Yet Another Parser Generator
Informace pro uživatele

Režim make

V tomto režimu může uživatel předložit programu soubor s definicí nějaké gramatiky a nechat si ji zkompilovat. Výsledek se uloží do souboru, který je pak k dispozici režimu **run**.

yapg make *inputFile* [**-o** *outputFile*] [**-l** *logFile*] [**-f**] [**-c** *compilerFlags*]

Yapg načte gramatiku z *inputFile*, zkompiluje ji a uloží výstup do *outputFile* (pokud nebyl *outputFile* určen, data se zapíší do *inputFile.par*). Pokud nebude určen *logFile*, program v případě existence LALR(1) konfliktů v parseru vypíše do souboru *inputFile.html* všechny stavy automatu a odůvodnění nalezených konfliktů. Pokud bude *logFile* určen explicitně, bude do něj program zapisovat podobu automatu i v případě úspěšného vytvoření parseru. Uživatel může také nastavením optionu **-f** vynutit výpočet LALR(1) lookahead množin pro všechny konfliktní itemy parseru, tj. i pro ty, jejichž konflikty by bylo možno vyřešit s pomocí SLR(1) lookahead množin. Obecně se tak prodlouží výpočet programu, výstupní parser by však měl přesněji popisovat chyby ve zpracovávaném vstupu (konkrétně řečeno bude schopen přesněji určit množinu očekávaných symbolů).

Jak má vypadat specifikace gramatiky předkládaná režimu make?

Na začátku většiny specifikací je nepovinná hlavička. Ta je uvozená klíčovým slovem

%header

a skládá se z prostého C# kódu. Většinou to budou na začátku tedy using statementy a pak nějaké namespace bloky s pomocnými třídami.

Za hlavičkou se nachází povinná část obsahující data pro lexer (popis terminálních symbolů). Celá sekce začíná následujícím řádkem:

%lexer

Definice terminálních symbolů vypadá následovně

identifikátor=*regulární_výraz*

Tímto se zadaný regulární výraz přiřadí k pojmenovanému terminálu. Každý terminál k sobě může mít přiřazen více než jeden regulární výraz. Na pořadí regulárních výrazu záleží, jelikož lexer bude hledat výše popsané regulární výrazy dříve a vrátí token s terminálem, kterému přísluší nejvyšší regulární výraz, jež bylo možno z dané pozice najít. Regulární výrazy smí používat veškeré vlastnosti, o kterých se píše v .NETí dokumentaci, jen by se měly vyvarovat používání pojmenované skupiny, jejichž jména jsou tvaru *__i*, kde *i* je celé číslo, neměly by používat shodně pojmenované skupiny ve dvou různých regulárních výrazech a zároveň se nemůžou spolehnout na číslování skupin.

Před tento seznam tokenů může uživatel umístit jméno terminálního symbolu, který v případě, že bude lexerem nalezen, nebude předložen parseru, ale bude zahozen. Takovýto odpadový

terminál si může uživatel zadefinovat na začátku specifikace následujícím způsobem

%null *identifikátor*

Takhle se dají např. jednoduše zahazovat komentáře a/nebo whitespace.

Lexer scanuje tokeny pomocí .NETího Regex enginu, u kterého se dá nastavit několik různých režimů (multiline, ignorecase..., viz. .NETí dokumentace). Tyto optiony může uživatel nastavit globálně pro celý lexer použitím regexí konstrukce **(?***optiony***)**, která je zdokumentována a popsána v .NETí dokumentaci v sekci o Miscellaneous constructs. Tato část je stejně jako definice odpadového terminálu nepovinná a nachází se mezi jménem odpadového terminálu a seznamem tokenů.

Nyní už se můžeme pustit do „neterminální“ úrovně jazyka, která začíná klíčovým řádkem

%parser

První povinnou součástí neterminální úrovně je definice počátečního symbolu, která vypadá následovně.

%start *identifikátor*

Parser se pak bude snažit najít složkový stromek, jež bude mít tento neterminální symbol v kořeni. Než se pustíme do přepisovacích pravidel, následuje ještě jedna nepovinná část a to je určení typu stavového objektu.

%userobject *jménoTypu*

_state je parametr dostupný v kódu všech přepisovacích pravidel, který se dá použít např. pro uchovávání nějakého stavu, který se mění při parsování vstupu.

Zbytek gramatiky tvoří přepisovací pravidla a typy pro neterminály. Přepisovací pravidla musí být zapsány ve standardní notaci BNF, za pravou stranou bude navíc blok C# kódu uzavřený ve složených závorkách, který popisuje jak spočítat hodnotu v neterminálu . Prázdný řetězec na pravé straně se nijak explicitně neznačí, pouze se pravá strana rovnou ukončí blokem kódu anebo ořítkem. Jména symbolů mohou sestávat z tzv. slovních znaků (\w neboli a-zA-Z0-9). Gramatika může obsahovat jednořádkové komentáře, které vždy začínají znakem **#**.

Uvnitř bloků kódu se lze odvolávat na následující hodnoty:

- _1, _2, ... - hodnota i-tého symbolu z pravé strany pravidla; u tokenů se jedná o řetězec, který jim ve vstupu odpovídá; u neterminálů se pak jedná o hodnotu vrácenou z bloku kódu, který bul spuštěn při tvorbě tohoto neterminálu (standardně typu object, pokud uživatel neurčí typ sám, viz. níže)
- _line1, _column1, _line2, _column2, ... - čísla řádků a sloupců, na kterých se vyskytují symboly z pravé strany
- _state - pomocný objekt (má vždy hodnotu null, pokud je parser spouštěn z příkazové řádky přes Yapg; pokud je parser spuštěn z kódu (Parser.Parse), pak je v proměnné _state hodnota druhého parametru metody Parse)

Krom přepisovacích pravidel můžou být v poslední části typy pro hodnoty neterminálů. Proměnné _1, _2... mají implicitně typ object, tímto způsobem je možné slíbit, že z daného neterminálu budeme vracet jen jeden konkrétní typ hodnoty. Tento typ můžeme následujícím způsobem nějakému neterminálu přiřadit

%type *neterminál jménoTypu*

Pokud by mělo *jménoTypu* obsahovat znaky < nebo >, je třeba jej uzavřít do dvojitých uvozovek.

Ukázka:

```
%header
using System;

%lexer
%null WHITE
(?i)
NUMBER=\d+(,\d+)?
PLUS=\+
MINUS=\-
MULTIPLY=\*
DIVIDE=/
LPAREN=\(
RPAREN=\)
WHITE=\s+

%parser
%start <expr>

%type <expr> double
%type <term> double
%type <factor> double

<expr> ::= <term>       { return _1; }
        | <expr> PLUS <term> { return _1 + _3; }
        | <expr> MINUS <term> { return _1 - _3; }

<term> ::= <factor>     { return _1; }
        | <term> MULTIPLY <factor> { return _1 * _3; }
        | <term> DIVIDE <factor> { return _1 / _3; }

<factor> ::= NUMBER { return Convert.ToDouble(_1); }
          | LPAREN <expr> RPAREN { return _2; }
```

Režim run

V tomto režimu si může nechat uživatel rozparsovat text v jazyce popsaném nějakou zkompilovanou gramatikou.

`yapg run` *`parserFile`*

Program načte z *`parserFile`* data lexeru a parseru v binární podobě (vytvořená předem pomocí režimu make anebo uživatelským programem využívajícím **yapg** jako knihovnu) a inicializuje lexer a parser. Program pak očekává na standardním vstupu text, který bude parsovat. Hodnotu kořenového neterminálu pak program vypíše na výstup.