Kódgenerálás

Kitlei Róbert

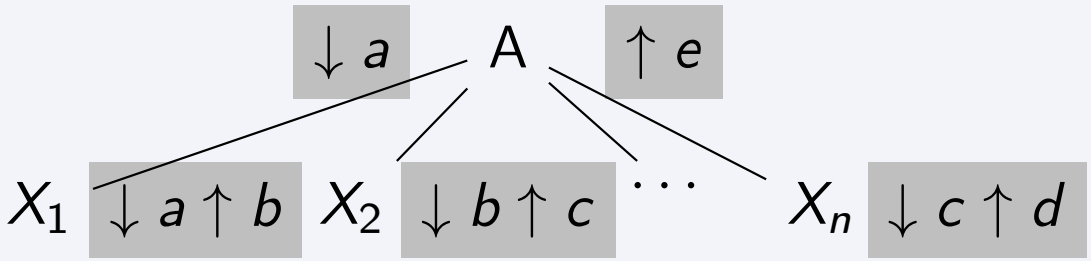
http://kitlei.web.elte.hu

2010. május 5.

Kódgenerálás

Az eddigi előadásokon szerepelt

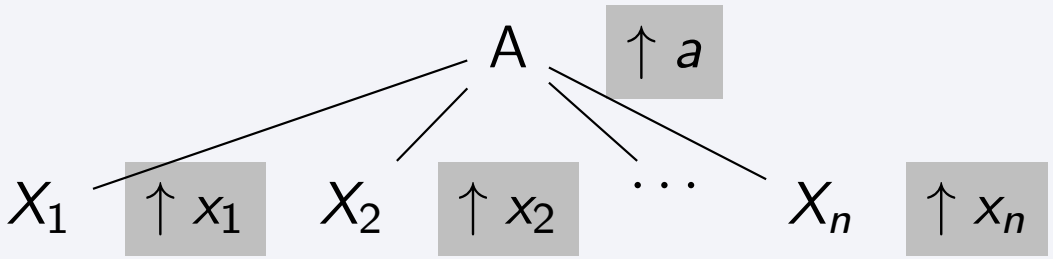
- szintaktikus elemzés (LALR(1))
- módszerek attribútumok terjesztésére a fában (L-ATG, S-ATG)



Most: kódgenerálás szemantikus rutinokkal

- elmélet
- az elmélet gyakorlati illusztrálása bisonc++ kóddal
- generált kód: x86, 32 bites, Intel szintaxisú (nasm-mal fordítható)

A bisonc++ szabályainak alakja



@Szem(↓ $x_1, \ldots, x_n$ ↑ a)

A nyelvtani jeleket kis-, a terminálisokat nagybetűk jelölik.

```
a:
    X1
    X2
    ...
    Xn
        { szemantikus rutin }
;
```

A rutinon belül X_k attribútumát `$k`-val lehet elérni, az aktuális nyelvtani jelét `$$`-ral. Minden jelnek csak egy attribútuma lehet; ha többre van szükség, rekordba szervezhetőek.

A bisonc++ szabályainak alakja

Több alternatívájú szabály:

```
a:
    alternatíva1
|
    alternatíva2
|
    ...
;
```

Epszilon-szabály:

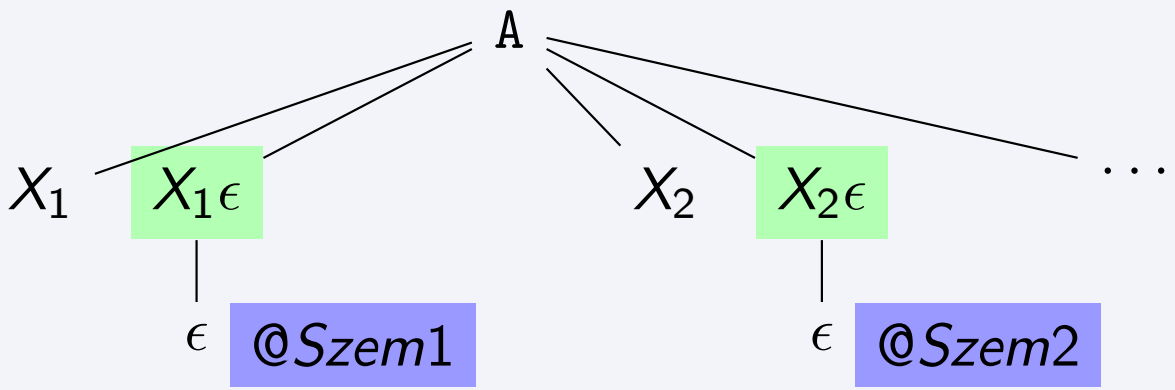
```
a:


;
```

A bisonc++ szabályainak alakja

Szemantikus rutinokat írhatunk a szabály belsejébe is, ez valójában epszilon-szabályt takar.

```
nemterminalis:
    X1          { rutin1; }
    X2          { rutin2; }
    ...
;
```



A dollárok számozásába a rutinok is beszámítanak, pl. X_2 attribútuma $3.

A bisonc++ szabályainak alakja

Az attribútumok típusozottak.

```
%type <atom> valtozo
```

Az összes lehetséges kifejezés típusát össze kell gyűjteni.

```
%union
{
  ...
  atom_leiro* atom;
  ...
}
```

A fentiek alapján a valtozo szabály attribútumának a típusa `atom_leiro*`. (Az `atom` a tárhely azonosítására szolgál.)

Az elemző felépítése

- elemzo.l: a lexikai elemző leírása (flex)
- elemzo.y: a szintaktikus elemző + szemantikus rutinok leírása (bisonc++)
- Parser.h és .cc: a szintaktikus és szemantikus elemző modul
  - atom_leiro.h és kifejezes_leiro.h
- kodgenerator.h és .cc: a kódgenerátor modul
- foprogram.cc

Szemantikus tevékenységek

A szemantikus tevékenységek felosztása

- a `kodgenerator` osztály statikus függvényei végzik a kódgenerálást
- minden mást a `Parser` objektum maga
  - szimbólumtábla kezelése
  - címkék generálása
  - típusvizsgálat
- a szimbólumtábla egyszerűsített
  - két hasítótábla
    - szimbólum neve (azonosító) $\mapsto$ típusa
    - alprogram neve $\mapsto$ visszatérési típusa, formális paraméterei
  - címkék verme, az örökölt attribútumok szimulálására
  - el van tárolva továbbá az aktuálisan fordított alprogram

Eltérések az S-ATG-ktől

Precedenciák megadásával egyszerűbben le tudjuk írni a nyelvtanokat.

$$
\begin{array}{rcl|l}
\text{DiszjLánc} & \rightarrow & \text{KonjLánc } \textit{or} \text{ DiszjLánc} & \text{KonjLánc} \\
\text{KonjLánc} & \rightarrow & \text{Negált } \textit{and} \text{ KonjLánc} & \text{Negált} \\
\text{Negált} & \rightarrow & \textit{not} \text{ NemNegált} & \text{NemNegált} \\
\text{NemNegált} & \rightarrow & (\text{ DiszjLánc }) & \text{változó}
\end{array}
$$

helyett

$$
\begin{array}{rcl}
\text{Kifejezés} & \rightarrow & (\text{ Kifejezés }) \\
& | & \textit{not} \text{ Kifejezés} \\
& | & \text{Kifejezés } \textit{and} \text{ Kifejezés} \\
& | & \text{Kifejezés } \textit{or} \text{ Kifejezés} \\
& | & \text{változó}
\end{array}
$$

Eltérések az S-ATG-ktől

- az S-ATG-k csak szintetizálják az attribútumokat
  - mi statikus változók segítségével örökölt attibútumokat is kezelünk
  - így L-ATG-hez hasonló funkcionalitást kapunk
- minden szabályban rögtön generáljuk a hozzá tartozó kódot
  - az S-ATG-kben felfele terjed a generált kód

A nyelvtan felépítése

Konstrukciónként bővített nyelvtan

- konstansok, ezek elérése
- globális változók
- értékadás
- kifejezések
  - mohó kiértékelésűek
  - lusta kiértékelésűek
- programkonstrukciók
  - elágazás
  - ciklus
- alprogramok
  - paraméterek nélkül
  - paraméterekkel
  - lokális változókkal

A program váza

program → @Prológus definíciók program

@KódPrológus utasítások @KódEpilógus

Példa

int i
program
i := 1

- a prológus és az epilógus állítják be a program kapcsolatát a környezettel
  - `@Prológus`: a printf kiíró rutint használjuk, elérhetővé kell tenni
  - `@KódPrológus`: a main függvény kezdete
  - `@KódEpilógus`: kilépés a programból

Konstansdeklarációk

- a nyelvünk szigorúan típusos
  - minden deklarációban ki kell írni a típust
  - két alaptípus
    - bool: egy bájton tárolt
    - int: négy bájton tárolt, előjeles
- mivel felhasználói típus (pl. rekord) definiálására nincsen eszköz a nyelvben, a változók méretét könnyű meghatározni
- a fordítóprogramban közvetlenül kezeljük a konstansokat
  - összetettebb konstansok (pl. string) esetén az adatszegmensbe kell helyezni őket
- a konstansok értékét kifejezéssel is meg lehetne határozni
  - ehhez a kifejezést lehetne átalakítani

Konstansdeklarációk

$\text{dekl}_{\text{konstans}} \rightarrow$ konstans típus ↑ típus azonosító ↑ név

értékadás számkonstans

@Allokál(↓ típus, név) @KonstDeklarál(↓ típus, név)

típus $\rightarrow$ bool ↑ típus | int ↑ típus

Példa

constant int i := 3

- az inicializált adatszegmensbe helyezzük
- a szimbólumtáblában meg kell jelölni konstansként, és később ezt betartatni
  - ez jelenleg nincsen benne a kódban

Változódeklarációk

$\text{dekl}_{\text{globvált}} \rightarrow$ típus ↑ típus azonosító ↑ név

@Allokál(↓ típus, név) @Deklarál(↓ típus, név)

Példa

int i

- globális változó: az inicializálatlan adatszegmensbe helyezzük

Értékadó utasítás

- csak egyszerű változók értékadását valósítjuk meg
  - a bal oldalon szereplő változónak a címére, a jobb oldalon szereplőnek az értékére van szükségünk
  - nyilvántartjuk, egy kifejezés címét vagy értékét számítottuk-e ki
    - konstansoknak csak az értékét lehet
    - változóknak kiszámítjuk a címét, majd feloldjuk, ha a bal oldalt áll
- összetett értékadás
  - egyes adatszerkezetekre (pl. rekord) automatikusan generálható
    - klónozás: mezőnkénti átmásolás
    - sekély másolás: mutatott adatok lemásolása a mutató másolása helyett
    - mély másolás: láncolt szerkezetek követése - kézzel megadott

Értékadó utasítás

$\text{ut}_{\text{értékadás}} \rightarrow$ változó $\uparrow \text{típus}^{c/e}_{\text{változó}}$ @MentAkku() :=

kifejezés $\uparrow \text{típus}^{c/e}_{\text{kifejezés}}$ @Felold($\downarrow \text{típus}^{c/e}_{\text{kifejezés}}$)

@TöltAdat() @Értékadás($\downarrow$ típus)

Példa

i := 3

- `@Felold`: ha az adat címét kaptuk meg, feloldjuk a mutatót
- `@MentAkku`, `@TöltAdat`: a tárcím átadása veremmel
- `@Értékadás`: a címre töltjük az adatot
- az értékadás előtt konverzióra lehet szükség
- ez a konstrukció akkor is működik, ha változó helyett pl. függvény visszatérési értéke adja a címet

Atomi kifejezések

$\text{kif}_{\text{atomi}}$	$\rightarrow$	változó	$\uparrow$ típus$^{c/e}$	
		konstans$_{\text{egész}}$	$\uparrow$ típus^{e}	@GenerálInt($\downarrow$ típus)
		konstans$_{\text{logikai igaz}}$	$\uparrow$ típus^{e}	@Generállgaz($\downarrow$ típus)
		konstans$_{\text{logikai hamis}}$	$\uparrow$ típus^{e}	@GenerálHamis($\downarrow$ típus)
változó	$\rightarrow$	azonosító	$\uparrow$ típus^{c}	@GenerálVáltozó($\downarrow$ típus)

Példa

123

- a kifejezéseket az akkumulátor regiszterbe értékeljük ki (eax)
- a típus$^{c/e}$ jelölésben a felső index azt mutatja, hogy cím vagy érték került az akkumulátorba

Mohó kiértékelésű kifejezések

$\text{kifejezés}_{\text{mohó}} \rightarrow$ kifejezés $\uparrow \text{típus}_1^{c/e}$ operátor @MentAkku()

kifejezés $\uparrow \text{típus}_2^{c/e}$ @Felold($\downarrow \text{típus}_2^{c/e}$)

@TöltAdat() @Felold($\downarrow \text{típus}_1^{c/e}$)

@$\text{Művelet}_{\text{operátor}}$($\downarrow \text{típus}_1$)

Példa

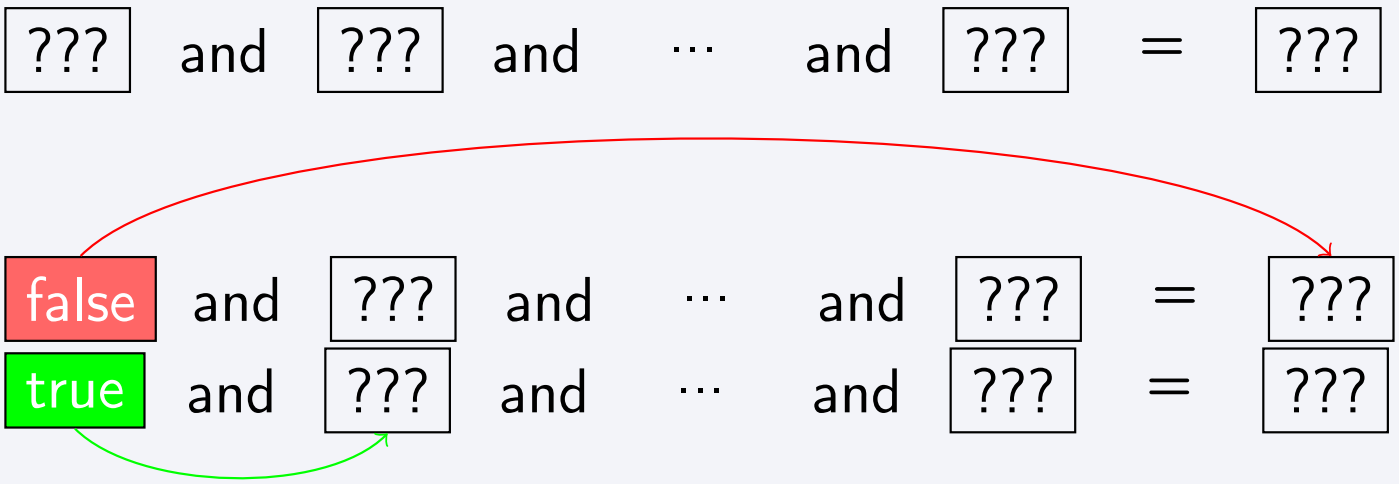
1 + 2

- a mohó kifejezések kiértékelik mindkét argumentumukat
- tipikusan ilyenek az aritmetikai műveletek

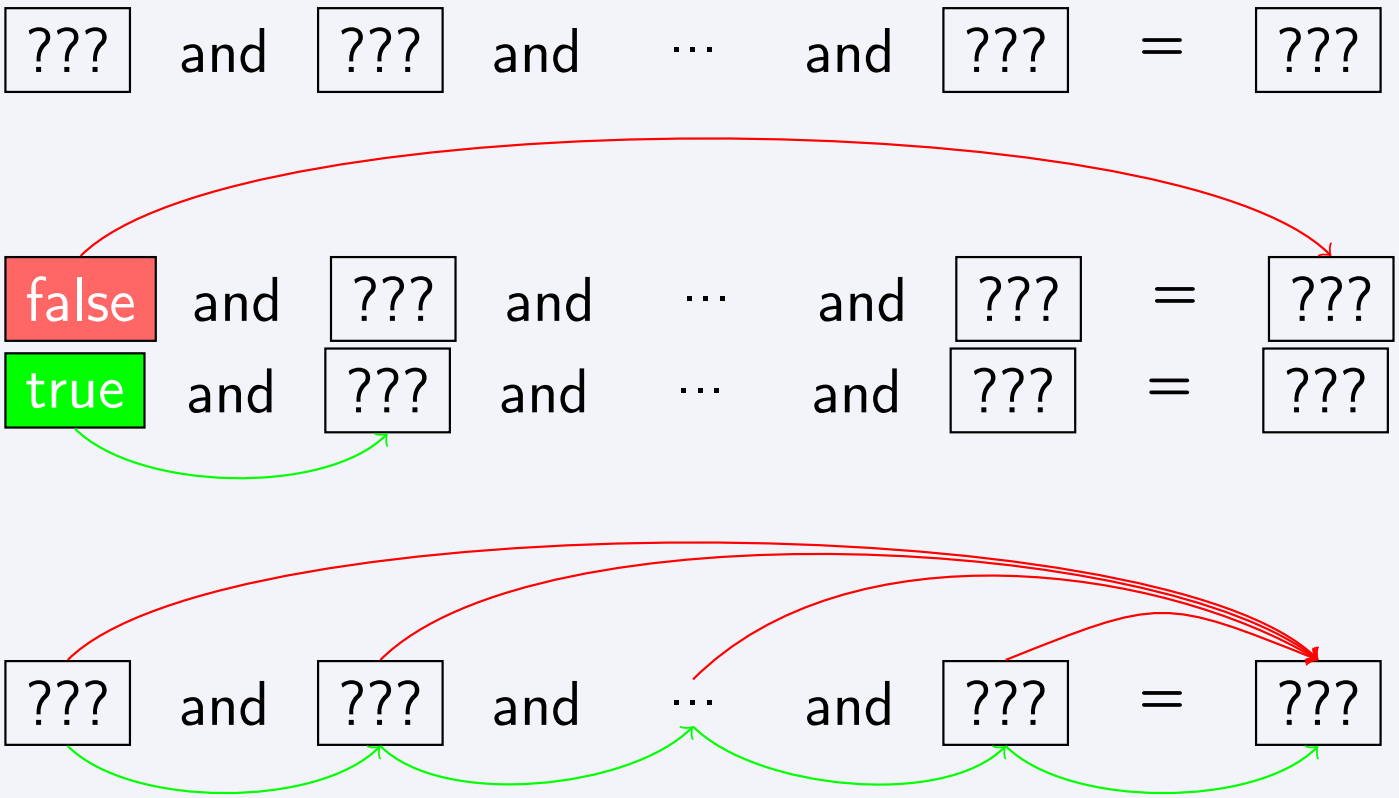
Lusta kiértékelésű (rövidzáras) kifejezések

$$\boxed{???} \text{ and } \boxed{???} \text{ and } \cdots \text{ and } \boxed{???} = \boxed{???}$$

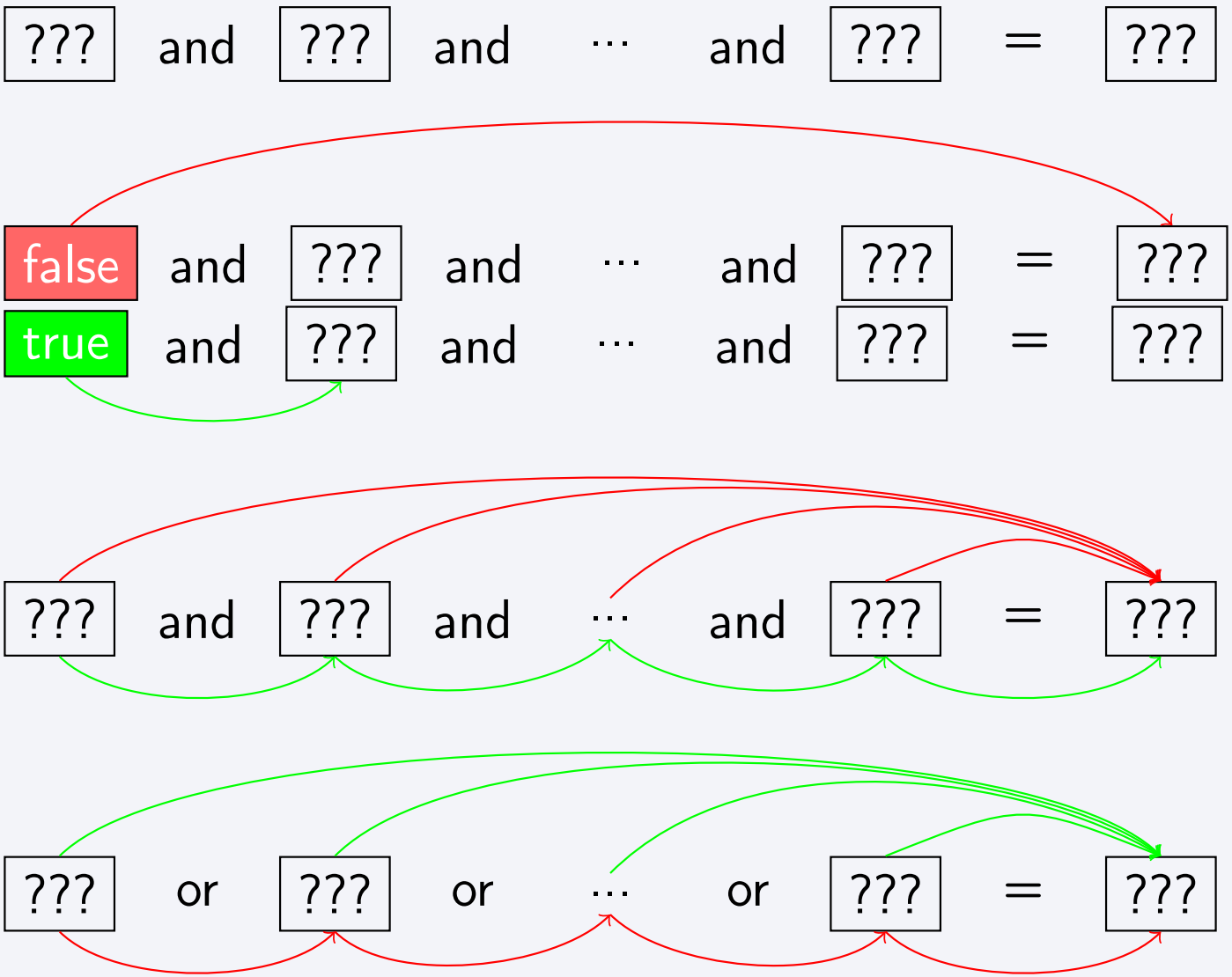
Lusta kiértékelésű (rövidzáras) kifejezések



Lusta kiértékelésű (rövidzáras) kifejezések



Lusta kiértékelésű (rövidzáras) kifejezések



Lusta kiértékelésű (rövidzáras) kifejezések

- előnyök a mohó megoldással szemben
  - nem kell elvermelni kif$_1$eredményét
  - a műveletet nem kell végrehajtani, a konstrukcióból adódik
  - az első ágtól függhet, hogy a második ág végrehajtható-e
    - $a \neq 0$ and $b/a = 1$

Elágazás utasítás

$ut_{\text{elágazás}} \rightarrow$ if kifejezés $\uparrow$ típus$^{c/e}$ @Felold($\downarrow$ típus$^{c/e}$)

@ÚjCímke($\uparrow$ címke$_{\text{hamis ág}}$) @Ugrás$_{\text{kif hamis}}$($\downarrow$ címke$_{\text{hamis ág}}$)

then utasítások else-ág $\downarrow$címke$_{\text{hamis ág}}$ $\uparrow$ címke$_{\text{vége}}$

endif @CímkétElhelyez($\downarrow$ címke$_{\text{vége}}$)

else-ág $\rightarrow$ ϵ @CímkétMásol($\downarrow$címke$_{\text{hamis ág}}$ $\uparrow$ címke$_{\text{vége}}$)

| else @ÚjCímke($\uparrow$ címke$_{\text{vége}}$)

@Ugrás($\downarrow$ címke$_{\text{vége}}$)

@CímkétElhelyez($\downarrow$ címke$_{\text{hamis ág}}$) utasítások

Végtelen ciklus

$\text{ut}_{\text{ciklus}\infty}$ $\rightarrow$ loop @ÚjCímke($\uparrow$ $\text{címke}_{\text{eleje}}$)

@CímkétElhelyez($\downarrow$ $\text{címke}_{\text{vége}}$) utasítások

endloop @Ugrás($\downarrow$ $\text{címke}_{\text{eleje}}$)

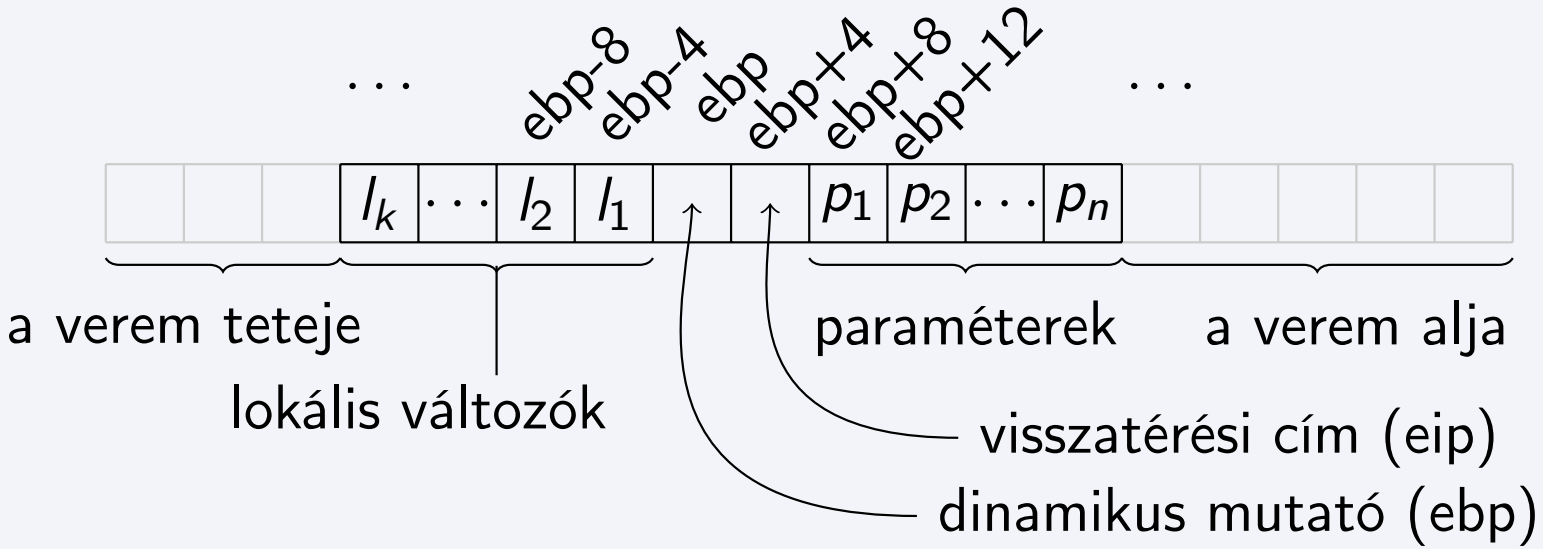
Elöltesztelő ciklus

$\text{ut}_{\text{elágazás}} \rightarrow$ while @ÚjCímke($\uparrow$ címke$_{\text{eleje}}$) @CímkétElhelyez($\downarrow$ címke$_{\text{eleje}}$) kifejezés $\uparrow$ típus$^{c/e}$ @Felold($\downarrow$ típus$^{c/e}$) @ÚjCímke($\uparrow$ címke$_{\text{hamis ág}}$) @Ugrás$_{\text{kif hamis}}$($\downarrow$ címke$_{\text{hamis ág}}$) do utasítások done @Ugrás($\downarrow$ címke$_{\text{eleje}}$) @CímkétElhelyez($\downarrow$ címke$_{\text{vége}}$)

- a hátultesztelő ciklus is hasonló

Aktivációs rekord

- aktivációs rekord: az alprogram futó példányához tartozó információk
  - dinamikus mutató: a hívó aktivációs rekordjára mutat
  - visszatérési cím: az alprogramból való kilépés után itt folytatódik a végrehajtás
  - paraméterek
  - lokális változók
- veremkeret: a futási idejű veremben tárolt aktivációs rekord



Aktivációs rekord

- környezet (display terület): ha az alprogramból elérhetőek rajta kívül eső, de nem globális változók, az itt levő statikus mutatókon keresztül érhetjük el őket
  - környezetre akkor van szükség, ha függvényeket is át lehet adni paraméterként (esetünkben nem)
- az aktivációs rekord egy kijelölt pontja a bázisa, minden címet erre relatívan határozunk meg benne
  - célszerű választás a dinamikus mutató tárcíme

Paraméterátadási módok

- paraméterátadási módok
  - a formális paraméter lokális változó a hívott alprogramban
    - **érték szerinti**: az alprogram veremkeretébe lemásolódik a paraméter értéke, a továbbiakban független tőle
    - **csak olvasható érték szerinti**
    - **eredmény szerinti**: az alprogram futásának végén visszamásolódik a lokális változó
    - **érték-eredmény szerinti**: az elején és a végén is másolódik
  - **cím szerinti:** a paraméter címe lesz a formális paraméter
  - **név szerinti** (ritka): az aktuális paraméter szövegének behelyettesítése az alprogram forrásszövegében

Formális paraméterlista

Az alprogramdeklaráció tartalmazza a **formális paraméterlistát**. Ez a következőket tartalmazza a formális paraméterek **nevét** és **típusát**.

Példa

int f(int i, bool b)
Ez a függvény két formális paramétert tartalmaz: az int típusú i-t és a bool típusú b-t.

A szimbólumtáblában az alprogramról a következő információkat tároljuk el.

- az alprogram nevét
- a hozzá tartozó formális paraméterlistát
- a visszatérési értékét
- a lokális változók listáját
  - ez nincsen jelenleg implementálva, lásd később

Formális paraméterlista

Az alprogramhívás tartalmazza az **aktuális paraméterlistát**.

Példa

Az `f(1+k, (55 < i) or (2 * j = 1500))` alprogramhívás két aktuális paramétert tartalmaz. Az első `1+k, típusa` int, `a második` $(55 < \text{i})$ or $(2 * \text{j} = 1500)$, `típusa bool.`

Alprogramhívás esetén két feladatunk van.

- megvizsgáljuk a típusokat, hogy páronként megfelelnek-e
- elkezdjük építeni a veremkeretet
  - sorban kiszámítjuk az aktuális paraméterek értékét, és betesszük őket a verembe
  - a sorrend a hívási konvenciótól függ

„Lokális változók”

- trükk: a tárgyalt program minden „lokális” változója valójában globális
  - ez két megkötést von maga után
    - nem lehet rekurzív alprogramokat írni
    - minden „lokális” változó neve különböző kell, hogy legyen
  - ezzel kihasználjuk, hogy nincsenek blokkok: globális nevet csak az aktuális alprogram lokális változója tud elfedni
- valódi lokális változókat is könnyű kezelni
  - az alprogram-leíró osztályban fel kell venni a lokális változók listáját (a paraméterek listájához hasonlóan)
  - a változók kódjának generálásakor (`kodgenerator::valtozo_azon`) ezeket is végig kell keresni
  - generált cím: `ebp - 4 * lokális_változó_indexe`

Javítási és bővítési lehetőségek a mintaprogramhoz

- a program jelenleg nem ad értelmes hibaüzeneteket (Syntax error)
- a program kimenete a sztenderd kimenet
- a típusleíró jelenleg csak a típus nevét tartalmazza
- a program nem tartalmaz hibakezelést
- a program nagyvonalúan bánik a memóriával
  - minden lefoglalt területet fel kellene szabadítania (delete)
  - alternatíva: szemétgyűjtő alkalmazása
- nincsenek konstans kifejezések
- a logikai értékek kiírása egybeesik az egészekével
- valódi lokális változók bevezetése
- további típusok és konstrukciók bevezetése: lebegőpontos számok, stringek, tömbök, rekordok stb.