Syntaktická analýza založená na několika gramatikách

Ondřej Koumar
xkouma02@stud.fit.vutbr.cz

18. března 2024

Motivace

Syntaktická analýza a obecně překladače často zůstávají v praxi poněkud vzdálené od teoretických konceptů. I přes bohatou teorii formálních jazyků, konkrétně například gramatik a gramatických systémů, se v praxi často při syntaktické analýze upřednostňují jednoduché postupy.

Má bakalářská práce se snaží spojit teoretický svět s praktickým využitím. Pro syntaktickou analýzu využívám gramatický systém s několika gramatikami, ale implementace zůstává jednoduchá a srozumitelná pomocí dvou zásobníkových automatů, LL tabulky a precedenční analýzy výrazů. Výsledkem této analýzy bude abstraktní syntaktický strom.

Implementovaný jazyk

Nejdříve se podíváme na jazyk, který je navrhnutými gramatikami popsán. Zatím nemá jméno, nazvěme jej pracovně KOUBP. Je založen na jazyce IFJ22, který je založen na jazyce PHP. Jazyk KOUBP je staticky typovaný jazyk, tedy datový typ všech proměnných a konstant je znám už za překladu.

Konstrukce jazyka

Jazyk KOUBP má čtyři základní datové typy, a to `int`, `float`, `bool`, `string`. Obsahuje konstrukce, které byste očekávali u každého imperativního programovacího jazyka – `while` a `for` cykly, příkazy `if-elseif-else` a `return`, jednoduché aritmetické výrazy a operátory typu sčítání a odčítání, relační operátory, ale například i konkatenaci řetězců či logické operátory `&&` a `||`. Kód může být libovolně rozčleněn do funkcí, které mohou a nemusí mít návratový typ.

Oddělovač příkazů

Všechny příkazy musí být ukončeny středníkem, který se používá jako oddělovač příkazů. Výjimkou jsou bloky kódu uzavřené ve složených závorkách. Za takovým blokem středník následovat nemusí, ale uvnitř bloku kódu je žádoucí každý příkaz oddělit.

Blok kódu může být použit samostatně, nicméně typicky jej vidíme za konstrukcemi `while`, `for`, `if`, případně při definici funkce. Příklad bloku kódu lze vidět na obrázku **??**.

```
prikaz;
{
    prikaz;
    prikaz;
}
prikaz;
```

Obrázek 1: Příklad použití bloku kódu

Středník lze použít nejen jako oddělovač příkazů, může být také chápán jako samostatný příkaz. Jeho význam se dá přirovnat k instrukci `NOP` ve strojovém jazyce/asembleru.

Gramatický systém

Gramatický systém, který popisuje jazyk KOUBP, je postaven na *CD (cooperating distributed)* gramatických systémech. Tento systém se skládá ze šesti komponent, z nichž každá reprezentuje část navrhovaného jazyka tak, aby pravidla byla systematicky rozdělena. Motivace k tomuto přístupu je podobná členění zdrojových kódů do logických celků. Zároveň je ale důležité, aby se moc nezvýšila náročnost implementace a vyplatilo se rozdělení do více gramatik udělat.

Následují gramatiky/komponenty, které dohromady tvoří gramatický systém jazyka KOUBP. V nadpise vizte ucelenou část jazyka, kterou daná pravidla tvoří.

Struktura programu

```
<program> -> <statement>₂ <statementList>₁
<program> -> <functionDef>₂ <statementList>₁

<statementList> -> <statement>₂ <statementList>₁
<statementList> -> <functionDef>₃ <statementList>₁
<statementList> -> ε
```

Příkazy a běžné konstrukce

```
<statement> -> if ( <expression>₄ ) <codeBlock>₅ <if2>₂
<statement> -> while ( <expression>₄ ) <codeBlock>₅
<statement> -> for ( <expression>₄ ; <expression>₄ ; <expression>₄ ) <codeBlock>₅
<statement> -> <expression>₄ ;
<statement> -> return <returnExp>₂ ;
<statement> -> <codeBlock>₅
<statement> -> ;

<if2> -> elseif ( <expression>₄ ) <codeBlock>₅ <if2>₂
<if2> -> else <codeBlock>₅
<if2> -> ε

<returnExp> -> <expression>₄
<returnExp> -> ε
```

Definice funkce

```
<functionDef> -> function funcName ( <params>₃ ) : <funcType>₃ <codeBlock>₅

<params> -> <type>₆ variable <params2>₃

<params2> -> , <type>₆ variable <params2>₃
<params2> -> ε

<funcType> -> <type>₆
<funcType> -> ε
```

Výrazy

```
<expression> -> <expression>₄ ⌊+ - * / . && || == != > < >= <=⌉ <expression>₄
<expression> -> ⌊! -⌉ <expression>₄
<expression> -> variable
<expression> -> constant
<expression> -> funcName ( <args>₄ )
<expression> -> ( <expression>₄ )

<args> -> <expression>₄ <args2>₄
<args> -> ε
```

<args2> -> , <expression>$_4$ <args2>$_4$
<args2> -> ε

Blok kódu

Všimněme si, že neterminály *codeBlock*$_5$ a *statement*$_2$ se mohou navzájem derivovat donekonečna a potenciálně by mohl nastat *deadlock*. Po teoretické stránce se s tímto faktem počítá a gramatiky se nebudou nijak přizpůsobovat, tato problematika bude vyřešena implementačně, zásobníkový automat se s pomocí LL tabulky bude chovat deterministicky.

<codeBlock> -> { <statements>$_5$ }
<codeBlock> -> <statement>$_2$

<statements> -> <statement>$_2$ <statements>$_5$
<statements> -> ε

Datové typy

<type> -> int
<type> -> float
<type> -> string
<type> -> bool

Implementace

Implementujeme syntaktický analyzátor v jazyce C++ s dvěma zásobníkovými automaty – jeden pro analýzu s LL tabulkou a druhý pro precedenční analýzu výrazů. Precedenční analýza výrazů bude prováděna obvyklým způsobem, tedy redukcí terminálů a případně pravidel na zásobníku.

Analýza s LL tabulkou bude upravena na používání uspořádaných dvojic *(číslo gramatiky, číslo pravidla)* namísto pravidel v položkách tabulky. Číslo gramatiky bude předáno tovární metodě, která vytvoří instanci jedné z gramatik dědících od abstraktní třídy `Grammar`. Tato třída obsahuje polymorfní metodu `Expand(unsigned ruleNumber)`, která vrací reverzovaný seznam terminálů a neterminálů reprezentující pravou stranu daného pravidla. Reverzované pravidlo se vloží na zásobník a analýza bude pokračovat dalším cyklem.

V každém kroku syntaktické analýzy bude generován uzel abstraktního syntaktického stromu.

Možná rozšíření

Syntaktický analyzátor a jeho datové struktury jsou koncipovány s ohledem na jednoduchou rozšiřitelnost projektu. Aktuálně pracujeme s pevně danou posloupností terminálů/tokenů, kterou analyzátor přijímá jako seznam. Nicméně, máme v plánu budoucí rozšíření projektu o lexikální analyzátor, což umožní psát skutečné programy v jazyce KOUBP.

Týká se to i jednoduchých sémantických kontrol, které jsou v každém kvalitním syntaktickém analyzátoru žádoucí.