表現形式を決定するにあたり，以下の条件を考慮した．

- 抽象構文木の外部提供
  CASE ツールを軽量化するためには，抽象構文木はツールの外部から与えられることが必要である．内部で構築する場合，複数のツールを組み合わせたときに，各ツールがそれぞれ構文解析を行うため，実行効率が悪い．また，ツール間で，抽象構文木上の特定のノード群を作業対象とするときに，ノード群を一意に表現する特殊な表現が必要であり，実装が複雑化する．一方，抽象構文木が外部から与えられる場合には，抽象構文木そのものに印を付けることで，ノード群を一意に特定でき，ツールの実装が容易になる．
- API の不使用
  抽象構文木を取り扱うときに，抽象構文木をリポジトリなどに格納し，専用の API を使ってアクセスする方法が考えられる．しかし，API を前提とすると，開発環境が，その API が提供されたプログラミング言語に限定されるため，得意な言語で利用できなかったり，複数の言語の API を用意した場合にはそれらを保守する必要が生じたりするといった問題がある．また，API を前提とすると，UNIX 系 OS の基本ツール群との連携も難しくなる．よって，API がなくても抽象構文木を扱えるよう，ソースプログラムの表現形式に工夫をこらす必要がある．
- UNIX 系ツールとの親和性
  UNIX 系の OS には，様々な優れたツールが存在し，それらをパイプなどで組み合わせることにより，より高度なツールを実現できる．特に，UNIX 系のツールの多くは，行を単位とした単純なテキストデータを扱うことで，相互の接続性を高めている．よって，1 行に 1 つの構文要素が出現する表現方法を用いることで，UNIX 系のツールとの連携を期待できる．
- ソースプログラムのテキストの保持
  CASE ツールで，ソースプログラムを書き換える場合，書き換えた箇所以外は前の状態を保持する必要がある．すなわち，修正するたびに，コメントを失ったり，インデントが変わったりすることは，プログラマが望まないことが多いためである．さらに，C 言語のように前処理系が存在する場合，前処理前のソースプログラムの構文解析は難しく，前処理後のソースプログラムから抽象構文木を生成しつつ，前処理前のソースプログラムのテキストとの関係を維持しなければならない．少なくとも，書き換えた後のプログラムが前処理後の状態になるツールでは実用性がない．
- 動作的側面の支援
  CASE ツールを構築するときには，プログラムの構造的な側面に着目するだけでなく，実行時の動作的な側面に着目することがある．たとえば，解釈実行系を構築する場合や，部分評価を実装するときには，制御フローやデータフローなど動作的な側面の情報が必要である．単純に抽象構文木が与えられたときは，前者の側面が中心であり，動作的な側面を扱うためには，最低限，実行順序に関する情報を含める必要がある．

3.2 StreamCode

本論文では，前節の要件を満たすソースプログラムの表現形式として StreamCode を提案する．StreamCode は，基本的には，抽象構文木を深さ優先でたどったときに，戻りがけ順に各ノードに対応する仮想命令を生成した命令体系である．ただし，CASE ツールを作る便宜上，一部の構文要素に関しては，その開始位置を表す特殊命令を含めている☆．この特殊命令は，抽象構文木をたどるときに行きがけ順に生成する．命令の詳細は 4 章で説明する．また，C 言語のソースプログラムを StreamCode に変換した例を**図 1** に示す．

構文木を深さ優先戻りがけ順にした理由は，1 行が 1 命令からなるテキストとして表現することで，UNIX 系のツールとの親和性を高めることができるためである．また，一般的な手続型のプログラミング言語では，基本的に仮想命令の並びと実行順序が一致する．関数呼び出しや，条件分岐，繰返しなどで実行位置が大きく変化するが，移動先の指標となる命令が存在するため，実質的に動作的な情報が含まれている．さらに，

☆ 戻りがけ順に命令を生成しているため，終了位置には本来の仮想命令が存在する．

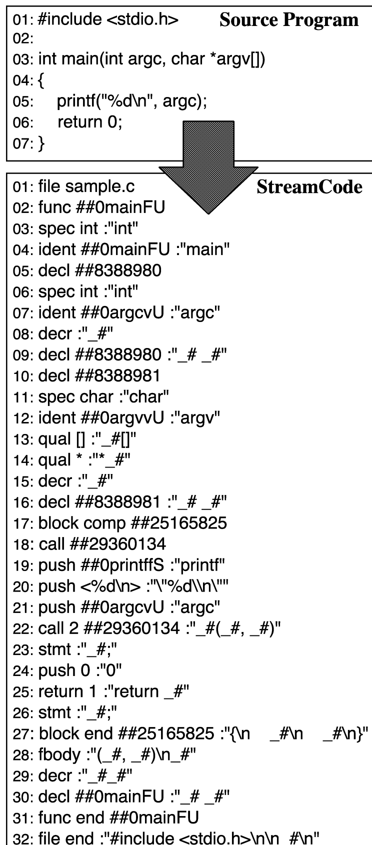

図 1　StreamCode の例
Fig. 1　An example of StreamCode.

構文要素間の親子関係が要素の並びによって表現され，直接，親子関係を表現する情報，すなわち構文木における辺が明示されないため，命令系列を書き換えるときに，親子関係を修正する作業が不要であり，書き換えがより容易に実現できるためでもある．

3.3　仮想命令の書式

仮想命令は，意味記述と表現記述の 2 つの部分から構成される．意味記述は命令の意味，すなわち，構文要素の意味を表現する．表現記述は，対応する構文要素がソースプログラム上にどのようなテキストの断片として現れているかを表す．構文要素が子の要素を持つ場合には，その子に該当する断片は穴として表現され，変換された命令系列の中でテキストの断片が重複して出現することはない．よって，命令系列の中のすべての断片を組み合わせることで，過不足なく，元のソースプログラムが再構成される．

また，各命令は正規表現を用いて構文解析を行うことができるため，読み込みのための特殊な API は不要である．たとえば，次の正規表現により，意味記述と表現記述を分離できる．

```
^([a-z]+ .*) :"(.*)"$
```

ここで，括弧は正規表現のグループ化を表すメタ文字で，最初の括弧の組に対応する文字列が意味記述であり，2 番目の括弧の組に対応する文字列が表現記述である．2 つの記述の間の空白を含めた 3 文字の組 “ :"” は区切を表し，この組合せは，どのような表現記述になったとしても出現することがなく，正確に 2 つの記述を分離できる．なお，表現記述内のダブルクウォートは直前にバックスラッシュを挿入してエスケープされ，また，改行文字など直接印字できない文字も同様にエスケープされた表現を用いる．

表現記述は，テキストを保持するための情報である．表現記述の表現方法として，テキストの断片を用いず，各命令に対応する断片の出現位置を用いる方法もある．しかし，出現位置は，ソースプログラムを書き換えると変化するため，整合性を維持するための作業が必要になる．一方，テキストを直接保持すれば，出現位置の整合性を維持する作業は不要である．ただし，欠点として，ソースプログラム中の特定の位置に対応する構文要素を探す場合には，テキストを再構築する必要があり，GUI ベースのツールを構築する場合などには実装が複雑になることがある．

3.4　ツール間の結合

StreamCode は，単純にテキストで表現するため，ツール間での StreamCode の受渡しはシェルなどが標準的に持つパイプや一時ファイルなどを介して容易に実現できる．また，1 行に 1 命令を記述するため，ツールを UNIX 系のツールと組み合わせたシェルスクリプトとして実装できる．

また，StreamCode の書式は簡易な正規表現で構文解析できるため，StreamCode の書式を一時的に拡張して，特定の作業に特化した情報を付加することができ，ツール作成時の自由度が高い．たとえば，ある作業を操作対象の命令系列を探索するツールと，変換を行うツールに分けて実装し，パイプで結合するように実装できる．このとき，前者のツールは求めた対象範囲の命令系列に独自の印を付加し，後者のツールは独自の印が付加された命令系列を解釈できるように実装する．ツールの内部表現として木構造など複雑な構造を用いている場合には，シリアライザなどによって，交換可能な形式に変換する必要があり，書式の拡張はシリアライザなどの入出力部分の実装に依存するため，自由度は低くなる．

4. C 言語のための StreamCode

4.1 命 令 体 系

命令体系は対象とするプログラミング言語によって異なるが，本論文では C 言語を対象とした命令体系を示す．基本的に，抽象構文木を命令体系として表現するため，この命令体系は他のプログラミング言語にも適用可能である．また，C 言語は前処理系があるため，前処理系を利用する他の言語にも応用可能である．

StreamCode は，**表 1** に示す 22 個の命令から構成される．各命令は，その命令名に続いてオペランドが続き，最後に表現記述で終わる．これらの命令は，Sapid[6),7)] の I-model を参考にしている．Sapid の I-model は，プログラマが直感的に理解している構文を表現したモデルで，実際の構文規則より大幅に簡略化されており，理解しやすい．また，命令のうち，macro は前処理のマクロ参照が存在するときに生成され，それ自体は C 言語の構文的な意味は持たない．また，nop 命令は，StreamCode を変更するときに使用する便宜上の命令であり，構文規則に対応する要素は存在しない．命令を削除するときに，表現記述の整合性を保てるよう，穴埋めの目的で使用する．

また，命令のうち，block，call，decl，file，func，member は開始位置を表す特殊命令として，表現記述を持たない形式の命令を使用する．たとえば，図 1 の StreamCode の 1 行目や 2 行目，5 行目が特殊命令である．開始位置を表す特殊命令が存在しなくても，終了位置から命令間の関係を逆にたどることで，開始位置を求めることができる．しかし，CASE ツールを構築するときには，開始位置があらかじめ分かっている方が作りやすいため，CASE ツールの構築経験を通じて必要と感じた 5 命令については特殊命令を採用した．特殊命令がないものでも，たとえば，stmt 命令であれば，開始位置は直前の stmt，block，decl 命令の直後であり，簡単なパターンマッチングで特定できる．

表 1 StreamCode の命令
Table 1 Operators of StreamCode.

命令	意味	オペランド
block	複合文，制御文	<種類> <ブロック識別子>
call	関数呼出し	<引数の数>? <呼出し識別子>
decl	宣言	—
decr	宣言子	—
fbody	関数本体	—
file	ファイル	(<ファイル名>|end)
func	関数定義	(end)? <記号識別子>
ident	識別子	<記号識別子>
init	初期化式	—
jump	ジャンプ	<ラベル識別子>
label	ラベル	<ラベル識別子>
list	リスト式	—
macro	マクロ参照	—
member	メンバリスト	(end)? <メンバリスト識別子>
nop	空式	—
op	演算	<演算子>
proto	プロトタイプ引数	<引数の数>
push	値の参照	(&?<記号識別子>|<定数>)
qual	型修飾子	<型修飾子>
return	リターン	<返値の数>
spec	型指定子	<型名>
stmt	ステートメント	—

表 1 のオペランドの中で，<ブロック識別子>，<呼出し識別子>，<メンバリスト識別子> はそれぞれブロック構造や関数呼び出しの開始位置と終了位置を一意に区別するための内部記号である．また，<ラベル識別子>は，ラベルを一意に区別するための内部記号である．<記号識別子>は，プログラム中の識別子を一意に区別するための内部記号であるが，識別子の特性を表す情報を含む．具体的には，識別子名と種類（変数，列挙子，メンバ，関数，タグ，型），有効範囲（局所，大域），定義者（ユーザ定義，システム定義）である．たとえば，図 1 では関数 main() の記号識別子は “##0mainFU” となっており，名前が “main” でユーザ定義の関数識別子であることを示している．なお，名前の前の数字は，同名の識別子が存在するときにそれらの間で区別するためのものである．これらの情報は StreamCode を解析すれば取得できる情報であるが，CASE ツールの構築の経験からあらかじめ用意した方がツールの構築が簡単になると判断したため，追加している．

4.2 解釈実行系への対応

命令体系を決定するにあたり，動作的な側面を表現できるよう，スタック型の仮想計算機の上で解釈実行することを想定した．そのため，値の参照はスタックに値を積む push 命令で表し，演算を表す op 命令は必要な値をスタックから取り出し，計算結果をスタックに積むと定義した．また，関数呼び出しについても，関数識別子の参照は関数へのポインタをスタックに積むことと定義し，call 命令は，関数へのポインタと実引数をスタックから取り出してから，関数呼び出しを行い，返り値がスタックに積まれた状態で戻ってくると定義した．実行可能であることは，簡単なインタプリタを作って確認した．

なお，命令系列の並びは，抽象構文木を深さ優先戻りがけ順にたどっているため，基本的に解釈する順序と一致する．ただし，論理演算子や 3 項演算子については，すべての部分式の命令の後に演算子の命令が出現するため，本来は解釈すべきではない式まで解釈す

ることになる．よって，インタプリタを作成する場合には，これらの演算子からなる式に実行制御のための特殊な命令を追加する変換フィルタを作成する必要がある．このことは，解釈実行系に適した命令体系と書き換え操作に適した命令体系が必ずしも一致しないことを示唆しており，今後も検討が必要である．

4.3 左 辺 値

解釈実行系に対応するため，左辺値と右辺値を区別している．具体的には，左辺値の式を，その式の抽象構文木の頂点に対応する命令の意味記述のオペランドに & を付加して表現する．ただし，左辺値が括弧で囲われている場合には，括弧内の式の抽象構文木の頂点を対象とする．たとえば，左辺が変数であれば，push 命令のオペランドの記号識別子の前に & を付加する．代入式 “x = y” を StreamCode で表現すると，以下のとおりになる．

```
push &##0xvU :"x"
push ##0yvU :"y"
op = :"_# = _#"
```

また，配列への要素や構造体（共用体）のメンバへの参照については，添字の参照やメンバの参照を表す記号を演算子としてとらえており，その演算子が抽象構文木の頂点となる．たとえば，代入式 “a[x] = b[y]” は以下のとおりになる．

```
push ##0avU :"a"
push ##0xvU :"x"
op &] :"_#[_#]"
push ##0bvU :"b"
push ##0yvU :"y"
op ] :"_#[_#]"
op = :"_# = _#"
```

なお，配列の識別子の参照は，配列の領域へのポインタ（アドレス）として定義し，添字記号を持つ op 命令は，添字式の評価結果を加算したアドレスを求め，& が付加されている場合には，そのまま値を返し，付加されていない場合には，そのアドレスが指す先の値を返すと定義した．

例外的に，アドレス演算子 & については，被演算数となる式の抽象構文木の頂点に対応する命令に & を付加する．アドレス演算子は，被演算数となる式の評価結果ではなく，式が表現する変数領域のアドレスを求める特殊な演算子であるためである．たとえば，式 “&x” を StreamCode で表現すると，以下のとおりになる．

```
push &##0xvU :"x"
op & :"&_#"
```

このとき，op & 命令には実行解釈の点では括弧と同様に意味を持たない．

4.4 制 御 文

if や while などの制御文はすべて block 命令で表現する．たとえば，if-else は以下のような命令群となる．

```
block if ##25165828
    ... 条件式...
block cond ##25165828
    ... 真節...
block else ##25165828
    ... 偽節...
block end ##25165828 :"if (_#) _# else _#"
```

この例の場合，##25165828 がブロック識別子を表し，すべての命令に付加されている．そのため，block 命令で表現される制御文や複合文が入れ子になっていても，同一のブロック識別子を持つ命令群を探せば対応関係が分かる．特に，perl の正規表現のように，後方参照が使える場合には，同一のブロック識別子を持つ命令を探索することはきわめて容易である．

また，条件式と真節の間の block cond 命令や真節と偽節の間の block else 命令は，条件式の終わり（真節の始まり）や，真節の終わり（偽節の始まり）を表す特殊命令である．他の制御文についても，if 文と同様に条件式や節から構成され，それぞれ特殊命令で区切っている．

4.5 表 現 記 述

表現記述には，各命令に対応するソースプログラムのテキストの断片が用いられる．断片には穴を含むものと含まないものがある．穴は，命令に対応する構文要素が子の構文要素（命令）を持つときに，その子の構文要素に該当する断片の存在を表す．子の構文要素に該当するテキスト断片は，対応する命令の表現記述に用いられるため，親の穴に子のテキスト断片をあてはめていくことで，完全なテキストを再構築できる．

穴は記号 “_#” で表す．ソースプログラムを再構築する際には，スタックを使用し，命令を上から下に向かって次の解釈を行うことで最終的にスタックにソースプログラム全体のテキストが積まれた状態になる．

- 表現記述（テキスト断片）に穴がない場合は，そのままテキスト断片をスタックに積む．
- 表現記述に穴がある場合は，穴の数だけスタックからテキスト断片を取り出し，後ろの穴から順次埋めていき，穴がなくなったらスタックにテキスト断片を積む．

例として，**図 2** に代入式 “x = a + b” の StreamCode および各命令ごとに上記の解釈を実施した直後

StreamCode	*stack*
push ##0xvU :"x"	x
push ##0avU :"a"	x | a
push ##0bvU :"b"	x | a | b
op + :"_# + _#"	x | a + b
op = :"_# = _#"	x = a + b

図 2 テキストの再構築
Fig. 2 Reconstruction of source text.

のスタックの状態を示す．

4.6 前 処 理

C 言語では，前処理の存在が前提となっている．ソースプログラムを正しく構文解析するためには，前処理後のソースプログラムで行う必要があるが，プログラマは前処理前のソースプログラムを編集しているため，前処理前のソースプログラムに基づいた StreamCode を生成できる必要がある．

そこで，まず，前処理後のソースプログラムを構文解析し，抽象構文木を深さ優先で探索しながら各命令を生成する．このとき，前処理によってテキストがどう変化したかを調べ，表現記述は前処理前のソースプログラムから切り出す．また，ある構文要素がマクロの展開によって生成されている場合には，そのマクロの参照を表す macro 命令を生成し，その構文要素の子については命令を生成しない．たとえば，以下のようなマクロ定義に対して，

```
#define ONE 1
```

以下のステートメントを変換すると，

```
ONE;
```

以下のとおりになる．

```
macro :"ONE"
push 1 :"_#"
stmt :"_#;"
```

このとき，push 命令は，除去しても元のステートメントを再構成できるが，マクロ ONE を展開した結果が 1 であることを意味するために存在している．これにより，前処理前のソースプログラムを変更するときに，前処理後の情報を利用できる．ただし，この効果はマクロの展開結果が 1 命令になる場合に限る．

また，ファイル包含のディレクティブ `#include` によって取り込まれた場合にも，その構文要素に対する命令は生成しない．なお，前処理後のプログラムの抽象構文木に基づいて生成するため，条件制御を行うディレクティブ `#if` などが含まれている場合には，解析時に指定された条件に合致する部分のみが解析対象となる．

一般的には，マクロなどは構文要素と適合するような使われ方をするため，多くのプログラムではこの手法で StreamCode に変換できる．しかし，前処理は，厳密には構文とは独立したテキストの書き換え処理であるため，構文要素を単位とした切り分けをする StreamCode とは，相性に問題がある．たとえば，マクロを展開した結果，抽象構文木の不完全な部分木に変換される場合には，変換できない．実例としては，以下のように，構造体が入れ子になっており，その入れ子を隠すためにマクロが定義されているときに，その入れ子のメンバへのアクセスを変換できない☆．

```
struct {
  struct {
    int _c;
  } _B;
} a;
#define c _B._c
```

このとき，式 “a.c” はマクロを展開すると “a._B._c” になるが，これは “(a._B)._c” と同等であるため，マクロ c の展開結果は，この式の抽象構文木のうち，“a.” を含まない不完全な部分木となる．

また，解釈実行の観点に立つと，前処理前のソースプログラムに基づいた StreamCode ではマクロが含まれるために解釈できない．前処理後のソースプログラムに基づいて StreamCode を生成すべきである．実際に，実装した StreamCode の変換系では，前処理後の StreamCode の生成も，オプションで選択できるようにしている．ただし，部分実行の結果に基づいてソースプログラムを書き換える場合など，前処理前と後の両方の情報が同時に必要になる場合への対応が十分ではなく，引き続き検討が必要である．

4.7 StreamCode 生成系の実装

C 言語のソースプログラムから StreamCode に変換するために，2 つのツール scat と tacs を作成した．scat は，コマンドライン引数に指定されたファイル，または，標準入力からソースプログラム読み込んで StreamCode に変換し，標準出力に出力する．tacs は，コマンドライン引数に指定されたファイル，または，標準入力から StreamCode を読み込んでソースプログラムを再構成し，標準出力に出力する．前処理

☆ たとえば，FreeBSD の net/if.h などネットワーク関係のヘッダファイルの中にこのような定義が散見される．

の解析や，構文解析については，Sapid の解析器を利用し，ツールは Sapid のパッケージに含まれるソフトウェア操作言語[8] で記述した．scat が約 1,400 行で，tacs が約 190 行である．なお，tacs は各命令の表現記述を解釈するのみであるため，どのような言語でも記述でき，perl で記述した場合には約 20 行であった．

また，約 2,400 行（前処理後に約 3,400 行）のソースプログラムを変換すると，StreamCode の命令数は約 9,600（前処理後で約 12,300）であり，構文解析なども含め変換時間は約 1.3 秒であった（FreeBSD 5.2.，CPU Pentium4 3.2 GHz，メモリ 1GB）．変換時間に関しては，十分に短く，軽量下流 CASE ツールの構築の基盤としては問題がないと考えている．

5. ツールの構築例

以下では，StreamCode を用いた CASE ツールの実現例を示す．

5.1 関数の実引数の追加

関数の実引数を追加する記述例を**図 3** に示す．このプログラムは以下のようなデバッグ情報の出力文を対象とする．

```
fprintf(stderr, "Debug: %s(%d)\n",
        msg(i), i);
```

すなわち，標準エラーに出力している fprintf 文で，出力が “Debug: ” で始まる文を対象とし，これを次のように書き換える．

```
fprintf(stderr, "Debug: %s(%d)[l.%d]\n",
        msg(i), i, __LINE__);
```

`__LINE__` は，コンパイラによって，ソースプログラム中の行番号に置き換えられる特殊なマクロである．fprintf 文は可変引数をとるため，引数の数は固定されず，それぞれ関数呼び出しや文字列定数が含まれることもある．そのため，単純なテキストとしての変換では実現が難しく，構文解析に基づいた変換が必要である．

図 3 は，大きく 3 つの部分から構成される perl スクリプトである．3 行目から 14 行目が書き換えの対象となる fprintf() の呼び出しに対応する StreamCode の正規表現の定義である．16 行目から 31 行目は書き換え後の StreamCode を生成するサブルーチンで，fprintf() の第 2 引数の文字列の書き換え，マクロ `__LINE__` の参照の追加，関数呼び出しの実引数の追加を行う．33 行目から 36 行目で，StreamCode の読み込みと置換を行い，最後にその結果を出力する．

この実装の中心は，置き換え対象の StreamCode のパターンと，置き換え後の StreamCode の生成であるが，これらは，いったん，置き換え前のプログラムや一部置き換えたプログラムを StreamCode に変換して，それらをもとに記述することで，簡単に記述できた．

```
01: #!/usr/bin/perl
02:
03: @pt = (  # target pattern
04:     "(",   # $1
05:     qq(call (\#\#\\w+)\n),   # $2
06:     qq(push \#\#0fprintffS :"fprintf"\n),
07:     qq(macro :"stderr"\n),
08:     qq(push \#\#0__stderrpVS :"_#"\n),
09:     ")",
10:     qq(push <(Debug: .*?)> :"\\\\"(.*?)\\\\""\n), # $3, $4
11:     qq(((.|\\n)*?)),   # $5, $6
12:     qq(call (\\d+) \\2 :(.*?)\n)   # $7, $8
13:       );
14: $pt = join("", @pt);
15:
16: sub gen()  # Generates new code for replacement.
17: {
18:    my ($head, $cid, $d1, $d2, $args, $argc, $argpt) = @_;
19:    ++$argc;  # Increases number of arguments.
20:    $argpt =~ s/\)"$/, _#\)"/;  # Digs hole in representation
21:    $d1 =~ s/(\\n)?$/ [l.%d]\1/; # add line number format.
22:    $d2 =~ s/(\\\\n)?$/ [l.%d]\1/; # add line number format.
23:    my @call = ( # new code template.
24:           $head,
25:           qq(push <$d1> :"\\"$d2\\""\n),
26:           $args,
27:           qq(macro :"__LINE__"\n),
28:           qq(call $argc $cid :$argpt\n)
29:          );
30:    return join("", @call);
31: }
32:
33: $code = join("", <>);  # Reads StreamCode.
34: # Substitutes code.
35: $code =~ s/$pt/&gen($1, $2, $3, $4, $5, $7, $8)/ge;
36: print $code; # Output new code.
```

図 3　行番号追加プログラム
Fig. 3　Program for adding line number.

5.2 その他のツール

これまでに，以下のような汎用性や再利用性のあるツールの構築も実験的に行った．軽量下流 CASE ツールの主旨に従えば，必ずしもツールに汎用性や再利用性を求める必要はないが，汎用性や再利用性はツールの作りやすさにも関係し，StreamCode の限界を見極めることができる．

- 識別子の変更
- 簡易な部分評価
- ソースプログラムの HTML 化
- 構文に基づく差分出力
- 関数の切り出し

識別子の変更は，変数名や関数名などをスコープ規則に従って正確に変更するツールである．目的の識別子を表す記号識別子を持つ命令を書き換えればよく，sed のスクリプトとして以下のように 1 行で実現で

きる．

```
sed -e '/α :"/s/ :".*"$/ :"β"/'
```

ここで，α には記号識別子を，β には変更後の識別子名が入るものとする．記号識別子を知るには，StreamCode を直接見るか，変数名と記号識別子との対応一覧を出力する簡単なプログラムを書けばよい☆．

簡易な部分評価では，定数式を計算して計算結果に置き換えたり，条件式が恒真または恒偽である条件文は簡略化するといった 7 つの変換を実現した．定数式の計算では，式に変数が混ざっている場合に，加算などの交換則が成り立つ演算に関しては辺の入れ換えを行って，できるだけ定数を集めるパターンも記述した．また，環境に依存した変換として，文字列の長さを返す関数 strlen() の引数が文字列定数だった場合には，文字列の長さを表す定数に置き換えるパターンを記述した．これらの変換操作は，それぞれ独立した変換パターンとして定義し，適用できなくなるまで繰返し適用することで，ソースプログラムを簡略化する．

HTML 化は，変数の参照から変数の宣言へのリンクを張ったり，予約語に色を付けるなどの作業であり，各命令ごとの書き換え規則を実現すればよく，22 命令分の変換規則を記述するのみである．

構文に基づく差分出力については，UNIX 系 OS では標準のツールである diff を使って StreamCode を比較することで，約 50 行のシェルスクリプトとして実現した．StreamCode がすでに構文を単位としたデータであるために，比較して得られる差分も必然的に構文を単位とした差分となる．ただし，そのままでは理解し難いため，ソースプログラムを HTML 化し，差分の部分に色を付けるようにした．具体的には，diff で ed が解釈できる形式の差分を求め，その差分に AWK を使って HTML の font タグを埋め込み，最後に差分を ed を使って StreamCode に適用している．StreamCode をソースプログラムに戻すと，差分の部分に font タグが入ったソースプログラムになる．

関数の切り出しは，文献 3) の Extract Method を C 言語に適用したものである．関数の切り出し範囲を指定するためには，GUI 環境が不可欠であり，Perl/Tk を使って実装した．この GUI 環境では，StreamCode を操作するツールをプラグインとして追加でき，各ツールには，ユーザが指定した対象要素の補正と StreamCode の操作をそれぞれ実装する．補正は，ユーザが目的の構文要素の一部を指定しても適切な範囲に修正する機能である．GUI 環境は約 300 行，関数切り出しツールは，補正が約 100 行，切り出し操作が約 650 行で，全体として約 770 行であった．

☆ 最も簡単な方法は grep で ident 命令を抜き出すことである．

6. 考　　察

本論文では，StreamCode を軽量下流 CASE ツールを実装するための基盤としているが，その有効性，すなわち，どのようなツールであれば簡単に作れるのかについて，また，ツールを作るうえでの問題点について，実装例の経験に基づいて考察を行う．

まず，StreamCode が表現している情報は抽象構文木であり，またすべての識別子については，記号識別子を付加することで区別しており，記号表に相当する情報も含まれている．そのため，原理的には，抽象構文木と記号表に基づいて行う作業はすべて実装可能である．

次に，軽量下流 CASE ツールは，簡単に実装できることが必要である．前章のツールのうち，識別子の変更や構文に基づく差分抽出は sed や diff，ed など UNIX 系のツールを用いることで，簡単なシェルスクリプトとして実現できた．これらは，単一の命令群に対する繰返し操作であったり，StreamCode から抽出する情報の構造が単純な操作であり，UNIX 系のツールを活用しやすいためである．

また，関数の実引数の追加や，簡易な部分計算，ソースプログラムの HTML 化は主に perl で記述し，これらは変換前と変換後の StreamCode のパターンを正規表現で書くことで実現している．これらは，抽象構文木上での局所的な構造に着目したパターン変換である．また，パターン化は，汎用性も求めると多少難しくはなるが，変換前と後のソースプログラムを用意し，それぞれ StreamCode に変換して正規表現で記述すればよく，比較的簡単である．ただし，StreamCode のパターン化は直感的に分かりやすい記述ではない．パターンを元のソースプログラムで記述し，StreamCode のパターンへ自動変換するなど，パターンの記述の支援は必要であり，検討を行っている．

前章で述べた実装例のうち，実装が最も難しかったツールは関数の切り出しである．特に，仮引数の型を決めるためには，切り出す範囲で参照している変数とその型を知る必要があるが，StreamCode には，変数と型の関係を直接結び付ける情報がなく，変数宣言を解析して求める必要があるため，全体の記述が長くなった．また，参照している変数を仮引数に置き換える作業では，構造体のメンバの参照など，複数の構文要素から構成される式を扱う必要があり，単純にパターンを固定できないため，手続的な探索プログラムを書く

ことになり，複雑化した．このように，パターンの定義が複雑であったり，正規表現では定義できないときには，目的の命令系列を手続き的に探すことが必要になり，StreamCode を使うことの優位性は低い．

また，簡単な部分評価において，変数に定数が代入されている場合には，その変数を定数として扱うような拡張を考えた場合，データフローに関する情報が必要となる．データフローの解析には，エイリアス問題など多くの問題があり，また，ポインタを含まない簡単なプログラムを対象にしたとしても，データフロー解析のプログラムは複雑であり，簡単には実装できない．むしろ，StreamCode を使うことでさらに複雑化する可能性がある．

以上のことより，StreamCode を有効に使える操作は，抽象構文木の局所的な構造に着目したパターン変換や情報抽出であり，これらは正規表現を用いて変換を実現したり，UNIX 系のツールで情報を加工したりすることで実現できる．一方，変数や式の型やデータフローなど，StreamCode を横断的に探索しながら解析する必要がある情報の抽出や，そのような情報を使用した操作には適していない．ただし，変数と型の関係や，作業対象の変数にのみ着目した部分的なデータフローの情報など，CASE ツールを作るうえで利用される典型的な情報を StreamCode に埋め込むフィルタを用意することで，局所的な構造に着目したパターン変換が可能になる可能性があり，今後も検討が必要である．

なお，軽量下流 CASE ツールを作るうえでの大前提として StreamCode の理解性の問題を考える必要がある．理解性が低いと，ソースプログラムと StreamCode の関係が分からず，ツール自体の記述が短かくなったとしても開発に時間がかかることになる．これまで数名の大学生や研究者に利用してもらった経験では，一般的なスタック型の仮想計算機の仕組みを理解できていれば，簡単な説明だけで大枠は理解できている．また，細かな部分については実際にソースプログラムを変換して見比べることで理解している．前章のツールのうち，関数の切り出しについては，大学 4 年生の学生が実装したが，表 1 を多少詳しくした説明のみを与え，あとは多少の試行錯誤を 1 人で繰り返すことで，短期間で習得できていた．よって，理解性については問題はないと考えている．

7. 関 連 研 究

軽量下流 CASE ツールが主に対象とするソースプログラムの書き換えは，プログラム変換の一部で Rephrasing と呼ばれる変換であり[9]，プログラムを同一のプログラミング言語の別のプログラムに書き換える変換である．プログラムの正規化や最適化，リファクタリング，改善（renovation）が同じ分野に含まれるが，それぞれ特定の目的に絞った書き換えであり，Rephrasing 全般のためのツール開発環境の研究は少ない．

本論文に近い研究としては，抽象構文木を XML 化し，既存の XML 技術を使って CASE ツールを構築する環境の提案がある[10)~12)]．しかし，文献 10)，11) は DTD を定義することやクロスリファレンサなど情報抽出を行うツールに中心が置かれ，ソースプログラムの書き換えをともなう CASE ツールの作りやすさについては議論していない．文献 12) は，リファクタリングに限定しており，一般的な CASE ツールの構築には言及していない．XML 化すれば，XSLT など XML を加工するための様々な技術が CASE ツール開発に利用できるが，発展途上であり，現状としては，CASE ツールの開発支援環境としては十分ではない．本論文では洗練された資産である UNIX 系のツールを利用して CASE ツールを構築することができる点で，CASE ツールを作りやすいといえる．ただし，XML の技術は急速に発展しており，将来的には立場が逆になることは否定できない．なお，本論文で，XML を採用していない理由には，StreamCode を考案した時点[13)] では XML が一般的な技術ではなかった点もある．

Eclipse の JDT や CDT などの環境では，内部に抽象構文木を持ち，様々な機能をプラグインとして追加できるため，汎用的な CASE ツールの環境としては優れている．しかし，パターンを記述する枠組みがないなど本論文が目指す軽量下流 CASE ツールを作る環境としては適していない．ただし，StreamCode を内部モデルとするソースプログラム編集環境を Eclipse 上で構築することは可能であり，検討している．

また，本論文の手法では前処理系に関する扱いが不十分であるが，C 言語のプログラムに対するリファクタリングに関しても前処理系が大きな問題であると認識されており[14),15)]，取り組みが行われている．これらの研究でも，マクロを含んだ抽象構文木を構築しようとしているが，完全な解決策には至っていない．

8. お わ り に

本論文では，再利用性を考慮せず，作業対象のソースプログラムに特化した軽量下流 CASE ツールを構築するための環境の基盤として，StreamCode を提案

した．StreamCode は，スタック型の仮想計算機に対する命令系列を用いた抽象構文木の表現形式であり，命令系列に対する操作として CASE ツールを実現する．命令系列は単純なテキストデータであり，特殊な API を用いず，UNIX 系のツールとの組合せや，perl によるパターン変換などを利用して，簡単にツールを構築できる．特に，抽象構文木の局所的な構造に着目したパターン変換や情報抽出に適している．

StreamCode の問題としては，前処理に関する情報や動作的な側面の情報の表現が不十分であることがあげられ，今後も検討が必要である．また，CASE ツールを作るときの開発環境の問題として，識別子の型を求めたり，データフローをたどったりするなど CASE ツールを作るうえで使われる典型的な情報の取得に問題があり，情報を取得するための支援ツールの充実が必要である．また，ソースプログラム中の特定の要素を指定したい場合には，GUI 環境が不可欠であり Eclipse などとの連携が必要である．

その他，本論文では述べていないが，コメントや前処理のディレクティブなど，仮想命令に対応しない要素を扱いたい場合があり，それらの要素を命令として取り出す変換ツールが必要であることが分かっている．また，構文要素を削除したときに穴埋めに使う nop 命令の除去や，構文要素を追加のための穴空けなどの典型的な操作を行う支援ツールも必要である．ただし，支援ツールを安易に増やすことは，保守性の低下を招くこともあり，必要最小限になるよう考慮する必要がある．

なお，StreamCode の変換ツールなどは，Sapid の配布パッケージに StreamedSapid として含まれており，公開されている．Sapid の配布パッケージは，Sapid のウェブページ[7] より入手可能である．

謝辞 本研究の一部は，文部科学省・科学研究費補助金（若手研究（B），課題番号 16700036）の助成を受けている．

参 考 文 献

1) Tama Communications Corporation: GNU GLOBAL source code tag system. http://www.gnu.org/software/global/
2) Eclipse. http://www.eclipse.org/
3) Fowler, M.: *Refactoring: Improving the design of existing code*, Addison-Wesley (1999).
4) Mens, T.: A Survey of Software Refactoring, *IEEE Trans. Softw. Eng.*, Vol.30, No.2, pp.126–139 (2004).
5) Aho, A.V., Kernighan, B.W. and Weinberger, P.J.: *The AWK Programming Language*, Addison-Wesley (1988).
6) 福安直樹，山本晋一郎，阿草清滋：細粒度ソフトウェア・リポジトリに基づいた CASE ツール・プラットフォーム Sapid，情報処理学会論文誌，Vol.39, No.6, pp.1990–1998 (1998).
7) Sapid Project: Sapid. http://www.sapid.org/
8) 吉田 敦，山本晋一郎，阿草清滋：CASE ツール開発のためのソフトウェア操作言語，情報処理学会論文誌，Vol.36, No.10, pp.2433–2441 (1998).
9) Bravenboer, M.: The Program Transformation Wiki. http://www.program-transformation.org/
10) H. Kawashima and K. Gondow: Experience with ANSI C Markup Language for a cross-referencer, *Proc. Domain-Specific Language Minitrack, 36th Hawaii Int. Conf. on System Sciences* (2003).
11) 吉田 一，山本晋一郎，阿草清滋：XML を用いた汎用的な細粒度ソフトウェアリポジトリ，情報処理学会論文誌，Vol.44, No.6, pp.1509–1516 (2003).
12) 丸山勝久：XML を用いた拡張性の高いリファクタリングツール，電子情報通信学会論文誌 D-I, Vol.J88-D-I, No.2, pp.175–185 (2005).
13) 吉田 敦，山本晋一郎，阿草清滋：ソースプログラムに対する変更操作が可能な細粒度ソフトウェアリポジトリの提案，ソフトウェア工学の基礎 V レクチャーノート/ソフトウェア学 20，pp.189–197, 近代科学社 (1998).
14) Garrido, A. and Johnson, R.: Challenges of Refactoring C Programs, *Proc. Int. Workshop Principal of Software Evoluation* (2002).
15) Vittek, M.: Refactoring Browser with Preprocessor, *Proc. European Conf. Software Maintenance and Reengineering*, pp.101–110 (2003).

(平成 16 年 9 月 29 日受付)
(平成 17 年 7 月 4 日採録)

吉田 敦（正会員）

1969 年生．1996 年名古屋大学大学院工学研究科情報工学専攻博士後期課程単位満了退学．同年豊橋技術科学大学知識情報工学専攻助手．2000 年より和歌山大学システム情報学センター講師．ソフトウェア開発環境の研究に従事．工学博士．