B. LRパーザジェネレータの実装

大堀先生の解説記事の概要

文脈自由文法と構文解析の目的

この課題で行う構文解析の対象は、以下で定義されるような文法Gで定義された言語である。

$$G = (N, T, P, S)$$

ここで、Nは非終端記号の集合、Tは終端記号の集合、Pは生成規則の集合、Sは開始記号である。

この文法Gを用いて定義される言語を$L(G)$とすれば、これは以下のように表せる。

$$L(G) = \{w | S \overset{*}{\Longrightarrow} w\}$$

ただし、$S \overset{*}{\Longrightarrow} w$は、$S$に対して生成規則を0回以上適用すると$w$が得られることを意味する。

非終端記号と終端記号が混ざった記号列α_iに対して、生成規則をちょうど1回適用するとα_{i+1}が得られることを$\alpha_i \Rightarrow \alpha_{i+1}$と表せば、$S \overset{*}{\Longrightarrow} w$は以下のように表せる。

$$S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n \Rightarrow w \qquad \cdots (1)$$

構文解析の目的は、wが与えられたときに、(1)のような記号列の書き換えの系列を求めることである。(1)が求まれば、開始記号にどの生成規則をどの順番で適用したかが明らかになり、これはwの文法構造が明らかになることを意味する。

最右導出

(1)に現れる$\alpha_i (1 \leq i \leq n)$および$w$を展開することを考える。そのために、3つの記号を定義する。

A_jはwをSに還元するとき、j番目のステップで得られた非終端記号とする。(還元が導出の逆順になっていることに注意、すなわち先ほどのiに対し$j = n + 1 - i$である)

β_jは終端記号と非終端記号が混在した記号列(ただしβ_1は終端記号のみからなる文字列)とする。w_jは終端記号のみからなる文字列とする。

このとき、(1)は

$$
\begin{aligned}
S &\Rightarrow \beta_n A_n w_n (= \alpha_1) \\
&\Rightarrow \beta_{n-1} A_{n-1} w_{n-1} w_n (= \alpha_2) \\
&\vdots \\
&\Rightarrow \beta_1 A_1 w_2 \cdots w_n (= \alpha_n) \\
&\Rightarrow w_1 w_2 \cdots w_n (= w)
\end{aligned}
$$

と書き直すことができる。

w_jが終端記号であることに注意すると、導出の各過程において記号列に含まれる非終端記号のうち最も右の非終端記号A_jが置き換えられていることがわかる。このような導出を特に最右導出といい、記号 $\underset{rm}{\Longrightarrow}$ で表す。また、最右導出を0回以上行うことを $\overset{*}{\underset{rm}{\Longrightarrow}}$で表す。

LR構文解析の基本原理

ある文脈自由文法Gによって定まる記号列の集合

$$
C_G = \{\alpha\beta | S \overset{*}{\underset{rm}{\Longrightarrow}} \alpha A w \underset{rm}{\Longrightarrow} \alpha\beta w\}
$$

を考える。このとき、次の**LR構文解析の基本原理**が成り立つ。

C_Gは正規言語である。

正規言語とは、正規表現で記述できる言語であり、非決定性有限オートマトン(NFA)または決定性有限オートマトン(DFA)で受理できる。これは、Gが与えられたら、C_Gを受理するNFAのN_GやDFAのD_Gを構成できることを意味する。N_Gは、サブセット構成と呼ばれる手続きによりD_Gに変換できることが知られている。そのため、この基本原理を証明するには、N_Gが受理する言語$L(N_G)$がC_Gに等しいこと、すなわち$L(N_G) \subseteq C_G$かつ$C_G \subseteq L(N_G)$であることを示せば良い。

オートマトンとACTION表

解説記事の第4節では、基本原理の証明の過程でN_Gの構成方法が与えられている。N_Gは状態の有限集合Q, 入力記号の集合Σ、状態遷移関数δ、初期状態s、受理状態を表すQの部分集合Fからなる。Qの各状態は、Gの生成規則に、パーザが入力記号列において読み込んだ位置を示す·を導入したもの、あるいは初期状態である。形式的に表すと、

$$Q = \{s\} \cup \{[A \to \alpha \cdot \beta] | A \to \alpha\beta \in P\}$$

となる。さらに、文字列終了を表す記号$ を導入すると、解説記事の第5節にあるような決定性オートマトンD_Gが得られる。

得られたオートマトンD_Gの状態遷移関数をもとに、ACTION表を作成することができる。ACTION表の各エントリには、以下の4つの動作のいずれかが記述される。このACTION表は、文法Gのパーザの動作を表現している。

- shift(q) : D_Gの 状態を記録するスタックにqをプッシュし、入力記号を1文字読み進める。
- reduce($A \to \beta$) : スタックを$|\beta| - 1$文字ポップし、記号列をAに還元する。
- accept : 構文解析が成功したことを通知し、終了する。
- error : 構文エラーを通知し、終了する。

LR(0)文法Gで定められるDFAの各状態qは、「もし還元項を含むなら、その状態はその還元項1つのみからなる」という性質を満たすので、ACTION表の1つのエントリにはシフトと還元の両方の動作が記述されることはない。yaccのようなパーザジェネレータを用いてパーザを生成するときにshift/reduce conflictやreduce/reduce conflictが発生することがあるが、これは1つのエントリにshiftとreduceが両方記載されている、あるいはreduceが2つ以上記載されていることを表している。このようなconflictが発生するとき、パーザジェネレータに入力した文法はLR(0)文法ではない。

解説記事と実装との対応

※ 風邪をひいてしまい、パーザジェネレータを構成するC++ファイルのうち、backend.cppとcodegenerator.cppの実装はできませんでした。そのため、以下で述べるバックエンドとコード生成の部分に関しては、実装の大まかな方針を示します。

作成したプログラムの概要

パーザジェネレータは、文法を記述したファイルを入力として受け取り、LR(0)パーザのソースコードを出力する。パーザジェネレータは、大きく分けて以下の3つの部分からなる。

- フロントエンド : 入力された文法をパーザジェネレータ内部のデータ構造で表現する
- バックエンド : パーザジェネレータ内部のデータ構造で表現されたデータ構造から、構文解析表(ACTION表とGOTO表)を作成する
- コード生成 : 作成した構文解析表から、パーザのソースコードを(ターゲット言語、すなわちパーザの実装言語で)出力する

入力される文法が言語G(実際にはBNFやPEG)で書かれ、出力されるパーザのソースコードが言語P(実際にはC言語やPython)で書かれるとすれば、パーザジェネレータは言語Gから言語Pへのコンパイラである、とみなすことができる。

この課題ではパーザジェネレータをC++で実装することを試みているが、パーザジェネレータの実装言語と入力する文法を表現する言語、出力するパーザのソースコードの言語が同一である必要はない。

パーザジェネレータは、以下の5つのファイルからなる。

- main.cpp プログラム全体を実行するmain関数が記述されたファイル
- frontend.cpp パーザジェネレータに入力されたテキストファイルを読み込んで、プログラム内部のデータ構造に変換する処理が記述されたファイル
- backend.cpp プログラム内部のデータ構造で表現された文法をもとに、ACTION表を作成する処理が記述されたファイル (8/25時点で未実装)
- codegenerator.cpp バックエンドで作成したACTION表をもとに、パーザのソースコードを出力する処理が記述されたファイル (8/25時点で未実装)
- variable.h 複数の.cppファイルで使われるグローバル変数を宣言したファイル

また、パーザジェネレータに入力する文法を記述したファイルとして、sample1.inputを用意している。これはテキストファイルであり、yacc文法ファイルに似た形式で書かれている。

解説記事とフロントエンドの対応

解説記事では、文法を$G = (N, T, P, S)$で定めている。この表記は、非終端記号の集合N、終端記号の集合T、生成規則の集合P、開始記号Sの区別を明確にしている。

一方、入力するファイルsample1.inputでは、これら4つをセクションごとに分けて明示していない。この入力ファイルは、以下のような構成を持つ。

- セクション1 (1st section) : 非終端記号とトークン(字句解析を済ませた後の終端記号)を宣言
- セクション2 (2nd section) : 生成規則を記述
- セクション3 (3rd section) : サブルーチンを記述

このsample1.inputでは、解説記事の第3節の例にある文法G_{PAREN}を実装している。

$$G_{PAREN} = (\{S, A\}, \{<, >\}, P, S)$$
$$P = \{S \to AA, A \to <>, A \to < A >\}$$

非終端記号の集合Nは、セクション1で `%nset` の後にスペース区切りで記述している。

```
%nset S A
```

終端記号の集合Tは、文字リテラルおよびトークンからなるものとする。文字リテラルはシングルクォーテーションで囲まれた文字であり、トークンは(ここでは)字句解析を通過した後の字句とする。

トークンは、もし存在するならば `%token` の後に記述されるが、G_{PAREN}では終端記号が `<` `>` の2つだけなので、今回は実装を見送った。

文字リテラルの終端記号 `<` および `>` は、セクション2で生成規則の中に現れている。文字リテラルはそれぞれの生成規則から容易に読み取れるので、セクション1で明示的に書く必要はない。

生成規則の集合Pはセクション2で記述されている。

```
S: A A
A: '<' '>'
A: '<' A '>'
```

それぞれの生成規則について、 `:` の左側(左辺)には非終端記号が記述され、 `:` の右側(右辺)には左辺から導出される記号列が書かれている。最初の生成規則の左辺は開始記号Sとみなされるので、左辺に開始記号を持たない規則は2番目以降に書かなければならない。

上記のsample1.inputを処理するフロントエンドのプログラムfrontend.cppは、非終端記号を `std::vector <std::string> non_terminal_vec` に格納する。また、生成規則を読み取る中で得られた開始記号を `std::string starting_symbol` に格納する。非終端記号は、パーザジェネレータ内部では `std::string` として扱われる。

終端記号は、トークンを含む場合はそのトークンをパーザジェネレータ内部で保持しなければならない。しかし、今回扱うG_{PAREN}にはトークンがなく文字リテラルのみであるから、非終端記号と異なり明示的には管理していない。

生成規則は、 `std::map <std::string, std::vector<std::vector<std::string>>> production_rule_map` という、左辺の非終端記号をキー、右辺の生成される記号列を値とする連想配列に格納する。ここで、 `std::vector<std::string>>` は生成規則1行分に相当する右辺の記号列である。G_{PAREN}では非終端記号Aから$<>$と$< A >$が導出されるので、

```
production_rule_map["A"][0] = ("'<'", "'>'");
production_rule_map["A"][1] = ("'<'", "A", "'>'");
```

となる。

バックエンドおよびコード生成部(8/25時点で未実装)の実装の方針

解説記事では、オートマトンを作ってから構文解析表(ACTION表)を作っている。
バックエンドの実装にあたっては、

- 読み取った生成規則に、パーザが読み込んだ位置を示す記号·および文字列終了を表す記号$を導入して、状態の集合と状態遷移関数を作成する (これによりオートマトンができる)
- オートマトンからACTION表を作成する

ことが必要となる。各々の状態は、生成規則と同様に `std::vector<std::string>>` で表現し、その集合は `std::vector<std::vector<std::string>>>` で表現できる、と考えられる。ACTION表の各エントリは、動作を表す1文字(shiftのs, reduceのr, acceptのa, errorのe)と、状態もしくは生成規則を表す番号のペア `std::pair<char, int>` とすればよい。

コード生成部の実装にあたっては、

- パーザのソースコードにスタックを導入する
- ACTION表に記述された動作を行うソースコードの出力を作る

ことが必要となる。スタックには、状態を表す番号を格納する。パーザのソースコードがC++で書かれるとすれば、 `std::stack state_stack;` といった文字列を出力するコードをパーザジェネレータ内に書かなければならない。また、ACTION表に記述された動作をパーザのソースコードで実装するには、条件分岐を用いれば良い。例えば、状態1で記号Aを読んだら入力を1文字読み進め、状態4に遷移するというshift(4)に対応したソースコードとしては、

```
if(state == 1 && input_x[0] == 'A') {
    state_stack.push(1);
    state = 4;
    input_x = increment(input_x);
}
```

などのような書き方が考えられる。コード生成部は、一般的なコンパイラの出力部分を参考にしながら実装すればよい。

参考文献

LALR parser generatorの作り方 2024年8月25日参照
プログラミング言語処理 2024年8月25日参照
Reading 18: Parser Generators 2024年8月25日参照
Parser Generator - an overview | ScienceDirect Topics 2024年8月25日参照

『コンパイラ [第2版] ー原理・技法・ツールー』A.V.エイホ、M.S.ラム、R.セシィ、J.D.ウルマン著、原田賢一訳、サイエンス社、2009、pp.260-277