Konstrukce kompilátorů

Obsah

1 Gramatika

2 Lexikální analýza

2.1 Datová struktura pro popis tokenů

3 Syntaktická analýza a syntaktické stromy

3.1 Ambiguita

Ambiguita (vícevýznamovost) v syntaktické analýze nastává, když daný vstupní řetězec odpovídá více než jednomu možnému syntaktickému stromu (AST). Jinými slovy, existuje více než jedna správná interpretace stejného vstupu podle definované gramatiky. Ambiguita je problém, protože syntaktický analyzátor (parser) potřebuje zvolit jedinou správnou interpretaci, jinak nemůže jednoznačně pokračovat v překladu.

V kontextu návrhu programovacích jazyků platí, že **dobře navržená gramatika je neambiguózní**.

3.1.1 Příčiny ambiguity

Ambiguita může vznikat z několika důvodů:

- **Překrývající se pravidla gramatiky** - Pokud dvě různá pravidla dokáží pokrýt stejný vstup, parser neví, které použít. Například:

  ```
  Expr ::= Number | Expr "+" Expr | Expr "*" Expr
  ```

  Řetězec $2 + 3 * 4$ může být interpretován buď jako $(2 + 3) * 4$ nebo jako $2 + (3 * 4)$
- **Nedostatek precedencí a asociativity** - Když nejsou jasně definovaná pravidla pro pořadí operací, vzniká více možných stromů.
- **Chybějící oddělovače** - V jazycích bez ukončovacích symbolů (např. středníku) může být problém určit, kde jeden příkaz končí a další začíná. To je zvlášť relevantní u návrhů s volnou syntaxí, kde konec určuje jen struktura závorek.
-

3.1.2 Dopady ambiguity na parser

- **LL a LR parsery** vyžadují, aby gramatika byla jednoznačná. Ambiguózní gramatiky mohou vést k chybám typu shift/reduce conflict nebo reduce/reduce conflict.
- **Backtracking parsery** (např. rekurzivní sestup s backtrackingem) zvládnou ambiguitu vyřešit zkoušením všech možností, ale to je výpočetně náročné a nevhodné pro kompilátory.

3.2 Abstraktní syntaktický strom

3.3 Pretty print

"Pretty printing" abstraktního syntaktického stromu (AST) je proces převodu stromové struktury, která reprezentuje zdrojový kód programu, zpět do čitelné textové podoby. Cílem není nutně replikovat přesný původní kód (například mezery, komentáře a redundantní závorky jsou často vynechány v AST), ale vytvořit srozumitelnou a formátovanou reprezentaci kódu. Pretty printing AST je v podstatě traverzace (průchod) stromu, kde se pro každý navštívený uzel generuje odpovídající řetězec kódu.

3.3.1 Princip Pretty Printingu

Klíčovými aspekty jsou:

- **Rekurzivní traverzace**: Pretty printer typicky používá rekurzivní funkci nebo metodu, která prochází stromem od kořene dolů. Pro každý typ uzlu existuje specifická logika pro jeho tisk.

- **Generování textu**: Každý uzel je převeden na svůj textový ekvivalent. Jednoduché uzly, jako jsou čísla, řetězce nebo názvy proměnných, se jednoduše vytisknou. Binární operátory (např. +, -,
-) se tisknou mezi jejich operandy. Příkazy (např. if, while, přiřazení) mají specifickou strukturu, která se generuje s klíčovými slovy jazyka a správným odsazením.
- **Formátování a odsazení**: Toto je klíčová část "pretty" tisku. Pro zvýšení čitelnosti se vnořené bloky kódu (např. těla funkcí, bloky if nebo while) odsazují. Pretty printer udržuje aktuální úroveň odsazení a přidává odpovídající počet mezer nebo tabulátorů před každý řádek kódu. Když je řádek příliš dlouhý, pretty printer se snaží najít vhodné místo pro zalomení řádku a pokračovat na novém odsazeném řádku. Někdy se přidávají prázdné řádky pro oddělení logických bloků kódu. Protože AST obvykle neobsahuje závorky, které jsou redundantní, pretty printer musí být schopen znovu vložit závorky tam, kde jsou potřeba pro zachování správné priority operátorů. To se obvykle řeší porovnáváním priority rodičovského operátoru s prioritou podřízeného operátoru.

3.3.2 Typické implementační vzory

- **Vzor Návštěvník (Visitor Pattern)**: Toto je velmi častý vzor pro procházení a zpracování AST. Definuje se rozhraní Visitor s metodou visit pro každý typ uzlu AST. Konkrétní PrettyPrinterVisitor implementuje tyto metody tak, aby generovaly příslušný kód. Každý uzel AST pak má metodu accept(Visitor v), která volá příslušnou visit metodu na návštěvníkovi. To odděluje logiku pretty printingu od definice samotného AST.
- **Rekurzivní funkce**: Jednodušší přístup pro menší jazyky nebo pro rychlé prototypy. Každá funkce odpovídá typu uzlu a rekurzivně volá funkce pro své potomky.

3.3.3 Odsazení v pretty printingu

Nejdůležitější v pretty printingu pro zvýšení přehlednosti je správné formátování výsledného výpisu, to znamená, zalomení řádků, mezery a odsazení od začátku řádku, které definuje úrovně vnoření syntaktického stromu respektive konstrukce v parsovaném jazyce.

4 Top-Down parsery

5 Bottom-Up Parsery

6 Tabulka symbolů

7 Zpracování chyb

Zpracování chyb je složitý mechanismus, který se dělí na tři hlavní fáze: **detekci chyb**, **zotavení z chyby** a **reportování chyb**.

7.1 Detekce chyb (Error Detection)

Detekce chyb je proces identifikace konstrukcí v kódu, které porušují pravidla daného jazyka. V kompilátoru nebo obecně v parserech probíhá na různých úrovních:

- **Lexikální chyby** - Vznikají, když lexikální analyzátor narazí na posloupnost znaků, kterou nedokáže rozpoznat jako platný token. Příkladem může být neznámý symbol nebo neukončený řetězec.
- **Syntaktické chyby** - Objevují se, když se tokeny objeví v nesprávném pořadí, což porušuje gramatiku jazyka. Klasickým příkladem je chybějící středník ';' na konci příkazu nebo nesprávně spárované závorky.
- **Sémantické chyby** Jsou složitější a objevují se až po parsování. Kód je sice syntakticky správný, ale porušuje sémantická pravidla. Například, při použití proměnné, která nebyla deklarována, nebo se přiřadí řetězec do proměnné číselného typu.

7.2 Zotavení z chyby (Error Recovery)

Jakmile kompilátor detekuje chybu, neměl by se zastavit. Cílem je pokračovat v analýze, aby mohl najít další chyby v kódu, aby mohl ohlásit všechny chyby najednou. Proces, který to umožňuje, se nazývá **zotavení z chyby**.

Existují dva základní přístupy:

- **Panický mód (Panic Mode)** - Toto je jednoduchý a často používaný mechanismus. Jakmile kompilátor narazí na chybu, začne ignorovat vstupní tokeny, dokud nenajde "bezpečný" nebo synchronizační token. Příkladem může být středník ';' nebo uzavírací složená závorka '{'. Kompilátor poté předpokládá, že daná chyba skončila a může začít znovu analyzovat od tohoto bodu. I když je implementace jednoduchá, tento přístup může vést k kaskádovým chybám (cascading errors), kdy jeden přeskočený token způsobí řadu dalších chyb, které ve skutečnosti neexistují.
- **Frázové zotavení (Phrase-level Recovery)** - Tento pokročilejší přístup se snaží provést lokální "opravu" chyby. Kompilátor se pokusí chybějící tokeny vložit nebo přebytečné smazat, aby se mohl vrátit do platného stavu. Díky tomu je přesnější a generuje méně kaskádových chyb, ale je také mnohem složitější na implementaci.

7.3 Mechanismy pro reportování chyb (Error Reporting)

Správa a reportování chyb je pro uživatele klíčová. Kvalitní kompilátor musí chyby sbírat, ukládat a prezentovat je srozumitelným způsobem. Existují dva hlavní přístupy, jak reportovat nalezené chyby:

- **Okamžité reportování (Immediate Reporting)** - Chyba je hlášena a vytištěna do výstupu okamžitě, jakmile je detekována. Jedná se o jednoduché řešení kdy uživatel okamžitě vidí, kde je problém, a může reagovat v reálném čase. Hodí se pro interaktivní prostředí a pro malé projekty. Okamžitý výpis chyb může být

nepřehledný, pokud kompilátor generuje kaskádové chyby. Zprávy o chybách se mohou míchat s jiným výstupem kompilátoru a mohou být chaotické. Navíc není možné předem seřadit chyby podle priority nebo umístění v souboru.

- **Bufferované reportování (Buffered Reporting)** - Chyby se nehlásí okamžitě. Místo toho se ukládají do **zásobníku chyb (error buffer)** jako struktury nebo objekty, které obsahují typ chyby, závažnost, umístění a zprávu. Po dokončení kompilace (nebo určité fáze) se celý zásobník zpracuje a vytiskne najednou. Chyby jsou prezentovány jako ucelený seznam, oddělený od ostatního výstupu. Chyby lze zároveň seřadit podle čísla řádku nebo sloupce, což usnadňuje jejich opravu. Je také možné filtrovat chyby podle jejich závažnosti, takže například je možné zobrazit jen chyby, ne varování. Reportovací systém může analyzovat shromážděné chyby a poskytnout lepší kontext nebo navrhnout možné řešení. Nevýhodou ale je že se chyby neobjeví, dokud kompilace nedoběhne do konce, což může zpomalit zpětnou vazbu, zejména u kompilací rozsáhlých souborů.

Pro většinu moderních kompilátorů je bufferované reportování preferovanou metodou. Umožňuje totiž organizovaný a přehledný výpis chyb, což je neocenitelné, zejména u větších projektů.

7.4 Chybový systém

Pro efektivní a přesné zpracování chyb nestačí pouze mít mechanismy pro jejich detekci a reportování. Kompilátor musí také uznat, že chyby vznikají v různých fázích kompilace a každá z nich nese odlišné typy informací. To vede k modulárnímu designu chybového systému, který se skládá z několika specializovaných částí.

Každá fáze kompilace má svůj vlastní detektor chyb a generuje specifické chybové záznamy, které jsou přizpůsobeny informacím dostupným v dané chvíli. Tyto chyby se pak ukládají do centrálního zásobníku chyb, který je buď okamžitě zpracovává, nebo je shromažďuje a vypíše na konci. Takový modulární přístup umožňuje robustní a přesné reportování chyb, což je pro vývojáře neocenitelné.

7.5 Zachytávání chyb ve fázi předzpracování kompilace

Zachyávání chyb z příkazové řádky, jako jsou chybné vstupní parametry nebo chybějící soubory, se liší od chyb uvnitř samotného zdrojového kódu. Tyto chyby se řadí do fáze **předzpracování kompilace**. Z pohledu architekta kompilátoru jde o systémové a infrastrukturní chyby, které je třeba ošetřit dříve, než se vůbec spustí hlavní fáze kompilátoru (lexikální, syntaktická, atd.).

Než se kompilátor pustí do analýzy zdrojového kódu, musí provést řadu kontrol na úrovni operačního systému a příkazové řádky. Tady se odehrává logika, která se stará o chyby, o kterých mluvíte:

- **Validace příkazové řádky** - Kompilátor analyzuje argumenty, které mu byly předány. Zde se kontroluje správnost syntaxe, platnost přepínačů a jejich hodnot.
- **Kontrola vstupních souborů** - Kompilátor ověří, zda zdrojové soubory, které mu byly předány, skutečně existují a jsou čitelné, případně zda existuje alespoň jeden vstupní soubor ke zpracování.
- **Nastavení prostředí** - Kompilátor může také kontrolovat proměnné prostředí nebo jiná systémová nastavení, která mohou ovlivnit jeho chování.

Zásadní rozdíl spočívá v tom, že tyto chyby brání kompilátoru v tom, aby vůbec zahájil svou hlavní práci. Zatímco lexikální, syntaktické a sémantické chyby se týkají obsahu zdrojového kódu, chyby z příkazové řádky se týkají volání kompilátoru samotného.

7.6 Kontext chyb

Aby byl chybový systém skutečně efektivní, musí být schopen zachytit a uložit relevantní data, která jsou k dispozici v dané fázi kompilace. Chybový objekt tak není jen jednoduchá zpráva, ale struktura, která obsahuje kontextové informace, které usnadní lokalizaci a opravu problému.

7.6.1 Chyby z lexikální analýzy

Lexer operuje s proudem znaků, a proto jsou informace o chybě na nejnižší úrovni. Lexikální chyba by měla obsahovat číslo řádku a sloupce kde se chyba ve zdrojovém kódu. Tato informace je kritická, protože umožňuje přesně ukázat na místo v kódu. Dále by měla obsahovat popis chyby, tedy například že chybí ukončovací apostrov řetězce, nebo že je chybný formát desetinného čísla. Dále může obsahovat doplňující data, buď samotný neplatný znak a nebo sekvenci kódu, která chybu způsobila.

7.6.2 Chyby ze syntaktické analýzy

Parser pracuje s tokeny a gramatikou, což mu umožňuje poskytnout podrobnější kontext. Opět by chyba měla obsahovat číslo řádku a sloupce kde se chyba ve zdrojovém kódu nachází. Opět by měla obsahovat i popis chyby a buď co parser očekával a nebo úsek kódu, který chybu způsobil. Propracovaný chybový systém pak možňuje předat jak očekávaný token (nebo jeden z možných tokenů), tak úsek zdrojového kódu ve kterém se chyba nachází případně i s popisem možného řešení.

7.6.3 Chyby ze sémantické analýzy

Sémantický analyzátor pracuje s abstraktním syntaktickým stromem (AST) a symbolovou tabulkou. Díky tomu dokáže generovat nejbohatší a nejpřesnější chybové zprávy. Pro lokalizaci chyby je opět potřeba číslo řádku a sloupce kde se chyba ve zdrojovém kódu nachází. Tato pozice je ale narozdíl od syntaktické nebo lexikální analýzy spolejena s celou gramatickou konstrukcí ve které se chyba nachází. To znamená, že chyba musí být popsána v rámci celé gramatické konstrukce. Jako doplňující informace se do chybového objektu vkládá i část abstraktního stromu ze kterého je složena textová podoba příkazu, která chybu vygenerovala. Pro každou chybu pak musí být vlastní funkce která z předaného AST složí chybový výpis, protože daná funkce bude vědět, že předaná část AST je daná gramatická konstrukce a bude vědět jak s ní manipulovat.

7.6.4 Mechanismy hlášení chyb

Tyto chyby se obvykle reportují okamžitě a přímo na standardní chybový výstup (stderr). Zpráva je stručná, jasná a nevyžaduje složitou strukturu. Tento typ chybového hlášení je navržen tak, aby byl jednoduchý, rychlý a efektivně komunikoval s uživatelem. Vzhledem k tomu, že se jedná o chyby na úrovni vstupu do kompilátoru, není nutné používat složité bufferovací mechanismy. Kompilátor pouze zkontroluje vstup, a pokud je neplatný, ohlásí problém a ukončí se.

8 Mezijazyk