1 Úvod do kompilátorů

1.1 Úvodem několik slov

Tyto texty obsahují úvod do návrhu programovacích jazyků a problematiky překladu programů. Téma pokrývá oblasti zahrnující lexikální analýzu (scanning), regulární výrazy, bezkontextové gramatiky a syntaktickou analýzu (parsing), překlad řízený syntaxí, abstraktní syntaktické stromy, úrovně-hladiny platnosti entit (scoping), tabulky symbolů a generování kódu.

1.2 Trochu historie

Objem našich znalostí o tom, jak organizovat a psát kompilátory prudko narostl od doby existence prvních kompilátorů. Je těžké určit přesnější datum vzniku prvního kompilátoru, protože zpočátku se objevovalo veliké množství pokusů a implementací, na kterých se podílelo nezávisle na sobě více skupin. První kompilátory se objevovali od počátku 50-tych let 20. století. V onom období byli považované za jeden z nejtěžších a nejkomplikovanějších problémů v oblasti softvéru.

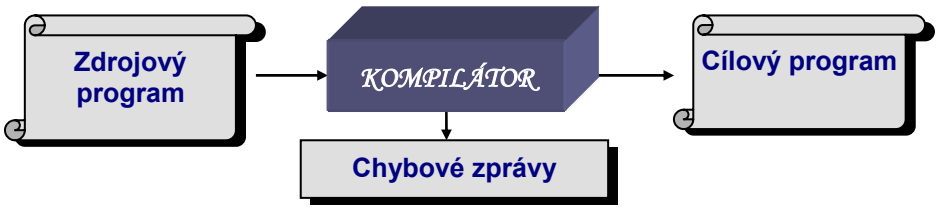
Termín **kompilátor** vznikl právě v tomto období. Programátor-analytička **Grace Murray Hopperová** použila pojmenování **kompilace** pro posloupnost podprogramů vybraných z připravené knižnice za účelem rychlejšího sestavení programu na řešení problému za pomoci počítače. Tomuto postupu se hovořilo taktéž **automatické programování**. V té době vládla hluboká skepse v úspěšnost tohoto postupu, ale pozdější vývoj hlavně díky vybudování teoretického aparátu ukázal správnost nastoupené cesty. Možno diskutovat i o terminologické správnosti názvu překladače. Spíš by bylo vhodnější pojmenování *translátor*, ale podobně jako je tomu ve výpočetní technice i u mnohých dalších pojmech, z historických důvodů se ujal právě termín **kompilátor**.

Jedním z prvních známých programovacích jazyků vyšší úrovně bol **FORTRAN** – *formula translation*, problémově orientovaný jazyk pro řešení matematických úloh. Ve firmě IBM byl k němu vyvinutý týmem vedeným **Johnem Backusem** mezi lety 1954-1957 kompilátor. Jeho vývoj spotřeboval 18 člověko-let (staff-years) implementace. Stavba tohoto kompilátoru nebyla založena na nějakém teoretickém základě. Jeho struktura, komponenty a techniky vznikali *ad-hoc*, t.j. pro daný případ během jeho budování. Přibližně v tom čase začal **Noam Chomsky** svůj výzkum v oblasti struktury přirozených jazyků.. Výsledky jeho práce se výrazným způsobem odrazili ve vývoji a tvorbě moderních jazyků a jejich kompilátorů (o tom však pohovoříme později).

Od těch dob vývoj velice pokročil a vyvinuli se systematické techniky pro vyřešení mnohých problémů, které se vyskytují během kompilace. Kromě samotných programovacích jazyků byli vyvinuty i kvalitnější vývojové prostředí a různé softvérové nástroje umožňující tvorbu kvalitních programů a programových systémů. Dnes už jsou kompilační techniky dobře srozumitelné a jednoduchý, ale kvalitní kompilátor může vzniknout za několik měsíců.

1.3 Co je to kompilátor?

Kompilátor patří k základům moderního zpracování údajů. Při tvorbě nového softvéru se dnes už nedá pracovat bez tohoto produktu. Je spolu s programovacím jazykem základním dorozumívacím prostředkem mezi autorem programu a procesorem. Pro člověka je strojový jazyk těžko čitelný a ladění v něm je velmi náročné. Vytvoření procesoru schopného zpracovávat program přímo ve vyšším jazyku je nesmyslem. První pohled na kompilátor z hlediska běžného uživatele možno charakterizovat jako takzvanou **černou skřínku**, obsah které nepoznáme. Poznáme však vlastnosti této skřínky a účel, na který ho můžeme využít:



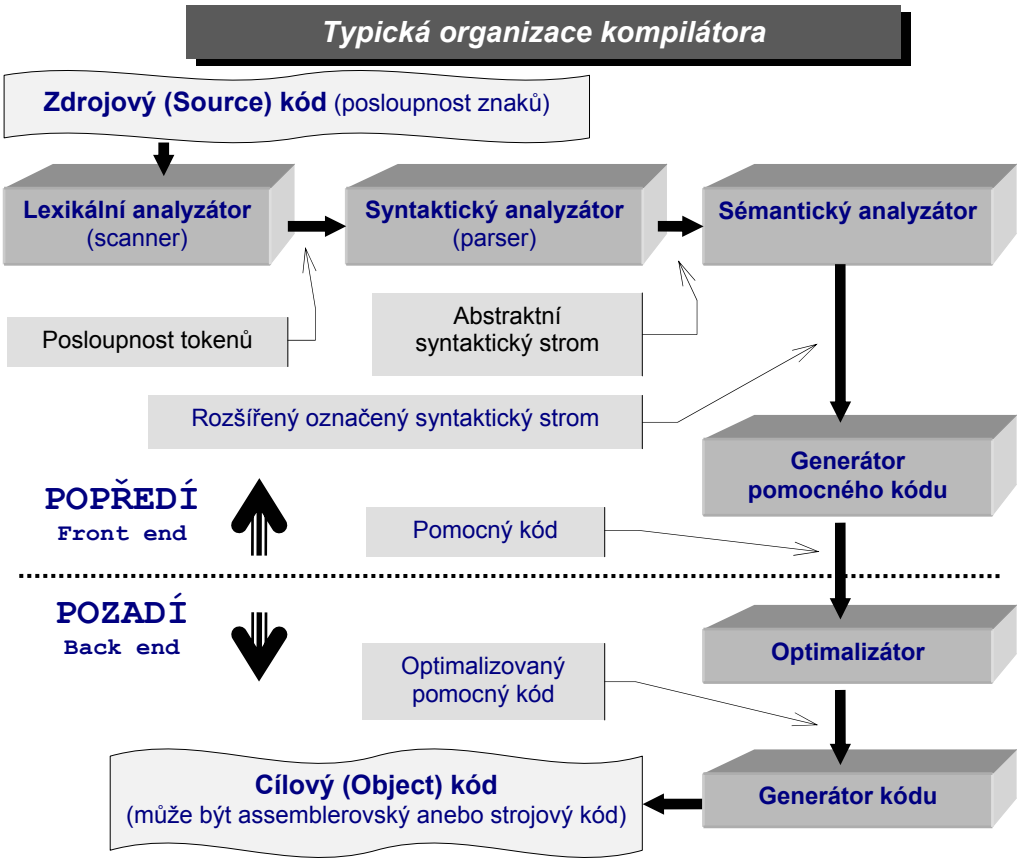
Koncepčně možno kompilátor rozdělit na dvě základní části:

- **Analýza** - rozpoznává nějaký zdrojový jazyk **L.**
- **Syntéza** - vytváří cílový kód v nějakém cílovém jazyku **L'.**

Analýza rozloží text zdrojového programu do podstatných významových částí a vytvoří vnitřní reprezentaci programu. Syntéza vytvoří z vnitřní reprezentace cílový program ve tvaru, který potom dokáže zpracovávat cílový procesor, pro který byl použitý kompilátor.

1.4 Běžná struktura kompilátoru

Kompilátor je samostatný program napsaný v nějakém hostitelském jazyku. (Například překladač jazyka Pascal s použitím jazyka C++ jako hostitelského jazyka).



Kompilátor pracuje ve více fázích; každá fáze překládá zdrojový program z jedné reprezentace do jiné. Různé kompilátory můžou obsahovat různé fáze, dokonce můžou být i různě seřazené.

1.5 Fáze kompilátora:

V procese kompilace se analýza skládá z troch fází:

1. **Lineární analýza** - proud znaků tvořící zdrojový program je čítaný zleva doprava, jsou vyhledávané významové posloupnosti znaků tvořící vždy nedělitelný jedno resp. víceznakový symbol - lexéma. Každé lexéme je přirazený token. Této fázi se častěji hovoří lexikální analýza, anebo **scanning**.
2. **Hierarchická analýza** - symboly anebo tokeny jsou sdružované do syntaktických celků podle přesně stanovených gramatických pravidel jazyka. Tuto fázi voláme syntaktickou analýzou - **parsingem**.
3. **Sémantická analýza** - vykonávají se určité kontroly pro zajištění správného významu příslušných komponent v celkovém kontexte programu, případně jeho částí.

1.5.1 SCANNER - lexikální analyzátor

Scanner je volaný z parseru, přičemž vykonává následovní činnosti:

- čte jednotlivé znaky ze zdrojového programu,

- seskupuje znaky do **LEXÉM** (jsou to posloupnosti znaků, které „patří k sobě“; pojmenovávají entitu),
- každý lexém odpovídá jednomu **TOKEN**-u (prvek jazyka s definovaným významem); scanner vrací následující token (případně další přídavnou informaci) do parseru,
- scanner může taktéž odhadovat lexikální chyby (např. nekorektní znaky).

Definice lexém, tokenů a nekorektních znaků závisí na definici zdrojového jazyka. V následující tabulce najdete několik příkladů přirazení tokenů textovým symbolům zdrojového jazyka.

Příklad v jazyce Pascal:

lexéma - její text	:	index	:=	tmp	37	10.2
odpovídající token	COLON	IDENT	ASSIGN	IDENT	INT-LIT	REAL-LIT
význam	dvojtečka	identifikátor	přirazení	identifikátor	celočís.literál	reálný literál

V prvním řádku je vždy posloupnost znaků dekódovaná jako lexéma jazyka. Ve druhém řádku je kód (=token) přirazený zjištěné lexémě. Ve skutečnosti jde o číselné vyjádření ve vnitřní reprezentaci. Pro lepší čitelnost scanningu budeme namísto takového kódu používat dohodnutý symbol složený ze znaků.. Ve vnitřním kódu mají všechny tokeny stejný rozměr, například jeden byte. Ve třetím řádku je význam lexémy v programovacím jazyku.

Poznámky: *- víceré lexémy můžou odpovídat stejnému tokenu,*
- tokeny COLON, atd. jsou v skutečnosti například jednobytové symboly (kódy),
- literál je číselná hodnota zapsaná v textovém tvaru.

Příklad překladu přiřazovacího příkazu v Pascalu:

zdrojový kód	**pozice**	**:=**	**zaklad**	**+**	**posun**	*****	**60**
tokeny	IDENT	ASSIGN	IDENT	PLUS	IDENT	TIMES	INT-LIT
význam	identifikátor	přiřazení	identifikátor	operátor +	identifikátor	krát	celočís. literál

Příklady nekorektních znaků v Pascalu: Např.. znaky **!** **?** **ctrl-a** mimo řetězce.

1.5.2 PARSER - syntaktický analyzátor

- seskupuje tokeny do "gramatických vět-frází", vyšetřuje gramatickou strukturu zdrojového programu,
- hledá syntaktické chyby, například v **Pascalu**:

jestli je zdrojový text	**pozice**	**:=**	*****	**5**
posloupnost tokenů je	IDENT	ASSIGN	TIMES	INT-LIT

!!! Všechno jsou to legální tokeny, ale jejich posloupnost je chybná !!!

- může hledat i některé "statické sémantické" chyby, Např.. nedeklarované proměnné, případně víckrát deklarované proměnné,
- generuje vnitřní reprezentaci programu vyjádřenou ve tvaru **abstraktního syntaktického stromu** a buduje program ve vnitřním jazyku.

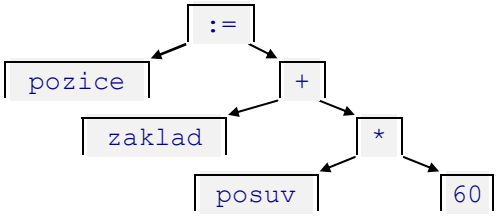
Příklad syntaktického stromu:

- vnitřní uzly stromu jsou OPERÁTORY,
- synové uzlu jsou jeho OPERANDY,
- každá část podstromu je samostatnou "logickou jednotkou", ***například***: podstrom s operátorem ***** i s jeho kořenem ukazuje, že násobení má vyšší prioritu jako součet, t.j. tato operace musí být vybrána jako významová jednotka,
  tedy ne: `zaklad + posuv`

Zdrojový kód

```
pozice := zaklad + posuv * 60
```

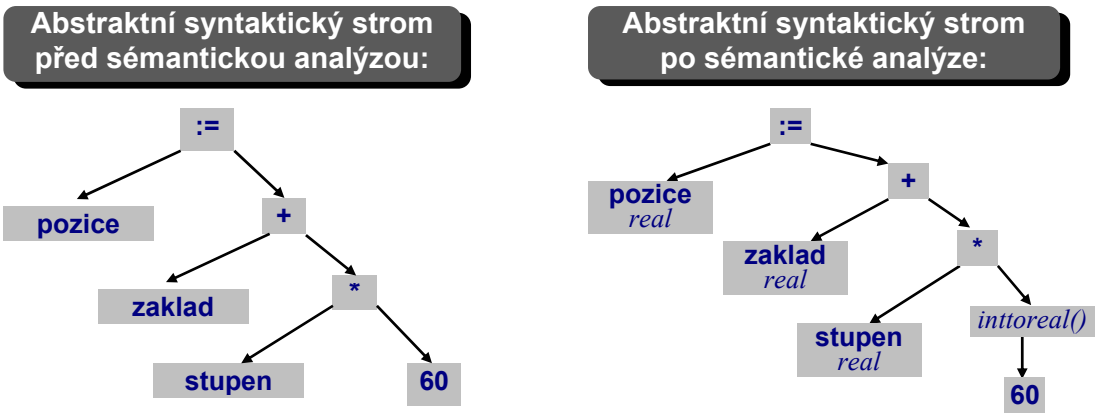
Abstraktný syntaktický strom



1.5.3 Sémantický analyzátor

Kontroluje do hloubky významovou stránku překládaného programu a hledá "statické sémantické" chyby, například chyby typů. Sbírá typové informace pro fázi generování kódu.

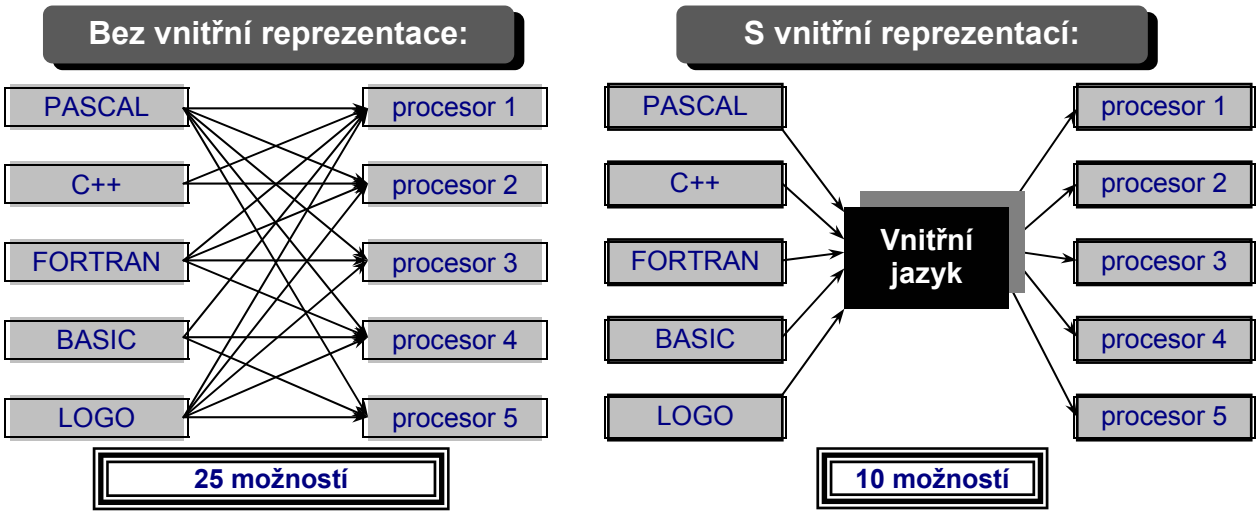
- Jako vstup používá hierarchickou strukturu vytvořenou fází syntaktické analýzy.
- Zjišťuje kompatibilitu typů jednotlivých elementů a v případě, že specifikace dovoluje určitou kompatibilitu mezi typy (například `integer` a `real`), doplňuje uzly do syntaktického stromu o možnost konverze typu hodnoty na kompatibilní typ. Hovoříme tomu **typová kontrola**.
- Výsledkem činnosti sémantiky může být **označení uzlů vlastnostmi**, případně i změna celého abstraktního syntaktického stromu.



1.5.4 Generátor vnitřního kódu

Překládá zápis z abstraktního syntaktického stromu do pomocného vnitřního kódu. Jde vlastně o vnitřní reprezentaci zdrojového programu jako jakýsi program pro abstraktní počítač. Táto reprezentace má dvě velmi důležité vlastnosti:

- je ho možné lehčeji vytvořit, než konkrétní cílový program,
- je lehčeji přeložitelný do cílového jazyka, než původní zdroj ve vyšším programovacím jazyce.



Vzniká tím koncepce, která určitým způsobem odděluje od sebe fáze analýzy a syntézy. Efektivněji je potom možné vyvíjet kompilátory mezi různými zdrojovými jazyky a cílovými jazyky.

Vnitřní jazyk, kód může mít více forem. Jednou z nejpoužívanějších forem je tzv. „tříadresový kód“ podobný assemblerovskému jazyku. V tomto kódu každá operace potřebuje ke své činnosti

- **nejvíce tři** operandy (dva pro zpracovávané hodnoty, jeden pro výsledek operace),
- jeden operátor pro vykonání operace a
- operátor přiřazení pro uložení.

Příklad zápisu v dohodnutém vnitřním jazyce:

```
temp1 :=      inttoreal(60)
temp2 :=      stupen * temp1
temp3 :=      zaklad + temp2
pozice :=     temp3
```

Tímto způsobem může být definována „jednotná šablona“ pro vykonávání akcí.

- Vytvoří se posloupnost operací, která jednoznačně určí posloupnost instrukcí určující pořadí, v jakém se budou vykonávat akce programu. Nemusí se už například uvažovat priorita aritmetických operací..
- Generují se dočasné jména pro hodnoty pro potřeby dalšího překladu..
- Některé instrukce můžou mít, samozřejmě, méně než tři operandy.
- Kromě uvedených operací musí reprezentace vykonávat i jiné typy instrukcí, jako jsou řídící instrukce, volání procedur a podobně.

Tato část kompilátoru vytváří **vnitřní kód**, který by měl být nezávislý na cílovém kódu. Je určený pro výstup informace z „**popředí**“ kompilátoru ve všeobecném tvaru, který možno zpracovat do konečného tvaru různými způsoby.

Poznámka: Tuto část (generátor vnitřního kódu) možno někdy nahradit přímo generátorem cílového kódu, jestli tvoříme cílový kód kompilátorem přímo na míru bez potřeby jeho dalších úprav. Zanikne tím struktura popředí a pozadí kompilátoru. Takovýto kompilátor však není je přenositelný na jiný typ procesoru.

1.5.5 Optimalizátor

Pokouší se vylepšit kód vytvořený generátorem pomocného kódu na základě znalostí cílového jazyka. Cílem může být vytvoření kódu rychleji pracujícího programu, anebo kódu zabírajícího méně místa. Přijímá pomocný kód z popředí a analyzuje možnosti jeho „vylepšení“ vzhledem na rychlost a na úsporu paměti. Výsledkem je **optimalizovaný vnitřní kód**. Například k předcházejícímu příkladu:

```
temp2 :=      stupen * 60.0
pozice :=     zaklad + temp2
```

Problematika tvorby optimalizátoru je složitým samostatným problémem, kterým se na tomto místě nebudeme podrobněji zaobírat!

1.5.6 Generátor cílového kódu

Generuje **cílový kód** z (optimalizovaného) pomocného kódu. Je úplně závislý na cílovém prostředí, ve kterém má pracovat přeložený program. Například:

```
LOADF     rate,R1
MULF      #60.0,R1
LOADF     zaklad,R2
ADDF      R2,R1
STOREF    R1,pozice
```

Důležitá je i volba cílového jazyka podle účelu.

- **Jazyk symbolických adres** - assemblerovský jazyk. Používá se v případě existence kvalitního překladače Asembleru. V našem případě bude vhodným cílovým jazykem pro potřeby výuky.
- **Čistý strojový kód** - generování kódu pro speciální instrukční množinu bez předpokladu existence operačního systému a knižnic. Funguje nezávisle na jiném softvéru.
- **Rozšířený strojový kód** - používané jsou rutiny OS a podporné rutiny (V/V operace, alokace paměti, matematické funkce apod.). Během vykonávání programu musí být tyto prostředky k dispozici.
- **Virtuální strojový kód** - složený je úplně z takzvaných virtuálních instrukcí.. Hovoříme tomu technika tvorby přenositelného počítače.

1.6 Kontext kompilátora

Už jsme hovořili o tom, že koncepčně pracuje kompilátor v jednotlivých fázích, přičemž v každé z nich je transformovaný zdrojový program postupně z jedné reprezentace do následující..

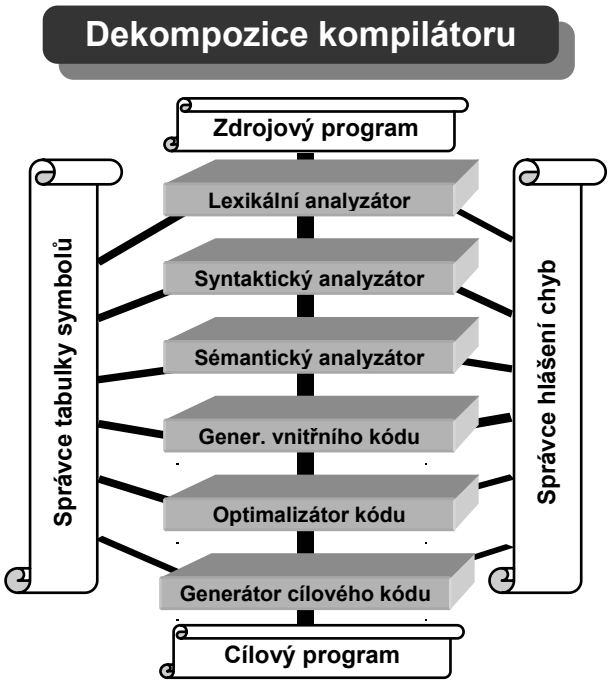
Na následujícím obrázku je uvedena dekompozice kompilátoru:

V praxi jsou některé fáze sloučeny. Například syntaktický a sémantický analyzátor se často považují za jeden komponent kompilátoru. Vnitřní reprezentace se v mnohých implementacích dokonce nevyskytuje. Optimalizátor kódu taktéž není nevyhnutný, ale dnes je velmi důležitou součástí kompilátoru.

Dvě aktivity:

- **Správce tabulky symbolů (Symbol Table Manager)** a
- **Správce hlášení chyb (Error Reporting)**

jsou přepojené se všemi fázemi. Přijímají a udržují informace o entitách programu během celého procesu kompilace.



1.6.1 Řízení tabulky symbolů

Důležitou součástí kompilátora je záznam identifikátorů použitých ve zdrojovém programu a sběr informací o různých atributech o každém z nich. Jako si později uvedeme, všechny identifikátory v programu mají ve vnitřním kódu přirazený jeden společný kód - token. Z důvodu rozlišitelnosti různých identifikátorů musíme po výskyte každého identifikátoru ve zdrojovém programu vést o něm potřebné informace, aby byla zabezpečena jednoznačnost. Atributy můžou obsahovat a poskytovat informace například

- o paměti nutné pro její alokaci,
- o druhu identifikátoru a typu odpovídající hodnoty,
- o počtu a typu argumentů procedury, funkce,
- o propojení na jiné identifikátory, a podobně...

Tabulka symbolů (Symbol Table) je datová struktura obsahující záznam o každém identifikátoru programu s položkami pro jeho atributy. Umožňuje rychle nalézt potřebné informace a doplnit je, případně využít v dalším průběhu kompilace.

Jestli je v programu lexikálním analyzátorem detekovaný nový identifikátor, potom je vložený do tabulky s minimální počáteční informací, protože během lexikální analýzy není možné určit další informace.. Atributy jsou do tabulky postupně přidávané během dalších fází kompilace.. Během generování vnitřního a cílového kódu jsou tyto informace využívané například pro alokaci proměnné, anebo konstanty do paměti..

1.6.2 Detekce chyb a správy o chybách

Po vzniku libovolné chyby během kompilace, či už lexikální, syntaktické anebo sémantické je potřebné vydat o tom správu a je třeba zajistit určité činnosti související s příslušnou chybou. Například:

- V první řade je třeba zjistit o jakou chybu jde a oznámit druh chyby.
- Je samozřejmostí podat správu o místě, kde byla chyba zjištěna.
- Důležité je zajistit v kompilátoru způsob pokračování činnosti kompilátoru po vydání příslušné správy.

Existuje více druhů chyb, které ovlivňují způsob další činnosti kompilátoru. Krom samotné chyby jako takové kompilátor může rozeznávat i váhu chyby a rozhodne se například nepokračovat v kompilaci (tzv. fatální chyba v Borland Pascale - `Error`), anebo na chybu pouze upozornit a pokračovat v kompilaci (tzv. varovná chyba - `Warning`). Možno tu vysledovat i dva způsoby chování kompilátoru po zjištění fatální chyby:

- zastavení překladu po fatální chybě - Borland Pascal,
- po oznámení pokračování v kompilaci a vyhledávání dalších chyb - Borland C++, Delphi, ...

První případ je triviální. V druhém případě je nutné aby se počítač „nezbláznil“ a nezačal vypisovat lavinovitě chyby, které v programu vlastně nejsou, ale vznikly tým, že kompilátor je po výskytu dezorientovaný a tím může považovat část textu omylem za chybný. V kompilátoru se to řeší postupem. který nazýváme **zotavení - Recovery**.

1.7 Obsah: