既存の構文解析器を利用した漸進的構文解析

対馬かなえ[1,a)]

概要：本稿では、漸進的構文解析 (incremental parsing) のように、一部だけ変更されたプログラムに対して構文解析を行う手法を提案する。incremental parsing との違いとしては、既存のコンパイラの構文解析器を使用して行う点である。そのため、特殊な構文解析器を作成する必要がなく、既存の言語を対象に一部だけ変更されたプログラムに対する構文解析の実装は容易になる。具体的には、コンパイラの構文解析器を使用して、構文同士の結合度による組み替え・他の構文の略奪の規則を導出し、それらが行われるプログラムを自動的に生成する。括弧等の抽象構文木の時点で失われる情報は人が補う必要があるが、それ以外のプログラムは扱うことができる。

キーワード：構文解析、漸進的構文解析、OCaml

1. はじめに

プログラミングにおいて静的エラーが果たしている役割は大きく、それらの検出により多くの実行時エラーが取り除かれている。静的エラーに関するデバッグはコンパイル時に行われることが多い。しかし、プログラマがプログラミング中にこれまでのプログラムを参照することは多く、デバッグの遅延はエラーの複製などをもたらすため、害悪である。プログラマの負担削減のためには、それよりも前の段階のプログラミング中にエラーの検出とデバッグを行う必要がある。

実際、Eclipse 等の IDE ではプログラミング時のエラー検出が行われている。しかしすべての言語でプログラミング中のエラー検出・デバッグが可能であるとは限らない。それらを実現するには、コンパイラの完全なプログラムに対する構文解析とは異なる、不完全なプログラムに対する構文解析が必要となる。

Eclipse 等でサポートされている機能としては、Syntax Error Recovery [3] が挙げられる。これは構文解析不可能になったならば、その先をいくらか無視し、その先から再び構文解析を開始する手法である。これはこれまでの構文解析器を利用できるため、軽量な手法である。しかし命令型言語のように、構文に対して区切りが多い言語に対しては有用であるが、 関数型言語のように構文の区切りが少なく、入れ子構造が多い言語に対しては無視される部分が多くなるため有用とはいえない。

関数型言語でのプログラミング中の検出とデバッグでは、Merlin [1], [4] が挙げられる。これは不完全な構文に対して足りない文字列等を適時内部的に補うことで、不完全なプログラムに対して構文解析を行っている。これは構文や構文解析に関して深い知識が必要となるため、それぞれの言語に特化した実装となり実装のコストが大きいという問題点が存在する。

本稿では、言語に特化せず、プログラミング途

[1] 国立情報学研究所
[a)] k_tsushima@nii.ac.jp

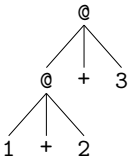
中の構文解析を行う手法を提案する。具体的には、コンパイラの構文解析器を使用し、プログラムの変更部分のみ再構文解析を行う。その結果をそれまでの構文木に組み込み、条件を保つように更新する。それによって、不完全なプログラムであっても、それまでに構造化していた部分を残すことが可能となる。

このそれまでの抽象構文木を用いるという手法は漸進的構文解析 [2] と類似している。違いとしては漸進的構文解析では特殊な構文解析器を必要とするが、本提案手法ではコンパイラの構文解析器を用いる。それにより、特殊な構文解析器を実装する必要がなく、軽量であるといえる。

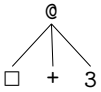
1.1 提案手法の概要

本節では、提案手法の概要について述べる。まず、１ ＋ ３ というプログラムを考える。このプログラムの抽象構文木は以下のように既に得ているものとする。

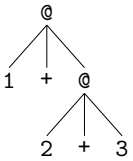
ここで、この式の一番初めの １ を １ ＋ ２ に変更する場合を考える (そのとき、プログラムのテキストは １ ＋ ２ ＋ ３ となる)。このとき、(1) 変更箇所である １ ＋ ２ のみ再び構文解析し、(2) 得られた抽象構文木を元プログラムの抽象構文木の一部を置き換える。これにより、以下の構文木が得られる。



このように、変更が起こった部分のみ再び構文解析し、元の抽象構文木に差し込むことで、新たな構文木が得られる。この方法では、もし変更部分が構文解析不可能であれば、その部分をホール (制約をもたないプログラム) で置き換えることで、それ以外の部分を保つことができる。たとえば、１ ＋ ２ を消した場合には以下のようになる。

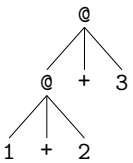


一部への変更がその部分のみに留まらず、変更が起こった部分以外へ影響する可能性も考えられる。例えば、プログラム １ ＋ ３ の ３ を ２ ＋ ３ に変更することを考える。この場合には、前述の素朴な方法では以下のような抽象構文木が得られるが、これはプログラム １ ＋ ２ ＋ ３ を構文解析した抽象構文木とは一致しない。



上の抽象構文木の問題点は、結合の強さに関する規則が守られていないことである。具体的には、足し算は左結合であるという規則が守られていない。このように部分的な再構文解析と挿入を行うと、全体として規則が守られなくなる可能性がある。

そのため、再構文解析を行った後には、全体として規則を保つような更新を行う。上の木に対してそのような更新を行うと、以下のような正しい抽象構文木を得られる。



3 節ではそのような変更規則を構文解析器から得る方法および反映する方法について述べる。

1.2 論文の流れ

まず 2 節で本論文で用いる構文を導入する。3 節では、1 箇所の構文解析可能な更新についての変更を反映する方法について述べる。具体的には、構文解析された一部分のプログラムが、どのようにそれ以外の部分に影響を与え、全体としてどのようなプログラムが得られるかについて述べる。4 節でまとめと今後の課題について述べる。

$$
\begin{array}{rcl}
(Op : operator) & ::= & \{+, -, *, ...\} \\
(M : term) & ::= & c^s \\
& | & M_1\ Op^s\ M_2 \\
& | & if^s\ M_1\ then\ M_2 \\
& | & if^s\ M_1\ then\ M_2\ else\ M_3 \\
\\
(s : strengh) & ::= & \{n, \{n, ..., n\}\} \qquad (n > 0)
\end{array}
$$

図 1 構文

$$
\begin{array}{rcl}
(M : term) & ::= & c \\
& | & if\ M_1\ then\ M_2 \\
& | & if\ M_1\ then\ M_2\ else\ M_3 \\
& | & (M)
\end{array}
$$

図 2 具象構文

2. 構文等の前提

本節では、本論文で用いる構文を導入する。構文は図 1 に示した。定数、二項演算、else 部分を持たない条件文、else を持つ条件文である。+ や ∗ のような演算子は Op で定義されているとする。

構文解析器に渡す具象構文は図 2 に示した。これは図 1 の構文に、括弧を追加したものとなっている。構文解析器はこの具象構文を受け取り、図 1 の構文構造に変換するものとする。

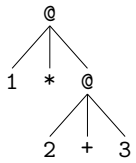
本論文中で扱う構文解析器の結合度は、∗、+、if 文の順で強く結合するものと仮定するが、もし異なる順であったとしても、それぞれのアルゴリズムは正しく動く。

3. 更新とそれによる変更

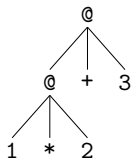
本論文では、1 箇所のみ更新した場合について考える。その更新が構文解析不可能であった場合には、ホール (制約をもたないプログラム) で置き換える。よって問題になるのはその 1 箇所の更新が構文解析可能なものであった時である。そのような場合、変更のあった部分のみ再び構文解析を行い、その変更を元の抽象構文木に組み込めば良い。しかし、1.1 節で既に見たように、そのまま組み込んだのでは正しい木とならないことも多い。本節ではそのような場合にどのように組み替えるか、および組み替えに必要な情報を既存の構文解析器を用いてどのように求めるか述べる。

3.1 組み替え

まず、`1 * 3` というプログラムの `3` を変更し、`1 * 2 + 3` とした場合の変更を考える。以下に示す抽象構文木は、再構文解析と挿入を行った結果、構文解析での結合度を守れていない例である。



この場合、より結合性の高い `*` の方が構文木の上の方にあるのが問題である。これを正しい木にするには、結合性を守るように組み替えを行えばよい。具体的には、`*` とその引数 `1`，`2 + 3` と、その部分式である `+` とその引数 `2`，`3` を組み替える必要がある。この際、`*` の方が強く結合するため、`2 + 3` の文字列としてはじめの部分 (ここでは `2`) を奪う格好になる。`2` が `*` の二項演算に奪われることによって、`1 * 2` という抽象構文木が得られる。`+ 3` はその外側に出現するコンテキストであるため、`(1 * 2)` を 1 つ目の引数として持つような以下の抽象構文木が得られる。



このような組み替えのために結合性に関する情報を導入する。結合性に関する情報は図 1 に示した s である。この一番はじめの部分がその構文自体の結合の強さであり、二番目の部分はその引数それぞれに要求される結合の強さである。数字が大きいほど強く結合していることを示している。例えば、(他に構文要素が存在しない時の) 足し算の二項演算の結合性は $^{\{1;\{1;2\}\}}$ となる。これは足し算では左結合であるため、一つ目の引数には足し算の二項演算がくることができるが、二つ目の引数 (右側) にはくることはできないことを示して

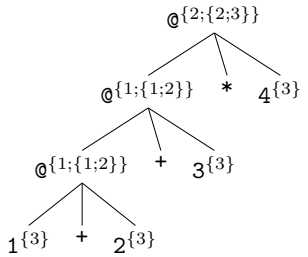
$$
\begin{array}{rcl}
decompose\_fst & : & strength * term \to term * context \\
decompose\_fst_s[exp] & = & let\ (f, r, c, cont) = decompose_l[exp]\ in \\
 & & (match\ (f, c)\ with \\
 & & \quad |\ (Some(f), \_) \to (f, fun\ x \to cont\ x\ l) \\
 & & \quad |\ (\_, Some(c)) \to (c, cont) \\
 & & \\
decompose\_last & : & strength \to term \to term * context \\
decompose\_last_s[exp] & = & let\ (f, r, c, cont) = decompose_l[exp]\ in \\
 & & (match\ (r, c)\ with \\
 & & \quad |\ (Some(r), \_) \to (r, fun\ y \to cont\ f\ y) \\
 & & \quad |\ (\_, Some(c)) \to (c, cont) \\
 & & \\
decompose & : & strength * term \to \\
 & & \quad term\ option * term\ option * term\ option * context \\
decompose_l[c^s] & = & (None,\ None,\ Some(c),\ \lambda x.x) \\
decompose_l[M_1\ Op^{\{s,ss\}} M_2] & = & if\ l > s\ then\ let\ (f_1, cont_1) = decompose\_fst_l[M_1]\ in \\
 & & \qquad let\ (r_2, cont_2) = decompose\_last_l[M_n]\ in \\
 & & \qquad let\ cont_3 = fun\ x\ y \to x\ Op^{\{s,ss\}}\ y\ in \\
 & & \qquad (f,\ l,\ None,\ fun\ x\ y \to cont_3(cont_1\ x)(cont_2\ y)) \\
 & & else\ (None,\ None,\ Some(M_1\ Op^{\{s,ss\}} M_2),\ \lambda x.x) \\
decompose_l[if^{\{s,ss\}}\ M_1\ then\ M_2] & = & if\ l > s\ then\ let\ (r_2, cont_2) = decompose\_last_l[M_2]\ in \\
 & & \qquad let\ cont_3 = fun\ x \to if\ M_1\ then\ x\ in \\
 & & \qquad (None,\ r_2,\ fun\ x \to cont_3(cont_2\ x)) \\
 & & else\ (None,\ None,\ Some(if^{\{s,ss\}}\ M_1\ then\ M_2),\ \lambda x.x) \\
decompose_l[if^{\{s,ss\}}\ M_1\ then\ M_2\ else\ M_3] & = & if\ l > s\ then\ let\ (r_2, cont_2) = decompose\_last_l[M_3]\ in \\
 & & \qquad let\ cont_3 = fun\ x \to if^{\{s,ss\}}\ M_1\ then\ M_2\ else\ x\ in \\
 & & \qquad (None, r_2, fun\ x \to cont_3(cont_2\ x)) \\
 & & else\ (None,\ None,\ Some(if^{\{s,ss\}}\ M_1\ then\ M_2\ else\ M_3),\ \lambda x.x)
\end{array}
$$

図 3 組み替えのためにプログラムを分解する関数 *decompose*

いる。

組み替えは図 3 および図 4 のプログラムを用いて行うことができる。*check_swap* はそれぞれのノードにおいて結合性が適切に守られているかをチェックし、適切でなければ適切になるように入れ替える関数である。*decompose* および *decompose_fst/last* は *check_swap* のための補助関数である。*decompose* は l よりも大きい結合性を持つ一番前の部分式、一番後ろの部分式、(そのどちらにも該当する場合には) 部分式本体、およびそれらを取り巻くコンテキストへ分解する関数である。*decompose_fst/last* は l よりも大きい結合性を持つ一番前の部分式もしくは一番後ろの部分式とそれらを取り巻くコンテキストを返す関数である。

例えば、以下の木で実行する場合を考える。



この木では、一番上のノードの掛け算とその下のノードの足し算の部分で結合性が適切ではない。掛け算の結合性情報は $^{\{2;\{2;3\}\}}$ であり、一つ目の

$$
\begin{array}{rcl}
check\_swap[c] & = & c \\
check\_swap[M_1^{s_1}\ Op^{\{l,\{l_1,l_2\}\}}\ M_2^{s_2}] & = & if\ s_1 >= l_1 \& s_2 >= l_2 \\
 & & then\ (check\_swap[M_1^{s_1}])\ Op^{\{l,\{l_1,l_2\}\}}\ (check\_swap[M_2^{s_2}]) \\
 & & else\ if\ s_1 < l_1\ then\ let\ (f, cont) = decompose\_fst_l[M_1]\ in \\
 & & \qquad let\ cont_2 = fun\ x \to x\ Op^{\{l,\{l_1,l_2\}\}}\ M_2\ in \\
 & & \qquad cont\ (cont_2\ f) \\
 & & else\ let\ (r, cont) = decompose\_last_l[M_2]\ in \\
 & & \quad let\ cont_2 = fun\ y \to M_1\ Op^{\{l,\{l_1,l_2\}\}}\ y\ in \\
 & & \quad cont\ (cont_2\ r) \\
check\_swap[if^{\{l,\{l_1,l_2\}\}}\ M_1^{s_1}\ then\ M_2^{s_2}] & = & if\ s_2 >= l_2\ then\ if^{l}\ M_1^{s_1}\ then\ check\_swap[M_2^{s_2}] \\
 & & else\ let\ (r, cont) = decompose\_last_l[M_2]\ in \\
 & & \quad let\ cont_2 = fun\ x \to if^{\{l,\{l_1,l_2\}\}}\ M_1\ then\ x\ in \\
 & & \quad cont\ (cont_2\ r) \\
check\_swap[if^{\{l,\{l_1,l_2,l_3\}\}}\ M_1^{s_1}\ then\ M_2^{s_2}\ else\ M_3^{s_3}] & = & if\ s_3 >= l_3 \\
 & & then\ if^{\{l,\{l_1,l_2,l_3\}\}}\ M_1^{s_1}\ then\ M_2\ else\ check\_swap[M_3^{s_3}] \\
 & & else\ let\ (r, cont) = decompose\_last_l[M_3]\ in \\
 & & \quad let\ cont_2 = fun\ x \to if^{l}\ M_1^{s_1}\ then\ M_2^{s_2}\ else\ x\ in \\
 & & \quad cont\ (cont_2\ r)
\end{array}
$$

図 4 必要に応じて組み替えを行う関数 *check_swap*

引数には 2 以上が要求されているが、実際には 1 の足し算の二項演算がきている。そのため、組み替えが必要になる。

この際、関数 *decompose* の対象となるのは、条件を満たしていない引数部分、`1 + 2 + 3` である。これを *decompose* を使用して 2 以上の結合の強さをもつ式とそれ以外のコンテキストへと分解する。分解すると、はじめの部分として `1` が、最後の部分として `3` が得られ、`(fun x y → (x + 2) + y)` がコンテキストとして得られる。このうち `3` が外側のコンテキスト `(fun z → z * 4)` に渡され、`3 * 4` が得られる。この組み替えの最終結果はコンテキスト `(fun x y → (x + 2) + y)` に `1` と `3 * 4` を渡して得られるもの、`(1 + 2) + (3 * 4)` となる。

3.2 組み替えの規則抽出

本節では構文解析器を使用して、組み替えの規則を自動的に抽出する方法について述べる。

まず、構文の構造をふたつ組み合わせたすべてのパターンを作成する。例えば、構文が掛け算と足し算のふたつのケースを考えると、以下の 8 通りの構文が作成できる。ここで _ は定数・変数等のそれ以上分解できない構文とする。紙面の省略のため、プログラム構造は括弧を用いて示す。

	プログラム構造	プログラム文字列
(1)	`(_ + _) + _`	`_ + _ + _`
(2)	`(_ * _) + _`	`_ * _ + _`
(3)	`_ + (_ + _)`	`_ + _ + _`
(4)	`_ + (_ * _)`	`_ + _ * _`
(5)	`(_ + _) * _`	`_ + _ * _`
(6)	`(_ * _) * _`	`_ * _ * _`
(7)	`_ * (_ + _)`	`_ * _ + _`
(8)	`_ * (_ * _)`	`_ * _ * _`

まず初期の結合度は `+` が `{1, {1; 1}}`、`*` が `{1, {1; 1}}` であるとする。上の文字列としてのプログラムを構文解析器にかけると、それぞれ構文解析結果が得られる。この結果を優先順位を使用して組み替えを行ったのち、もとの構造と比較し、違いを調べる。

初期の結合度は全てを許すため、組み替えは行われない。次に組み替えで得られた構造と、文字列を構文解析器にかけて得られた構造の比較を行う。(1), (2), (4), (6) は一致するが、(3), (5), (7),

(8) は一致しない。この違いを使用して結合に関する規則を求めていく。(3) の例を考えると、構文解析器による正しい構造は (_ + _) + _ であるが、組み替え後の構造は _ + (_ + _) である。このことから + の二つ目の引数には _ + _ が現れないこと、つまり、+ の二つ目の引数の結合度はより大きい必要があることがわかる。そのため、+ の結合度を {1, {1; 2}} と更新する。更新されるたびに、またはじめに作成したすべての構造と新たな結合度を用いて組み替えを行う。以下のプログラム構造が組み替え後のものである。

	組み替え前の構造	組み替え後の構造
(1)	(_ + _) + _	(_ + _) + _
(2)	(_ * _) + _	(_ * _) + _
(3)	_ + (_ + _)	(_ + _) + _
(4)	_ + (_ * _)	(_ + _) * _
(5)	(_ + _) * _	(_ + _) * _
(6)	(_ * _) * _	(_ * _) * _
(7)	_ * (_ + _)	_ * (_ + _)
(8)	_ * (_ * _)	_ * (_ * _)

これらを実際の構造と比較すると、(4), (5), (7), (8) が一致しない。(4) の例を考えると、構文解析器による正しい構造は _ + (_ * _) であるが、組み替え後の構造は (_ + _) * _ である。このことから * の一つ目の引数はより強い結合度が求められることがわかり、* の結合度を {1, {2; 1}} と更新する。これらの更新された結合度を使用して、またはじめから組み替えを行うと以下のプログラム構造が得られる。

	組み替え前の構造	組み替え後の構造
(1)	(_ + _) + _	(_ + _) + _
(2)	(_ * _) + _	(_ * _) + _
(3)	_ + (_ + _)	(_ + _) + _
(4)	_ + (_ * _)	_ + (_ * _)
(5)	(_ + _) * _	_ + (_ * _)
(6)	(_ * _) * _	_ * (_ * _)
(7)	_ * (_ + _)	_ * (_ + _)
(8)	_ * (_ * _)	_ * (_ * _)

これを実際の構造と比較すると、(6), (7), (8) が一致しない。(6) の例では、構文解析器による正しい構造は (_ * _) * _ となるが、組み替え後の構造は _ * (_ * _) である。これらから * の二つ目の引数の結合度を {1, {2; 2}} と更新する。すべての部分ノードの結合度が 2 となったため、* のノード自身の結合度も 2 と変更し、{2, {2; 2}} とする。この結合度を使用して、組み替えを行うと以下のプログラム構造が得られる。

	組み替え前の構造	組み替え後の構造
(1)	(_ + _) + _	(_ + _) + _
(2)	(_ * _) + _	(_ * _) + _
(3)	_ + (_ + _)	(_ + _) + _
(4)	_ + (_ * _)	_ + (_ * _)
(5)	(_ + _) * _	_ + (_ * _)
(6)	(_ * _) * _	(_ * _) * _
(7)	_ * (_ + _)	(_ * _) + _
(8)	_ * (_ * _)	_ * (_ * _)

これを実際の構造と比較すると、(8) が一致しない。(8) の例では、構文解析器による構造は (_ * _) * _ となるが、組み替え後の構造は _ * (_ * _) である。これらから、* の二つ目の引数の結合度をより高くし、{2, {2; 3}} と更新する。この更新された結合度によって、組み替え後の構造と、構文解析器で得られる構造が一致するため、最終的な結合度になる。

このように組み替えと、結果の比較による更新を行うことで、結合度に関する情報を構文解析器を用いて自動的に得ることができる。

図 5 に、結合度を求めるためのアルゴリズムを示した。*update_loop* がメイン関数である。これは構文を二つ組み合わせ、その文字列のリスト *str* と (組み合わせて作った) 抽象構文木 *ast* の組のリストを必要とする。このリストを *update_loop* の第一引数および第二引数として渡す。第二引数としても渡しているのは、途中で結合度の強さの更新が起こった場合には、またはじめから行うためである。*update_loop* では、*ast* を現在の結合度の情報で組み替えたものと、文字列としての *str* を構文解析器にかけたものを *update* に渡す。もし更新が起きた場合には *update* は true を返すため、

元のリスト *orig* を使用してはじめからやり直す。もし更新が起きない場合には、リストの他の要素について再帰する。リストに要素が無くなった場合には、最後まで結果が一致したことを示すため、最終的な結合度に関する情報が得られている。

update は組み替え後の構文木と、構文解析器によって得られた構文木を比較し、結合度の情報を更新する (ここでは結合度の情報は内部で保持され、破壊的に書き換えられていると仮定している)。基本的に構文解析器の木では c となっている部分に関して、部分式の結合度を 1 上げる。前者の木と後者の木が構文として一致しない場合には、組み替え後の構文木のノードが上に現れていることが問題であるため、式自体の結合度を 1 上げる。その際、部分式の結合度も同時に上げる。

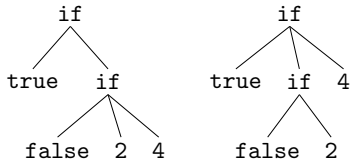
3.3 強奪

本節では、もう一つの変換規則について述べる。3.1 節および 3.2 節は構文と構文の組み替えであったが、本節は構文の一部が他の構文に取られるという更新である。

例えば、二種類の if 文を使用した以下の式を考える。

```
if true then if false then 2 else 4
```

このプログラムでは以下の二つの構文木の可能性が考えられる。



左は `else` 部分が内側の `if` 文に属しているもの、右は `else` 部分が外側の `if` 文に属しているものである。実際にこのプログラムを構文解析器にかけると、左の構文木となる。これは内側のものが優先されて構文解析されているためである。本節では右のような構文木を左のように変換する規則について述べる。

強奪は、構文木として直接接続している構文同士でなくても成り立つ。例えば、`if true then 1 + if false then 2 else 4` [*1] でも上と同じような二通りの if 文が考えられる。

図 6 に強奪の更新を行う関数を示した。本論文の構文では if 文に関してのみ強奪が行われる。そのため、この関数は式とその時の else 部分を持ち歩く (option 型であり、存在しない場合には None となる)。p は括弧がついている場合の結合度であり、それ以上である場合にはその外と中での else 部分のやりとりは行われない。それ以外の場合には、else 部分を持ち歩きつつ、else がない if 文を見つけると、その else 部分を埋めた式を返している。これをすべての部分を対象に順に行うことで、else の配置を適切に保っている。

if 文と同じく強奪が行われる match 文でたびたび観測されるが、match 分の入れ子を記述した際、ユーザの意図しない式の強奪によるエラーが発生する。match 文の場合には、外側の match 文のパターンの一部が内側の match 文によって強奪され、型エラーおよび意味的なエラーとなるケースがあるためである (そのためユーザは強奪されないように、括弧の挿入を行う必要がある)。本論文のような強奪の組み替えでは、検出することができるため、強奪時にユーザへ問い合わせることで、このようなエラーを防ぐことができると考えられる。

3.4 強奪の規則抽出

本節では強奪の規則を自動的に抽出する方法について述べる。強奪の規則の抽出では、文字列でのプログラムが重要になる。例として if-then-else 文 `if true then 1 else 1` を考えると、途中までの部分 `if true then 1` はそれ単体でプログラムとして受理される。そしてその残り部分 `else 1` は、if-then 文に接続することができ、if-then-else 文となる。このように、まず一部でも受理される構文を探し、その後に残りを受け取れる構文を探せば良い。

図 7 に強奪に必要な情報を集める関数 *grab_main* を示した。これらの関数では、プログラムを文字列のリストとして表現したものを用

[*1] この式は OCaml で型エラーとなるが、構文解析可能である

$$
\begin{array}{rcl}
update & : & term * term \to bool \\
 & & \text{(* 一つ目が組み替え後の構文木、二つ目が構文解析器の実際の構文木 *)} \\
update[c_1, c_2] & = & false \\
update[M_1\ Op_1\ M_n, N_1\ Op_2\ N_n] & = & if\ f[M_1, N_1] \\
when\ Op_1 = Op_2 & & then\ (\mathrm{Op}_2 \text{ の一つ目の部分式の結合度を 1 上げる; } true) \\
 & & else\ if\ f[M_2, N_2] \\
 & & \quad then\ (\mathrm{Op}_2 \text{ の二つ目の部分式の結合度を 1 上げる; } true) \\
 & & \quad else\ true \\
update[if\ M_1\ then\ M_2, if\ N_1\ then\ N_2] & = & if\ f[M_1, N_1] \\
 & & then\ (\text{if\_then の一つ目の部分式の結合度を 1 上げる; } true) \\
 & & else\ if\ f[M_2, N_2] \\
 & & \quad then\ (\text{if\_then の二つ目の部分式の結合度を 1 上げる; } true) \\
 & & \quad else\ false \\
update[if^s\ M_1\ then\ M_2\ else\ M_3, & = & if\ f[M_1, N_1] \\
\quad if\ N_1\ then\ N_2\ else\ N_3] & & then\ (\text{if\_else の一つ目の部分式の結合度を 1 上げる; } true) \\
 & & else\ if\ f[M_2, N_2] \\
 & & \quad then\ (\text{if\_else の二つ目の部分式の結合度を 1 上げる; } true) \\
 & & \quad else\ if\ f[M_3, N_3] \\
 & & \quad\quad then\ (\text{if\_else の三つ目の部分式の結合度を 1 上げる; } true) \\
 & & \quad\quad else\ true \\
update[M, \_] & = & (\text{M の式の結合度を 1 上げる; } true) \\
 & & \\
f & : & term * term \to bool \\
f[M, N] & = & if\ M = N\ then\ false \\
 & & else\ N = c\ then\ true \\
 & & else\ false \\
 & & \\
update\_loop & : & (((string\ list) * term)\ list) * (((string\ list) * term)\ list) \to unit \\
update\_loop[[], orig] & = & () \\
update\_loop[((str, ast) :: rest), orig] & = & if\ update[check\_swap[ast], \texttt{parse}\ str]\ then\ update\_loop\ orig\ orig \\
 & & else\ update\_loop\ rest\ orig
\end{array}
$$

図 5 組み替えに必要な結合に関する規則を求める関数 *update_loop*

いる。例えば、`if true then 1 else 1` は [”if”; ”true”; ”then”; ”1”; ”else”; ”1”] と表現される。関数 *rem_rule* はこのような文字列のリストのリスト (複数の構文情報) を受け取り、*rem_loop* を使用して、その文字列のリストの前方部分だけで成り立つ構文がないか探索する。存在する場合には、元の構文、前方部分だけの構文、残りの文字列のリストを返す。*rem_rule* は最終的には、(元の構文, 前方部分だけの構文, 残りの文字列のリスト) のリストを返す。関数 *grab_rule* は *rem_rule* で得られた情報を使用し、ある構文に残りの文字列のリストを追加して、新たな構文として成り立たないかを探索する。成り立つ場合には、元の構文、前方部分だけの構文、残りの文字列のリスト、追加される構文、追加されたあとの構文を返す。これらの情報を使用することで、図 6 の関数 *grab* を作ることができる。

4. まとめと今後の課題

本論文では、漸進的構文解析のように一部だけ変更されたプログラムに対して構文解析を行う手法を提案した。コンパイラの構文解析器を使用す

$$
\begin{array}{rcl}
grab & : & term * term\ list \to term * list\ option \\
grab[c^s, Ms] & = & (c^s, []) \\
grab[M_1\ Op^s M_2, Ms] & = & let\ Ms = if\ s >= p\ then\ []\ else\ Ms\ in \\
 & & let\ (M_1', \_) = grab[M_1, []]\ in \\
 & & let\ (M_2', Ms') = grab[M_2, Ms]\ in \\
 & & (M_1'\ Op^s M_2', Ms') \\
grab[if^s\ M_1\ then\ M_2, Ms] & = & let\ Ms = if\ s >= p\ then\ []\ else\ Ms\ in \\
 & & let\ (M_1', \_) = grab[M_1, []]\ in \\
 & & let\ (M_2', Ms') = grab[M_2, Ms]\ in \\
 & & (match\ Ms'\ with \\
 & & \quad | fst :: rest \to (if^s\ M_1'\ then\ M_2'\ else\ fst, rest) \\
 & & \quad | [] \to (if^s\ M_1'\ then\ M_2', [])) \\
grab[if^s\ M_1\ then\ M_2\ else\ M_3, Ms] & = & let\ Ms = if\ s >= p\ then\ []\ else\ Ms\ in \\
 & & let\ (M_1', \_) = grab[M_1, []]\ in \\
 & & let\ (M_2', Ms') = grab[M_2, (M_3 :: Ms)]\ in \\
 & & (match\ Ms'\ with \\
 & & \quad | fst :: rest \to (if^s\ M_1'\ then\ M_2'\ else\ fst, rest) \\
 & & \quad | [] \to (if^s\ M_1'\ then\ M_2', []))
\end{array}
$$

図 6 強奪の更新をする関数 $grab$

ることで、特殊な構文解析器を実装する必要がないという利点が存在する。組み替えに必要な結合の規則や強奪の規則に関してもコンパイラの構文解析器を用いることで求めている。

本論文での構文の定義に、let 文、パターンマッチ (match 文) 等を追加し、実装を行ったところ、同様に正しく更新を行うことができた。ただし、より複雑な構文に拡張した際にも、組み替えと強奪のみで更新が行えるかは未知であるため、今後構文を拡張していきたい。また、本論文では、一箇所の変更のみについて考えた。実際には複数の変更 (たとえば片方で "(" を、もう片方で ")" を書くなど) が容易に起こりうる。その際にそれらを含む全体の式に対して再構文解析を行うのは明らかに無駄であり、なるべく元の構造を保ちたいという本研究の趣旨と異なる。よってそのような変更について、よりそれまでの構造を保ったままで構文解析する手法が必要となる。

参考文献

[1] F. Bour, T. Refis and S. Castellan, "Merlin, an assistant for editing OCaml code," The OCaml Users and Developers Workshop 2013.

[2] C. Ghezzi and D. Mandrioli, "Incremental Parsing," ACM Transactions on Programming Languages and Systems (TOPLAS) Volume 1 Issue 1, (1979).

[3] J. Roehrich, "Syntax-Error Recovery in LR-parsers," Programmiersprachen Volume 1 of the series Informatik-Fachberichte pp 175–184 (1976).

[4] https://github.com/the-lambda-church/merlin

$$
\begin{array}{rcl}
rem\_loop & : & string\ list * string\ list * term \\
 & & \quad \rightarrow (term * term * (string\ list))\ list \\
rem\_loop[[], prog, orig] & = & [] \\
rem\_loop[fst :: rest, prog, orig] & = & let\ prog' = prog@[fst]\ in \\
 & & try((orig, \texttt{parse}\ prog', rest) :: (rem\_loop[rest, prog', orig])) \\
 & & with\ Syntax\_error \rightarrow rem\_loop[rest, prog', orig] \\
 & & \\
rem\_rule & : & (string\ list)\ list \rightarrow (term * term * (string\ list))\ list \\
rem\_rule[[]] & = & [] \\
rem\_rule[fst :: rest] & = & (rem\_loop[fst, [], \texttt{parse}\ fst]) :: rem\_rule[rest] \\
 & & \\
grab\_loop & : & (string\ list) * (term * term * (string\ list))\ list * term \\
 & & \quad \rightarrow (term * term * (string\ list) * term * term)\ list \\
grab\_loop[strs, [], orig] & = & [] \\
grab\_loop[strs, (M_{orig}, M_{rem}, strs_{rem}) :: rest, orig] & = & try\ ((\texttt{parse}\ (strs@strs_{rem})); \\
 & & \quad (M_{orig}, M_{rem}, strs_{rem}, orig, \texttt{parse}\ strs) \\
 & & \quad :: grab\_loop[strs, rest, orig]) \\
 & & with\ Syntax\_error \rightarrow grab\_loop[strs, rest, orig] \\
 & & \\
grab\_rule & : & (string\ list)\ list * (term * term * (string\ list))\ list \\
 & & \quad \rightarrow ((term * term * (string\ list) * term * term)\ list)\ list \\
grab\_rule[[], rems] & = & [] \\
grab\_rule[fst :: rest, rems] & = & (grab\_loop[fst, rems, \texttt{parse}\ fst]) :: (grab\_rule[rest, rems]) \\
 & & \\
grab\_main & : & (string\ list)\ list \rightarrow \\
 & & \quad ((term * term * (string\ list) * term)\ list)\ list \\
grab\_main[strs] & = & let\ rem\_list = rem\_rule\ strs\ in \\
 & & let\ grab\_list = grab\_rule\ strs\ rem\_list\ in \\
 & & grab\_list
\end{array}
$$

図 7 強奪に必要な情報を集める関数 $grab\_main$