パーサの段階的開発のための対話による文法編集に向けて

奥田 勝己　千葉 滋

本論文では，パーサを生成することを目的とする文法を，lake 記号を用いて段階的に記述する手法を提案する．プログラムの構文解析を行うパーサはさまざまなソフトウェアツールに応用されるが，パーサを生成することを目的とした文法の記述は容易ではない．この一因は文法が完全に記述し終わるまで動作確認可能なパーサを得ることが困難なことである．そこで，提案手法では lake 記号をソフトウェアにおけるテストスタブのように用いることで，編集中の文法から常に実行可能なパーサを得られるようにする．この結果，動作確認のためのテキストハイライトや文法規則追加のための操作を例示により実現することができる．プロトタイプを用いたケーススタディでは，構文拡張のためのプログラム変換器の開発に提案手法を応用できることを確認した．

1　はじめに

パーサは，コンパイラやインタプリタだけでなくソフトウェア開発の様々な用途に用いられる．例えば，構文拡張のためのプログラム変換器やソフトウェアメトリクスの計測ツール，リファクタリングツールなどもパーサをその一部とする．これらの用途では，必ずしも対象言語の完全な文法を必要としない．例えば，Java の if 文を抽出するだけであれば，Java の完全な構文解析を必要とはせず，if 文のみを抽出できるような構文解析ができればよい．このように興味があるプログラム構文のみを抽出するパーサを生成可能な文法は island 文法 [4] と呼ばれる．開発者は island 文法を記述することで，対象言語の完全な文法と比べて少ない文法規則で必要なパーサを得ることができる．Island 文法は，PEG（Parsing Expression Grammar）などの文法記述言語で記述することができるが，興味があるプログラム構文以外に対応する規則を適切に作成するためには労力が必要である．

Katsumi Okuda, 東京大学 / 三菱電機株式会社, The University of Tokyo / Mitsubishi Electric Corporation.
Shigeru Chiba, 東京大学, The University of Tokyo.

我々の lake 記号 [5] はこの労力を軽減することを目的とした PEG の新しい文法記号である．Lake 記号は island 文法に適した最短一致を実現するためのワイルドカードとしての機能をユーザに提供する．ユーザは文法中で省略したい位置に lake 記号を記述することで，比較的容易に island 文法を記述することができる．

しかし，lake 記号を導入しても island 文法のデバッグは難しいという問題が残る．Island 文法が扱う言語は，元の完全な言語とは一致しない．このため，開発者は元の言語の仕様書と比べながら island 文法中の誤りを見つけることはできない．したがって，開発者は試行錯誤しながら island 文法をデバッグする必要があった．

本論文では，正しい island 文法の記述を容易化することを目的とし，island 文法を対話的に編集する手法を提案する．本手法では，lake 記号をソフトウェアテストにおけるスタブのように使用することで，対話的かつ段階的に文法を更新するための操作を提供する．これによって小さな動作確認済みの文法規則を組み合わせて上位の文法規則を作ることができる．また，常に動作するパーサが得られることを利用し，例題プログラムを用いたテキストハイライトによる動作確認手段や操作適用時における候補の提示機能

```
{
  doit();
  if (fooIsNeeded()) {
    foo();
    if (barIsNeeded()) {
      for (i = 0; i < 3; i++) {
        bar();
        if (bazIsNeeded()) {
          for (j = 0; j < 3; j++) {
            baz();
          }
        }
      }
    }
  }
}
```

図 1　入れ子になった if 文

を提供する．

2　Island 文法記述時の課題

本節では，本研究で扱う lake 記号を用いた island 文法をその応用とともに説明し，記述時における課題を説明する．

2.1　Island 文法の応用

本論文で対象とする lake 記号を用いた island 文法は，一部のプログラム構文のみに興味があるような応用に適している．例えば，lake 記号を用いた island 文法は構文拡張のためのプログラム変換器のパーサ開発に便利である．以下では，特定のドメインにけるプログラムの記述容易性や可読性を向上するための構文拡張を想定し，プログラム変換器向けの island 文法の応用を説明する．

例として，図 1 のような入れ子になった if 文が多用される場合を考えよう．このような場合，拡張構文である `onlyWhen` 構文を導入し，図 2 のように書き直して入れ子の数を減らしたいとする．一見すると `onlyWhen` は，普通の関数呼び出しに見える．しかし，実際には与えられた式が真の場合のみ自身を囲うブロックの末尾までの後続文を実行することを意味している．

この `onlyWhen` 構文は，`onlyWhen` を if 文に変換するプログラム変換器で実現することができる．例えば， このプログラム変換器は図 2 のプログラムを図 1 のプログラムに変換する．この変換は sed による置換のようにパターンの書き換えで実現することができる．プログラム変換器はプログラム中に下記のようなパターンを検出した時,

```
{
   <<A>>
   onlyWhen(<<B>>);
   <<C>>
}
```

以下のように書き換える．

```
{
  <<A>>
  if (<<B>>) {
    <<C>>
  }
}
```

ここで`<<A>>`, `<<B>>`, `<<C>>`はプログラムの省略を表す．プログラム変換器はこのパターンが見つからなくなるまで書き換えを繰り返すことで，プログラム中のすべての `onlyWhen` 構文を変換する．なお，`onlyWhen` 構文は図 2 の 3 行目の `onlyWhen` キーワードで始まっているのではなく，１行目の`{`が開始である点に注意が必要である．プログラム変換器は `onlyWhen` 文自身が属するブロックの末尾を検出する必要があるため，`onlyWhen` 構文の開始位置は `onlyWhen` 文を含むブロックの開始位置の`{`と一致する．onlyWhen 構文の書き換えでは，onlyWhen 構文のみを抽出できればよく，その前後や`<<A>>`，`<<B>>`，`<<C>>`の部分の詳細な構文木は不要である．Lake 記号を用いた island 文法はこのようなパーサの生成に用いることができる．

```
1  {
2    doit();
3    onlyWhen(fooIsNeeded());
4    foo();
5    onlyWhen(barIsNeeded());
6    for (i = 0; i < 3; i++) {
7      bar();
8      onlyWhen(bazIsNeeded());
9      for (j = 0; j < 3; j++) {
10       baz();
11     }
12   }
13 }
```

図 2　`onlyWhen` 構文の例

2.2 Lake 記号を用いた island 文法

Island 構文解析はプログラム中で興味のある構文だけを詳細に解析し，それ以外の部分を読み飛ばす構文解析の技法である．興味がある構文を island，それ以外の部分を water と呼ぶ．先述の onlyWhen の変換において，{ <<A>> onlywhen(<<B>>); <<C>> } という onlyWhen 構文のパターンを見つけて，その構文木を得ることは island 構文解析である．{, onlyWhen, (,), }から構成される onlyWhen 構文は island であり，その前後や内側の<<A>>，<<B>>，<<C>>は water として読み飛ばして構わない．Island パーサはしばしば生成器を用いて開発され，生成器の入力となる文法は island 文法 [4] と呼ばれる．Island 文法は，island 部分に対応する文法規則と，それ以外の部分を water として読み飛ばすための文法規則から構成される．water に対応する文法規則は，ワイルドカードのように入力文字を読み飛ばすことのみできれば良い．このため water 用の文法規則群は，island のための文法規則群より少なくて済む．したがって，island 文法の規則数は，対象とする言語の完全な文法と比べて少ない傾向にある．例えば，Python の if 文のみを抽出するための island 文法の規則数は，Python の完全な文法の規則数に比べて 82% [5] 少ない．プログラムの一部の構文にのみ興味があるソフトウェアツールの開発では，記述量が少ない island 文法は有用である．しかし，water に対応する規則を書くことは容易でないという問題があった．

Lake 記号 [5] は，water のための規則の記述を簡易化することを目的とした PEG（Parsing Expression Grammar）[1] の構文拡張である．文法の作成者は lake 記号を用いることで island の一部を water として扱う文法を容易に記述することができる．興味があってきちんと解析したい構文であっても，その途中部分は読み飛ばしたいときがある．例えば onlyWhen 構文では，途中に現れる<<A>>，<<B>>，<<C>>の詳細は興味が無いため water として読み飛ばしたい．そのような island の内側の water を lake と呼び，lake 記号は主に lake の記述を行うための文法記号である．Lake 記号は非終端記号のように文法の記述に使用することができる．非終端記号との区別がつくように lake 記号は<>で囲まれる．例えば，<lake>は lake 記号である．また，lake 記号は非終端記号と同じように自身を左辺とする規則を付加的に持つこともできる．付加的な規則を持たない lake 記号は，lake の内側や周りの island の一部を除く任意の文字にマッチする特別なワイルドカード記号として機能する．ユーザは lake 記号を PEG の繰り返し演算子である* や+ 演算子と組み合わせて使用することで，island 間の任意の文字列を**最短一致**でマッチさせることができる．Lake 記号は，Perl や Ruby の正規表現実装における 1 つ以上の任意の文字列の繰り返しの最短一致である .+? と似た働きをし，island の一部または全体に到達するまでマッチし続ける．例として，セミコロン（;）で終了する文を island とする場合を考えよう．この文に対応する PEG の文法規則は下記のとおりである．

```
statement <- <e>+ ';'
```

上記の規則において<>で囲まれた文法記号<e> が lake 記号である．この規則は，<e>+ ';' が入力文字列にマッチした時に非終端記号 statement が認識されることを意味する．ここで<e>+ ';' は，';' で終わる lake 記号 <e>の 1 つ以上の繰り返しの最短一致であり，例えば x = 1; のような文にマッチする．<e>は正規表現の単純なワイルドカード記号であるドット（.）とは異なる．上記の規則において，<e>+を単純なワイルドカード記号を用いて.+と置き換えることはできない．この場合 . が ';' にもマッチしてしまうためである．例えば，入力文字列が x = 1; y = 1; の場合，正規表現の.+; は最長一致のため入力全体，すなわち 2 つの文とマッチしてしまう．この時 . は x = 1 直後の';' にマッチしている．一方，<e>+ ';' は 1 つの文にのみマッチする．このため，<e>+は.+とは異なる．また， <e>+は正規表現における最短一致である .+? とも異なる．これを示すため，前述の文法規則を以下のように変更しよう．

```
statement <- <e>+ ';' / block
```

ここで/ は PEG の優先度付き選択演算子である．また，非終端記号 block は，例えば{ k = 1; }のように波括弧 {} で囲まれた文字列を認識するものとする．修正後の statement はセミコロンで終わる最

短の文字列に加えて `block` も認識する．上記の文法において `block` は，island である．このため，`<e>+` は `block` が認識する文字列にはマッチしない．一方，`<e>+` を `.+?`に置き換えた場合，`.+?`は `block` が認識する文字列にもマッチする．例えば，`.+?`は`{ k = 1;` を認識するが，本来これは `block` の一部として認識されるべきである．このため，`block` が意図通りに認識されなくなる．

パーサは，lake 記号が island やその一部とマッチすることを防ぐため，マッチすべきでないテキストパターンを知る必要がある．これらのパターンは，*alternative* 記号と呼ばれる終端記号または非終端記号の集合で与えられる．例えば，上述の例では `';'` や `block` が *alternative* 記号である．lake 記号は，この *alternative* 記号群を自動的に計算することができるため，island 文法向けの最短一致用ワイルドカード記号として使用することができる．

lake 記号は，暗黙的に *alternative* 記号群とマッチしないが，ユーザが lake 記号として認識したいパターンを明示的に指定するための手段も提供する．ユーザは，lake 記号を左辺とする規則を作り，右辺にその lake 記号がマッチすべきパターンを記述できる．付加的な規則を持つ lake 記号は，その右辺のパターンのマッチに失敗した場合に最短一致のワイルドカード記号として機能する．ユーザはこの仕組みを用いて lake に入れ子になった island も記述することができる．

2.3 記述時の課題

onlyWhen 構文の変換には onlyWhen 構文を抽出するための構文解析が必要であるが，lake 記号を用いた island 文法を使うことで少ない文法規則でそのためのパーサを得ることができる．図 3 は `onlyWhen` 構文を island として抽出するための island 文法である．図 3 では，`<sea>`と`<other>`は lake 記号である．先述の`<<A>>`と`<<B>>`による省略箇所は`<sea>*`が対応し，`<<C>>`による省略箇所は`<other>*`が対応する．

1 行目ではプログラム全体を lake 記号`<sea>`の繰り返しとして定義している．2 行目の規則は，lake 記号`<sea>`が `island` と `water` を認識することを示す．`<sea>`は，入力文字列が `island` と `water` の両方にマッチしなかった場合，最短一致のワイルドカードとして island またはその一部以外の任意の文字とマッチする．4 行目で`<sea>`は`{`の後で使用され，先述の`<<A>>`に対応する．ここで，`<sea>`は `island` または `water` に入力がマッチしなかった場合，`'onlyWhen'` が現れるまで最短一致で任意の文字にマッチする．また，`<sea>`は，`'onlyWhen'` 以外に island の一部である`}` や `)` にもマッチしない．同様に 4 行目の`<other>` も`<sea>`と概ね同じであるが，`'onlyWhen'` を除外しない点が`<sea>`とは異なる．`<other>`の規則は，5 行目で`<sea>`と同様にして与えられる．このように lake 記号を用いることで少ない規則数で構文拡張のための文法を得ることができる．

```
1  program <- <sea>*
2  <sea> <- island / water
3  island <-
4    '{' <sea>* 'onlyWhen' group ';' <other>* '}'
5  <other> <- island / water
6  water <- group / block
7  group <- '(' <sea>* ')'
8  block <- '{' <sea>* '}'
```

図 3　Lake 記号を用いた island 文法

しかし，正しい文法を記述することはそれほど容易ではない．もし 6 行目の `water` の規則の右辺に `block` を記載し忘れた場合，この island 文法から得られるパーサは期待通りに動作しない．`block` を明示的に `water` に指定しない場合，ブロックの一部である閉じ波括弧を `island` の一部として解釈してしまう．例えば，下記の `onlyWhen` 構文を解析するとき

```
{ onlyWhen(x); { doit(); } }
```

パーサは，意図とは異なり 1 文字手前で以下を `onlyWhen` 構文として抽出してしまう．

```
{ onlyWhen(x); { doit(); }
```

このプログラムを正しく解析するためには，`{ doit(); }` が `water` として認識されるようになっている必要がある．このためには `block` が `water` の規則の右辺に記述されている必要がある．

デバッグの対象が island 文法ではなく，完全な文法である場合，ユーザは言語の仕様書と見比べながら

個々の規則を確認することができる．しかし，island 文法の場合，一部を省略しているのでその方法は有効でない．正しい文法を得るには，実行してみて結果を確認する作業を繰り返すしかない．したがって，これを効率化するツール支援が必要である．

3 対話による文法の作成

3.1 段階的な文法の詳細化

本節では，小さな island 文法から開始し段階的に詳細化しながら目的とする island 文法を得ることができる文法編集ツール *PEGSEED* を提案する．*PEGSEED* は文法の詳細化後も常に動作するパーサが得られるようにすることで，ユーザが詳細化するたびに文法の正しさを確認・デバッグすることができるようにする．

常に実行可能パーサが得られるように文法を更新しながら，任意の island 文法を得られるようにすることは容易ではない．そこで，*PEGSEED* では，編集可能な PEG のメタ文法を定め，そのメタ文法に基づく文法規則を生成可能な UI 操作を提供する．ユーザは，*PEGSEED* の提供する UI 操作で island 文法に規則を追加していくことで，メタ文法から生成可能な任意の island 文法を作成することができる．

UI 操作で追加する規則には lake 記号を使うこともできる．*PEGSEED* における lake 記号の利用は，ソフトウェア開発におけるテストスタブの利用に似ている．ソフトウェアの開発ではテスト済みのモジュールを結合することで，より大きなモジュールの試験を進める．一部のモジュールが未実装の場合，代わりに簡単な実装のプログラムをスタブとして用いることで，上位モジュールの試験を進めることができる．文法の作成にも同じアプローチを適用することが提案手法のアイデアである．テスト済みの文法規則を組み合わせてより大きな文法規則を作成する．一部の文法規則が未作成の場合，その規則に対応する入力文字列を lake 記号で読み飛ばす．すなわち，lake 記号をテストスタブのように使用する．

3.2 編集対象文法のメタ文法

```
program <- <lake>
<lake> <- string
string <- r’"([^"]|\")*?"\s*’
```

図 4 初期文法

```
1  grammar = rul
2  rule = token_rule | choice_rule |
       sequence_rule
3  token_rule = nonterminal ’<-’ regex
4  sequence_rule = nonterminal ’<-’ (nonterminal
       ’*’?| lake ’*’)+
5  choice_rule = (nonterminal / lake) ’<-’
       nonterminal (’/’ nonterminal)*
```

図 5 生成される文法の EBNF によるメタ文法

提案手法では，すべてを water として扱う island 文法を初期文法とし，これに対する規則の追加と動作確認を繰り返すことで，目的の文法を作成する．図 4 が初期文法である．開始記号は `program` であり，プログラム全体を文字列リテラルまたはワイルドカード記号でマッチさせる．文字列リテラルを初期文法で認識させるのは，文字列中には任意に文字列が含まれる可能性があるため，その一部を誤って別の規則で認識することを防ぐためである．ユーザは，初期文法から開始し，*PEGSEED* が提供する操作を適用することで段階的に新しい規則を追加し，文法を更新する．文法の更新は，主に（1）新しい字句を認識するための規則の追加,（2）既存規則の左辺の非終端記号や lake 記号を組み合わせた新しい規則の追加によって行われる．新しい規則を追加するとき，その一部を省略することができる．省略したい部分に lake 記号をワイルドカードとして使うことで，一部を詳細化せずに済む．

PEGSEED が提供する操作は，図 5 の EBNF によるメタ文法で表現される PEG を生成することができる．1 行目のメタ文法規則が示すように生成される PEG は 1 つ以上の文法規則（`rule`）から構成される．また，*PEGSEED* が生成する各文法規則 (`rule`) は，`token_rule`，`sequence_rule` または `choice_rule` のいずれかである．*PEGSEED* では，この 3 種類の規則それぞれを生成するための操作を提供している．`token_rule` が対応する規則は，字句を認識するための規則である．また・`sequence_rule`

が対応する規則は，定義済みの非終端記号や lake 記号を組み合わせて作られる規則である．新しく作成した規則の左辺の非終端記号は，必ず `choice_rule` と対応する規則の右辺に追加される．

3.3　ツールによる文法の編集支援

PEGSEED では，編集中常に動作するパーサが得られることを利用し，文法の動作確認や規則追加のためのツール支援が可能である．編集中の文法からパーサが得られるため，*PEGSEED* は対象言語の例題プログラムをリアルタイムで構文解析した結果から，各規則が例題プログラムのどこにマッチしたかを常に取得できる．*PEGSEED* は，例題プログラム中においてユーザが選択した規則とマッチする部分をハイライト表示する．また，例題プログラムのテキスト領域をユーザが選択すると，*PEGSEED* は，その領域と対応する文法記号の一覧を構文木から得る．*PEGSEED* では，これを用いて例題ベースの GUI 操作によって新しい規則を追加することができる．

4　ケーススタディ

文法編集システム *PEGSEED* のプロトタイプを用いて構文拡張のための文法を作成し，提案手法の有用性を確認した．以下では 2 節で紹介した `onlyWhen` 構文の変換に用いる island 文法を作成したケースを紹介する．この `onlyWhen` 構文の変換器は C/C++や Java など C ライクな文法を持つ言語に適用することができる．

図 2 の `onlyWhen` 構文を構成する字句は，`onlyWhen`, `{`, `}`, `(`, `)` である．まずは，これらを正しく認識できるようにした後，これらを段階的に組み合わせることで最終的に `onlyWhen` 構文を認識できるようにする．まず，初期文法に字句である `onlyWhen` を認識できるように規則を追加すると下記の文法となる．

```
1  program <- <lake>*
2  <lake> <- onlyWhen / string
3  string <- r’"([^"]|\")*?"\s*’
4  onlyWhen <- r’(?<!\w)onlyWhen(?!\w)\s*’
```

4 行目が初期文法に対して新たに追加された規則である．4 行目の右辺は字句である `onlyWhen` にのみマッチする正規表現である．左辺の非終端記号である `onlyWhen` は，2 行目の規則の右辺にも加えられる．ユーザは *PEGSEED* が提供する字句追加操作を用いて，この文法の更新を行うことができる．ユーザは，*PEGSEED* 上のテキストエディタで図 2 のプログラムを開き onlyWhen キーワードの 1 つを選択してから字句追加の操作を選択する．すると，*PEGSEED* は図 6 に示すダイアログを表示する．ユーザは，ダイアログ上で新しく追加する規則の左辺の非終端記号と右辺の正規表現を入力することができる．*PEGSEED* はユーザが選択したテキストからこれらの初期値を作成することができる．また，図 6 のダイアログでは新しく作成した非終端記号をどの規則に加えるかを選択できる．*PEGSEED* はユーザが選択したテキストの位置から追加の対象となる規則の候補を自動的に選択する．`onlyWhen` の字句を追加する場合，ユーザはダイアログの OK をクリックするのみで期待通りの規則を追加することができる．ユーザは，この規則を追加した直後に，この規則が正しいことを *PEGSEED* のテキストエディタ上で確認することができる．ユーザが非終端記号の一覧から `onlyWhen` を選択すると，テキストエディタ上で対応するテキスト領域がハイライトされる．この様子を図 7 に示す．図 7 では期待通りに 3 箇所がハイライト表示されている．もし期待通りにハイライトされていない場合，ユーザは前回の操作を取り消し，再度規則を追加しなおすこともできる．

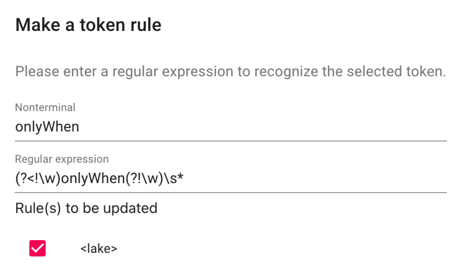

図 6　字句規則を追加するためのダイアログ

`onlyWhen` と同様にしてその他の字句も GUI 操作のみで追加することができる．すべての字句の規則を追加した後の文法は下記の通りである．

```
program <- <lake>*
```

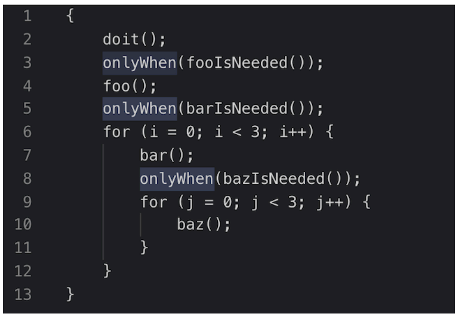

図 7　ハイライトによる確認

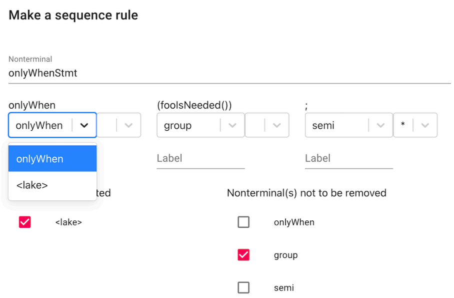

図 8　連結操作のためのダイアログ

```
2   <lake> <- semi / rpar / lpar / rcub / lcub /
        onlyWhen / string
3   string <- r'"([^"]|\")*?"\s*'
4   onlyWhen <- r'(?<!\w)onlyWhen(?!\w)\s*'
5   lcub <- r'\{\s*'
6   rcub <- r'\}\s*'
7   lpar <- r'\(\s*'
8   rpar <- r'\)\s*'
9   semi <- r';\s*'
```

5 行目から 9 行目の規則が新たに追加された規則である．

次にこれらの字句を組み合わせて，より大きな island の規則を追加する．追加する規則は図 5 における `sequence_rule` に対応する規則である．まず，ここでは onlyWhen 構文の一部である条件部を新しい island として追加する．条件部は () で囲まれた部分である．onlyWhen 構文の変換では条件部の式の構文木は不要である．このため，() で囲まれた内側は water として扱うことにする．この条件部のための規則は既存の非終端記号と lake 記号を連結することで作ることができる．GUI による連結操作で条件部に対応する規則を加えた文法は下記の通りである．

```
1   program <- <lake>*
2   <lake> <- group/ semi / rcub / lcub / onlyWhen
        / string
3   string <- r'"([^"]|\")*?"\s*'
4   onlyWhen <- r'(?<!\w)onlyWhen(?!\w)\s*'
5   lcub <- r'\{\s*'
6   rcub <- r'\}\s*'
7   lpar <- r'\(\s*'
8   rpar <- r'\)\s*'
9   semi <- r';\s*'
10  group <- lpar <lake>* rpar
```

ここで 10 行目が新たに加えた規則である．ユーザは `group` の規則を追加した後，期待通りの箇所にマッチすることを例題プログラムのハイライトで直ちに確認することができる．

同様にして動作確認済みの非終端記号 `onlyWhen`，`group`，`semi` を連結して onlyWhen 文を island として追加する．ユーザが onlyWhen 文を例題テキストで選択し，連結操作を選択すると，*PEGSEED* は図 8 のダイアログを表示する．図 8 のダイアログでは，新しく追加する規則の左辺の非終端記号と右辺のパターンを記述できる．右辺のパターンは複数のリストボックスから適切な文法記号を選択することで作成することができる．*PEGSEED* は既に onlyWhen 文の構成要素に対応する非終端記号を認識できるため，この情報を用いて適切な候補をユーザに提示する．図 8 で新たに追加した規則は下記の通りである．

```
onlyWhenStmt <- onlyWhen group semi
```

同様にして連結操作を繰り返して適用することで最終的に図 9 の文法を得ることができる．なお，13 行目と 14 行目は連結操作で `onlyWhen` 構文に対応する規則を作成した時に追加されたものである．ひとつ前の操作では 12 行目の `block` の規則を連結操作で追加している．仮にこの 12 行目の追加操作が漏れていた場合，`onlyWhen` 構文に対応する `onlyWhenConstruct` の規則を追加した直後に例題プログラムをハイライトすると図 10 のようになり，ユーザは直ちに誤りに気付くことができる．図 10 では，本来 13 行目までがハイライトされるべきところが 12 行目までしかされていない．ハイライト表示から 6 行目から始まる onlyWhen 構文が 11 行目の{で終了していることが分かる．11 行目の}は実際には for 分のブロックの一

```
program <- <lake>*
<lake> <- onlyWhenConstruct / block / group /
     water
string <- r'"([^"]|\")*?"\s*'
onlyWhen <- r'(?<!\w)onlyWhen(?!\w)\s*'
lcub <- r'\{\s*'
rcub <- r'\}\s*'
lpar <- r'\(\s*'
rpar <- r'\)\s*'
semi <- r';\s*'
group <- lpar <lake>* rpar
onlyWhenStmt <- onlyWhen group semi
block <- lcub <lake>* rcub
<other> <- onlyWhenConstruct / block / group /
      water
onlyWhenConstruct <- lcub <lake>* onlyWhenStmt
      <other>* rcub
```

図 9　対話操作で作成した onlyWhen 構文のための island 文法

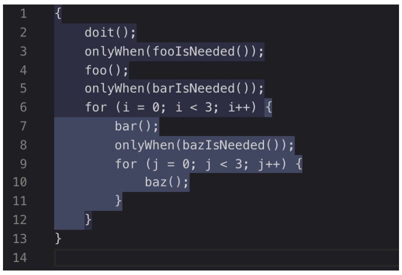

図 10　`block` の規則が漏れている時の onlyWhen 構文のハイライト

部である．このことからユーザは，`block` を認識する規則が抜けていることに気付くことができる．

5　関連研究

対話的な文法編集ツールとして Parsify [3] や Parsimony [2] が存在するが，island 文法向きではない．Parsify や Parsimony では，GUI 操作で文法を教える．たとえば，エディタ上で式を選択し，式であることをラベル付けといったことを繰り返すことで，文法を作成していく．しかし，途中の文法は曖昧であり，曖昧性を取り除くには完全な文法を教える必要があった．それに対し，本論文で提案した *PEGSEED* では，常に動作するパーサを得ることができる文法の更新を繰り返すことで目的とする文法を記述する．このため，特定の構文のみを抽出することを目的とするような場合には，完全な文法を記述することなく必要なパーサを得ることができる．

6　まとめ

本論文では，段階的に動作確認をしながらパーサを開発するための対話による文法編集手法とその実装である *PEGSEED* を提案した．本手法では，対話的に文法に規則を追加するための操作を提供する．操作で追加する規則の一部は lake 記号で省略することができる．これは lake 記号をテストスタブのように用いることに相当し，編集中の文法からも常に動作するパーサを得ることができる．*PEGSEED* は常に動作するパーサを用いることで，例題プログラムを用いたハイライト機能や操作適用時における候補の提示機能を提供する．*PEGSEED* を用いて onlyWhen 構文の抽出を行い，その有用性を確認した．今後，その他の構文拡張についても有用性を確認する予定である．

参 考 文 献

[1] Ford, B.: Parsing Expression Grammars: A Recognition-based Syntactic Foundation, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, New York, NY, USA, ACM, 2004, pp. 111–122.

[2] Leung, A. and Lerner, S.: Parsimony: An IDE for Example-guided Synthesis of Lexers and Parsers, *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, Piscataway, NJ, USA, IEEE Press, 2017, pp. 815–825.

[3] Leung, A., Sarracino, J., and Lerner, S.: Interactive Parser Synthesis by Example, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, New York, NY, USA, ACM, 2015, pp. 565–574.

[4] Moonen, L.: Generating Robust Parsers Using Island Grammars, *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, Washington, DC, USA, IEEE Computer Society, 2001, pp. 13–22.

[5] Okuda, K. and Chiba, S.: Lake symbols for island parsing, *Art Sci. Eng. Program.*, Vol. 5, No. 2(2021), pp. 11.