修士論文

OPGに基づくアドホックな大規模文字列データ解析のための並列データ処理系

A Parallel Data Processing System for Large Text Data based on OPG

令和2年1月28日提出

指導教員　　田浦 健次朗 教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

48-186408 劉 啓恒

概 要

高性能なパーサジェネレータが多く実装されているが，そのほとんどが逐次に動くパーサを生成するものであり，プロセッサの計算資源を十分に利用できず，大規模なデータを効率よく解析できない．我々は，並列字句解析のためのヒューリスティックを取り入れ，OPG の並列構文解析アルゴリズムに基づいた並列処理系を提案する．実際の JSON データを使用した実験では，複数のプロセッサを使用した場合，良い台数効果が得られている．

目 次

図 目 次

表 目 次

第1章　序論

1.1　大規模テキストデータ処理に関する研究背景

我々が日々扱っているデータは，人間の可読性の観点から大きく 2 種類に分けられる．バイナリデータとテキストデータである．いずれのデータに関しても，コンピュータがそれを処理し，情報を取り出す必要がある．バイナリデータは，人間が直接読むことが比較的少なく，コンピュータが処理しやすいように設計されていると言えるだろう．一方，テキストデータは，どちらかというと人間が読むために設計されたものである．ただし，テキストデータに関しては，可読性と同時に機械も処理できなければならない．その一例として，多くのアプリケーションがクライアントとサーバとの通信時に使用する JSON というフォーマットがある．JSON は，人間が直接読み書きしやすく，かつコンピュータが処理しやすいように設計されている．もう一つの例として，サーバのログが挙げられる．サーバのログは人間が直接書かず，基本的には人間が読むためのものである．しかし，ログの量が大きい場合など，ときにはコンピュータが自動的にログを分析して適切な処理を行う場合もある．この場合も，適切な処理系を作る必要がある．

広く使われる特定のフォーマットに対しては，パーサを手書きで作り，しっかり最適化をかけるのが一般的である．しかし，数回しか使われず，かつ一般的でない形式に対しては，わざわざ手書きでパーサを作る利点は少ない．この場合，パーサジェネレータ（第 1.2.2 節）と呼ばれるツールを使い，パーサのプログラムを自動生成する．

一般的なテキストファイルの処理系は，入力データを字句解析と構文解析と呼ばれる処理をして，情報を抽出する．上述することは，一般的なテキストファイルの処理系に当てはまるが，特に大規模なテキストデータを処理する際には，処理系の性能が重要となる．例えば，Wikidata のダンプファイルは数百ギガバイトほどの JSON ファイルであり，通常の処理系では処理に時間がかかりすぎる．処理系の高速化は，アルゴリズムの改良はもとより，データをできるだけキャッシュに格納するなど，CPU の計算資源をより有効に利用する方針が取れる．そして，ハードウェア資源をより有効に活用するには，現在のハードウェアの進化のトレンドを踏まえると，並列化するのが大きなポイントである．すなわち，入力データを複数の部分に分割して，それぞれ並列に解析するというやり方である．

特定のフォーマットに対して，そのフォーマットの特徴を我々人間が見て，並列化のための処理を書くのは困難なことではない．実際，広く使われるフォーマットは，すでに並列パーサが実装さ

れている [13, 12, 14, 23]．しかし，第 1.3 節で説明するように，アドホックに文法を定義して，並列パーサを自動的に生成するのは難しいことである．さらに，解析自体も字句解析（第 1.2.4 節）と構文解析（第 1.2.5 節）の 2 つの段階に分けられ，例えば構文解析のフェーズのみ並列化しても，字句解析が逐次のため，全体のボトルネックとなる．

字句解析は，処理が単純なため無視されがちであるが，これが入力文字を一つずつ処理する唯一のフェーズであり，全体の処理時間の多くを占める [1, 4]．ただ，第 1.3 節で説明するように，字句解析の並列化は原理的には可能である．

構文解析の並列化は難しく，形式文法の特徴に大きく依存する．通常，ある形式文法のクラスは，その構文解析アルゴリズムとともに提案されることが多い．そして，その構文解析アルゴリズムにヒューリスティックをいれ，高速化等が行われる．入力を複数のチャンクに分割してもなお，各チャンクを独立に構文解析できる性質を **local parsability** と呼ぶことが多い．Local parsability をもつ形式文法のクラスが研究されているが [21, 16, 5]，多くは理論の段階に止まっており，現在実装できるのは，第 1.4 節で説明する OPG のみである．

1.2 基本的な概念

1.2.1 字句解析と構文解析

本節では，データを解析する際に行う字句解析と構文解析の概念について説明する．それぞれの具体的なやり方は，第 1.2.4 節と第 1.2.5 節を参照されたい．

まず，生の入力データは文字列である．文字列を読み取り，それを適切な場所で区切り，**トークン**と呼ばれる構文解析の最小単位を得る過程を**字句解析**と呼ぶ．字句解析を行うツールを一般的には**字句解析器**，**レキサ**，あるいは**スキャナ**と呼ぶ．厳密には，区切られたそれぞれの部分文字列は **lexeme（語彙素）**と呼ばれる．そして，語彙素と，後段の構文解析におけるその語彙素のもつ意味とを合わせて**トークン（字句）**と呼ぶ．ただし，本文では混乱の恐れがない場合においては，語彙素という言葉はあまり用いず，トークンという言葉を使用する．

例えば，足し算を行う電卓の受け付ける入力はリスト 1.1 のものが考えられる．

リスト 1.1: 足し算を行う電卓の入力例

```
1.23+4
```

スキャナは，この入力文字列を適切な方法で処理し，最終的には,「1.23」,「+」,「4」の 3 つのトークンの列を生成する．

構文解析は，字句解析の後段の処理に当たる．構文解析の入力は，字句解析より得られたトークン列である．トークン列を，意味を持つ木構造にする過程が**構文解析**である．構文解析を行うツールを**構文解析器**，あるいは**パーサ**と呼ぶ．構文解析により生成される木構造のことを**構文木**と呼

ぶ．トークンは，構文木の葉に位置する．また，処理系によっては，生成される構文木から，さらに後段の処理において不要な情報を抜き取り，より簡潔な構造を生成するものもある．このような構文木は**抽象構文木**と呼ばれる．構文解析より直接得られる構文木は**具象構文木**と呼ばれる．本文では，特定の言語や入力形式を対象とせず，抽象構文木を扱わない．したがって，単に具象構文木のことを構文木と呼ぶことにする．

字句解析と構文解析は，一続きで行われることがほとんどであるため，字句解析と構文解析のことを，合わせて構文解析と呼ぶことも少なくない．本文では，字句解析の手法と構文解析の手法を別々に扱うので，これらの名称も区別して使用する．

また，字句解析と構文解析の性質から，例えば字句解析の段階では必ずしもトークン列をメモリ上に展開する必要はなく，構文解析器が繰り返して字句解析器に対して問い合わせを行うような実装がより効率的である．そして，構文解析器も，実際には必ずしも構文木を全てメモリ上に展開するとは限らない．

さらに，通常ならば，字句解析の段階が終わってから構文解析が始まるが，構文解析器が字句解析器にフィードバックして，字句解析器の動作を変える処理系も存在する．本文では，このような処理系は考えない．

1.2.2　パーサジェネレータ

ここで，上記の「JSON を処理する」と「サーバのログを処理する」という二つの例の違いについて考えてみる．まず，JSON を分析する場合は，既存の JSON 用のパーサが多く存在する．かつ，そのアプリケーションを製作するプログラミング言語の生産性によっては，言語レベルのサポートも考えられる．JSON は決まったデータフォーマットだからこそ，このようなことが簡単にできる．一方，サーバのログを分析したいといっても，そのログが別々のアプリケーションによって生成されたものかもしれないし，ログの形式に関しては何も決まったルールがあるわけではない．従って，それぞれの形式に対して,「別々に」パーサを作らなければならない．

一般的には，決まった形式に対して，その形式の特徴に応じて細かい制御ができるので，パーサを手書きで作るのは難しいとはいえない．そして，その形式の特徴に応じた最適化もできる．しかし，それでもパーサというほぼ機械的なツールを，わざわざ手書きで作らなければならない，というのはプログラムの生産性の観点でいうと非効率的である．このような需要に対して，**パーサジェネレータ**と呼ばれるツールが作られている．パーサジェネレータを使えば，プログラマはそのデータの文法を記述するだけで，自動的にパーサのプログラムを生成することができる．

パーサジェネレータは，基本的には文脈自由文法（第 1.2.5 節）の部分集合を対象とし，これに属する文法をパーサのプログラムに変換する．これができるのは，ある文法の集合に対して，それを機械的に解析できるアルゴリズムが存在するからである．

ここで注意すべきなのは，パーサジェネレータを使うからといって，解析したいデータの文法を「事前に知らなければならない」ということに変わりはない，ということである．言い換えると，十分に賢いパーサジェネレータであれば，与えられた文法を分析して，効率の良い（並列化も含む）パーサのプログラムを生成する可能性が高い，ということである．ただ，今実装されているパーサジェネレータは，十分に賢いとはいえず，また多くの研究はパーサジェネレータが対象とする形式文法の表現力に注目している [18, 19]．このようなツールは，モダンなプログラミング言語といった複雑な構造を対象とすることが多く，ときには生成されるパーサの速度が犠牲になる．

1.2.3 BNF

BNF（Backus-Naur Form，BN 記法）は，文法を定義するのに使用する記法である．数十年前に提案されており，数々の改良が加えられている．現在使われているのは，ほとんどその派生である．BNF は簡潔でわかりやすく，本文では BNF を厳密に定義しないが，例えばリスト 1.2 が BNF を用いた文法の記述例である．

リスト 1.2: BNF を用いた文法の記述例

```
n : [0-9]+ ;
S : A
  | B
  ;
A : A '+' B
  | B '+' B
  ;
B : B '*' n
  | n
  ;
```

この例は，足し算と掛け算を表した文法である（括弧は含まない）．

字句解析と構文解析の規則は，通常は分けて記述する．また，字句解析の規則の記述は，BNF とは無関係であるが，その記述法は構文解析の規則の記述法とよく似ている．したがって，本節ではこれらをまとめて説明する．例えばこの例では，1 行目が字句解析の規則であり，それ以降が構文解析の規則である．字句解析の規則は，左辺が**終端記号**，右辺が（実用性の観点から）拡張された正規表現である．この正規表現は，しばしその終端記号の**パターン**と呼ぶ．構文解析の規則は，左辺が**非終端記号**，右辺が記号（終端記号または非終端記号）の列である．また，上記の `A` と `B` の生成規則に縦棒「|」が入っているが，これは複数の生成規則をまとめて記述できるものである．規則からもわかるように，非終端記号は，1 つ以上の記号を置き換えられるものとして定義される．

本文で用いられる拡張正規表現の記号を表 1.1 に示す．

表 1.1: 本文で用いられる拡張正規表現の記号

記号	意味
.	任意の一文字
?	直前の表現が 0 個または 1 個
*	直前の表現が 0 個以上
+	直前の表現が 1 個以上
[...]	括弧内の文字のどれか一文字
^	「[...]」の中に記述するときは，その括弧内の文字以外の一文字

また，拡張正規表現の記号を構文解析の規則に記述できるような拡張もあるが，生成される構文木の形が文法から読み取りにくいという観点から，本文では触れないことにする．

1.2.4 有限オートマトンによる字句解析

本節では，有限オートマトンを使った字句解析の方法を紹介する．より詳しい説明は，形式言語の教科書 [1] 等を参照されたい．ここでは，第 3 章で提案する手法の基礎となっている部分を紹介する．

まず，有限オートマトンを定義する．本文で扱う有限オートマトンは，非決定性有限オートマトンと決定性有限オートマトンとがある．

定義 1.1. 非決定性有限オートマトン（Nondeterministic Finite Automaton，NFA）は，$\{S, \Sigma, T, s_0, F\}$ から構成される．

- S は，状態の有限集合である．
- Σ は，入力文字の有限集合である．
- $T : S \times (\Sigma \cup \{\epsilon\}) \to 2^S$ は，遷移関数である．これは，入力文字（あるいは空の入力文字）と状態のタプルを受け取り，状態を返す関数である．2^S は，集合 S の冪集合である．ϵ は，空の入力文字を意味する．すなわち，入力文字を受け取らずに遷移することを示す．このような遷移のことを特に**エプシロン遷移**と呼ぶ．
- $s_0 \in \Sigma$ は，開始状態である．
- $F \subseteq S$ は，受理状態の集合である．

複数の開始状態を許すという資料もあるが，本文では，開始状態を唯一のものとする．ただし，ここに本質的な違いがあるわけではなく，複数の開始状態にエプシロン遷移する新しい開始状態を 1 つ作れば，両者は実質的に等価になる．

定義 1.2. 決定性有限オートマトン（Deterministic Finite Automaton，DFA）は，NFA の一種である．ただし，遷移関数 T に関しては，

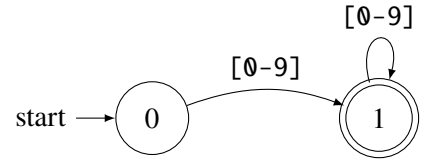


図 1.1: 決定性有限オートマトンの例

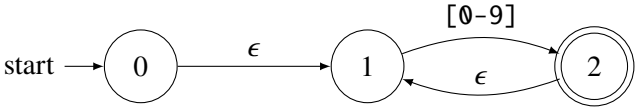


図 1.2: NFA の例

- 入力は，$S \times \Sigma$ である．すなわち，空の入力文字を含まないものである．
- 出力は，大きさが 1 であるような S 部分集合と空集合からなる集合である．すなわち，ある入力文字と状態が与えられたときに，遷移先となる状態の数は高々1 である．

有限オートマトンは，有向グラフで表すことができる．グラフの各辺は遷移を表し，各頂点は状態を表す．また，通常は受理状態を二重丸として表記する．

例えば，リスト 1.2 において，`[0-9]+`に対応する DFA を図 1.1 に示す．

開始状態は，特殊な矢印で示されており，各遷移を表す矢印の隣には，その遷移に対応する入力文字である．

次に，有限オートマトンを使った文字列のマッチングを説明する．文字列のマッチングは，与えられた有限オートマトンによって，文字列が受理されるかどうかを判断する作業である．現在の状態を開始状態とし，入力文字を一つずつ読み取り，現在の状態を遷移関数に従い更新する．入力文字を全て読み切った時点で，更新された現在の状態が受理状態であれば文字列は受理される．ここで注意すべきなのは，NFA がエプシロン遷移を含み，かつある入力文字を受け取ったときに複数の遷移先が存在する場合がある，ということである．すなわち,「現在の状態」というのは，複数の状態が取りうるということである．ある状態と，その状態からエプシロン遷移で到達可能な状態を合わせて，その状態の**エプシロンクロージャ（epsilon closure）**と呼ぶ．

全ての正規表現は，それと等価な NFA に変換可能である．単純な文字は，その文字に応じた遷移を作成すれば良い．正規表現の連結は，二つの NFA をエプシロン遷移で順に繋ぐ．正規表現のユニオンは，新しい開始状態を作り，両方の NFA の開始状態にエプシロン遷移する．繰り返しは，NFA の開始状態に戻るようなエプシロン遷移を作成する．

全ての NFA は，それと等価な DFA に変換可能である．等価というのは，両者の受理する文字列の集合が同じであるということである．証明は例えば [1] に譲るが，ここでは NFA を等価な DFA に変換する方法を説明する．

例えば，図 1.2 に示す NFA を DFA に変換する．

まず，状態 0 から開始し，この状態のエプシロンクロージャは $\{0, 1\}$ であるから，NFA の状態

集合 $\{0,1\}$ を表す DFA の開始状態を構築する．この状態を 01 する．次に，$\{0,1\}$ から，0 から 9 までのいずれかの文字を読んだら状態集合 $\{2\}$ に遷移するが，やはりこのエプシロンクロージャを考えると，$\{1,2\}$ となる．これを DFA の一つの状態として構築する．この状態を 12 とし，状態 01 から状態 12 への遷移を作成する．そして，$\{1,2\}$ から，0 から 9 までのいずれかの文字を読んだら，やはり状態集合 $\{1,2\}$ に遷移する．これはすでに DFA の状態 12 として作られているので，新しい状態を作成せずに，状態 12 から状態 12 への遷移を作成すれば変換は終了である．その結果，DFA の各状態の名前の違いを除けば，図 1.1 と同じ DFA が変換の結果として得られる．

このアルゴリズムはすなわち，ある入力によって到達可能な状態の集合を，構築される DFA の一つの状態とする方法である．このアルゴリズムを，**部分集合構成法**と呼ぶ．状態の集合 S の冪集合の大きさは，$2^{|S|}$ なので，構築される DFA の状態の集合の最大値も，明らかに $2^{|S|}$ である．

ある DFA から，状態数が最小となるように，等価な DFA を構築する操作を最小化と呼ぶ．DFA の最小化の最も実装しやすい方法を以下に紹介する．まず，DFA の遷移関数を反転して NFA を作り，得られた NFA を部分集合構成法より DFA に変換する．そして，この操作をもう一度繰り返すと，元の DFA を最小化した DFA が得られる．

文字列のマッチングに NFA ではなく DFA を使うことの最も大きい利点は速度にある．NFA を使用する場合は，複数の遷移先が可能なので，その全ての状態を考える必要があるのに対し，DFA は常に高々1 つの遷移先しか存在しない．ただし，DFA の状態数は,「通常は」NFA よりも多いことにも注意されたい．

次に，有限オートマトンによる字句解析について説明する．各終端記号に対応する正規表現（パターン）は，オートマトンに変換される．また，字句解析器は，入力文字列を正しく複数のトークンに分割しなければならない．つまり，字句解析器は，単なる受理かどうかを返すのではなく，正しくどれかのパターンを選び，トークンを生成していく必要がある．これが，字句解析と一回のマッチングの大きな違いである．

通常は，字句解析の規則に対応する全てのオートマトンから一つのオートマトンを構成する．ここではまず，NFA を構成するときの字句解析器の動作について説明する．例えば，n 個の終端記号に対応するパターン p_1，p_2，$\cdots$，p_n があるときに，パターン $p_1|p_2|\cdots|p_n$ に対応するオートマトンを構成する．このとき，各終端記号のパターンの対応するオートマトンの受理状態は，その終端記号にマークされる．これが，マッチングが成功したときに，出力として保持される終端記号の情報である．

NFA の受理状態に遷移したときにおいても，複数のパターンにマッチしたという可能性が常にある．このとき，どのパターンを選ぶか（すなわちどの終端記号として解析するか），あるいはトークンを生成せずにこのままマッチングを続けるかは完全に処理系に依存する．ここでは，広く使われている**最長一致（maximal munch）**の手法を説明する．通常は，受理状態に遷移したときの入力の位置を記録して，さらにこれ以上マッチングできないところまでマッチングを続ける．

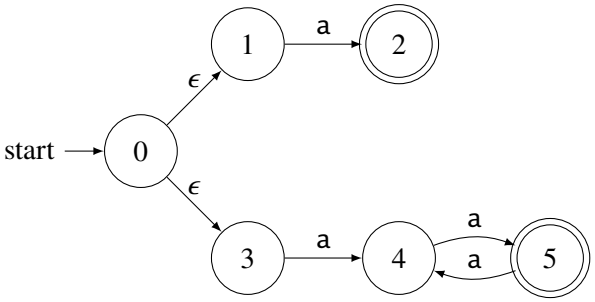


図 1.3: リスト 1.3 より生成される字句解析の NFA

もし途中で受理状態に遷移したならば，その入力の位置にポインタを更新する．そして，最後の状態が受理状態でなければ，記録した位置まで入力ポインタを巻き戻し，マッチングを終了する．さらに,（これは非常にレアなケースではあるが）同じ語彙素で複数のパターンにマッチングした場合は，どのパターンを選びトークンを生成するかは，例えば規則間の優先順位を決めれば解決可能である．例えば，処理系によっては，先に宣言した規則がより高い優先順位をもつ場合がある．ここで注意すべきなのは，通常は，なるべくこのような状況が発生しないように，一つのパターンが他のパターンのプレフィックスとならないように，フォーマットが決められる．

例えば，リスト 1.3 に示す規則は，最終的に図 1.3 に示す NFA に変換される．

リスト 1.3: 最長一致が適用される例

```
A : a ;
A2 : (aa)+ ;
```

この例では，状態 2 が記号 A に対応しており，状態 5 が記号 A2 に対応している．これに aaa を入力し，字句解析を行うときの動作を説明する．字句解析器は，受理状態に遷移したときの入力位置を保持するポインタと，入力を読み取るためのポインタを 2 つもつ．字句解析器が 1 つ目の a を読み取ったら，状態集合 $\{2,4\}$ に遷移する．このとき，受理状態である状態 2 が集合に含まれるので，パターンのマッチングが終わる可能性があることを意味する．従って，字句解析器はそのままトークンを生成せず，このときの位置をいったん記録する．なぜなら，字句解析器は，可能なトークンのうち，最も長いものを生成しなければならないからである（最長一致）．次に，2 つ目の入力 a を読み取り，状態集合 $\{5\}$ に遷移する．状態 5 しか含まれないが，これもまた受理状態なので，字句解析器はポインタをこの位置に更新する．さらに，3 つ目の入力 a を読み取り，状態集合 $\{4\}$ に遷移する．ここで入力が終了したが，受理状態ではないため，入力を先ほど記録したポインタの位置まで戻し，その状態（状態 4）に対応する記号 A2 をトークン A2(aa) として生成する．最後に，再び 3 つ目の入力 a を読み取り，トークン A(a) を生成する．

次に，各オートマトンから DFA を構成するときの動作を説明する．基本的なやり方は NFA を構成するときと変わらない．まず，NFA を上述の手法により構成する．この NFA を，部分集合構成法により DFA に変換し，各受理状態のマークを，DFA の受理状態に引き継ぐ．DFA の各受

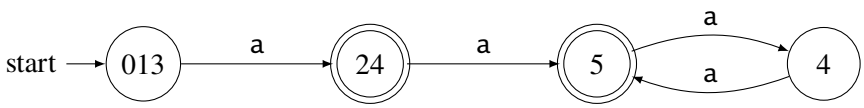


図 1.4: リスト 1.3 より生成される字句解析の DFA

理状態は，NFA のいくつかの状態の集合に対応し，そのうちの 1 つ以上の状態が受理状態であることは明らかである．NFA の受理状態のマークに基づき，DFA の各受理状態にもマークを付ければ良い．もし DFA の一つの受理状態が複数の記号に対応する場合は，より先に宣言される記号に対応させるのが一般的である．NFA を利用する場合は，字句解析器が動的にこのような選択をしなければならないのに対し，DFA を利用する場合は，この曖昧性は字句解析を始める前に解決される．

例えば，上記の例において，図 1.3 の NFA を，図 1.4 に示す DFA に変換できる．DFA の各状態の名前は，その状態が NFA のどの状態に由来するかを示している．

1.2.5 文脈自由文法による構文解析

本節では，文脈自由文法が既知のものとして，本文で使用する構文木の構築等の説明を行う．実際にファイルを解析するときは，概ね文脈自由文法のサブセットである形式文法が使用される．

文脈自由文法による構文解析は，本質的には，開始記号から入力文字列，または入力文字列から開始記号への導出である．

文脈自由文法による構文解析は，曖昧である．すなわち，ある文法とその文法に従う何らかの入力が与えられた場合，複数の構文木が構築される可能性がある，ということである．

よく知られている例として，例えばリスト 1.4 に示すのが一般的なプログラミング言語の条件分岐である．言語によって細部が異なるかもしれないが，概ねこれと同じ構造をしている．

リスト 1.4: 曖昧さを含む文法の例

```
stat : IF expr THEN stat
     | IF expr THEN stat ELSE stat ;
     | OTHER ;
```

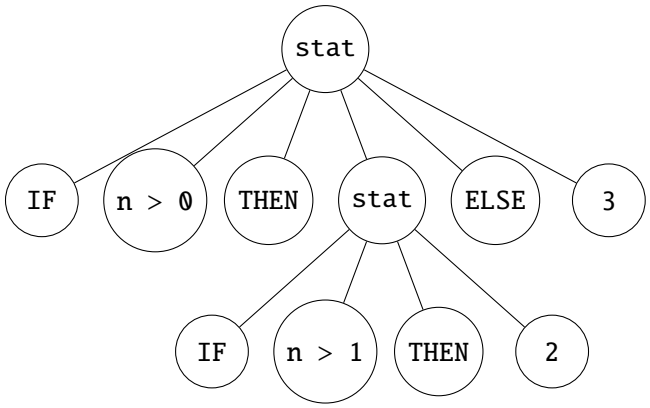
この文法は，曖昧である．2 通りの構文木に解析できる入力を，リスト 1.5 に示す．

リスト 1.5: リスト 1.4 に対する入力の例

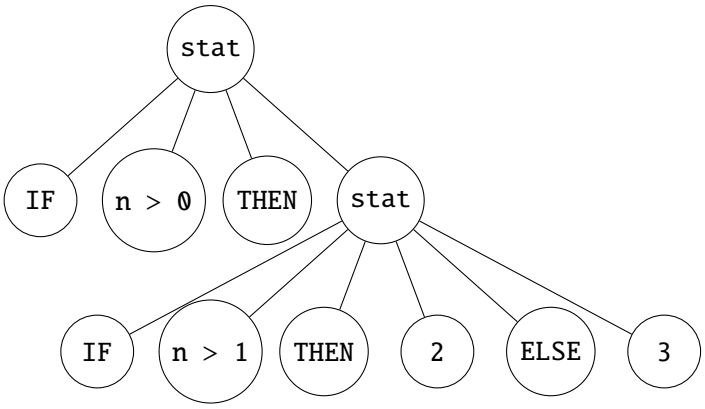
```
if n > 0 then if n > 1 then 2 else 3
```

この入力は，図 1.5a と図 1.5b の構文木に解析でき，`else` が 1 つ目の `if` にくっつくか，あるいは 2 つ目の `if` にくっつくかという意味の違いがある．簡単のために，`expr` 等は省略した．

単なる足し算を解析するときでも，このような曖昧な文法を作ることができるが，足し算は結合則を満たすのでこの問題は実質的な影響を及ぼさない．しかし，上記の `if` と `else` の場合は，



(a) else が 1 つ目の if にくっつく場合



(b) else が 2 つ目の if にくっつく場合

図 1.5: 曖昧性のある文法から生成される 2 種類の構文木

プログラムの意味が変わってしまう．したがって，実用的な構文解析に利用するためには，文脈自由文法から範囲を狭め，曖昧さをなくさなければならない．

また，複数の非終端記号の規則がある場合に，いかに正しくかつ効率的に生成規則を選ぶかが，実用的な構文解析器を作る上で重要となる．さらに，上記の 2 つの条件を満たし，文法クラスの表現力が強いほど良いと言える．これが，現在行われている構文解析の研究の方向である．

1.2.6 PEG

解析表現文法（Parsing Expression Grammar，PEG）は，曖昧さのない形式文法である．ここで注意すべきなのは，曖昧さの有無は，その言語の表現力と直接的な関係がないということである．実は，PEG は，曖昧さのない点においては，よく CFG と比較されるが，CFG で表現できるが PEG で表現できない言語が存在するかどうかはまだ定かでない [9]．逆に，PEG で表現できるが CFG で表現できない言語が存在する可能性は高い（がまだ証明されていない）[8].

CFG において，「|」は左右の生成規則のどれかを選ぶことを意味するが，PEG では，これの代

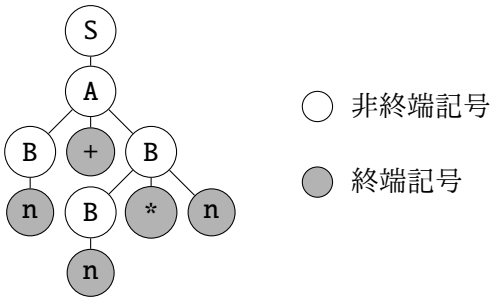


図 1.6: リスト 1.2 の文法に入力 n+n*n が与えられたとき構築された構文木

わりに，順番に試すことを意味する「/」を使用する．すなわち,「/」の左側に来る規則は，右側に来る規則よりも優先される．もし左側の規則に，入力文字列（の一部）がマッチした場合は，右側の規則は適用されない．他にも，PEG は強力な先読みを持っている．

PEG の規則の定義をそのまま構文解析器として実装することができる．そして，その実装も多く存在する[1]．解析速度に関しては，PEG の構文解析器は入力の長さに対して，最悪指数時間がかかってしまう．また，線形時間で解析できるアルゴリズムも存在するが，メモリを大量に消費してしまう．

1.2.7 ボトムアップ構文解析

ボトムアップ構文解析は，構文木を葉から根まで作成する構文解析法の総称である．ボトムアップ構文解析の典型的な手法として，shift-reduce 構文解析と呼ばれるものがある．Shift-reduce 構文解析の詳細は，教科書 [1] を参照されたい．本節では，第 1.4 節で紹介する OPG と関連のある部分のみ説明する．

Shift-reduce 構文解析では，パーサはスタックをもち，そこに文法記号を入れ，解析したい入力記号列の残りの部分を入力バッファに保持する．また，入力を左から右まで順に読む．パーサは，各位置における入力が与えられた場合，これをそのままスタックに積んで次の入力を読む（shift）か，スタックの先頭のいくつかの記号を生成規則に従って置換する（reduce）．最終的に，スタックには開始記号のみが入っており，かつ入力が空になれば受理となる．例えばリスト 1.2 に示す文法に対して，開始記号を S とし，入力が n+n*n として与えられた場合のパーサの動作を表 1.2 に示す．

ただし，スタックの底及び入力の右端を $ で表すことにする．また，この場合構築された構文木を図 1.6 に示す．

表 1.2 は，単にこの入力を正しく解析できるアクションを示しただけであり，構文解析器がなぜそのアクションを取ったのか，という根拠を示していない．実際，shift-reduce 構文解析は,「現

[1]例：PEGTL. https://github.com/taocpp/PEGTL（2020 年 1 月アクセス）

表 1.2: リスト 1.2 の文法に入力 n+n*n が与えられたときの shift-reduce パーサの動作

ステップ	スタック	入力	動作
1	$	n+n*n$	shift
2	$n	+n*n$	B : n による reduce
3	$B	+n*n$	shift
4	$B+	n*n$	shift
5	$B+n	*n$	B : n による reduce
6	$B+B	*n$	shift
7	$B+B*	n$	shift
8	$B+B*n	$	B : B * n による reduce
9	$B+B	$	A : B + B による reduce
10	$A	$	S : A による reduce
11	$S	$	受理

在の状態から次の記号に shift する」か「スタックにある記号を reduce する」か，という曖昧性があり，これは shift-reduce 競合と呼ばれる．また，reduce すると決められる場合においても，複数の reduce 先が存在する（すなわち左辺は異なるが右辺は同じ生成規則が複数存在する）場合，構文解析器はどの非終端記号に reduce するか決めることができず，reduce-reduce 競合と呼ばれる．すなわち，shift-reduce 構文解析もまた，ある種の構文解析法の総称である．Shift-reduce 構文解析に基づき，曖昧性を含まない文法を取り扱う手法として LR(1) などが実用化されている．また，第 1.4 節で紹介する構文解析法も，shift-reduce 構文解析に基づいたものである．

1.3 並列字句解析と並列構文解析の難しさ

この節では，並列字句解析と並列構文解析を行うにあたり，解決しなければならない問題と現状について説明する．第 1.1 節でも説明したように，入力ファイルを効率よく並列に解析するには，字句解析と構文解析の両方を並列化しなければならない．そして，並列字句解析と並列構文解析の難しさは，本質的には同じである．有限オートマトンを使う字句解析器は,「現在の状態」を記憶し，それを更新しながら解析を進めていく．構文解析器は，手法にもよるが，基本的には「スタック」に類似したデータ構造をもつ．そのスタックの中身は解析が進むにつれ変化する．入力ファイルを分割し，途中から解析を始めた場合，本来その位置において解析器の持つべき情報（本文では，この情報のことを状態と呼ぶ）がわからないという問題に遭遇する．これが並列字句解析と並列構文解析の本質的な難しさである．

DFA を使う字句解析の場合，その DFA の全ての状態から解析を試みることで，字句解析を原理的に並列化可能である．ただし，この場合は，DFA の状態数の分だけ，計算量が大きくなる．本文では,「ある開始状態からマッチングを開始し，それに応じた終了状態を得る」計算を，一つ

のパスと呼ぶ．上に述べるやり方では，最終的にはたった一つのパスしか選ばれないので，その他の全ての計算が無駄になってしまう．通常は，簡単な規則でも，生成される DFA は数十個の状態があり，複雑な規則になるとこれを遥かに超える状態数もあり得る．したがって，並列字句解析を愚直に実装する場合，並列化による台数効果が出るとしても，それが逐次の字句解析よりも性能が出ることは考えにくい．この中で，正規表現の並列マッチングの研究は行われているが [11, 22, 15, 17]，字句解析への応用に関しては，非常に単純な並列字句解析器は，並列の正規表現マッチングができれば自明であるが，特に flex[20] のような複雑なバックトラックを許容するほどの並列字句解析器はまだできていない．

並列構文解析は，並列字句解析のようにはいかない．なぜなら，字句解析器の取れる状態は，高々DFA の状態数の分あるのに対し，構文解析器の取れる状態は，無限にあるからである．そして，ほとんどの構文解析アルゴリズムは逐次のものである．通常は，構文解析器の状態を決めるには，入力の先頭から順に読む必要がある．現状として，構文解析を並列に行うことが可能で，かつ実装しやすい構文解析法は，第 2.2.1 節で紹介する OPG（第 1.4 節参照）の構文解析法があげられる．

1.4 演算子優先順位文法

演算子優先順位文法（Operator Precedence Grammar，OPG）は，Floyd らによって提案された形式文法である [7]．OPG は，表現力においては広く使われているほかの形式文法ほど強力ではないが，local-parsability と呼ばれる性質をもつ．これは，構文解析を入力の途中から始めることができる，という性質である．

本文では，OPG を扱う際に，まずいくつかの表記方法を約束する．

- 空の文字列を ε で表す．
- 終端記号を a，b のように，アルファベットの小文字で表す．
- 非終端記号を A，B のように，アルファベットの大文字で表す．
- 終端記号の列を r，s などのように表す．
- V_T は終端記号の集合を表す．
- V_N は非終端記号の集合を表す．
- 何らかの記号，すなわち $V_T \cup V_N \cup \{\varepsilon\}$ の要素を α，β のように，ギリシャ文字で表す．

R を生成規則の集合，S を開始記号とするときに，ある文脈自由文法 $G = (V_T, V_N, R, S)$ について考える．もしある生成規則の右辺に隣り合う非終端記号がなければ，その生成規則が operator 形式になっていると呼ぶ．さらに，R の全ての生成規則が operator 形式になっていれば，文法 G が OG （operator grammar の略）であると呼ぶ．G が OG になっているときに，$\mathcal{L}$，$\mathcal{R}$ を次のように定義する．

- $\mathcal{L}_G(A) = \{a \in V_T | A \overset{*}{\Rightarrow} Ba\alpha\}$
- $\mathcal{R}_G(A) = \{a \in V_T | A \overset{*}{\Rightarrow} \alpha aB\}$

ただし，B は ε も含むこととする．また，$A \overset{*}{\Rightarrow} \alpha$ とは，A は何らかの生成規則（複数ステップを経ても良い）によって α を生成できる，という意味である．すなわち，やや厳密性に欠けるが，$\mathcal{L}_G(A)$ と $\mathcal{R}_G(A)$ はそれぞれ，A の一番左側，そして一番右側に来る終端記号の集合を意味する．

次に，終端記号の間の優先順位を定義する．

- a は b より優先順位が高いことを $a > b$ と記述する．$a = b \Leftrightarrow \exists A \rightarrow \alpha Db\beta, D \in V_N \wedge a \in \mathcal{R}_G(D)$
- a は b と優先順位が同じことを $a = b$ と記述する．$a > b \Leftrightarrow \exists A \rightarrow \alpha aBb\beta, B \in V_N \cup \{\varepsilon\}$
- a は b より優先順位が低いことを $a < b$ と記述する．$a < b \Leftrightarrow \exists A \rightarrow \alpha aD\beta, D \in V_N \wedge b \in \mathcal{L}_G(D)$

注意すべき点として，$a > b$ は必ずしも $b < a$ を意味しないことである．また，同じく $a = b$ も $b = a$ を意味するとは限らない．リスト 1.2 にある生成規則 `B : B * n | n` がその例である．

全ての終端記号の間の優先順位を記録した表を **OPM（Operator Precedence Matrix）**と呼び，OPM の各セルには，$>$，$=$，$<$ あるいは $\perp$（関係なし）のいずれかが入る．OPM の各セルに優先順位が 1 つしかない，すなわち任意の 2 つの終端記号の間の優先順位がただ 1 つに決まれば，その文法が OPG であると呼ぶ．

また，OPG では，入力の左端と右端には自動的に終端を示す特別な終端記号が入り，本文ではこれを $ と表記する．$ は，ほかの任意の終端記号 a に対して $ < a$ が成り立ち，かつほかの任意の終端記号 a は $ に対して $a >$ $ が成り立つ．

1.5 研究の目的

本研究は，大規模な（数ギガバイト～数百ギガバイト）の入力ファイルを，アドホックに解析する並列処理系の設計・作成・評価を目的とする．

高性能な並列パーサが必要とされるたびに，プログラマが手書きでそれを作る必要がなくなり，文法を変更しようとするとき，文法ファイルだけ変えれば良くなる．また，手書きの並列パーサと同レベルあるいはそれ以上の性能を出し，高性能な並列パーサが存在しないフォーマットに関しても，より簡単な解析ツールの作成法の指針となる．

1.6 本論文の構成

本文の構成は以下である．

第 2 章 関連研究 並列字句解析と並列構文解析の代表的な研究，及び本研究の礎となる研究を紹介する．

第 3 章 処理系の提案と実装 本研究で実際に設計・実装した処理系の説明を行う．特にその適用範囲や欠点，実装できていない点と今後の方向についても詳しく説明する．

第 4 章 評価 本研究で実装した処理系の評価を，実際の大規模なデータを使用して評価する．

第 5 章 結論 本研究の結論を述べる．

第2章　関連研究

本章では，並列字句解析及び並列構文解析に関する先行研究の一部を取り出し，その概要を紹介する上で，本研究の目的を達成する上での問題点を説明する．

2.1　並列字句解析に関する研究

並列字句解析は，第 1.3 節でも述べたように理論的なやり方が存在する．本節では，わかりやすいように，DFA による正規表現マッチングを考える．入力データを複数のチャンクに分割したとき，各チャンクの最初の位置における字句解析器の状態が確定しないという問題が，DFA の全ての状態からマッチングを開始させることで解決できる．

既存の研究は，主に正規表現の並列マッチングに関するものである．入力データの並列マッチングは，本質的には遷移関数を変形することである．この節では，有限オートマトンを変形する方法と，並列マッチング自体の最適化を行う方法をそれぞれ取り上げ，紹介する．

2.1.1　SFA

Sinya らは，有限オートマトンから SFA（Simultaneous Finite Automata）を構成し，入力の並列マッチングを行う方法を提案した [22]．遷移関数を合成した結果をあらかじめ計算し，それを構造化したものが SFA である．これと類似するものとして，NFA と DFA を使用した計算があげられる．DFA は，一般的に NFA よりも状態数が多く，よりメモリを使用するが，NFA を使用した場合，複数の状態を追従しなければならないのに対し，DFA の場合は追従すべき状態は常に 1 つである．DFA を使用した並列マッチングは，常に全ての可能な開始状態を考え，全てのパスを計算しなければならないが，SFA を使用する場合，考えられる全ての遷移パターン（どの状態からどの状態に遷移したか）が SFA の状態として表されるため，追従すべき状態は常に 1 つになる．

SFA の状態数は，ベースとなる DFA の状態数を D とすると，最大で D^D になる．また，現実的には，おおよそ D^4 以下であるという研究結果が出されている．

SFA を使えば，入力の任意の位置からマッチングを始めることができる．最終的に，入力の最初からマッチングを始めたときの結果と合成すれば，入力文字列が受理かどうかがわかる．

2.1.2 正規表現の並列マッチングの最適化

DFA の並列計算自体も，アルゴリズムを見直すことにより高速化できる．Mytkowicz らは，有限オートマトンの並列計算の最適化手法を提案した [17]．同論文には，DFA の並列計算を SIMD 命令を使い，高速化する手法も紹介されている．

まず，SIMD 命令を使った高速化を説明する．S と T をそれぞれ長さ m と n の配列としたとき，S と T の gather 演算 $\otimes$ は，次の式を意味する．

$$(S \otimes T)[i] = T[S[i]]$$

すなわち，T がデータであり，S には T にアクセスするためのインデックスが入っている．状態遷移表に配列を使用した DFA の並列計算は，gather と同じである．例えば，ある入力文字に対する状態遷移を T，現在の状態の配列を S とすると，次の状態を格納した配列 S' は，

$$S' = S \otimes T$$

により計算できる．これが意味することは,「次の状態の配列を計算する」ことが SIMD gather により高速化できる，ということである．

次に，最適化手法について紹介する．入力のマッチングにおいて，ある入力文字を読み取ったあと，複数の異なる状態から同じ状態に遷移する可能性がある．このとき，この先どのような入力文字が来ても，これらのパスは常に同じ遷移をすることになる．すなわち，これらのパスを 1 つにし，計算すべきパスの数を減らすことができる．この最適化は，同論文では convergence optimization と名付けている．

そして，論文では，range coalescing と呼ばれる最適化手法が紹介されている．ある入力文字を読み取ったあとに取れる状態の集合を，その入力文字の**レンジ**と呼ぶ．通常，入力文字のレンジは，有限オートマトンの状態の集合よりも小さい．この最適化手法は，各入力文字のレンジにある状態に基づき，入力文字ごとに遷移表をあらかじめ作るものである．その利点は，遷移表がメモリ上でより小さな連続した範囲を占め，アクセスが速くなることである．

また，複数の状態を次々と計算する過程は，gather に相当するが，この論文の発表時点では，CPU には SIMD 命令として，gather 命令が組み込まれていなかった．したがって，著者らは，他の SIMD 命令を使い，gather に相当する操作を実装した．現在は，SIMD gather 命令がすでに CPU に組み込まれるようになり，SIMD gather を自前で実装する必要がなくなった．

2.2 並列構文解析に関する研究

2.2.1 OPG の並列構文解析アルゴリズム

この節では，Barenghi らが提案した OPG の並列構文解析アルゴリズム [3] について述べる．

第 1.2.7 節で述べたように，shift-reduce 構文解析では，パーサはスタックをもち，入力を読む（shift）か，スタックにある記号の列を非終端記号に変換する（reduce）．OPG の構文解析アルゴリズムは shift-reduce 構文解析であり，パーサはスタックを 1 つもつことになる．ただし，スタックには，記号 x 及び優先順位 p のペア (x, p) が要素として格納される．この場合 x は終端記号あるいは非終端記号のいずれかである．また，本文では，スタックの左側が底，新しい要素が右側に積まれるように記述する．

Barenghi らのアルゴリズムでは，パーサに与えられた記号列が入力の全部か一部かに関わらず，常に同じ逐次の構文解析アルゴリズムが適用される．従って，構文解析アルゴリズムは，入力記号列の一部を引数として受け取るべきである．また，アルゴリズムを明確にするために，パーサは現在の入力位置を示すポインタをもつ．さらに，後述するように，並列に構文解析を行う場合においても，2 つのパーサの構文解析の結果を結合させるときに同じ逐次の構文解析アルゴリズムが適用される．そのため，パーサのスタックには初期値を与える必要がある．

パーサに入力記号列 u が与えられた場合の逐次の構文解析アルゴリズムを図 2.1 に示す．

逐次に構文解析を行う場合は，入力を $\$s\$$ として与え，スタックの初期値を $(\$, \bot)$ として図 2.1 のアルゴリズムを適用すれば良い．そして，スタックが $(\$, \bot)(S, \bot)$ となる場合にのみ受理する（S は開始記号）．

並列に構文解析するときに，入力をいくつかのチャンクに分ける．このとき，境界上にある終端記号は 2 つのチャンクに共有される．すなわち，隣り合う 2 つのチャンクをそれぞれ c_1，c_2 とするとき，$c_1.b = c_2.a$ である．次に，それぞれのチャンクに対して図 2.1 のアルゴリズムを適用し，それぞれ得たスタックを S_1，S_2 などとする．そして，隣り合う 2 つのスタックを図 2.2 に示すアルゴリズムに従って結合する．

例えば，入力 $n+n+n*n*n+n*n+n$ をそれぞれ，$u_1 = n+n+$，$u_2 = +n*n*n+n$，$u_3 = n*n+n$ と，3 つの部分に分割する．それぞれ逐次の構文解析アルゴリズムを適用した場合のスタックは，$S_1 = (\$, \bot)(A, \bot)(+, <)$，$S_2 = (+, \bot)(B, \bot)(+, >)(n, <)$，$S_3 = (n, \bot)(*, >)(n, =)(+, >)(B, \bot)(\$, >)$ となる．S_2 と S_3 を結合する場合，$S_2^L = (+, \bot)(B, \bot)(+, >)$，$S_2^R = (n, <)$ となり，$S_3^L = S_3$，S_3^R は空となる．そして，$S_{\text{combine}}(S_2^L, S_2^R) = (+, \bot)(n, <)$，$u_{\text{combine}}(S_3^L) = *n + B\$$ である．

パーサは入力記号列 u 及びスタック S を受け取る．ただし，入力記号列は $u = asb$，逐次の場合，スタックは $(a, \perp)$ に初期化される．また，head は記号を指すポインタである．head は s の最初の記号を指す．

1. head にある記号 x を読む．また，スタックの一番右側にある終端記号を y とする．y と x の優先順位を考える．
2. $y < x$ であれば，スタックに $(x, <)$ を積んで，head を次の位置に進める．
3. $y = x$ であれば，スタックに $(x, =)$ を積んで，head を次の位置に進める．
4. x が非終端記号であれば，スタックに $(x, \perp)$ を積んで，head を次の位置に進める．
5. $y > x$ であれば，
   (a) スタック S に $<$ が含まれていなければ，スタックに $(x, >)$ を積んで，head を次の位置に進める．
   (b) そうでなければ，スタックにある $<$ のうち，一番右側のものについて考え，そこからスタックの一番右側までの記号列を reduce する．すなわち，スタックに n 個の要素が入っており，その $<$ が i 番目の要素であるとしたとき，$(x_i, <)(x_{i+i}, p_{i+1}) \cdots (x_n, p_n)$ を取り出して $x_i x_{i+1} \cdots x_n$ を reduce する．ただし，元のアルゴリズム [3] には記述されていないが考えうる状況として，x_{i-1} が非終端記号であれば（この場合 p_{i-1} は必ず $\perp$ となる）これも取り出して reduce の対象に含める．非終端記号 A に reduce されたときに，$(A, \perp)$ をスタックに積む．
6. 入力が空になるまで，ステップ 1 から繰り返す．ただし，逐次の場合は，開始記号を B としたときに，スタックが $(a, \perp)(B, \perp)$ になるまで繰り返すとする．

図 2.1: OPG パーサに入力記号列 u を与えた場合の逐次の構文解析アルゴリズム

2.2.2 PAPAGENO

PAPAGENO は，第 2.2.1 節で述べたアルゴリズムに基づき，Barenghi らが実装した処理系である [4].

PAPAGENO は，処理系全体のボトルネックをなくすため，並列字句解析を実装している．PAPAGENO の並列字句解析は，入力文字列を空白や改行文字で分けるという前提で実装されている．例えば，JSON の場合は，空白や改行文字は,「字句の本当の区切り」か「文字列リテラルの中」にしか出現しない．また，プログラミング言語である Lua の場合は，空白と改行文字の出現場所は，高々「コメントの中」と，3 種類に増えるだけである．したがって，入力文字列をこのような場所で分割した場合，DFA の開始状態を限定させることができる．この手法は，昨今使用されている文字列データフォーマットに対してはほぼ常に適用できるといえる．ただし，論文中でも言及されたように，そもそも入力ファイルの中に，このような区切り文字が十分に出現しなければ，効率よく入力ファイルを解析できない．ここで注意すべきことは，区切り文字が出現する回数が多いだけでなく，それらがうまく入力中にばらまかれていなければならない，ということである．

1. S_1，S_2 を隣り合う 2 つのスタック（S_1 は S_2 の左側にある）とし，それぞれのスタックを，左側に $<$，右側に $>$ が現れないように分割する．分割されたスタックを S^L，S^R と表記する．すなわち，例えば S_1 は $<$ を含まない S_1^L と $>$ を含まない S_1^R の 2 つの部分に分割される．
2. S_1 について，$S_{\text{combine}}(S_1^L, S_1^R) := (a, \bot)S_1^R$ を計算する．ただし，a は，S_1^L の一番右側にある記号である．
3. S_2 について，$u_{\text{combine}}(S_2^L)$ を計算する．$u_{\text{combine}}(S)$ は，S の一番左側の要素を取り除き，優先順位を無視して記号のみからなる列である．すなわち，$S = (x_1, p_1)(x_2, p_2)\cdots(x_n, p_n)$ とするとき，$u_{\text{combine}}(S) = x_2 \cdots x_n$ である．
4. $S_{\text{combine}}(S_1^L, S_1^R)$ をスタックの初期値，$u_{\text{combine}}(S_2^L)$ を入力として，図 2.1 のアルゴリズムを適用する．また，得られるスタックを S_m と記述する．
5. S_1^L，S_m，S_2^R を順に連結して，これを結果として返す．ただし，S_1^L の一番右側の記号は，S_m の一番左側の記号と同じものであり，連結するときに，S_m の一番左側の要素を取り除くものとする．

図 2.2: 隣り合う 2 つのスタックを結合するアルゴリズム

例えば極端な場合，最初の 100 文字が全部空白であり，次の 100 万文字に空白が一つも入っていないような入力データがあるとしても，このデータを効率よく解析することはできない．

論文の中では，並列字句解析と並列構文解析の性能を，JSON 及び Lua の入力ファイルを使い，16 スレッドまでそれなりの台数効果が出ると発表されている．しかし，我々は PAPAGENO を使用した検証実験では，並列字句解析，並列構文解析の両方で，いずれも台数効果を再現できなかった．検証実験では，公開されている PAPAGENO のリポジトリにある JSON の入力ファイルを使用した．

台数効果が出ない理由に関しては調査中であるが，ドキュメント等の不足により困難である．以下に現在判明していることについて述べる．PAPAGENO の並列字句解析は，flex[20] をベースに実装されている．並列構文解析の段階で，スレッドはコマンドラインで指定した数だけ作成され，各チャンクの構文解析を行う．また，並列構文解析アルゴリズムを性質を考え，スタックを結合させること自体も計算が必要であることから，PAPAGENO には，`LOG_RECOMBINATION` というフラグが存在する．このフラグをオンにしてコンパイルすると，コマンドラインで指定した数よりも少し多く，スレッドが作成される．これらの余分のスレッドは，スタック結合タスクのために使われる．また，このフラグをオフにして（デフォルトではオフ）コンパイルすると，スタック結合タスクは最初に作成した複数のスレッドとは別の 1 つのスレッドによって実行される．

2.2.3 OPG 以外の形式文法

構文解析を並列化するには，何らかの方法でチャンクの開始状態を決める必要がある．OPG の構文解析は，終端記号間の優先順位さえわかれば，構文解析を開始できる．したがって，入力の

途中から構文解析を開始させるためには，一つ前のチャンクの最後の記号を見ればよかった．

OPG の表現できる言語（Operator Precedence Language，OPL）以外にも，例えば LCPL[5] という形式言語があり，入力の途中から構文解析を開始可能である．これは，OPL よりも広い [16]．しかし，LCPL を解析するための理論が十分に立っておらず，現状としては，「構文解析に必要な情報を正しく生成できれば」解析可能であるが，その情報を有限時間で正しく生成する方法がわかっていない．

第3章　処理系の提案と実装

3.1　提案手法

3.1.1　並列字句解析

第 2.1.2 節で DFA の並列計算の高速化手法を紹介したが，これらの高速化は，簡単ではあるが効果を発揮する．本研究は，これらの高速化手法より，実装しやすく，かつ多くの入力フォーマットに対して効果を発揮する手法を提案する．

字句解析が原理的に可能であることを述べたが，以下にそのやり方を詳しく説明する．最初のフェーズでは，分割された各チャンクに対して，DFA の全ての状態から開始して逐次の字句解析を行う．各開始状態に対応した，結果のトークン列及びチャンクの終わりにおける終了状態を記録する．そして，隣り合うチャンクを，例えばチャンク $i-1$ とチャンク i について，チャンク $i-1$ の各終了状態とチャンク i の各開始状態がマッチングした結果を繋ぎ合わせる．結果，チャンク $i-1$ とチャンク i の結果が合併され，チャンク $i-1$ の開始状態とチャンク i の終了状態が新しいチャンクの開始状態と終了状態となる．最終的に，全てのチャンクの結果が合併されるが，最初のチャンクだけ開始状態が DFA の開始状態であると定まるので，全体として結果のトークン列 1 つ以下となる．もし上記の過程で隣り合うチャンクの終了状態と開始状態がマッチングせず，繋ぎ合わせることに失敗した場合，それは入力文字列が規則にしたがっていないことを意味する．本文では，以降規則にしたがっていない入力文字列を考えないことにする．

上記の並列字句解析手法には，3 つ，欠点が存在する．

1. 1 つの字句の途中で分割された場合，隣り合うチャンクを繋ぎ合わせるときに特別な処理が必要である．特にランダムな分割だとこのようなことが起こる可能性が非常に高い．
2. DFA の全ての状態に対してトークン列を作成しているが，最終的には 1 つしか使用されない．また，通常の処理系は，トークン列をメモリ上に展開せず，構文解析器が字句解析器に対してトークンを問い合わせる形をとる場合が多く，こちらのほうが効率が良い．しかし，この方法ではトークン列をメモリ上に展開せざるをえない．
3. 通常の規則から構築される DFA の状態数は数十〜数百あり，その全てを計算しては，コア数が数十ある計算機でも，逐次の字句解析よりも良い性能が出せる希望は薄い．

問題 1 と 2 を解決するため，上記の手法に修正を加える．修正後の手法が，本提案手法が基礎とするベースラインである．

1. 分割された各チャンクを，DFA の全ての状態から計算を開始させる．このとき，トークンを生成せず，そのチャンクの各開始状態に対応した終了状態のみ記録する．
2. 最初のチャンクだけ開始状態が DFA の開始状態であると定まるので，全てのチャンクの正しい開始状態と終了状態を順に決める．
3. 各チャンクの開始位置から入力を少し読み，開始位置を，それが一つの字句の始まりであるようなところに進める．そして，各チャンクの終了位置も次のチャンクの開始位置に応じて適切に調整する．
4. このようにできた各チャンクは，完全に独立した部分となるので，それぞれに対して逐次の字句解析を行う．

この手法は，トークンを実際に生成するのは最後のフェーズであり，問題 1 は明らかに解決される．また，問題 2 は，最後のフェーズで使用される逐次の字句解析器の実装次第で解決できる．

以降，本文ではフェーズ 1 のことを**プリスキャン**と呼ぶことにする．

上記の手法では，フェーズ 2 とフェーズ 3 は計算量が非常に少なくほぼ無視できるが，まだプリスキャンの部分は計算量が多い．かつ，最初に提示した手法と異なり，合わせて入力文字列を 2 回スキャンすることになってしまうので，結局逐次の字句解析より高いスループットを出す可能性は低い．

並列字句解析を高速に行うには，パスの数を削減しなければならない．例えば，SFA を使用することで複数のパスの遷移が 1 つの状態として表される．また，convergence を行うことで，動的にパスを消すことができる．

本研究は，可能な開始状態ができるだけ少なくなるように，入力を分割する方法を提案する．第 2.1.2 節でも説明したように，ある入力文字を読み取った後，DFA の取れる全ての状態の集合が，その入力文字のレンジである．そこで，ジェネレータが入力の規則を分析し，各入力文字のレンジを求めることができる．

例えば，リスト 3.1 には，一般的な整数を表す規則を示している．

リスト 3.1: 整数を表す規則

```
NUMBER : '0'|'-'?[1-9][0-9]* ;
```

これを DFA に変換した結果を図 3.1 に示す．

この例では，文字「-」を読んだあとの状態は，状態 3 しかあり得ない．すなわち，文字「-」のレンジは，状態集合 {3} である．同様に，文字「0」のレンジは，集合 {1, 2} であり，文字「1」のレンジは，{2} である．一般的に，規則が多くなると文字のレンジも大きくなるが，それでも

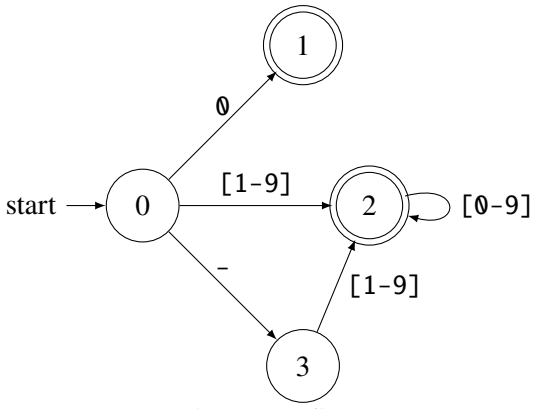


図 3.1: 整数を認識する DFA

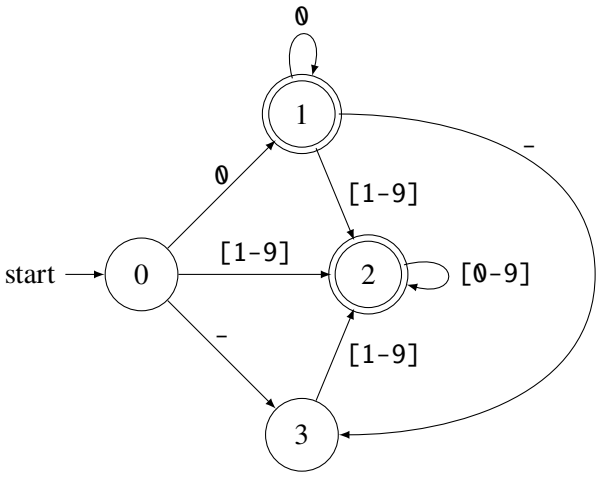


図 3.2: プリスキャン用の DFA

実際に使われる規則であれば，このように比較的にレンジの小さい文字を見つけることは可能である．レンジの小さい入力文字を分割位置にすれば，可能な開始状態を多く減らすことができる．

第 1.2.4 節で説明したように，通常のスキャンは，受理状態に遷移したときの入力の位置を記憶する必要がある．これは，複数のトークン候補があったときに，最長一致の規則を適用するためである．しかし，プリスキャンの段階で，それぞれのパスに関して入力の位置を保持するのはコストがかかる．さらに，今後プリスキャンを SIMD 化により高速化することを検討するとき，複雑なバックトラックロジックは SIMD 化を困難にする．簡単な状態遷移でプリスキャンを行うことが理想的であり，提案手法は，字句解析で 2 つの異なる DFA を使用する．具体的には，最初のプリスキャンのフェーズでは，フェーズ 3 以降とは状態数が同じ別の DFA を使用する．プリスキャンの DFA を構成するには，オリジナルの DFA に，各受理状態から，開始状態の次の状態へ指す遷移を作成する．新しく作る遷移の入力は，開始状態からその遷移先への遷移の入力と同じである．

例えば，図 3.1 に示す DFA からは，図 3.2 のプリスキャン用の DFA が生成される．

この手法は，通常のフォーマットに対しては成立するが，以下に述べるマイナーなケースでは

正しく動作しない．

最長一致の規則がどのようなときに適用されるかというと，あるパターンが他のパターンのプレフィックスになっているときである．そのため，字句解析器は，受理状態に遷移したときでも，すぐにはトークンを生成できず，そのときの位置をいったん保存しなければならない．言い換えると，このときの字句解析器の動作は,「入力を読み取り次の状態に遷移する」か，あるいは「トークンを生成し開始状態から改めてマッチングを開始する」かのどちらかである．このような決定的でない遷移は，同じ状態数をもつ DFA における単純な遷移として表すことができない．なぜなら，決定的でない遷移は，オートマトンが非決定的であることを意味するそのものであり，NFA を等価な DFA に変換しようとすると，状態数が最大で 2 の冪乗になるからである．状態数を増やさずに prescan を行うのが本提案手法の基礎になっており，かつ第 1.2.4 節でも説明したように，実際のフォーマットは，このような規則が出現しないように，うまく設計されている．したがって，実装した処理系は，プリスキャンの構築時に非決定性が生じる規則は受け付けないことにした．

3.1.2 並列構文解析

PAPAGENO の並列字句解析はともかく，並列構文解析において台数効果が出ないのは，固定数のスレッドを使用するからであると考える．OPG の並列構文解析アルゴリズムは，各チャンクを解析する最初のフェーズだけでなく，各チャンクの解析で得られるスタックを結合させることも，同様にタスクであり時間がかかる．

並列ライブラリである Intel TBB は，ユーザがタスクを作れば，あとは処理系が適切にそれらを並列に実行する．

OPG の並列構文解析アルゴリズムはそのまま fork-join に適用できるので，我々は，Intel TBB ライブラリを使い，OPG の並列構文解析アルゴリズムを再実装することにした．

3.2 処理系全体の設計

実装する処理系は，大きく本体とジェネレータの 2 つの部分に分けられる．処理系の構成を図 3.3 に示す．

ここで，本体は，入力を取り扱うバッファやストリーム，字句解析と構文解析のロジック等からなる．ジェネレータは，解析器本体とは別のプロジェクトであり，解析に必要な情報を，解析の前にあらかじめソースコードとして生成するプログラムである．文法ファイルはユーザが記述するものである．文法ファイルをパーサジェネレータ（第 3.4 節）のプログラムに渡せば，処理系本体の必要とするソースコードが生成される．このソースコードを，処理系本体のソースコード

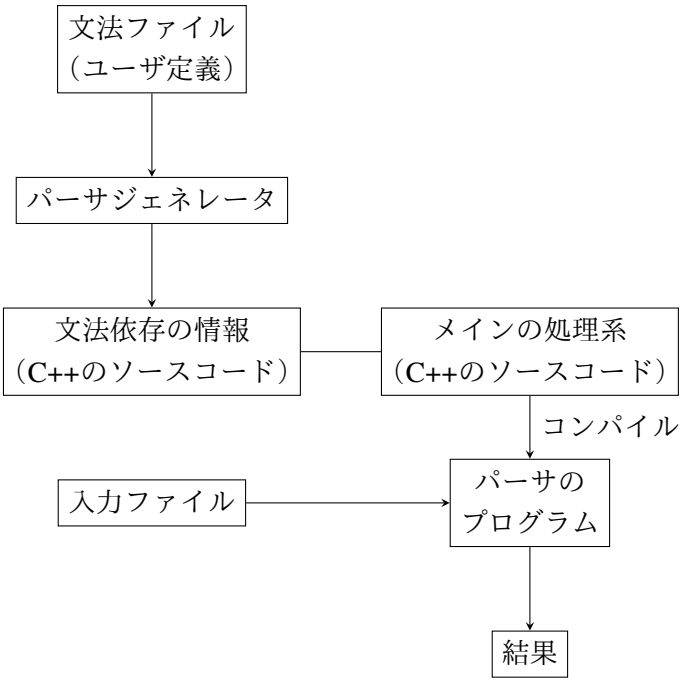


図 3.3: 実装した処理系の構成

と同時にコンパイルすれば，解析器が生成される．解析器は，定義した文法に従う入力を認識し，それを解析することができる．

特に，処理系本体のソースコードは，文法には依存せず，あらかじめコンパイルしても良い．

3.3 処理系本体の実装

本節では，処理系本体の実装について説明する．処理系は，実装の効率と実行の効率両方を考え，C++言語を使用した．スタンダードは，C++14 である．

本節で実装を説明するときに，ソースコードを示すことがあるが，それが処理系で実装されたもののごく一部であり，完全なものではない．示したソースコードは，あくまでその文脈における説明を補完するものである．

3.3.1 記号のデータ構造

記号は，リスト 3.2 で示すように，非終端記号と終端記号をまとめて同じ構造体 `Symbol` にした．

リスト 3.2: 記号クラスの定義

```
struct Symbol {
   using InnerValueType = uint32_t;
   // ...
private:
```

```
    InnerValueType _value;
    SemanticValue _semanticValue;
};
```

これは，無駄な継承によるオーバヘッドを避けるためである．パーサは，ある記号が非終端記号か終端記号かを判断する必要があるが，それが頻繁には起こらない．非終端記号か終端記号を区別するのに，_value フィールドのあるビットが，非終端記号のときには 1，終端記号のときには 0 となるように，あらかじめジェネレータのほうで決めている．パーサは，ある記号のオブジェクトを扱うときに，確認すべきなのは，その記号の_value フィールドのみである．

特殊な記号として，実装では INVALID と SIDE の 2 種類にしている．INVALID は，解析に失敗したなどを意味する無効な記号や C 言語での EOF と似た意味をもつチャンクの終わりを示すものである．SIDE は，字句解析の結果のトークン列の両端に暗黙に加えられ，構文解析器に渡される．これらの記号の_value は，生成される記号のそれと矛盾しないように，あらかじめ決められている．

3.3.2 入力の扱い

字句解析器の対象とする入力は文字列であり，また構文解析器の対象とする入力はトークン列である．両者とも入力を先頭から順に読む性質がある．これは，入力をストリームとして扱うことができることを意味する．

まず，入力ファイルを扱うストリーム InputStream について説明する．ファイルを効率に扱うためには，通常は入力ファイルをメモリにマッピングする．ただし，マッピングされた配列は，アクセスする位置によっては，ディスクからロードする必要がある．これは，本来の処理とは無関係な時間を計測にもたらしてしまう．また，入力ファイルを分割して扱う必要があり，やはり内部でバッファを持たせたほうが実装がより簡潔になる．これら問題に対処するため，2 種類のバッファ型 MappedBuffer 及び LoadedBuffer を実装した．MappedBuffer は，初期化時に mmap システムコール等を使用しファイルをロードする．LoadedBuffer は，初期化時にファイルを内容を全部ユーザ空間の配列にロードする．また，InputStream は，これらのバッファのいずれかをもち，バッファ型はテンプレートとした（リスト 3.3）．どのタイプのバッファを使用するかは，テンプレート引数の指定により切り替えられる．

リスト 3.3: 入力ファイルを扱うストリーム

```
template <class Buffer>
class InputStream {
    Buffer _buf;
    // ...
};
```

次に，記号列を扱うストリームについて説明する．並列構文解析では，トークン列を途中で分割して，隣り合うチャンクの解析において，1 つだけトークンを共有する必要がある．また，後半のチャンクを解析するパーサは，入力があたかも完全なストリームのように記号を読み進めていく．すなわち，記号列を扱うストリームは，途中で分割されても，分割されたそれぞれの部分やはりストリームとして成り立つ必要がある．このような需要に対し，一番簡単な方法は，構文解析器にはストリームであるかのように見せかけ，中ではバッファを用い全てのトークンを保持する方法である．これをリスト 3.4 に示す BufferedSymbolStream として実装した．

リスト 3.4: トークン列を扱うストリーム

```
class BufferedSymbolStream {
    const std::vector<Symbol> _tokens;
    // ...
};
```

字句解析器が新しいトークンを生成するたびに，std::vector::emplace_back を呼び出し，それを配列に追加していく．各チャンクに対して，トークン列をメモリに保持することは非常に非効率的である．具体的には，トークンの数が多いときに，メモリを大量に消費する．また，std::vector の性質から，そのキャパシティが足りないときに，メモリの再確保やコピーが発生する．これは，処理系にとって非常に無駄な処理であり，トークン列をメモリに展開するときとしないときとでは，実行時間に 20 倍以上の差を確認した．ただし，現状では，トークン列をメモリに展開せずとも完全にストリームとして振舞う実装はまだできていない．

また，例えば並列構文解析において，隣り合うチャンクのスタックを結合させるとき，右側のスタックにある記号を入力記号列として扱う必要がある．これに対処するため，パーサのスタックの中身を参照しながら記号のストリームとして振舞う StackSymbolStream を実装した（リスト 3.5）．

リスト 3.5: パーサのスタックを記号列に見せかけるストリーム

```
class StackSymbolStream {
public:
    StackSymbolStream(const Stack &stack) noexcept;
    // ...
};
```

3.3.3 字句解析の実装

字句解析において，DFA をはじめとする情報が必要になる．この情報は，ジェネレータによって生成される（第 3.4 節）．まず，DFA の状態は，リスト 3.6 で示すように，State 構造体で表した．

リスト 3.6: DFA の状態

```
struct State {
    using InnerValueType = int_fast32_t;
private:
    InnerValueType _value;
};
```

簡単なために，状態を表す整数型を 32 ビット（実験のプラットフォーム）としたが，状態数が少ないときに，これを 8 ビットなどにすると，使用するメモリが少なくなり，プログラムがより高速に動作する．簡単な実験では，状態を表す整数を 32 ビットから 8 ビットにすることで，字句解析では約 40 パーセントの速度向上を確認した．また，DFA の状態遷移に失敗したときの状態を表すものとして，`INVALID_STATE` という状態が常に存在する．これには通常の状態とはかぶらない特殊な`_value` フィールドが与えられる．

字句解析に必要な情報は，ほとんど `ScannerDFA` というクラスに記述した（リスト 3.7）．

リスト 3.7: 字句解析に必要な DFA の情報

```
class ScannerDFA {
public:
    static const State startState;
    static const std::size_t stateCount;
    static const State::InnerValueType *getTransitionsByInput(char c);
    static const std::vector<State> &range(char c);
    static bool hasSmallRange(char c);
    static bool shouldSkip(Symbol raw);
    // ...
};
```

これに対応する cpp のファイルは，パーサジェネレータによって生成される．

DFA の状態遷移関数は，リスト 3.8 に示す二次元配列とした．

リスト 3.8: 状態遷移表

```
static const State::InnerValueType transitionTable[VALID_CHARACTER_COUNT][STATE_COUNT];
```

`getTransitionsByInput` は，ある入力文字が与えられたときに，それぞれの状態がどの状態の遷移するかを表す一続きの状態の列を返す．これは，後ほど処理系に SIMD 命令を取り入れるときのためである．また，`shouldSkip` は，与えられた記号が，字句解析器によって無視されるかを示す．典型的な使い方として，空白文字の扱いである．多くの字句解析器は，一続きの空白文字をいったん一つのトークンとして認識はするが，それを意味のあるトークンとして生成せずにただ読み飛ばす．このとき，そのようなトークンに対して `false` を返すソースコードを，ジェネレータが生成すれば良い．

並列字句解析は，逐次の字句解析器を必要とする．並列の字句解析器と逐次の字句解析器の両方とも，入力として文字のストリームを受け取り，トークン列を返す．我々は，時間の計測がしやすいように，トークン列を生成するところを単独に分け，リスト 3.9 に示す実装とした．

リスト 3.9: 逐次の字句解析器

```
template <class InputStream>
class SequentialScanner {
    std::vector<Symbol> _tokens;
public:
    SequentialScanner(InputStream &inputStream) noexcept;
    void scan();
};
```

並列の字句解析器は，リスト 3.9 とほとんど似たインタフェースとなっており，入力のストリームは，複数に分けてもやはりそれぞれストリームとして振舞う必要がある点で異なっている．現状の実装では，InputStream は内部でバッファを持っており，この条件を満たしている．

並列の字句解析器は，入力を分割し，プリスキャンを行うなどする必要がある．入力を分割した各チャンクを独立に扱うため，チャンクの各種の情報を保持する_ChunkInfo 構造体を作成した（リスト 3.10）．各チャンクに対するプリスキャン等の操作は，ほとんど_ChunkInfo のオブジェクトとのやりとりになる．

リスト 3.10: チャンクの情報を保持する構造体

```
struct _ChunkInfo {
    int id;
    std::size_t start;
    std::size_t size;
    std::vector<State::InnerValueType> startStates;
    std::vector<State::InnerValueType> endStates;
    State startState;
    std::vector<Symbol> tokens;
};
```

あるチャンクが初期化されるときに，そのチャンクの開始位置より 1 つ前の文字のレンジに応じ，startStates が正しく初期化される（最初のチャンクは，開始位置より 1 つ前の文字は存在しないが，このチャンクだけ DFA の開始状態からマッチングを始める）．プリスキャンのフェーズでは，各チャンクの endStates の中身を，逐次に更新する．次に，各チャンクの真の開始状態を保持する変数 startState や size 等を適切に更新する．最後に，各チャンクに対して，逐次の字句解析を行い，その結果を tokens フィールドに保存する．

std::vector に記号を保持するのはメモリの無駄であり，かつ大きな性能低下の原因となる．トークン列を保持しないよう，今後は実装が変更される予定である．

入力は複数のチャンクに分割され，それぞれのチャンクは一つの_ChunkInfo オブジェクトに対応する．全体では std::vector<_ChunkInfo>で複数のチャンクの情報を保持するようにした．入力の分割，すなわち std::vector<_ChunkInfo>の構築は，実装では並列字句解析器の split メソッドが行っている．このメソッドは逐次に実行され，あらかじめ計算される予想のチャンクのサイズに基づき，チャンクの終了位置をいったん計算する．そして，その終了位置から少し読み進め，レンジの小さい（リスト 3.7 の hasSmallRange が true を返すような文字）を見つけ，そこをチャンクの終わりとする．この操作を入力の終わりまで繰り返し，std::vector<_ChunkInfo>を構築する．

第 3.1.1 節で提案したフェーズ 1 とフェーズ 4 は，並列化される．ただし，特にプリスキャンのフェーズは，各チャンクは必ずしも可能な開始状態の数が同じではなく，計算量が異なることが多い．より良い負荷分散を達成させるために，CPU の数と同じ数だけのチャンクに分けるのではなく，実装では，最低のチャンクサイズを 10 MB とした上で，CPU の数よりも多い数のチャンクを作ることを勧める．そして，並列化には，Intel TBB の parallel_for 関数を使用した（リスト 3.11）．

リスト 3.11: プリスキャンの並列化

```
void prescan(std::vector<_ChunkInfo> &chunks) {
    tbb::parallel_for(
        tbb::blocked_range<int>(0, chunks.size()),
        [&](const tbb::blocked_range<int> &r) {
            for (auto i = r.begin(); i < r.end(); i++) {
                this->subPrescan(chunks[i]);
            }
        }
    );
}

void subPrescan(_ChunkInfo &chunk) {
    // ...
}
```

リスト 3.11 では，subPrescan は，あるチャンクをプリスキャンしてそのチャンクの_ChunkInfo オブジェクトを更新するメソッドである．

3.3.4 構文解析の実装

構文解析器に必要な情報は基本的には，終端記号間の優先順位と，生成規則である．終端記号間の優先順位は，パーサが shift か reduce を行う判断をするときに使われ，生成規則は，reduce と判断されたときに記号列をどの非終端記号に reduce か決めるときに使われる．記号間の優先順位は，リスト 3.12 で示すように定義され，リスト 3.13 で示すように，get_precedence 関数により取得する．

リスト 3.12: 終端記号間の優先順位

```
enum Precedence {
    TAKE, EQUAL, YIELD, NONE
};
```

リスト 3.13: 終端記号間の優先順位を取得する関数

```
Precedence get_precedence(Symbol::InnerValueType x, Symbol::InnerValueType y);
```

この関数の実装は，ジェネレータによって生成される．

構文解析器は，記号列を非終端記号に reduce する操作を頻繁に行う．そこで，多くの生成規則は比較的短く，かつ同じプレフィックスを含む生成規則が複数出現することが普通であることを

図 3.4: リスト 1.2 の文法に対応するトライ木

踏まえ，効率のために，全ての生成規則をトライ木の構造でまとめて保持することにした．生成規則の右側にある記号を，順にトライ木の検索のキーとし，左側の非終端記号を，検索の結果とする．例えばリスト 1.2 に示す文法は，トライ木に構成すると図 3.4 に示すものとなる．

トライ木を使うことで，記号列の reduce 先の探索は速くなる．トライ木の構築や検索を手書きで実装し，リスト 3.14 で示すように，自動生成される挿入のコードが簡潔になるよう実装した．

リスト 3.14: 自動生成されるトライ木の中身

```
this->insert({0x1, 0x80000000}); // json : STRING
this->insert({0x2, 0x3, 0x80000000}); // json : LBRACE RBRACE
this->insert({0x2, 0x80000001, 0x3, 0x80000000}); // json : LBRACE member RBRACE
this->insert({0x2, 0x80000002, 0x3, 0x80000000}); // json : LBRACE members RBRACE
// ...
```

現状では，トライ木の実装に `std::map` を使用している．これは，実装自体は簡単であるが，メモリを消費する．トライ木の高速な検索や省メモリに関する研究は行われており [10, 2]，ここの最適化には議論の余地が残る．

OPG の構文解析器は，reduce を行うときに，スタックのトップにある < の優先順位とペアになる記号から，スタックのトップにある記号まで，一つの非終端記号に変換する．すなわち，OPG の構文解析器の使用するスタックは，トップにある < の位置を高速に取得する必要がある．そこで，リスト 3.15 に示すように，記号と優先順位を格納するコンテナの他に，それぞれの < の優先順位の出現位置を記憶する．

リスト 3.15: スタックの実装

```
class ParserStack {
    std::vector<Pair> _data;
    std::vector<std::size_t> _yields;
public:
    std::vector<Symbol> popLastGroup();
    // ...
};
```

入力データを並列に構文解析するときに，入力データを複数のチャンクに分割して最終的に 1 つのスタックに結合させるが，この場合，ある結合操作が必要とするデータは，その下の 2 つのスタックである．すなわち，構文解析のタスク間の依存関係は図 3.5 に示すものとなる．

図 3.5 からわかるように，例えばタスク 2 を実行するには，タスク 4 とタスク 5 が終了することが必須であるが，タスク 6 とタスク 7 の状態には一切影響されない．すなわち，図 3.5 の下の方から順に上へ並列に実行するよりも，各タスク間の依存関係のみ記述し，実行可能なタスクを優先させた方が効率的である．これは並列化の fork-join モデルに相当する．Intel TBB を利用すれば，このような需要に対し簡単な記述のみで対応できる．我々は，パーサの並列化において，手動で複数のスレッドを作成するのではなく，リスト 3.16 に示すように，Intel TBB の機能を利用する．

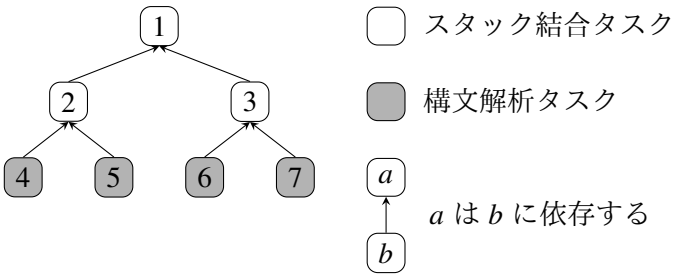


図 3.5: タスク間の依存関係

リスト 3.16: 並列構文解析の実装

```
void parse() {
    // ...
    tbb::parallel_invoke(
        [&]{ leftParser.parse(); },
        [&]{ rightParser.parse(); }
    );
    // ...
    // Combine two stacks
}
```

図 3.5 において，各構文解析タスクは互いに独立である．また，構文解析アルゴリズムから，例えばタスク 2 にはタスク 4 とタスク 5 のスタックが渡されるが，これらのスタックがほかのタスクに使われることはない．すなわち，スタックの実装には排他制御の必要はない．したがって，スタック結合タスクが，渡されたスタックをコピーせず，インプレースに変更するように実装した．

3.4 パーサジェネレータの実装

パーサジェネレータは，処理系本体とは分離されたプログラムである．また，提案するパーサジェネレータは本番の処理の前に一度だけ実行され，実行効率は求められない．そこで，実装の効率を考え，Swift 言語を用い実装した．

ユーザは一つのファイルに字句解析の規則と構文解析の規則を記述する．ジェネレータは，与えられた全ての終端記号と非終端記号に，唯一の整数値を与える．特に，終端記号は 0 から始まる連続した番号が与えられる．非終端記号は，最初のビットが 1 であるような整数値が与えられる．

パーサジェネレータが計算すべき情報は，主に字句解析のための DFA と構文解析のための終端記号間の優先順位である．構文解析の規則は，そのままリスト 3.14 に示した構造として書き出される．

終端記号間の優先順位は，第 1.4 節で説明した方法により計算され，一次元の配列として書き出される．各終端記号の番号は 0 から開始されるため，その値はインデックスとして用い配列をアクセスできる．

`ScannerDFA` の実装は，パーサジェネレータによって生成される．プリスキャンの DFA を構築

表 3.1: 正しく解析されないときのパーサの動作

ステップ	スタック	入力	動作
1	$($, ⊥)	[NULL] $	shift
2	$($, ⊥)([, <)	NULL] $	shift
3	$($, ⊥)([, <)(NULL, <)	] $	reduce
4	$($, ⊥)([, <)(json, ⊥)	] $	shift
5	$($, ⊥)([, <)(json, ⊥)(], =)	$	エラー

したのち，状態遷移関数を二次元の配列として書き出す．配列の一つ目の次元は入力文字，二つ目は状態である．すなわち，各入力文字に対する遷移がメモリ上に連続するように配置している．その理由は以下の 2 点である．

- 入力から文字を 1 つ読んだあと，複数の状態を計算するためのデータがメモリ上で連続する．
- 今後 SIMD 化をするときに，この一続きの遷移をそのまま gather 命令に渡すことができる．

現状，マルチバイトの文字を考慮しておらず，一次元目の大きさは 128 と決められている．

3.5 FNF

ある OPG に，`A : B` のような，右側が 1 つの非終端記号しかない生成規則を**含まない**ときに，その文法が **FNF（Fischer Normal Form）** であると呼ぶ [6]．第 2.2.1 節に述べた構文解析アルゴリズムは，FNF であるような OPG のみ対象とする．以下では，FNF でない OPG を，上記の構文解析アルゴリズムが解析できない理由を，JSON の一部の文法を用いて示す．

JSON の一部の文法をリスト 3.17 に示す．ここで，`NULL` は，`null` という文字列を示すトークンである．

リスト 3.17: 正しく解析されない文法の一例

```
json : NULL;
array : '[' elements ']';
elements : json;
```

この文法に従い，入力 `[ NULL ]` を解析する場合を考える．この場合のパーサの動作を 表 3.1 に示す．

ステップ 4 において，(json, ⊥) はさらに (elements, ⊥) に reduce されるべきであるが，生成規則 `elements : json` の右側に終端記号が存在しないため，パーサは正しく reduce せず，次に shift しようとする．その結果，ステップ 5 において，([, <)(json, ⊥)(], =) を reduce できない，というエラーが生じる．

我々が普段書く文法は大体 FNF ではなく，どこかに `A : B` のような生成規則が存在する．これを解決するために，

1. 非 FNF な文法を，FNF な文法に自動的に変換するツールを作成する．
2. プログラマが，細心の注意を払って FNF な文法を作る．

と 2 つのアプローチが考えられる．

前者を今後の課題とし，今回我々は，後者のアプローチに従い，FNF である文法を作成した．以下に，この問題に対する本研究のワークアラウンドを示す．

例えば，JSON のオブジェクトを定義しようとするとき，普段我々はリスト 3.18 に示す文法を書くと思われる．

リスト 3.18: 変換前の文法（FNF でない）

```
object : '{' '}'
       | '{' members '}' ;
members : members ',' member
        | member
        ;
```

しかし，明らかにこれは FNF ではないので，何らかの修正が必要である．我々は今回，JSON の文法に対する変換のトリックを提示する．このトリックを使えば，簡単に FNF である文法に書き換えることができる．JSON の文法の場合，問題となる生成規則は，再帰的なもので，例えばオブジェクトや配列といった，中に空・一個・複数の要素からなるもののみである．すなわち,「中身が一個」というような生成規則は,「中身が空」と同様に，定義を 1 つ上のレベルに追い出せば良い．

書き換え後の文法をリスト 3.19 に示す．

リスト 3.19: 変換後の JSON の文法（FNF である）

```
object : '{' '}'
       | '{' member '}'
       | '{' members '}' ;
members : members ',' member
        ;
```

ただし，これはあくまで JSON の文法に限った方法であり，任意の文法には適用できない．プログラマはこの問題に遭遇するその都度，このようなワークアラウンドを考えなければならない．そのため，やはり非 FNF の文法を変換するツールが必要と考え，その作成を今後の課題とする．

3.6 スキャナレスな OPG

我々は最初，並列字句解析を提案・実装する前に，字句解析器を使用せず，構文解析器のみで入力ファイルを解析するスキャナレスな方法を検討した．理由を以下に示す．

- 文法が複雑になるが，パーサの全体の実装が簡単になる．

- OPG をスキャナレスにすると，local-parsing の性質を利用して，トークンが分割されることなく，入力を任意の位置で分割して，構文解析を始めることができる．

その結果，OPG の表現力では，字句解析器を使用しない場合，JSON のような簡単なデータ構造にも対応できないという結果に達したことを報告する．

スキャナが存在しないため，プログラマは字句解析のルールをパーサの文法に記述しなければならない．文法をスキャナレスにすると，表現力が低下すると考えられる．なぜならば，もともと字句解析によって解決される曖昧さをパーサが解決する必要があるからである．その方法として，もともと字句解析のルールとして記述されるものを，トークンをそのまま非終端記号に書き換えれば良い．

OPG をスキャナレスにすることの欠点は，字句解析で広く利用される正規表現を自由に文法中に入れることができなくなる点である．例えば，`elements : element ',' elements` について，`','` の後ろに 0 個以上のスペースを入れたいとする．1 個のスペース自体は，`spaces : ' ' | ' ' spaces` と表現できるが，`spaces` をそのまま `','` の後ろに入れることはできない．なぜならば，`spaces elements` と，連続した 2 つの非終端記号が現れ，OPG でなくなるからである．そのため，`spaces` を展開して，`elements` の中に直接記述する方法が考えられるが，0 個以上のスペースを入れる場合，`elements` のルールはリスト 3.20 のようになる．

リスト 3.20: スキャナレスな OPG で記述できない例

```
elements : element ',' elements
         | element ',' ' ' elements
         | element ',' ' ' ' ' elements
         | element ',' ' ' spaces ' ' elements
         ;
spaces : ' ' | ' ' spaces;
```

しかし，`elements :  element ',' ' ' ' ' elements` を考えると，`' '=' '` となるが，`spaces :  ' '  spaces` を考えると，`' '<' '` となり，矛盾である．似た例として，以下のものが考えられる．

- 正規表現の `[\t\n]+` などは，スキャナレスな OPG によって表すことができない．
- 文字列リテラルは，予約語と同時に宣言するとアルファベット間の優先順位に矛盾が生じる．

3.7 今後改良できる点

3.7.1 無駄なプリスキャンの削減

現状の実装は，律儀に各フェーズを実行している．例えば，あるチャンクの開始状態が 1 つに定まるときでも，そのチャンクのプリスキャンは行うようになっている．これは，次のチャンクの

開始状態が 1 つに定まらないときのためである．しかし，このチャンクに関しては，開始状態が 1 つに定まるため，通常の字句解析をすぐに開始可能である．そして，この字句解析が終われば，次のチャンクの開始状態が自然と定まるので，結局このチャンクのプリスキャンは無駄になってしまう．ただし，この最適化をしようとすると，複数のフェーズが混じり，実装がかなり複雑になってくる．

3.7.2 入力データの分割

レンジの大きさによる入力文字の選定は，現状レンジの大きさが 3 以下であるという条件で決めている．例えば，JSON の場合は，レンジの大きさが 1 であるような入力文字が非常に多い．そのため，レンジの大きさが 1 よりも大きいような入力文字で区切るよりも，その先の入力文字を少し読み，レンジの大きさが 1 であるような文字を見つけたほうが，そのチャンクのプリスキャンを速く終える可能性が高い．各入力文字のレンジは，遷移関数を分析すればすぐに得られるが，理想的には，ジェネレータが勝手に各入力文字のレンジの情報から，適切な区切り文字を割り出すべきである．現状では，区切り文字のレンジの大きさの最大値を決める方法に関しては，まだ議論の余地が残っている．

また，レンジの大きさが小さいような入力文字がデータの中にあまり出現しない場合，区切り文字のレンジの大きさをある一定の値以下であると制限してしまうと，入力を分割するところで時間がかかってしまう．入力を分割するところは，現状では実装を簡単にするため逐次にしている．この部分に時間がかかると，全体のボトルネックとなる恐れがある．

そして，論文の執筆時点では，レンジの小さい適切な入力文字をジェネレータが見つけなかった場合でも，正確なエラーメッセージが出せず，ユーザは生成されたソースコードを直接確認しなければならない．仮にその生成されたコードを使用し，処理系をコンパイルした場合，解析器が異常終了する．これは，今後修正される予定である．

3.7.3 SIMD 命令の実装

第 2.1.2 節で紹介したように，可能な状態数がある程度より多い場合，SIMD 命令を使い，複数の状態の計算をまとめて行うことで高速化できる．しかし，実際の規則で検証したところ，多くの状態のレンジの大きさが 2 以下で抑えられるため，この高速化を利用する機会がほぼない．SIMD gather を使用すると，データをコピーするなどのコストがかかり，さらに gather 命令自体も，単純な計算よりもサイクル数がかかるため，かえって性能が落ちる可能性がある．これを踏まえ，本処理系は SIMD gather を使用した並列計算は実装していない．ただし，入力文字のレンジが比較的に大きい場合，SIMD 命令を使うべきである．SIMD gather を使用した並列計算を実装し，かつ入力文字のレンジの大きさによって動的に切り替えるようにすることは改良できる点である．

3.7.4　アクションの実装

本処理系は，現状ではただ入力を構文解析するだけのツールである．言い換えると，入力が与えられた文法に従うかどうかしか判定できない．通常広く使われる処理系は，大抵入力を解析したのち（あるいは解析をしながら）何かアクションを実行するものである．このようなアクションのことは**セマンティックアクション（Semantic Action）**と呼ばれている．例えば，ログ解析のときに，ユーザのログインを記録するログからユーザ名とログイン時刻を抽出してファイルに書き出すなど，通常このようなアクションは出力を伴うものである．

セマンティックアクションは，入力データから情報を取り出す意味で，完全な処理系としては必要なものであり，その実装を今後のタスクとする．

第4章　評価

本章では，実装した処理系を，大規模な JSON のファイルを使い評価した結果を述べる．全ての計測は 5 回繰り返した結果を本文に反映している．本章でのプロット値は，その 5 回の平均値，またエラーバーは，5 回の測定値の標準誤差を示している．

4.1　評価環境

評価に使用した環境を表 4.1 に示す．

また，評価でコア数を変え，プログラムに指定されたコアで実行させるために，`taskset --cpu-list` を使用した．例えば，8 コアでプログラムを実行させる場合は，`taskset --cpu-list 0-7` と指定する．

4.2　JSON を解析する性能

Wikidata は，全ての内容を一つの JSON の配列として含むダンプ形式を提供する[1]．性能評価にその一部を取り出した約 10 GB の JSON ファイルを使用した．フルの JSON ファイルを使わなかった理由は，現状の実装では全てのトークンをいったんメモリに保存することによるメモリ不足である．また，解析に先立ち，入力ファイルの内容を全てメモリに読み込むようにした．

並列字句解析のうち，プリスキャンの性能を図 4.1 に示す．ここで，第 3.1.1 節で説明したベースラインの性能も同グラフに示した．

表 4.1: 評価環境

CPU	Xeon E5-2699 v3 （2.3 GHz）
CPU のコア数	36 (18 cores × 2 sockets)
メモリ	PC4-17000 768GB (384 GB × 2 sockets)
OS	Ubuntu 18.04 64bit
コンパイラ	GCC 7.4.0
最適化オプション	`-O3 -flto -march=native -mtune=native`

[1] `https://dumps.wikimedia.org/wikidatawiki/entities/` より入手できる．2020 年 1 月アクセス．

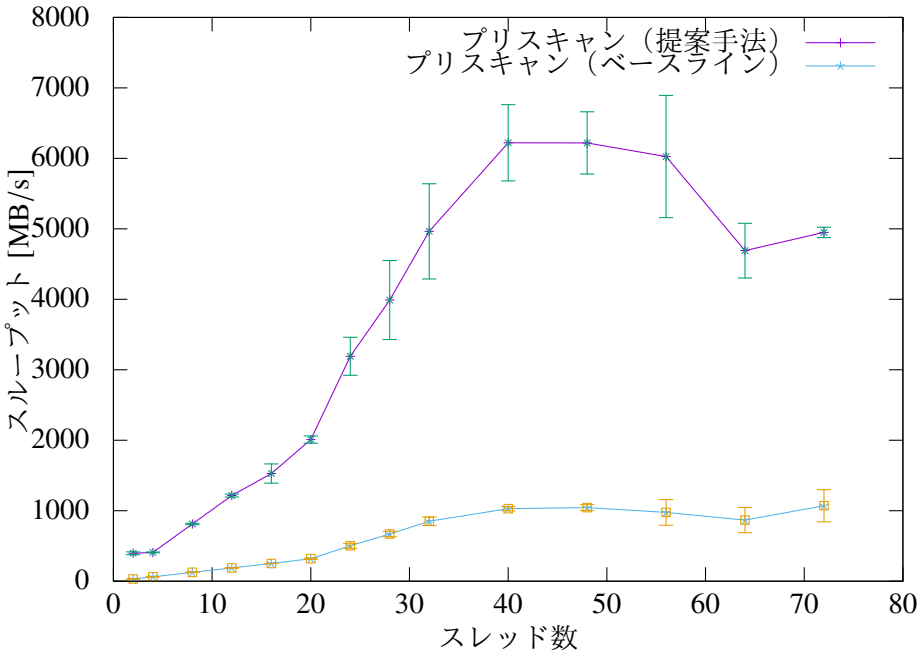


図 4.1: プリスキャンの性能

入力は約 10 MB の複数のチャンクに分けられ，横軸は使用したスレッド数，縦軸はプリスキャンのスループットを示している．プリスキャンは，入力の分割などいくつか逐次の部分を含むが，これらがの処理がプリスキャン全体の処理よりも非常に少ない．したがって，グラフに示したような直線的な性能向上となっており，プリスキャン自体のスループットは，72 スレッドで 10 GB 弱という結果が出ている．グラフからわかるように，入力文字のレンジを考慮した分割をするとき，明らかな性能向上が見られた．この結果は我々の手法が有効であることを示している．

次に，並列構文解析の性能を図 4.2 に示す．

タスクを分割するかどうかを決める閾値は 1000 である．すなわち，与えられたトークン列の長さが 1000 よりも大きければ 2 つに分割し並列に実行する．与えられたトークン列の長さが 1000 以下であればそのトークン列を逐次に構文解析する．様々な実験を重ねた結果，閾値を極端に小さい（例えば閾値を 3 にすると明らかにタスクを分割しすぎである）か大きい（例えば閾値を `SIZE_MAX` にすると逐次の構文解析そのものである）値にしなければ，構文解析の性能に大した変化は見られなかった．

そして，処理系全体の性能は，4 スレッド，16 スレッドと 72 スレッド使用時のスループットを表 4.2 に示す．

第 3.3.2 節でも言及したように，現状では記号を全てメモリに保存した上で構文解析を始めるようになっている．例えば 72 スレッド使用時に，トークン列をメモリ上に展開する場合とトークンを保存しない場合と比べ，処理時間は 20 倍以上異なっている．したがって，現状表 4.2 は処理系

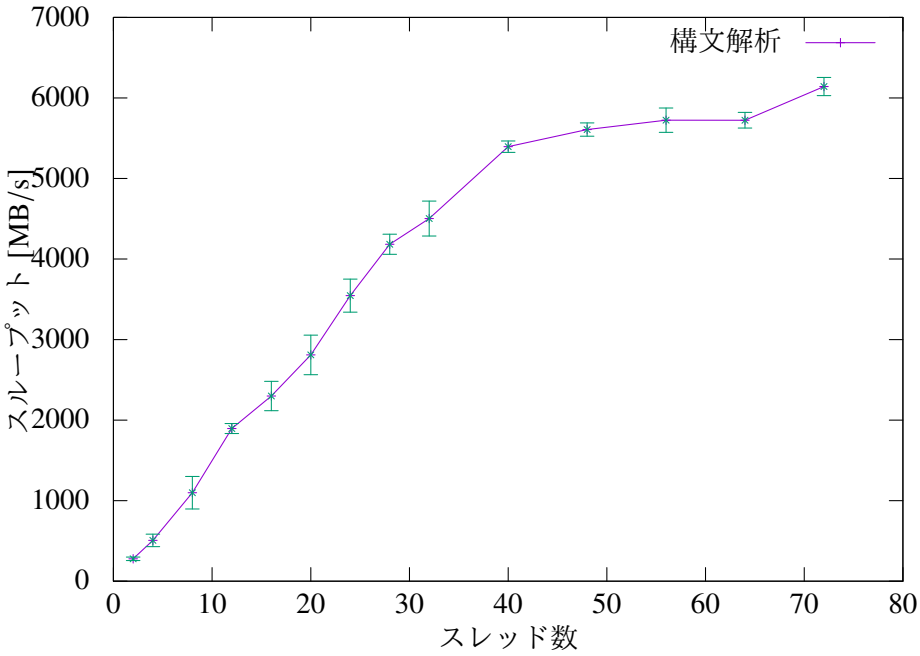


図 4.2: 並列構文解析の性能

表 4.2: 処理系全体のスループット

スレッド数	スループット
4	114 MB/s
16	170 MB/s
72	180 MB/s

の本来出せるはずのスループットとはかけ離れているものである.

第5章　結論

5.1　まとめ

本研究は，大規模な文字列データを，その場限りで文法を定義するだけで高速に解析を行う処理系を提案・実装した．字句解析に関しては，ヒューリスティックに基づき並列化を行い，また構文解析は既存の OPG の並列構文解析手法を利用した．処理系は，複数のスレッドを使用した場合に確かに台数効果が出ており，実データによく適用できるといえる．

5.2　今後の課題

本処理系は，実装がよく最適化されていない．結果として大きな影響をもたらしたのは，トークン列そのままメモリ上に展開しているところである．また，第 3.3 節や第 3.7 節で述べたように，様々なところで一番基礎的な実装をしており，これらの改良を行えば処理系の性能向上につながると考える．

提案した処理系は，OPG の並列構文解析アルゴリズムを使用しているが，構文解析を並列化することにより得られる性能向上は，文法と入力データに大きく依存する．評価では，一般的であると考えられる構造をもつ入力データを使ったが，一部のコーナーケースでは並列構文解析は高速化の効果を発揮しない．極端な例をリスト 5.1 に示す．

リスト 5.1: 並列構文解析による高速化がない例

```
S : S + n | n + n ;
```

これは，単純に数値の足し算を表した文法である．例えば一続きの数値の足し算を入力し，並列構文解析を行った場合，最初のチャンクを除く全てのチャンクは，記号を reduce できずにそのままスタックに保持することがすぐにわかる．その理由を説明すると，`S + n` または `n + n` を reduce できるのは，スタックの先頭に終端記号$が入っているときのみである．すなわち，この文法は最初のチャンク以外では reduce が行われない．そのため，この文法に従った入力に並列構文解析アルゴリズムを適用したところで，逐次の構文解析より良い性能を原理的には出せない．性能とは無関係に，このようなことは**結合性**という問題に一般化され，OPG は，演算子（OPG では全ての終端記号が演算子とみなされる）の結合性を優先順位として取り入れている．

足し算のように，結合性とは無関係に本来は自由に解析しても結果が構わない例が多数存在する．特に大規模なデータは，何かの構造の繰り返しからなり，各部分の関係が薄い場合が多い．OPG を拡張し，構文解析アルゴリズムを改良すれば，並列化したときにより良い性能向上が得られると考えられる．

謝辞

本論文の執筆にあたって，まず指導教員である田浦先生にお礼を申し上げます．先生ご自身が多忙な中，研究の細かいところまで熱心に指導・議論してくださいました．そして，修士2年の一年間，佐藤さんから色々なアドバイスをいただきました．参考になる論文を教えていただいたり，研究の詳細について議論してくださいました．心より感謝を申し上げます．さらに，研究テーマが関連している楽さんにもお礼を言いたいと思います．時間を気にせず，よく研究に関する議論に付き合ってくれました．研究室のほかのメンバーにもお礼を申し上げます．スライドの修正や発表練習にお付き合いいただきました．

この研究は皆様のご支持があってこそできたものです．本当にありがとうございました．

参考文献

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] Nikolas Askitis and Ranjan Sinha. HAT-trie: A cache-conscious trie-based data structure for strings. *Conferences in Research and Practice in Information Technology Series*, Vol. 62, pp. 97–105, 2007.

[3] Alessandro Barenghi, Stefano Crespi Reghizzi, Dino Mandrioli, and Matteo Pradella. Parallel parsing of operator precedence grammars. *Information Processing Letters*, Vol. 113, No. 7, pp. 245–249, 2013.

[4] Alessandro Barenghi, Stefano Crespi Reghizzi, Dino Mandrioli, Federica Panella, and Matteo Pradella. Parallel parsing made practical. *Science of Computer Programming*, Vol. 112, No. P3, pp. 195–226, 2015.

[5] Stefano Crespi Reghizzi, Violetta Lonati, Dino Mandrioli, and Matteo Pradella. Toward a theory of input-driven locally parsable languages. *Theoretical Computer Science*, Vol. 658, pp. 105–121, 2017.

[6] Michael J Fischer. Some properties of precedence languages. In *Proceedings of the first annual ACM symposium on Theory of computing*, pp. 181–190. ACM, 1969.

[7] Robert W Floyd. Syntactic analysis and operator precedence. *Journal of the ACM (JACM)*, Vol. 10, No. 3, pp. 316–333, 1963.

[8] Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time - Functional pearl. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*, pp. 36–47, 2002.

[9] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, Vol. 31, pp. 111–122, 2004.

[10] Steffen Heinz, Justin Zobel, and Hugh E. Williams. Burst tries: A fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, Vol. 20, No. 2, pp. 192–223, 2002.

[11] Peng Jiang and Gagan Agrawal. Combining SIMD and many/multi-core parallelism for finite state machines with enumerative speculation. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, pp. 179–191, 2017.

[12] Jihyun Lee, Yeoul Na, and Seon Wook Kim. Design of HTML parallel parser with semantic-based input splitting. *International Conference on Electronics, Information, and Communications, ICEIC 2016*, pp. 3–6, 2016.

[13] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. Mison: A fast JSON parser for data analytics. *Proceedings of the VLDB Endowment*, Vol. 10, No. 10, pp. 1118–1129, 2017.

[14] Wei Lu, Kenneth Chiu, and Yinfei Pan. A parallel approach to XML parsing. *Proceedings - IEEE/ACM International Workshop on Grid Computing*, pp. 223–230, 2006.

[15] Daniel Luchaup, Randy Smith, Cristian Estan, and Somesh Jha. Speculative parallel pattern matching. *IEEE Transactions on Information Forensics and Security*, Vol. 6, No. 2, pp. 438–451, 2011.

[16] Dino Mandrioli and Matteo Pradella. Generalizing input-driven languages: Theoretical and practical benefits. *Computer Science Review*, Vol. 27, pp. 61–87, 2018.

[17] Todd Mytkowicz. Data-Parallel Finite-State Machines. pp. 529–541, 2014.

[18] Terence Parr and Kathleen Fisher. LL(*): the foundation of the ANTLR parser generator. *ACM SIGPLAN Notices*, Vol. 46, No. 6, p. 425, 2011.

[19] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive LL(*) parsing: The power of dynamic analysis. *ACM SIGPLAN Notices*, Vol. 49, No. 10, pp. 579–598, 2014.

[20] Vern Paxson, et al. Flex–fast lexical analyzer generator. *Lawrence Berkeley Laboratory*, 1995.

[21] Stefano Crespi Reghizzi and Matteo Pradella. Higher-order operator precedence languages. *Electronic Proceedings in Theoretical Computer Science, EPTCS*, Vol. 252, No. Afl, pp. 86–100, 2017.

[22] Ryoma Sin'ya, Kiminori Matsuzaki, and Masataka Sassa. Simultaneous finite automata: An efficient data-parallel model for regular expression matching. *Proceedings of the International Conference on Parallel Processing*, pp. 220–229, 2013.

[23] Zhijia Zhao, Jianhua Sun, Xipeng Shen, Michael Bebenita, and Dave Herman. HPar: A Practical Parallel Parser for HTML'Taming HTML Complexities for Parallel Parsing. *ACM Transactions on Architecture and Code Optimization*, Vol. 10, No. 4, pp. 1–25, 2013.