



Nyelvi elemző keretrendszer fejlesztése

Nagy Gergely

Nyelvi elemzés

Szövegek feldolgozása

- Gyakori feladat szabályos szövegek feldolgozása:

- címek bekérése

Magyar tudósok körútja 2.
Budapest
1117

- algebrai kifejezés kiértékelése

$a = 2 * x * (3 + 4 / y)$

- programkód értelmezése

```
for (int i = 0; i < 15; ++i) cout << i;
```



Egy informatikus munkája szöveges elemzők helyes működésén alapszik.

A feldolgozási feladat

- A feladat tulajdonképpen **három lépésből áll:**

- 1 a szöveg elemzése



megfelel-e egy adott szabályrendszernek (nyelvtan)

- 2 köztes reprezentáció építése és transzformációja



adatszerkezet, ami jellemzi a szöveget (tartalmilag és formailag) és alkalmas a további feldolgozásra (pl. szintaxisfa)

- 3 kimenet előállítás



a feldolgozás célja – a program által előállított eredmény

A feldolgozási feladat (folytatás)

Ez a szerkezet megfigyelhető kis és komplex programoknál egyaránt:

zenekatalógus házfeladat

- 1 zenei albumok beolvasása
- 2 láncolt lista építése
- 3 kigyűjtés (pl. legfrisebb magyar előadók által készített metal albumok)

LLVM

- 1 adott programnyelv feldolgozása (**frontend**)
 - ♣ *C/C++ (Clang), Delphi, C#, CommonLisp, ...*
- 2 köztes reprezentáció (**IR – intermediate representation**) előállítása és optimalizálása
 - ♣ *gépfüggetlen assemblyszerű alacsony szintű, RISC-jellegű leírás*
- 3 adott architektura gépi kódjának előállítása (**backend**)
 - ♣ *x86, AMD, ARM, NVIDIA PTX, MIPS, PowerPC, ...*

A feldolgozási feladat (folytatás)

- A program felhasználói csak az 1. és a 3. réteggel találkoznak
- A közbenső réteg
 - rejtve marad
 - a szerkezetét és működését nem kell megérteni
- LLVM-nél ez igaz
 - a programozóra
 - egyedi frontend/backendet készítőjére
- Egy **feldolgozó keretrendszer** jellemzői:
 - a bemeneti és kimeneti formátumok **deklaratív leírása**
 - **nem kell foglalkozni:**
 - formális nyelvi elemzés
 - belső adatszerkezet
 - generálás mechanizmusa

Az elemzési feladat

- Ez az előadás az első réteggel, a **nyelvi elemző** elkészítésével foglalkozik
- Cél: biztosítani egy deklaratív leírásmódot a nyelvi szabályok számára



A szabványos leírási módokhoz hasonló szintaktikával adhatóak meg a nyelvtanok

- Meg kell vizsgálni, hogy a szöveg megfelel-e a szabályrendszernek:

- **igen** → egy jól kezelhető adatszerkezet építünk



Egy szintaxisfát (AST – abstract syntax tree) épít, ami jellemzi a szöveg szerkezetét és tartalmazza a szöveg által hordozott információt

- **nem** → jelezzük a hiba helyét és az eltérést a várt alaktól



Megadja a hibás szövegrészletet, jelzi a hiba pozícióját és megmondja, hogy melyik szabály volt az, ami nem teljesült

Létező elemző keretrendszerek

Nyelvi könyvtárak

- A legtöbb nyelvhez készített könyvtár nem tartalmaz nyelvi elemzőt
- A Python nyelvhez kapjuk a `pyparse`-t

Független könyvtárak

- Két csoportba sorolhatóak:
 - 1 kódot generáló könyvtárak (`yacc`, `lex`, `Flex`, `Bison`)
 - + nyelvtan megadása szabványos formában (BNF, EBNF)
 - kevert nyelvű bemenet & generált kód nem illeszkedik a projektbe
 - 2 a nyelvtant nyelvi elemekkel leíró könyvtár (`Spirit`, `pyparse`)
 - + a program szerves része az elemző és maga a nyelvtan is

Nyelvtanok formális megadása

- A nyelvi elemzés első lépése a nyelvtan formális leírása – például:

```
<közterület neve> <közterület típusa>
<szám>
<település neve>
<irányítószám>
```

EBNF (Extended Backus-Naur Form)

- Szabványos, sok eszköz támogatja.

```
kifejezés ::= összeg
összeg ::= szorzat (('+' | '-') szorzat)*
szorzat ::= tényező (('*' | '/') tényező)*
tényező ::= szám | zárójeles
zárójeles ::= '(' kifejezés ')'
```

Rekurzív alászálló elemzés

Minden lépésben a következőt tesszük:

- megvizsgáljuk, hogy a **soronkövetkező karakter megfelel-e** az aktuális szabálynak
- ha a szabály további szabályokból áll → azokat vizsgáljuk tovább
 - ha egy karakter megfelel → **továbbmegyünk** a következő karakterre
 - ha nem találtunk egyezést →
 - **visszalépünk** az előző szabályra
 - ha van **alternatíva**, azt is megvizsgáljuk
- **Minden szabály egy logikai függvény:**
 - illeszkedés → lépünk a következő karakterre és vissza: IGAZ
 - nincs illeszkedés → nem léptetünk és vissza: HAMIS
- **Főszabály:** van egy szabály, ahonnan az elemzés indul (az aritmetikai példában: kifejezés)

Tokenizálás és elemzés

- Sok elemzőeszköz két részre bontja a feladatot:
 - 1 tokenizálás:
 - karakterszintű vizsgálat – a nyelv szintaktikai elemeit ismeri fel
 - absztrakt elemek (token) sorozatát adja tovább
 - a szóköz karaktereket feldolgozza, elnyeli
 - 2 elemzés:
 - a tokenek sorozatát kapja bemenetként
 - ezeknek próbálja megfeleltetni a szabályokat
 - szóközőkkel és egyéb karakterekkel már nem kell foglalkoznia
- A bemutatott elemző nem így működik:
 - nem tudunk teljesen elrugaszkodni a karakterszinttől
 - + nem esik szét két részre a nyelvtan – egy egység marad

Rekurzív alászálló elemzés C++11-ben

Egy konkrét elemző megvalósítása

- Megvizsgálunk egy **konkrét elemzőt**.
 - A keretrendszer az itt megismert módszert alkalmazza
 - Ugyanazok a mechanizmusok működnek majd csak absztraktabb szinten
 - Ez az előadás arra is szeretne példa lenni, hogy egy **algoritmusból hogyan lehet keretrendszert** készíteni
-
- Minden **nyelvtani szabálynak egy függvény** felel majd meg
 - A szabályok (és így a függvények) **tetszőleges mélységben hivatkozhatnak** egymásra
 - Ez adja a rekurzív jelleget
 - Az **alászállás jelentése**: ahogy egyre mélyebbre megyünk a függvényhívásokban, úgy közeledünk a szöveg teljes feldolgozásához

Elemzési pozíció és intervallum

- Az elemzést `std::string`eken végezzük
- Két fogalom leírására is használjuk az alábbi típust:

```
using match_range =  
    std::pair<  
        std::string::const_iterator,  
        std::string::const_iterator  
    >;
```

- **Kontextus (context):** az elemzett intervallum, aminek az alsó határát folyamatosan léptetjük
- **Találat (result):** sikeres illesztés esetén a megtalált szövegrészlet határait tartalmazza

Egy karakter fedolgozása

- Egy karakterre illeszkedő függvény
- Az elfogadott karakterhalmazt egy `std::string`ben adhatjuk meg.

```
bool character(match_range &context, std::string const &values,
               char &result) {
    if (context.first == context.second) return false; // (1)


    for (char c : values) {
        if (c == *context.first) { // (2)
            result = c;
            ++context.first;
            return true;
        }
    }

    return false; // (3)
}
```

Egy karakter feldolgozása (folytatás)

A fenti kód egy általános szabály 3 fontos elemét tartalmazza:

- 1 **Leállási feltétel:** ha elértünk a kontextus végére, akkor álljon le az elemzés
- 2 **Sikeres illeszkedés:** ilyenkor a teendőink:
 - a találat elmentése
 - a kontextus-mutató léptetése
 - visszatérés IGAZ értékkel
- 3 **Sikertelen illesztés:**
 - a kontextus-mutatót nem állítjuk el



A sikertelen illesztés nem feltétlenül jelent hibát. Gondoljunk az alternatívákra!

 - visszatérés HAMIS értékkel

Egy karakter feldolgozása (folytatás)

A `match_range` értéke inicializáláskor:

```
std::string s = "4543987543987a";  
               ^           ^  
match_range context = { s.begin(), s.end() };
```

A `match_range` értéke egy `character()` hívás után:

```
character(context, "345", c);  
               ^           ^  
context.first context.second
```

Összetett szabály

- Érdemes megnézni egy **összetett (nem karakterszintű) szabályt** is.
- Ez a kifejezéses nyelvtan **zárójeles** szabálya:

```
bool brace(match_range &context, int &result) {
    match_range local = context; // (1)
    int tmp;
    char c;

    if (character(local, "(", c) && expression(local, tmp) &&
        character(local, ")", c)) { // (2)
        context = local; // (3)
        result = tmp;
        return true;
    }

    return false; // (4)
}
```

Összetett szabály (folytatás)

- 1 **Lokális kontextus** szükséges, mert a meghívott szabályok elállíthatják
- 2 **Több szabály hívása – ÉS kapcsolat:**
 - sorban hajtódnak végre
 - csak akkor megy tovább, ha az előtte lévők IGAZ-zal tértek vissza
 - → ez az **összefűzés** művelete
- 3 **Sikeres illeszkedés:**
 - a megkapott (globális) kontextus átállítása a lokális alapján
 - a találat elmentése
 - visszatérés IGAZ értékkel
- 4 **Sikertelen illesztés:** visszatérés HAMIS értékkel

Összetett szabály (folytatás)

- Nincs leállási feltétel – **nem ellenőrizzük**, hogy elértük-e a szöveg végét
- Az összetett szabályok **mindig meghívnak további összetett szabályokat vagy közvetlen karakterszintű szabályokat**
- A hívási láncban **legalul mindenképp karakterszintű szabályok vannak**
- Ezért elegendő ezekben, az ún. **terminális szimbólumokra** illeszkedő szabályokban ellenőrizni
- Az összetett szabályokat **nem-terminális szimbólumoknak** hívják

Logikai operátorok az elemzőkben

- Ahogy láttuk, az **ÉS** kapcsolat az összefűzést, egymásrakövetkezést fejezi ki
- Hasonlóképp a **VAGY** az alternatívák elemzésére használható:

```
if (number(context, num) || brace(context, result)) ...
```


- Fontos, hogy **vegyes logikai kifejezéseket nem szabad alkalmazni**:

```
r1(c) && r2(c) || r3(c)      // (ROSSZ!)
```

- A fenti kifejezés elvben vagy r1-re és r2-re illeszkedik egymás után, vagy r3-ra önmagában
- Csakhogy ha r1 illeszkedik, de r2 nem, akkor utána r3-mal próbálkozik, de a **rossz pozícióról**, hiszen azt r1 léptette
- Erre a lehetőségre a keretrendszert nem használó, „kézzel írt” elemzőkben mindig figyelni kell

Elemző keretrendszer C++11-ben

Miért érdemes használni?

- A „kézzel írt” elemzőkben a szabályokat megvalósító függvények **maguk dolgozzák fel a találatokat**
 - Így a nyelvtan és a feldolgozó műveletek összemosódnak
 - Hiába lenne egy nyelvtan vagy néhány szabály újrahasznosítható – ha **más-képp kell feldolgozni** a szöveget, **újra kell írni**
 - Egy keretrendszer **további szolgáltatásokat tud nyújtani**:
 - automatikusan elemző fát épít
-  Az elemző fa (abstract syntax tree – AST) olyan adatszerkezet, ami tükrözi az elemzett szöveg struktúráját, így könnyű feldolgozást biztosít.
- jól használható hibaüzeneteket ad

Miért érdemes használni? (folytatás)

- A szabályok leírása az ismétlődő kódrészletek helyett:
 - EBNF-hez hasonló alakban
 - deklaratív, tömör, áttekinthető
- Rengeteg **újrafelhasználható szabályt** tartalmaz, így **magas szinten** tudjuk megfogalmazni a nyelvtant – gyors, egyszerű fejlesztés
- Könnyen **bővíthető** – egyszerű kiegészíteni saját szabályokkal, amikkel **még** **tömörebbé** tehetjük a nyelvtanunkat

Miért érdemes használni? (folytatás)

- Gyakori megoldás manapság más nyelvekben a **deklaratív nyelvi elemkészlet** biztosítása:
 - a programozó a feladatra koncentrálni
 - a megoldás automatikusan előáll a probléma leírásából
 - könnyen olvasható, lehetővé teszi a belső optimalizációt
- Ilyen pl. a **C#-ban a LINQ (Language Integrated Query)**:

```
var query_where1 = from a in svcContext.AccountSet
    where a.Name.Contains("Contoso")
    select a;
```

- Vagy **Java 8-tól a streams**:

```
myList
    .stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

Egy példa

- Az alábbi példa egy hexadecimális szám leírását mutatja be:

```
rule hex_num, hex_prefix, hex_digit;  
  
hex_num <=> hex_prefix << +hex_digit;  
hex_prefix <=> character("0") << character("x");  
hex_digit <=> character("0123456789abcdef");
```

- Látható, hogy a **C++ nyelv operátorait használjuk** – nem pontosan egyeznek az EBNF-ével, de könnyen érthetőek
- Az elemzési **algoritmus teljesen rejtve marad**:
 - ez egy deklaratív leírás
 - belül egy olyan adatszerkezet épül, ami azt az elemző algoritmust valósítja meg, amit kézzel megírnánk

Egy példa (folytatás)

- Az értékadás operátora: `<=<`
- Az egymásrakövetkezésé: `<<`



EBNF-ben nincs ilyen operátor, egyszerűen egymás után írjuk az elemeket. Ez C++-ban nem működne, muszáj valamilyen operátort közéjük tenni.

- Az ismétlés operátora: `+`



Ez az EBNF-ben postfix operátor, azonban a C++-ban csak a `++` és a `--` tudnak postfixek lenni, ezért itt a prefix operátorokat használjuk: `+`, `-`, ``, `!`*

- Az alternatíva operátora: `|`

A `base_rule` osztály

- Az elemző osztályhierarchia alapja a `base_rule` osztály
- Feladatai/metódusai:
 - `match`: az elemzéssel kapcsolatos általános adminisztratív teendők elvégzése és az egyedi, illeszkedést vizsgáló függvény hívása
 - `clone`: a szabályosztályoknak klónozhatóaknak kell lenniük (erről később)
 - `operator []`: szemantikai eseménykezelő megadása



A szemantikai eseménykezelő olyankor fut le, amikor egy szabály illeszkedik. Ez a legegyszerűbb módja annak, hogy egy nyelvtan elemzését elvégezzük. A hátránya azonban az, hogy összetett szabályok esetén egy részlet illeszkedésekor még nem lehetünk biztosak abban, hogy az egész is fog. Ezért komolyabb nyelvtanok esetén nem alkalmazható – itt az egyszerű megvalósíthatósága miatt mutatjuk. Az igazi megoldás az AST építés.

A `base_rule` osztály (folytatás)

- A **szemantikai eseménykezelő** megvalósításához az `std::function` osztály-sablont használjuk.
- Így megadható: globális függvény, funktor, metódus, lambda.
- Az illeszkedő szövegrészletet a könnyű feldolgozhatóság érdekében `std::string`-ben kapja meg.

```
using semantic_action =  
    std::function<  
        void(std::string const &  
        >;
```


A `base_rule` osztály (folytatás)

1 test:

- ez teszi egyedivé az osztályt – a konkrét elemzési algoritmust tartalmazza
- privát elérésű, mert csak a `match` függvénynek kell hívnia

2 match:

- az elemzéssel kapcsolatos szabályfüggetlen, általános dolgok
- ő hívja a virtuális és privát `test` függvényt
- a `match` nem virtuális, hiszen ő minden szabályra egységes, a közös műveleteket tömöríti
- a `match` és a `test` a **nem-virtuális interfész** mintát valósítják meg

A *base_rule* osztály (folytatás)

A *match* függvény:

```
bool base_rule::match(match_range &context, match_range &
    matching_range) {
    match_range local = context, result; // (1)

    if (test(local, result)) { // (2)
        context = local; // (3)
        matching_range = result;

        if (the_semantic_action) { // (4)
            the_semantic_action(
                std::string(result.first, result.second)
            );
        }
        return true;
    }
    return false; // (5)
}
```


A `base_rule` osztály (folytatás)

A `match` függvény:

- 1 Lokális másolatot készít a kontextusról
- 2 Meghívja a virtuális `test` metódust az elemzés elvégzésére
- 3 Siker esetén frissíti a globális kontextust...
- 4 ...és meghívja a szemantikai eseménykezelőt (ha létezik az adott szabályhoz)
- 5 Sikertelenség esetén egyszerűen HAMIS értékkel tér vissza és nem változtatja a globális kontextust.

Ezek a lépések nagyon hasonlítanak a „kézi” módszernél megismertekhez.

A character szabály

- Nézzük meg egy szabály megvalósítását!
- A character egy karakterre való illesztést végez

```
class character : public base_rule {  
    private:  
        std::string values;  
  
        virtual bool test(match_range &context, match_range &  
            matching_range) override;  
  
    public:  
        character(std::string const &values) : values(values) {}  
  
        virtual std::shared_ptr<base_rule> clone() const override;  
};
```

- A character függvényhez hasonlóan egy std::stringben veszi át az elfogadott karaktereket – de itt a konstruktorban

A character szabály (folytatás)

- A test végzi az elemzési feladatot
- A kód nagyon hasonló a character függvényhez

```
virtual bool character::test(match_range &context, match_range &
    matching_range) override {
    if (context.first == context.second) return false; // (1)

    for (auto c : values) {
        if (*context.first == c) { // (2)
            matching_range = std::make_pair(
                context.first, context.first + 1
            );
            ++context.first;
            return true;
        }
    }
    return false; // (3)
}
```

A character szabály (folytatás)

A nagyon egyszerű kód miatt itt nem készítünk lokális másolatot a kontextusról.

- 1 Mivel terminális szimbólumot vizsgálunk, figyelni kell a szöveghatárokról
- 2 A szokásos feladatok sikeres illeszkedés esetén:
 - A találat elmentése egy `match_range`-ben
 - A kontextus léptetése
 - Visszatérés IGAZ-zal
- 3 Sikertelen illesztés esetén visszatérés HAMIS-sal

Egy operátor megvalósítása

- Az operátor túlterhelés önmagában nem elég, hiszen nem a hívás pillanatában végezzük az elemzést
- Az operátor egy olyan adatszerkezetet hoz létre, ami:
 - reprezentálja a nyelvtan szerkezetét
 - később képes végrehajtani a az elemzési feladatot
- Az operátorok is a `base_rule` leszármazottai, hiszen ők is egy **illesztési feladatot** végeznek el

Egy operátor megvalósítása (folytatás)

1. A konstruktorában **átveszi azt a szabályt, amire az operátor vonatkozik**
 - A test metódusa pedig megvalósítja az **operátor által előírt logikát**
 - vezérlési szerkezetek és
 - az eltárolt szabály hívása segítségével

Egy operátor megvalósítása (folytatás)

- 1 Lokális kontextus létrehozása
- 2 A legalább egyszeri előfordulás megvalósítása:
 - (a) ha egyszer illeszkedik
 - (b) próbáljuk meg utána annyiszor, amennyiszor csak lehet
- 3 A szokásos műveletek sikeres találat esetén
- 4 Sikertelen illesztés esetén visszatérés HAMIS értékkel

Egy operátor megvalósítása (folytatás)

- Maga a **felüldefiniált operátor** a következőképp néz ki:

```
repetition operator +(base_rule const &a_rule) {  
    return repetition(a_rule.clone());  
}
```

- Eddig nem beszéltünk arról, hogy miért kell:
 - clone függvény
 - miért okos mutatókat vesz át a konstruktor

A keretrendszer memóriamodellje

- Nézzünk meg egy tipikus kódrészletet:

```
a_rule <=& +character("abc");
```

- Itt két ideiglenes változó keletkezik:
 - egy character
 - egy repetition
- A repetition példánynak **el kell tárolnia** a character-t, az a_rule-nak pedig a repetitiont
- Ráadásul **referenciaként** kell átvenniük őket a **többalakúság** miatt

A keretrendszer memóriamodellje (folytatás)

- Az ideiglenes változók a legközelebbi **kiértékelési pont** után megszűnnek
- Ezért mindenképp **le kell őket másolni**
- Csakhogy mivel referenciaként vesszük át őket → **nem ismerjük a tényleges típusukat**
- Erre megoldás a `clone` függvény, amit szoktak **virtuális másolókonstruktor**nak is hívni:

```
virtual std::shared_ptr<base_rule> clone() const override {  
    return std::shared_ptr<base_rule>(new repetition(*this));  
}
```

- Minden osztály felüldefiniálja és **ősz típusú pointerként ad vissza egy másolatot saját magáról**

A keretrendszer memóriamodellje (folytatás)

- Van egy további problémánk is:

```
rule addition, addend, expression;  
  
addition <=< addend << *( character("+") << addend );  
addend <=< range('0','9') | expression;  
expression <=< character("(") << addition << character(")");
```

- A szabályok **körkörösén hivatkoznak egymásra** → nincs olyan sorrend, amelyben **ne szerepelne valamelyikben egy még definiálatlan szabály-ra való hivatkozás**
- Ha a **szabályok referenciáit tudnánk eltárolni**, akkor ezzel nem lenne baj
- Csakhogy láttuk, hogy másolnunk kell, az ideiglenesen létrejövő beépített szabályok miatt

A keretrendszer memóriamodellje (folytatás)

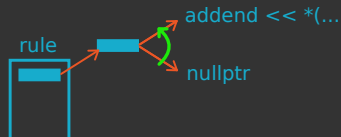
- Amikor (1)-ben az addendet lemásoljuk, egy **üres szabályt** másolunk le:

```
addition <=< addend << *( character( "+" ) << addend ); // (1)
addend <=< ...
```

- Olyan típus kéne, ami
 - nem változik meg attól, hogy értéket kap
 - azaz üres állapotában ekvivalens a tartalmat nyert állapotával

- A **megoldás egy plusz indirekció** bevezetése:

- a szabály tartalmazzon egy pointert, ami annak a mutatónak a címét tárolja, ami definíciójára mutat



1. ábra. A dupla indirekció

További eszközök a keretrendszerben

- Operátorok:

- *lhs* | *rhs*: VAGY-kapcsolat
- *!op*: opcionális
- *+op*: legalább egyszeri jelenlét
- **op*: tetszőleges számú jelenlét (0 is)
- *-op*: *op* előtti üres karakterek elnyelése
- *~op*: *op* előtti üres karakterek elnyelése (kivéve az újsor jelet)

- Szabályok:

- *in_range*: karakter intervallum (pl: $\text{in_range}('o', '9') \equiv ['o', '9']$)
- *integer*, *real*: egész illetve valós számok
- *string*: szövegrészlet valamilyen szeparáló karakterek között
- *keyword*: programnyelv kulcsszava
- *identifier*: programnyelv azonosítója

AST készítés

Az elemzés eredménye

- Az eddigi példákban a végeredmény egy logikai érték volt:
 - **Igaz:** az elemzett szöveg megfelelt a nyelvtannak
 - **Hamis:** szintaktikai hiba volt a szövegben
- Valójában mindkét esetben bővebb információra van szükség:
 - helyes esetben kell egy adatszerkezet, amiből kényelmesen kinyerhetőek az adatok – ez lesz a szintaxis fa (AST)
 - hibás esetben fontos:
 - pontosan hol történt a hiba
 - mely szabály elemzésekor történt – vagyis mi az, amit vártunk

A szintaxis fa

- Minden nyelvtan belső szerkezete is fa:
 - minden egység (kezdve a mondattal) kisebb, át nem fedő egységekre osztható
 - amelyek további kisebb részekre oszthatóak
 - Például a mondatok tagmondatokra (mellé-, vagy alárendelő tagmondatokra)
 - A tagmondatok jelzős, határozós szerkezetekre, stb.
 - Ezek a szerkezetek pedig jelzőkre, határozókra, állítmányra, alanyra, stb.
- Ha ez a szerkezet tulajdonképpen már jelen van, akkor nem nagy energiával ténylegesen elő is állítható az elemzés során

Időpont példa szintaxis fákhhoz

- Tekintsük az alábbi nyelvtant:

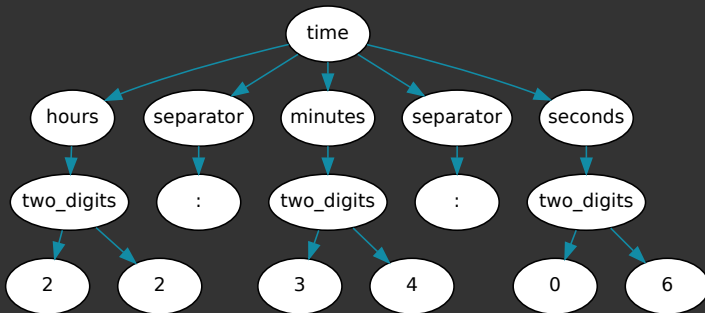
```
time <= hours < separator < minutes < separator < seconds;  
  
hours      <= two_digits;  
minutes    <= two_digits;  
seconds    <= two_digits;  
  
two_digits <= in_range('0','9') < in_range('0','9');  
separator  <= character(":");
```

- A fenti nyelvtan leírja például az alábbi időpontot:

22:34:06

Időpont példa szintaxis fákhhoz (folytatás)

- A fenti példához (22:34:06) tartozó absztrakt szintaxisfa:



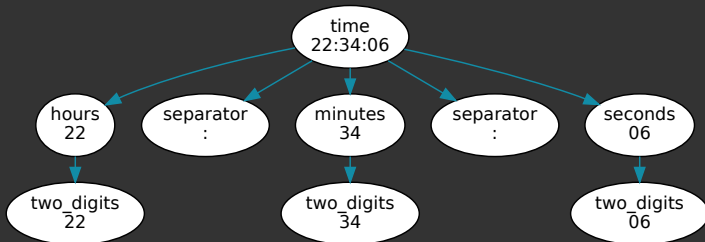
2. ábra. Az elméleti AST

Időpont példa szintaxis fákhhoz (folytatás)

- A fa szerkezete a következő:
 - a gyökere a főszabály
 - minden szabály gyermekei az általa tartalmazott szabályok
 - a levelek a terminális szimbólumok (itt: számjegyek és kettőspontok)
- Ez egy logikus és áttekinthető szerkezet
- Előnye, hogy a bejáró algoritmusnak már nem kell hibakezeléssel foglalkoznia – ha elkészült, akkor nem volt szintaktikai hiba
- A gyakorlati elemzéshez kicsit kényelmetlen:
 - le kell menni a legalsóbb szintre, hogy a konkrét beolvasott szimbólumot elérjük
 - olyan szabályokon is át kell lépnünk (például *two_digits*), ami nem érdekes a számunkra elemzéskor, csak a nyelvtan írását könnyítette

Időpont példa szintaxis fákhhoz (folytatás)

- Nézzük meg az alábbi szintaxisfát:



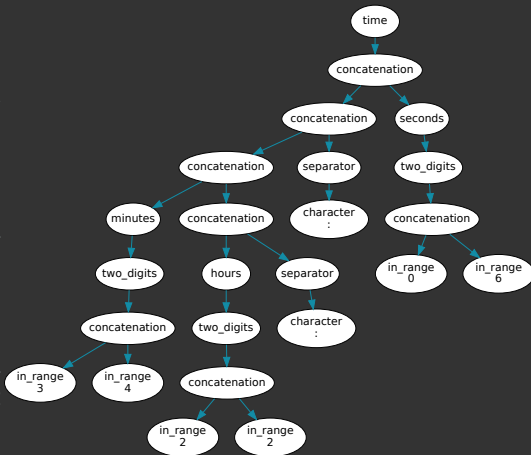
3. ábra. Egy jobban kezelhető AST

Időpont példa szintaxis fákhhoz (folytatás)

- Ez a fa is minden szabályt tartalmazza
- A terminális szimbólumokat viszont már nem
- A szabályok neve mellett minden csomópont tartalmazza a szabályhoz megadott szövegrészletet
- Így ha szeretnénk bejárni az egész fát, megtehetjük
- Bárhol megállhatunk a mélységi bejárás során, ahol a rendelkezésreálló információ már a kellő felbontásban van jelen
- Például, ha a percekre vagyunk kíváncsiak, akkor a *minutes* csomópont már megadja nekünk a "34" stringet, amit számmá alakíthatunk és tudunk vele dolgozni

A nyelvtan fája

- Maga a nyelvtan is egy fa, amit az elemzés bejár
- Ugyanakkor ez még az elméleti AST-nél is több elemből áll – a használata nagyon kényelmetlen lenne
- Mivel ezen dolgozunk, ezért ebből kell kialakítanunk az ideális AST-t



4. ábra. A nyelvtan fája

A fa reprezentációja a kódban

- A fa egy csomópontját a következő adatszerkezet írja le:

```
struct node {  
    char const *name;  
    match_range matching_range;  
  
    std::vector<std::shared_ptr<node>> children;  
};
```

- A csomópont neve \equiv a megfelelő szabály neve
- Az adott szabály által lefedett szövegrészlet



Csak az elejére és a végére mutató pointerek, különben a fa többszörösen tartalmazná a vizsgált szöveget.

- A csomópont gyermekei


A fa építése

- Triviális faépítő algoritmus:
 - A karakterszintű elemek elkészítik a leveleket és visszatérnek velük
 - A magasabb szintű elemek
 - átveszik a gyermekeiktől kapott részfat
 - létrehozzák a saját reprezentációjukat
 - hozzáfűzik a gyermekeiket a saját csomópontjukhoz
 - visszatérnek a kiegészített részfával
 - A főszabály visszaadja a teljes fat
- Ezzel az elméleti AST-t elő lehet állítani
- A számunkra kényelmes fat azonban nem

A fa építése (folytatás)

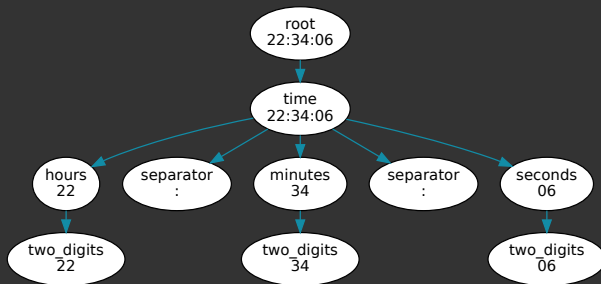
- Az előállítandó fában bizonyos elemek (pl. összefűzés operátor)
 - nem adnak hozzá elemet a fához
 - a gyermekeik adhatnak, akár többet is (pl. ha két szabályt fűz össze)
 - vagy akár több szinten is (pl. egy szabályt és egy összefűzést fűz össze, amely tartalmaz szabályt)
- Így nagyon bonyolult lenne az adminisztráció
- Helyette fordítunk egyet:
 - Minden elem kap egy csomópontot (a részfájának a gyökér elemét)
 - Vagy hozzáfűzi magát, vagy nem
 - Továbbadja a gyermekeinek, aki szintén vagy hozzáfűzik magukat, vagy nem

A fa építése (folytatás)

- A fenti algoritmussal bármennyi szint kimaradhat az elemek között
 - A szabály elemek (rule):
 - létrehoznak maguknak egy új elemet
 - továbbadják az új elemet a gyermekeik felé
 - sikeres illesztés esetén:
 - elmentik a csomópontba a feldolgozott szöveget
 - hozzáadják az új csomópontot a kapott fához
 - sikertelenség esetén nem tesznek semmit
-  *A csomópontokat dinamikusan, okos pointerrel hozzák létre; sikertelenség esetén ezek felszabadulnak*
- A többi elem: egyszerűen továbbadja a gyermekeinek a megkapott gyökeret, illetve a karakterszintű elemek még ennyit sem tesznek

A fa építése (folytatás)

- Az egyetlen furcsa következmény: kell egy gyökérelem, amit a legfelsőbb szintű szabály megkap – extra elem a fában, nincs igazi jelentése
- A keretrendszer fel van rá készítve és beleteszi a teljes lefedett szöveget, így ez is hasznosítható



5. ábra. A SyntX által előállított fa

A fa építése (folytatás)

- A match és test metódusok kiegészülnek egy újabb argumentummal:

```
class base_rule {  
    private:  
        virtual bool test(  
            match_range &context,  
            match_range &matching_range,  
            std::shared_ptr<node> &root // a fa gyökere  
        ) = 0;  
  
    public:  
        bool match(  
            match_range &context,  
            match_range &matching_range,  
            std::shared_ptr<node> &root // a fa gyökere  
        );  
        ...  
};
```


A fa építése (folytatás)

- A szabályok közül egyedül a rule az, aki csomópontot ad a fához:

```
bool rule::test(match_range &context, match_range &matching_range
    , std::shared_ptr<node> &root) {
    match_range local = context, the_match;
    std::shared_ptr<node> rule_node
        = std::make_shared<node>(name);    // (1)
    if (*the_rule == nullptr) throw "Inner_rule_not_set_in_a_rule";

    if ((*the_rule)->match(local, the_match, rule_node)) { // (2)
        matching_range.first = context.first;
        matching_range.second = the_match.second;
        context = local;

        rule_node->set_match_range(matching_range); // (3)
        root->children.push_back(rule_node);        // (4)
        return true;
    }
    return false;
}
```

A fa építése (folytatás)

- 1 A rule létrehoz egy új csomópontot, ami a szabály nevét kapja
- 2 A csomópontot továbbadja a belső szabályainak
- 3 A sikeres illesztés után a lefedett szöveget beállítja a csomópontban
- 4 Az új csomópontot hozzáadja az argumentumként kapott fához

A fa építése (folytatás)

- A köztes szabályok (pl. concatenation) egyszerűen továbbadják a root elemet
- A concatenationnél a logikai kiértékelési sorrend révén először a bal- (1), majd a jobboldali (2) operandus adhatja hozzá a fához a csomópontjait:

```
bool concatenation::test(  
    match_range &context, match_range &matching_range,  
    std::shared_ptr<node> &root  
) {  
    match_range local = context, lhs_match, rhs_match;  
  
    if (lhs->match(local, lhs_match, root)           // (1)  
        && rhs->match(local, rhs_match, root)) {    // (2)  
        ...  
    }
```

Hibaüzenetek generálása

Szintaktikai hibák kezelése

- A `match` és `test` függvények visszatérése logikai, megmondja, hogy sikeres volt-e az illeszkedés
- Gond:
 - egy szabály elbukása még nem feltétlenül jelent problémát:

```
title <= keyword("Mr") | keyword("Mrs");
```

- Egy logikai értéknél pontosabb információra van szükség:
 - hol van a szintaktikai hiba
 - milyen szabály bukott el (mit vártunk volna)

Szintaktikai hibák kezelése (folytatás)

- **Nehézség:**

- Ha bármelyik karakterszintű szabály elbukik, attól egy másik elemzési út még lehet helyes
- Tetszőleges számú karakterszintű szabály elbukhat egy elemzés során
- Csak a legfelső szabályról mondhatjuk biztosan, hogy ha elbukik, az hiba



ezen a ponton viszont már nincs meg az az információ, hogy mi okozta a hibát

Egy minta üzenet

- A hibaüzenet mondja meg:
 - pontosan melyik karakteren történt a hiba
 - melyik karakterszintű szabály bukott el
 - melyik `rule` bukott el (érthetőbbé teszi az üzenetet)
- Például: az időpontos nyelvtannak ha a következő bemenetet adjuk:

22:3_:06


akkor az üzenet legyen:

```
A character in the range: ['0', '9'] failed while trying to
  match a(n) two_digits:
22:3
    -
      :06
    ^
```

A hibahely megkeresése

- Igazából nincsen pontos algoritmus a hiba megkeresésére
 - az egyik szabály elbukása pontosan ugyanolyan, mint egy másiké
 - ♣ *Ha az ember ránéz egy szövegre, hamar nyilvánvaló lesz, hogy „mi van elírva”, de az algoritmus számára nem megkülönböztethető egyik hiba a másiktól.*
 - pl. ha három alternatíva közül egyik sem illeszkedik, akkor vajon melyiket akarhatta a szerző?
 - ♣ *Könnyű lenne úgy megírni a kódot, hogy az utolsónál írjon hiba-üzenetet, de egyáltalán nem biztos, hogy az lenne a legjobb tipp, ráadásul lehet, hogy valójában egy teljesen más elemzési ágban lesz az igazi találat.*

A hibahely megkeresése (folytatás)

- Heurisztikát lehet alkalmazni:
 - A legtovább jutott szabály valószínűleg a jó irányba ment.
 - A legtávolabbi hibahelyet érdemes visszaadni a lehető legtöbb hozzátartozó információval együtt.
 - A maximumkereséshez szükség van:
 - egy struktúrára, ami tartalmazza a hibahelyet
 - egy olyan példányra ebből a típusból, amit minden szabály elér
-  *így hiba esetén el tudják dönteni, hogy lehetséges jelölt-e az aktuális pozíció*

A hibahely megkeresése (folytatás)

- A típus:

```
using syntax_error = std::tuple<  
    std::string, std::string::const_iterator, std::string  
>;
```

- A tuple elemei:

- 1 az elbukott karakterszintű szabály leírása



Részletes leírás arról, hogy mit várt az elemző azon a pozíción.

- 2 a hiba pozíciója
- 3 a rule, amiben a hiba történt

A hibahely megkeresése (folytatás)

- A `syntax_error` példány a `match` és `test` metódusok egy újabb argumentuma lesz
- Így a teljes aláírás:

```
virtual bool test(  
    match_range &context, match_range &matching_range,  
    std::shared_ptr<node> &root, syntax_error &error  
) = 0;
```

- Minden szabály továbbadja a gyermekeinek, a karakterszintűek és a rule-ok írhatják

A hibahely megkeresése (folytatás)

- A karakterszintű szabályok, ha elbuknak:
 - Ellenőrzik, hogy az elemzési pozíció nagyobb-e, mint az aktuálisan az errorban lévő
 - Ha igen, akkor beleteszik a saját leírásukat és az aktuális elemzési pozíciót
- Ezt a logikát ki lehet emelni egy közös ősbe:

```
class character_level_rule {  
    private:  
        virtual std::string description() const = 0;  
    public:  
        void set_error_message(std::string::const_iterator  
            current_position, syntax_error &error) {  
            if (current_position > std::get<1>(error))  
                error=syntax_error(description(), current_position, "");  
        }  
        ...  
};
```

A hibahely megkeresése (folytatás)

- Így egy karakterszintű szabály test metódusa:

```
bool character::test(
    match_range &context, match_range &matching_range,
    std::shared_ptr<node> &root, syntax_error &error)
{
    if (context.first == context.second) return false;
    for (auto c : values) {
        if (*context.first == c) {
            matching_range = std::make_pair(context.first , context.
                first + 1);
            ++context.first;
            return true;
        }
    }
    set_error_message(context.first, error); /* (*) */
    return false;
}
```

A hibahely megkeresése (folytatás)

- Természetesen minden karakterszintű szabály így két őssel rendelkezik:

```
class character : public base_rule, public character_level_rule
```

- és mind elkészíti a saját leírását:

```
std::string character::description() const {  
    return format(  
        "A_{}_character_{}_from_{}_the_{}_character_{}_set:{}_\\\"%0\\\"\"", values  
    );  
}
```



A format egy felokosított és C++-osított printf.

A hibahely megkeresése (folytatás)

- A karakterszintű szabályok csak a leírásukat és a pozíciót írják bele a `syntax_errorba`
- Azt nem tudják, hogy milyen `rule-on` belül vannak
- Ahogyan a hiba miatt a rekurzió visszafejtődik, előbb-utóbb elérkezünk egy `rule` szabályhoz
- A `rule` onnan tudja, hogy bele kell tennie a nevét, hogy üres string a tuple harmadik eleme:

```
bool rule::test(...) {  
    ...  
    if ((*the_rule)->match(local, the_match, rule_node, error)) {  
        ...  
    }  
    else {  
        if (std::get<2>(error) == "") std::get<2>(error) = name;  
    }  
}
```

A hibahely megkeresése (folytatás)

- Magát az üzenetet egy globális függvény állítja elő az elemzési kontextus (context) és a syntax_error alapján

```
std::string error_message(  
    match_range const &context,  
    syntax_error const &error  
);
```

- Ő gondoskodik a hibás sor megkereséről és a sor három részre töréséről:
 - hiba előtti szövegrészlet
 - a hibás karakter
 - a hiba után rész

Generátor kifejezések

Az eredmények feldolgozása

- A keretrendszert használó programozó számára a feladat:
 - a nyelvtan helyes megírása
 - az előállított adatszerkezet feldolgozása
- Az utóbbi jelen pillanatban az AST bejárását jelenti:
 - Rekurzív, fabejáró algoritmust kell írni
 - Ha változtatni kell a nyelvtanon (\rightarrow változik az AST), jelentősen bele kell nyúlni a feldolgozó algoritmusba is
 - Keveredik a fabejárás és az adatfeldolgozás

Generátor kifejezések

- A keretrendszer biztosít egy deklaratív leírási módot, amivel bejárhatjuk a fát és leírhatjuk az elvégzendő feladatokat
- Nincs szükség rekurzióra:
 - minden nyelvtani szabályhoz létezik egy generátor szabály
 - a generátorokra ugyanolyan operátorok alkalmazhatóak, mint a nyelvtani szabályokra
 - amikor egy szabály egy másikra hivatkozik (és ezzel egy új részfat ad az AST-hoz), ott egy generátor is hivatkozik egy másikra
- A rendszer tartalmaz egy sor tipikus feladatra felkészített generátort
- Létezik olyan generátor, ami egy lambdát/funktort vesz át, ezzel minden feladat megoldhatóvá válik

Az AST bejárása

- A fa bejárását egy iterátor segíti, amit a node osztály biztosít:

```
using node_iterator =  
    std::vector<std::shared_ptr<node>>::const_iterator;
```

- A `node::cbegin` és `node::cend` metódusok a csomópont gyermekein való végigiterálást biztosítják:

```
node_iterator cbegin() const {  
    return children.cbegin();  
}
```

Az AST bejárása (folytatás)

- Az összes generátor megkapja az aktuális csomópontra mutató iterátort:
 - az iterátortól elkérhetik a csomópont adattagjait
 - a csomópont nevét (ami a hozzá tartozó szabály neve is)
 - a lefedett szövegre mutató iterátorokat
 - sikeres generálás esetén a generátorok növelik az iterátort és igaz értékkel térnek vissza
 - sikertelenség esetén nem állítják el az iterátort és hamis értékkel térnek vissza
- A generátorok nagyon hasonlóan működnek, mint az elemző szabályok

Az AST bejárása (folytatás)

- Minden generátornak meg kell adni, hogy mi annak a szabálynak a neve, amit generál
- A generátorok fontos feladata, hogy leellenőrizzék, hogy a megfelelő szabályra mutató iterátort kapták-e
 - Ha nem, akkor az azt jelentheti, hogy AST szerkezete nem felel meg az elvártnak
 - Ilyenkor a generátor hamis értékkel tér vissza
 - Az AST-generátor eszközkészlet nyelvén ez két dolgot jelenthet:
 - a szöveg szemantikailag hibás
 - egy másik elemzési út fog a helyes bejáráshoz vezetni
- Ahol a nyelvatanban a VAGY-operátor szerepel, vagyis több alternatíva képzelhető el, ott a generátorok is VAGY-kapcsolatban kell, hogy álljanak
 - A generátorok a nevek leellenőrzésével tudják eldönteni, hogy a megfelelő alternatívát generálják-e éppen

Az AST bejárása (folytatás)

- Az is előfordulhat, hogy egy csomóponttal kapcsolatban semmiféle generálási feladat nincsen
- A generálás során minden csomópontot érinteni kell, az ilyeneket is
- Létezik „semmit sem csináló” generátor, azonban ennek is van feladata:
 - le kell ellenőriznie, hogy a megfelelő nevű csomóponton áll-e
 - növelnie kell az iterátort
- Az ilyen csomópontok felhasználhatóak arra, hogy könnyen szétválasszunk alternatívákat
- Néha azonban egyszerűen csak átugorjuk őket – például az időpontos példában a separator szabály ilyen

Időpont generálás

- Egy egyszerű példa az időpontos nyelvtanhoz:

```
gtime
  <=> ghours << skip("separator") << gminutes
  << skip("separator") << gseconds;

ghours
  <=> print_value("two_digits") << print_text("□hours□");

gminutes
  <=> print_value("two_digits") << print_text("□minutes□");

gseconds
  <=> print_value("two_digits") << print_text("□seconds.\n");
```

- Az eredménye:

```
22 hours 34 minutes 06 seconds.
```


Példa egy VAGY-kapcsolatra

```
gnumber <=>
perform("hexa_num", /* Hexadecimális szám */
[&the_numbers, &a_number, &ss](generator_range &context) ->
    bool {
        match_range match = (**context.first).get_match_range();
        ss.clear();
        ss << std::hex << std::string(match.first, match.second);
        ss >> a_number;
        the_numbers.push_back(a_number);
        ++context.first; return true;
}) | perform("oct_num", /* Vagy oktális szám */
[&the_numbers, &a_number, &ss](generator_range &context) ->
    bool {
        match_range match = (**context.first).get_match_range();
        ss.clear();
        ss << std::oct << std::string(match.first, match.second);
        ss >> a_number;
        the_numbers.push_back(a_number);
        ++context.first; return true;
});
```

Példa egy VAGY-kapcsolatra (folytatás)

- Az előző példa bemutatja:
 - hogyan lehet a keretrendszerrel tetszőleges generátort „helyben” elkészíteni a `perform` általános generátor és egy lambda segítségével
 - hogyan lehet a VAGY-operátorral két alternatíva közül választani
 - hogy néha érdemes a részeredményeket vagy közösen használt erőforrásokat mindenki számára elérhető helyen tárolni és a generátoroknak átadni
- A `gnumber` generátorhoz tartozó nyelvtani szabály alakja:

```
number <=<= oct_num | hexa_num;
```

- Mind a nyelvtan, mind a generátor egy számlistát képes feldolgozni:

```
list_of_numbers <=<= number << +(character(",") << number);  
glist_of_numbers<=<+=gnumber; /*A character nem kerül az AST-be*/
```

A teljes keretrendszer

Syntx: a teljes, letölthető keretrendszer

Az első C++11 nyelvű implementáció:

<https://gitlab.com/nagygr/syntx>

Python nyelvű implementáció:

<https://gitlab.com/nagygr/syntxpy>

Az új, generátorokat is tartalmazó implementáció:

<https://gitlab.com/nagygr/syntxii>