Integrated Development Environment

textový editor, debugger a kompilátor

Adamés Kleopatón z Trolu

12. 3. 2025

GEVO

Obsah

Co je IDE

Pod zkratkou **IDE** (Integrated Development Environment) obvykle rozumíme aplikaci, která obsahuje **textový editor**, **debugger** a **kompilátor** s podporou aspoň jednoho programovacího jazyka.

Co je IDE

Pod zkratkou **IDE** (Integrated Development Environment) obvykle rozumíme aplikaci, která obsahuje **textový editor**, **debugger** a **kompilátor** s podporou aspoň jednoho programovacího jazyka.

Smyslem **IDE** je existence aplikace, kterou programátor během své práce na daném projektu nemusí vůbec opustit. Proto **IDE** často obsahují navíc též funkce jako

- správu Git repositářů a integraci s Git cloud platformami (GitHub, AWS, Azure Repos, ...),

Co je IDE

Pod zkratkou **IDE** (Integrated Development Environment) obvykle rozumíme aplikaci, která obsahuje **textový editor**, **debugger** a **kompilátor** s podporou aspoň jednoho programovacího jazyka.

Smyslem **IDE** je existence aplikace, kterou programátor během své práce na daném projektu nemusí vůbec opustit. Proto **IDE** často obsahují navíc též funkce jako

- správu Git repositářů a integraci s Git cloud platformami (GitHub, AWS, Azure Repos, ...),
- search engine a vbudovaný prohlížeč,

Co je IDE

Pod zkratkou **IDE** (Integrated Development Environment) obvykle rozumíme aplikaci, která obsahuje **textový editor**, **debugger** a **kompilátor** s podporou aspoň jednoho programovacího jazyka.

Smyslem **IDE** je existence aplikace, kterou programátor během své práce na daném projektu nemusí vůbec opustit. Proto **IDE** často obsahují navíc též funkce jako

- správu Git repositářů a integraci s Git cloud platformami (GitHub, AWS, Azure Repos, ...),
- search engine a vbudovaný prohlížeč,
- real-time collaboration tools (Live Share, Code Sandbox, ...),

Co je IDE

Pod zkratkou **IDE** (Integrated Development Environment) obvykle rozumíme aplikaci, která obsahuje **textový editor**, **debugger** a **kompilátor** s podporou aspoň jednoho programovacího jazyka.

Smyslem **IDE** je existence aplikace, kterou programátor během své práce na daném projektu nemusí vůbec opustit. Proto **IDE** často obsahují navíc též funkce jako

- správu Git repositářů a integraci s Git cloud platformami (GitHub, AWS, Azure Repos, ...),
- search engine a vbudovaný prohlížeč,
- real-time collaboration tools (Live Share, Code Sandbox, ...),
- okno s AI chatbotem,

Co je IDE

Pod zkratkou **IDE** (Integrated Development Environment) obvykle rozumíme aplikaci, která obsahuje **textový editor**, **debugger** a **kompilátor** s podporou aspoň jednoho programovacího jazyka.

Smyslem **IDE** je existence aplikace, kterou programátor během své práce na daném projektu nemusí vůbec opustit. Proto **IDE** často obsahují navíc též funkce jako

- správu Git repositářů a integraci s Git cloud platformami (GitHub, AWS, Azure Repos, ...),
- search engine a vbudovaný prohlížeč,
- real-time collaboration tools (Live Share, Code Sandbox, ...),
- okno s AI chatbotem,
- ...

Obsah

Co to je textový editor

Program na úpravu **prostého textu**. Obvykle bývá aspoň jeden textový editor součástí operačního systému.

Co to je textový editor

Program na úpravu **prostého textu**. Obvykle bývá aspoň jeden textový editor součástí operačního systému.

Prostý text vs. formátováný text:

- **Prostý text** sestává z pouhé representace znaků. Každý znak je representován buď posloupností dané délky (nejčastěji 1, 2 nebo 4 B) nebo délky proměnlivé, podle daného kódování (ASCII, UTF-8, ISO/IEC, ...).

Co to je textový editor

Program na úpravu **prostého textu**. Obvykle bývá aspoň jeden textový editor součástí operačního systému.

Prostý text vs. formátováný text:

- **Prostý text** sestává z pouhé representace znaků. Každý znak je representován buď posloupností dané délky (nejčastěji 1, 2 nebo 4 B) nebo délky proměnlivé, podle daného kódování (ASCII, UTF-8, ISO/IEC, ...).
- Oproti tomu **formátovaný text** obsahuje navíc metadata, formátová data (font a velikost písma, tučnost a styl celkově), data o odstavcích (odsazení, zarovnání, rozmístění slov a písmen a výšku řádku) a data o stránce (velikost a okraje).

Funkce textového editoru

Základní

- **najít a nahradit** (kontrolované nahrazení částí textu podle zadaného stringu či regulárního výrazu),

Funkce textového editoru

Základní

- **najít a nahradit** (kontrolované nahrazení částí textu podle zadaného stringu či regulárního výrazu),
- **vyjmout, kopírovat a vložit**,

Funkce textového editoru

Základní

- **najít a nahradit** (kontrolované nahrazení částí textu podle zadaného stringu či regulárního výrazu),
- **vyjmout, kopírovat a vložit**,
- čtení prostého textu s UTF-8 kódováním,

Funkce textového editoru

Základní

- **najít a nahradit** (kontrolované nahrazení částí textu podle zadaného stringu či regulárního výrazu),
- **vyjmout, kopírovat a vložit**,
- čtení prostého textu s UTF-8 kódováním,
- **základní formátování textu** (obtékání, automatické odsazení),

Funkce textového editoru

Základní

- **najít a nahradit** (kontrolované nahrazení částí textu podle zadaného stringu či regulárního výrazu),
- **vyjmout, kopírovat a vložit**,
- čtení prostého textu s UTF-8 kódováním,
- **základní formátování textu** (obtékání, automatické odsazení),
- **undo a redo** (historie změn),

Funkce textového editoru

Základní

- **najít a nahradit** (kontrolované nahrazení částí textu podle zadaného stringu či regulárního výrazu),
- **vyjmout, kopírovat a vložit**,
- čtení prostého textu s UTF-8 kódováním,
- **základní formátování textu** (obtékání, automatické odsazení),
- **undo a redo** (historie změn),
- efektivní posun v dokumentu (skok na řádek, na konec stránky, na konec dokumentu atd.).

Funkce textového editoru

Pokročilé

- **profily** (nastavení pro konkrétní soubory či typy souborů),

Funkce textového editoru

Pokročilé

- **profily** (nastavení pro konkrétní soubory či typy souborů),
- úprava více souborů najednou,

Funkce textového editoru

Pokročilé

- **profily** (nastavení pro konkrétní soubory či typy souborů),
- úprava více souborů najednou,
- úprava po sloupcích,

Funkce textového editoru

Pokročilé

- **profily** (nastavení pro konkrétní soubory či typy souborů),
- úprava více souborů najednou,
- úprava po sloupcích,
- **folding** (skrytí části souboru podle rozsahu řádků či syntaktického prvku),

Funkce textového editoru

Pokročilé

- **profily** (nastavení pro konkrétní soubory či typy souborů),
- úprava více souborů najednou,
- úprava po sloupcích,
- **folding** (skrytí části souboru podle rozsahu řádků či syntaktického prvku),
- **transformace dat** (sloučení obsahu jiného textového souboru do upravovaného),

Funkce textového editoru

Pokročilé

- **profily** (nastavení pro konkrétní soubory či typy souborů),
- úprava více souborů najednou,
- úprava po sloupcích,
- **folding** (skrytí části souboru podle rozsahu řádků či syntaktického prvku),
- **transformace dat** (sloučení obsahu jiného textového souboru do upravovaného),
- **zvýraznění syntaxe** (zvýraznění – barvou, tučností atd. – částí textu podle kontextu závislém na typu upravovaného souboru),

Funkce textového editoru

Pokročilé

- **profily** (nastavení pro konkrétní soubory či typy souborů),
- úprava více souborů najednou,
- úprava po sloupcích,
- **folding** (skrytí části souboru podle rozsahu řádků či syntaktického prvku),
- **transformace dat** (sloučení obsahu jiného textového souboru do upravovaného),
- **zvýraznění syntaxe** (zvýraznění – barvou, tučností atd. – částí textu podle kontextu závislém na typu upravovaného souboru),
- **rozšíření** (textové editory pro programátory jsou modulární – umějí integrovat nové funkce podle potřeby a jsou programovatelné),

Funkce textového editoru

Pokročilé

- **profily** (nastavení pro konkrétní soubory či typy souborů),
- úprava více souborů najednou,
- úprava po sloupcích,
- **folding** (skrytí části souboru podle rozsahu řádků či syntaktického prvku),
- **transformace dat** (sloučení obsahu jiného textového souboru do upravovaného),
- **zvýraznění syntaxe** (zvýraznění – barvou, tučností atd. – částí textu podle kontextu závislém na typu upravovaného souboru),
- **rozšíření** (textové editory pro programátory jsou modulární – umějí integrovat nové funkce podle potřeby a jsou programovatelné),
- **příkazový řádek** (speciální okno pro spouštění pokročilých funkcí editoru).

Obsah

Co to je a co umí debugger

Program, který **spouští a analyzuje** jiné programy.

Co to je a co umí debugger

Program, který **spouští a analyzuje** jiné programy.

Funkce debuggeru

- **zastavení běhu** programu na zvoleném příkazu,

Co to je a co umí debugger

Program, který **spouští a analyzuje** jiné programy.

Funkce debuggeru

- **zastavení běhu** programu na zvoleném příkazu,
- **krokování** programu příkaz po příkazu,

Co to je a co umí debugger

Program, který **spouští a analyzuje** jiné programy.

Funkce debuggeru

- **zastavení běhu** programu na zvoleném příkazu,
- **krokování** programu příkaz po příkazu,
- **zobrazení či úprava** obsahu paměti, CPU registrů a zásobníku volání (zásobník se zavolanými funkcemi a jejich parametry)

Jak funguje debugger

Debug symbols

Soubory obsahující **samostatně spustitelné kusy strojového kódu** spolu s odkazem na původní soubor a číslem řádku, jež representují.

Jak funguje debugger

Breakpointy

Něco jako „záložky v knize“.

Jak funguje debugger

Breakpointy

Něco jako „záložky v knize“.

Proces nastavení breakpointu:

1. Zvolíme číslo řádku v příslušném debuggeru.

Jak funguje debugger

Breakpointy

Něco jako „záložky v knize“.

Proces nastavení breakpointu:

1. Zvolíme číslo řádku v příslušném debuggeru.
2. Debugger si zapamatuje pozici breakpointu jako číslo řádku a umístění souboru s příslušným kódem.

Jak funguje debugger

Breakpointy

Něco jako „záložky v knize“.

Proces nastavení breakpointu:

1. Zvolíme číslo řádku v příslušném debuggeru.
2. Debugger si zapamatuje pozici breakpointu jako číslo řádku a umístění souboru s příslušným kódem.
3. Debugger vloží do paměti za poslední instrukci kódu tzv. **breakpoint trap**, speciální CPU instrukci, která zastaví běh programu.

Jak funguje debugger

Breakpointy

Něco jako „záložky v knize“.

Proces nastavení breakpointu:

1. Zvolíme číslo řádku v příslušném debuggeru.
2. Debugger si zapamatuje pozici breakpointu jako číslo řádku a umístění souboru s příslušným kódem.
3. Debugger vloží do paměti za poslední instrukci kódu tzv. **breakpoint trap**, speciální CPU instrukci, která zastaví běh programu.
4. Tato speciální instrukce ukončí proces běžícího programu přes interrupt signal.

Jak funguje debugger

Breakpointy

Něco jako „záložky v knize“.

Proces nastavení breakpointu:

1. Zvolíme číslo řádku v příslušném debuggeru.
2. Debugger si zapamatuje pozici breakpointu jako číslo řádku a umístění souboru s příslušným kódem.
3. Debugger vloží do paměti za poslední instrukci kódu tzv. **breakpoint trap**, speciální CPU instrukci, která zastaví běh programu.
4. Tato speciální instrukce ukončí proces běžícího programu přes interrupt signal.
5. V moment interruptu předá operační systém kontrolu debuggeru, který je navržen, aby na tento signál reagoval zastavením programu (zkrátka nespustí další soubor v pořadí).

Jak funguje debugger

Step Into/Out/Over

Jak funguje krokování:

1. Debugger má uložený tzv. **instruction pointer**, ukazatel na další instrukci programu, která se má vykonat.

Jak funguje debugger

Step Into/Out/Over

Jak funguje krokování:

1. Debugger má uložený tzv. **instruction pointer**, ukazatel na další instrukci programu, která se má vykonat.
2. Když program krokujeme, debugger postupně vykonává instrukce pod svým IP.

Jak funguje debugger

Step Into/Out/Over

Jak funguje krokování:

1. Debugger má uložený tzv. **instruction pointer**, ukazatel na další instrukci programu, která se má vykonat.
2. Když program krokujeme, debugger postupně vykonává instrukce pod svým IP.
3. Po vykonání každé instrukce debugger přečte (přes OS) hodnoty všech proměnných definovaných v programu, jejichž adresy má uloženy v tabulce debug symbolů. Tímto umožní i zápis do proměnných.

Jak funguje debugger

Vyhodnocení výrazů za běhu

Jak to funguje:

1. Debugger zkompiluje zadaný výraz.

Jak funguje debugger

Vyhodnocení výrazů za běhu

Jak to funguje:

1. Debugger zkompiluje zadaný výraz.
2. Dosadí za proměnné a funkce adresy uložené v tabulce debug symbolů.

Jak funguje debugger

Vyhodnocení výrazů za běhu

Jak to funguje:

1. Debugger zkompiluje zadaný výraz.
2. Dosadí za proměnné a funkce adresy uložené v tabulce debug symbolů.
3. Program spustí a vytiskne výsledek (závislý na typu výrazu – obvykle před výraz zkrátka vloží `return`).

Jak funguje debugger

Vyhodnocení výrazů za běhu

Jak to funguje:

1. Debugger zkompiluje zadaný výraz.
2. Dosadí za proměnné a funkce adresy uložené v tabulce debug symbolů.
3. Program spustí a vytiskne výsledek (závislý na typu výrazu – obvykle před výraz zkrátka vloží `return`).

Tahle funkce též umožňuje existenci **podmínečných breakpointů**.

Obsah

Co to je a co umí kompilátor

Program na **překlad kódu** z jednoho programovacího jazyka do jiného. Většinou překládá programy v high-level jazycích do low-level jazyků, v nichž napsané instrukce už umí OS přímo vykonat.

Co to je a co umí kompilátor

Program na **překlad kódu** z jednoho programovacího jazyka do jiného. Většinou překládá programy v high-level jazycích do low-level jazyků, v nichž napsané instrukce už umí OS přímo vykonat.

Primárně rozlišujeme **single-pass** a **multi-pass** kompilátory.

Co to je a co umí kompilátor

Program na **překlad kódu** z jednoho programovacího jazyka do jiného. Většinou překládá programy v high-level jazycích do low-level jazyků, v nichž napsané instrukce už umí OS přímo vykonat.

Primárně rozlišujeme **single-pass** a **multi-pass** kompilátory.

Kompilace programu obyčejně probíhá ve třech stádiích: **front end**, **middle end** a **back end**.

Single-pass vs. multi-pass

Single-pass kompilátory

- jsou rychlejší,
- vyžadují méně paměti,
- jsou snadno debugovatelné a jejich efektivita snadno měřitelná,

Single-pass vs. multi-pass

Single-pass kompilátory

- jsou rychlejší,
- vyžadují méně paměti,
- jsou snadno debugovatelné a jejich efektivita snadno měřitelná,

ale zato

- nepodporují globální proměnné a lokální proměnné musejí být předem definovány;
- vyžadují striktní pořadí funkcí a importů,
- nepodporují dynamické typování,
- obecně vyrábějí méně optimalisovaný kód.

Stádia kompilace

Front end

Kompilátor zkontroluje **syntaxi** a **typy proměnných** (ve staticky typovaných jazycích). Přepíše program do tzv. **mezijazyka** (intermediate representation).

Stádia kompilace

Front end

Kompilátor zkontroluje **syntaxi** a **typy proměnných** (ve staticky typovaných jazycích). Přepíše program do tzv. **mezijazyka** (intermediate representation).

Postupuje ve (aspoň) třech krocích:

1. Lexikální analýza (‚lexing‘): **kategorisace symbolů** na proměnné, klíčová slova, funkce, hodnoty, pomocné symboly atd.

Stádia kompilace

Front end

Kompilátor zkontroluje **syntaxi** a **typy proměnných** (ve staticky typovaných jazycích). Přepíše program do tzv. **mezijazyka** (intermediate representation).

Postupuje ve (aspoň) třech krocích:

1. Lexikální analýza (‚lexing‘): **kategorisace symbolů** na proměnné, klíčová slova, funkce, hodnoty, pomocné symboly atd.
2. Syntaktická analýza (‚parsing‘): tvorba **derivačního stromu** (parse tree), se kterým kompilátor pracuje v dalších stádiích.

Stádia kompilace

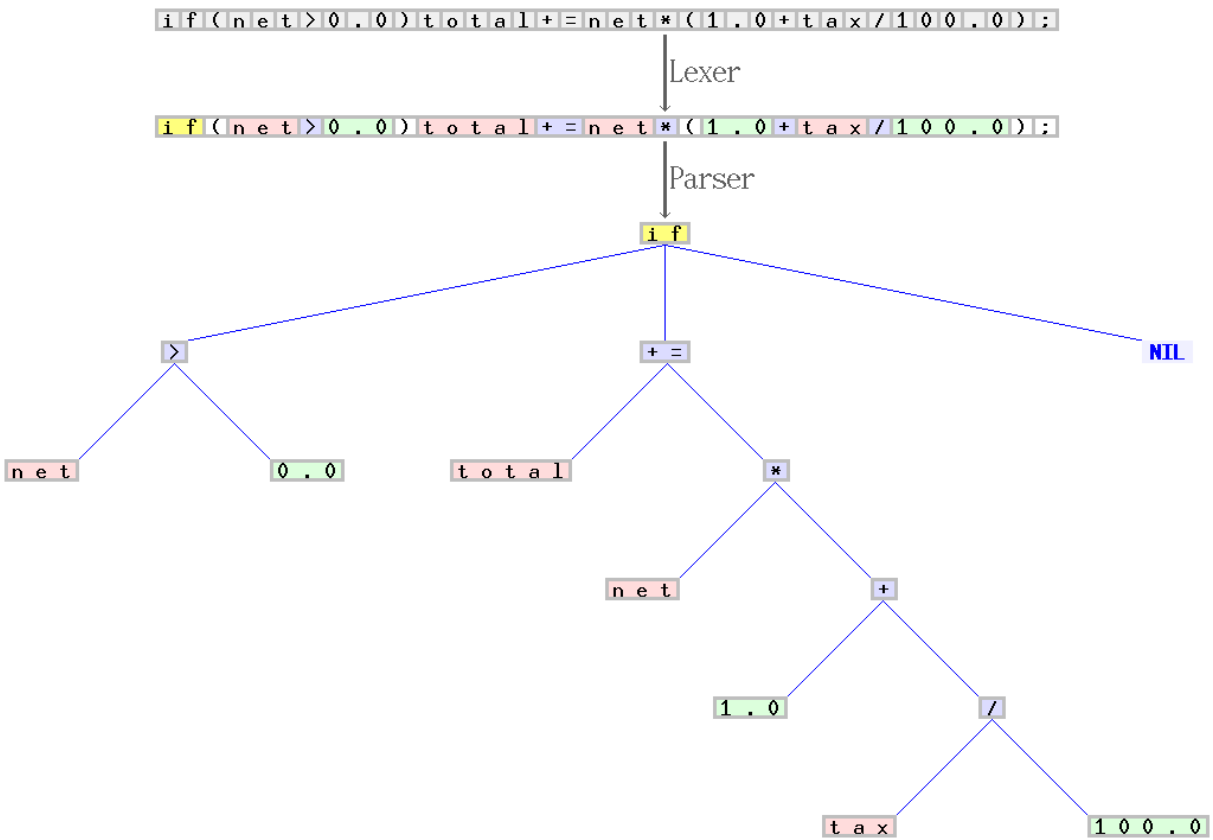
Front end

Kompilátor zkontroluje **syntaxi** a **typy proměnných** (ve staticky typovaných jazycích). Přepíše program do tzv. **mezijazyka** (intermediate representation).

Postupuje ve (aspoň) třech krocích:

1. Lexikální analýza (‚lexing‘): **kategorisace symbolů** na proměnné, klíčová slova, funkce, hodnoty, pomocné symboly atd.
2. Syntaktická analýza (‚parsing‘): tvorba **derivačního stromu** (parse tree), se kterým kompilátor pracuje v dalších stádiích.
3. Semantická analýza: kontrola typů, deklarace proměnné před použitím, existence návratové hodnoty v každé větvi funkce atd.

Stádia kompilace



Stádia kompilace

Middle end

Též **optimalisace**. Kompilátor přetvoří derivační strom do časově i prostorově více efektivní podoby.

Stádia kompilace

Middle end

Též **optimalisace**. Kompilátor přetvoří derivační strom do časově i prostorově více efektivní podoby.

Postupuje ve (aspoň) dvou krocích:

1. **Analýza**: kompilátor vyrobí **control-flow graph** (vlastně flowchart) průběhu každé funkce a též **call graph** celého programu. Ty se použijí k

Stádia kompilace

Middle end

Též **optimalisace**. Kompilátor přetvoří derivační strom do časově i prostorově více efektivní podoby.

Postupuje ve (aspoň) dvou krocích:

1. **Analýza**: kompilátor vyrobí **control-flow graph** (vlastně flowchart) průběhu každé funkce a též **call graph** celého programu. Ty se použijí k
   - sestrojení **use-define** řetězců (vlastně řetězce deklarace proměnné a jejích použití),

Stádia kompilace

Middle end

Též **optimalisace**. Kompilátor přetvoří derivační strom do časově i prostorově více efektivní podoby.

Postupuje ve (aspoň) dvou krocích:

1. **Analýza**: kompilátor vyrobí **control-flow graph** (vlastně flowchart) průběhu každé funkce a též **call graph** celého programu. Ty se použijí k
   - sestrojení **use-define** řetězců (vlastně řetězce deklarace proměnné a jejích použití),
   - analýze **závislostí** (kus kódu Y závisí na kusu kódu X, když se X musí spustit před Y),

Stádia kompilace

Middle end

Též **optimalisace**. Kompilátor přetvoří derivační strom do časově i prostorově více efektivní podoby.

Postupuje ve (aspoň) dvou krocích:

1. **Analýza**: kompilátor vyrobí **control-flow graph** (vlastně flowchart) průběhu každé funkce a též **call graph** celého programu. Ty se použijí k
   - sestrojení **use-define** řetězců (vlastně řetězce deklarace proměnné a jejích použití),
   - analýze **závislostí** (kus kódu Y závisí na kusu kódu X, když se X musí spustit před Y),
   - analýze **ukazatelů** (vytvoření tabulky adres, na které proměnné v programu ukazují),

Stádia kompilace

Middle end

Též **optimalisace**. Kompilátor přetvoří derivační strom do časově i prostorově více efektivní podoby.

Postupuje ve (aspoň) dvou krocích:

1. **Analýza**: kompilátor vyrobí **control-flow graph** (vlastně flowchart) průběhu každé funkce a též **call graph** celého programu. Ty se použijí k
   - sestrojení **use-define** řetězců (vlastně řetězce deklarace proměnné a jejích použití),
   - analýze **závislostí** (kus kódu Y závisí na kusu kódu X, když se X musí spustit před Y),
   - analýze **ukazatelů** (vytvoření tabulky adres, na které proměnné v programu ukazují),
   - …

Stádia kompilace

Middle end

2. **Optimalisace**: na základě předchozí analýzy převede kompilátor derivační strom do funkčně ekvivalentní a více efektivní podoby. Mezi optimalisační techniky patří
   - **inline expansion**: nahrazení volání funkce jejím obsahem,

Stádia kompilace

Middle end

2. **Optimalisace**: na základě předchozí analýzy převede kompilátor derivační strom do funkčně ekvivalentní a více efektivní podoby. Mezi optimalisační techniky patří
   - **inline expansion**: nahrazení volání funkce jejím obsahem,
   - **dead-code elimination**: odstranění nedosažitelného kódu (pomocí CFG),

Stádia kompilace

Middle end

2. **Optimalisace**: na základě předchozí analýzy převede kompilátor derivační strom do funkčně ekvivalentní a více efektivní podoby. Mezi optimalisační techniky patří
   - **inline expansion**: nahrazení volání funkce jejím obsahem,
   - **dead-code elimination**: odstranění nedosažitelného kódu (pomocí CFG),
   - **constant folding**: výpočet konstant nezávislých na předchozím kódu,

Stádia kompilace

Middle end

2. **Optimalisace**: na základě předchozí analýzy převede kompilátor derivační strom do funkčně ekvivalentní a více efektivní podoby. Mezi optimalisační techniky patří
   - **inline expansion**: nahrazení volání funkce jejím obsahem,
   - **dead-code elimination**: odstranění nedosažitelného kódu (pomocí CFG),
   - **constant folding**: výpočet konstant nezávislých na předchozím kódu,
   - **transformace cyklů**: metody zrychlení cyklů (např. *distribuce* a *sjednocení*, *permutace*, ...),

Stádia kompilace

Middle end

2. **Optimalisace**: na základě předchozí analýzy převede kompilátor derivační strom do funkčně ekvivalentní a více efektivní podoby. Mezi optimalisační techniky patří
   - **inline expansion**: nahrazení volání funkce jejím obsahem,
   - **dead-code elimination**: odstranění nedosažitelného kódu (pomocí CFG),
   - **constant folding**: výpočet konstant nezávislých na předchozím kódu,
   - **transformace cyklů**: metody zrychlení cyklů (např. *distribuce* a *sjednocení*, *permutace*, ...),
   - **odstranění zbytečných proměnných**: proměnné ukazující ve stejný čas na stejnou adresu jsou sjednoceny v jednu,

Stádia kompilace

Middle end

2. **Optimalisace**: na základě předchozí analýzy převede kompilátor derivační strom do funkčně ekvivalentní a více efektivní podoby. Mezi optimalisační techniky patří
   - **inline expansion**: nahrazení volání funkce jejím obsahem,
   - **dead-code elimination**: odstranění nedosažitelného kódu (pomocí CFG),
   - **constant folding**: výpočet konstant nezávislých na předchozím kódu,
   - **transformace cyklů**: metody zrychlení cyklů (např. *distribuce* a *sjednocení*, *permutace*, ...),
   - **odstranění zbytečných proměnných**: proměnné ukazující ve stejný čas na stejnou adresu jsou sjednoceny v jednu,
   - **automatická paralelisace**: kusy kódu, které mohou být provedeny souběžně se rozdělí mezi procesorová jádra (příkladem jsou podmínky, jejichž všechny větve se někdy vykonají, ale nezávisejí na sobě).

Stádia kompilace

Back end

Kompilátor přeloží optimalisovaný derivační strom do **CPU instrukcí** a vytvoří **strojový kód**.

Stádia kompilace

Back end

Kompilátor přeloží optimalisovaný derivační strom do **CPU instrukcí** a vytvoří **strojový kód**.

Probíhá ve (přinejmenším) dvou fázích:

1. Optimalisace **podle stroje**: optimalisace kódu závislá na specifických parametrech CPU, na němž má být program spouštěn. Příkladem je tzv. **peephole optimisation**, kdy kompilátor nahrazuje sady obecných CPU instrukcí za efektivnější sady cílového CPU;

Stádia kompilace

Back end

Kompilátor přeloží optimalisovaný derivační strom do **CPU instrukcí** a vytvoří **strojový kód**.

Probíhá ve (přinejmenším) dvou fázích:

1. Optimalisace **podle stroje**: optimalisace kódu závislá na specifických parametrech CPU, na němž má být program spouštěn. Příkladem je tzv. **peephole optimisation**, kdy kompilátor nahrazuje sady obecných CPU instrukcí za efektivnější sady cílového CPU;
2. **Generování kódu**: optimalisovaný kód je přeložen do strojového jazyka cílového systému (třeba `C` pro Linux a `NASM` pro Windows). Tento proces zahrnuje například

Stádia kompilace

Back end

Kompilátor přeloží optimalisovaný derivační strom do **CPU instrukcí** a vytvoří **strojový kód**.

Probíhá ve (přinejmenším) dvou fázích:

1. Optimalisace **podle stroje**: optimalisace kódu závislá na specifických parametrech CPU, na němž má být program spouštěn. Příkladem je tzv. **peephole optimisation**, kdy kompilátor nahrazuje sady obecných CPU instrukcí za efektivnější sady cílového CPU;
2. **Generování kódu**: optimalisovaný kód je přeložen do strojového jazyka cílového systému (třeba `C` pro Linux a `NASM` pro Windows). Tento proces zahrnuje například
   - rozhodnutí, které proměnné uložit v paměti a které v registrech,

Stádia kompilace

Back end

Kompilátor přeloží optimalisovaný derivační strom do **CPU instrukcí** a vytvoří **strojový kód**.

Probíhá ve (přinejmenším) dvou fázích:

1. Optimalisace **podle stroje**: optimalisace kódu závislá na specifických parametrech CPU, na němž má být program spouštěn. Příkladem je tzv. **peephole optimisation**, kdy kompilátor nahrazuje sady obecných CPU instrukcí za efektivnější sady cílového CPU;
2. **Generování kódu**: optimalisovaný kód je přeložen do strojového jazyka cílového systému (třeba `C` pro Linux a `NASM` pro Windows). Tento proces zahrnuje například
   - rozhodnutí, které proměnné uložit v paměti a které v registrech,
   - rozdělení výpočetní síly paralelně běžícím kusům kódu,

Stádia kompilace

Back end

Kompilátor přeloží optimalisovaný derivační strom do **CPU instrukcí** a vytvoří **strojový kód**.

Probíhá ve (přinejmenším) dvou fázích:

1. Optimalisace **podle stroje**: optimalisace kódu závislá na specifických parametrech CPU, na němž má být program spouštěn. Příkladem je tzv. **peephole optimisation**, kdy kompilátor nahrazuje sady obecných CPU instrukcí za efektivnější sady cílového CPU;
2. **Generování kódu**: optimalisovaný kód je přeložen do strojového jazyka cílového systému (třeba `C` pro Linux a `NASM` pro Windows). Tento proces zahrnuje například
   - rozhodnutí, které proměnné uložit v paměti a které v registrech,
   - rozdělení výpočetní síly paralelně běžícím kusům kódu,
   - generování debug symbolů a tabulek.