Formális nyelvek és fordítóprogramok alapjai

Fordítóprogramok témakör jegyzete

Készült Dévai Gergely előadásai és gyakorlatai alapján

Sárközi Gergő, 2021-22-2. félév
Nincsen lektorálva!

Tartalomjegyzék

1. Fordítóprogramok bevezetés **3**
1.1. Fordítás és szerkesztés 3
1.2. Fordítóprogramok logikai felépítése 4

2. Lexikális elemzés (scanner / lexer) **5**
2.1. Bevezetés 5
2.2. Whitespace, kommentek 5
2.3. Tokenizálás 5
2.3.1. Tokenekhez csatolt információk 6
2.3.2. Lexikális hibák 6

3. Szintaktikus elemzés (syntax checker) **7**
3.1. Bevezetés 7
3.2. Szintaxisfa 7
3.3. Nyelvtan elvárt tulajdonságai 7
3.4. Nem egyértelmű szintaxisfa megoldása 7
3.5. Szintaxisfa felépítése 7
3.5.1. Felülről lefelé elemzés (pl. ANTLR) 8
3.5.2. Alulról felfelé elemzés (pl. Bison) 9
3.5.3. Backtracking (extra, nem az anyag része) 10

4. Szemantikus elemzés (semantic analyzer) **11**
4.1. Bevezetés 11
4.2. Feladatok kategóriái 11
4.3. Szimbólumtábla 12
4.4. Típusrendszerek 13

4.5. Ki adja meg a típusokat? . . . 13
4.6. Típuskonverzió . . . 13
4.7. Attribútumnyelvtan . . . 14

5. Assembly **16**
5.1. Bevezetés . . . 16
5.1.1. C használat (`extern` függvények) . . . 16
5.1.2. Fordítás . . . 16
5.2. Általános információk . . . 16
5.3. Adattárolás . . . 17
5.3.1. Regiszterek . . . 17
5.3.2. Verem . . . 17
5.3.3. Statikus memória . . . 18
5.4. Programkód . . . 18
5.5. Műveletek . . . 19
5.5.1. Általános tudnivalók . . . 19
5.5.2. Aritmetikai műveletek . . . 19
5.5.3. Összehasonlítás . . . 20
5.5.4. Ugrás (feltétel nélküli, feltételes ugrás) . . . 20
5.5.5. Adat mozgatás, másolás (feltétel nélküli, feltételes) . . . 20
5.5.6. Bitműveletek . . . 21
5.5.7. Függvény hívás és visszatérés . . . 21
5.6. Copy-paste-elhető sablonok . . . 22
5.6.1. Egyágú elágazás . . . 22
5.6.2. Kétágú elágazás . . . 22
5.6.3. Számláló ciklus . . . 22

6. Kódgenerálás **23**
6.1. Bevezetés . . . 23
6.2. Kódgenerálás attribútumnyelvtannal . . . 23
6.3. Generált kód . . . 23
6.4. Végeredmény . . . 23

7. Gyakorlati jegyzet **24**
7.1. `while.cc` . . . 24
7.2. `while.l` . . . 24
7.3. `while.y` . . . 25
7.4. `implementation.hh`, `implementation.cc` . . . 26
7.5. Assembly integráció, kódgenerálás . . . 26
7.5.1. Új típus létrehozása . . . 26

1. Fordítóprogramok bevezetés

- Fordítóprogram lényege: magas szintű és gépi kód között áthidal
- Fordítás (compiler) vs értelmezés (interpreter)
  - Fordítás: optimalizálás, alaposabb ellenőrzés, platformonként
  - Értelmezés: rugalmasabb, csak futási idő van (pl. Python) Dinamikus bemenet is futtatható egyszerűen
- Virtuális gép (JVM, CLR): fordított kód (bájtkód, CLI) értelmezése
- JIT: bájtkód fordítása gépi kódra futási időben (cél: gyorsítás)
  - Kezdetben csak értelmezés, hot spots-ok keresése
  - Ezután hot spots-ok lefordítása gépi kódra
    - Extra: futási idejű infókat is fel lehet használni fordításkor
  - Miért nem mindent induláskor? $\Longrightarrow$ lassú indulás

1.1. Fordítás és szerkesztés

- Statikus könyvtár (.a, .lib):
  - Beépül a futtatható állományba
- Dinamikus könyvtár (.so, .dll):
  - Pozíciófüggetlen objektumok vannak benne
  - Függőség, hivatkozás egy rendszeren jelen levő könyvtárra
  - Ugyan azt a példányt használja minden processz
  - Csak programkódról van szó, nincs közös memória

1.2. Fordítóprogramok logikai felépítése

- Bemenet: forráskód
- Analízis: lexikális, szintaktikus, szemantikus elemzés (ez a sorrend)
  - Minden hibának ki kell itt derülnie
  - Lexikális elemzés:
    - Kimenet: tokenek (lexikális elemek) sorozata
    - Whitespace-t általában figyelmen kívül hagyjuk
  - Szintaktikus elemzés:
    - Kimenet: szintaxisfa
    - Tokenek belső szerkezetének feltérképezése, ellenőrzése
    - Token: szintaktikus elemzés nyelvtanának terminális szimbóluma
  - Szemantikus elemzés: (statikus szemantika)
    - Kimenet: szintaxisfa attribútumokkal, szimbólumtábla
      - Attribútum példa: milyen típusú egy kifejezés
      - Szimbólumtábla példa: milyen típusú egy változó
    - pl. változó deklarálva volt-e, típushelyes-e minden
- Szintézis: kódgenerálás, optimalizálás
  - Általában nem egy lépés, hanem transzformációk sorozata
  - Váltják egymást, először valami köztes nyelv, végül tárgykód
  - Kódgenerálás:
    - Nem érdemes egyből gépi kódra fordítani
    - magas szint $\rightarrow$ (fordító) $\rightarrow$ Assembly $\rightarrow$ (assembler) $\rightarrow$ gépikód
  - Optimalizálás:
    - Példa: szintaxisfa optimalizálás $\rightarrow$ kódgen. $\rightarrow$ assembly optim.
- Kimenet: tárgykód

2. Lexikális elemzés (scanner / lexer)

2.1. Bevezetés

- Feladat: forrásszöveg elemi egységekre bontása
- Bemenet: karaktersorozat
- Kimenet: lexikális elemek (tokenek)
- Eszközök: reguláris kifejezések, véges determinisztikus automaták

2.2. Whitespace, kommentek

- Whitespace fel van ismerve, de nem készül belőle token
- Behúzás-érzékeny nyelvek: számon van tartva, készül belőle behúzás szintet növelő/csökkentő token
- Kommenteket is felismeri a lexer, ebből sem készül token

2.3. Tokenizálás

- Egy tokenhez a reguláris kifejezés általában elég egyszerű, hiszen ezek az elemi (legkisebb) egységek.
- A leghosszabb illeszkedést keressük: addig vesszük a következő karaktert, amíg már semmi nem illeszkedik. Ekkor az előző karaktersorozatra illeszkedő regexek közül vesszük a sorrendben az elsőt.
  - Leghosszabb illeszkedés elve: leghosszabb illeszkedést keressük
  - Prioritás elve: több regex közül a sorrendben az első "nyer"
  - A leghosszabb illeszkedés fontosabb, mint a sorrend
- Implementáció:
  - Reguláris kifejezést VDA-vá alakítjuk
  - VDA implementáció: elágazásokkal / táblázattal

2.3.1. Tokenekhez csatolt információk

- Minden tokenhez: pozíció (első és utolsó karakter: sor és oszlopszám)
  - Hibaüzenetekhez szükséges
- Azonosítókhoz, literálokhoz: a megadott érték (lásd szemantikus elemzés)

2.3.2. Lexikális hibák

- Akkor van hiba, ha semmi nem match-elt
- Egyszerű megoldás: egy karaktert kidobunk (kiírunk hibát), többi karakterrel folytatjuk az elemzést
- Példa: illegális karakter (karakter nincs a nyelv abécéjében)
- Példa: lezáratlan string (hiba a sor végén derül ki)
- Példa: lezáratlan többsoros megjegyzés (fájl végén derül ki)

3. Szintaktikus elemzés (syntax checker)

3.1. Bevezetés

- Feladat: forrásszöveg szerkezetének felderítése, formai ellenőrzése
- Bemenet: tokenek
- Kimenet: szintaxisfa és szintaktikus hibák
- Eszközök: környezetfüggetlen nyelvtanok, veremautomaták

3.2. Szintaxisfa

- Levelek balról jobbról olvasva kiadják a bemeneti tokeneket
- Grammatika alapján jön létre; az szabályozza, hogy mi megengedett

3.3. Nyelvtan elvárt tulajdonságai

- Redukáltság: nincsenek felesleges nemterminálisok
- Ciklusmentesség: az baj, ha A-ból eljuthatunk A-ba anélkül, hogy bármilyen terminálist előállítanánk
- Egyértelműség: minden szóhoz pontosan egy szintaxisfának kell tartoznia
  - Többféle levezetés szabályos, ez mást jelent!

3.4. Nem egyértelmű szintaxisfa megoldása

- Példa: nyelvtan számok összeadására (Bonyolítás: szorzást is belevesszük)
  - Helytelen: $E \to \mathbb{N} \mid E + E$
  - Helyes: $E \to \mathbb{N} \mid E + \mathbb{N}$
- Operátor precedencia megadása (erre a példa nem tér ki)
- Operátor asszociativitásának megadása

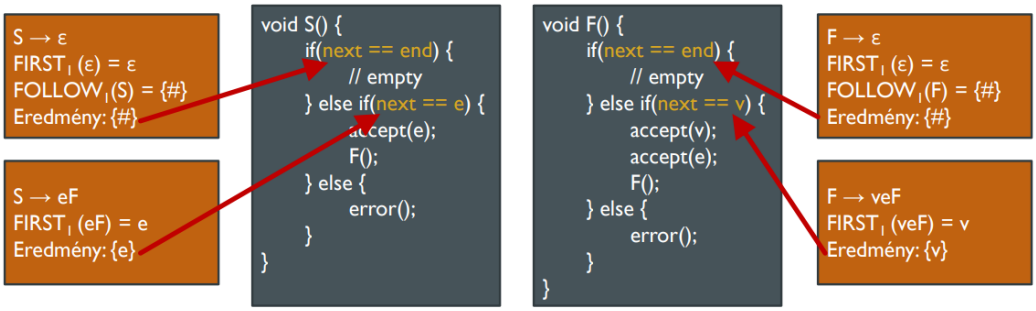
3.5. Szintaxisfa felépítése

- Két fő család: felülről lefelé és alulról felfelé

3.5.1. Felülről lefelé elemzés (pl. ANTLR)

- Startszimbólumból indulva terminálisok felé haladunk
- Input feldolgozása: balról jobbra
- Legbaloldalibb levezetés: több terminális esetén a legbaloldalibb
- Előreolvasás: következő pár token alapján válasszunk szabályt
  - Egy nyelvtan LL(k) tulajdonságú, ha felülről lefelé elemzés során legfeljebb k token előreolvasásával meghatározható a legbaloldalibb feldolgozatlan nemterminálishoz alkalmazandó nyelvtani szabály
  - Van olyan nyelvtan, amihez nem létezik ilyen k, de a programozási nyelvek nyelvtanai nem ilyenek
- Legyen α egy tetszőleges szimbólumsorozat: terminálisok és nemterminálisok
- $FIRST_1(\alpha) = \alpha$-ból levezethető szimbólumsorozatok első karakterei
  - ϵ része ha az üres szó levezethető, továbbá $FIRST_1(\epsilon) = \{\epsilon\}$
  - α egy terminálissal kezdődik $\implies$ az a terminális az eredmény
- $FOLLOW_1(\alpha) =$ kezdőszimbólumból levezethető szimbólumsorozatokban milyen terminálisok állhatnak α után
  - α a levezethető szimbólumsorozat legvége $\implies$ # kerül a halmazba
  - Kiszámolása nem triviális a rekurzív szabályok miatt
- Rekurzív leszállás: LL(1) elemzés egy implementációja
  - LL: Left to right, using Leftmost derivation
  - Legyen minden nemterminális egy függvény és minden szabályalternatíva egy-egy elágazás-ág (hibakezelés külön ágban)
  - Szabály jobboldal: terminálishoz *accept*, nemterminálishoz fv hívás
    - *accept*: tovább olvasás ha next == paraméter, egyébként hiba
  - Elágazás feltételek megírása, kiszámolása:
    - Legyen a szóban forgó szabály: $A \to \alpha$
    - $FIRST_1(\alpha)$ halmaz kiszámolása, ha van benne ϵ, akkor kivesszük és vesszük az uniót a $FOLLOW_1(A)$ halmazzal
    - Ezen elemek a feltételek: $next == x_1 \,||\, next == x_2 \,||\, ...$

- Tétel: környezetfüggetlen nyelvtan pontosan akkor LL(1) tulajdonságú, ha bármely $A \to \alpha$, $A \to \beta$ szabályokra a fenti feltételek diszjunktak
- Van verem: a call stack
- Név eredete: rekurzív nyelvtan esetén rekurzív függvény hívások

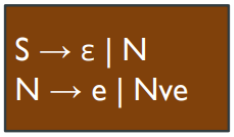


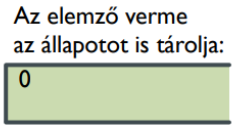
- Ábra: felülről lefelé elemzés, LL(1), rekurzív leszállás

3.5.2. Alulról felfelé elemzés (pl. Bison)

- Terminálisokból indulva a startszimbólum felé építjük a fát
- Input feldolgozása: balról jobbra
- Legjobboldali levezetés inverzét állítja elő
- Van olyan nyelvtan, amit ez támogat de a felülről lefelé elemzés nem
- LR(1) elemzés: Left to right, using Rightmost derivation
- Léptetés: következő token elhelyezése verem tetején
- Redukció: veremben lévő szabály jobboldal helyettesítése szabály baloldallal, közben a szintaxisfa bővítése
  - Nem feltétlenül csökkenti a verem méretét: ϵ szabály 1-gyel növeli
- Egy nyelvtan LR(k) tulajdonságú, ha legfeljebb k token előreolvasásával meghatározható, hogy léptetés vagy redukció jöjjön és redukció esetén a használandó szabály is kiderül
- LR(1) elemzés implementálása; léptetés vagy redukció eldöntése
  - Előreolvasás és pillanatnyi állapot alapján (tehát táblázat)

- Verem: tokeneket és állapotokat tárol (nem hagyományos VDA)
- Kezdő állapot: 0 (nincs hozzárendelt bemenet, csak állapot van)
- Bemenet lehet: terminális, nemterminális (kivéve S), input vége
- Elfogadás: $S \rightarrow ...$ szabály és input vége "találkozása" esetén
- Táblázat létrehozása nyelvtanból: nem a tananyag része
- Tétel: környezetfüggetlen nyelvtan pontosan akkor LR(1) tulajdonságú, ha a táblázat konfliktusmentesen kitölthető





	e	v	Input vége	N
0	Léptetés: 2	Hiba	Elfogadás: S→ε	I
I	Hiba	Léptetés: 3	Elfogadás: S→N	Hiba
2	Hiba	**Redukció: N→e**	Redukció: N→e	Hiba
3	Léptetés: 4	Hiba	Hiba	Hiba
4	Hiba	Redukció: N→Nve	Redukció: N→Nve	Hiba

- Ábra: Alulról felfelé elemzés, LR(1)
  - Oszlopok: bemenet vagy nemterminális
  - Sorok: pillanatnyi állapot
  - Cellák: mit kell tenni és hogyan (nemterminális esetén csak állapot)
    - Léptetés: verembe helyezni a bemenetet és az odaírt számot
    - Redukció: veremben szabály jobboldal helyettesítése baloldallal
      - Új "bemenet": a nemterminális (a szabály baloldal)
      - Új állapot: baloldali nemterminális oszlopából a szám, de előbb pop-olni kell a vermet: ott maradt állapot sora kell
    - Elfogadás: speciális redukció: készen vagyunk

3.5.3. Backtracking (extra, nem az anyag része)

- Lassú és nem tudjuk meg, hogy hol a hiba: annyit kapunk vissza, hogy sehogy sem sikerült illeszteni a bemenetet
- Ezen okokból nem használjuk fordítóprogramokra; vannak jobb megoldások

4. Szemantikus elemzés (semantic analyzer)

4.1. Bevezetés

- Feladat: statikus szemantika (pl. változó deklaráltsága, típusa) ellenőrzése
- Bemenet: szintaxisfa
- Kimenet: szintaxisfa attribútumokkal, szimbólumtábla (+hibák)
- Eszközök: attribútumnyelvtanok

4.2. Feladatok kategóriái

- Szimbólumtábla
  - Deklarációk feldolgozása
  - Azonosítószimbólumok deklarációhoz kötése
  - Hatókör, láthatóság szabályok ellenőrzése
    - Hatókör: hol létezik
    - Láthatóság: hol érhető el
- Attribútumnyelvtan
  - Típusellenőrzés, típuslevezetés
  - Típuskonverziók
- Warning-ok (pl. konstans nullával osztás)
  - Ezzel mi nem foglalkozunk

4.3. Szimbólumtábla

- Oszlopai:
  - Név: szöveg
  - Fajta: függvény, paraméter, lokális változó, stb.
  - Típus: $int \to void$, int, stb.
  - Deklaráció: pl. 1. sor, 10. oszlop
  - Használat: pl. 3. sor, 10. oszlop, ... (lista)
- Beszúrás: új deklaráció esetén
  - Beszúrás előtt mindig keresés van: újradeklarálás tiltása
- Keresés: szimbólum neve alapján kereshetünk
  - Használatát érdemes feljegyezni (pl. refaktorálás miatt)
- Implementáció veremként:
  - Beszúrás: tetejére (*push*)
  - Keresés: fentről (tetejéről) lefelé, első találatnál megállunk
- Blokk-index vektor (ez is egy verem)
  - Új blokk esetén elmentjük a szimbólumtábla-verem tetejét (*push*)
  - Blokkból kilépés esetén:
    - Törlés tábla-veremből (*pop*), amíg teteje nem a mentett érték
    - Lementett értéket kitöröljük a blokk-index vektorból (*pop*)
- Újradeklarálás megtiltása, de egyes elfedések megengedése
  - Újradeklarálás miatti kereséskor nem megyünk végig a szimbólumtáblán
  - Blokk-index teteje által mutattot deklaráció felett (exkluzív) keresünk
  - Üres a blokk-index vektor $\implies$ egész szimbólumtáblában keresünk

4.4. Típusrendszerek

- Statikus (fordítási időben történik a típusellenőrzés)
  - Futás időben nem kell típusinformációt tárolni
    - Kivétel: Java instanceof, C++ alosztályok
  - Sikeres fordítás esetén nem lehet futási idejű típushiba
    - Kivétel: fordítás és futás idejű könyvtár verziók eltérnek
  - pl. Ada, C++, Haskell
- Dinamikus (futási időben történik a típusellenőrzés)
  - Futási időben típusokat is kell tárolni, nem csak értékeket
  - Utasítások végrehajtása előtt típushelyességet ellenőrizni kell
  - Futási időben derülnek ki egyes hibák, cserébe flexibilisebb
  - pl. Lisp, Erlang

4.5. Ki adja meg a típusokat?

- Programozó adja meg: deklarációk típusozottak
  - Több lehetőség típusellenőrzésre
  - Egyszerűbb fordítóprogram, gyorsabb fordítás
- Típuskikövetkeztetés, típuslevezetés: deklarációk általában típus nélkül
  - Műveletek alapján fordítóprogram kitalálja a kifejezés típusát
  - Kényelmes programozni, de olvashatóságért érdemes típusozni
  - pl. Haskell

4.6. Típuskonverzió

- Egy kifejezés típusát változtatja meg
- Lehet automatikus (implicit) vagy manuális (explicit)
- Osztályhierarchia miatti típuskonverzió is van (szülő osztály metódusa)
- Típuskonverzió miatt az érték is megváltozhat (pl. int-ből double)

4.7. Attribútumnyelvtan

- Szintaxist leíró nyelvtan szimbólumaihoz attribútumokat rendelünk
  - Szemantikus elemzéshez kell (deklaráltság-, típusellenőrzés)
  - Kódgeneráláshoz kell (literál értéke, maga a kód)
  - Kódoptimalizáció számára is kell
- Szabályokhoz akciókat (programkódot) rendelünk
  - Szemantikus ellenőrzést végzünk
  - Kiszámoljuk pl. a szabály baloldal típusát a jobboldal típusaiból
  - Azonosító esetén a szimbólumtáblát használjuk:
    - Neve alapján lekérdezzük a szimbólumtáblát
    - Megnézzük, hogy deklarált-e és mi a típusa
  - Szimbólum többször szerepel szabályban: sorszámozzunk
- Szintetizált attribútum:
  - Szabály bal oldalán állt, amikor kiszámoltuk (bal oldal kap értéket)
  - Alulról felfelé közvetít információt a szintaxisfában
  - pl. kifejezés típusa
- Kitüntetett szintetizált attribútum: lexikális elemző ad neki értéket
  - Ehhez módosítani kell a lexikális elemzőt (ne csak tokent adjon)
  - pl. azonosító neve, literál értéke
- Örökölt attribútum (ciklus példával szemléltetve)
  - Start szabályoknál ($S \to ...$) hamisra állítani a jobb oldalt
  - Normális szabályoknál a bal oldal értékét örökli a jobb oldal
  - Ciklus szabály: igazra állítjuk a jobb oldal megfelelő szimbólumát
  - Fentről lefelé terjed az információ a szintaxisfában
  - Bisonban ilyen nincsen
    - Ezt S-attribútumnyelvtannak hívjuk (nincsenek örökölt attribútumok)
    - Örökölt attribútumok nem illeszkednek jól az LR elemzéshez
    - Megoldás: szimbólumtábla használata ilyenkor (nem tananyag)

- Megszorítások, elvek, szabályok
  - Csak az adott szabály bal és jobb oldalával foglalkozunk
    - Nem olvasunk a fán feljebb vagy lejjebb
    - Ha szükséges, akkor definiálunk szabályokat és segítségükkel egyesével hordozunk információt a szintaxisfa szintjein
  - Egy attribútumértéket csak egy akció határozhat meg
    - Ha több akció is ugyan azt az értéket írja felül, az baj
- Nem triviális, hogy az akciókat milyen sorrendben futtatjuk
  - Indoklás: ha ez egyik akció felhasználja egy másik akció eredményét...
- Attribútumok és kód generálás összefüggése
  - Generált kód is lehet egy attribútum
  - Kódgenerálásnál gondolni kell az attribútumok kiszámítási sorrendjére

5. Assembly

5.1. Bevezetés

- Assembly: alacsony szintű, hardverközeli programozási nyelvek csoportja
- Assembler: fordítóprogram assembly-ről gépi kódra
- Mi 32 bites, x86-os architektúrát, NASM szintaxisú assembly-t használunk

5.1.1. C használat (`extern` függvények)

- Pár függvény C-ben (`io.c`) van megírva egyszerűség kedvéért
  - Assembly-ben túl sok időbe tellene megcsinálni
- Függvények: `write_natural`, `read_natural`, `write_boolean`, `read_boolean`

5.1.2. Fordítás

- `nasm -felf Xyz.asm && gcc -m32 io.c Xyz.o -o Xyz`
- Xyz-t ki kell cserélni valami értelmes névre
- `-m32`: 32 bites mód (64 bites OS esetén is)

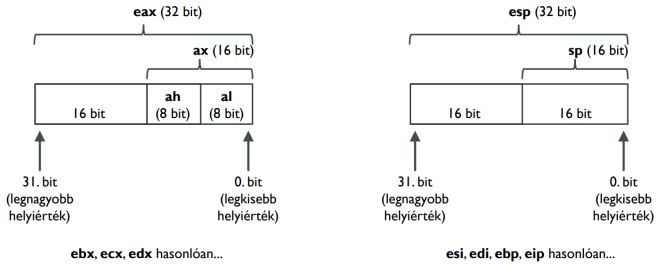
5.2. Általános információk

- Komment: ; után
- Méretek: byte (1), word (2), dword (4)
- abc|xyz jelentése: két memória/regiszter terület egymás után olvasandó
  - Szám esetén a magasabb helyi érték van bal oldalt (intuitív)

5.3. Adattárolás

5.3.1. Regiszterek

- Gyors, kis kapacitás, pillanatnyi művelet operandusai itt vannak általában
- Általános célú regiszterek: eax, ebx, ecx, edx, esi, edi
- Speciális regiszterek:
  - eip: instruction pointer, következő végrehajtandó utasításra mutat
  - eflags: jelző bitek (pl. aritmetikai művelet során van beállítva)
- C-ből hívott függvényben csak ezeket szabad módosítani: eax, ecx, edx
  - A tárgyban nekünk az összes kódunk ilyen
  - Ha mégis módosítjuk, vissza kell őket állítani
  - Összes regiszter mentése/visszaállítása: `pusha`, `popa`



5.3.2. Verem

- Itt van: fv paraméter, fv visszatérési érték, fv lokális változó (!!!)
- 0 felé nő, ezért így is lehet törölni (pop) belőle:`add esp,N`
- esp: stack pointer, verem tetejére mutat
  - Verembe push után csökken az értéke
- ebp: base pointer; aktív eljáráshoz tartozó adatokra mutat a veremben
- Műveletek: (mindkét esetben csak 2 vagy 4 bájtos lehet az operandus)
  - `push <regiszter/memória/konstans>`
  - `pop <regiszter/memória>`
- Ha marad benne valami: C program általában segfault-ol

5.3.3. Statikus memória

- Kezdőérték nélküli memóriaterület: `section .bss`
  - 42db double: `a: resd 42`
- Kezdőértékkel rendelkező memóriaterület: `section .data`
  - 42 kezdőértékkel double: `a: dd 42`
  - 'x' karakter kezdőértékkel bájt: `x: db 'x'`
- Memóriahivatkozás: `dword [a]` vagy `dword [a+4*ecx]`
  - Mindkét esetben 4 bájtról van szó `dword` miatt
  - Néha elhagyható a `dword`: amikor kikövetkeztethető a méret (mert pl. egy regiszter a másik operandus)

5.4. Programkód

- `section .text` alatt
- C program main függvénye: `main` címke; fájl tetején: `global main`
- Önálló assembly program esetén: `main` helyett `_start`
- Fájl tetején: `extern read_naural`, stb. (io.c használatához)
- `ret`: visszatér a függvényből, az érték: eax regiszter értéke (konvenció)

5.5. Műveletek

5.5.1. Általános tudnivalók

- Általában legfeljebb 1 memóriahivatkozás megengedett (alap: regiszter)
- Regiszterből implicit kiderül, hogy hány bájtról van szó
  - Ilyenkor a memóriahivatkozásnál nem kell odaírni
  - De oda kell írni pl. 1 konstans és 1 memóriahivatkozás esetén

5.5.2. Aritmetikai műveletek

- `add mihez,mit`
- `sub miből,mit`
- `inc mit`
- `dec mit`
- `mul mivel`: edx|eax = eax * <mivel>
  - legyen <mivel> 4 bájtos
- `div mivel`: alábbi egész osztást végzi el: edx|eax / <mivel>
  - Eredmény: eax=hányados, edx=maradék
  - Történhet adatvesztés, amikor pl. edx|eax nagy és <mivel>=1
- Előjeles számok esetén: `i` prefix, pl. `imul`

5.5.3. Összehasonlítás

- `cmp mit,mihez`: `eflags`-be menti az eredményt
- Felhasználható feltételes ugráshoz, feltételes mozgatáshoz, stb.

Jelentés	Szuffix	Kiírva	Kiírva 2	Szuffix 2
mit == mihez	e	equal		
mit != mihez	ne	not equal		
mit < mihez	b	below	not above equal	nae
mit >= mihez	nb	not below	above equal	ae
mit > mihez	a	above	not below equal	nbe
mit <= mihez	na	not above	below equal	be

5.5.4. Ugrás (feltétel nélküli, feltételes ugrás)

- Címkére szokás ugrani
- `jmp hová`: feltétel nélküli ugrás
- `j<szuffix> hová`: feltételes ugrás, előtte `cmp` szükséges
  - `j<szuffix> near hová`: 1 bájt helyett 4-en tárolja a távolságot
  - Ezt a `near` változatot használjuk kódgeneráláshoz!
  - Sima `jmp` 4 bájtot használ, ott nem kell `near`

5.5.5. Adat mozgatás, másolás (feltétel nélküli, feltételes)

- Valójában másolás: régi érték úgy marad
- `mov hová,honnan`: feltétel nélküli mozgatás
- `cmov<szuffix> hová,honnan`: feltételes mozgatás

5.5.6. Bitműveletek

- `and mihez,mit`
- `or mihez,mit`
- `xor mihez,mit`
- `not mit`: ez lehet memóriaterület és regiszter is

5.5.7. Függvény hívás és visszatérés

- `call cimke`: "függvény" hívás
  - Következő utasítás címét (`eip`) beteszi a verembe
  - Az operandusként adott címkéhez ugrik a vezérlés
- `ret`: veremből pop (ami az `eip` volt) és oda ugrik a vezérlés
- Paraméterátadás:
  - `call` előtt verembe push-olunk, utána pedig pop-olunk
  - Függvényen belül `[esp+4]`-től olvasunk a veremből (`mov`-val)
    - Hiszen `esp`-nél 4 bájton az `eip` régi értéke található
- C függvény hívása:
  - Címke (amihez ugrunk) a függvény neve
  - Paramétereket fordított sorrendben tesszük a verembe, tehát az első paraméter lesz a verem tetején
    - Hogy pl. `printf` megszámolhassa paramétereinek számát
  - Visszatérési érték: `eax` regiszterben
    - Nem fér bele: általában memóriába kerül, `eax`-ben pointer

5.6. Copy-paste-elhető sablonok

5.6.1. Egyágú elágazás

```
cmp eax,ebx
jne vege
... ; Végrehajtva, ha eax == ebx
vege:
```

5.6.2. Kétágú elágazás

```
cmp ebx,ebx
jne nemegyenlo
... ; Végrehajtva, ha eax == ebx
jmp vege
nemegyenlo:
... ; Végrehajtva, ha eax != ebx
vege:
```

5.6.3. Számláló ciklus

```
mov ecx,0 ; 10-től számolunk felfelé, ecx a számláló
eleje:
cmp ecx,10
je vege
...         ; Ciklusmag
inc ecx
jmp eleje
vege:
```

6. Kódgenerálás

6.1. Bevezetés

- Feladat: alacsonyabb szintű reprezentációra (végül tárgykódra) alakítás
- Bemenet: szintaxisfa attribútumokkal, szimbólumtábla
- Kimenet: tárgykód
  - Gépi kódot generálni követlenül csak nagyon indokolt esetben
  - Jobb alternatíva: LLVM, Assembly, C, Java bájtkód, stb.
  - Transzláció: magas szintű nyelvek közötti fordítás
- Eszközök: kódgenerálási sémák

6.2. Kódgenerálás attribútumnyelvtannal

- Eddig típusellenőrzéshez használtunk attribútumnyelvtanokat
- Most generált kódot is attribútumként kezeljük
- Szimbólumtáblában label-t (memóriacímkét) tárolunk minden változóhoz

6.3. Generált kód

- Kifejezés visszatérési értéke: `eax` regiszterben (pl. szám esetén)
- Szimbólumtáblában memóriafoglalásokhoz szükséges kód: start szabályban
  - Kezdőérték nélküli változók lesznek (méretük típustól függ)

6.4. Végeredmény

```
global main
extern <io.c-ből külső címkék...>

section .bss
<változók...>

section .text
main: <programutasítások...>
      mov eax,0
      ret
```

7. Gyakorlati jegyzet

- Figyeljünk a megfelelő újsor karakterre: LF kell (CRLF nem jó)
  - Főleg teszt fájlnál jön elő, "Unexpected character: " formában

7.1. `while.cc`

- C++ forrásfájl, mi írjuk kézzel, gcc-nek adjuk át
- A `while` program belépési pontja is itt található
- Pár flex/bison dolog itt van implementálva
- Itt lehet a kitüntetett szintetizált attribútumoknak értéket adni

7.2. `while.l`

- Lexikális elemzéshez szükséges, flex-nek adjuk át
- Token regexek és hozzájuk tartozó C++ kód található itt
  - Nem kötelező a kódban return-t használni
  - Lexikális elemzéshez elég lenne stdio-ra kiírás is
  - Token-ek a `while.y` fájlban vannak definiálva
- Regex-ek feldolgozása:
  - Először a leghosszabb illeszkedést keressük meg
  - Ha ez többre is match-el, akkor sorrendben az elsőt vesszük
  - Más esetben nem számít a sorrend
- Használt függvények:
  - `YYText()`: mire illeszkedett a regex
  - `lineno()`: hányadik sorban vagyunk
    - Be kell (be van) kapcsolni: `%option yylineno`

7.3. `while.y`

- Szintaktikus és szemantikus elemzéshez szükséges, bison-nak adjuk át
- Környezetfüggetlen nyelvtan van benne
  - Terminális: nagybetűs
  - Nemterminális: kisbetűs
  - Sorrend akkor számít, ha nem egyértelmű a nyelvtan (ami hiba)
- Lehetséges hibák, konfliktok:
  - shift/reduce conflicts: nem egyértelmű a nyelv
- Tokeneket ebben a fájlban definiáljuk
  - Több definíció is kerülhet egy sorba
  - `%token ...`: normál token, sorrend nem számít
  - `%left ...`: bal asszociatív token
  - `%precedence ...`: prefix operátor
  - Ahol számít a sorrend: minél lejjebb, annál erősebb
- Attribútumok (szemantikus elemzéshez):
  - `%token <std::string> T_ID`: egy kitüntetett szintetizált attr.
  - `%token <type> expression`: egy szintetizált attr.
    - Típusok a `implementation.hh`-ban vannak definiálva (enum)
  - C++ kódban a 2. (jobb oldali) szimbólum értéke `$2`, helye `@2`
  - C++ kódban a bal oldal értéke (írásra): `$$ = ...`
- Emlékeztető: bison a szabály jobb oldalát cseréli le a bal oldalra
  - Hasonlóan: `$$` értékét jobb oldali értékekből számoljuk (nem fordítva)
- Emlékeztető: deklarációk nem attribútumokkal vannak típusozva, hanem szimbólumtábla segítségével (ami az `implementation.hh` része)

7.4. `implementation.hh`, `implementation.cc`

- Szemantikus elemzéshez szükséges, de csak a gcc-nek adjuk át
- Típusok itt vannak definiálva (egy enum-ként)
- Szimbólum tábla itt van definiálva (`std::map<std::string, type>`)

7.5. Assembly integráció, kódgenerálás

- expression attribútum: régihez hasonlóan type + a generált kód
- command, commands attr: generált kód
- szimbólumtábla: címkét is tárol
- start szabály: stdout-ra kiírjunk az assembly forráskódot
- Kifejezések eredménye
  - `natural` eredménye: eax regiszterben
  - `boolean` eredménye: al regiszterben (1=igaz, 0=hamis)
- `make && ./while MY-TEST.test > test-out.asm && nasm -felf test-out.asm && gcc -m32 io.c test-out.o -o test-out && ./test-out`

7.5.1. Új típus létrehozása

- while.l: típus és literál token létrehozása
- while.y:
  - start szabály: lefoglalt memória méretét be kell állítani
  - declaration: típus tokenhez
  - expression: literál tokenhez
  - értékadást és érték kiolvasást ki kell egészíteni
- while.cc: kitüntetett szintetizált attribútumnak (literálnak) értékadás
- Akár több regiszter is használható az adat tárolására (de bonyolít)