論　文

FPGAによる構文解析器の合成

マイ　マイクオン† 　　本多峻† 　　倉光君郎†

Mai MAICUONG†, Honda SHUN†, and Kuramitsu KIMIO†

あらまし　解析表現文法 (PEG) は、2004 年に Ford によって提案された形式文法であり、正規表現や文脈自由文法の代替として人気が高まっている。本稿では、より高い性能要求を目指すため、PEG の FPGA 実装、特に PEG 演算子の仮想マシン化によるバーチャルマシン方式について報告し、性能に関する初期レポートを行う予定である。

キーワード

1. まえがき

近年、IoT(Intenet of things) の発展につれ、様々な機器への組み込みやすさやより高度なパケット処理が必要になった。また、ビックデータ時代になった今は、より高速な構文解析技法が求められている。そのため、組み込みやすさと高速な処理の面から、FPGA が注目されている。FPGA を用いて、構文解析器を実現する研究がいくつかある。

例えば、文脈自由文法を使って FPGA 上の構文解析 [**?**] や FPGA 上で正規表現を用いたパターンマッチングマシン [**?**] などの研究がある。

しかし、FPGA で構文解析器を実現するのは非常に手間がかかる。例えば、ソフトウェア開発と異なり、FPGA の回路では、遅延が発生しているため、開発者は常に遅延時間を意識しなければならない。FPGA 開発の際に、論理検証の他、タイミング検証も必要である。また、新しいプロトコルが続々誕生されている今、それに対応するには、しばしば構文解析器を書き換える必要がある。

そのため、本研究の目的は自動生成及び高速処理が可能であり、そして組み込みやすい構文解析器を実現する。構文を定義するために、解析表現文法（Parsing Expression Grammar、以降 PEG）を用いる。PEG は形式言語を一連の規則を使って表したものであり、PEG を用いることにより、事前に字句分析する必要がなく、線形時間で構文解析可能というメリットがある。また、FPGA 構成を記述するために、VHDL（VHSIC Hardware Description Language）という言語を用いる。VHDL とはデジタル回路、特に集積回路を設計するための言語であり、具体的な回路を考慮せずに動作だけを記述すればハードウェアの動作を定義することができるため、ソフトウェア・プログラミングと同様な手法でハードウェアの設計を行うことができる。VHDL は現在標準化され、ハードウェア記述言語の標準的仕様として多く採用されている。

本論文の構成は次の通りである。第２章では、FPGA 及び PEG について述べる。第３章及び第４章は FPGA の設計と実装になる。第６章は性能評価のための実験である。また、第７章及び第８章はそれぞれ関連研究と結論になる。

2. 解析表現文法

構文解析とは、定義された文法に従ってテキストの構造を解析することである。本研究では、構文解析を実現するために、形式文法を使って、ＦＰＧＡ上で構文解析器を生成する。形式文法はたくさんの種類がある。例えば、解析表現文法（PEG）や文脈自由文法（CFG）などがある。

†

本研究では、構文解析を構築するために、解析表現文法を使用する。ＰＥＧは `A <= e` というルールの集合である。解析表現は表 2.1 にある値と演算子を組み合わせた式である。

PEG のメリットとして挙げられるのは書きやすく、強力な表現力をもち、また線形時間で処理することができるところである。

3. 仮想マシン

命令には、基本命令、特化命令と制御用命令がある。基本命令は実際にルールの文字とテキストファイルの文字を比較し、文字を消費するかを判断する。特化命令は基本命令に特殊な処理を加えた命令である。例えば、0 個以上の文字リテラルを処理する命令は RBYTE になる。制御用命令はジャンプ、呼び出し、スタック操作などの命令である。

3.1 文字リテラル、文字クラス及び任意の文字

文字リテラルを処理するための命令は BYTE である。例えば、A = 'a' のルールである場合、 BYTE 'a' という命令が生成される。この命令は、現在評価しているテキストファイルの文字が'a' であれば成功で、そうでなければ失敗となる。同様に、文字クラス、任意の文字を処理するための命令はそれぞれ SET と ANY である。例えば、A = [1-9] の場合、SET [1-9] という命令が生成される。

3.2 非終端記号

非終端記号は CALL 命令を使って、非終端記号の定義されたところにジャンプする。ジャンプする前に、現在命令の次の命令を Return stack に push する。また、RET という命令があるとき、Return stack からジャンプ先を取得する。例えば、以下のＰＥＧファイルの場合、

$$
\begin{aligned}
Add &= Value(+/-)Value \\
Value &= [1-9]
\end{aligned}
$$

生成された命令列は以下となる。

$$
\begin{aligned}
Add\ & L1\ CallL5 \\
& L2\ Set[+/-] \\
& L3\ CallL5 \\
& L4\ Ret \\
Value\ & L5\ Set[1-9] \\
& L6\ Ret
\end{aligned}
$$

まず非終端記号があるため、その非終端記号（L5) へジャンプする。Ｌ 5 にジャンプする前に、Ｌ 2 を Return stack に push される。ここで L5 が成功したら、L6 に進み、L6 は RET であるため、Return stack からジャンプ先を pop される。先ほど L2 を Return stack に push されたため、L2 にジャンプされる。次に L2 が成功したら、L3 に進む。L3 で Value を呼び出し、Ｌ 4 を Return stack に push される。Value の命令が終ったら、Ｌ 4 に戻る。L4 で戻る先がないため、プログラムを終了する。どこから失敗した場合、プログラムが終了される。

3.3 オプション

オプションを処理するため、2 つの方法がある。オプションの対象は上記の基本要素であれば、その命令と組み合わせて特化命令として実行する。例えば、'a'? である場合、OBYTE 'a' というオプションと BYTE を組み合わせた命令が生成される。オプションの対象はグルーピングまたは非終端記号の場合、ALT という命令を使う。PEG ではバックトラックがあり、どこかの命令が失敗しても、前の状態に戻り、処理を続けることがある。ALT は失敗する時の戻り先を保存する Fail stack に push する命令である。どこかが失敗した時、Fail stack からジャンプ先を取得する。
例えば、A = ('a' 'b')?　'c' の場合、'a' を評価する前に、ALT で'c' を評価するための位置を Fail stack に push する。この場合、以下の命令列が生成される。

$$
\begin{aligned}
& L1\ ALTL4 \\
& L2\ BYTE'a' \\
& L3\ BYTE'b' \\
& L4\ BYTE'c'
\end{aligned}
$$

'a', 'b' でマッチした場合、L2,L3,L4 の順に実行される。一方、例えば'a' で失敗した場合、Fail stack からジャンプ先を取得する。先ほど ALT で L4 を Fail stack に push されたため、L4 が pop されて、次に実行する命令が L4 となる。オプションの対象が非終端記号の場合、ALT の後に CALL が実行される。このように、ALT によってオプションを処理することができる。

3.4 0 個 以 上

0 回以上の場合も処理方法が 2 つに分かれる。対象が BYTE, SET, ANY の場合、RBYTE, RSET, RANY という特化命令が実行される。対象はグルーピング、選択又は非終端記号の場合、ALT 及び JUMP を使って処理する。まず、ALT でこの処理が終ったときのジャンプ先を Fail stack に push する。次に、グルーピング、選択、非終端記号の要素を順番に実行し、成功した場合、JUMP でグルーピング、選択、非終端記号の最初の要素に戻る。この処理はどこかが失敗するまで繰り返される。どこかが失敗した場合、Fail stack から、ジャンプ先を取得する。
例えば、

```
Value = '1' ( '+' '2')* '+' '3'
```

というの PEG の場合、以下のような命令列が生成される。

$L1\ BYTE'1'$
$L2\ ALTL6$
$L3\ BYTE'+'$
$L4\ BYTE'2'$
$L5\ JUMPL3$
$L6\ BYTE'+'$
$L7\ BYTE'3'$
$L8\ RET$

まず、ALT でL 6 を Fail stack に push する。次に順番に L3, L4 を実行し、終わったら JUMP 命令で L3 に戻る。 L3, L4 のどこかで失敗するまで、繰り返しが続く。一方、失敗すると、Fail stack に L6 を pop されて、L 6 が実行される。
1 個以上の場合、基本命令と 0 個以上を組み合わせて実行する。例えば、A = 'a'+ の場合は、BYTE 'a' 及び RBYTE 'a' のように命令が生成される。

3.5 選　　択

選択の場合、バックトラックを削減するために、選択肢の最初の文字を評価し、最初にどの選択肢にするかを決める。この処理を実行する命令は FIRST という。複数の選択肢の最初の文字が同じである場合、各選択肢を評価する前に、この選択肢が失敗である場合、別の選択肢に移すためのジャンプ先を ALT 命令で Fail stack に保存する。
例えば、

```
Value = ('1' '1')/('1' '2')/'3'
```

という PEG の場合、以下のような命令が生成される。

$L1\ \ FIRST$
$'1'\ \ -> L3$
$'3'\ \ -> L8$
$default\ -> L2$
$L2\ \ FAIL$
$L3\ \ BYTE'1'$
$L4\ \ ALTL7$
$L5\ \ BYTE'1'$
$L6\ \ RET$
$L7\ \ BYTE'2'jumpL6$
$L8\ \ BYTE'3'jumpL6$

まず、最初の文字によって、jump 先が分かれている。最初の文字が'1' であれば L3 に、'3' であれば L8 に、そうでなければ L2 に移ってＦａｉｌ処理が発生する。L3 にジャンプした場合、まず'1' を消費する。この場合、選択肢が'1' '1' 又は'1'　'2' の２つがあるため、'1' '1' の選択肢を評価する前に、この選択肢が失敗した場合のジャンプ先を ALT 命令で Fail stack に push する。もし、この選択肢が成功した場合、L6 の RET で

プログラムを終了させる。一方、失敗した場合、選択肢'1' '2' を評価し、つまり L7 が実行される。

3.6 否定先読み・肯定先読み

否定先読みの場合、処理方法が 2 つに分かれる。対象が BYTE、SET、ANY の場合、NBYTE, NSET, NANY という特化命令を実行される。対象はグルーピング、選択又は非終端記号の場合、まずグルーピング、選択、非終端記号を評価する前に、失敗した時のジャンプ先を Fail stack に push する。その後、グルーピング、選択、非終端記号の要素を順序に評価する。どこかで失敗が発生したら、ALT で保存したジャンプ先にジャンプする。この場合は次の解析表現に進むことになる。
ただし、グルーピング、選択、非終端記号が成功した場合、つまり否定先読みが失敗した場合、もし Fail 処理を発生させるとしたら、ALT で保存されたジャンプ先に飛ぶことになるため、次の解析表現に進んでしまう。そのため、まず SUCC 命令で ALT で保存したジャンプ先を消してから、Fail 処理を発生させる。
例えば、

$Value$ =!('1'+)[1 − 9]+

という PEG の場合、以下のような命令列が生成される。

$L1$ $ALT L6$
$L2$ $BYTE'1'$
$L3$ $RBYTE'1'$
$L4$ $SUCC$
$L5$ $FAIL$
$L6$ $SET[1-9]$
$L7$ $RSET[1-9]$
$L8$ RET

肯定先読みの場合、対象の表現を順序に評価することになるが、文字を消費しないため、評価する前に、POS 命令で今の文字の位置を保存しておく。評価が終ったら、BACK 命令で、保存された位置に戻る。例えば

```
Value =&('1') [1-9]+
```

という PEG の場合、以下のような命令列が生成される。

$L1$ POS
$L2$ $BYTE'1'$
$L3$ $BACK$
$L4$ $SET[1-9]$
$L5$ $RSET[1-9]$
$L6$ RET

4. 設　　計

FPGA に情報を渡す前に、前処理として PEG ファイルから命令列を生成する。この処理は本研究室が開発している Nez ライブラリを使う。FPGA は命令列を受け取り、テキストを解析する。FPGA の全体回路図は図 4.1 となる。

4.1 制 御 部

基本的に命令実行に 2 つのステップが必要である。ステップ 1 では、命令及び文字を並列に専用のレジスタにロードする。Instruction Address Register(IAR) に保存されたアドレスでメモリをアクセスし、命令データを Instruction Data Register(IDR) にロードされる。同時に Text Address Register(TAR) に保存されたアドレスに格納されたテキストを Text Data Register(TDR) にロードされる (図 4.2)。

次のステップでは、IDR のデータをデコーダに転送され、デコーダによって、実行される命令用回路が決まる。また、命令に応じて必要な情報を実行回路に送り、命令用回路が実行される (図 4.3)。実行結果によって、次の処理が決まる。

制御用命令を除いて命令用回路の出力は next char, next ist, fail がある。next char= '1' の時、文字を消費することになる。next ist = '1' の時、次の命令に進む (図 4.4)。

fail = '1' の時、Fail 処理を発生させる (図 4.5)。Fail 処理では、Fail stack に要素があれば、Fail stack を pop し、ジャンプ先を取得する。Fail stack に要素がなければ、パース失敗となり、プログラムを終了する。

5. 実装と性能評価

6. ま と め

謝辞 本論文は私が横浜国立大学理工学部数物電子情報系学科電子情報システム EP に在籍中の研究成果をまとめたものである。本論文を執筆するにあたり、横浜国立大学工学部電子情報工学科准教授である倉光君郎先生には、指導教官として研究の方針をご指導頂きました。また、FPGA について様々なアドバイスをしていただいた株式会社イーツリーズ・ジャパンの三好健文様、資料作成に付き合っていただいた関口渚、森谷鴻平、山口真弥、本多峻、田村健介、須藤建、千田忠賢先輩及び B4 のみなさんに深く感謝の意を表します。

加えて、私の学生生活を長きにわたって見守って下さった私の家族に深く感謝致します。

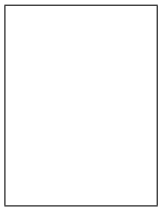
文 献

[1]

付 録

1.

（平成 xx 年 xx 月 xx 日受付）



Abstract

Key words