Formální jazyky a překladače - cvičení 5

Zadání semestrálních prací

Do stávajícího překladače PL/0 doplňte následující. Při řešení úkolů není třeba zavádět nové instrukce, vše jde vyřešit se stávajícími. Výjimky, kdy se nová instrukce zavést musí, jsou výslovně uvedena v zadání.

1. příkazy pro formátovaný vstup a výstup - READ a WRITE
   příkazy READ a WRITE akceptují libovolný počet parametrů, např. READ(a, b, c) nebo WRITE(a, b)
   příkaz WRITE umožňuje formátovaný výstup: WRITE(a, b:7, c:06) vypíše proměnnou a normálně, proměnnou b alespoň sedmi znaky (zarovnáno doprava mezerami), proměnnou c alespoň šesti znaky (zarovnáno doprava nulami)
   některé parametry příkazu WRITE mohou být i řetězce, např. WRITE("promenna a", a:2, "promenna b", b:05)
   k vyřešení úlohy rozšiřte interpret o instrukce REA 0, 0 (vloží na vrchol zásobníku číslo načtené z klávesnice) a WRI 0, 0 (vypíše na terminál znak, který odpovídá ASCII hodnotě na vrcholu zásobníku)
   uvědomte si, že instrukce WRI umí vypsat pouze ASCII hodnotu; pro výpis integer hodnoty tedy musíte použít nějaký trik, ne zavádět novou instrukci!
2. cyklus FOR i:=výraz TO|DOWNTO výraz DO příkaz
   dejte pozor na to, že výrazem může být libovolný aritmetický výraz, příkazem libovolný příkaz včetně bloku BEGIN-END, cyklu FOR apod.
3. cyklus FOR i:=výraz TO výraz STEP výraz DO příkaz
   dejte pozor na to, že výrazem může být libovolný aritmetický výraz, příkazem libovolný příkaz včetně bloku BEGIN-END, cyklu FOR apod.
4. podmíněný příkaz IF podmínka THEN příkaz ELSE příkaz
   dejte pozor na to, že příkazem může být libovolný příkaz včetně bloku BEGIN-END, podmínky IF apod.
   umožněte i neúplný podmíněný příkaz, tj. konstrukci IF ... THEN ... (tedy bez části ELSE)
   správně ošetřete i konstrukci IF ... THEN IF ... THEN ... ELSE ...
5. příkaz rozvětvení (CASE nebo SWITCH)
   příklad použití:

```
6.  CASE a OF
7.    1:       příkaz1;
8.             příkaz2;
9.             BREAK;
10.   4:       příkaz3;
11.   2..5:    příkaz4;
12.            BREAK;
13.   DEFAULT: příkaz5;
14. END
```

   je-li hodnota proměnné a rovna 1, provedou se příkazy 1 a 2, při hodnotách 2, 3, 5 příkaz 4, při hodnotě 4 příkazy 3 a 4; v ostatních případech se provede pouze příkaz 5

15. cyklus REPEAT ... UNTIL podmínka
    mezi klíčovými slovy REPEAT a UNTIL může být libovolné množství příkazů včetně cyklu REPEAT-UNTIL
16. cyklus DO příkaz WHILE podmínka
    mezi klíčovými slovy DO a WHILE musí být právě jeden příkaz, má-li jich tam být více, musí být uzavřeny v bloku begin-end.
17. příkaz skoku GOTO
    příkaz GOTO umožňuje skok na návěští, např.

```
18.  ...
19.  GOTO navesti;
20.  ...
21.  navesti:
22.  ...
23.  GOTO navesti;
24.  ...
```

    všechna návěští je nutno před začátkem bloku (tj. společně s proměnnými) deklarovat konstrukcí LABEL návěští1, návěští2, ...
    návěští jsou lokální, tj. v procedurách proc1 a proc2 mohou být deklarována (a používána) návěští se stejným jménem
    skok může být nelokální (tj. lze skočit z procedury do zpět do volající procedury). V případě nelokálního skoku zajistěte, aby byl zásobník v korektním stavu!
    Nezapomeňte na to, že nelokální skok může například skákat z rekurzivně zavolané procedury zpět do hlavního programu

25. podmíněné přiřazení I := IF podmínka THEN výraz ELSE výraz
    konstrukci na pravé straně přiřazení považujte za výraz; tím pádem je možná i konstrukce např.

```
26.  i := IF a<5
27.         THEN IF b<5 THEN 1 ELSE a+b
28.         ELSE 3+(4* IF b<6 THEN 1 ELSE 2);
```

29. násobné přiřazení jako v C, např. A := B := C := D
    uvažujte i případ, kdy je posledním členem v přiřazení výraz, např. a:=b:=c+5
    dejte pozor na to, že příkaz nemusí být nutně v jedné řádce, tj. je možné i

```
30.  a := b
31.    := c :=
32.       d :=
33.            e+f;
```

    při tvorbě cílového kódu se pokuste o mírnou optimalizaci, nezapomeňte na to, že instrukce LOD a STO jsou náročné na provedení

34. procedury s parametry předávanými hodnotou
    zachovejte i možnost definování procedur bez parametrů, při volání procedur kontrolujte správný počet parametrů

dejte pozor na to, že parametry se v rámci procedury chovají jako lokální proměnné; překladač musí zareagovat na to, pokoušíme-li se v proceduře definovat proměnnou stejného jména, jako jeden z parametrů

procedury stejného jména, které se liší počtem parametrů, buď zakažte (lehčí varianta), nebo korektně vyřešte (složitější varianta, bude hodnocena malým plus)

příklad:

```
35.  VAR a;
36.  PROCEDURE proc1;
37.  BEGIN
38.   ...
39.  END;
40.  PROCEDURE proc2(a, b);
41.  BEGIN
42.   ...
43.  END;
44.  BEGIN
45.   a := 4;
46.   CALL proc1;
47.   CALL proc2(1, a+3*(a-6));
48.  END.
```

49. procedury s parametry předávanými odkazem (resp. INOUT)

podrobnosti jsou obdobné jako u předchozího zadání, procedur s parametry předávanými hodnotou

na konci příkladu bude mít proměnná a hodnotu 2:

```
50.  VAR a;
51.  PROCEDURE proc(x);
52.  BEGIN
53.   x := x+1;
54.  END;
55.  BEGIN
56.   a := 1;
57.   CALL proc(a);
58.  END.
```

zamyslete se také nad tím, jak by měl fungovat následující program a diskutujte možnosti implementace:

```
VAR a;
PROCEDURE proc(x, y);
BEGIN
 x := 1;
 y := 2;
END;
BEGIN
 CALL proc(a, a);
END.
```

59. funkce s parametry předávanými hodnotou

podrobnosti jsou obdobné jako u zadání procedur s parametry předávanými hodnotou

příklad:

```
60.  VAR x;
61.  FUNCTION f(a, b);
62.  BEGIN
63.   IF a<b THEN RETURN a*a - b*b;
64.   RETURN b*b-a*a;
65.  END;
```

```
66.  BEGIN
67.   x := 5;
68.   x := f(3, x*(4-x+f(2, 3)));
69.  END.
```

70. funkce s parametry předávanými odkazem
    podrobnosti jsou obdobné jako u zadání funkcí s parametry předávanými hodnotou a procedur s parametry předávanými odkazem
71. inkrementace/dekrementace jako v C, tj. I++, ++I, I--, --I
    tuto konstrukci považujte za výraz, čili je možná konstrukce např.

```
72.  a := a + (2*a++) + (a++ + ++a);
```

73. Předělejte v PL/0 implementaci operátorů <, > atd. tak, aby byly běžnými aritmetickými operátory, které vracejí celé číslo, a daly se tedy používat v běžných aritmetických výrazech, podobně jako v C. Dále přepracujte příkazy if a while, aby reagovaly na nulovou/nenulovou hodnotu podobně jako v C. Navíc implementujte nové binární operátory AND, OR a unární operátor NOT, které budou pracovat s celočíselnými argumenty.
    Příklad:

```
74.  var a, b;
75.  begin
76.    a := 2;
77.    b := a < 3;
78.    b := b OR (a > 5  AND NOT b);
79.    if b OR (a>1) then a := 3;
80.  end.
```

81. Změňte typ všech proměnných v PL/0 na "fixed point". Jedna proměnná nyní bude v zásobníku zabírat dvě pozice, blíže k vrcholu zásobníku bude uložena desetinná část. Číslo 1.0 se tedy uloží do zásobníku instrukcemi LIT 0, 1; LIT 0, 0. Implementace desetinné části je na uvážení autora.
    Bude zapotřebí předělat aritmetické operátory, porovnávací operátory a přiřazení.
    Do interpretu jazyka PL/0 nezasahujte, vše jde vyřešit stávajícími instrukcemi.
82. ******** datový typ BOOLEAN a výrazy s operátory OR, AND, NOT
    přepracujte konstrukci VAR a tabulku symbolů tak, aby uchovávala typ proměnné pro typovou kontrolu; překladač bude na nesoulad typů reagovat chybovým hlášením
    přepracujte podmíněné příkazy tak, aby akceptovaly pravdivostní hodnotu, tj. např. IF bool_hodnota THEN ...
    dodefinujte přiřazovací příkaz, aby fungoval i pro boolovské výrazy, např. a:= i<3; v boolovských výrazech umožněte logické operátory, tj. např.

```
83.  a := FALSE;
84.  a := i>3 AND i<10;
85.  IF i+j > 10 THEN a:=1;
86.  IF i>3 OR (j>5 AND NOT j>10) THEN a:=2;
```

explicitní typovou konverzi nemusíte nutně řešit, ale její vypracování bude hodoceno významným plus, např.

```
VAR b:boolean;
    i:integer;
BEGIN
 b := TRUE;
 i := (integer) b;
 b := (boolean) i;
 i := (integer) i > 5;
END.
```

87. ******** typ RATIO pro práci se zlomky
    přepracujte konstrukci VAR a tabulku symbolů tak, aby umožňovaly kontrolu typů; překladač bude na nesoulad typů reagovat implicitní typovou konverzí
    zachovejte stávající implementaci zásobníku; hodnoty typu RATIO řešte tak, aby zaplňovaly dvě buňky zásobníku (čitatel, jmenovatel)
    do interpretu nezavádějte nové instrukce, aritmetické operace a převody řešte pomocí podprogramů
    hodnoty zlomků se budou zadávat jako {čitatel}{jmenovatel}, např. {3+6*2}{5};
    čitatele a jmenovatele považujte za celá čísla Příklad:

```
88.  var i1: integer;
89.      r1, r2: ratio;
90.  begin
91.   i1 := 2;
92.   r1 := {1}{i1};
93.   r2 := 2 * r1;
94.   r2 := r2 + 3*r1 + {1+i1}{i1};
95.   i1 := r1;                        /* celociselne deleni, i1 = 0 */
96.   r2 := {1 + {1}{2}}{2};           /* celociselne deleni, r1 = {1}{2}
   cili 0.5 */
97.  end.
```

98. ******** typ REAL s aritmetickými operacemi, mix s INTEGER implicitní konverzí (zaokrouhlováním)
    přepracujte konstrukci VAR a tabulku symbolů tak, aby umožňovaly kontrolu typů; překladač bude na nesoulad typů reagovat implicitní typovou konverzí
    pokuste se zachovat stávající zásobník; hodnoty typu REAL řešte tak, aby zaplňovaly dvě buňky zásobníku
    do interpretu implementujte instrukce pro práci s desetinnými čísly, které jsou protějškem stávajících instrukcí; navíc zaveďte instrukce ITR a RTI pro konverzi INTEGER<->REAL
    příklad:

```
99.  VAR i:integer;
100.     a, b:real;
101. BEGIN
102.  i := 3;
103.  a := 3;
104.  b := 3.5;
105.  i := a + b*i;
106.  i := 3.5;
107. END.
```

108. ******** typ REAL s aritmetickými operacemi, mix s INTEGER explicitní konverzí
    podrobnosti jsou obdobné jako u předchozího zadání, typu REAL s implicitní konverzí
    při nesouladu typů bude překladač reagovat chybovou hláškou
    příklad:

```
109. VAR i:integer;
110.     a, b:real;
111. BEGIN
112.  i := 3;
113.  a := (real) 3;
114.  b := 3.;
115.  i := (integer) (a + b*(real)i);
116.  i := (integer) 3.5;
117. END.
```

118. ******** typ ukazatel, práce s dynamickou pamětí (NEW, DELETE, &, ^)

přepracujte konstrukci VAR a tabulku symbolů tak, aby umožňovaly kontrolu typů

hromadu (heap) zaveďte v paměti pro zásobník, heap bude narůstat z opačné strany; interpret si musí hlídat přetečení, množství volného místa apod.

pro řešení této úlohy budete pravděpodobně potřebovat zavést nové instrukce - NEW 0 0 (rezervace paměti), DEL 0 0 (uvolnění paměti), PLD 0 0 (načtení hodnoty z adresy), PST 0 0 (uložení hodnoty na adresu); samotná adresa není parametrem instrukce, ale je také uložena v zásobníku (zdůvodněte!)

pointerovou aritmetiku není nutné řešit, ale její implementace bude hodnocena významným plus

velmi významným plus by byl garbage collector :o)

příklad:

```
119. VAR p, q:^integer;
120.     i:integer;
121. BEGIN
122.  i  := 1;
123.  p  := &i;
124.  p^ := 2;
125.  p  := NEW(integer);
126.  p^ := 3;
127.  q  := p;
128.  q^ := 1 + p^;
129.  DELETE(p);
130. END.
```

131. ******** typ vícerozměrné pole s konstantními mezemi

přepracujte konstrukci VAR a tabulku symbolů tak, aby umožňovaly kontrolu typů; na nesoulad typů reaguje překladač chybovou hláškou

obsah pole nechť je uložen v zásobníku podobně jako jiné proměnné; pole bude pochopitelně zabírat několik buněk zásobníku

implementujte jednoduchý přiřazovací příkaz, který bude kopírovat obsah pole (a := b)

k vyřešení úlohy budete pravděpodobně potřebovat zavést nepřímé adresování - instrukce PLD 0 0 (načtení proměnné) a PST 0 0 (uložení proměnné), adresa L A (podobně jako u instrukcí LOD a STO) nechť je uložena jako dvě čísla na vrcholu zásobníku

příklad:

```
132. TYPE pole  = array[1..5] of integer;
133.      pole2 = array[1..3] of array[2..4] of pole;
134. VAR a:pole2;
135.     b:array[1..5] of integer;
136.     c1, c2:pole;
137.     d:array[0..3, 1..4, 0..1] of integer;
138. BEGIN
139.  b[1] := 1;
140.  b[0] := 1;    /* RUNTIME ERROR! */
141.  a[1][2][1] := 1;
142.  d[0, 1, 1] := 2;
143.  c1 := c2;
144.  b := c1;      /* ERROR - typ pole versus nepojmenovany typ */
145.  a[1][2] := c1;
146. END.
```

147. ******** typ záznam (RECORD)

přepracujte konstrukci VAR a tabulku symbolů tak, aby umožňovaly kontrolu typů; na nesoulad typů reaguje překladač chybovou hláškou

obsah záznamu nechť je uložen v zásobníku podobně jako jiné proměnné; záznam bude pochopitelně zabírat několik buněk zásobníku

implementujte jednoduchý přiřazovací příkaz, který bude kopírovat obsah záznamu (a := b)
příklad:

```
TYPE pozice = RECORD OF
               x, y: integer;
              END;
      bod = RECORD OF
               p: pozice;
               hodnota:integer;
            END;
VAR p1, p2: pozice;
    b1, b2: bod;
    p: RECORD OF x, y: integer; END;  /* OK - p je sice jmeno polozky v nejakem zaznamu, ale to nevadi */
    bod: integer;                     /* ERROR - bod je definovany typ */
BEGIN
 p1.x := 1; p1.y := 2;
 p2 := p1;
 p  := p1;   /* ERROR - nepojmenovany typ versus typ pozice */
 b1.p.x := 3;
 b2.p := p1;
 b1.p := b2.p;
 b1.hodnota := p1.x + p1.y;
END.
```

******** úprava překladače, aby tvořil nativní kód/assembler pro H8S
v tomto zadání je samozřejmě nutné přepracovat kompletní instrukční sadu překladače PL/0

******** úprava překladače, aby tvořil nativní kód/assembler pro Intel
v tomto zadání je samozřejmě nutné přepracovat kompletní instrukční sadu překladače PL/0

******** úprava překladače, aby tvořil assembler hypotetického procesoru definovaného v zadání 31
v tomto zadání je samozřejmě nutné přepracovat kompletní instrukční sadu překladače PL/0

Kromě rozšiřování překladače PL/0 jsou dispozici další zadání:

28. Napište knihovnu, která bude implementovat rozšířené načítání číselných hodnot. Uživatel bude potom mít možnost napsat číslo i aritmetickým výrazem.
    příklad možného použití:

```
fgets(vyraz, delka_radky, stdin);
i = zpracuj_vyraz(vyraz);
```

Pro vylepšení práce rozšiřte funkčnost na práci s proměnnými. Pak by použití mohlo vypadat například takto:

```
inicializuj_interpret();
do {
  fgets(vyraz, delka_radky, stdin);
  i = zpracuj_vyraz(vyraz);
} while (vyraz[strlen(vyraz)-1] == ';')
```

Pak by uživatel na výzvu "Zadej hodnotu" mohl vložit třeba sekvenci

```
a = 1;
b = 2;
a*a + b*b - 10
```

Syntaktickou analýzu proveďte metodou rekurzivního sestupu, výraz průběžně vyhodnocujte při zpracování.

31. Implementujte zadání 28, ale syntaktickou analýzu provádějte pomocí zásobníkového automatu pro LL gramatiku.
32. Implementujte zadání 28, ale syntaktickou analýzu provádějte pomocí zásobníkového automatu pro LR gramatiku.
33. ******** Implementujte jednoduchý jazyk, který umožňuje práci s proměnnými, přiřazovací příkaz, podmíněný příkaz, příkaz cyklu a procedury bez parametrů. Je dán následující gramatikou:

```
34. program   -> var procedura { var procedura }
35. var       -> {int identifikátor {, identifikátor} ; }
36. procedura -> void identifikátor ( ) blok ;
37. blok      -> begin var { příkaz ; } end
38. příkaz    -> identifikátor := výraz |
39.              if výraz then příkaz |
40.              while výraz do příkaz |
41.              identifikátor ( ) |
42.              blok |
43.              e
44. výraz     -> člen [ [ == | != | < | > | <= | >= ] člen | e ]
45. člen      -> [ + | - | e ] term { [ + | - ] term }
46. term      -> faktor { [ * | / ] faktor }
47. faktor    -> identifikátor | číslo | ( výraz )
```

Jedna z procedur se musí jmenovat main.
Inspirujte se implementací jazyka PL/0 a metodou rekurzivního sestupu přeložte zdrojový text programu do formy spustitelného kódu pro virtuální stroj s následujícími parametry:

- pro program paměť programu a registr PC (program counter, ukazuje na instrukci, která se má vykonat)
- pro data zásobník, registry SP (stack pointer, ukazuje na první volnou pozici zásobníku) a BP (base pointer, offset pro některé instrukce)
- instrukce
  - PUSHN <číslo> - uloží do zásobníku <číslo> a zvýší SP o 1; `[SP] := <číslo>; SP++`
  - PUSHA <číslo> - uloží do zásobníku obsah adresy <číslo> a zvýší SP o 1
  - PUSHB <číslo> - uloží do zásobníku obsah z adresy (BP + <číslo>) a zvýší SP o 1
  - PUSHR <registr> - uloží do zásobníku obsah registru <registr> a zvýší SP o 1; registr může být SP, BP nebo PC
  - POPA <číslo> - uloží na adresu <číslo> hodnotu z vrcholu zásobníku a sníží SP o 1

- POPB <číslo> - uloží na adresu (BP+<číslo>) hodnotu z vrcholu zásobníku a sníží SP o 1
- POPR <registr> - uloží do registru <registr> hodnotu z vrcholu zásobníku a sníží SP o 1; registr může být SP, BP nebo PC
- ADD - sečte dvě čísla na vrcholu zásobníku a součet uloží místo nich, ve finále se SP tedy sníží o 1;`[SP-2] := [SP-2] + [SP-1]; SP--`
- SUB - rozdl, `[SP-2] := [SP-2] - [SP-1]; SP--`
- MUL, DIV, MOD - analogicky k ADD a SUB
- LT - je-li [SP-2] < [SP-1], odebere operandy a na zásobník vloží 1 (jako true) nebo 0 (jako false), ve finále se SP tedy sníží o 1; `[SP-2] := [SP-2] < [SP-1] ? 1 : 0; SP--`
- GT, LEQ, GEQ, EQ, NEQ - analogicky k LT
- NEG - hodnotě vrcholu zásobníku změní znaménko, hodnota SP se nemění; `[SP-1] := -[SP-1]`
- NOT - logická negace, hodnota SP ne nemění; `[SP-1] := [SP-1] == 0 ? 1 : 0`
- JMP <číslo> - nepodmíněný skok; `PC := <číslo>`
- JT <číslo> - podmíněný skok, je-li na vrcholu zásobníku true (nenulová hodnota); hodnota SP se sníží o 1; `PC := [SP-1] == 0 ? PC+1 : <číslo>; SP--`
- JF <číslo> - podmíněný skok, je-li na vrcholu zásobníku false (nulová hodnota); analogicky k JT
- CAL <číslo> - nepodmíněný skok do podprogramu na adrese <číslo>, do zásobníku se uloží návratová hodnota; `[SP] := PC+1; SP++; PC := <číslo>`
- RET - návrat z podprogramu; provede-li se návrat na hodnotu -1, vrátí se řízení operačnímu systému; `PC := [SP-1]; SP--`
- ADS <číslo> - zvýší obsah SP o <číslo>; `SP := SP + <číslo>`
- na začátku práce jsou BP a PC rovny 0, SP rovno 1, v zásobníku je hodnota -1

Pro ilustraci se podívejte na ukázku zdrojového textu programu a jeho překladu do instrukcí.

48. ******** Napište základní interpret jazyka LISP.
49. ******** Napište program, který převede regulární výraz na deterministický konečný automat.
    Regulární výraz bude zadaný např. (a(ahoj|nazdar)*bcd*)*, což znamená znak a, libovolné množství řetězců ahoj či nazdar, znaky b, c, libovolné množství znaků d; tato sekvence se může libovolněkrát opakovat. Prázdné slovo budeme označovat znakem @. Má-li se v regulárním výrazu vyskytnout některý ze znaků @, (,), |, * nebo \ ve smyslu skutečného vstupu, musí být zapsán escape-sekvencí \@, \(, \), \|, * nebo \\. Vstupem nechť je standardní vstup (stdin), výstupem standardní výstup (stdout). Výstupní automat bude zapsán ve formě tabulky ve stylu

```
50.  stav vstup:stav vstup:stav ...
51.  stav vstup:stav vstup:stav ...
52.  ...
```

    kde `stav` bude celé číslo a vstup jeden znak, který se může objevit na vstupu automatu (předpokládáme ASCII, žádné záludnosti). Viz též zadání 32.

53. ******** Napište program, který převede nedeterministický konečný automat na regulární výraz.
    Vstupem nechť je standardní vstup (stdin), výstupem standardní výstup (stdout). Vstupní automat bude zapsán ve formě tabulky ve stylu stejném jako v zadání 31, výstupní regulární výraz nechť kombinuje znaky jako atomické regulární výrazy, (.../.../...) jako výběr, (...)* jako iteraci a @ jako prázdné slovo. Pokud se na vstupu převáděného automatu mohou vyskytnout znaky @, (,), |, * nebo \ musí být tento znak na výstupu vypsán escape-sekvencí jako \@, \(, \), \|, *, \\.
54. ******** Napište program, který pro danou LL(1) bezkontextovou gramatiku sestaví rozkladovou tabulku, kde k bude vstupem programu.
    Gramatika bude zadána např.

```
55.  E : E + T | T
56.  T : T * F | F
57.  F : i | (E)
```

    velká písmena budou znamenat neterminální symboly, znaky `:` a `|` budou mít speciální použití, pro prázdný řetěz se bude používat znak `@`, všechno ostatní budou terminální symboly; vstupem nechť je standardní vstup (stdin), výstupem standardní výstup (stdout). Na výstupu se objeví zadaná gramatika s očíslovanými pravidly (číslování od 1) a rozkladová tabulka ve tvaru:

```
Gramatika:
1 neterminál : pravidlo
2 neterminál : pravidlo
...

Tabulka:
           terminál terminál ...
neterminál pravidlo pravidlo ...
neterminál pravidlo pravidlo ...
...
```

    Pokud pro nějakou kombinaci neterminál/terminál neexistuje v tabulce pravidlo, vypište do daného políčka -. Pokud pro nějakou kombinaci neterminál/terminál existuje více pravidel (čili gramatika není LL(1)), vypište do tabulky všechna taková pravidla a do výstpu napište varování, že vstupní gramatika není LL(1).
    Pro kontrolu implementujte program, který vámi vytvořenou tabulku načte a pokusí se s její pomocí akceptovat zadaný řetězec.

58. ******** Napište program, který pro danou SLR(1) bezkontextovou gramatiku sestaví rozkladovou tabulku, kde k bude vstupem programu.
    Gramatika bude zadána např.

```
59.  E : E + T | T
60.  T : T * F | F
```

61. `F : i | (E)`

    velká písmena budou znamenat neterminální symboly, znaky `:` a `|` budou mít speciální použití, pro prázdný řetěz se bude používat znak `@`, všechno ostatní budou terminální symboly; vstupem nechť je standardní vstup (stdin), výstupem standardní výstup (stdout). Na výstupu se objeví zadaná gramatika s očíslovanými pravidly (číslování od 1) a rozkladová tabulka ve tvaru:

```
Gramatika:
1 neterminál : pravidlo
2 neterminál : pravidlo
...

Tabulka:
        akce                                | symbol    symbol ...
        terminál  terminál  ...  @          |
symbol  pravidlo  pravidlo  ...  pravidlo   | pravidlo  pravidlo ...
symbol  pravidlo  pravidlo  ...  pravidlo   | pravidlo  pravidlo ...
...
```

    Pokud pro nějakou kombinaci symbol/symbol či symbol/terminál neexistuje v tabulce pravidlo, vypište do daného políčka -, pro akceptaci písmeno A a pro přijetí písmeno P. Pokud pro nějakou kombinaci symbol/symbol existuje více pravidel (čili gramatika není SLR(1)), vypište do tabulky všechna taková pravidla a do výstpu napište varování, že vstupní gramatika není SLR(1).
    Pro kontrolu implementujte program, který vámi vytvořenou tabulku načte a pokusí se s její pomocí akceptovat zadaný řetězec.

62. ******** Implementujte jazyk PL/0 pomocí nástrojů LEX/YACC.
63. ******** Doplňte v překladači PL/0 generování tzv. debuginfo - informací pro debugger, která řádka zdrojového kódu odpovídá dané části cílového kódu - a upravte některý stávající debugger tak, aby toto debuginfo dokázal využívat.

Zadání označená hvězdičkami jsou složitější. Pokud takové zadání chcete, řekněte to předem - a pak si můžete vybrat. Následně se můžete těšit z toho, že průměrně zpracované těžké zadání vám přinese lepší ohodnocení, než skvěle zpracované lehké zadání. V ostatních případech se zadání přiřadí náhodně (pomocí tajné hash funkce a hodu kostkou). Pokud se vám toto zadání nebude líbit, máte možnost opravného hodu. Ale pozor! Druhé zadání je definitivně platné a vrátit se k původnímu není možné!

Všechna zadání se mohou splnit v Pascalu, C, C# nebo Javě, volitelně je také k dispozici PHP a ADA (automaticky považováno za těžké zadání, tj. můžete si vybrat libovolné číslo úlohy i

jazyk). Jazyk "jednoduchého zadání" bude rovněž přidělen náhodně, a to současně se zadáním čísla úlohy.

Odevzdání semestrální práce

Jakmile úlohu vyřešíte a sepíšete dokumentaci, uděláte několik věcí. Vytvoříte adresář, který se bude jmenovat jako vy plus kód zadání (příjmení_jméno_zadani_jazyk, bez diakritiky, např. Fik_Maxipes_5_ADA). Do něj umístíte zdrojový kód, přeložený program spustitelný v prostředí Windows (bez virů), elektronickou formu dokumentace (ve formátu PDF) a několik příkladů (programů v PL/0, včetně výstupu překladače), které předvedou, že vše funguje, jak má. Celý adresář zabalíte (nejlépe ZIPem) a odešlete mailem na moji adresu. Poté vytisknete dokumentaci, kterou mi osobně předáte. Při předávání dokumentace se zároveň přesvědčíme, zda mail došel. Semestrálka se považuje za odevzdanou v okamžiku převzetí tištěné dokumentace.

Důležité: archiv se semestrální prací se bude jmenovat tak, jak má. Archiv bude obsahovat zdrojový kód, spustitelný program, dokumentaci, příklady a nic víc! Nemám chuť zjišťovat, jestli je debuginfo k něčemu užitečné, která verze platí a podobně. Máte-li chuť přidat do archivu další věci (nebo vyžaduje-li to zadání), připojte do archivu soubor 00README.TXT, který popíše, co je co. Archiv rozhodně nevzniká tak, že se vezme pracovní adresář, který se zabalí a odešle!

V dokumentaci bude uvedeno vaše jméno, e-mail a datum odevzdání. Dále tam bude formulace úlohy, **stručný a výstižný popis řešení** (vyjma úloh, kde se klade na dokumentaci důraz) a výpis vámi doplněného/upraveného kódu (malým stupněm písma, asi 8 pt) v rozsahu tak akorát, abych mohl (teoreticky) zkontrolovat správnost implementace i bez kompletního zdrojáku. Krom toho přidejte do dokumentace několik vzorových programů pro PL/0 (děláte-li semestrálku pro PL/0) jak ve formě zdrojového textu, tak ve formě generovaného kódu, na kterých demonstrujete, že překladač skutečně dělá, co má. Dále napíšete, zda jste se v původním překladači setkali s nějakou chybou as popíšete způsob jejího odstranění. Při psaní dokumentace mějte na paměti, že její kvalita se promítne do vašeho hodnocení, dbejte proto jak na kvalitu textu, tak na formu. Rozsah dokumentace se typicky pohybuje okolo 3-4 stran (včetně titulního listu), výpisy zdrojového textu by měly tvořit menší část rozsahu.

Zdrojový kód, který mi pošlete, musí obsahovat následující. Na začátku (každého) souboru bude v komentáři vaše jméno a datum odevzdání. Veškerý kód, který doplníte, bude okomentován. Tyto komentáře budou vypadat např. takto:

```
/* FM: tento radek je k nicemu */
```
Místo FM doplníte vlastní iniciály. Jde o to, abych příkazem
```
grep '/* FM'
```
našel všechny řádky, které jste změnili nebo přidali.

Na tomto místě budiž řečeno, že semestrální práce musí být pouze vaše dílo, nesmíte využívat práce vašich současných nebo předchozích či budoucích kolegů. Při odevzdávání práce si typicky popovídáme o principu řešení; ve FJP se dá snadno poznat, kdo si na řešení přišel sám a kdo ho jen opsal.