VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

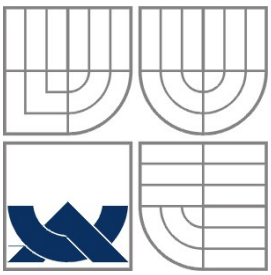
PŘEKLADAČ JAZYKA STAVOVÉHO DIAGRAMU DO JAZYKA CHILL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

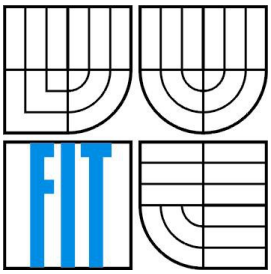
AUTOR PRÁCE
AUTHOR

ZUZANA GOLDMANNOVÁ

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

PŘEKLADAČ JAZYKA STAVOVÉHO DIAGRAMU DO JAZYKA CHILL

COMPILER OF STATE DIAGRAM LANGUAGE TO CHILL LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR
ZUZANA GOLDMANNOVÁ

VEDOUCÍ PRÁCE
SUPERVISOR
Ing. ONDŘEJ RYŠAVÝ, Ph.D.

BRNO 2015

Abstrakt

Tato práce se zabývá gramatikou existujícího proprietárního jazyka stavových diagramů, který se používá k modelaci stavových automatů použitých v pobočkových ústřednách. Cílem práce byla analýza gramatiky a stávajícího překladače z tohoto jazyka do jazyka CHILL a návrh a implementace překladače nového. Tento překladač bude lépe udržovatelný a odstraňovat chyby a nedostatky stávajícího řešení. Závěrem je tento překladač otestován za použití jak již existujících souborů, které byly dříve vytvořeny pro stávající překladač, tak souborů, které jsem vytvořila pro testovací účely.

Abstract

This thesis deals with the grammar of an existing proprietary state diagram language, which is used to simulate state diagrams employed in telephone exchanges. The goal of this work is to analyze the grammar and existing compiler of this language to CHILL language and also to design and implement a new compiler. This new compiler will be easier to manage and will eliminate errors and imperfections of the current arrangement. Finally this compiler is tested using existing files, that were formerly created for the current compiler, and also using files I created for testing purposes.

Klíčová slova

Překladač, CHILL, DASTEP, DAPAS, lexikální analýza, syntaktická analýza, gramatika, token, Backus-Naurova forma, abstraktní syntaktický strom, PLY

Keywords

Compiler, CHILL, DASTEP, DAPAS, lexical analysis, semantic analysis, grammar, token, Backus-Naur form, abstract syntax tree, PLY

Citace

Zuzana Goldmannová: Překladač jazyka stavového diagramu do jazyka CHILL, bakalářská práce, Brno, FIT VUT v Brně, 2015

Překladač jazyka stavového diagramu do jazyka CHILL

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením pana Ing. Ondřeje Ryšavého, Ph. D.
Další informace mi poskytl konzultant z firmy iXperta s.r.o., pan Ing. Petr Jelen.
Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Zuzana Goldmannová
20. května 2015

Poděkování

Chtěla bych poděkovat vedoucímu mé bakalářské práce, panu Ing. Ondřeji Ryšavému, Ph. D., a také konzultantovi z firmy iXperta s.r.o., panu Ing. Petru Jelenovi, za ochotu ke konzultacím a cenné rady.

Obsah

Obsah....1
1 Úvod....3
2 Význam některých názvů a pojmů používaných v této práci....4
2.1 DASTEP....4
2.2 DAPAS....4
2.3 CHILL....4
2.4 SD soubor....4
3 Současný překladač a motivy pro vývoj nového....5
3.1 Motivy pro vývoj nového překladače....5
3.2 Struktura a další problémy současného překladače....5
4 Návrh nového překladače....7
4.1 Implementační jazyk....7
4.2 Pomocný nástroj pro lexikální a syntaktickou analýzu....7
4.3 Použitá verze jazyka Python....7
4.4 Struktura nového překladače....8
5 Gramatika jazyka DAPAS....9
5.1 Některé konstrukce jazyka DAPAS a jejich převod do BNF....9
5.2 Klíčová slova v jazyce DAPAS....10
5.3 Vznik nového pravidla „brief_explanation“....11
5.4 Pravidlo pro název události....12
5.5 Pravidla pro komentáře....13
5.6 Zrušená pravidla....13
6 Lexikální analýza....14
6.1 Moduly ply.lex a dastep_lex....14
6.2 Chyby, které mohou nastat během lexikální analýzy....15
7 Syntaktická analýza....16
7.1 Moduly ply.yacc a dastep_yacc....16
7.2 Třída Node....16
7.3 Chyby, které mohou nastat během syntaktické analýzy....16
8 Sémantická analýza a generování výsledného kódu jazyka CHILL....18
8.1 Sémantická analýza a tabulka symbolů....18
8.2 Generování kódu pro konektory....18
9 Práce s chybami a varováními....20
9.1 V rámci lexikální analýzy....20
9.2 V rámci syntaktické analýzy....20
10 Spouštění programu, jeho parametry a návratové kódy....21
10.1 Parametry programu....21
10.2 Nový parametr -debug....22
10.3 Návratové kódy programu....23
11 Mód zpětné kompatibility....24
12 Možná vylepšení....25
12.1 Makra....25

12.2 Chybové hlášky....25
13 Testování....26
14 Závěr....28
Literatura....29
Seznam příloh....30
Příloha A....31
Příloha B....34

1 Úvod

Překlad mezi proprietárním jazykem stavových diagramů DAPAS a jazykem CHILL je v zadavatelské firmě součástí systému vývoje vestavěných systémů pro telefonní ústředny. Překladač, který k tomuto účelu momentálně používá, je program starý více než dvacet let a jeho spolehlivost a udržovatelnost za to dobu klesla natolik, že se firma rozhodla pro jeho nahrazení.

Zadáním a i cílem této bakalářské práce byl návrh a realizace nového překladače, který by implementoval jeho základní funkcionalitu, odstraňoval problémy a nedostatky s ním spojené a zároveň byl snadno udržovatelný.

Součástí práce je i analýza gramatiky jazyka DAPAS, vytvoření jejího formálního zápisu a popis postupu při návrhu a následné implementaci nového řešení překladače.

Vysvětlení názvů DAPAS, CHILL a dalších pojmů specifických pro telekomunikace a zadavatelskou firmu iXperta s.r.o. se věnuji v kapitole 2. Vzhledem k absenci psaných pravidel pro převod mezi jazyky DAPAS a CHILL byla analýza fungování současného překladače velmi důležitou součástí mé práce, proto jej v kapitole 3 krátce zmíním spolu s problémy s ním spojenými. Kapitola 4 popisuje některé kroky a rozhodnutí při návrhu nového překladače a kapitola 5 je věnována gramatice jazyka DAPAS. Kapitoly 6, 7 a 8 provázejí jednotlivými fázemi procesu překladu tak, jak je implementuje nový překladač. Nalezení a výpis chyb jsou důležitou částí každého překladače a v této práci je jim věnována kapitola 9. Kapitola 10 obsahuje informace o spouštění překladače a jeho parametrech a návratových hodnotách. V kapitole 11 představím systém, díky kterému jsou soubory napsané pro stávající verzi překladače přeložitelné i překladačem novým, navzdory drobným úpravám v gramatice. Kapitola 12 obsahuje návrhy na možná vylepšení a nové funkce programu a kapitola 13 obsahuje informace o způsobu s výsledcích jeho testování.

2 Význam některých názvů a pojmů používaných v této práci

V této bakalářské práci budu používat názvy a pojmy, které jsou specifické pro telekomunikace nebo pro firmu iXperta s.r.o. (původním Unify, ale v průběhu psaní práce došlo k přejmenování), pro kterou jsem překladač vyvíjela. Účelem této kapitoly je vysvětlení a krátký popis těch nejdůležitějších užívaných pojmů.

2.1 DASTEP

DASTEP je program používaný ve firmě iXperta s.r.o., který funguje jako přemostění mezi etapou návrhu a etapou implementace programů v jazyce CHILL. Takovéto programy se navrhují za použití stavových diagramů, které se pak převedou do jazyka CHILL a v této formě se dále využívají.

DASTEP očekává na vstupu soubor SD, který obsahuje daný návrh ve formě jazyka DAPAS a automaticky z něj vytvoří zdrojový kód v jazyce CHILL. Takto vytvořený zdrojový kód se pak používá stejně, jako kdyby byl vytvořen „ručně“. Jde tedy o překladač mezi jazyky DAPAS a CHILL. [1]

2.2 DAPAS

DAPAS je jazyk, který by se dal popsat jako jazyk stavových diagramů a stejně jako program DASTEP byl vytvořen ve firmě iXperta s.r.o. pro interní potřeby. Díky tomuto jazyku je možné stavový diagram navrhovaného systému popsat textovou formou, jako program. Každý symbol z grafické reprezentace diagramu je ekvivalentní příkazu v jazyce DAPAS. [2]

2.3 CHILL

CHILL je jazyk, který se používá pro programování vestavěných systémů užívaných v telefonních ústřednách. Jde o silně typovaný, blokově strukturovaný a objektově orientovaný jazyk. CHILL je standardizovaný dle standardu ITU-T. [3]

2.4 SD soubor

Jako SD soubor se označuje takový soubor, který obsahuje zdrojový kód v jazyce DAPAS. Soubory SD jsou vstupem pro překladač DASTEP.

3 Současný překladač a motivy pro vývoj nového

V současnosti používaný překladač DASTEP je program, který byl původně vyvinut ve firmě Siemens přibližně v roce 1993 a je implementován v jazyce Pascal. Program postupně měnil majitele, až jej dostala do užívání firma iXperta s.r.o., která je zadavatelem této práce na vývoj nového překladače.

Při analýzách současného programu jsem na doporučení konzultanta vycházela převážně z dokumentace a testování překladače metodou tzv. černé skříňky (bez nahlížení do zdrojového kódu). Zdrojové kódy mi nicméně byly dány k dispozici a nakonec jsem do části z nich nahlížela ve dvou případech detailnější analýzy konkrétních chybových hlášek ze syntaktické části programu. Tyto části byly velmi špatně čitelné a navíc psané v německém jazyce.

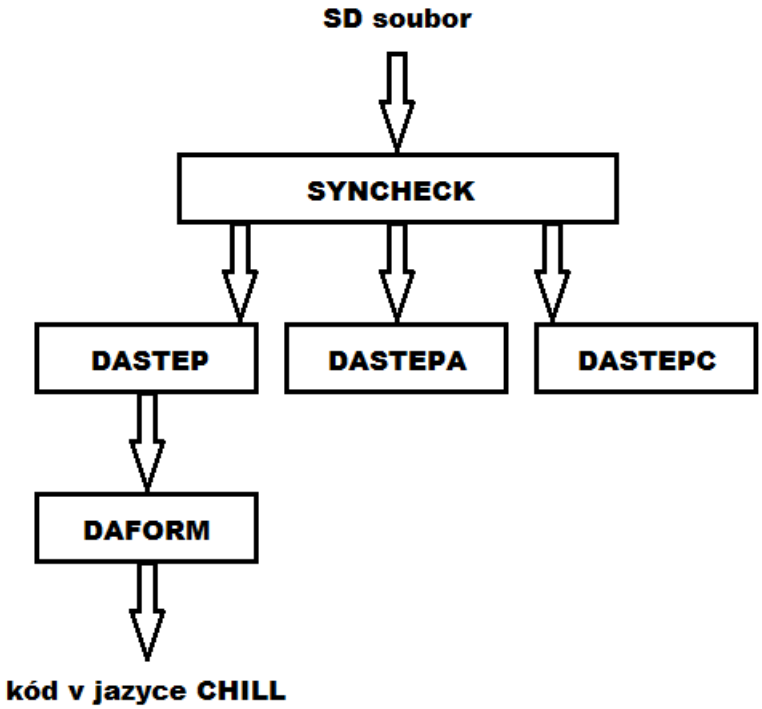
3.1 Motivy pro vývoj nového překladače

Stávající program DASTEP je z několika ohledů nevyhovující. Prvním důvodem je použitý jazyk Pascal, který je nemoderní a vhodný spíše k výuce, než k programování reálných aplikací. Jeho nedostatky jsou například absence prostředků pro zpracování chybových situací a také skutečnost, že nemá řetězce a pole proměnné délky [9] (dopad tohoto nedostatku zmiňuji v kapitole 3.2).

Program byl v minulosti již několikrát portován mezi různými platformami a to v kombinaci s jeho stářím a neustálými změnami vlastníka a osoby za něj zodpovědné vedlo k tomu, že neexistuje ucelená dokumentace, která by zaznamenávala jeho přesné fungování a všechny provedené změny. Existují dva dokumenty, [1] z roku 1993 a [2] z roku 1995, nicméně při testování stávajícího programu jsem zjistila, že oba se v několika případech liší od reálného fungování překladače. Některé části nejsou zdokumentovány vůbec, nebo naopak funkce či gramatická pravidla, zaznamenané v dokumentacích [1] a [2], již nejsou ve skutečnosti podporovány. I toto se stalo důvodem, proč se firma rozhodla pro vývoj nového překladače.

3.2 Struktura a další problémy současného překladače

Současný překladač je ve skutečnosti tvořen více programy, nicméně je pro ně zaužívané obecné označení program/překladač DASTEP, podle názvu jednoho z nich, a tohoto označení se v bakalářské práci držím také.



Obrázek 3.1: Schéma fungování současného překladače

Na obrázku 3.1 je graficky znázorněno rozdělení částí současného překladače. Ten umožňuje výstup nejen v jazyce CHILL (část DASTEP) ale i v jazyce C (část DASTEPC) a jazyce symbolických adres (část DASTEPA).

Vstupním bodem celého systému je program SYNCHECK, který provede lexikální a syntaktickou analýzu. Vzhledem k velikosti tabulek v programu SYNCHECK je vstupní soubor omezen například maximálním možným počtem stavů a událostí [2]. Tyto restrikce vznikly z důvodu použitého jazyka Pascal a nového programu se tedy týkat nebudou. Z dokumentace [1] vyplývá, že pokud SYNCHECK nalezne chyby, pak je vypíše a další zpracování končí. V opačném případě předá zpracovávaný vstupní kód dalšímu programu (dále se zaměřím pouze na DASTEP, protože DASTEPA a DASTEPC se již nepoužívají). Toto je jeden z příkladů, kdy dokumentace neodpovídá skutečnosti, protože i v případě chyb je kód předán dál a DASTEP se pokusí kód přeložit, přičemž pokračuje ve výpisu chyb. Chybové výpisy pak příliš neodpovídají skutečnému problému, dochází k jejich duplikaci a v některých případech se program dokonce úplně zasekne. Pokud ale nedojde k závažným chybám, je vygenerovaný kód předán programu DAFORM, který do něj přidá nové řádky a odsazení tak, aby byl výsledný CHILL kód snadno čitelný.

4 Návrh nového překladače

Jedním z prvních kroků při návrhu nového překladače bylo zjištění požadavků na implementační jazyk a operační systém. V případě operačního systému bylo nutné, aby program fungoval na systému Unix, konkrétně jeho distribuci *openSUSE 11*. Ohledně jazyka neměla firma žádné specifické požadavky, nicméně s pomocí konzultanta jsme výběr zúžili na jazyky Java a Python, z důvodů snadné přenositelnosti a také mých předchozích zkušeností s těmito jazyky. Jako pomocný nástroj pro lexikální a syntaktickou analýzu mi byl konzultantem doporučen nástroj ANTLR.

4.1 Implementační jazyk

Po zvážení kladů a záporů jazyků Java a Python jsem nakonec dala přednost Pythonu. Programy v jazyce Java sice obecně běží rychleji, tato výhoda však nabývá na významu až v případě větších projektů. Jednou z výhod jazyka Python je také jednodušší a rychlejší psaní kódu. [4]

4.2 Pomocný nástroj pro lexikální a syntaktickou analýzu

Doporučený nástroj ANTLR sice podporuje vygenerování lexikálního a syntaktického analyzátoru do jazyka Python, sám je však napsán v Javě. Tato kombinace dvou jazyků není příliš vhodná, proto jsem se se souhlasem konzultanta zaměřila na nástroj PLY, který je implementací nástrojů *lex* a *yacc* pro Python a sám je napsán také v Pythonu napsán [5]. *Lex* a *yacc* jsou základní nástroje pro lexikální a syntaktickou analýzu pro systém UNIX a jazyk C [6], PLY je ale na rozdíl od nich zároveň použitelný i pro platformu Windows.

4.3 Použitá verze jazyka Python

Požadavkem bylo, aby překladač fungoval na firemním serveru s nainstalovanou verzí překladače Python 2.6.9, nicméně v rámci co nejsnadnější přenositelnosti a udržovatelnosti jsem se rozhodla program napsat tak, aby byl spustitelný i ve verzi Python 3 pro případ, že by se firma časem rozhodla na novější verzi Pythonu přejít. Tuto vlastnost podporuje i použitá verze PLY 3.4 [5].

Mezi verzemi 2.6 a 3 jazyka Python existuje rozdílů několik, ale v případě překladače DASTEP byl relevantní pouze rozdíl ve funkci *print*, která bývá klasicky nejčastějším důvodem, proč kód ve verzi 2.X neběží pod 3.X. V případě Pythonu 2.X se totiž jedná o příkaz se specifickou syntaxí, v Pythonu 3.X jde o vestavěnou funkci [8].

Jednou z možností jak docílit toho, aby stejný kód běžel pod 2.X i 3.X je skript '2to3' v Pythonu 3 [8]. Vzhledem k tomu, že má být primární verzí Python 2.6, byla tato možnost zavržena. Pro potřeby tohoto projektu jsem tedy využila druhý způsob, a to příkaz

```
from __future__ import print_function
```

Díky tomuto příkazu lze používat v Pythonu 2.6 funkci *print* z verze 3 a není přitom nutné měnit kód při pozdější migraci [8]. Tato možnost neexistuje ve verzi Pythonu starší než 2.6. [7].

4.4 Struktura nového překladače

Rozdělení současného programu popsané v kapitole 3.2 je velmi nepraktické a v novém překladači pro takové dělení není důvod, proto všechny funkce zastává jeden program. Požadavkem byla implementace pouze překladu do jazyka CHILL, nicméně program je konstruován tak, aby se snadno v případě potřeby dal vygenerovaný syntaktický strom použít pro překlad i do jiných jazyků, například na základě hodnoty přidaného přepínače.

```
src
| - - code_generation
      | - - __init__.py
      | - - generate_CHILL.py
| - - ply
      | - - __init.py
      | - - lex.py
      | - - yacc.py
| - - compatibility_mode.py
| - - dastep.py
| - - dastep_lex.py
| - - dastep_yacc.py
| - - errors_warnings.py
| - - parser.out
| - - parsetab.py
```

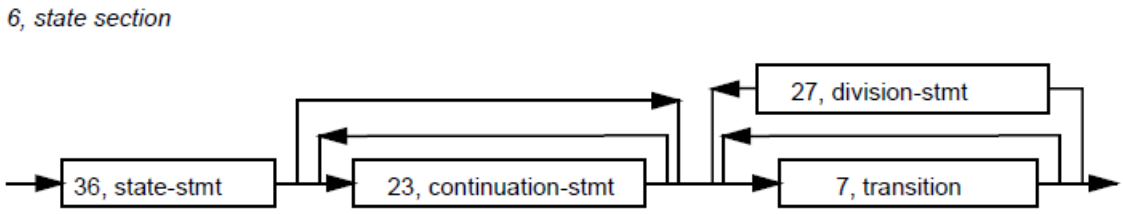
Vstupním bodem programu je modul *dastep.py*. Složka *ply* obsahuje moduly pro lexikální a syntaktickou analýzu z nástroje PLY. Do budoucna se ve složce *code_generation* předpokládá přidání nových modulů, které budou implementovat překlad do jiných jazyků. Soubory *parser.out* a *parsetab.py* jsou vygenerovány nástrojem PLY.

5 Gramatika jazyka DAPAS

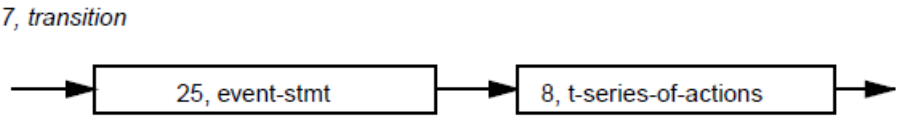
V původních dokumentech [1] a [2] je pro popis syntaxe použita grafická forma metajazyka. Textové řetězce v kroužcích jsou označeny jako konečné symboly a řetězce v obdélnících jako symboly, které jsou dále popsány dalšími syntaktickými pravidly [1, 2]. Tyto konečné symboly jsou synonymem pro tokeny [10]. Obrázky s pravidly ve formě metajazyka v této práci jsou převzaty z dokumentací [1] a [2].

5.1 Některé konstrukce jazyka DAPAS a jejich převod do BNF

Na obrázcích 5.1 a 5.2 je ukázka použitého metajazyka v dokumentaci pro popis syntaxe. Tato forma popisu je nevhodná pro programovací účely a protože nástroj PLY vyžaduje syntaxi specifikovanou v Backus – Naurově formě (BNF) [5] byl dalším z kroků v návrhu nového překladače převod gramatiky do BNF a také specifikace jednotlivých tokenů.



Obrázek 5.1: Ukázka původní gramatiky jazyka DAPAS



Obrázek 5.2: Ukázka původní gramatiky jazyka DAPAS

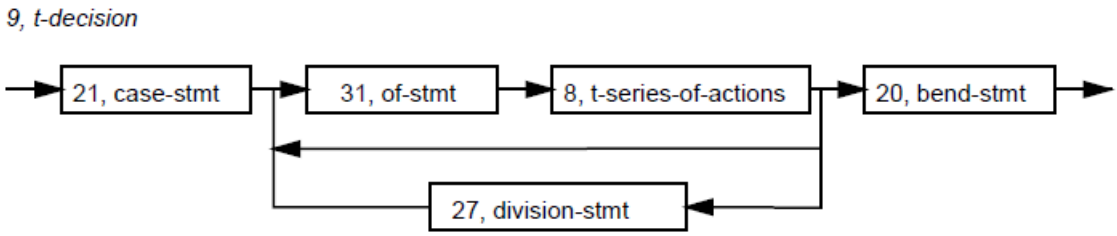
Část pravidel z obrázků 5.1 a 5.2 jsem do BNF převedla následovně:

```
state_section : state_stmt continuation_stmt transition
                state_section
              | state_stmt continuation_stmt transition

transition : event_stmt t_series_of_actions division_stmt transition
           | event_stmt t_series_of_actions transition
           | event_stmt t_series_of_actions

continuation_stmt : empty
                  | CONT text commentary continuation_stmt
```

Vzhledem k tomu, že BNF ve své základní formě neumožňuje využití regulárních výrazů pro opakování, musela jsem možnost opakování řešit pomocí rekurze, jak je vidět například na pravidle `continuation_stmt`.



Obrázek 5.3 : Pravidlo t-decision

Na obrázku 5.3 je znázornění pravidla *t-decision*, pomocí kterého se implementuje větvení programu. Jeho zvláštností je, že každé větvi, kromě té první, může předcházet symbol *division-stmt* [2]. Převod tohoto pravidla do formátu BNF byl poněkud složitější a vyžádal si vznik pomocného pravidla *t_decision_first*.

```
t_decision : case_stmt t_decision_branch_first bend_stmt

t_decision_branch_first : of_stmt t_series_of_actions
                            t_decision_branch
                        | of_stmt t_series_of_actions

t_decision_branch : of_stmt t_series_of_actions t_decision_branch
                  | of_stmt t_series_of_actions
                  | division_stmt of_stmt t_series_of_actions
                    t_decision_branch
                  | division_stmt of_stmt t_series_of_actions
```

Všechna pravidla nové gramatiky ve formátu Backus-Naurovy formy (BFN) jsou k práci přiložena jako Příloha A.

5.2 Klíčová slova v jazyce DAPAS

V programovacích jazycích většinou bývají klíčová slova rezervovaná, tzn. nelze je použít jako identifikátory [10]. Toto ale není případ jazyka DAPAS, ve kterém sice existuje seznam klíčových slov, ale aby se jednalo skutečně o slovo klíčové, musí toto slovo být prvním symbolem na řádku, předcházet mu mohou pouze bílé znaky [2].

Pokud se klíčové slovo nachází jinde, než na začátku řádku, nejedná se automaticky o chybu, pouze toto slovo pak není považováno za klíčové, ale za identifikátor.

Příklad 1.:

```
STATE IN
```

Ačkoliv obě slova *STATE* i *IN* mohou být v jazyce DAPAS klíčová, tady se za klíčové považuje pouze *STATE*, protože se nachází na začátku řádku. *IN* je identifikátor, v tomto případě název stavu.

Příklad 2.:

```
STATE
IN
```

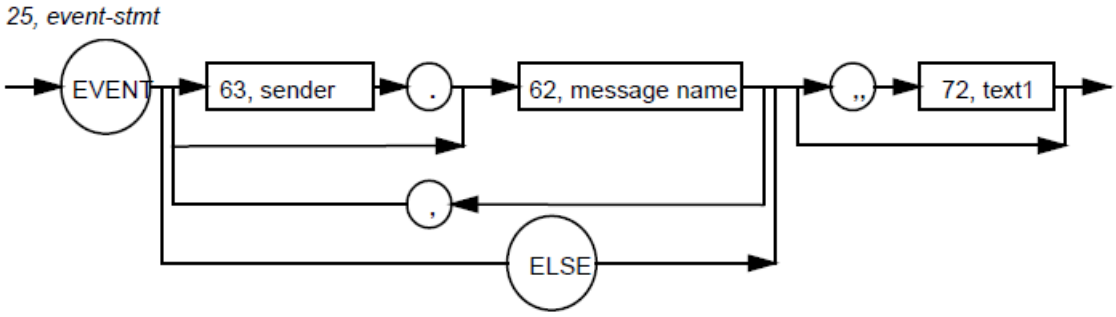
Ve druhém příkladu jsou obě slova na seznamu možných klíčových slov a zároveň na začátku řádku, tudíž budou obě považována za klíčová. Pravidla gramatiky ale tuto konstrukci neumožňují, proto by došlo k syntaktické chybě.

Příklad 3.:

```
STATE
AAA
```

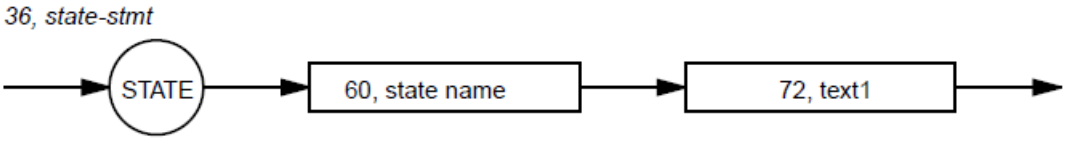
Slovo *STATE* je rozpoznáno jako klíčové. *AAA* není v seznamu klíčových jazyka DAPAS, proto se jedná pouze o identifikátor názvu stavu i přesto, že se nachází na začátku řádku.

5.3 Vznik nového pravidla „brief_explanation“



Obrázek 5.4: Pravidlo event-stmt

Na obrázku 5.4 je v metajazyce znázorněno pravidlo *event-stmt*. Symbol *text1* je nepovinný a může být použit pro krátký popis události [2]. Tento popis je následně generován i do výsledného souboru. Jak je vidět, *text1* se musí nacházet za symbolem dvou čárek.



Obrázek 5.5: Pravidlo state-stmt

V případě pravidla na obrázku 5.5 slouží *text1* v podstatě ke stejnému účelu (pro krátký popis stavu). U tohoto pravidla ale pravděpodobně došlo při psaní původní dokumentace [2] k chybě, protože v metajazyce je zakreslen jako povinný, v následném slovním popisu už je ale správně označen za nepovinný. Dokumentace také nezmiňuje žádný symbol, který by měl použití symbolu *text1* předcházet, nicméně při testování původního programu DASTEP jsem zjistila, že mu musí předcházet symbol jedné čárky.

Toto by představovalo další problémy při lexikální analýze, kde se snažím textové symboly tohoto typu vracet skutečně jako tokeny *TEXT* (a jsou podmíněny právě předcházejícím symbolem dvou čárek). Zároveň se v pravidlech vyskytuje další případ stejného využití symbolu *text1* a stejně jako v případě pravidla *event-stmt* mu musí předcházet symbol dvou čárek.

V případě pravidla *state-stmt* se tedy jedná o zvláštní výjimku, ke které jsem nenašla žádné opodstatnění. Navíc i při analýze stávajících SD souborů jsem objevila několik případů, ve kterých jejich autor použil čárky dvě. Druhá se tak stala součástí popisného textu, což jistě nebylo zamýšleným výsledkem a ukazuje to na zmatečnost v důsledku inkonzistence gramatiky pro použití těchto krátkých popisných textů.

Z těchto důvodů jsem se po domluvě s konzultantem z firmy iXperta rozhodla pro změnu v gramatice v případě pravidla *state-stmt*. Nová gramatika předpokládá, že nepovinnému textu pro krátký popis stavu předchází symbol ne jedné, ale dvou čárek (token *DOUBLECOMMA*). Tím došlo k unifikaci gramatiky pro krátké popisné texty a mohla jsem pro ně vytvořit nové unifikované pravidlo *brief_explanation*:

```
state_stmt : STATE ID brief_explanation commentary
brief_explanation : DOUBLECOMMA text
                  | empty
```

Podmínkou pro přijetí této změny ze strany firmy ale bylo zajištění, že i stávající SD soubory s původní gramatikou budou novým programem DASTEP přeložitelné. Této problematice se více věnuji v kapitole 11.

5.4 Pravidlo pro název události

Na obrázku 5.4 je vidět pravidlo pro název události (v dokumentaci [2] označen jako název zprávy), který se má skládat z částí *sender* a *message name,* oddělených tečkou. Obě tyto části jsou v základu pouze identifikátory. Tentýž předpoklad je i pro pravidlo *message-stmt*. Toto jsou jediné dva případy, kdy má znak tečky nějakou speciální funkci, ve všech ostatních může být zcela běžnou součástí identifikátoru. Z tohoto důvodu jsem se rozhodla nevytvářet samostatný token pro znak tečky, ale vytvořit zjednodušené pravidlo pro název zprávy:

```
messages : ID COMMA messages
         | ID
```

Součástí tohoto pravidla je následná kontrola symbolu ID na požadovaný tvar názvu zprávy za pomoci regulárních výrazů.

5.5 Pravidla pro komentáře

V jazyce DAPAS se komentáře píší za znak ';' (středník). Původní dokumentace [2] nicméně zmiňuje pouze tzv. řádek s komentářem. Tento řádek je definován jako řádek, jehož prvním nebílým znakem je ';', následovaný řetězcem jakýchkoliv znaků. Ve skutečnosti ale tyto komentáře nemusejí být pouze na samostatném řádu, ale v podstatě kdekoliv. Proto jsem na vhodná místa v gramatice přidala neterminál *commentary*, který odpovídá pravidlu pro komentář, ale nemusí se nacházet pouze na samostatném řádku.

5.6 Zrušená pravidla

Dokumentace [2] obsahuje několik konstrukcí jazyka DAPAS, při jejichž použití stávající překladač vypíše chybu, že se se tyto konstrukce již nepoužívají. Toto řešení bylo při úpravách programu v minulosti jednodušší, než předělávat lexikální a syntaktickou analýzu.

V rámci další spolupráce s firmou iXperta s.r.o. budu vytvářet novou dokumentaci pro překladač a gramatiku, která tyto konstrukce nebude zahrnovat, není tedy důvod, aby je překladač rozpoznával. Pravidla tohoto druhu nejsou tedy součástí nové gramatiky.

6 Lexikální analýza

Lexikální analyzátor čte zleva doprava proud znaků, který tvoří zdrojový kód právě překládaného programu a na žádost od syntaktického analyzátoru v něm postupně identifikuje tzv. lexémy, což jsou smysluplné posloupnosti znaků, kterou umí lexikální analyzátor rozeznat. Z každého takto nalezeného lexému vytvoří token, které jej reprezentuje, a odešle syntaktickému analyzátoru. [6, 10, 11]

6.1 Moduly ply.lex a dastep_lex

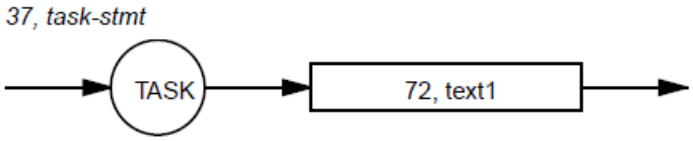
V nástroji PLY funkci lexikálního analyzátoru zastává modul *ply.lex* a, stejně jako unixový nástroj *lex,* vyžaduje pro své fungování seznam tokenů a jejich definici pomocí regulárních výrazů [5, 10]. Všechny tyto definice jsem zahrnula do modulu *dastep_lex.py*.

Další běžnou funkcí lexikálního analyzátoru je i odstranění bílých znaků a komentářů [11]. V případě jazyka DAPAS je odstranění bílých znaků žádoucí, odstranění komentářů nikoliv. Komentáře v tzv. hlavičkové části kódu v souborech SD jsou totiž převáděny do výstupního kódu CHILL.

Vzhledem k pravidlům pro klíčová slova v jazyce DAPAS popsaných v kapitole 5.2 nebylo možné pouze tato slova definovat, ale zároveň také zjistit, jestli se nacházejí na začátku řádku. K tomu jsem využila tzv. stavy lexikálního analyzátoru PLY. Po každém znaku nového řádku se analyzátor přepne do stavu *keyword*, ve kterém následující lexém, který by odpovídal tokenu *ID*, porovná se seznamem klíčových slov. Pokud se v něm lexém nachází, je vrácen token klíčové slovo, v opačném případě token *ID*.

Těchto stavů využívám také pro komentáře (jejich gramatika je popsána v kapitole 5.5). Po identifikování tokenu *SEMICOLON* (odpovídá znaku ';') se analyzátor přepne do stavu *commentary* a veškeré další znaky až do konce řádku přijímá jako token *TEXT*.

Další případy, kdy využívám přepínání stavů, jsou po některých klíčových slovech, například *TASK*, a také po dvou po sobě následujících znacích ',' (čárka), které označuji jako token *DOUBLECOMMA*. Podobně jako u komentářů je po identifikování těchto tokenů třeba načítat další znaky jako *TEXT*.



Obrázek 6.1: Pravidlo task-stmt

Na obrázku 6.1 je pravidlo *task-stmt*. Symbol *text1* je v dokumentaci [2] definován jako posloupnost jakýchkoliv tisknutelných znaků kromě středníku. Tato posloupnost nekončí znakem konce řádku, ale znakem středníku, který symbolizuje začátek komentáře, nebo klíčovým slovem na začátku nového řádku.

Kvůli takovýmto pravidlům byl zapotřebí způsob, jak určit, zda se na novém řádku nachází klíčové slovo, nebo se má pokračovat v načítání tokenu *TEXT*. Pro toto jsem využila funkci klonování

lexikálního analyzátoru, která je vhodná právě pro případy, kdy je třeba se předem podívat na následující token [5].

Tato specifická pravidla jazyka DAPAS, kvůli kterým nelze pro specifikaci lexémů použít pouze regulární výrazy, mají na svědomí to, že lexikální analýza je poměrně složitá a bohužel také ne úplně přehledná.

6.2 Chyby, které mohou nastat během lexikální analýzy

V rámci lexikální analýzy jsem definovala tři druhy chyb, které mohou nastat. Jejich detekci a zpracování se podrobněji věnuji v kapitole 9.1.

`E_NUM`

Pokud token začíná číslicí, pak předpokládám, že bude obsahovat pouze číslice. V případě, že obsahuje něco jiného, dojde k této chybě.

`E_NUM_RANGE`

Pokud token obsahuje pouze číslice, pak kontroluji hodnotu výsledného čísla. Dle dokumentace [2] jsou povoleny pouze číselné hodnoty v rozmezí 1-999, nicméně původní program ve skutečnosti povoluje i hodnotu 0. Proto jsem tuto hodnotu zahrnula do povoleného rozmezí a k chybě *E_NUM_RANGE* dojde v případě, že je číslo vyšší než 999.

`E_ILLEGAL_CHAR`

Modul *lex.py* obsahuje funkci *t_error*, která má na starosti chyby, které nastanou, pokud se na vstupu objeví nepovolené znaky [5]. V těchto případech vracím chybu *E_ILLEGAL_CHAR*. Ve skutečnosti by ale k této chybě dojít nemělo, protože každý znak je v nějakém tokenu definován.

7 Syntaktická analýza

Syntaxe zdrojového jazyka je specifikována gramatikou, která je založena na konečném množství pravidel. Pomocí těchto pravidel pak syntaktický analyzátor ověří, že posloupnost tokenů, které mu poskytl lexikální analyzátor, tvoří syntakticky korektní program. [6, 11]

7.1 Moduly ply.yacc a dastep_yacc

V nástroji PLY implementuje funkci syntaktického analyzátoru modul *ply.yacc* [5]. Veškeré definice pravidel gramatiky jsem zařadila do modulu *dastep_yacc.py*. Nástroj PLY neposkytuje žádnou zvláštní funkci pro vytvoření abstraktního syntaktického stromu, který by reprezentoval syntaktickou strukturu získanou ze zdrojového kódu během syntaktické analýzy [6]. Konstrukce takového stromu ale v nástroji PLY není složitá [5].

7.2 Třída Node

Pro implementaci abstraktního syntaktického stromu jsem definovala třídu *Node*.

```
class Node:
     def __init__(self, type, children=None, leaf=None,
lexpos=None)
```

Argument *type* slouží k lepší orientaci ve stromě, většinou jde o název pravidla, podle kterého byl uzel vytvořen. Argument *lexpos* využívám pouze pro listy stromu a udržuje v sobě informaci o pozici, na které se ve vstupním kódu nachází první znak tokenu v tomto listu. Argument *leaf* je definován, pokud se jedná o list stromu, v opačném případě je definován argument *children*, který obsahuje seznam následujících uzlů stromu.

7.3 Chyby, které mohou nastat během syntaktické analýzy

Během syntaktické analýzy může být ve zdrojovém kódu nalezeno několik druhů syntaktických chyb. Jejich detekci a zpracování je věnována kapitola 9.2.

`E_BRIEF_STATE`

V kapitole 5.3 jsem popsala problém s gramatikou pro krátký popisný text, který lze použít po definici stavu. Tento text má podle upravené gramatiky následovat za tokenem *DOBLECOMMA*. Chyba *E_BRIEF_STATE* nastává v případě, že se na vstupu namísto očekávaných dvou čárek nachází pouze jedna.

E_USE

Při testech na existujících SD souborech jsem v některých narazila na klíčové slovo *USE*. Jednalo se o staré soubory, které se dnes již s největší pravděpodobností nepoužívají, protože klíčové slovo *USE* není součástí zdokumentované gramatiky a současný program DASTEP soubory s tímto slovem není schopen přeložit. Některá ze starších verzí programu však toto slovo musela podporovat,

protože jsem měla k dispozici soubory s výsledným kódem CHILL, který byl dříve z těchto SD souborů vygenerován. Při analýze souborů jsem zjistila, že klíčové slovo *USE* plnilo stejnou funkci jako *PROC*, které jej později nahradilo. Rozhodla jsem se tedy definovat chybu *E_USE*, která nastane v případě, že se ve vstupních kódu objeví klíčové slovo *USE*, a upozorní na jeho zastaralost.

`E_NO_MESS_NAME`

V kapitole 5.2 jsem popisovala pravidlo pro názvy zpráv. Chyba *E_NO_MESS_NAME* nastane v případě vynechání povinné části *message name*.

`E_UNEXPECTED_TOKEN`

Tato chyba nastává v případě, že aktuálně přijatý token neodpovídá syntaktickým pravidlům.

`E_MISSING_TOKEN`

Chyba chybějícího tokenu značí, že se nepodařilo zredukovat celé pravidlo, ale zároveň už na vstupu nejsou další tokeny.

8 Sémantická analýza a generování výsledného kódu jazyka CHILL

Po úspěšném provedení syntaktické analýzy se v ní vytvořený abstraktní syntaktický strom předá modulu *generate_CHILL*. V tomto modulu se postupně prochází syntaktickým stromem a předgenerovávají se části výsledného CHILL kódu. Zároveň zde probíhá sémantická analýza a na závěr se části vygenerovaného kódu vypíší v očekávaném pořadí.

Dokumentace [1] a [2] sice popisují samotný jazyk DAPAS, nevěnují se ale téměř vůbec jeho převodu do jazyka CHILL. Způsob a pravidla pro převod konstrukcí jazyka bylo tedy nutné získávat spouštěným současného překladače a analýzou jeho výstupů.

8.1 Sémantická analýza a tabulka symbolů

Bývá běžnou praxí, že plnění tabulky symbolů a sémantická analýza probíhají před samotným generováním výsledného kódu. V novém překladači DASTEP ale tyto akce probíhají víceméně zároveň v rámci modulu *generate_CHILL*. Důvodem k tomuto kroku byla některá pravidla gramatiky jazyka DAPAS. Například pravidlo *task-stmt*, ve kterém se po klíčovém slově *TASK* nachází tokeny *TEXT*.

TASK QF_V_REST_CAUSED_BY_ETHER := FALSE

Tento příklad je jedním z reálných použití konstrukce *task-stmt,* část za slovem TASK ale nepodléhá žádným konkrétnějším pravidlům. Převedení konstrukce do jazyka CHILL spočívá v odstranění slova *TASK* a přidání středníku.

QF_V_REST_CAUSED_BY_ETHER := FALSE;

Součástí převodu je ale v některých případech i kontrola, zda se v tokenu *TEXT* nevyskytuje tzv. importované jméno, jehož forma je definována v [1]. Je tedy třeba dodatečně pomocí regulárních výrazů vyhledávat tato importovaná jména. V uvedeném příkladu splňuje podmínky pro importovaná jména řetězec *QF_V_REST_CAUSED_BY_ETHER*. Takto nalezené řetězce jsou ve výsledném kódu CHILL součástí seznamu *SEIZE*, který se vypisuje na začátku.

Překladač DASTEP nevytváří tabulku symbolů v pravém slova smyslu, ale spíše pomocnou strukturu, ve kterém uchovává všechny informace, které by mohly mít vliv na vznik chyb.

8.2 Generování kódu pro konektory

Zajímavou částí jazyka DAPAS z hlediska generování kódu jsou tzv. konektory *IN* a *OUT*. Pomocí nich lze vytvářet určitý druh skoků v rámci programu.

```
OUT 1
TASK TASK_AFTER_OUT
IN 1
TASK TASK_AFTER_IN
```

Pokud se ve zdrojovém kódu jazyka DAPAS objeví předchozí posloupnost příkazů, pak se v tomto místě vygeneruje

```
KONNEKTOR := TEST_STATE_001;
TASK_AFTER_OUT;
KONNEKTOR := TEST_STATE_001;
```

Následně je do konečné části souboru s generovaným kódem CHILL umístěn seznam konektorů *IN*.

```
DO WHILE KONNEKTOR /= KEIN_SPRUNG;
  IN_AKTIONEN :
    CASE KONNEKTOR OF

    (TEST_STATE_001):
      /*##TEST_MOD_0002700000 */
      KONNEKTOR := KEIN_SPRUNG;
      TASK_AFTER_IN;
      CURR_STATE := TEST_STATE;

  ELSE
            KONNEKTOR := KEIN_SPRUNG;

  ESAC IN_AKTIONEN;
OD;
```

Zjednodušeně lze říci, že pomocí konektoru *IN* se označí místo na které lze skákat a poté se na toto místo skočí konektorem *OUT*.

9 Práce s chybami a varováními

Chybám a varováním je dedikován celý modul *errors_warnings*. V tomto modulu jsou definovány tři třídy: třída *Dastep_error* pro chyby, třída *Dastep_Warning* pro varování a třída *Errors_Warnings*, která obsahuje seznamy všech nalezených chyb a varování, metody pro jejich seřazení, tisk a také metodu pro výpočet čísla řádku, na kterém se daná chyba nebo varování nachází. Modul *errors_warnings* zároveň přehledně na jednom místě uchovává texty veškerých chybových a varovných hlášek.

Chybové hlášky a formát jejich výpisu vychází částečně z původního překladače a vždy obsahuje informaci o tom, na kterém řádku k chybě došlo. Nejprve se vypíše vypíší nalezené chyby a poté varování. Chybový pak může vypadat například takto:

```
ERROR!   Line     7: Message name 'TEST_EVENT' already defined in
state section
WARNING! Line     5: State name 'TEST_STATE' does not appear in a
'NEXST' statement
```

9.1 V rámci lexikální analýzy

Funkce *t_error* modulu *ply.lex* se volá v případě, že se ve vstupním kódu objeví nepovolený znak během lexikální analýzy. Tento znak je pak zahozen a pokračuje se analýzou dalšího znaku ze vstupu. Díky tomu se naleznou všechny nelegální znaky a ne pouze první. [5]

Pokud během lexikální analýzy došlo k chybám, tak se všechny tyto chyby vypíší a program se ukončí. Naopak pokud lexikální analýza proběhla bez chyb, tak následuje analýza syntaktická.

Seznam všech typů chyb, které mohou nastat během lexikální analýzy je v kapitole 6.2.

9.2 V rámci syntaktické analýzy

Modul *ply.yacc* se s chybami vyrovnává zavoláním funkce *p_error*, kde argumentem je token, který chybu způsobil. Syntaktický analyzátor následně přejde do módu pro zotavení po chybě, ve kterém zůstává a zahazuje tokeny, dokud se mu úspěšně nepodaří uplatnit pravidlo na alespoň tři po sobě jdoucí tokeny. [5]

Bohužel značná část pravidel jazyka DAPAS obsahuje méně než tři tokeny, takže se může stát, že program neodhalí ihned všechny syntaktické chyby, které by způsobily třeba právě zahozené tokeny. Tyto chyby se však projeví v okamžiku, kdy se ve vstupním kódu opraví chyby předcházející.

Pokud během syntaktické analýzy dojde k chybě z důvodu dosažení znaku konec souboru (*EOF*) je funkce *p_error* zavolána s argumentem *None* [5]. V tomto případě bohužel PLY nenabízí možnost, jak snadno zjistit, během zpracovávání kterého pravidla k chybě došlo.

Když se během syntaktické analýzy neobjeví chyby přijde na řadu analýza sémantická a generování kódu v jazyce CHILL. V případě syntaktických chyb nemohl být vygenerován abstraktní syntaktický strom, není tedy možné dál pokračovat v překladu a proto se vypíší nalezené chyby a program je ukončen.

10 Spouštění programu, jeho parametry a návratové kódy

Překladač DASTEP se ve firmě iXperta s.r.o. nepouští přímo, ale pomocí pomocného skriptu *do.dastep*, který má na starosti veškeré kontroly před samotným spuštěním. Kontroluje například existenci zadaných souborů a také zajišťuje, že bude DASTEP spuštěn vždy se správně zadanými parametry. [1]

10.1 Parametry programu

Díky existenci pomocného skriptu *do.dastep* není v programu potřeba téměř žádná kontrola zadaných parametrů. Samozřejmě je ale nutné zajistit, že nový překladač bude přijímat stejné parametry a ve stejném pořadí jako původní program. Text nápovědy, kterou jsem vytvořila, krátce popisuje spouštění programu a význam jeho jednotlivých parametrů.

```
Usage: dastep.py ilib namueb set|dummy err|dummy mac1|dummy mac2|
dummy source [-debug]
ilib = superset library
namueb = output file for the step table
set = output file for the states, or keyword 'dummy'
err = name of error file, or keyword 'dummy'
mac1 = central macro library for the expansion, or keyword 'dummy'
mac2 = private macro library for the expansion, or keyword 'dummy'
source = input file in dapas language
-debug prints out also tokens and whole representation of syntax
tree
```

Použití parametrů *ilib* a *set* nemá na výsledný překlad žádný vliv. Jejich význam a funkcionalita nejsou téměř zdokumentovány a a byly pravděpodobně ztraceny v průběhu let.

S parametry pro knihovny maker *mac1* a *mac2* nový překladač sice umí pracovat, samotné používání maker v překládaném kódu ale zatím podporováno není (více o makrech v kapitole 12.1).

Pokud je zadán parametr *err* jsou chybové výpisy přesměrovány do souboru specifikovaného tímto parametrem. Když je místo názvu souboru použito klíčové slovo *dummy* jsou tyto výpisy směrovány na standardní chybový výstup [1]. Parametr *err* ve stávajícím překladači nefunguje a chybové výpisy tak nelze automaticky přesměrovat. Nový překladač ale tuto možnost plně podporuje a jde o případ obnovení jedné ze ztracených funkcionalit.

Význam parametru *source* je trochu jiný, než by se běžně předpokládalo. Indikuje sice název zdrojového souboru s překládaným kódem, ale pouze pro účel vygenerování hlavičky ve výsledném kódu jazyka CHILL, která obsahuje řádek s názvem zdrojového souboru. Samotný překládaný kód se do programu dodává přes standardní vstup.

Namueb je název souboru, do kterého se uloží vygenerovaný CHILL kód.

10.2 Nový parametr -debug

Jediným rozdílem v parametrech oproti původnímu programu je poslední, nepovinný parametr *-debug*.

Během vývoje překladače jsem pro ladící účely potřebovala přehledně vidět, jak přesně vypadá abstraktní syntaktický strom, který byl vytvořen během syntaktické analýzy. Pro tento účel jsem ve třídě *Node*, která syntaktický strom implementuje, vytvořila metodu *indent*. Tato metoda tiskne vygenerovaný abstraktní syntaktický strom i s příslušným odsazením tak, aby byl snadno čitelný.
Ukázka části abstraktního syntaktického stromu vytištěného pomocí metody *indent*:

```
CHILD(type: state_section)
     |CHILD(type: state_stmt)
     |     |CHILD(type: STATE)
     |     |     |LEAF: STATE
     |     |     |lexpos: 65
     |     |CHILD(type: state_name)
     |     |     |LEAF: TEST_STATE
     |     |     |lexpos: 71
     |CHILD(type: transition)
     |     |CHILD(type: event-stmt)
     |     |     |CHILD(type: EVENT)
     |     |     |     |LEAF: EVENT
     |     |     |     |lexpos: 83
     |     |     |CHILD(type: message)
     |     |     |     |CHILD(type: message_name)
     |     |     |     |     |LEAF: TEST_EVENT
     |     |     |     |     |lexpos: 89
     |     |CHILD(type: t_series_of_actions)
     |     |     |LEAF: None
     |     |     |lexpos: None
```

Po dokončení práce na překladači DASTEP jsem se rozhodla metodu *indent* neodstraňovat. Důvodem bylo to, že v budoucnu počítám s dalším vývojem tohoto překladače a tedy i možnými zásahy do gramatiky. Právě při takovýchto zásazích a jejich následném ladění může být metoda *indent* velmi užitečná. Zároveň ale není žádoucí, aby metoda zůstala zachována jen jako nepoužívaná část zdrojového kódu. Její použití by tak záviselo na tom, že vývojář, který by chtěl využít její funkce, musí nějakým způsobem měnit zdrojový kód překladače a metodu *indent* volat.

Proto jsem vytvořila nový nepovinný parametr *-debug*, který nejen že zajistí zavolání metody *indent* po provedení syntaktické analýzy, ale zároveň v průběhu lexikální analýzy vypisuje informace o jednotlivých nalezených tokenech. Jeho používání není primárně zamýšleno pro ty, kteří budou program DASTEP jako překladač využívat, ale pro vývojáře, který bude zodpovědný za program a jeho další případný vývoj.

10.3 Návratové kódy programu

Běh programu může skončit s jedním ze tří návratových kódů. Kód *0* je vrácen v případě, že nebyly nalezeny žádné chyby ani varování a zdrojový soubor byl tedy úspěšně přeložen. Pokud byly vygenerovány varování, ale žádné chyby, tak program vygeneroval soubor s CHILL kódem a skončil s návratovou hodnotou *2*. V případě návratového kódu *3* byly při běhu programu nalezeny chyby a překlad skončil neúspěchem. Není rozlišováno, v rámci které analýzy k chybám došlo.

Tento systém návratových kódu byl vytvořen tak, aby odpovídal fungování stávajícího překladače.

11 Mód zpětné kompatibility

Součástí požadavků na nový překladač DASTEP byl předpoklad, že bude schopen přeložit všechny již existující SD soubory. I vzhledem ke změně v gramatice pro krátký popisný text za definicí stavu, popsané v kapitole 5.3, by toto bez dalšího opatření nebylo možné. Tímto opatřením se stal modul *compatibility_mode*.

Zdrojový kód nejprve projde běžnou lexikální a syntaktickou analýzou, jejímž výstupem může být seznam chyb. Pokud se v tomto seznamu nacházejí chyby, které jsem definovala jako chyby kompatibility, dojde k sepnutí módu zpětné kompatibility. Chyby kompatibility jsou momentálně dvě a jedná se o *E_BRIEF_STATE* a *E_USE* z kapitoly 7.3.

Funkce módu zpětné kompatibility spočívá v tom, že vstupní kód upraví, aby nedocházelo k chybám kompatibility. V případě chyby *E_BRIEF_STATE* přidá před krátký popisný text druhou čárku, aby nyní odpovídaly požadovanému tokenu *DOUBLECOMMA*. V rámci opravy chyby *E_USE* nahradí ve vstupním kódu zastaralá klíčová slova *USE* za nynější *PROC*. V obou případech dojde ve vstupním řetězci k přidání znaku a tím i posunutí pozic všech následujících znaků o jedna. Při tisku chyb a také pro funkci na rozpoznání čísla řádku, která se využívá při generování výsledného kódu, je ale potřeba hodnota pozice před posunem, proto je nutné všechny tyto posuny zaznamenat a následně s nimi počítat.

Takto upravený vstupní kód poté znovu projde celou lexikální a syntaktickou analýzou a dále se pracuje s abstraktním syntaktickým stromem vygenerovaným v tomto druhém průchodu. Všechny chyby kompatibility, ke kterým došlo v rámci prvního průchodu, jsou převedeny na varování a zároveň se přidá varování o sepnutí módu kompatibility.

V současnosti se mód zpětné kompatibility zapíná sám a to v případě, že došlo k chybám kompatibility. Do budoucna předpokládám spíše například přidání nového parametru programu, který by tento mód řídil. V ideálním případě časem i úplné zrušení modulu *compatibility_mode* a požívání pouze nově nastavených pravidel gramatiky.

12 Možná vylepšení

Program je momentálně plně funkční a použitelný, nicméně do budoucna se počítá s jeho dalším vývojem a možným vylepšením. Především jde o podporu maker a také zlepšení v oblasti detekce a výpisu chyb.

12.1 Makra

Ačkoliv se nejednalo o součást zadání této bakalářské práce, tak původní program DASTEP používání maker ve zdrojovém kódu v jazyce DAPAS podporuje. Tato makra slouží především k uložení volání procedur, která lze jejich použitím velmi zkrátit. Makra jsou definována se speciálních souborech nazvaných jako knihovny maker a jejich volání je, stejně jako například v jazyce C, uvozeno znakem '#'. [2]

Pravidla pro volání maker nejsou součástí současné gramatiky jazyka DAPAS, protože pro jejich zpracování předpokládám zavedení nějaké formy „preprocesoru“.

Knihovny maker mají svou vlastní gramatiku a pro zavedení podpory maker bude zapotřebí vytvořit nový lexikální a syntaktický analyzátor, který bude fungovat nad touto gramatikou.

12.2 Chybové hlášky

V kapitole o práci s chybami a varováními 9.2 jsem popsala chování nástroje PLY v případě, že během syntaktické analýzy dojde k chybě z důvodu nezredukovaného pravidla a dosažení znaku konce souboru (EOF).
Například při použití následující konstrukce pro větvení, dojde k chybě *E_MISSING_TOKEN*, kterou jsem pro tento případ definovala.

```
CASE some_case
  OF branch_1
    TASK some_task
  OF branch_2
```

Důvodem pro tuto chybu je chybějící klíčové slovo *BEND*, které je nutnou součástí příkazu pro větvení a značí jeho ukončení. Program tak sice správně určí, že ve zdrojovém SD souboru nějaký token chybí, není už ale schopen uživateli podat žádné podrobnější informace.

Podobně pokud se za uvedenou konstrukci přidá například klíčové slovo *STATE*, dojde k vyvolání chyby *E_UNEXPECTED_TOKEN*, ze které se ale uživatel nedozví že zapomněl ukončit větvení.

Jako vhodné vylepšení programu tedy do budoucna plánuji rozšíření zpracování chyb a chybových hlášek tak, aby měly větší vypovídající hodnou a byly uživatelsky přívětivější.

13 Testování

Program byl testován převážně na operačním systému openSUSE 11 s překladačem Python verze 2.6.8 a 2.6.9, ale i na systému Windows 7 s Pythonem 3.

Pro účely testování překladače DASTEP vzniklo celkem 383 testovacích případů. Necelou čtvrtinu z tohoto čísla tvoří testovací případy vytvořené z již existujících souborů SD a tyto testy jsou pro odlišení označeny pořadovými čísly začínajícími číslicí 9.

Všechny ostatní testy, číslované již klasicky počínaje testem číslo 000, jsem vytvářela ručně za účelem testování konkrétních konstrukcí jazyka DAPAS a chování původního překladače.

Každý testovací případ má vlastní složku pojmenovanou *testXXX (XXX* je číslo testu) a je tvořen několika soubory. Tím hlavním je soubor *input*, který obsahuje testovaný kód v jazyce DAPAS. Dále je ve složce soubor *c6*, který obsahuje kód v jazyce CHILL tak, jak jej ze souboru *input* vygeneroval původní překladač. Pokud původní překladač skončil s chybami a CHILL kód nebyl vygenerován, pak chybí také soubor *c6*. Soubor *output* obsahuje výpisy z překladu původním programem, včetně výpisů chyb a varování. Každý testovací případ jsem ještě opatřila souborem *note*, který obsahuje poznámky k jednotlivým testům.

Testování zjednodušeně probíhá tak, že v rámci každého testovacího případu se nový překladač spustí nad souborem *input* a případný vygenerovaný CHILL kód se uloží, stejně jako výpisy nalezených chyb a varování. Poté dojde k porovnání CHILL kódů vygenerovaných oběma překladači, přičemž se ignorují některé řádky v hlavičkách souborů, například s datem vygenerování. Nejprve jsou soubory porovnány včetně rozmístění všech bílých znaků, výsledek ve výsledcích testů odpovídá sloupečku *SPACE*. V případě přesné shody je zaznamenán výsledek *OK*. Pokud nový překladač nevygeneroval kód, ale přitom ten původní ano, je ve výsledku testu znak '!'. Znak '+' je naopak u testů, ve kterých původní program kód z důvodů chyb nevygeneroval, ale nový překladač žádné chyby nenašel. Pokud se vygenerované kódy obou překladačů liší, je zaznamenána hodnota *FAILED* a přejde se k porovnání s vynecháním bílých znaků a zaznamenání jeho výsledku do sloupce *NOSPACE*.

V jazyce CHILL slouží bílé znaky pouze k oddělení jednotlivých tokenů, které tvoří kód, jinak je jejich využití čistě estetickou záležitostí. Překladač výsledný kód generuje s vhodným odsazením a zalomením dlouhých řádků. Právě v zalamování řádků se chování obou překladačů může občas trochu lišit, nikdy ale nedojde k zalomení uprostřed tokenu. Proto, pokud se po odstranění bílých znaků výstupy obou překladačů rovnají, jde o rovnocenný kód.

Celý test je proto považován za úspěšný, pokud obsahuje *OK* v jednom nebo druhém sloupci. Účelem těchto testů tedy bylo srovnání výstupů původního a nového překladače DASTEP, výpisy chyb nejsou porovnávány.

Testy v nichž není žádný příznak úspěchu jsou takové, při kterých oba překladače detekovali chyby v souboru SD a nevygenerovaly tedy kód CHILL. Výpisy nalezených chyb nejsou žádným automatickým způsobem porovnatelné, proto jsou tyto testy bez označení úspěchu.

Následně jsem analyzovala postupně všechny testy, které neměly označení OK. Výsledkem analýzy je zjištění, že takovéto testy nelze označovat za neúspěšné z hlediska správného fungování překladače. Původní překladač totiž někdy vygeneruje kód i přes zjevné chyby, nebo naopak označí za chyby konstrukce jazyka DAPAS, které dokumentace [2] popisuje jako přípustné. Tyto zdánlivě

neúspěšné testy jsou ve skutečnosti těmi, ve kterých nový překladač napravuje nedostatky a chyby toho starého.

Testovací případy jsou součástí bakalářské práce, ale pouze takové, které se zakládají na mnou vytvořených SD souborech. Ve výsledcích testů (příloha A) jsou zahrnuty i výsledky testů začínajících číslem 9, ve kterých je testován překlad SD souborů pocházející od firmy iXperta s.r.o.

14 Závěr

Cílem této bakalářské práce bylo navrhnout a vytvořit překladač mezi jazykem stavových diagramů (DAPAS) a jazykem CHILL. Součástí zadání bylo i vytvoření formálního popisu gramatiky jazyka a vytvořený program měl nakonec být otestován na správnost generovaného kódu.

Gramatika jazyka DAPAS je ve firemních dokumentech popsána graficky formou metajazyka a pro programovací účely ji bylo nutné převést do Backus-Naurovy formy. V rámci tohoto převodu jsem se snažila dodržet rozdělení na jednotlivá „pravidla“ použitá v metajazyce, někdy ale bylo potřeba toto rozdělení upravit nebo i přidat nová pravidla. V konečném výsledku jde ale o popis stále stejné gramatiky, kromě úmyslných změn v rámci jejího vylepšení popsaných v této práci.
Program, který jsem navrhla a implementovala splňuje zadané požadavky a je konstruován s ohledem na budoucí úpravy a rozšíření. Není závislý na požadované verzi Pythonu 2.6, ale funguje i pro novější verzi Pythonu 3. Výsledný kód překladače je vhodně komentovaný a logicky členěný na jednotlivé moduly.

Velkou část mé práce tvořila také testovací část, v rámci které vzniklo téměř 400 testovacích případů z nichž většinu tvoří mnou napsané soubory a díky kterým testuji jednotlivé konstrukce jazyka DAPAS. Součástí těchto testů byly i soubory, které se ve firmě používají pro překlad původním překladačem a bylo nutné, aby je dokázal správně přeložit i můj program. Toto se z velké části podařilo a v případech, kdy byl soubor pro můj program nepřeložitelný, se nakonec ukázalo, že skutečně obsahoval chyby takového rázu, že k překladu nemělo dojít.

Literatura

[1] KOSSAKOWSKI, G. 1993. *DASTEP: Operating Description*. Interní dokument firmy iXperta s.r.o.

[2] KOSSAKOWSKI, G. 1995. *DAPAS Language: Language Description*. Interní dokument firmy iXperta s.r.o.

[3] ITU. 1999. *ITU-T Z.200: CHILL - The ITU-T Programming Language*.

[4] *Python: Comparing Python to Other Languages* [online]. [cit. 2015-05-18]. Dostupné z WWW: https://www.python.org/doc/essays/comparisons/

[5] BEAZLEY, David M. *PLY (Python Lex-Yacc)* [online]. [cit. 2015-05-18]. Dostupné z WWW: http://www.dabeaz.com/ply/ply.html

[6] MEDUNA, Alexander. 2008. *Elements of Compiler Design*. New York: Taylor & Francis Group. ISBN 1-4200-6323-5.

[7] *Python v2.6.9 documentation* [online]. 2013. [cit. 2015-05-19]. Dostupné z WWW: https://docs.python.org/2.6/

[8] LUTZ, Mark. 2009. *Learning Python*. 4th ed. Beijing: O'Reilly, xlix, 1160 s. ISBN 978-0-596-15806-4.

[9] *Wikipedie: Otevřená encyklopedie: Pascal (programovací jazyk)* [online]. c2014 [cit. 18. 05. 2015]. Dostupný z WWW: <http://cs.wikipedia.org/w/index.php?title=Pascal_(programovac%C3%AD_jazyk)&oldid=12093151>

[10] AHO, Alfred V. 2007. *Compilers: principles, techniques*. 2nd ed. Boston: Pearson ; Addison Wesley, xxiv, 1009 s. ISBN 03-214-9169-6.

[11] PITTMAN, Thomas a James PETERS. *The Art of Compiler Design: Theory and Practice*. New Jersey: Practice-Hall. 1992, 420 s.. ISBN 0-13-046160-1.

Seznam příloh

Příloha A: Implementovaná gramatika jazyka DAPAS
Příloha B: Výsledky testů

Příloha A

```
state_diagram : leader generation_parameters state_section

leader : comment_line
       | empty

comment_line : SEMICOLON TEXT comment_line
             | SEMICOLON TEXT

comment_line : SEMICOLON comment_line
             | SEMICOLON

generation_parameters : generation_stmt
                      | empty

generation_stmt : GENPARAM TEXT leader_commentary generation_stmt
                | GENPARAM TEXT leader_commentary

state_section : state_stmt continuation_stmt transition
                state_section
              | state_stmt continuation_stmt transition

state_stmt : STATE ID brief_explanation commentary

state_stmt : STATE ID COMMA error

continuation_stmt : empty
                  | CONT text commentary continuation_stmt

transition : event_stmt t_series_of_actions division_stmt transition
           | event_stmt t_series_of_actions transition
           | event_stmt t_series_of_actions
```

```
t_series_of_actions : task_stmt t_series_of_actions
                    | procedure_call_stmt t_series_of_actions
                    | message_stmt t_series_of_actions
                    | comment_stmt t_series_of_actions
                    | new_state_stmt t_series_of_actions
                    | in_connector_stmt t_series_of_actions
                    | out_connector_stmt t_series_of_actions
                    | t_decision t_series_of_actions
                    | relocation_stmt t_series_of_actions
                    | x_procedure_call_stmt t_series_of_actions
                    | empty

task_stmt : TASK text commentary

procedure_call_stmt : PROC text commentary

procedure_call_stmt : USE text commentary

message_stmt : MESSG messages brief_explanation commentary
             | MESSGE messages brief_explanation commentary
             | MESSGI messages brief_explanation commentary

comment_stmt : COMM text commentary

new_state_stmt : NEXST ID commentary
               | NEXST SAME commentary

in_connector_stmt : IN ID COMMA NUMBER commentary
                  | IN NUMBER commentary

out_connector_stmt : OUT ID COMMA NUMBER commentary
                   | OUT NUMBER commentary

t_decision : case_stmt t_decision_branch_first bend_stmt

case_stmt : CASE text commentary

bend_stmt : BEND commentary

t_decision_branch_first : of_stmt t_series_of_actions
                          t_decision_branch
                        | of_stmt t_series_of_actions
```

```
t_decision_branch : of_stmt t_series_of_actions t_decision_branch
                  | of_stmt t_series_of_actions

t_decision_branch : division_stmt of_stmt t_series_of_actions
                    t_decision_branch
                  | division_stmt of_stmt t_series_of_actions

of_stmt : OF text commentary

relocation_stmt : OUT

x_procedure_call_stmt : XPROC ID commentary
                      | XPROC ID COMMA NUMBER commentary
                      | XPROC NUMBER commentary

event_stmt : event_keyword messages brief_explanation commentary
           | event_keyword ELSE brief_explanation commentary

event_keyword : EVENT
              | EVENTE
              | EVENTI

messages : ID COMMA messages
         | ID

division_stmt : FRAC commentary

brief_explanation : DOUBLECOMMA text
                  | empty

leader_commentary : SEMICOLON TEXT leader_commentary
                  | SEMICOLON leader_commentary
                  | empty

commentary : SEMICOLON TEXT commentary
           | SEMICOLON commentary
           | empty

text : TEXT text
     | TEXT

empty :
```

Příloha B

TEST CASE	SPACE	NOSPACE	TEST CASE	SPACE	NOSPACE
test000	OK		test050	OK	
test001			test051		
test002			test052	OK	
test003			test053		
test004			test054	+	
test005			test055	OK	
test006			test056		
test007			test057	OK	
test008	OK		test058	OK	
test009			test060	OK	
test010	OK		test061	OK	
test011	OK		test062	OK	
test012	OK		test063	OK	
test013	OK		test064	OK	
test014	OK		test065	OK	
test015	FAILED	FAILED	test066	OK	
test016			test067	OK	
test017	OK		test068	OK	
test018	OK		test069	OK	
test019	OK		test070	FAILED	OK
test020	OK		test071	FAILED	FAILED
test021	OK		test072	!	
test022	FAILED	FAILED	test073	OK	
test023	FAILED	OK	test074	OK	
test024	OK		test075	OK	
test025	!		test076	!	
test026	!		test077	OK	
test027	FAILED	FAILED	test078	OK	
test028	+		test079	!	
test029	!		test080	OK	
test030	!		test081	FAILED	FAILED
test031	!		test082	FAILED	OK
test032	!		test083	FAILED	OK
test033	FAILED	FAILED	test084	FAILED	OK
test034	OK		test085	!	
test035	!		test086	OK	
test036			test087	OK	
test038	FAILED	FAILED	test088	FAILED	OK
test039	OK		test089	OK	
test040	OK		test090	FAILED	OK
test041	OK		test091	!	
test042	OK		test092	FAILED	OK
test043			test093	FAILED	OK
test044			test094	FAILED	OK
test045			test095	OK	
test046			test096	OK	
test047			test097	OK	
test048	OK		test098	OK	
test049	OK		test099	!	

test100	!		test153	OK	
test101	OK		test154	OK	
test102	OK		test155	OK	
test103	OK		test156	OK	
test104	OK		test157	OK	
test105	OK		test158	!	
test106	OK		test159	OK	
test107	OK		test160	OK	
test108	OK		test161	OK	
test109	OK		test162	OK	
test110	OK		test163	OK	
test111	OK		test164	OK	
test112	FAILED	FAILED	test165	OK	
test113	OK		test166	OK	
test114	OK		test167	OK	
test115	OK		test168	OK	
test116	OK		test169	OK	
test117			test170	FAILED	OK
test118	OK		test171	FAILED	OK
test119	OK		test172	OK	
test120	FAILED	FAILED	test173	OK	
test121	FAILED	OK	test174	OK	
test122	OK		test175	OK	
test123	OK		test176	OK	
test124	OK		test177	!	
test125	OK		test178	!	
test126	OK		test179		
test127	!		test180	OK	
test128	!		test181	OK	
test129	!		test182	FAILED	FAILED
test130	!		test183	OK	
test131			test184	OK	
test132	!		test185	OK	
test133	!		test186	OK	
test134	!		test187	OK	
test135	OK		test188		
test136	!		test189	OK	
test137	!		test190		
test138	!		test191	OK	
test139	OK		test192	OK	
test140	OK		test193	OK	
test141			test194	OK	
test142	OK		test195	OK	
test143	!		test196	OK	
test144	OK		test197	OK	
test145	OK		test198	OK	
test146	OK		test199	+	
test147	OK		test200	OK	
test148	OK		test201	OK	
test149	FAILED	FAILED	test202	OK	
test150	OK		test203	OK	
test151	OK		test204	OK	
test152	OK		test205	OK	

test206	+		test259	FAILED	OK
test207	OK		test260	FAILED	OK
test208	OK		test261	OK	
test209	OK		test262	OK	
test210	OK		test263	OK	
test211	OK		test264	OK	
test212	OK		test265	OK	
test213	OK		test266	OK	
test214	OK		test267	OK	
test215	OK		test268	OK	
test216	OK		test269	OK	
test217	OK		test270	+	
test218	OK		test271	OK	
test219	OK		test273	OK	
test220	OK		test274	FAILED	OK
test221	OK		test275	FAILED	OK
test222			test276	FAILED	OK
test223	OK		test277	FAILED	OK
test224	OK		test278	OK	
test225	OK		test279	OK	
test226	OK		test280	FAILED	FAILED
test227	OK		test281	OK	
test228	FAILED	OK	test282	FAILED	FAILED
test229	FAILED	OK	test283	FAILED	FAILED
test230	FAILED	OK	test284	OK	
test231	FAILED	OK	test285	OK	
test232	OK		test286	OK	
test233	OK		test287	OK	
test234	OK		test288	OK	
test235	OK		test289	OK	
test236	OK		test290	OK	
test237	OK		test291	OK	
test238	FAILED	OK	test292	FAILED	OK
test239	FAILED	FAILED	test293	FAILED	FAILED
test240	OK		test294	OK	
test241	OK		test295	OK	
test242	FAILED	FAILED	test296	FAILED	FAILED
test243	OK		test297	OK	
test244	OK		test298	OK	
test245	OK		test299	FAILED	OK
test246	OK		test300	!	
test247	OK		test301	!	
test248	OK		test302	OK	
test249	FAILED	OK	test303	OK	
test250	OK		test304		
test251	OK		test305	FAILED	OK
test252	OK		test306	FAILED	OK
test253	OK		test307	FAILED	OK
test254	OK		test308	FAILED	OK
test255	OK		test922	FAILED	OK
test256	OK		test923	FAILED	OK
test257	OK		test924	FAILED	OK
test258	FAILED	OK	test925	FAILED	OK

```
test926         +                   test963         FAILED    OK
test927         FAILED    OK        test964         FAILED    OK
test928         FAILED    OK        test965         FAILED    OK
test929         FAILED    OK        test966         FAILED    FAILED
test930         FAILED    OK        test967         FAILED    OK
test931         FAILED    OK        test968         FAILED    OK
test932         FAILED    OK        test969         FAILED    OK
test933         FAILED    OK        test970         FAILED    OK
test934         FAILED    OK        test971         FAILED    OK
test935         FAILED    OK        test972         FAILED    OK
test936         FAILED    OK        test973         FAILED    OK
test937         FAILED    OK        test974         FAILED    OK
test938         FAILED    OK        test975         FAILED    OK
test939         FAILED    OK        test976         +
test940         FAILED    OK        test977
test941         FAILED    OK        test978         FAILED    FAILED
test942         FAILED    OK        test979         FAILED    OK
test943         FAILED    OK        test980         FAILED    OK
test944         FAILED    OK        test981         FAILED    OK
test945         FAILED    OK        test982         FAILED    OK
test946         FAILED    OK        test983         FAILED    OK
test947         FAILED    OK        test984         FAILED    OK
test948         FAILED    OK        test985         FAILED    OK
test949         +                   test986         FAILED    OK
test950         FAILED    OK        test987         FAILED    OK
test951         FAILED    OK        test988         FAILED    FAILED
test952         FAILED    OK        test989         FAILED    OK
test953         FAILED    OK        test990         FAILED    FAILED
test954         FAILED    OK        test991         FAILED    OK
test955         FAILED    OK        test992         FAILED    OK
test956         FAILED    OK        test993         FAILED    OK
test957         FAILED    OK        test994         FAILED    FAILED
test958         FAILED    OK        test995         FAILED    OK
test959         FAILED    FAILED    test996         FAILED    OK
test960         FAILED    OK        test997         FAILED    FAILED
test961         FAILED    OK        test998         FAILED    FAILED
test962         FAILED    OK

Number of tests:        383
Number of tests OK:     292
Number of tests FAILED: 26
```