Dokumentace úlohy DKA: Determinizace konečného automatu v Python3 do IPP 2011/2012
Jméno a příjmení: Jan Hrivnák
Login: xhrivn01

Dokumentace k projektu IPP 2011/2012, DKA

Analýza zadání

Cílem tohoto projektu bylo vytvořit skript na zpracování a determinizaci konečného automatu. Výsledný automat musí ekvivalentní (přijímající stejný jazyk) s původně zadaným.

Postup řešení

Nejdříve skript provádí kontrolu zadaných parametrů. K tomu se využívá knihovna *getopt*. Nastavují se zde proměnné, podle zadaných parametrů a ihned se také kontroluje správnost zadaných parametrů, jejich povolené kombinace a také, zda některý parametr nebyl zadán vícekrát. V případě zadání parametru *-i*, se již během načítání dat veškeré znaky převádí na malé.

Dále se zpracuje zadaný vstupní soubor a celý automat se uloží do paměti. Této části se budu ještě věnovat později. Některé syntaktické chyby jsou hlášeny ihned a některé jsou, dle jejich principu, prováděny až po načtení celého vstupního souboru, kdy má skript všechny potřebné údaje pro kontrolu těchto chyb.

Pokud není zadán parametr *-e* ani *-d* (nebo jejich slovní, delší verze), tak se pouze tiskne výstupní soubor v požadovaném formátu. K tomuto účelu jsem si vytvořil funkci *tiskni()*, která dostává potřebné objekty ve kterých je automat uložen. I přes původní plán napsat projekt čistě pomocí metod OOP, jsem zde několikrát zvolil obyčejnou funkci. Jsou ale logicky využity především v místech, kde by se opakoval stejný kód, který se ale nevztahuje k žádnému jednomu konkrétnímu objektu, aby mohl být zařazen jako metoda některé třídy.

V případě zadání parametru *-e* nebo *-d* se dále provádí výpočet epsilon uzávěru a odstranění epsilon pravidel. V případě, že je parametrem požadována i determinizace automatu se provede i tato činnost. Poté se automat vytiskne.

Nyní se na některé části své implementace podívám více z blízka.

Zpracování vstupního souboru

Pro načtení vstupního souboru jsem si implementoval vlastní lexikální analýzu. Tuto třídu jsem vytvořil jako typ iterátor, který při každém vrací jeden nalezený a zpracovaný token. Samotná implementace je založena na načtení celého vstupu do proměnné, kde se poté v cyklu načítání iteruje pouze zvyšováním jedné integerové hodnoty (pozice znaku ve vstupu). Toto řešení jsem zvolil vzhledem k poměrně přesně dané struktuře vstupního souboru, kdy nejsou příliš pravděpodobné rozsáhlé soubory, pro které by bylo lepší pracovat v otevřeném souboru a nikoliv v takto načteném celém souboru.

Díky poměrně přesnému formátu vstupu má konečný automat, který využívá lexikální analýza, pouze 8 stavů a je tak stále poměrně přehledný. Spolu s označením načteného tokenu se vrací i jeho hodnota.

Uložení dat v paměti

Pro uložení stavů automatu a vstupní abecedy jsem implementoval třídu *množina()*, která obsahuje pouze jeden seznam, ve kterém jsou uložena data. Dále jsou zde implementovány metody na přidání prvku do seznamu (s ošetřením vkládání duplicit), a také metodu pro vypsání seznamu. Ta byla hojně využívána především při testování.

Jednou z nejtěžších částí celého skriptu bylo vymyšlení metody pro uložení přechodů. Nakonec jsem zvolil formu, kde se jednotlivé stavy ukládají jako jména do slovníku. Jako výsledek každého zadaného jména (identifikátoru stavu) se předává další vnořený slovník. Ten je rozdělen podle vstupních znaků, která mají být v daném místě přečteny. Tento slovník již vrací pouze obyčejný zanořený seznam, který obsahuje cílové stavy, do kterých se z tohoto stavu s tímto symbolem, dá dostat (na začátku je automat nedeterministický, tudíž zde může být více stavů a je potřeba toho seznamu).

Jasnější to snad bude po ukázce:

Vstupní přechody: s a->p, s a->q, s b->q, p a->p, p a->q, p b->q
Uložení v paměti: {s:{a:[p, q], b:[q]}, p:{a:[p, q], b:[q]}}

I přes nepříliš elegantní implementaci, má toto řešení hlavní výhodu v tom, že pro zjištění cílového stavu například ze stavu S pomocí symbolu 'b', můžeme použít zápis: *prechod[S][b]* a dostaneme požadovaný seznam možných cílových stavů.

Implementačně jsem to vyřešil jako novou třídu trida_prechodu(), která kromě uložení výše popsaných dat, poskytuje metody pro přidání přechodu, spočítání epsilon uzávěru uloženého automatu, odstranění epsilon pravidel, a také samotnou determinizaci. Jako další jsou zde implementovány některé metody pro výpis částí automatu (pro testovací účely), a tak také vypsání samotného automatu do normální formy, které se volá při finálním zápisu na výstup.

Determinizace

Tato část skriptu je provedena jako 3x zanořený for cyklus, který postupně prochází všechny vygenerované stavy (negeneruje tedy nedostupné). Dále pro každý znak vstupní abecedy zjistí možné cílové stavy tohoto přechodu pro všechny stavy, ze kterých se sám skládá (např. pro stav S1_S2 a znak A, se zde vyhledají všechna pravidla S1 A-> a S2 A->). Z těchto zjištěných možných cílových stavů se po seřazení vygeneruje nový stav, který se uloží pro další procházení.

Domnívám se, že efektivnější řešení než 3 vnořené cykly se zde použít nedá. Je totiž nutné projít všechny vygenerované (tedy dostupné) stavy, dále je potřeba projít všechny znaky ze vstupní abecedy pro daný stav, a poté je potřeba projít všechny možné cílové stavy v tomto přechodu (stále je zde ještě logicky zadán nedeterministicky automat).

Výpočet epsilon uzávěru nějakého stavu (používá se ještě před determinizací), je založen na stejném principu zanořených cyklů, neboť také musí procházet řadu stavů a v nich hledat „cíl“ pro každý vstupní symbol z abecedy.

Doplňující informace k řešení

Několik dalších věcí, které by měli pomoci k lepšímu pochopení mé implementace.

Pro přehlednost je zdrojový soubor rozdělen do tří souborů. Hlavní je dka.py obsahující samotný skript. Jeden pomocný soubor funkce.py obsahuje vyextrahované funkce, které by se jinak museli v kódu zbytečně kopírovat. Poslední soubor tridy.py obsahuje třídu iterátoru pro načtení vstupního souboru, třídu pro uložení a práci s přechody automatu a jednoduchou třídu pro uložení množin a práci s nimi.

Vzhledem ke snaze vracení uživatelsky přívětivých chybových hlášek, jsem ustoupil od tvoření vlastní funkce pro tisk chyb. Chyby se tak hlásí přímo v místě jejich vzniku, kde se tiskne chybová hláška, a poté se skript korektně ukončí s příslušným návratovým kódem.

Závěrem

Vzhledem k prvnímu setkání s jazykem Python3 jsem několikrát měnil vývojové prostředí. Projekt jsem nejdříve vyvíjel přímo na referenčním serveru merlin. Zhruba v polovině vývoje, jsem ale přešel na vývojové prostředí PyCharm a mj. jsem tak začal využívat interpret jazyka i na systému Windows. Na serveru merlin poté již docházelo pouze občas ke kontrolám kompatibility a úplně nakonec, po otestování vlastními testy, i ke spuštění dodaných oficiálních testů.

Projekt byl poměrně náročný na zamyšlení se nad vlastním fungování skriptu, ale to mě na tom bavilo. Naštěstí na rozdíl od prvního skriptu v tomto předmětu, zde byly velmi přesně definované požadavky na formát vstupu/výstupu (ve smyslu, že nebylo potřeba testovat příliš mnoho kombinací, které by mohly nastat) a nebylo tak potřeba příliš dlouho ladit program pro testovací skripty.