Yet Another Parser Generator – ta zajímavá dokumentace

Teorie

Nejprve si popíšeme, jak vypadá tzv. **LR(0) parser**. Parser čte vstup po jednom symbolu a jakmile suffix symbolů, které už načetl, odpovídá pravé straně nějakého pravidla, redukuje tyto symboly na symbol na levé straně pravidla. Parser má konečný počet „stavů“, které reprezentují různé zajímavé situace, ve kterých se může ocitnout. Parser tyto „stavy“ kupí na zásobník a tak se sám může vyskytovat v nekonečně mnoha různých konfiguracích, což je označení, které budeme používat pro dvojici tvořenou zásobníkem stavů a vstupem, který se má ještě zpracovat. Na stavy, které parser pokládá na zásobník, se dá nahlížet jako na množiny tzv. itemů. Itemem budeme rozumět nějaké přepisovací pravidlo gramatiky, ve kterém je na nějaké pozici mezi symboly či na kraji vložen ukazatel, tečka. Tato tečka označuje, co ještě parser čeká, že načte, pokud by chtěl redukovat symboly tímto pravidlem. Na začátku má parser na zásobníku pouze počáteční stav. Počáteční stav obsahuje item za každé přepisovací pravidlo, na jehož levé straně je počáteční symbol gramatiky. Všechny tyto itemy mají tečku hned na začátku; cíl parseru bude tedy nějak ze vstupu vydobýt symboly, které tvoří pravou stranu některého z těchto pravidel. Tyto itemy však tvoří pouze jádro počátečního stavu, každý stav musí být totiž tvořen uzavřenou množinou itemů. Množina itemů M je uzavřená, právě když platí

$$\forall\ A \to \alpha . B \beta \in M\ , \forall\ B \to \gamma \in P : B \to . \gamma \in M$$

kde ***α***, ***β*** i ***γ*** jsou posloupnosti symbolů a ***P*** je množina pravidel gramatiky. Pokud si tedy parser dělá v nějakém stavu zálusk na nějaký neterminál, podívá se na pravidla, kterými ho může přepsat, a začne hledat symboly z jejich pravé strany.

Stavy automatu a jejich konstrukce se nám bude představovat líp, když si je umístíme do grafu. Za každý symbol ***X***, který se někde v množině itemů stavu ***p*** objevuje napravo od tečky, povede ze stavu ***p*** hrana označená symbolem ***X*** do dalšího stavu. Jádro tohoto nového stavu bude tvořeno všemi itemy stavu ***p***, kde se symbol ***X*** objevoval napravo od tečky, jenom tečka už bude posunuta za ***X***. Stav bude pochopitelně navíc obsahovat další prvky, které ho uzavřou.

Tímto se nám už nabízí způsob, kterým tyto stavy zkonstruovat a to je prostý průchod tímto grafem. Stavy jsou s množinami itemů isomorfní a tak pokud povede hrana do nějakého nového stavu tvořeného množinou itemů, kterou jsem už potkali, nevytvoříme stav nový, ale napojíme hranu do stavu již existujícího. Tak máme zaručen i konečný počet stavů, za předpokladu, že vstupní gramatika je konečná.

Parser pracuje tak, že si naloží na zásobník počáteční stav a začíná následovně pracovat:

Pokud z vrchního stavu vede hrana označená terminálním symbolem, který je právě na vstupu, parser jej načte a posune se po hraně do nového stavu (přihodí stav na zásobník).

Pokud ve vrchním stavu existuje item, který má tečku už na konci, parser podle jeho přepisovacího pravidla zredukuje symboly, které už načetl. To znamená, že se posune o ***k*** stavů zpět (odstraní ze zásobníku ***k*** stavů) a z nově vrchního stavu se posune po hraně označené symbolem ***A***, kde ***A*** je symbol na levé straně onoho přepisovacího pravidla a ***k*** je počet symbolů na jeho pravé straně. Hrana označená symbolem ***A*** v takovém stavu musí vždy existovat; jedinou výjimkou je počáteční stav, kde se itemy s přepisovacími pravidly pro počáteční symbol objevily z naší vůle. Pokud však ale parser v tuto chvíli zároveň dočetl celý vstup, ukončí parsování a ohlásí úspěšný výsledek, v opačném případě zahlásí chybu.

Z výše popsaného je vidět, že jednání parseru nemusí být vždy deterministické. Někdy bude možné redukovat podle více pravidel najednou (tzv. **reduce**/**reduce** konflikt) a jindy zase bude možné jak redukovat, tak číst (**shift**/**reduce** konflikt). Gramatiky, jejichž LR(0) parsery jsou deterministické, nejsou ale příliš použitelné a tak se budeme snažit konstruovat **SLR(1)** a **LALR(1)** parsery. Nadále se budu zaměřovat na konstrukci LALR(1) parserů, jelikož jsou silnější než SLR(1) parsery (množina gramatik s deterministickými SLR(1) parsery je vlastní podmnožinou těch s

deterministickými LALR(1) parsery).

Naším cílem bude pro každý finální item $A \rightarrow \omega.$ v nějakém stavu q spočítat množinu terminálů, při jejichž spatření na začátku zbývajícího vstupu můžeme ve stavu q podle pravidla $A \rightarrow \omega$ redukovat. Mějmě konfigurace, do kterých se dostaneme z konfigurací s vrchním stavem q redukcí podle $A \rightarrow \omega$ a které vedou k úspěšnému doparsování. Množinu přípustných terminálů pravidla $A \rightarrow \omega$ ve stavu q budou tvořit symboly, které se v těchto konfiguracích vyskytují na začátku vstupu. Těmto množinám budeme říkat **lookahead množiny**.

Povšimneme si, že při redukci podle $A \rightarrow \omega$ se vrátíme o $|\omega|$ (počet symbolů v řetězci ω) stavů a poté přejdeme přes hranu označenou symbolem A. Pro hrany označené neterminálním symbolem si tedy definujeme tzv. **Follow** množiny. **Follow** množina hrany ze stavu p označené symbolem A je definována jako množina symbolů, které se mohou objevit na začátku vstupu v konfiguracích, které měly vrchní stav p a přesunuli se po hraně označené symbolem A. Dá se pak jednoduše dokázat, že **lookahead množina** nějakého itemu $A \rightarrow \omega.$ ve stavu q je jen sjednocením **Follow** množin hran vedoucích ze stavů p označených symbolem A, kde stavy p myslíme stavy, ze kterých existuje cesta do q označená symboly ve ω.

Teď si všimneme zajímavého vztahu mezi **Follow** množinami neterminálních hran. Nechť existuje hrana ze stavu p do stavu r označená neterminálem A a stav r obsahuje item tvaru $B \rightarrow \alpha A.\gamma$, kde γ je nulovatelný řetězec (řetězec, který jde přepsat na prázdný řetězec). Mějme dále hrany označené symbolem B, které vedou ze stavů p', ze kterých existuje cesta do p pod řetězcem α. Follow množina hrany z p do r pak obsahuje všechny prvky z Follow množin takových hran. Důvod je takový, že pokud přejdeme z p do r pod symbolem A, můžeme potom bez načtení dalšího symbolu zredukovat z ničeho řetězec γ a poté redukovat podle pravidla $B \rightarrow \alpha A \gamma$, čímž se vrátíme do nějakého stavu p' a z něj se vydáme po hraně označené symbolem B. Tudíž jakýkoliv symbol, který by byl v tu chvíli přípustný, bude přípustný i po přejití přes hranu ze stavu p do stavu r. Tuto relaci mezi hranou z p do r a hranou z p' pod symbolem B budeme nazývat **includes**.

Druhým klíčovým pozorováním, nebo spíš definicí, je to, že **Read** množina nějaké neterminální hrany je podmnožinou její **Follow** množiny, kde **Read** množinou hrany ze stavu p označené symbolem A definujeme jako množinu symbolů, které lze načíst před tím, než bude redukován nějaký řetězec obsahující A. Poté můžeme dokázat i silnější tvrzení, které nám dá jednoduchý způsob, jak **Follow** množiny spočítat. **Follow** množina nějaké hrany je totiž sjednocení **Read** množin hran, které jsou z této hrany dosažitelné po relaci **includes**.

Mezi **Read** množinami je podobný vztah jako mezi **Follow** množinami. **Read** množina hrany ze stavu do p do stavu p' obsahuje **Read** množiny hran ze stavu p' označených nějakým nulovatelným symbolem. Této relaci budeme říkat relace **reads**. Další analogickou definicí budou takzvané **DR** (**Direct Read**) množiny. **DR** množina nějaké neterminální hrany ze stavu p do stavu p' je množina terminálních symbolů, které se objevují na hranách vedoucích ze stavu p'. **DR** množiny jsou samozřejmě podmnožinou **Read** množin. Můžem pak dokázat i to, že **Read** množiny nejsou ničím jiným než sjednocením **DR** množin hran dosažitelných po relaci **reads**.

LALR(1) parser je poté definován a tvořen stejně jako LR(0) parser, jenom redukci podle nějakého finálního itemu budeme připouštět pouze v případě, že další symbol na vstupu leží v **lookahead množině** onoho itemu, čímž se dramaticky sníží počet konfliktů v gramatice.

Hezčí definice využívající bohatou notaci a obrázky, případně vynechané důkazy, se dají najít v článku DeRemera a Pennella [1] na stranách 619-622 a 645-647. Článek je volně dostupný na internetu.

Implementace

Parsery zkonstruované naším programem budou reprezentovány dvojící tabulek. Parsovací tabulka se indexuje prvně vrchním stavem a druhak terminálním symbolem, který nás čeká na vstupu, a hodnotami jsou akce, které by měl parser v dané konfiguraci vykonat. V jedné buňce takové tabulky může být například akce **shift** s číslem nějakého stavu, což značí, že parser má terminál ze vstupu načíst a přihodit řečený stav na zásobník. Dále může v buňce parsovací tabulky

sedět akce **reduce** a číslo nějakého přepisovacího pravidla, což pro parser znamená to, že má ze zásobníku odhodit tolik stavů, kolik je symbolů na pravé straně onoho přepisovacího pravidla a z nově vzniklé konfigurace se posunout na další stav podle goto tabulky. Goto tabulka pro každý stav a právě vzniklý neterminál určuje číslo stavu, který se má přihodit na zásobník. Další přípustnou hodnotou v parsovací tabulce je **fail**, což značí chybu ve vstupu; zadaný text není v popsaném jazyce.

Náš parser si krom těchto dvou tabulek musí pamatovat i jména symbolů, abychom mohli hlásit smysluplné chyby, a podobu přepisovacích pravidel (minimálně neterminál na levé straně a počet symbolů na pravé straně).

Definici gramatiky, včetně specifikace lexeru a parseru, přečteme a rozparsujeme pomocí parseru vygenerovaného naším nástrojem. Ke gramatice si přidáme terminální symbol ***$end***, který značí konec vstupu, a neterminál ***$start***, který bude skutečným počátečním symbolem gramatiky. Do gramatiky zamícháme i přepisovací pravidlo **<*$start*> ::= <*start*> *$end***, kde **<start>** je uživatelský počáteční symbol, pro který už může existovat vícero přepisovacích pravidel. Důvod, pro toto rozšíření, je přidání symbolu ***$end***, který nám umožní se chovat ke konci vstupu konzistentně, stejně jako k jakémukoliv jinému symbolu před námi (bude moct náležet do lookahead množin a bude mít hezky vlastní sloupek v parsovací tabulce). Symboly jsou spojitě očíslovány s tím, že terminály mají nižší čísla než neterminály, ***$end*** má číslo 0 a ***$start*** má číslo rovné počtu terminálů (čili se jedná o neterminál s nejnižším číslem). Nejdůležitější krok zpracování gramatiky je metoda ComputeTables třídy GrammarProcessor, která je zodpovědná právě za konstrukci stavů, **lookahead množin** a tabulek parseru.

Lookahead množiny spočítáme jednoduše sjednocením relevantních **Follow** množin. **Follow** množiny spočítáme, stejně jako **Read** množiny, algoritmem Digraph, který jsem prezentoval v zápočtové práci k Algoritmům a datovým strukturám I, kde jsem i dokazoval jeho správnost. Je to algoritmus na hledání silně souvislých komponent grafu, dá se však ale jednoduše upravit, aby hledal reflexivní tranzitivní uzávěr našich relací.

Relace **reads** i **includes** jsou reprezentovány pomocí orákula, které je dopočítává za běhu, jelikož po žádné se neprojde více než jednou a nenapadá mě nijak efektivnější způsob předčasného výpočtu.

ComputeTables nejprve vytvoří LR(0) parser průchodem grafu stavů, poté zjistí stavy, ve kterých stavech se vyskytují konflikty a začne v nich dopočítávat **lookahead množiny**. Pokud si uživatel nevynucuje **LALR(1) lookahead množiny**, tak zkusíme nejdříve spočítat **SLR(1) lookahead množiny**. Ty se dají vypočítat také algoritmem Digraph spuštěným na grafu, kde vrcholy jsou neterminály a hrany jsou podobné relaci **includes** (dá se to říct i tak, že existuje hrana z ***A*** do ***B***, pokud platí relace **includes** mezi nějakou hranou označenou symbolem ***A*** a nějakou hranou označenou symbolem ***B***; viz. strana 636 v [1]). Počáteční množiny pro algoritmus Digraph na výpočet **SLR(1) lookahead množin** jsou sjednocení **Read** množin hran označených nějakým stejným neterminálem. Pokud SLR(1) nevyřeší všechny konflikty, spočítáme pro zbylé sporné stavy **LALR(1) lookahead množiny**.

Pokud v parseru zůstanou **shift**/**reduce** konflikty, vyřešíme je tak, že dáme přednost akci **shift**. Důvod, proč tak činíme, je čistě pragmatický; řešíme tak např. konflikt „plandajícího else“, kde pokud si vybereme nějaké pravidlo, podle kterého bychom ho chtěli řešit, měli bychom docela problém napsat jednoznačnou a hlavně užitečnou gramatiku. Proto jsem se inspiroval u tvůrců bisona a **shift**/**reduce** konflikty jsou pro můj generátor závdavek akorát pro warning. ComputeTables je schopná vygenerovat i HTML log grafu stavů, ve kterém si může uživatel procházet výsledný parser i s **lookahead množinami** a zkoumat konflikty. U každého konfliktu je navíc přítomno ospravedlnění, proč se daný konfliktní symbol nachází ve sporné **lookahead množině**. Tento nápad jsem přejal ze systému SAIDE [2].

Posledně se zastavím ještě nad lexerem. Uživatel pro něj může popisovat podobu terminálních symbolů regulárními výrazy a původně jsem i zvažoval, že napíši vlastní automat na regulární výrazy, ale pak jsem si to naštěstí vymluvil. Uvědomil jsem si, že to, co bych chtěl provozovat, by byl automat, který by zkoušel paralelně všechny výrazy, které si uživatel zadal.

Když jsem tuto ideu přeložil zpět do jazyka regulární výrazů, napadlo mne vytvořit regulární výraz, ve kterém by byly všechny uživatelovy výrazy odděleny ořítkem. Lexer tedy funguje tak, že si nechá z-JIT-kompilovat takovýhle regulární výraz od .NETího Regex enginu a spouští ho na vstupní řetězec za konec posledního nalezeného tokenu. Jednotlivé, ořítky oddělené, výrazy tvoří pojmenované skupiny a díky elegantnímu návrhu .NETích tříd na práci s regulárními výrazy není těžké po úspěšném namatchování celého řetězce zjistit, která skupina se namatchovala.

Reference

[1] DeRemer Frank and Pennello Thomas: Efficient Computation of LALR(1) Look-Ahead Sets; ACM Transactions on Programming Languages and Systems, Vol. 4, No. 4, October 1982, Pages 615-649.
http://www.win.tue.nl/~wsinswan/softwaretools/material/DeRemer_Pennello.pdf

[2] Passos Leonardo Teixeira, Bigonha Mariza A. S. and Bigonha Roberto S. : A Methodology for Removing LALR(k) Conflicts