学位論文最速文法マスター

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF PHYSICS,
ELECTRICAL AND COMPUTER ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF YOKOHAMA NATIONAL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF ENGINEERING

1264904 MAI MAI CUONG
February 2016

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Master of Engineering.

(KURAMITSU Kimio) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Master of Engineering.

(HAMAGAMI Tomoki)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Master of Engineering.

(HOGE Fuga)

Approved for the University Committee on Graduate Studies

Abstract

Acknowledgements

本論文は私が横浜国立大学大学院工学府物理情報工学専攻電気電子ネットワークコース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。ここに深く感謝の意を表します。

(stab)

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。

Contents

List of Figures

Chapter 1

はじめに

近年、ＩｏＴ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。そこで、組み込みやすさと高速な処理の面から、FPGA() というデバイスが注目されている。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、研究１では正規表現を使って、ＦＰＧＡ上で構文解析器を作る。正規表現からＦＰＧＡ合成に必要なＨＤＬファイル、ネットリスト又はコンフィグレーションデータに変換する。しかし、パーサごとに回路を合成しなければならず、手間がかかる。また、開発者にはある程度ＦＰＧＡの知識が必要である。また、研究２で文脈自由文法を使ってＦＰＧＡでパーサを実現する研究がある。こちらはサブ回路を作ってたくさん並ぶ。ただし、n 個のサブ回路が n+1 文字しか受理できず、性能的にはよいとは言えない。

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いしきしなければならない。また、FPGA 開発の際には、合理合成、配線などという段階あり、検証時間が大きい。さらに、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、構文解析器を書き換える必要がある。

そこで、本研究の目的は自動生成可能、高速な処理、そして組み込みやすいパーサを実現する。方法としては、解析表現文法から FPGA 上でパーサが自動生成する。ユーザはＦＰＧＡの知識が必要なく、文法を定義する peg ファイルとテキストを入力するだけで、結果が得られる。

本論文の構成は次の通りである。

第１章：初めに第２章：ＦＰＧＡ及び解析表現文法ＰＥＧ第３章：実装方針第４章：デザイン第５章：実装第６章：実験第７章：関連研究第８章：結論

Chapter 2

関連知識

この章では、ＦＰＧＡ及びＰＥＧについて説明する。

2.1 FPGA

FPGA とは、ユーザー側で回路を変更することができるデバイスのことである。FPGA（Field Programmable Gate Array）とは、デジタル回路を構築する大規模集積回路の一種である。通常の集積回路の場合、出荷された回路を変更することができないが、FPGA の場合、出荷された回路も使用者側で任意の論理機能を実装できる。FPGA は並列度の高い電子回路や機能の専門化によって機能向上した電子回路を容易に構成できる。ＦＰＧＡは並列処理に擦れており、並列化によって高速化することが可能である。また、ユーザ側が自由に回路を変更することが可能であるため、ある問題に特化した専用回路を作り、性能を向上することができる。

さらに、稼働中の機器のハードウェアも変更できるため、処理の対象や内容に合わせて、状況に応じた回路に書き換えることで、最小限の回路で、多様な処理を実行できる。

一般にＦＰＧＡ開発の際、ハードウェア記述言語（ＨＤＬ）が用いられる。本研究では、ＨＤＬの一種であるＶＨＤＬを使用する。

2.2 PEG

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上でパーサを自動生成する。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、パーサを構築するために、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（）にある値と演算子を組み合わせた式である。

ＰＥＧのメリットは書きやすく、強力な表現力をもっている。また、Ｐａｃｔｒａｃｋ を使って線形時間で処理することができる。

Chapter 3

実装方針

本研究では、Vitual Machine を採用している。PEG ファイルから、実行すべき命令列を生成する。命令セットは以下となる。命令では、文字の操作の命令と制御用の命令がある。

文字操作の命令は BYTE, SET, ANY がある。こちらの命令は実際に文法定義ファイルの文字とテキストからの文字を比較し、文字を消費するかを判断する。BYTE は文字リテラルを処理するための命令である。例えば、A = 'a' のＰＥＧ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の文字が'a' であれば成功でそうでなければ失敗である。SET は文字クラスを処理するための命令である。例えば、A = [1-9] の場合、[SET [1-9]] という命令が生成される。ANY は任意の文字を処理するための命令である。例えば、A = . の場合、[ANY] という命令がある。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。RET という命令の時、Return stack からジャンプ先を取得する。例えば、以下のＰＥＧファイルはこのような命令列が生成される。Add = Value (+/-) Value Value = [1-9]

生成された命令列 : Add L1 Call L5 L2 Set [+/-] L3 Call L5 L4 Ret Value L5 Set[1-9] L6 Ret

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。Ｌ５にジャンプする前に、Ｌ２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、Ｌ４を Return stack に push される。Value の命令が終ったら、Ｌ４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。

ＡＬＴはジャンプ先を Fail stack に push する命令である。PEG では、Option, R , 選択という演算子があり、どこかの命令が失敗しても、処理が続けることがある。どこかが失敗した時、Fail stack からジャンプ先を取得する。ＡＬＴの使い道はグルーピング、非終端記号、選択を処理する時に使う。具体的に下記に述べる。

Option を処理するため、2 つの方法がある。対象となる Option は上記の基本要素であれば、その命令と組み合わせて一つの命令として実行する。例えば、'a'?である場合、[OBYTE 'a'] という Option とＢＹＴＥを組み合わせた命令が生成される。対象となる Option はグルーピングまたは非終端記号の場合、ＡＬＴという命令を使って飛び先を stack に push する。例えば、A = ('a' 'b')?　'c' の場合、'a' を実行する前に、ALT で'c ' を評価するための位置を stack に push する。この場合、以下の命令列が生成される。

L1 ALT L4 L2 BYTE 'a' L3 BYTE 'b' L4 BYTE 'c'

'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 が発生し、stack から飛び先を取得する。先 ALT で L4 を 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。Ｏｐｔｉｏｎ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 Option を処理することができる。

０回以上の場合も処理が２つに分かれる。対象がＢＹＴＥ，ＳＥＴ，ＡＮＹの場合、RBYTE, RSET, RANY として組み合わせた命令を実行する。対象はグルーピング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・非終端記号の要素を順番に実行し、成功したら、ＪＵＭＰでグルーピング・非終端記号の最初の要素に戻る。この処理をＦａｉｌが発生するまで、繰り返される。Ｆａｉｌが発生したら、先ＡＬＴで push した Fail stack から、ジャンプ先を取得する。例えば、以下のＰＥＧはこのような命令列が生成される。Value = '1' ('+' '2')* '+' '3'

L1 BYTE '1' L2 ALT L6 L3 BYTE '+' L4 BYTE '2' L5 JUMP L3 L6 BYTE '+' L7 BYTE '3' L8 RET

まず、ＡＬＴでＬ６を Fail stack に push する。つぎに順番に l 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、Ｌ６が実行される。１回以上の場合、基本命令と０回以上にみなして実行する。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。

Choise の場合、backtrack を削減するために、選択肢の最初の文字によって、最初にどの選択にするかを決める。この処理をする命令は 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 stack に保存する。例えば、以下のＰＥＧはこのような命令が生成される。Value = ('1' '1')/('1' '2')/'3'

L1 FIRST '1' -¿ L3 '3' -¿ L8 default -¿ L2 L2 FAIL L3 BYTE '1' L4 ALT L7 L5 BYTE '1' L6 RET L7 BYTE '2' jump L6 L8 BYTE '3' jump L6

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Ｆａｉ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。

否定先読みの場合、処理が２つに分かれる。対象がＢＹＴＥ，ＳＥＴ，ＡＮＹの場合、N BYTE, N SET, N ANY として組み合わせた命令を実行する。対象はグルーピング又は非終端記号の場合、まずグルーピング・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・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・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、以下のＰＥＧはこのような命令列が生成される。Value = !('1'+) [1-9]+ 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。例えば以下のＰＥＧはこ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

シーケンスは連続な命令で実行される。

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。

Chapter 4

設計

まず、各命令に対応するのサブ回路を作る。制御部の信号によってどのサブ回路が実行されるのか決められる。

4.1 命令用回路

4.1.1 文字操作命令

文字操作命令はBYTE, SET, ANYがある。

BYTE命令は、テキストの文字と形式文法の文字が一致するかどうかを判断する。ＢＹＴＥ命令用回路の入力はTRG, PEGからの文字、テキストの文字がある。一致する場合、match信号が'1'となり、逆に一致しない場合はfail信号が'1'となる。Trgがある場合だけ、作動する。Ｔｒｇがないときはmatch及びfailが'0'となる。matchが'1'になった場合、次の命令に進む処理をする。failが'1'の場合、Alt命令で保存されたスタックからとび先をもらう。スタックにデータが存在しない場合、プログラムを終了し、パース失敗となる。

ANY命令は、１文字を消費するという命令である。ＡＮＹ命令用の回路の入力はTRGで、出力はnextchar信号である。TRGがある場合だけ、nextcharが'1'となり、文字を消費する処理を始める。

SET命令用回路の入力は、TRG, PEGからの２の文字、テキステキステキストの文字がある。文字の範囲か、文字の選択かによって、比較条件が異なる。マッチした

場合、match が'1' となり、マッチしない場合、fail が'1' となる。

4.1.2 組み合わせ命令

組み合わせ命令は Option, O 個以上、否定先読みを BYTE, SET, ANY を組み合わせた命令である。

Option の場合、0 個マッチしてもよいため、TRG がある場合、match が'1' となり、この回路から fail 信号が発生されない。しかし、文字を消費するかを決めるため、１個マッチした場合、nextchar が'1' となり、文字を消費する。

0 個以上の場合、マッチしたら、nextchar 信号が'1' となり、文字を消費する。また、nextchar 信号が TRG とつながっており、マッチした場合、命令をもう一度実行する（図）。ただし、新たな文字をロードするには、１クロックが必要であるため、nextchar を１クロック遅らせてＴＲＧに入る。このように、マッチしなくなるまで、テキストを消費しながら、回路を動作する。一方、マッチされなくなったら、nextcmd 信号が'1' になり、次の命令に進む。

否定先読みの場合、条件を逆にするだけである。

4.1.3 その他の命令

CALL 命令用回路は、ＴＲＧがあるとき、まず Return stack に命令がもっている保存用の命令アドレスを追加し、Return stack の先頭をインクリメントする。同時に命令が持っているジャンプ用の命令アドレスをプログラムレジスタにロードする。

ＡＬＴ命令用回路は、ＴＲＧがあるとき、Fail stack に命令が持っている保存用アドレスを追加し、Fail stack の先頭をインクリメントする。

SUCC 命令用回路は、ＴＲＧがあるとき、Fail stack から先頭の命令アドレスを消し、Fail stack の先頭をデクリメントする。

FAIL 命令用回路は、ＴＲＧがあるとき、fail を'1' となり、Fail 処理を発生させる。

4.2 制御部

4.3 制御部

実効ステップは以下となる。ステップ１:PR の値が MAR に転送ステップ２：メモリからデータをＩＲに転送ステップ３：命令デコード及ぶ命令実行

ステップ３が終ったら、またステップ１に戻

Chapter 5

実装

Chapter 6

実験

Chapter 7

関連研究

Chapter 8

結論

Bibliography

[1] KonohaScript: A Static Scripting Language. http://konohascript.org.

[2] K. Tim and W. Rob. The goal structuring notation-a safety argument notation. In *Proc. of Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004.