4. fejezet

Fordítók felépítése

Grafikus Processzorok Tudományos Célú Programozása

Berényi Dániel – Nagy-Egri Máté Ferenc

Fordítók

Kézzel assembly kódot írni nem érdemes, mert:

- Egyszerűen nem skálázik
  nagy problémákhoz arányosan sok kódot kell megírni, kevés absztrakció van
- Nem biztonságos, nehezen olvasható, nehezen érthető, nehezen karbantartható
- Nem hordozható
  másik hardverre (pl. mikrokontroller, ARM alapú telefon) újra kell írni

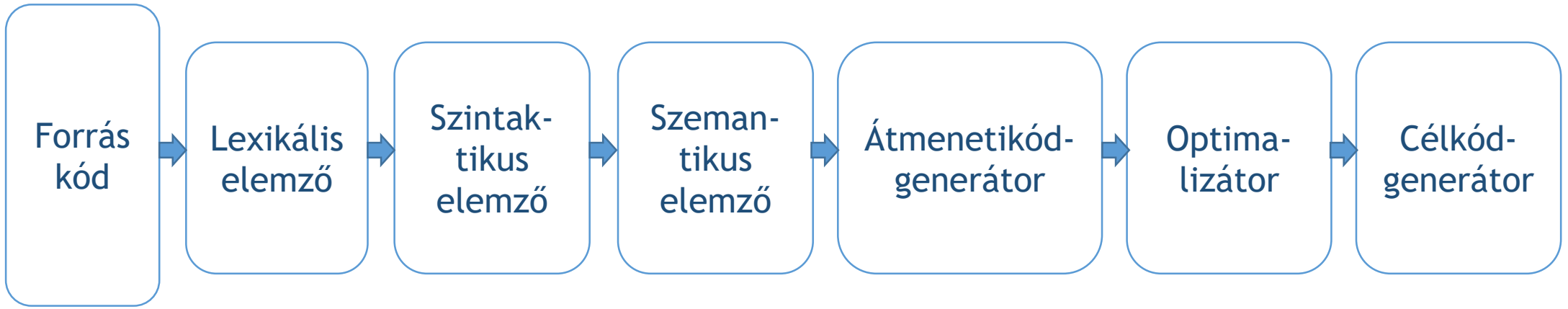
Fordítók

Inkább:

Magasabb szintű nyelveket használunk, amelyekhez tartozó fordítók képesek a magas szintről előállítani a gépi kódot.

Fordítók

A fordítók viszonylag bonyolult programok.
Ahhoz, hogy jól ki tudjuk használni a képességeiket főleg optimalizációk terén, célszerű ismerni a felépítésüket



Fordítók

Egy fordítót megírni bonyolult. Mégis, szinte minden nap létrejön egy új programozási nyelv...

Hogyan?

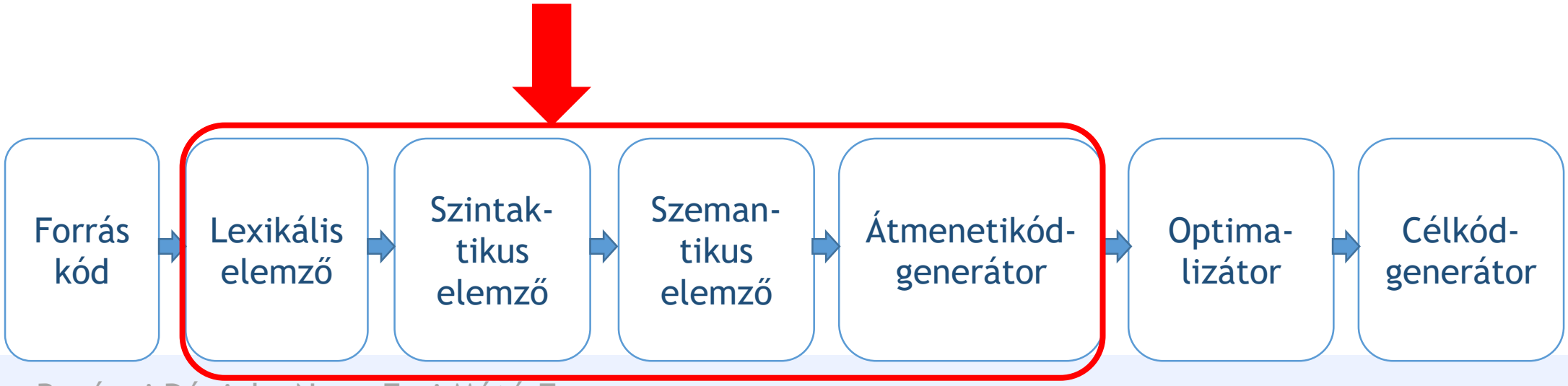
A válasz: modularitás!

Fordítók

A magas szintű nyelvek elvonatkoztatnak a hardver részleteitől
Pl.: regiszterek száma és neve, utasításkészlet, ...

Másrészt a nyelv lényegében a szintaxis + szemantika

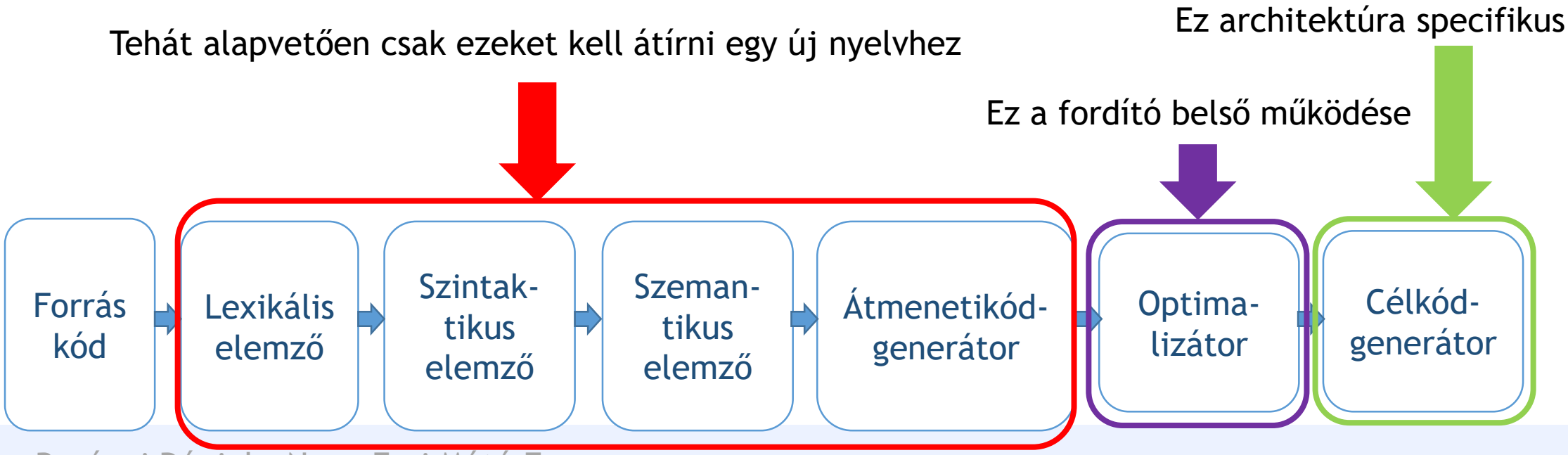
Tehát alapvetően csak ezeket kell átírni egy új nyelvhez



Fordítók

A magas szintű nyelvek elvonatkoztatnak a hardver részleteitől
Pl.: regiszterek száma és neve, utasításkészlet, ...

Másrészt a nyelv lényegében a szintaxis + szemantika



Fordítók

Ebből az is következik, hogy az optimalizációk minősége

(pl.: mennyire hatékony kódot generál, milyen jól vektorizál)

csak a nyelvek kifejező képessége miatt térhet el!

Optimalizációk

Néhány optimalizáció, amit a fordítók általában támogatnak:

- Ciklusokon:
  indukció elemzés, szétbontás, egyesítés, megfordítás, megcserélés, invariánsok kiemelése, egymásba ágyazások optimalizációja, kifejtés
- Kifejezéseken:
  közös alkifejezések kiemelése, konstansok műveleteinek kiértékelése, betöltés/kimentés átrendezése
- Algebrai és lebegőpontos formulákon:
  +, * asszociativitása, disztributivitása, kiemelés, osztás helyett reciprokkal szorzás, kommutativitás
- Utasítás szintű párhuzamosság:
  SSE/AVX automatikus generálása

Optimalizációk

Általában nem kell tudni túl sok részletet, csak engedélyezni az optimalizációkat...

Persze a fordító csak azokat az átalakításokat tudja megtenni, amiket ismer / felismer.

Optimalizációk

Érdemes tudni, hogy:

- A fordítók átlátnak az osztályok, struktúrák létrehozásán, tagjainak a hozzáférésén, felszabadításán addig, amíg mindez a stack-en történik!

- Ez azt is jelenti, általában hogy nem kell feláldozni a kód olvashatóságát a sebességért!

Optimalizációk

Érdemes tudni, hogy:

- a pointerek, new/delete/dinamikus allokációk, volatile változók, virtuális függvényhívások, beolvasás perifériákról, adatfolyamokból (fájlok, hálózat) megakadályozza, hogy a fordító érveljen, átlásson a konstrukciókon és optimalizálhasson.

  Habár a legtöbb téren, különösen a virtuális függvényhívások optimalizációján állandó fejlesztések történnek!

Optimalizációk

Érdemes tudni, hogy:

- A fordító csak azt a kódot érti maximálisan, amiről tudja mit jelent!

Intrinsic függvények!

Intrinsic függvények

Intrinsicnek nevezünk egy olyan konstrukciót (általában függvényt), amelynek kezeléséről, tulajdonságairól a fordító extra információkkal rendelkezik.

Pl.:

- SSE/AVX utasítások
- Speciális matematikai függvények
- Adatmozgatás és bitműveletek

A speciális matematikai függvények általában jelentősen átfednek a nyelvek standard beépített függvényeivel. Mindig ellenőrizzük a fordító leírását!

Intrinsic függvények

Mi a különbség egy intrinsic függvény és egy kézzel megírt változat között?

- A fordító pontosan tudja, milyen gépi kódot kell generálnia belőle
- A pontosság általában az utolsó jegyig garantált
- Valószínűleg tudja kezelni / ismeri a speciális értékeit
- Ismerhet algebrai átalakításokat hozzá
- Speciálisan hatékony implementációi lehetnek egyes adattípusokra
  int, float, double, ...

Egy kézzel írt esetben ezek egyike sem teljesül!

Intrinsic matematikai függvények

Egy részleges lista az általában támogatott matematikai függvényekről:

```
abs, acos, asin, atan, atan2, ceil, cos, cosh, exp,
fabs, floor, fmod, ln, log10,
memory {compare, copy, set},
pow, rot, sin, sinh, sqrt,
string {concat, cmp, cpy, length},
tan, tanh
```

Statikus kódanalízis

A statikus kódanalízis egy speciális fajta fordítás, amikor a fordító kifejezetten olyan algoritmusokat futtat, amik azokat a gyakori fejlesztői hibákat próbálják megtalálni, amiket a nyelv egyébként nem szűr ki.

Számos fordító támogat valamilyen szintű statikus analízist.

Általában segítenek megtalálni:

- Az inicializálatlan változókat, érvénytelen referenciákat
- Null- vagy érvénytelen pointereket
- Dupla felszabadítást delete-nél, delete utáni használatot, vagy fel nem szabadított dinamikus memóriákat

Kapcsolók...

Egyszer az életben, szánjuk rá az időt, és olvassuk el a fordító kapcsolóit, beállításait:

- Hogyan kell engedélyezni az optimalizációkat?
- Hogyan kell bekapcsolni a vektorizációt és a kiterjesztett utasítás készletek használatát (SSE, AVX)
- Milyen intrinsic függvények vannak támogatva?
- Hogyan hatnak az optimalizációk a numerikus pontosságra? (fast vs. precise)
- Hogyan kapcsoljuk be a statikus analízist?

Kapcsolók...

Clang:

http://clang-analyzer.llvm.org/docs/UsersManual.html

Gcc:

https://gcc.gnu.org/onlinedocs/

MSVC:

https://docs.microsoft.com/en-gb/cpp/build/building-c-cpp-programs

https://docs.microsoft.com/en-gb/cpp/build/reference/compiler-options