Demonstrační cvičení IFJ
Implementace překladače IFJ21

25. 10. 2021

Zbyněk Křivka

Obsah demonstračního cvičení

- struktura překladače
- představení IFJ21
- lexikální analýza pro IFJ21 (bílé znaky)
- kombinovaná syntaktická analýza
- víceprůchodová syntaktická analýza
- návrh tabulky symbolů
- představení IFJcode21

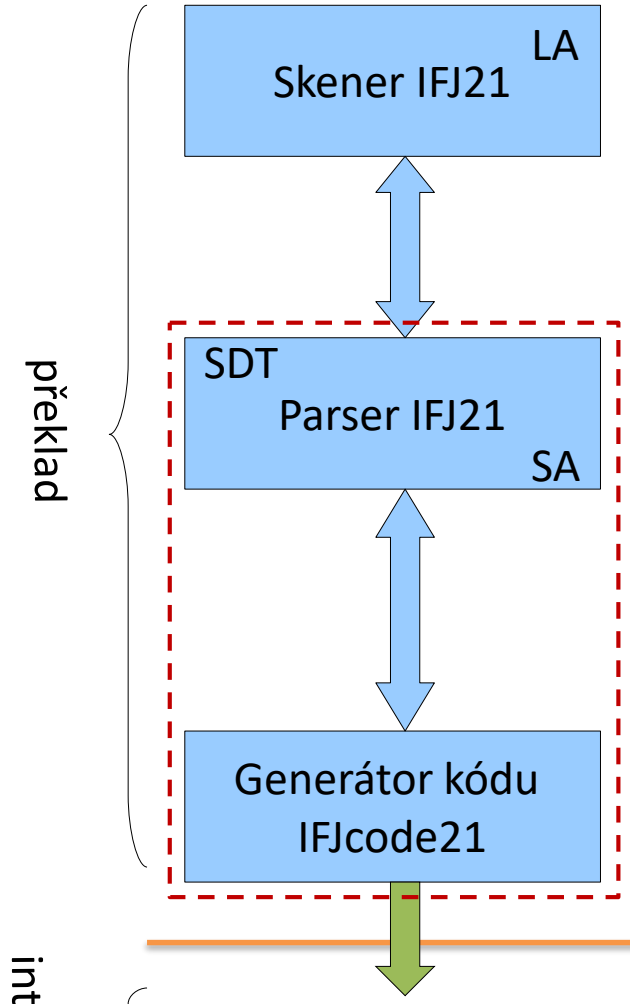
- Prohlášení: Existují i jiná správná (i lepší) řešení! Nediskutujeme bonusová rozšíření.

Poznámky

- Interpreter pro IFJcode21 zveřejněn v *Souborech* předmětu, složka *Projekt*
  - *Krátký popis na Wiki*
- /pub/courses/ifj/ic21int @ Merlin
- Verze pro 64-bitový Linux nebo Windows

- Příklady IFJ21 a IFJcode21 uhladím a též zveřejním

Schéma překladače



- Rozděluje zdrojový text na lexémy
- Jednotlivé lexémy jsou reprezentovány tokeny
- Formálním modelem je **konečný automat**

- Simuluje tvorbu derivačního stromu
- Provádí sémantické kontroly
- Generuje vnitřní reprezentaci (**3AC**, AST)
- Formální modely **LL-gramatika, precedenční analyzátor** (pro výrazy), **rekurzivní sestup**

- Převádí vnitřní reprezentaci (AST) na konkrétní instrukce IFJcode21 (průchod AST)
- Generuje výstupní textový soubor s IFJcode21

- Vykonávání instrukcí IFJcode21
- Správa hodnot proměnných, zásobník volání
- Vstupně/výstupní operace

IFJ21: Přehled

- Inspirováno jazykem Teal/Lua, vestavěné do jazyka C
- Imperativní, case-sensitive, kompilovaný (`tl run …`)
- Staticky typovaný: **integer**, **number**, **string**, **boolean**
- Vestavěné funkce: readi/n/s, write, tointeger, substr, ...
- Netradiční operátory: .. (konkatenace), # (délka řetězce)
- Uživatelské funkce (**function**), nepojmenované hlavní tělo
- Deklarace funkce: **global** fun : **function**(**integer**) : **string**
- Příkazy:
  - Definice proměnné: **local** x : **integer** = 10
  - Přiřazení: x, err = 10, "popis chyby"
  - Podmíněný příkaz (**if**-**then**-**else**-**end**), příkaz cyklu (**while-do-end**)
  - Volání funkce (i rekurzivní), příkaz návratu z funkce (**return**)

IFJ21: Hello

```
// Hello World example in IFJ21
// run it on Merlin by: tl run hello.tl ifj21.tl
require "ifj21"

function hlavni_program()
  write("Hello from IFJ21", "\n")
end
hlavni_program()
```

```
// Shorter Hello World example in IFJ21
require "ifj21"

write("Hello from IFJ21\n")
```

mohu volat s promennými
nebo literály, né výrazy

IFJ21: Faktoriál (stdin vs soubor)

```
-- Program 1: Vypocet faktorialu (iterativne)
require "ifj21"

function main() --Hlavni telo programu
  local a : integer = nil
  local vysl : integer
  write("Zadejte cislo pro vypocet faktorialu\n")
  a = readi()
  if a == nil then
      write("a je nil\n")
      return else end
  if a < 0 then
    write("Faktorial nelze spocitat\n")
  else
    vysl = 1
    while a > 0 do
      vysl = vysl * a
      a = a - 1
    end
    write("Vysledek je: ", vysl, "\n")
  end
end

main()
```

mohu to vypustit a bude to mít stejný efekt

prázdná větev else

IFJ21: Funkce

```
require "ifj21"
global g : function (integer) : integer  -- deklarace funkce

function f(x : integer) : integer
  if x > 0then return x-1
  else
    write("calling g with ", x)
    return g(x)
  end
end

function g(x : integer) : integer
  if x > 0 then
    write("calling f with ", x)
    return f(x)
  else return -200 end  -- proč musí být před end bílý znak?
end

function main()
  local res : integer = g(10)
  write(res)
end main()
```

main je zde dalsí prikaz, bude se tedy volat fce

IFJ21: Vícenásobné přiřazení

```
require "ifj21"

function foo(x : integer, y : integer) : integer, integer
  local i : integer = x
  local j : integer = (y + 2) * 3
  i, j = j+1, i+1  -- vyhodnocuj zprava a přiřazuj později
  return i, j
end

function main()
  local a : integer = 1
  local b : integer = 2
  a, b = foo(a, b)  -- returns 13, 2
  if a < b then
    print(a, "<", b, "\n")
  else
    print(a, ">=", b, "\n")
    local a : integer = 666
  end
  print(a)  --[[ prints 13 ]]
end

main()
```

IFJ21: Rozsah platnosti a viditelnost

```
require "ifj21"

function program()
  local y : integer = 20
  if 1 == 1 then
    y = 10
    write(y)
  else
    write(y)
  end
  write(y) -- vypise 10

  local i : integer = 1
  while i <= 10 do
    local i : integer = i + 5
    i = i + 1
    write(i)
    -- nekonecny cyklus (7...)
  end
end

program()
```

buřt here

IFJ21: Bílé znaky

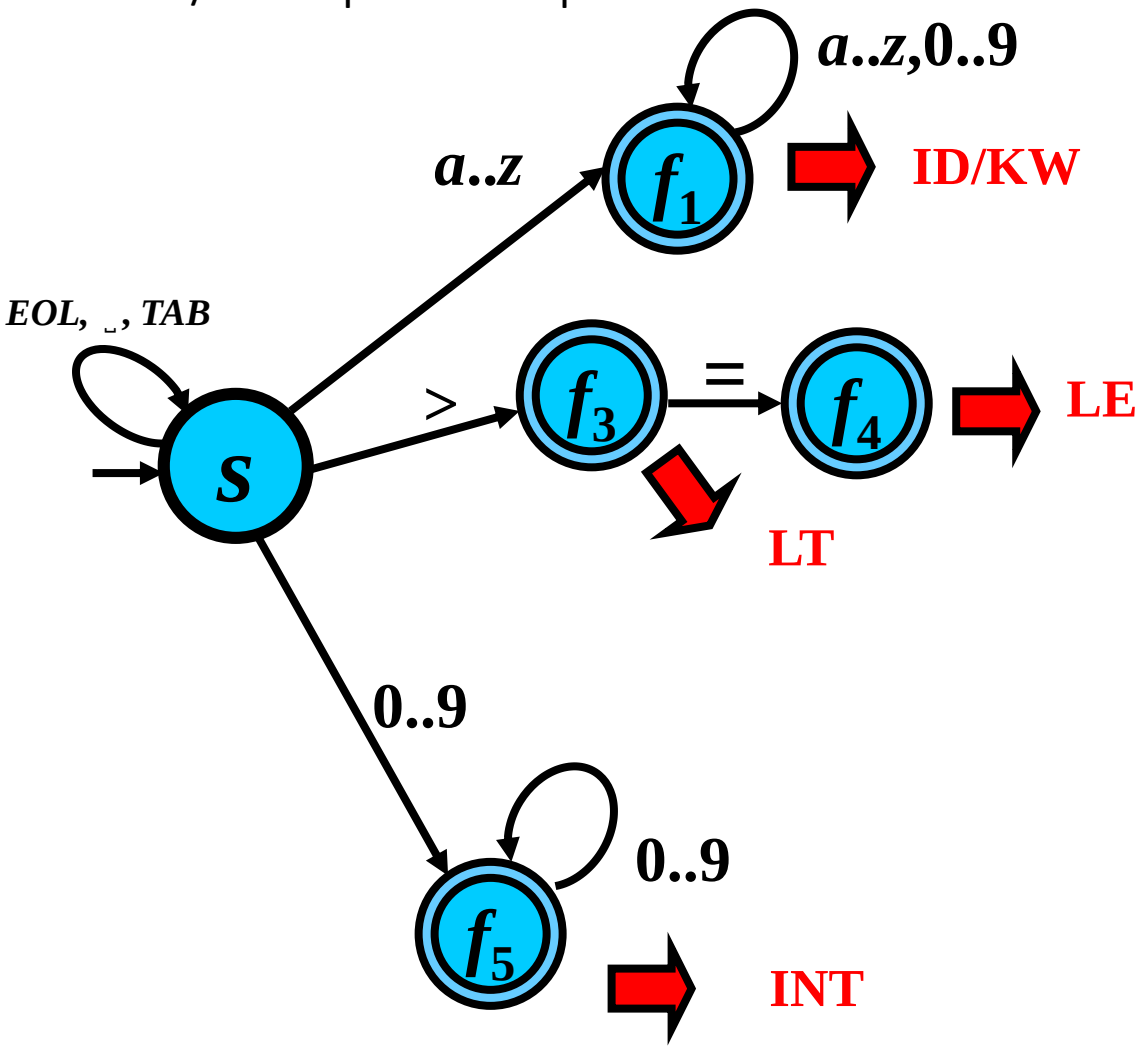
```
require "ifj21"

function whitespaces
()
local
s
:
string
=
"\x3A0"write(s, "\n")

s = "a\255b"
write(s
,
"\n")x=0
-
1write(x)
end whitespaces()
```

Lexikální analýza, jednoduché

Ilustrační/nekompletní DKA pro LA:




```
if␣i<
13then␣else
end
```

Občas nutno vrátit znak (buffer, ungetc).

Je nutné vracet bílé znaky?

Syntaktická analýza

- Doporučenou metodou je **rekurzivní sestup (RS)** v kombinaci s **precedenční syntaktickou analýzou (PSA)** pro výrazy
- Nutno zajistit korektní předávání řízení mezi oběma metodami

```
if a > 5 then…
while(i+1) < 5*5 do…
```

- Po přijetí tokenu **if** lze jednoznačně spustit **PSA**
- Pozor na problém s rozpoznáním příkazu přiřazení či volání funkce:

```
a = fce()
a = a - 1
fce()
```

jak to řešit?? kouknu do tablky symbolu a podle toho se rozhodnu

- Po přijetí tokenu '=' nevíme, jestli nenásleduje volání funkce
- Token(y) načtený navíc je nutno předat **PSA**
- Zatím jsme neřešili, jak proběhne předání řízení zpět **RS** …

Ukončení precedenční SA

```
if a > 5 then …
```

- U **then** víme, že jde o konec výrazu podmínky u **if**.

```
a = a - 1 EOL
+ 1 write(a)
```

- **IFJ21:** Výraz může končit i před neočekávaným identifikátorem – prozkoumejte další tokeny značící konec výrazu, tj. konec vstupu pro **PSA** – *virtuální* token '$'.
- **IFJ21**: Bílé znaky nejsou známkou konce výrazu.

Jedno/Dvouprůchodová analýza

```
global f : function(int) : int
function g()
    local j : integer = 0
    j = f(3)
    f(j)
end
function f(i:int):int
    return i
end
```

id zatím není v TS | vložím funkci vč. signatury, chybí tělo

v TS je deklarovaná funkce, zkontroluji typy, generuji volání této funkce

kontrola, že použitá funkce bude i definovaná

- **Jednoprůchodová** SA **s doplněním** informací později
  - Speciální seznam chybějících informací resp. akcí/kontrol k dodatečnému provedení. Např. pozdější definice.
  - Generování AST/IR pro pozdější generování instrukcí. Např. použití návěští před jeho vygenerováním do cílového programu.

- **Dvouprůchodová** SA – deklarace slouží pro vyhnutí se dvouprůchodové SA

Tabulka symbolů (TS)

- Skener ze své podstaty nepotřebuje TS.
- Předá-li Skener token, musí Parser provádět další kontroly.
- Příklad:

```
global jinaFunkce : function(string) : string
function novaFunkce(par : string)
     local prom : string = par
     prom = jinaFunkce(par)
end
```

Ve všech tučných případech rozezná Skener identifikátor.

- Má však Skener vkládat nebo vyhledávat v TS?
- Jak se chovat při duplicitě/chybějícím záznamu v TS? (definice funkce nebo volání funkce?)
- U proměnných se musí Parser starat o rozsah platnosti.

Návrh tabulky symbolů

- Co všechno je potřeba pro ověření sémantiky a generování kódu?
- Záznamy pro **proměnné** a **funkce**
- Globální TS (funkce), lokální TS pro každou funkci

```
function foo(param : integer) : integer
        local prom1 : integer = param
        //local prom1 : integer  // error: redef. variable
        return prom1 + 5
end
# Proměnné z hlavního těla -> GTS nebo LTS?
…
```

- Implementace lokální TS (dle varianty zadání a …):
  - Zásobník/seznam TS (symbol → struktura s atributy)
  - TS zásobníků (symbol → zásobník struktur s atributy)
  - TS se složeným klíčem (úroveň zanoření + symbol → struktura s atributy)
  - …

Návrh tabulky symbolů

- **Informace o identifikátorech** (jméno, druh (F/V), …)

a) **Informace o proměnných** (init?, dat. typ?, použita?)

b) **Informace o funkcích:**
   - seznam formálních parametrů a jejich typů
   - návratové typy
   - lokální TS
   - Byla již definována? Kompletní informace?
   - Interní reprezentace těla funkce (příp. návěští funkce)
   - …

A kam s hodnotami proměnných?

- Nemůžeme je všechny uložit do TS? Ne!
- Proč? Během interpretace již nebude TS k dispozici!
  - Musíme pracovat s instrukcemi IFJcode21 a jeho paměťovým modelem …

Derivační strom vs Abstraktní syntaktický strom

```
x = f(10, par)
```

- Výňatek LL gramatiky pro volání funkce (ne IFJ21):

⟨STMT⟩ → **id = id (** ⟨PL⟩ **)**

⟨PL⟩ → ε

⟨PL⟩ → ⟨TERM⟩ ⟨PL2⟩

⟨PL2⟩ → **,** ⟨TERM⟩ ⟨PL2⟩

⟨PL2⟩ → ε

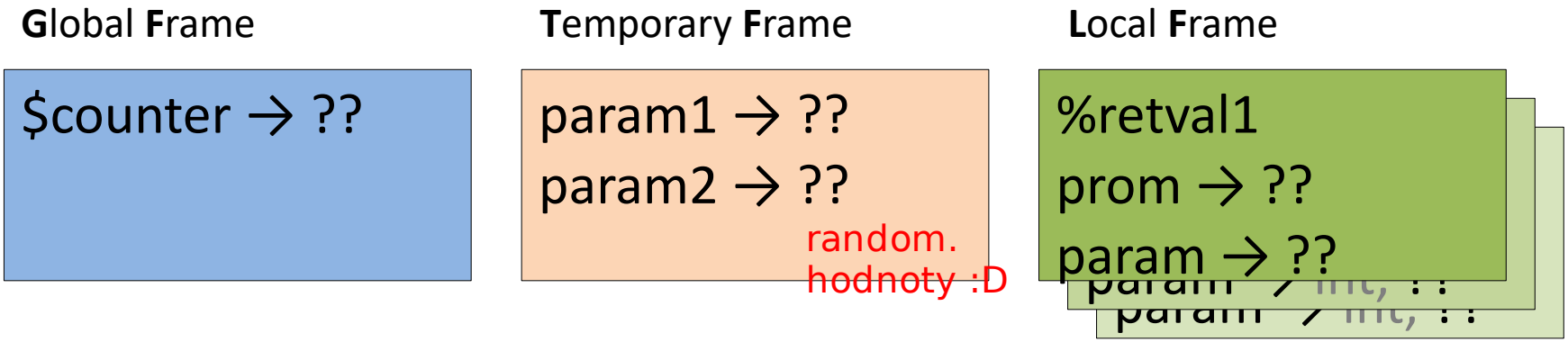
⟨TERM⟩ → **id**

⟨TERM⟩ → **int**

IFJcode21

- **IFJcode21** versus IFJ21:
  - Case-sensitive jen částečně (operandy), .IFJcode21
  - Volnější pravidla pro identifikátory (%, &, $, *, …)
  - Operandy oddělujeme **bílým znakem** (ne čárkou!)
  - IFJcode21 je dynamicky typovaný (typ dán hodnotou)
- **IFJcode21** versus typický assembler:
  - Redundantní operační kódy (3AC i zásobníkový kód)
  - Chybí registry, vysokourovňovější práce s pamětí
    - Rámce (slovníky proměnných, tj. přístup přes identifikátor)
  - Nepřístupný čítač instrukcí (PC) a zásobník volání

Paměťový model IFJcode21

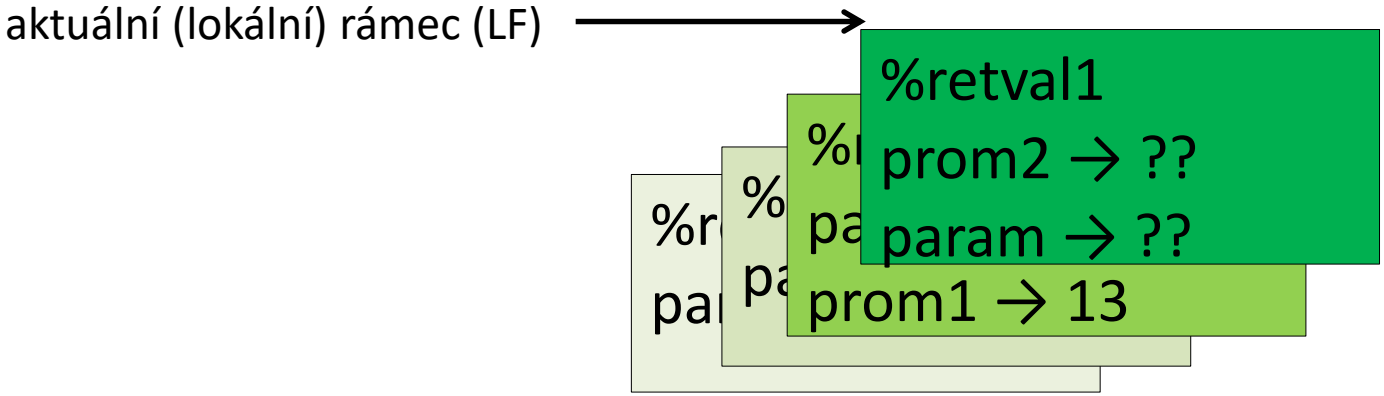


- **Rámec** = slovník proměnných (přístup přes identifikátor)
- **Záznam v rámci obsahuje**:
  - identifikaci proměnné (pozor na kolize speciálních prom.)
  - hodnotu
  - Typ proměnné? Je proměnná inicializovaná?

    ⇒ Instrukce TYPE
- **Lokální TS ⇔ Lokální rámec**
- Globální TS (bez funkcí) ⇔ Globální rámec

Jak pracovat s rámci proměnných

- Pro lokální proměnné využijeme 3 druhy rámců:
  - **zásobník lokálních rámců**



    - Rámec vytvořen při vstupu do nového rozsahu platnosti, zrušen při jeho opuštění; sdružovat rozsahy platnosti IFJ21?
  - **připravovaný a ukončený rámec** – použijeme dočasný rámec (TF)
    - vysvětlíme při popisu volání funkce

Jak pracovat s rámci proměnných

- Začneme vrcholem zásobníku rámců.
- Při volání funkce **vytvoříme nový** rámec a **umístíme jej na vrchol zásobníku** (stane se aktuálním).
- Při návratu z funkce získáme původní kontext **odstraněním rámce dokončené funkce z vrcholu zásobníku**.

- A nyní podrobněji …

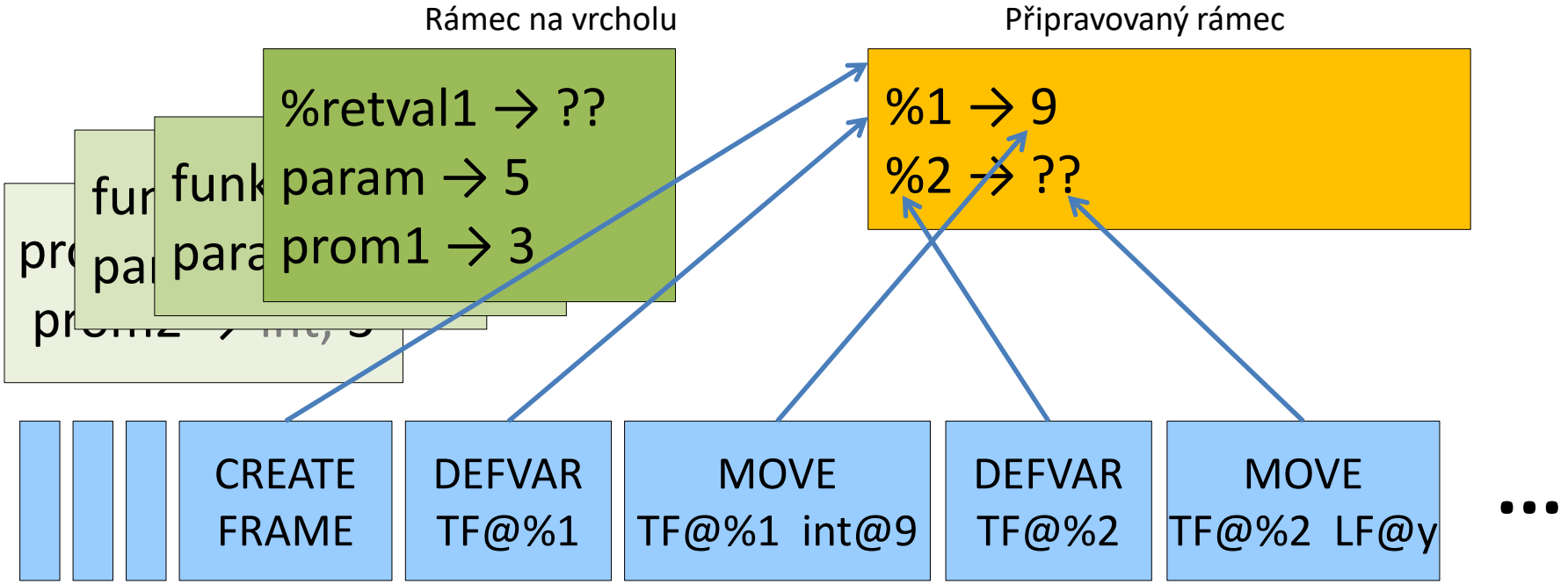
Překlad volání funkcí

```
function mojeFunkce (p1:int, p2:int) : int, int
        # vypocet
        return NavratovaHodnota1, p1+p2
end
…
        výsledek --[[,zahod]] = mojeFunkce(9, y)
…
```

- Třeba vyřešit tři problémy:
  1. Jak dostat do nového rámce parametry?
  2. Jak se vrátíme za volající instrukci?
  3. Jak získat **návratové hodnoty**?

1. Předání parametrů

- Využijeme **připravovaný rámec**
  - Vytvoříme jej jako dočasný rámec.
  - Vytvoříme proměnné pro parametry (nutně nemusíme ještě znát jejich jména; generická %1, %2, …) a do nich nakopírujeme hodnoty parametrů.



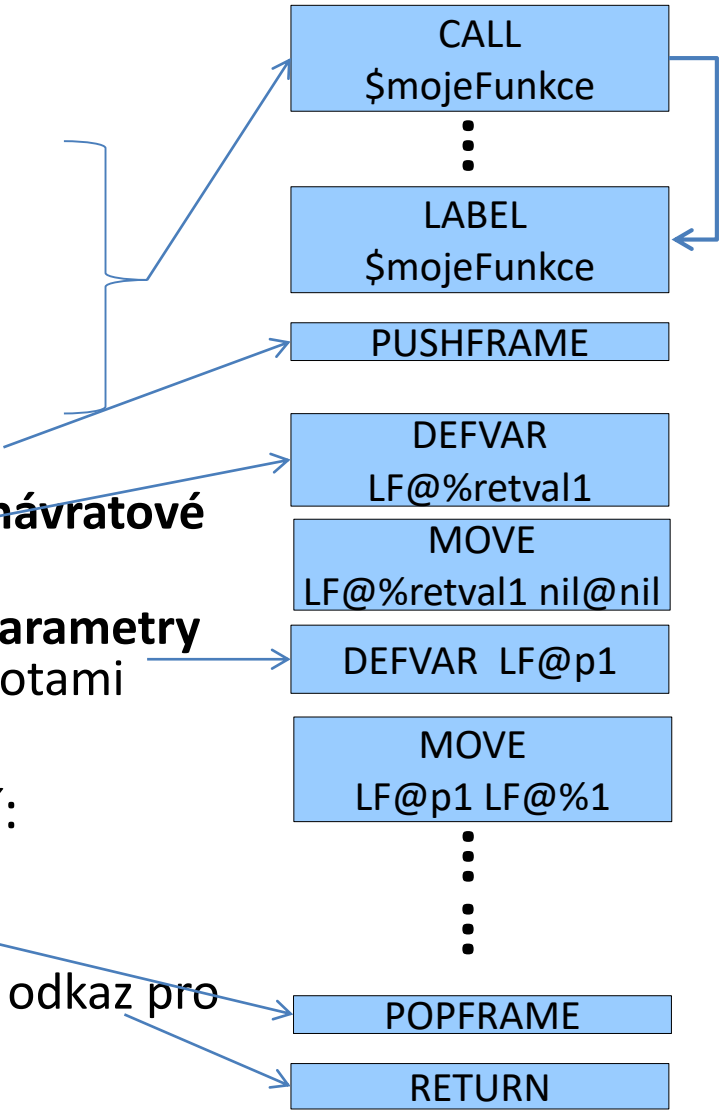
2. Samotné zavolání funkce

Při zavolání funkce provedeme následující:

1. Uložíme do **zásobníku volání** odkaz pro návrat na **následující instrukci**.
2. Přejdeme na **první instrukci funkce**.
3. Přesuneme připravovaný rámec na **vrchol zásobníku lokálních rámců**.
4. Definujeme pomocné proměnné pro **návratové hodnoty**.
5. Definujeme proměnné pro **formální parametry** funkce a naplníme je předanými hodnotami

Při ukončení funkce provedeme následující:

1. Přesuneme vrchol zásobníku rámců do **ukončeného rámce**.
2. **Načteme ze zásobníku volání** uložený odkaz pro návrat.



3. Získání návratové hodnoty

- Využijeme **ukončený (dočasný) rámec**
  - Pomocí speciálních proměnných z něj získáme návratové hodnoty (%retval1, …).
  - Odstranění TF se provede při dalším vykonání CREATEFRAME.

Souhrn: Volání funkce v IFJcode21

```
y = foo(10, "Hi X!")
print(y)
```

```
.IFJcode21
JUMP $$main
...
LABEL $foo
PUSHFRAME
DEFVAR LF@%retval1
MOVE LF@%retval1  nil@nil
DEFVAR LF@param1
MOVE LF@param1  LF@%1
DEFVAR LF@param2
MOVE LF@param2  LF@%2
# code of function foo
MOVE LF@%retval1  float@0x0p+0
POPFRAME  # callee restores
RETURN
...
```

```
LABEL $$main # main body
DEFVAR GF@y  # global var
...
CREATEFRAME
DEFVAR TF@%1
MOVE TF@%1 int@10
DEFVAR TF@%2
MOVE TF@%2 string@Hi\032X!
CALL $foo
MOVE GF@y TF@%retval1
WRITE GF@y
# end of main body
```

Definice proměnné v těle cyklu

```
…
while i < 2 do
    local a : integer = i
    i = i + 1
    local j : integer = 0
    while j < 2 do
        local b : integer = a + j
        write(b, j)
…
```

- Jak se vyhnout opakované definici proměnné *j*, *a* a *b*? Např.
  - Vytknout definici přes DEFVAR před cyklus (nejvíc vnější)
  - U „vnitřních“ proměnných si značit jejich zanoření ve jméně (i$1, a$2, j$2, b$3) (Např. „write(b, j)“ pracuje s b$3 a j$2)
  - Na místě lexikální definice (např. „**local** b : integer = a + j“) provést pouze reinicializaci
- Alternativa: vytvářet vždy nové rámce LF, ale nutno zajistit přístup k neredefinovaným proměnným (např. *i*) z překrytých rámců

Práce s konstantami/literály

- Značíme si v IR či TS a při generování vkládáme přímo do instrukcí IFJcode21
- Práce s řetězcovými literály:
  - Skener IFJ21 načítá Teal/Lua řetězec

    "Hello World\x0A!"
  - Převod na řetězcovou reprezentaci v IFJcode21

    string@Hello\032World\010!
  - Vstup ze standardního vstupu
    - zajišťuje instrukce READ např. **READ** GF@test string

      Hello World↵

      načte do GF@test řetězec bez konce řádku

      "Hello World"

Práce s konstantami/literály

- Práce s desetinnými literály:
  - Skener IFJ21 načítá Teal/Lua číslo

    3.14
  - Převod na hexadecimální reprezentaci v IFJcode21

    float@0x1.91eb851eb851fp+1
  - Vstup ze standardního vstupu
    - zajišťuje instrukce READ např. **READ** GF@test float

      3.14↵

      načte do GF@test číslo, které se pak tiskne hexadecimálně

      0x1.91eb851eb851fp+1

Nekompatibilita s Teal (viz. write v ifj21.tl) – tisk celých čísel z proměnných typu number

Práce s návěštími

- Instrukce pro podmíněný skok vyžaduje cíl skoku
- Nutno si vygenerovat unikátní návěští
- A na vhodné místo vložit skok na toto návěští
- A na vhodné místo vložit toto návěští

JUMP $mojeFunkce$if2false

...

LABEL $mojeFunkce$if2false

Větvení: If-Then-Else

Pravidlo: <if-then-else>

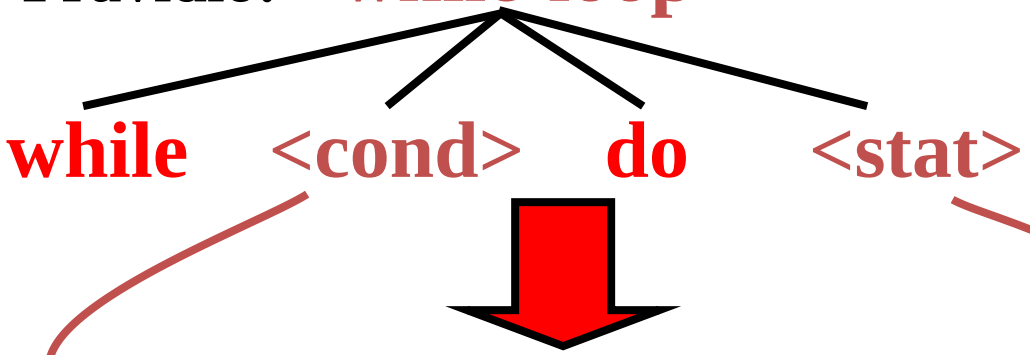
if <cond> **then** <stat$_1$> **else** <stat$_2$>

Sémantická akce:

```
{ // vyhodnocení cond
  // do proměnné c.val
(not , c.val, , c.val)
(goto, c.val, , L1   )
 // kód stat1          }
(goto,       , , L2   )
(lab , L1    , ,      )
 // kód stat2          }
(lab , L2    , ,      )
```

While cyklus

Pravidlo: <while-loop>



Sémantická akce:

```
(lab , L1    , ,          )
{// vyhodnocení cond
 // do proměnné c.val
(not , c.val, , c.val)
(goto, c.val, , L2    )
 // kód stat           }
(goto,       , , L1    )
(lab ,  L2   , ,          )
```

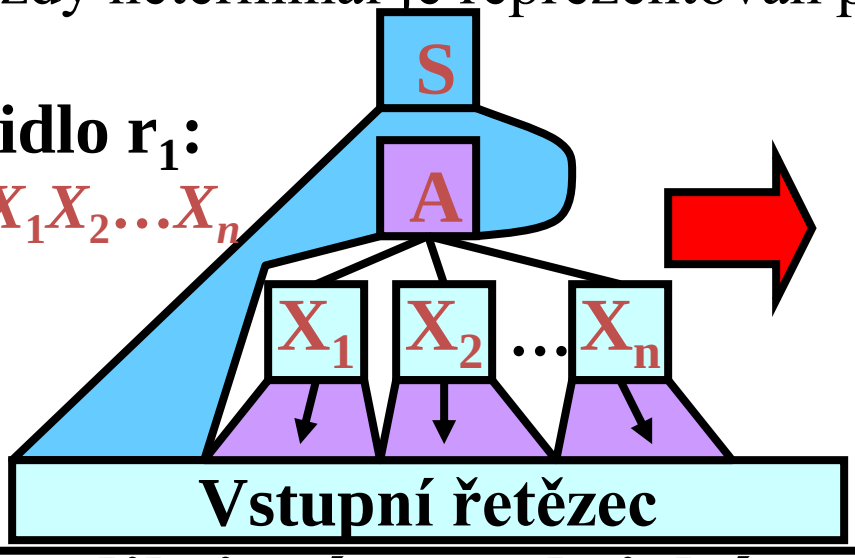
Implementace LL Analyzátoru

1) Rekurzívní sestup

• Každý neterminál je reprezentován procedurou, která řídí SA:

Pravidlo r_1:

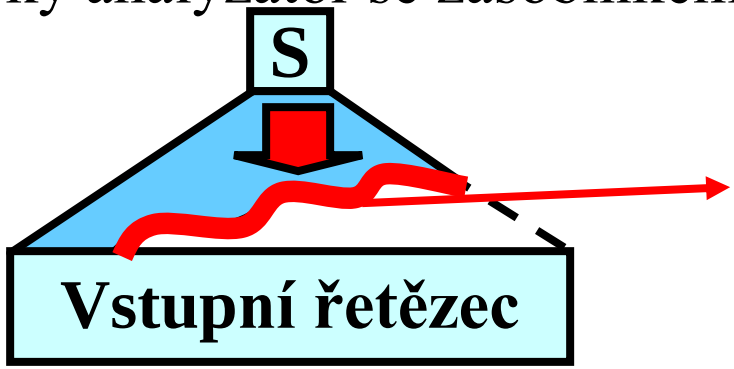
$A \rightarrow X_1 X_2 \ldots X_n$




```
function A: boolean;
begin
  {X1 analýza}
  {X2 analýza}
      …
  {Xn analýza}
end
```

2) Prediktivní syntaktická analýza

• Syntaktický analyzátor se zásobníkem řízený tabulkou



Právě tyto symboly v tomto pořadí jsou uloženy na zásobníku.

Rek. sestup založený na LL-tabulce

1: <prog> → begin <st-list>
2: <st-list> → <stat> ; <st-list>
3: <st-list> → end
4: <stat> → read id
5: <stat> → write <item>
6: <stat> → id := add (…
7: <it-list> → , <item> <it-list>
8: <it-list> →)
9: <item> → int
10: <item> → id

	beg	end	rd	wr	id	int	,	(	)	;	:=	add
<prog>	1											
<st-list>		3	2	2	2							
<stat>			4	5	6							

```
function stat: boolean;
begin
   stat := false;
   if token = 'read' then begin
      { simulace pravidla 4: <stat> → read id}
      GetNextToken;
      stat := token = 'id';
      GetNextToken; end
   else if token = 'write' then begin
      { simulace pravidla 5: <stat> → write <item>}
      …
end;
```

Rekurzivní sestup versus syntaxí řízený překlad

```
function stat: boolean;
begin
   stat := false;
   if token = 'read' then begin
      GetNextToken;
      stat := token = 'id';
       { Sém. kontroly: Je id proměnná nebo funkce? }
       { Sém. akce: Generování vnitřní reprezentace (3AC). }
      GetNextToken; end
   else if token = 'write' then begin
      { Sémantické kontroly a akce}
      …
   else if token = 'id' then begin
      GetNextToken;
       if token = ':=' then begin GetNextToken;
          if token = 'add' then begin
          …
end;
```

Rekurzivní sestup – atributy

```
x = f(10, prom1)
```

- Výňatek LL gramatiky pro volání funkce:

⟨STMT⟩ → **id = id (** ⟨PARLIST⟩ **)**

⟨PARLIST⟩ → ε

⟨PARLIST⟩ → ⟨TERM⟩ ⟨PARLIST2⟩

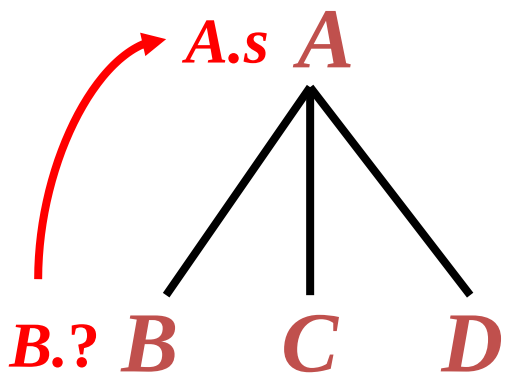
⟨PARLIST2⟩ → **,** ⟨TERM⟩ ⟨PARLIST2⟩

⟨PARLIST2⟩ → ε

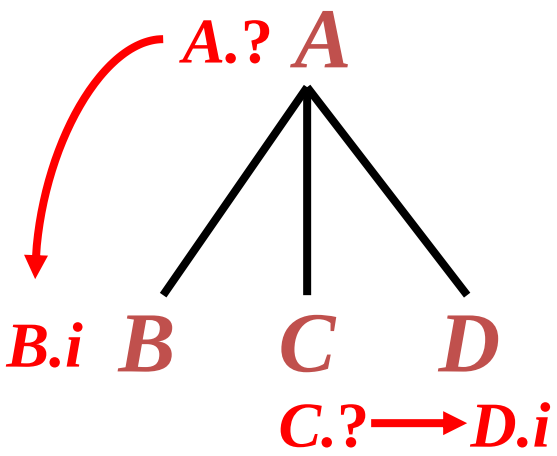
- Jak provádět sémantické akce u parametrů?
  a) ↓: ID funkce + pořadí zpracovávaného parametru
  b) ↑: plnění seznamu skutečných parametrů

Překlad shora dolů: Úvod

- **LL-gramatiky s atributy**
- **Dva zásobníky:**
  - **pro synt. analýzu** × **pro sémant. analýzu**
- **Dva typy atributů:**
  - **syntetizované:**
    (z dítěte na rodiče)



dědičné:
(z rodiče na děti nebo mezi sourozenci)



Prediktivní analýza – atributy

`f(10, prom1)`

$\langle STMT \rangle \rightarrow$ **id** {checkFun(id); L.i ← id} **(** $\langle L \rangle$ **)**

$\langle L \rangle \rightarrow \varepsilon$ {L.s ← checkPar(L.i, _,0)}

$\langle L \rangle \rightarrow$ {T.i ← (L.i,1)} $\langle T \rangle$ {L2.i ← T.s} $\langle L2 \rangle$ {L.s ← L2.s}

$\langle L2_1 \rangle \rightarrow$ **,** {T.i ← ($L2_1.i_2$, $L2_1.i_2$+1)} $\langle T \rangle$ {$L2_2$.i ← T.s} $\langle L2_2 \rangle$

$\langle L2 \rangle \rightarrow \varepsilon$ {checkParCount($L2.i_1$, $L2.i_2$)}

$\langle T \rangle \rightarrow$ **id** {checkPar($T.i_1$, id, $T.i_2$); T.s = T.i}

- Sémantické akce se ukládají též na zásobník a nutno je „interpretovat“ (s možným využitím sémantického zásobníku):
  a) ↓: ID funkce + pořadí zpracovávaného parametru
  b) ↑: propaguji aktualizované pořadí

Precedenční analýza – pomocné/dočasné proměnné

- Syntaktický a sémantický zásobník v Parseru
- Co cílový kód? Záleží na využití instrukční sady:
  - Při generování kódu v postfixové notaci (zásobníkový kód) → datový zásobník operandů i mezivýsledků
  - Při generování klasického 3AC (ne IFJcode21 syntaxe)

    ADD TF@$tmp1 LF@x LF@y

    MUL LF@vysl TF@$tmp1 int@2

→ vytváření dočasných proměnných v rámci

Vaše dotazy ???

Další dotazy

- Zotavení z chyb – návratový kód první chyby!
- Relační operátory bez asociativity?
- Implicitní konverze ve výrazech – viz zadání!
- Nástroj pro tvorbu LL tabulky
- Mírně zastaralé "Jak na projekt" z roku 2008
  - Od 2017 vytváříte překladač (ne interpret)!!!
- Minimální funkcionalita?
  - lex. + synt. analýza, generování kódu (definice proměnné, jednohodnotové přiřazení, základní výrazy, výpis), dále implementace příkazů a nakonec volání funkce
- ... čtěte zadání, wiki, fórum, Discord ...