sallang

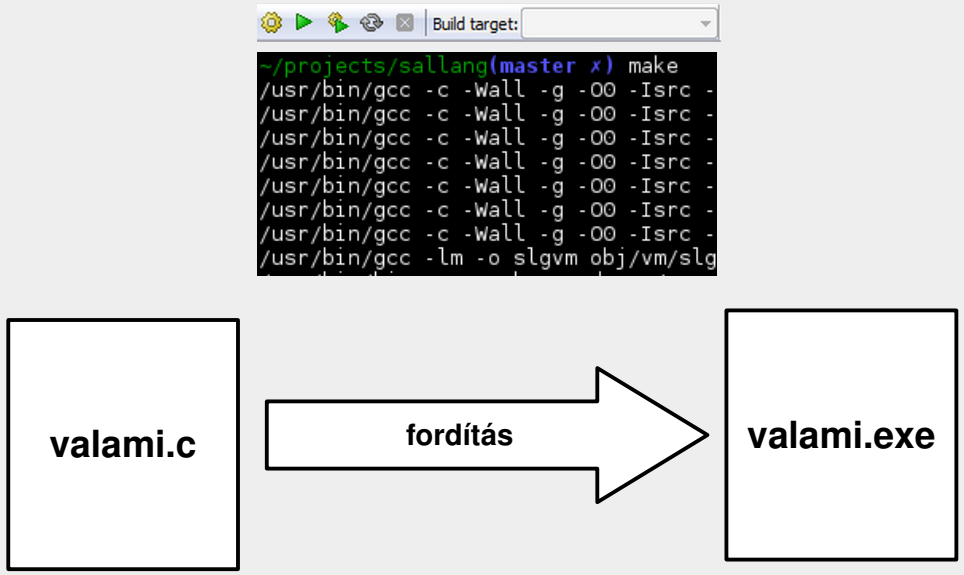
avagy

Fordítótervezés dióhéjban

Sallai Gyula

- Az előadás egy kis példaprogramon keresztül mutatja be fordítók belső lelki világát
- De mit is jelent, az hogy fordítóprogram?

Mit csinál egy fordító?



A fordító feladata a kényelmes programozási nyelvet (pl. C) gépi kódra (pl. x86) fordítani.

Egy fogaskerék ikonnal vagy egy paranccsal meghívható, nem igazán látjuk, mi történik a motorháztető alatt.

Programozási kihívások terén egészen sokat tud nyújtani egy fordító írása:

- Modellezés, tervezés
- Nyelvi elemzés
- Adatszerkezetek, hatékony algoritmusok
- Optimalizálás
- Számítógép-architektúrák

sallang – Saját progamozási nyelv

```
let
    var a: int;
    var i: int;
    var oszto: int;
    var vanoszto: bool;
in
    a = 15;
    vanoszto = false;
    i = 2;

    while (!vanoszto && i < a) {
        if (a % i == 0) {
            vanoszto = true;
            oszto = i;
        }

        i = i + 1;
    }

    print(vanoszto);

    if (vanoszto) {
        print(oszto);
    }
end
```

- Egyszerű szintaxis
- Alap vezérlési szerkezetek
- Egész és logikai típus
- Saját virtuális gép a végrehajtáshoz

A példa fordító még alfa-előtti állapotban szerepel, jelenleg csak a fontos vezérlési szerkezetek (if, else, while) működnek, illetve csak egész és logikai típusokkal tudunk dolgozni.

A dián egy sallangban írt program látható, ami meghatározza egy szám legkisebb valódi osztóját.

A program két részből épül fel: deklarációs blokk és utasítás blokk. A deklarációs blokkban nem deklarált változókat nem használhatjuk.

A végrehajtást nem igazi CPU-n végezzük, erre van egy saját virtuális gép. Így a nyelv működése hasonló a Java-hoz.

Lexikális elemzés

```
    while (!vanoszto && i < a) {
        if (a % i == 0) {
            vanoszto = true;
            oszto = i;
        }

        i = i + 1;
    }
```

```
T_WHILE, '(', '!', T_ID, T_AND, T_ID, '<', T_ID, '{',
  T_IF, '(', T_ID, '%', T_ID, T_EQ, T_INT_LITERAL, ')', '{'
    T_ID, '=', T_BOOL_LITERAL, ';',
    T_ID, '=', T_ID, ';'
  '}'

  T_ID, '=', T_ID, '+', T_INT_LITERAL, ';'
'}'
```

A lexikális elemzés során eltávolítjuk a minket nem érdeklő részeket (whitespace, kommentek, stb.) a releváns szabad karaktereket pedig sokkal kényelmesebben kezelhető tokenekké alakítjuk.

A tokenek együttes jelentéssel bíró karaktersorozatok, azaz pl. a “while” karaktersorozatot nem így, hanem egyetlen konstansként (T_WHILE) tudjuk kezelni.

A képen látható, hogy milyen tokenfolyam keletkezik a forrásfájlból, a formázást az olvashatóság kedvéért hagytam meg.

Nyelvtani elemzés

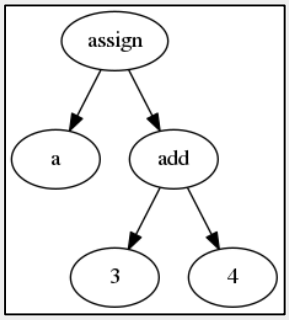
```
id:
    T_ID;

scalar:
    T_INT_LITERAL
    | T_BOOL_LITERAL;

expr:
    id
    | scalar
    | id '=' expr
    | expr '+' expr
    | expr '-' expr
    ...
    ;
```

a = 3 + 4

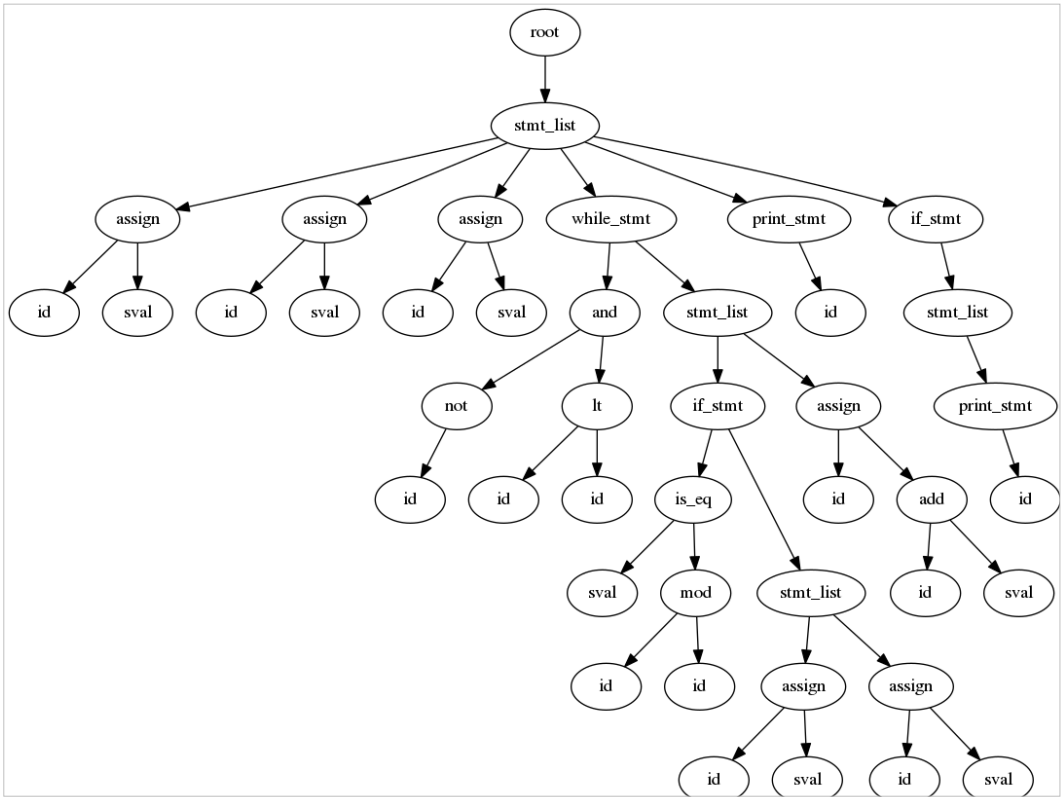
```
T_ID,'=',
  T_INT_LITERAL,
     '+',
  T_INT_LITERAL
```



A tokenekből hozunk létre egyre komplexebb szabályokat. Pl. Előírjuk, hogy a *skalár* vagy egész szám literális vagy logikai literális token. Az azonosító *(*itt: *változónév)* egy azonosító token. Ezután előírjuk, hogy a kifejezés vagy egy skalár, vagy egy azonosító, illetve egy vagy több kifejezés és valamilyen operátor.

Az illesztés után elkezdünk egy fát építeni. Az egyes kifejezések gyermekei az operandusai lesznek.

Ezzel a módszerrel egy teljes forrásfájlt le tudunk írni.



A szintaktikai elemzés után egy ehhez hasonló fát építünk a programunkból.

Ezt az adatszerkezetet Absztarkt Szintaxisfának (AST) hívjuk, mert már kevésbé függ a forrásnyelvtől (hasonló szemantikájú, más szintaxisú nyelvből is építhető ugyanilyen AST).

Szemantikus elemzés

- Van egy szintaktikailag helyes szövegünk, de értelmes-e?
- Deklaráltuk a használt változókat?
- Megfelelő típusok közt végzünk műveleteket?

```
let
    var a: int;
    var b: bool;
    var c: int;
in
    a = 3 + 4;
    b = 4;
    c = a;
end
```

```
~/projects/sallang(snode x) ./slgc docs/examples/semantic-error.sl
error in line 7: Type mismatch: Expected 'bool', found 'int'.
Compilation failed.
```

A következő lépés a szemantikus elemzés. Bár tudjuk, hogy szintaktikailg helyes programot írtunk, még nem lehetünk biztosak benne, hogy nincs tele értelmetlenséggel (típusosság megsértése, ismeretlen változók használata, stb.)

Szimbólum tábla

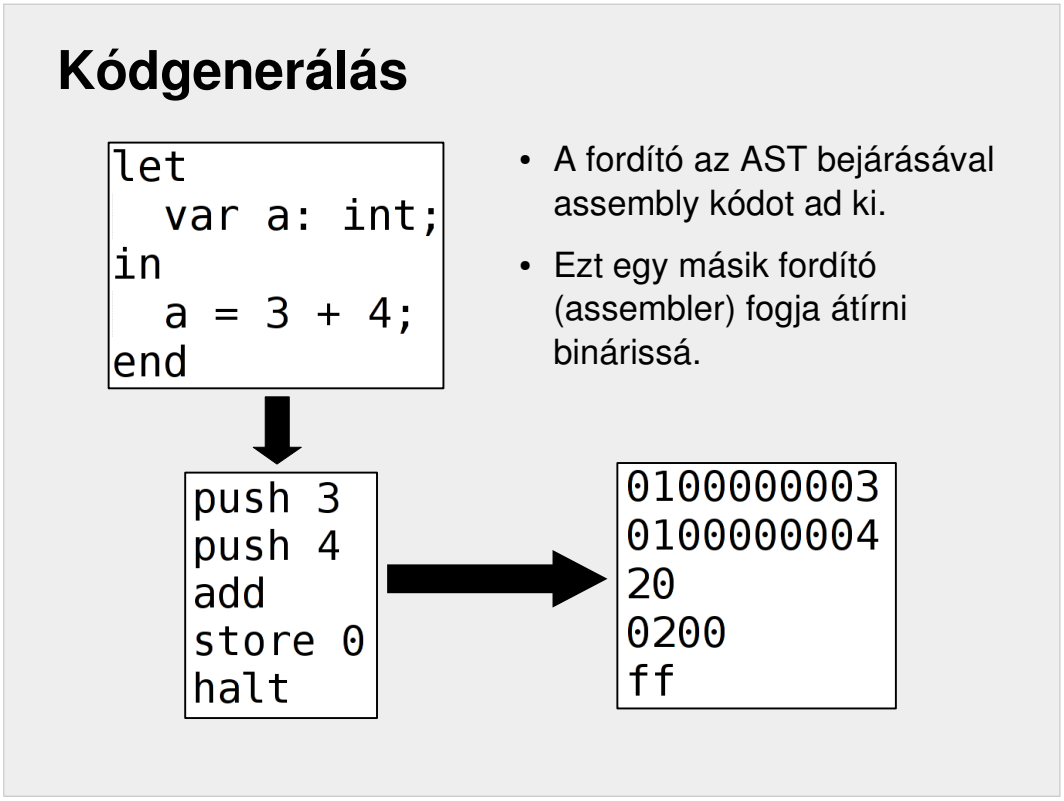
```
let
    var a: int;
in
    b = 2;
    print(b + 4);
end
```

- Honnan tudjuk, hogy egy változót deklaráltunk-e már?
- Tárolnunk kell a már látott azonosítókat.
- Ha kétszer próbáljuk ugyanazt a változót deklarálni vagy deklarálatlan változót használni – szemantikai hiba!

```
~/projects/sallang(snode ✗) ./slgc docs/examples/undeclared-var.sl
error in line 4: Use of undeclared variable 'b'.
Compilation failed.
```

A tárolt azonosítókat egy szimbólum táblának nevezett adatszerkezetbe rakjuk. Minden látott azonosító bekerül ide, az itt nem szereplő azonosítók használatakor a fordító hibát jelez.

Legjobban egy hash táblával lehet megvalósítani, ahol a kulcs az azonosító neve, az érték pedig az ahhoz tartozó információk.



Kódgeneráláskor az AST-t járjuk be, a megfelelő csúcsoknál kiadjuk a megfelelő utasítást. (Pl.: *add* típusú csúcsnál a virtuális gép *add* utasítását adjuk ki. Feltételnél pedig a feltétel kiértékelése után egy feltételes ugrást.)

A generált assembly kódot ezután az assembler átfordítja végül binárissá, a számítógép nyelvére. Ezután végre tudjuk hajtani a programunkat.

A példa egy saját, egészen kellemesen használható virtuális gépre fordít, amely nagyon sok enyhítést alkalmaz (közvetlen *print* utasítás, “végtelen” számú regiszter, stack alapú műveletvégzés, stb.)