BI-PJP

6. března 2024

Obsah

1 Jaká je funkce lexikálního analyzátoru? Jak je lexikální analyzátor typicky implementován?

(Lexan, Lexer, nebo i Scanner)

- Lexan vezme kód (např. c++, scala, brainfuck...) a vrátí posloupnost tokenů.
  - Tyto tokeny jsou terminální symboly bezkontextové gramatiky, která popisuje syntaxi vstupního jazyka (právě toho c++, scala brainfuck...).
- Jazyk lexikálních tokenů lze typicky popsat regulární gramatikou, lze tedy implementovat jako konečný automat (jinak je tomu u jazyků jako Python a Haskell, kde porovnáváme indentaci řádků).
- Lexikální analyzátor implementuje lexikální gramatiku daného jazyka.
- Rozpoznává tokeny (lexikální elementy):
  1. `identifikátory` → např. `písmeno (písmeno or číslice)*`
  2. `klíčová slova` → např. `'if' or 'then' or 'else'`
  3. `literály (čísla, řetězce)` → např. `číslice {číslice}*`
  4. `speciální symboly` → např. `'+' or '-' or '<' or '<=' or ':='`
  - Některé tokeny potřebují k sobě více info, např. číslo si chce pamatovat svou hodnotu, ale klíčové slovo nic dalšího nemusí potřebovat (záleží na implementaci).
- Lexan přeskakuje whitespaces a komentáře.
  - To co je white space se může v různých jazycích měnit, v c++ např. tab je whitespace ale v pythonu není.
- Lexan rozpoznává a reaguje na direktivy překladače.
  - `#include, #iff ...`
  - Direktivy jsou součástí metajazyka a řeší se v lexeru nebo v presprocesorové části lexeru. Doplní se soubory, nastaví konstanty.
- Chyby co Lexan detekuje:
  1. neznámý znak,
  2. neukončený řetězec do konce řádku,
  3. chyba v komentáři.
- Implementační tipy na lexan:
  - načtení do bufferu se systémem dvou pointerů,
  - pomocí gramatiky a tabulky můžeme to implementovat.

2 Jak je definována funkce FIRST? Jak se funkce FIRST vypočítá? K čemu se funkce FIRST používá při konstrukci LL analyzátoru?

- Používá se ke konstrukci rozkladové tabulky při LL(1) analýze.
- Funkce FIRST spolu s tabulkou nám pomáhají rozhodnout, které pravidlo máme použít, při daném symbolu terminální abecedy na vstupu.
- Funkce FIRST je definována pro libovolný *řetězec* z terminálů a neterminálů
- FIRST(α) je množina všech terminálů (případně i ε), kterými mohou začínat řetězce generovatelné z α
- Nechť máme gramatiku $G = (N, T, P, S)$.

$$\text{FIRST}(\alpha) = \{k : \alpha \models^* k\beta, k \in T, \alpha, \beta \in (N \cup T)^*\} \cup \{\varepsilon : \alpha \models^* \varepsilon\}$$

- Pravidla pro výpočet FIRST:
  - FIRST(ε) = $\{\varepsilon\}$
  - FIRST(a) = $\{a\}$, když $a \in T$
  - FIRST(A) = first(A), když $A \in N$
  - FIRST($A\alpha$) = first(A), když $A \in N$ a $\varepsilon \notin$ first(A)
  - FIRST($A\alpha$) = (first(A) $- \{\varepsilon\}$) $\cup$ FIRST(α), když $A \in N, \varepsilon \in$ first(A)

  Algoritmus výpočtu first(A) pro všechna $A \in N$:

  1. pro všechna $A \in N$ first(A) = $\emptyset$
  2. pro všechna pravidla $A \to \alpha$:

     first(A) = first(A) $\cup$ FIRST(α)
  3. opakovat krok 2, pokud se alespoň jedna množina first(A) změnila.

3 Jak je definována funkce FOLLOW? Jak se funkce FOLLOW vypočítá? K čemu se funkce FOLLOW používá při konstrukci LL analyzátoru?

- Slouží ke konstrukci rozkladové tabulky při LL(1) analýze (a tím pádem i k detekci konflitků).
- Funkce FOLLOW podobně jako funkce FIRST napomáhá na základě jednoho symbolu na vstupu rozhodnout, které pravidlo použít.
- Výpočet funkce FOLLOW je relevantní pouze pokud existuje nějaké pravidlo s ε na pravé straně pravidla.
- Nechť máme gramatiku $G = (N, T, P, S)$. Funkce FOLLOW je definována pro libovolný *neterminál* $A \in N$.
- FOLLOW(A) je množina všech terminálů (případně i ε ve významu EOF), které se mohou vyskytovat ve větných formách bezprostředně za symbolem A.
- FOLLOW(A) = $\{k : S \models^* \alpha A \beta,\ k \in \text{FIRST}(\beta)\}$.
- Algoritmus pro výpočet FOLLOW(A) pro všechna $A \in N$:
  1. FOLLOW(S) = $\{\varepsilon\}$

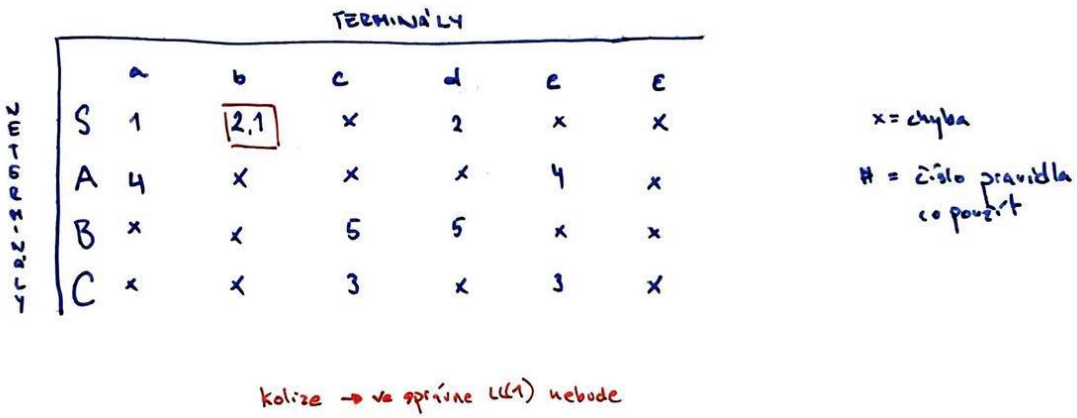
     FOLLOW(A) = $\emptyset$ pro všechny $A \in N, A \neq S$,
  2. pro všechna pravidla:

     má-li pravidlo tvar $X \to \alpha Y \beta$, pak
     - FOLLOW(Y) = FOLLOW(Y) $\cup$ (FIRST(β) $- \{\varepsilon\}$)

     má-li pravidlo tvar $X \to \alpha Y \beta$ a $\varepsilon \in$ FIRST(β), pak
     - FOLLOW(Y) = FOLLOW(Y) $\cup$ FOLLOW(X)
  3. opakovat krok 2, pokud se alespoň jedna množina FOLLOW(A) změnila.

4 K čemu slouží rozkladová tabulka LL analyzátoru? Jak se vytvoří?

- Rozkladová tabulka pro LL(1) gramatiku $G = (N, T, P, S)$ je zobrazení $\mathrm{M}(A, a)$, $A \in N$, $a \in T \cup \{\varepsilon\}$, jehož hodnotou je číslo pravidla, které se má použít při expanzi neterminálu A a dopředu přečteném symbolu, nebo značka chyby.
- Jinak řečeno tabulka udává, které pravidlo má být použité (pro určitý neterminál a terminál na nepřečteném vstupu), při levém rozkladu. Proto chceme, aby každé políčko mělo jen jednu hodnotu, jinak máme nedeterminismus.
- ε v tabulce značí EOF.
- Vytvoření rozkladové tabulky:
  - je-li $A \to \alpha$ i-té pravidlo v P, pak $i \in \mathrm{M}(A, a)$ pro všechna $a \in \mathrm{FIRST}(\alpha)$ bez $\{\varepsilon\}$,
  - je-li $A \to \alpha$ i-té pravidlo v P a $\varepsilon \in \mathrm{FIRST}(\alpha)$, pak $i \in \mathrm{M}(A, a)$ pro všechna $a \in \mathrm{FOLLOW}(A)$,
  - $\mathrm{M}(A, a) =$ *chyba* ve všech ostatních případech.



5 Co znamená, že je gramatika LL(1)? Jak můžeme zjistit, že je gramatika LL(1)?

- Hlavní myšlenka LL(1) gramatiky spočívá v tom, že na základě jednoho terminálního symbolu na vstupu a neterminálu na vrcholu zásobníku existuje pouze jedno možné pravidlo pro expanzi.
- Existují dva způsoby jak definovat tuto gramatiku:
  1. $G = (N, T, P, S)$ je LL(1) gramatika, když pro každou dvojici pravidel $A \to \alpha|\beta$ platí:
     - $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$
     - pokud $\varepsilon \in \text{FIRST}(\alpha)$, potom $\text{FOLLOW}(A) \cap \text{FIRST}(\beta) = \emptyset$
  2. $G = (N, T, P, S)$ je LL(1) gramatika, když pro každé dvě levé derivace platí:
     - $S \models^* wA\alpha \models w\beta\alpha \models^* wx$
     - $S \models^* wA\alpha \models w\gamma\alpha \models^* wy$

     kde $\text{FIRST}(x) = \text{FIRST}(y)$, platí, že $\beta = \gamma$
- Jeden ze způsobů jak rozpoznat LL(1) gramatiku je, že v rozkladové tabulce nejsou přítomny kolize (konflikty).
- LL(1) gramatika nikdy neobsahuje levou rekurzi.

6 Co znamená, že je jazyk LL? Jaký je vztah třídy LL jazyků ke třídě deterministických bezkontextových jazyků?

- LL gramatika patří mezi bezkontextové.
- Vlastnosti LL gramatiky:
  - Každá LL(k) gramatika je jednoznačná.
  - Žádná LL(k) gramatika není levě rekurzivní.
  - Pro danou bezkontextovou gramatiku G a dané pevné $k \geq 0$ je rozhodnutelné, zda G je nebo není LL(k).
  - Pro danou bezkontextovou gramatiku G je nerozhodnutelné, zda je LL(k) gramatikou pro nějaké $k \geq 0$.
  - Je-li dána bezkontextová gramatika G, která není LL(k) a pevné k, je nerozhodnutelné, zda G má ekvivalentní gramatiku, která je LL(k).

    Musí se manuálně vyzkoušet odstranit všechny konflikty a levé rekurze, poté zjistit zda to odstraňování nevytvořilo jiné a opět zkusit odstranit ty. Nevíme, zda se dostaneme do stavu, že je gramatika LL(k).
- Bezkontextový jazyk L se nazývá LL(k) jazyk, jestliže existuje LL(k) gramatika G, $k \geq 0$, taková, že $L = L(G)$
- Bezkontextový jazyk L se nazývá LL jazyk, jestliže existuje LL(k) gramatika G pro nějaké $k \geq 0$ taková, že $L = L(G)$
- Čím větší k, tím větší množinu jazyků tím jde popsat

$$\text{LL}(0) \subsetneq \text{LL}(1) \subsetneq \text{LL}(2) \subsetneq \cdots \subsetneq \text{LL}(n).$$

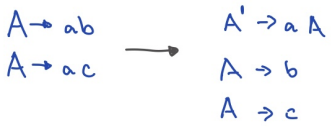
- Pro každou LL(k) gramatika lze udělat i LR(k) gramatiku. Naopak tomu tak není.
- Ne každý deterministický bezkontextový jazyk je LL (např., $a^m b^n : m \geq n \geq 0$) (ale každý bezkontextový jazyk je LR).

7 Jaká je funkce LL(1) analyzátoru? Vysvětlete princip implementace LL(1) analyzátoru metodou rekurzivních funkcí.

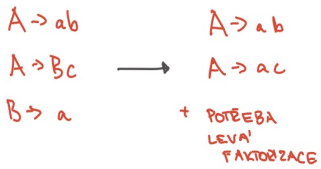
- LL(1) analyzátor je užitečný pro kontrolu syntaxe vstupního kódu a v rámci něho můžeme vytvořit další reprezentaci kódu, která zachycuje jeho význam.
- LL(1) analyzátor popisuje LL(1) gramatiku, která popisuje syntaxi programovacího jazyka.
- LL(1) analyzátor je deterministický a kouká na jeden prvek před sebe.
- Metoda rekurzivních funkcí popisuje způsob implementace parseru.
- Metoda rekurzivních funkcí:
  - pro každý neterminál existuje jedna funkce,
  - tělo dané procedury se větví dle pravidel pro daný neterminál, typicky se k tomu používá `switch, if-else` (control flow řídí, která pravidla se mají použít podobně jako to udává rozkladová tabulka),
  - máme funkci pro operaci srovnání pro terminály.

8 Co znamená konflikt FIRST–FIRST? Jak je možné odstranit tento konflikt transformací gramatiky?

- Konflikt znamená, že na základě přečtení jednoho následujícího symbolu na vstupu se nejsme schopni rozhodnout, které pravidlo gramatiky použít.
- FIRST-FIRST konflikt nastává pokud máme neterminál A a k němu pravidla $A \to \alpha$ a $A \to \beta$, a α a β mohou začínat na stejný terminál, tedy FIRST(α) a FIRST(β) mají neprázdný průnik.
- Přímá FIRST-FIRST kolize se dá odstranit *levou faktorizací*.
- Levá faktorizace:
  - Přidáme nový neterminál, který "vytkne"terminál a poté původní pravidla přijdou o svůj první terminál.



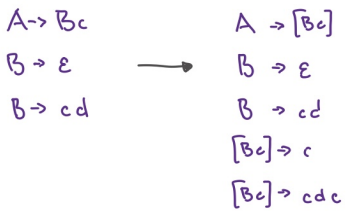
- Nepřímá FIRST-FIRST kolize se dá odstranit *rohovou substitucí*.
- Rohová substituce:
  - Doplníme za neterminál a poté se vypořádáme s případně nově vzniklými problémy.



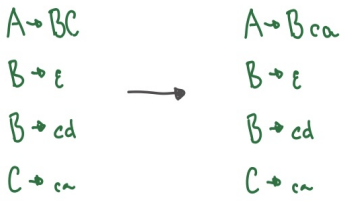
- Při obou postupech odstranění kolize může vzniknout jiný typ kolize, nebo levá rekurze, předem nevíme, jestli odstraněním problému bude gramatika v pořádku.

9 Co znamená konflikt FIRST–FOLLOW? Jak je možné odstranit tento konflikt transformací gramatiky?

- Konflikt znamená, že na základě přečtení jednoho následujícího symbolu na vstupu se nejsme schopni rozhodnout, které pravidlo gramatiky použít.
- Vzniká pouze pokud existuje ε-pravidlo, které dělá problémy.
- FIRST–FOLLOW konflikt nastává, když pro nějaký neterminál A existují pravidla $A \to \varepsilon$ a $A \to \alpha$, které způsobují, že existuje symbol, který je v FOLLOW(A) a zároveň v FIRST(α).
- Přímá FIRST–FOLLOW kolize se dá odstranit *pohlcením terminálu.*
- Pohlcení terminálu:
  - Udělat nový neterminál, který nás provede možnostma co se z toho může stát.



- Nepřímá FIRST–FOLLOW kolize se dá odstranit *abstrakcí pravého kontextu*
- Extrakce pravého kontextu:
  - Doplníme za druhý neterminál.



- Při obou postupech odstranění kolize může vzniknout jiný typ kolize, nebo levá rekurze, předem nevíme, jestli odstraněním problému bude gramatika v pořádku.

10 Jak je definována atributová gramatika? K čemu se používá? Jaký je rozdíl mezi dědičnými a syntetizovanými atributy?

- Co je atributová gramatika:
  - Je to rozšíření bezkontextové (překladové) gramatiky o atributy ("proměnné").
  - Jde o způsob jak popsat sémantiku, ne pouze syntaxi.
  - Atributy mohou nabývat hodnotu z oboru hodnot funkce definované pravidlem.
  - Je to Turing complete formalismus podobně jako lambda calculus.
  - Každý atribut je přiřazený syntaktickému symbolu gramatiky (neterminálu, či terminálu), atribut a symbolu X se označuje $X.a$
  - Mohou se objevit při výpočtech cykly, což je velmi nežádoucí (nelze to poté dopočítat)
- K čemu se používá:
  - Rozšířené (překladové a atributované) gramatiky jsou uloženy ve vstupním definičním souboru a program překladače je vygenerován pak na základě těchto definičních souborů.
  - Je to způsob jak popsat sémantiku ne pouze syntaxi.
  - Používá se často k generování nějakého mezikódu (v parseru typicky).
  - Byť pravidla pro vypočítání AST a tříadresného kódu se různí, dají se zapsat v této fázi a určitým způsobem jsou ekvivalentní.
- Atributy:
  - Atributy mohou být dědičné a syntetizované,
  - vstupní-terminání symboly mají pouze syntetizované,
  - výstupní-terminální symboly mají pouze dědičné,
  - neterminální symboly mohou být oba typy atributů,
  - hodnota atributu je udána atributovým (sémantickým) pravidlem,
  - vše co lze popsat atributovou gramatikou lze popsat pouze pomocí syntetizovaných pravidel,
  - $A \to \alpha X \beta \mid X.d = \mathrm{f}(A \to \alpha X \beta), A.s = \mathrm{g}(A \to \alpha X \beta)$.

11 Jak je definována L–atributová gramatika?

(L jako levá, nikoliv LL)

- Závislosti mezi atributy mohou být různé a ne vždy lze hodnotu daných atributů vyhodnotit jedním průchodem atributovým derivačním stromem. K tomu abychom mohli hodnoty vyhodnotit během jednoprůchodové syntaktické analýze byla zavedena L-atributovaná gramatika.
- V L-atributované gramatice lze každý atributovaný derivační strom vyhodnotit jedním průchodem při zpracovávání vstupních symbolů (listů stromu) zleva doprava.
- Pokud je gramatika L-atributová, pak jde operace LL (i LR) analýzy rozšířit o pravidla tak, že výpočet atributů je realizován jednoprůchodovým atributovým překladem řízeným LL (LR) analýzou.
- Pravidla L-atributové gramatiky jsou:
  - nechť máme pravidlo $A \to \alpha X \beta$,
  - potom dědičný atribut d symbolu X závisí na dědičných atributech A a atributech (syntetizovaných i dědičných) z α,
  - a každý syntetizovaný atribut s symbolu A závisí pouze na dědičných atributech A a na atributech (syntetizovaných i dědičných) z pravé strany ($\alpha X \beta$).
- L-atributová gramatika je například ta, která má pouze syntetizované atributy (také S-atributová gramatika).
- Implementace LL analýzy při L-atributové gramatice:
  - Máme pro každý neterminál jednu proceduru,
  - dědičné atributy neterminálu jsou vstupní parametry,
  - syntetizované atributy neteterminálu jsou výstupní parametry,
  - atributová sémantika pravidla je implementována v kódu procedury.

12 Jak se rozšíří implementace LL(1) analyzátoru za účelem provádění jednoprůchodového formálního překladu a výpočtu atributů?

- Zavedeme L-atributované gramatiku, kde lze každý atributovaný derivační strom vyhodnotit jedním průchodem při zpracovávání vstupních symbolů (listů stromu) zleva doprava.
- Pokud je gramatika L-atributová, pak jde operace LL (i LR) analýzy rozšířit o pravidla tak, že výpočet atributů je realizován jednoprůchodovým atributovým překladem řízeným LL (LR) analýzou.
- Pravidla L-atributové gramatiky jsou:
  - nechť máme pravidlo $A \to \alpha X \beta$,
  - potom dědičný atribut d symbolu X závisí na dědičných atributech A a atributech (syntetizovaných i dědičných) z α,
  - a každý syntetizovaný atribut s symbolu A závisí pouze na dědičných atributech A a na atributech (syntetizovaných i dědičných) z pravé strany ($\alpha X \beta$).
- L-atributová gramatika je například ta, která má pouze syntetizované atributy (také S-atributová gramatika).
- Implementace LL analýzy při L-atributové gramatice:
  - Máme pro každý neterminál jednu proceduru,
  - dědičné atributy neterminálu jsou vstupní parametry,
  - syntetizované atributy neteterminálu jsou výstupní paramety.
  - atributová sémantika pravidla je implementována v kódu procedury.

13 Co je a k čemu se používá tříadresový kód?

(3AC/TAC - Three Address Code)

- Je to jeden ze dvou základních mezikódů (IR - Intermediate Representation).
- Používá se často, protože to zvyšuje přenosnost kódu a pro optimalizace.
- Obecně se dá říct, že pro 3AC mezikód se dají využít jazyky, které jsou svou komplexitou někde mezi high-level languages (jako C, ale ne jeho celá funkcionalita) a low-level languages (jako asm - asembly languages).
- U 3AC jazyka jsou jeho instrukce o maximálně třech operandech.
- Na pravé straně má instrukce maximálně jednu operaci, například:

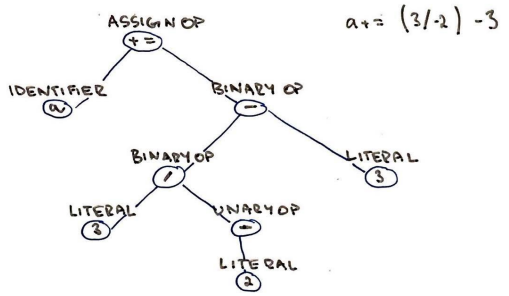
  `x*y+z => tmp1 = x * y; tmp2 = tmp1 + z`

- Instrukce mohou být:
  - Pro binární operace `x = y binOp z`
  - Pro unární operace `x = unOp y`
  - Pro kopírování `x = y`
  - pro control flow `false? x goto y`
- Můžeme mít 3AC pro zásobníkový počítač:
  - Má to jednoduchý překlad výrazů a interpret
  - Místo registrů má zásobník (nejsme limitovaní na 16/32 registrů)
  - Obtížně se optimalizuje
  - Sémantický rozdíl mezi tím jak dnešní procesory fungují (registry vs. zásobník)

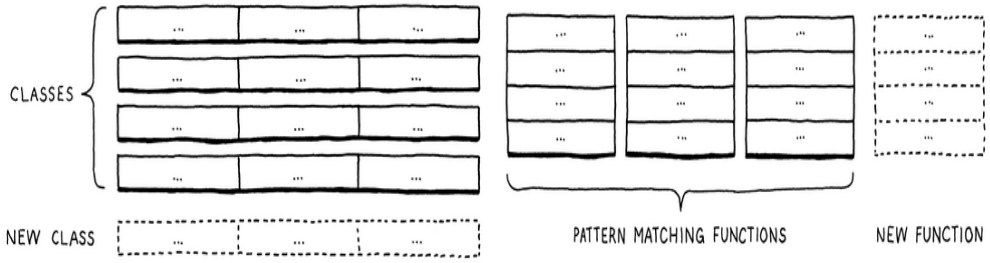
14 Co je a k čemu se používá abstraktní syntaktický strom?

(AST - Abstract Syntax Tree)

- Je to jeden ze dvou základních mezikódů (IR - Intermediate Representation).
- Říkáme *abstraktní* protože nezaznamenává každý detail zdrojového kódu, jako například středník a podobně (to je hlavní rozdíl oproti parse trees).
- Je vetšinou výsledkem parseru.
- AST se používá pro sémantickou analýzu.
- AST zaznamenává význam kódu na vstupu jako strukturu operátorů a jejich operandů.



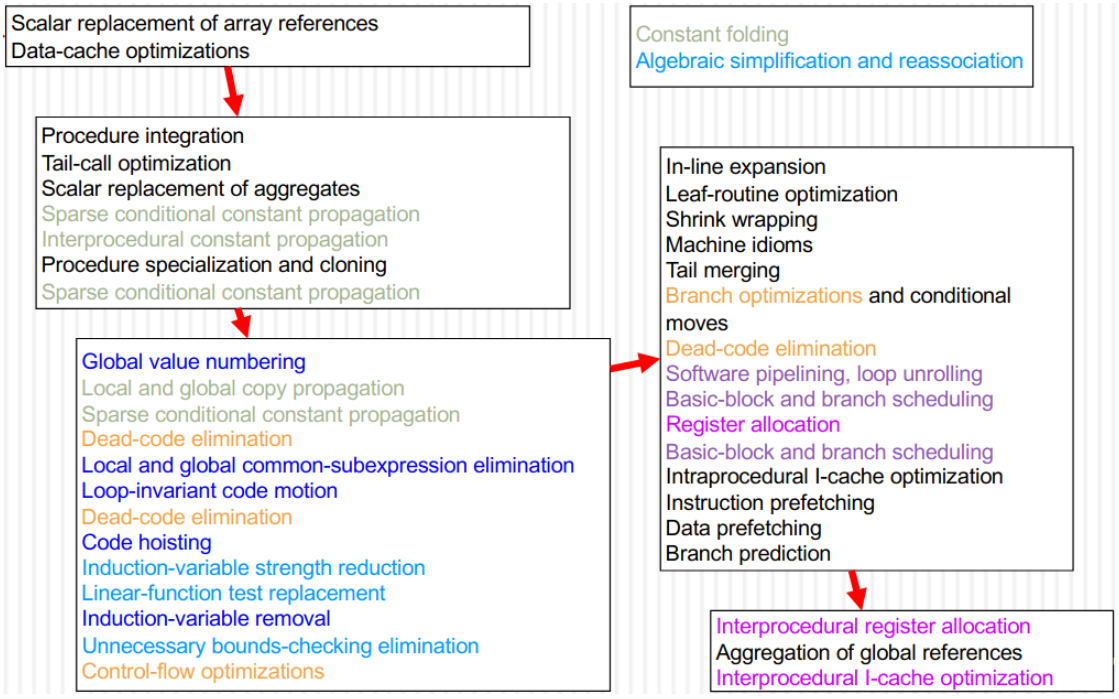
- Vnitřní uzly jsou operátory (nějaká akce) a jejich potomci jsou operandy (argumenty té akce).
- Listy AST jsou jednoduché operátory - literály.
- Pravidla v atributové gramatice vytvářejí a předávají si pointery na uzly.
- AST může mít extra informace o kódu (například pozice tokenu v zdrojovém kódu), aby pomohl dobrým chybovým hláškám.
- Dá se implementovat pomocí visitor pattern, aby dovololal jednoduché přidávání method a zároveň tříd (simulování principů funkcionálního programování). Visitor pattern přidává další *level of indirection*, který toto umožňuje.



- Pomocí struktury AST můžeme generovat kód, který reprezentuje jeho význam.

15 Uveďte příklady lokálních optimalizací kódu.

- Optimalizace znamená, že jsme nějak upravili kód, který se vykonává, aby byl lepší (například rychlejší, paměťově méně náročný atd).
- Kód se dá optimalizovat jak na frontendu tak i backendu překladače.
- V rámci frontendu se dělají optimalizace lexikální, syntaktické i základní sémantické.
- Optimalizace se dají klasifikovat na základě toho kolik o programu musíme vědět:
  - Lokální: takové, které se dějí v rámcí jednoho bloku
  - Globální: založené na analýze funkcí/procedur (několik bloků)
  - Interprocedurální: založené na analýze celých programů
- Kromě tohoto dělení se na optimalizace můžeme koukat i tak, že se dělí na HW závislé a nezávislé.
- Příklady frontend optimalizace logických výrazů:
  - `X | true  -> true`
  - `X | false -> X`
  - Přes atributovanou gramatiku lze popsat toto "zkrácení" výrazu
- V backendu se optimalizace dělají vnitřní reprezentací kódu.
- Postup optimalizací na backendu překladače.



- Příklady lokálních optimalizací:
  - Strength reduction: například místo násobení $*4$ se posune bitově o $<< 2$
  - Common sub expression elimination: místo opakovaného vypočítávání stejné věci se vytvoří extra proměnná, kde se to vypočte pouze jednou. Toto lze dělat i přes DAG
  - Code motion: Invariant expression (neměnící se výraz) se vypočte pouze jednou (například ve `for`).
  - Loop unrolling: nějak ve obsah `for` expanduje, aby se nemuselo tolikrát (nebo vůbec) skákat v kódu.
  - Dead code elimination: kód, který je zbytečný (`if (true) ...`), nebo se nikdy nevykonná lze smazat.
- Peephole optimization technique je způsob jak menší kusy kódu optimalizovat. Okénko je fixní velikosti a tudíž to nemá příliš velký overhead. Díky tomu se optimalizují věci jako constant folding ($4 * 2 = 8$), constant propagation, elimination of redundant stores and loads, strenght reduction (např. místo $2 * r_2 -> r_2 + r_2$), elimination of algebraic identities.