Лексичка анализа и њене примене

2026/2027

Курсеви превођења на Р смеру

- Курсеви
  - “Лексичка анализа и њене примене” и
  - “Компилација програмских језика”

  су део целине која се бави питањима теорије и праксе превођења програмских језика и обраде структурираних текстуалних докумената.

- Иако није формални, “Лексичка анализа” јесте суштински предуслов за полагање испита из “Компилације програмских језика”
- Изборни предмет “Конструкција компилатора” - наставак ова два курса

Расподела поена



Предуслови

- Програмирање 1
- Програмирање 2

За курс “Компилација програмских језика” додатни предуслов је:

- Објектно оријентисано програмирање

Литература

- Белешке са предавања и вежби
- Душко Витас, Преводиоци и интерпретатори, Математички факултет, 2006

За оне који желе да знају више:

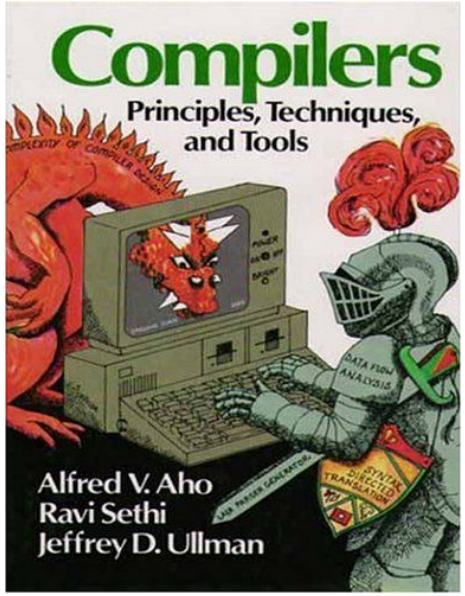
- Aho, Sethi, Ullman, Lam, Compilers: Principles, Techniques, and Tools, 2nd Edition (aka Dragon Book)
- Andrew Appel, Modern Compiler Implementation in C
- Један доста детаљан преглед литературе: https://github.com/aalhour/awesome-compilers

Теорија и пракса

- На вежбама се пре свега приказују технике решавања практичних задатака применом специјализованих алата
  - засебних попут grep/sed/lex/...,
  - уграђених у програмске језике опште намене попут python-a
- Предавања прилично дубоко покривају релевантне делове теорије формалних језика и аутомата
- Циљ предавања је да дају објашњење како алати који се на вежбама користе раде “изнутра”
- Предавања (осим првог и последњег) су суштински математички предмет (дефиниције, леме, теореме, докази, ...)
- Вежбе су суштински рачунарски предмет (осим рачунских задатака)

Компилатори су “софтверске звери”

- “GCC Compiler Is Up To 7.3 Million Lines Of Code (2012)”
- “GCC Soars Past 14.5 Million Lines Of Code (2015)”
- Практично нема напредне технике програмирања ни напредног алгоритма који се не користи у неком делу савременог компилатора
  - грамзиви алгоритми,
  - динамичко програмирање,
  - хеуристике,
  - пробабилистички алгоритми,
  - ...
- Озбиљан инжењерски подухват у развоју софтвера
- На (не)срећу, ми ћемо само додирнути један мали делић целе машинерије!



Приступи превођењу програмских језика

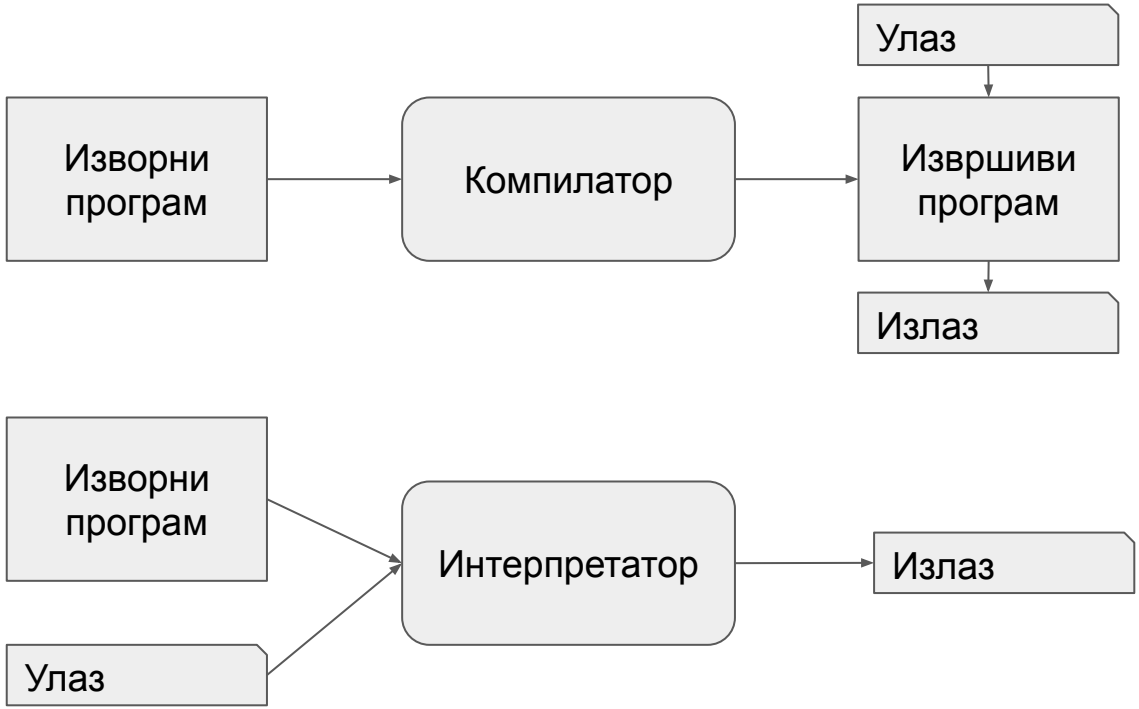
Појава виших програмских језика

- Рачунарство не би било могуће да нема виших програмских језика
  - комплексне системе је практично немогуће директно имплементирати у коду који рачунар може директно да извршава (машинском коду)
- Виши програмски језици и њихово аутоматско превођење тј. извршавање су настали у самом зачетку савременог рачунарства
  - Идеје: Конрад Цузе, виши програмски језик Планкалкул 1942-1945, Корадо Бем у дисертацији 1951, ...
  - Autocode, 1952, Алик Глени, први имплементирани компилатор
  - A-0, 1952, Грејс Хопер, први пут употребљен термин компилатор
  - Fortran, 1954-1957, Џон Бекус и тим у компанији IBM, први компилатор који је “заживео”
- Развој компилатора је у изузетно тесној вези са развојем програмских језика и њиховим дизајном

Компилатори и интерпетатори

- Традиционално се за превођење и извршавање програма користе **компилатори** и **интерпретатори**
- Компилатори
  - преводе програм са улазног језика (нпр. C, Java, ...) на циљни језик (најчешће асемблер тј. машински језик или језик неке виртуалне машине)
  - улаз је **изворни кôд** (енгл. *source code*), а излаз је **објектни кôд** (енгл. *object code*)
  - фаза превођења и фаза извршавања програма су раздвојене
  - једном преведен програм се може извршавати више пута, без присуства компилатора
  - постоје и компилатори са једног на други више програмски језик (енгл. *source to source*)
- Интерпретатори
  - читају наредбу по наредбу улазног језика (нпр. Python, језик неке виртуелне машине) и извршавају је
  - фаза превођења и извршавања програма су испреплетане
  - програм се не може извршавати без интерпретатора

Компилатори и интерпретатори



Када компилатор, а када интерпретатор?

- Извршавање компилираних програма је брже него интерпретираних
- Грешке се лакше откривају код компилираних програма - грешка у некој линији интерпретираног програма се открива тек када се та линија извршава
- Програми који се интерпретирају имају бржи развојни циклус
- Неки језици (нпр. C, C++, Pascal, ...) се традиционално компилирају, неки интерпретирају (нпр. Python, JavaScript, PHP, Ruby, ...), а за неке постоје и компилатори и интерпретатори (нпр. Haskell)
  - статички типизирани језици (типови познати у време превођења)
  - динамички типизирани језици (типови познати тек у време извршавања)
- У новије време подела није баш овако једноставна и често се користи нека комбинација компилације и интерпретације (нпр. JIT)

Компилација у ширем и ужем смислу

> Пут од изворног кода до програма који се успешно извршава подразумева неколико различити фаза (и различитих алата), које све заједно често подразумевамо под процесом компилације.

- претпроцесирање
- компилација (у ужем смислу)
- асемблирање
- повезивање
- пуњење

Надаље ћемо под термином компилација подразумевати само компилацију у ужем смислу.

Претпроцесирање (енгл. preprocessing)

- Неки језици користе претпроцесор који се извршава пре процеса компилације (мада се некада може сматрати и да је нулта фаза компилације)
  - Претпроцесорске директиве у језику C: `#include`, `#define`, ...
- Ако се користи gcc, резултат можемо видети навођењем опције `-E` (нпр. `gcc -E zdravo.c`)

```c
/* Program ispisuje "Zdravo, svete!" */
#include <stdio.h>
#define OK 0

int main() {
  printf("Zdravo, svete!");
  return OK;
}
```

```c
...
# 292 "/usr/include/stdio.h" 3 4
extern int printf (const char *__restrict __format, ...);
...
# 3 "zdravo.c" 2

# 5 "zdravo.c"
int main() {
  printf("Zdravo, svete!");
  return 0;
}
```

Асемблирање (енгл. assembling)

- Превођење програма на асемблерском кôду у објектни кôд на машинском језику.
- Ако се користи gcc, резултат компилације пре асемблирања се може видети навођењем опције `-S` (нпр. `gcc -S zdravo.c`)

```
        .file   "zdravo.c"
        .text
        .section        .rodata
.LC0:
        .string "Zdravo, svete!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

Повезивање (енгл. linking)

- Објектни кôд добијен компилацијом појединачних модула се коришћењем програма који се назива **повезивач** (енгл. linker) повезује са објектним кодом *статичких библиотека* чиме се добија јединствен **извршиви програм** (енгл. executable file)
- Ако се користи gcc, компилација без повезивања се остварује опцијом `-c` (нпр. `gcc -c zdravo.c`) и резултат је објектни кôд у `.o` датотеци
- Повезивање се постиже ако се gcc-у предају објектне датотеке:

```
gcc -c kod1.c
gcc -c kod2.c
gcc kod1.o kod2.o -o program
```

Грешке у повезивању

- Најчешће грешке су:
  - објекат (функција, глобална променљива, ...) коју један објектни модул очекује у неком другом модулу, не постоји ни у једном објектном модулу који се повезује
  - два различита објектна модула која се повезују садрже објекат (функцију, глобалну променљиву, ...) истог имена
- Нпр. наредни програм пролази компилацију, али се повезивач буни, јер не проналази дефиницију функције `print` ни у једном објектном модулу

```
#include <stdio.h>

void print(int x);

int main() {
   print(5);
   return 0;
}
```

```
/usr/bin/ld: /tmp/cc0XkDQH.o: in function `main':
tmp.c:(.text+0xe): undefined reference to `print'
collect2: error: ld returned 1 exit status
```

Пуњење (енгл. loading)

- Извршиви програм се пре извршавања пуни у меморију и повезује са *динамичким библиотекама* (обично смештеним у `.dll` или `.so` датотекама) које се учитавају током извршавања програма, коришћењем програма који се зове **пунилац** (енгл. loader)

Виртуелне машине (бајткод)

- Компилатори често генеришу кôд за неку виртуелну машину, који се приликом покретања програма интерпретира (или поново компилира)
  - Pascal се преводио на P-code (pascal code)
  - Java се преводи на JVM (java virtual machine)
  - C# се преводи на .NET CLR (common language runtime)
  - LLVM - генеричка виртуелна машина која се користи за разне језике
- Кôд за виртуелне машине се данас често назива **бајткод** (енгл. bytecode) (нарочито када је у питању Java, мада се термин користи и као генерички)

Зашто бајткод?

- Предности:
  - портабилност (исти програм на бајткоду се може лако покретати на потпуно различитим платформама)
  - једноставнија имплементација компилатора (бајткод је ближи вишем програмском језику, него реални асемблер)
- Недостаци
  - традиционално спорији приступ, нарочито ако се бајткод интерпретира
  - ипак, перформансе нису угрожене (могу се чак понекад добити и боље перформансе) када се бајткод компилира (уместо да се интерпретира)

Компилација бајткода (уместо интерпретације)

- JIT (енгл. Just In Time) компилација
  - сваки део бајткода (тело петље, тело функције итд.) се преводи у објектни кôд пре свог првог извршавања
  - добијени превод (објектни кôд) се затим више пута користи
  - познати детаљи конкретне машине (не само архитектура), па оптимизација може бити боља
  - *спекулативна оптимизација* (проба се, па ако су перформансе лоше, проба се другачије)
- AOT (енгл. Ahead Of Time) компилација
  - подразумева генерисање објектног кода за циљну машину пре извршавања програма
  - и класична компилација изворног директно на објектни кôд је AOT, међутим, термин AOT се чешће користи када се бајткод пре свог извршавања комплетно преведе на објектни кôд
  - познати детаљи конкретне машине (не само архитектура), па оптимизација може бити боља

Процес компилације

Пролази

- У зависности од тога колико пута се програм обрађује компилатори могу бити:
  - **једнопролазни**
  - **вишепролазни**
- Вишепролазност не подразумева обавезно да се више пута пролази кроз оригинални изворни кôд - у сваком пролазу се програм може трансформисати у репрезентацију која погодује наредном пролазу

Једнопролазни компилатори

- Једнопролазни компилатори током читања изворног кода одмах емитују објектни кôд
- Једнопролазни компилатори су једноставнији и бржи, али често генеришу неефикаснији објектни код него вишепролазни, јер неке оптимизације захтевају више пролаза
- Неки конструкти језика су уведени због захтева једнопролазности
  - нпр. декларације функција пре позива

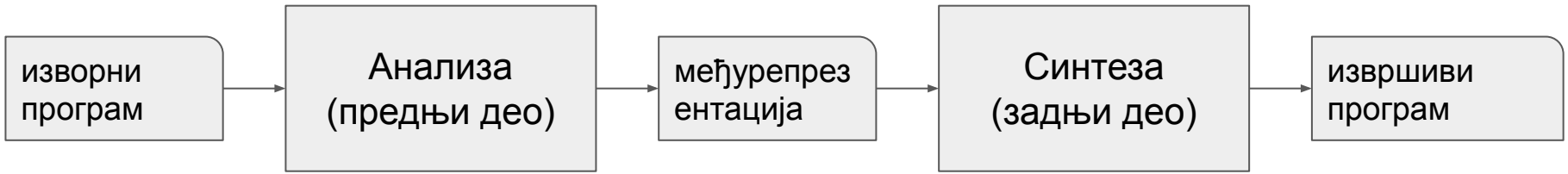
Вишепролазни компилатори

- Данас су компилатори углавном вишепролазни
- Ранији компилатори су били вишепролазни да би се превазишла хардверска ограничења (компилатор је био разбијен у више мањих, једноставнијих програма)
- Данас су компилатори вишепролазни да би се могли да изврше напредне оптимизације које захтевају више пролаза
- Неки програмски језици захтевају постојање више пролаза
  - нпр. у језику Java у класи се може позвати метода која је дефинисана у истој класи, али негде у коду иза места на ком је позвана
- Вишепролазни компилатори су модуларнији, па се једноставније имплементирају и једноставније се анализира и доказује њихова коректности

Анализа и синтеза

- Превођење (компилација) програма обухвата
  - **анализу** улазног програма, његово расчлањавање (каже се и парсирање) и представљање у неком апстрактном облику (нпр. у облику дрвета)
  - **синтезу** излазног програма (генерисање програма на циљном језику, најчешће асемблеру)
  - анализа и синтеза комуницирају преко **међујезика** који је по својим својствима негде између вишег програмског језика и асемблера
- Интерпетација програма у класичном смислу обухвата анализу, али нема синтезе

Анализа и синтеза



Шта ћемо ми проучавати?

- Нагласак у наша два курса је у великој мери на **анализи**
- Синтезу ћемо покрити само информативно (тек ради “опште културе”)
- Теорија и технике које се користе за анализу програма применљиве су на анализу било ког облика структурираног текста. Нпр.
  - анализа математичких израза у програмима за нумеричка и симболичка израчунавања
  - анализа текста обележеног језицима за обележавање (html, tex, markdown, ...)
  - анализа конфигурационих датотека (jason, yaml, ...)
  - анализа текста на природном језику
  - ...
- Не очекујте да ћете после положена оба курса моћи да направите комплетан ефикасан компилатор - на добром сте путу, али је потребно одслушати бар још и изборни курс “Конструкција компилатора” у коме се детаљније проучава синтеза (и оптимизација)

Предвање почиње у 8:25
http://etc.ch/LKSF

Фаза анализе

- Лексичка анализа
- Синтаксичка анализа
- Семантичка анализа
- Генерисање међукода

Назива се још и **предњи део** (енгл. **frontent**) компилатора

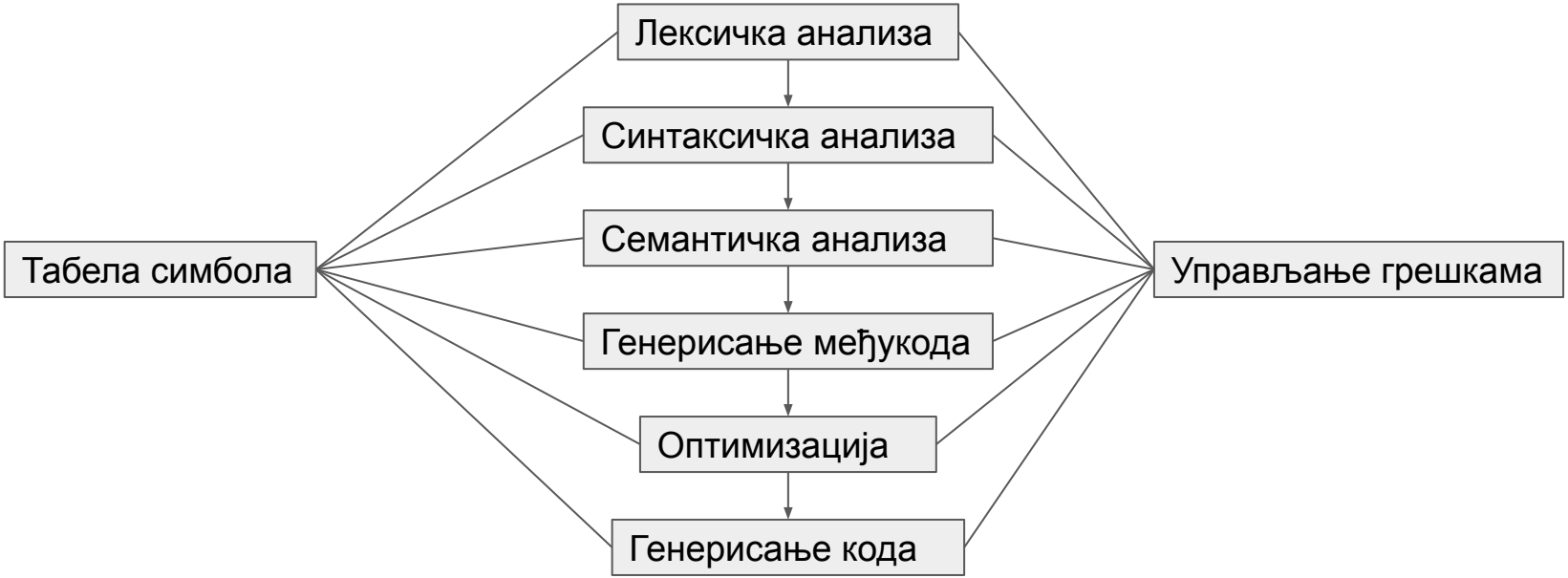
Фаза синтезе

- Оптимизација међукода (машински независна)
- Генерисање кода
  - регистарска алокација
  - одабир инструкција
  - распоређивање инструкција
  - оптимизација кода (машински зависна)

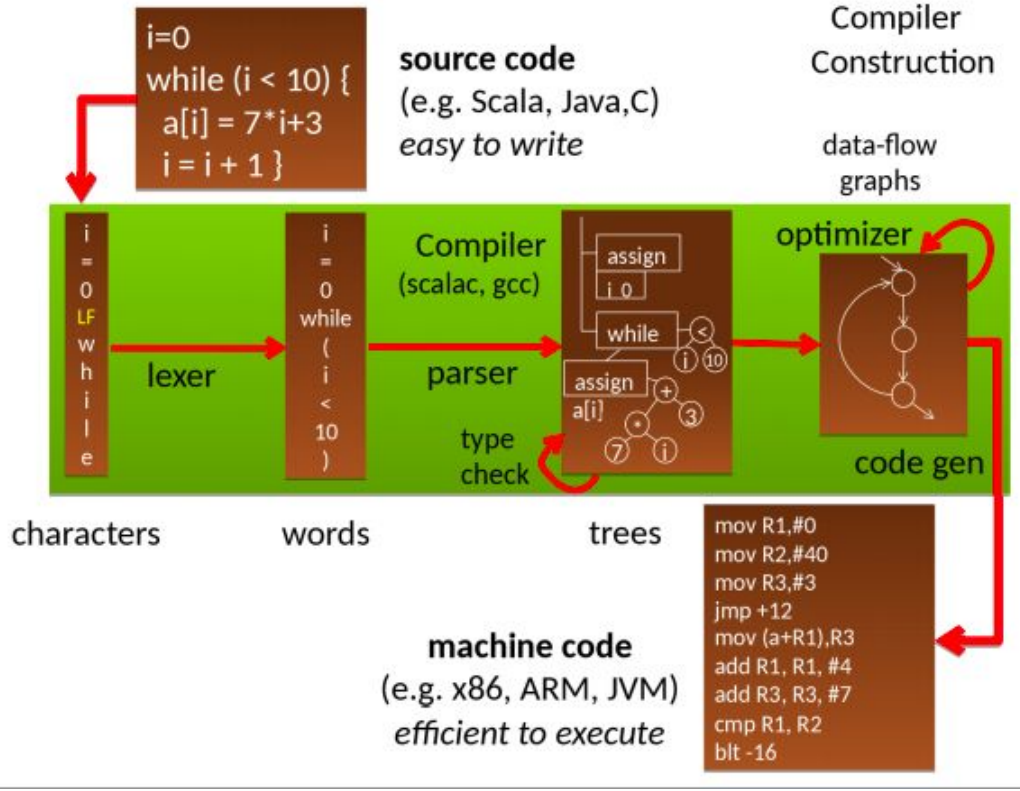
Назива се још и **задњи део** (енгл. **back end**) компилатора.

Оптимизација међукода се због некада назива и **средњи део** (енгл. **middle end**), јер често не зависи ни од детаља изворног, ни од детаља циљног језика

Дијаграм фаза



Фазе компилације

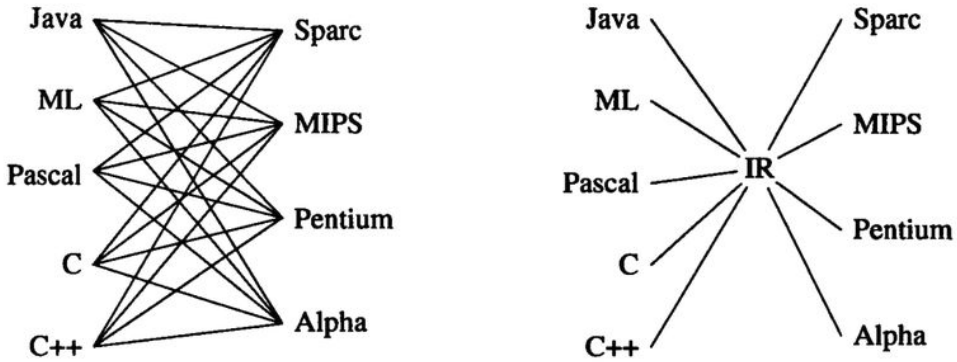


Зашто фазе?

- Компилација је веома сложен процес и једини начин да се савлада је да се подели на већи број јасно дефинисаних, једноставнијих потпроцеса
- Фазе нису обавезно исто што и пролази
  - један пролаз може бити подељен на више фаза (нпр. лексичка и синтаксичка анализа се скоро увек извршавају у једном пролазу),
  - у неким фазама може извршити више пролаза (нпр. оптимизација је често вишепролазна)
- Неке фазе се заиста извршавају временски једна након друге (вишепролазно), док су некада фазе испреплетане у времену и извршавају се наизменично (једнопролазно)
- Фазе често сукцесивно трансформишу програм - излаз једне фазе је улаз у наредну
- Неке информације су глобалне и користе се у разним фазама (табеле симбола, поруке о грешкама, ...)

Зашто предњи и задњи део и међукод?

- Исти задњи део може да се употреби за компилацију више различитих програмских језика
- Исти предњи део може да се употреби за компилацију на више различитих платформи
- У пракси јако заступљено (.NET IL, LLVM, ...)



Лексичка анализа

Задатак лексичке анализе

- Током лексичке анализе улазна струја карактера (обично прочитаних из датотеке или неке ниске) се дели на мање јединице (**лексеме**), лексеме се категоришу и придружују им се обележја (**токени**) категорија којима припадају
- У аналогији са природним језиком лексичка анализа би одговарала подели реченице на речи и одређивању врсте сваке речи
- Поред токенизације, лексички анализатор може имати и друге задатке:
  - елиминација коментара (ако нема претпроцесора)
  - придруживање вредности неким лексемама
  - ...
- Модул који врши лексичку анализу назива се **лексички анализатор**, **лексер**, **токенизер**, **сканер**, ...

Табела симбола

- Током лексичке анализе у табелу симбола се уписују препознати идентификатори и придружују им се одређене релевантне информације
  - нпр. врста и колона у коду где је тај идентификатор пронађен
- Ова табела се користи, али и допуњава током наредних фаза компилације
  - нпр. уписују се информације о типовима

Пример лексичке анализе

```
double coeff = 1 + 2;
double niz1[10], niz2[10];
niz2[0] = coeff * x;
for (int i = 0; i < 9; i++) {
    double y = coeff * x;
    niz2[i+1] = niz1[i+1] + i*y;
}
```

`double`	KEYWORD
`coeff`	IDENTIFIER
`=`	OPERATOR
`1`	LITERAL
`+`	OPERATOR
`2`	LITERAL
`;`	SEPARATOR
...	...

Најчешће лексичке категорије

- идентификатори (`x`, `y1`, `NZD`, `prviSabirak`, `drugi_sabirak`, ...)
- кључне речи (`int`, `double`, `for`, `if`, `return`, `break`, `struct`, `auto`, ...)
- оператори (`+`, `-`, `*`, `/`, `%`, ...)
- литерали (`3`, `4.2`, `-3.5e12`, “`zdravo svima`”, ...)
- сепаратори (`(`, `)`, `{`, `}`, `;`, ...)
- коментари (`/* ovo je komentar */`)

Језик C има преко 100 токена (када се уброје преко 40 кључних речи, преко 50 оператора, ..)

Лексичке грешке

- Нису толико честе
- Примери:
  - char s[] = “abc”; error: stray ‘\342’ in program Unicode наводници
  - int a = 09; error: invalid digit "9" in octal constant
  - printf("Zdravo); error: unterminated string or character constant

Како се описују и препознају токени

- “Идентификатор се састоји од слова, цифара и подвлаке, али не може да почне цифром”
- `[a-z_][a-z0-9_]*` - **регуларни израз**
  - оператор `*` описује да се оно на шта је примењен понавља нула или више пута
  - “почиње словом или подвлаком, за чим иде нула или више слова, цифара или подвлака”
- Специјални алати у које програмер уноси регуларни изразе на основу којих се аутоматски генерише кôд лексичког анализатора који чита карактер по карактер улазне струје и препознаје лексеме и токене
- Централни део курса “Лексичка анализа” посвећен је управо овој теми

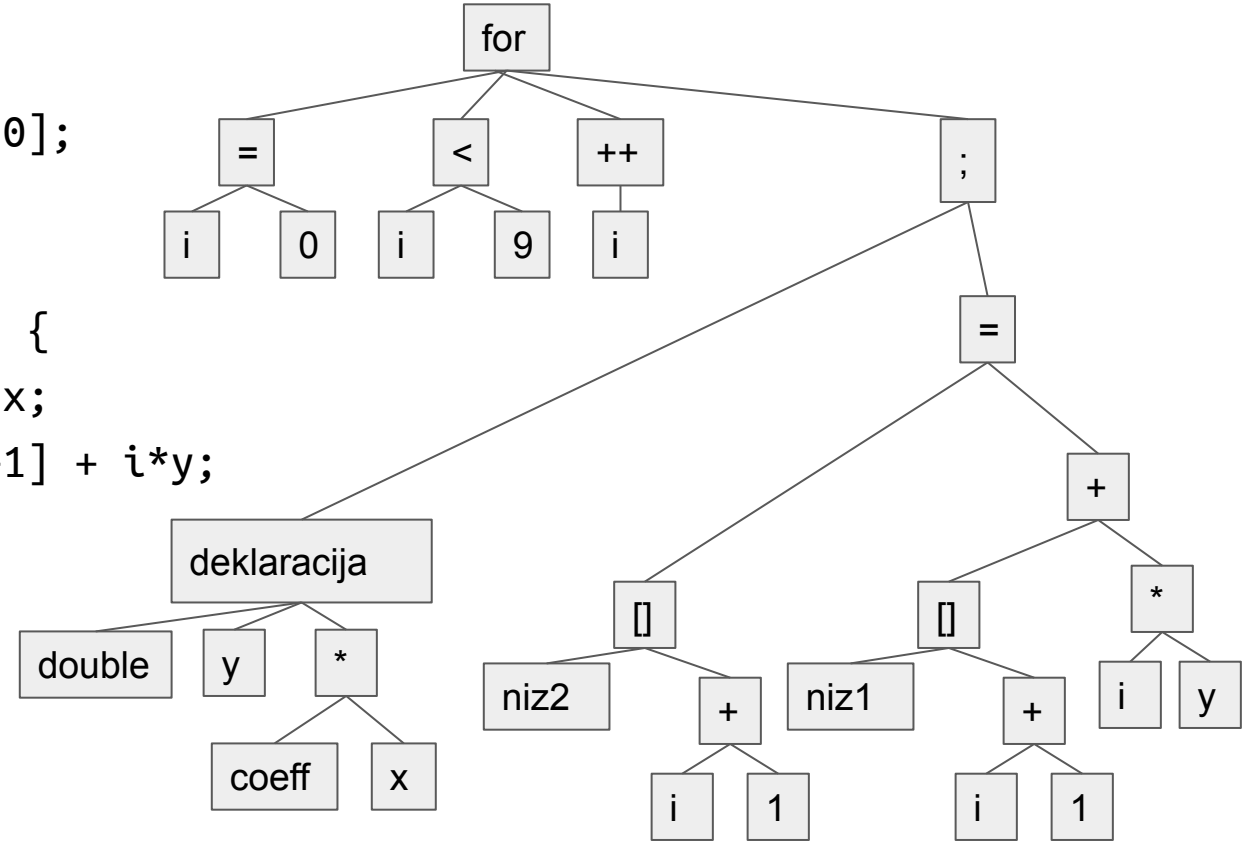
Синтаксичка анализа

Задатак синтаксичке анализе

- Синтаксички анализатор чита серију токена које је препознао лексички анализатор, проверава да ли су токени поређани у складу са **синтаксичким правилима** програмског језика и ако јесу, структуру програма приказује у виду **синтаксичког дрвета**
- `int x = 3 + 5;` синтаксички исправно
  `x = int + 3 5;` синтаксички неисправно
- Ово би одговарало провери да ли су речи у реченици сложене у складу са граматичким правилима језика и да се одреди граматичка структура реченице (субјекат, предикат, ...)
  - *“Синтаксичка анализатор бити не правилима исправно.”* - синтаксички неисправно
- Модул који ради синтаксичку анализу назива се **синтаксички анализатор** или **парсер**

Пример синтаксичке анализе

```
double coeff = 1 + 2;
double niz1[10], niz2[10];
niz2[0] = coeff * x;
int i;
for (i = 0; i < 9; i++) {
    double y = coeff * x;
    niz2[i+1] = niz1[i+1] + i*y;
}
```



Најчешће синтаксичке категорије

- Израз
  - `3+4*x`
  - `i < n`
  - `a == b && c + 3 < d`
  - `a < 3 ? “da” : “ne”`
  - `sin(x+y) * 4`
  - ...
- Наредба
  - x = 3;
  - i++
  - printf(“%d”, x);
  - if (a < 3) puts(“Zdravo”);
  - for (int i = 0; i < n; i++) scanf(“%d”, &a[i]);
  - while(--n) { i++; j--; }
  - ...

Најчешће синтаксичке категорије

- Декларација
  - int x;
  - int a = 5;
  - int NZD(int, int);
  - ...
- Дефиниција
  - int zbir(int a, int b) { return a + b; }
  - struct razlomak { int imenilac; int brojilac; }
  - ...
- ...

Синтаксичке грешке

- Велики број различитих грешака
  - parse error before `...'
  - syntax error
  - expected ... before ...
    - expected ‘)’ before ‘char’
    - expected ‘;’, ‘,’ or ‘)’ before ‘{’ token
    - expected ‘;’ before numeric constant
  - ...
- Разлози су различити:
  - заборављене отворене или затворене заграде
  - заборављен симбол ;
  - потпуно промашена синтакса

Како се описује синтакса?

- Формалне граматике
  - Бекусова нотација (енгл. Backus-Naur Form, BNF)
  - Проширена Бекусова нотација (енгл. Extended Backus-Naur Form, EBNF)
- Синтаксички дијаграми

- Програмер описује синтаксу коришћењем неког облика формалне граматике и на основу тога се аутоматски генерише кôд синтаксичког анализатора
- Ово је централна тема курса “Компилација програмских језика”

Бекусова нотација - пример

Пример описа целобројних литерала помоћу BNF:

```
<ceo broj> ::= <neoznacen ceo broj> |  <znak broja> <neoznacen ceo broj>
<neoznacen ceo broj> ::= <cifra> |  <neoznacen ceo broj><cifra>
<cifra> ::= 0|1|2|3|4|5|6|7|8|9
<znak broja> ::= +|-
```

Пример извођења:

```
<ceo broj> ⇒ <neoznacen ceo broj> ⇒ <neoznacen ceo broj><cifra> ⇒
<cifra><cifra> ⇒ 7<cifra> ⇒ 78
```

Пример описа аритметичких израза помоћу BNF:

```
<izraz>  ::= <izraz> + <term> | <term>
<term>   ::= <term> * <faktor> | <faktor>
<faktor> ::= ( <izraz> ) | <ceo broj>
```

Проширена Бекусова нотација (EBNF) - пример

- {...} - садржај се понавља 0 или више пута
- [...] - садржај се јавља опционо

Пример описа аритметичких израза помоћу EBNF:

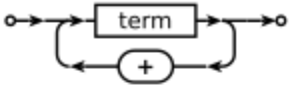
```
<izraz> ::= <term> {"+" <term>}
<term> ::= <faktor> {"*" <faktor>}
<faktor> := "(" <izraz> ")" | <ceo broj>
```

Пример описа наредбе if помоћу EBNF:

```
<if_naredba> ::=  if "(" <bulovski_izraz> ")" <niz_naredbi>
                  [ else <niz_naredbi> ]
<niz_naredbi> ::= <naredba> | "{" { <naredba> } "}"
```

Синтаксички дијаграми - пример

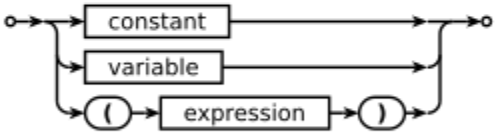
expression:



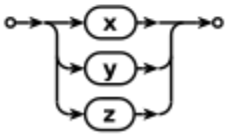
term:



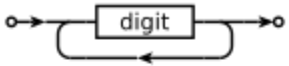
factor:



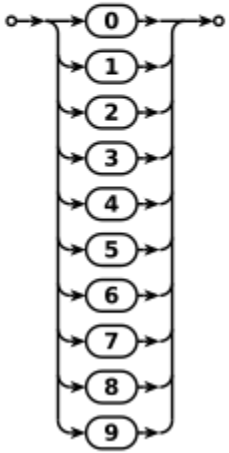
variable:



constant:



digit:



Семантичка анализа

Задатак семантичке анализе

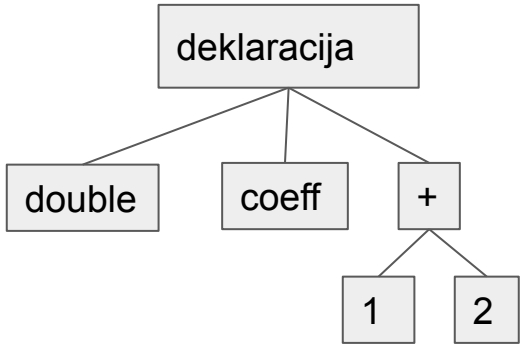
- Семантичка анализа је додатан пролаз којим се проверавају грешке које је тешко описати синтаксичким правилима
- Пример синтаксички исправне, али семантички неисправне реченице је често следећи (потиче од Ноама Чомског):
  - *Безбојне зелене идеје бесно спавају.*
- Ипак, паралеле са говорним језиком су овде мало натегнуте
- Најзначајнији задатак семантичке анализе је **провера типова** и **имплицитна конверзија типова**
- Семантичка анализа обично анализира дрво изграђено током фазе синтаксичке анализе и незнатно га модификује
- Код динамички типизираних програмских језика семантичка анализа се врши током извршавања програма (интерпретације или JIT компилације)

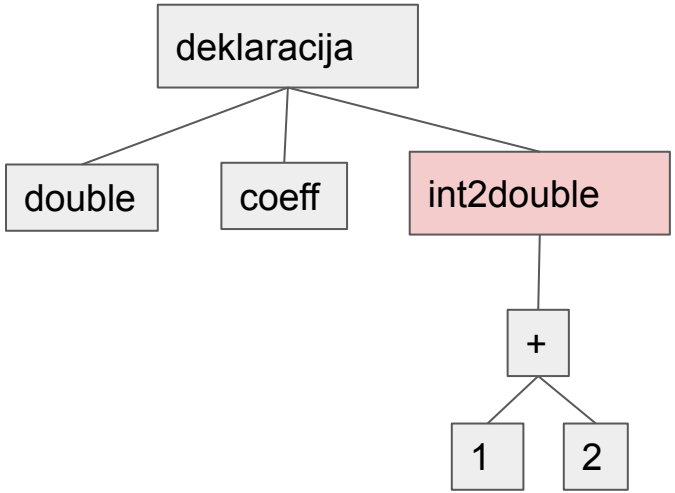
Најчешће грешке које се пријављују у фази семантичке анализе

- непозната (недекларисана) променљива, функција, ...
- вишеструко декларисана променљива у истом досегу
- погрешан број аргумената функције
- погрешан тип операнда или аргумента
  - нпр. сабирање две константне ниске
  - прослеђивање вредности типа double функцији која очекује char*
  - примена оператора (нпр. +) на вредност позива void функције
  - индексирање низа вредношћу која није целобројна
  - индексирање скалара
- `break` или `continue` ван петље (или наредбе switch)
- позив методе која није статичка из статичке методе у класи
- ...

Имплицитна конверзија типова

```
double coeff = 1 + 2;
...
```





Генерисање међукода

Троадресни међукод

- Најчешћи облик међукода је троадресни међукод
  - Нешто између асемблера и језика вишег нивоа
  - Бесконачно променљивих (регистара) - нове променљиве се уводе кад год затреба
  - Изрази имају највише два операнда
  - Доделе вредности израза променљивама (регистрима)
  - Контрола тока сведена на безусловне и условне скокове (GOTO)
  - Обично се задржавају класичне функције
  - Информације о типовима не морају бити задржане (мада у многим реалним форматима јесу)
  - ...

Пример троадресног међукода

```
double coeff = 1 + 2;
double niz1[10], niz2[10];
niz2[0] = coeff * x;
int i;
for (i = 0; i < 9; i++) {
    double y = coeff * x;
    niz2[i+1] = niz1[i+1] + i*y;
}
```

```
  t1 := 1 + 2
  coeff := int2double(t1)
  t2 := coeff * x
  niz2[0] := t2
  i := 0
petlja:
  if i >= 9 goto kraj
  y := coeff * x
  t3 := i + 1
  t4 := niz1[t3]
  t5 := i * y
  t6 := t4 + t5
  t7 := i + 1
  niz2[t7] := t6
  i := i + 1
  goto petlja
kraj:
```

Оптимизација

Задатак оптимизације

- Задатак оптимизације је побољшање перформанси кода без промене његовог видљивог понашања (каже се да се чува семантика програма)
- Оптимизује се по различитим критеријумима (процењено време извршавања, величина кода, ...)
- Добијени кôд не мора бити оптималан
- Баланс између времена потребног за извођење оптимизације и ефикасности оптимизованог кода - најсложеније оптимизације се често не исплати радити
- За разлику од осталих фаза које су суштински решене пре скоро пола века, оптимизација је фаза која се и даље интензивно проучава
- “Срце” савремених компилатора

Постојање оптимизације олакшава раније фазе

- Предњи део компилатора и генератор међукода могу слободно да генеришу кôд који је прилично велики и неефикасан (али правилног облика) рачунајући на то да ће оптимизатор тај кôд прилично поправити
- Неке неефикасности настају јер се програмер изражава на језику високог нивоа и нема могућност да се изрази на ефикаснији начин (рачуна се на то да ће ефикасност донети компилаторска оптимизација)
  - Нпр. `x = a[j] * a[j]` се преводи у троадресни кôд у коме се за сваки приступ низу изнова израчунава померај у бајтовима (и то ће сигурно бити накнадно оптимизовано)

```
t1 := 4 * j
t2 := a[t1]
t3 := 4 * j
t4 := a[t3]
x := t4 * t2
```

Машински независне и машински зависне оптимизације

- Неке оптимизације нимало не зависе од рачунара на ком ће се програм извршавати.
  - Нпр. `0 * x` се може заменити са `0` и на сваком рачунару ће то бити брже
- Неке оптимизације веома зависе од рачунара на ком ће се програм извршавати.
  - Нпр. да ли се уместо `7*x` може вршити `(x << 3) - x` веома зависи од односа брзине извршавања операција множења, шифтовања и одузимања
- Оптимизација се зато често извршава у разним фазама и пролазима
  - Највећи део се изврши као оптимизација међукода
  - Машински зависне оптимизације се извршавају као део фазе генерисања кода

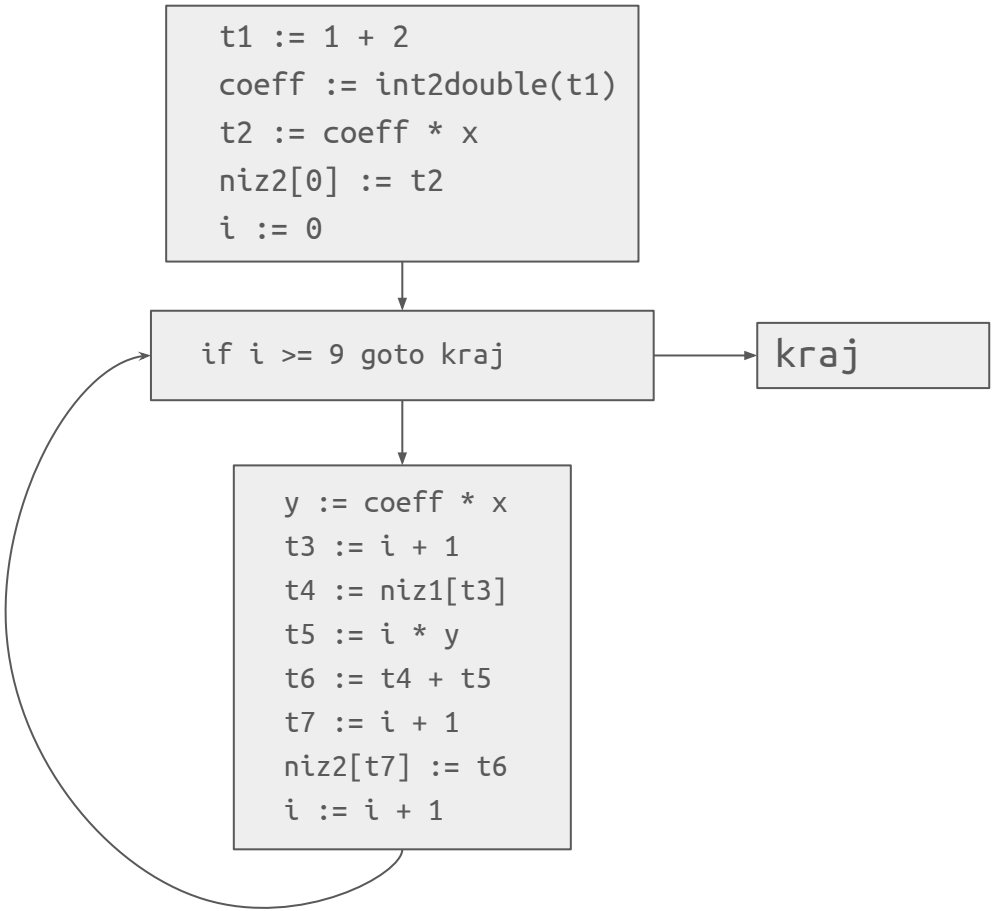
Локална и глобална оптимизација

- Најједноставније је вршити оптимизације делова програма “линијске структуре” - **локалне оптимизације**
- Оптимизација разгранатих и цикличних програма је компликованија, јер је потребно пратити сложен ток података кроз програм - **глобалне оптимизације**
- Најкомпликованије је вршити оптимизације које укључују зависности разних функција које се користе у програму - **интерпроцедурална оптимизација**

Граф контроле тока програма

- У троадресном коду је потребно идентификовати делове кода “линијске структуре”, такве да се све наредбе у њима увек извршавају од прве до последње (не може се нити ускочити, нити искочити из таквог блока)
- Такви блокови се онда на основу условних и безусловних скокова у коду повезују у граф контроле тока програма
- Сваки блок се оптимизује локалним оптимзацијама
- Граф се користи за анализу тока података која се користи за глобалне оптимзације

Граф контроле тока - пример




```
  t1 := 1 + 2
  coeff := int2double(t1)
  t2 := coeff * x
  niz2[0] := t2
  i := 0
petlja:
  if i >= 9 goto kraj
  y := coeff * x
  t3 := i + 1
  t4 := niz1[t3]
  t5 := i * y
  t6 := t4 + t5
  t7 := i + 1
  niz2[t7] := t6
  i := i + 1
  goto petlja
kraj:
```

Неки кораци оптимизације

- Неке се инструкције могу обрисати
  - `x := x + 0`
  - `x := x * 1`
- Неке се могу упростити - **редукција снаге** (енгл. *strength reduction*)
  - `x := x * 0` `x := 0`
  - `y := 2 * x` `y := x + x` или `y := x << 1`
- Операције над константама се могу извршити током компилације уместо током извршавања програма - **упрошћавање константи** (енгл. *constant folding*)
  - `x = 2 + 3` `x = 5`
  - `if 0 < 2 goto L` `goto L`
  - `if 2 < 0 goto L` `/* може се обрисати */`
  - Потребно је бити опрезан. Компилација и извршавање не морају да се врше на истој машини, па резултати могу бити различити (различита ширина речи или различита прецизност реалних бројева).

Неки кораци оптимизације

```
a := b + c
b := a - d
c := b + c
d := a - d
```

```
a := b + c
b := a - d
c := b + c
d := b
```

- **Елиминација заједничких подизраза** (енгл. *common subexpression elimination*)
- Изрази `b+c` и `a-d` се израчунавају два пута.
- Израз `b+c` се други пут рачуна над промењеном вредношћу `b` и зато ће вредност у другом израчунавању бити другачија него у првом
- Израз `a-d` се други пут израчунава над истим операндима као и први пут и зато се друго израчунавање може избећи

Неки кораци оптимизације

- **Пропагација константи** (енгл. *constant propagation*)

```
a := 1                  a := 1
b := a + 2              b := 1 + 2
a := a + 3              a := 1 + 3
c := a + 4              c := a + 4
```

- **Пропагација копија** (енгл. copy propagation)
  - променљиве дефинисане као копије других, обично су непотребне

```
x := y                  x := y
z := x + y              z := y + y
x := x + 2              x := y + 2
w := z * x              w := z * x
```

Неки кораци оптимизације

- **Елиминација мртвог кода** (енгл. *dead code elimination*)
  - Ако се променљива не користи након неке доделе, та додела може да се обрише

~~x := y~~
z := y + y
x := y + 2
w := z * x

Секвенца локалних оптимизација

- Ни једна локална оптимизација не ради пуно сама за себе.
- Обично једна оптимизација омогућава неку наредну.
- Оптимизације се извршавају једна по једна, док год је то могуће.
- Ако је време компилације ограничено, процес оптимизације се може зауставити у било ком тренутку.

```
a := 1
b := a + 2
a := a + 3
c := a + 4
```

```
a := 1
b := 1 + 2
a := 1 + 3
c := a + 4
```

```
a := 1
b := 3
a := 4
c := 4 + 4
```

```
a := 1
b := 3
a := 4
c := 8
```

```
b := 3
a := 4
c := 8
```

Глобалне оптимизације - проблеми

- Да ли је могуће извршити пропагацију константе x тј. заменити x са 3?

```
x := 3
if a < 10 goto L1
z := x + 2
x := x + 1
L1:
y := 4 + x
```

```
x := 3
if a < 10 goto L1
z := 3 + 2
x := 3 + 1
L1:
y := 4 + x
```

- Да би се то одредило, потребно је анализирати све могуће гране и проверити да ли се вредност променљиве може променити, што је доста компликовано када је програм разгранате и цикличке структуре

Оптимизација петљи

- Померање инваријантног кода (енгл. invariant code motion)
  - Ако се исто израчунавање врши више пута у петљи има смисла урадити га једном, пре петље
  - Пример (приказан на нивоу изворног кода):

```
for (int i = 0; i < n; i++) {
    r = sqrt(x*x + y*y);
    printf("%f", a[i] * r);
}
```

```
r = sqrt(x*x + y*y);
for (int i = 0; i < n; i++)
    printf("%f", a[i]*r);
```

Оптимизација петљи

- Индукционе променљиве и редукција снаге
- Множење бројачем петље се своди на сабирање

```
for (int i = 0; i < n; i++)
        a[i] = i;
```

```
    i := 0
petlja:
    if i >= n goto kraj
    t1 := 4 * i
    a[t1] := i
    i := i + 1
    goto petlja
kraj:
```

```
    t1 := 0
    i := 0
petlja:
    if i >= n goto kraj
    a[t1] := i
    t1 := t1 + 4
    i := i + 1
    goto petlja
kraj:
```

```
  t1 := 1 + 2
  coeff := int2double(t1)
  t2 := coeff * x
  niz2[0] := t2
  i := 0
petlja:
  if i >= 9 goto kraj
  y := coeff * x
  t3 := i + 1
  t4 := niz1[t3]
  t5 := i * y
  t6 := t4 + t5
  t7 := i + 1
  niz2[t7] := t6
  i := i + 1
  goto petlja
kraj:
```

Пример оптимизације

```
  t2 := 3.0 * y
  niz2[0] := t2
  i := 0
  y := 3.0 * x
  t5 := 0
petlja:
  if i >= 9 goto kraj
  i := i + 1
  t4 := niz1[i]
  t6 := t4 + t5
  niz2[i] := t6
  t5 := t5 + y
  goto petlja
kraj:
```

Генерисање кода

Генерисање кода

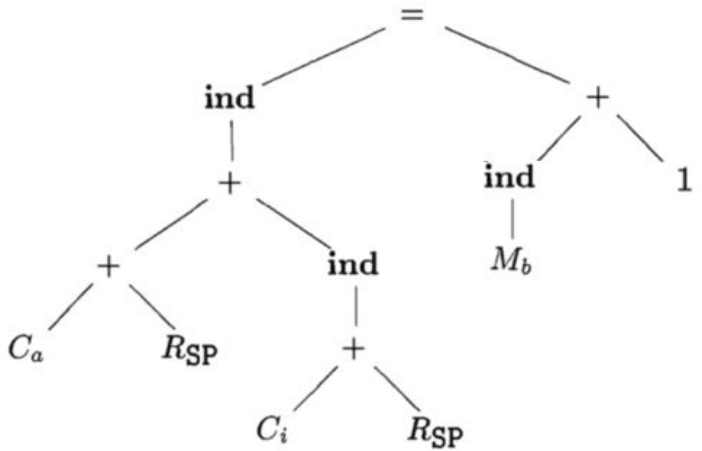
- Оптимизовани међукод се преводи у завршни асемблерски тј. машински код
- Неколико значајних одлука које утичу на квалитет генерисаног кода
  - **Одабир инструкција** (енгл. *instruction selection*) - одређује се којим машинским инструкцијама се моделују инструкције троадресног кода
  - **Алокација регистара** (енгл. *register allocation*) - одређује се локација на којој се свака од променљивих складишти
  - **Распоређивања инструкција** (енгл. *instruction scheduling*) - одређује се редослед инструкција који доприноси квалитетнијем искоришћавању регистара, али и проточне обраде и паралелизације на нивоу инструкција
- Ове фазе су прилично испреплетане - најчешће се прво ради одабир инструкција док се распоређивање инструкција некада раде и пре и после алокације регистара

Одабир инструкција

- Инструкције међукода се на различите начине могу превести у реалне асемблерске инструкције (нарочито ако је у питању нека CISC архитектура)
- На пример, ако променљиву треба увећати за 1
  - можемо искористити генеричку операцију сабирања уз непосредно адресирање
  - можемо искористи специјализовану инструкцију инкрементирања
- На пример, ако треба сабрати број у регистру и број у меморији
  - можемо искористити индиректно адресирање
  - можемо прво извршити пренос податка из меморије у регистар, па онда искористити регистарско адресирање
  - други начин се више исплати ако ће се тај податак користити неколико пута (и у наредним операцијама) - жртвујемо регистар, али добијамо на брзини
- Троадресни кôд се представља дрветом, које се онда прекрива појединачним инструкцијама на оптимални начин
  - грамзива стратегија или динамичко програмирање

Одабир инструкција - пример, варијанта 1

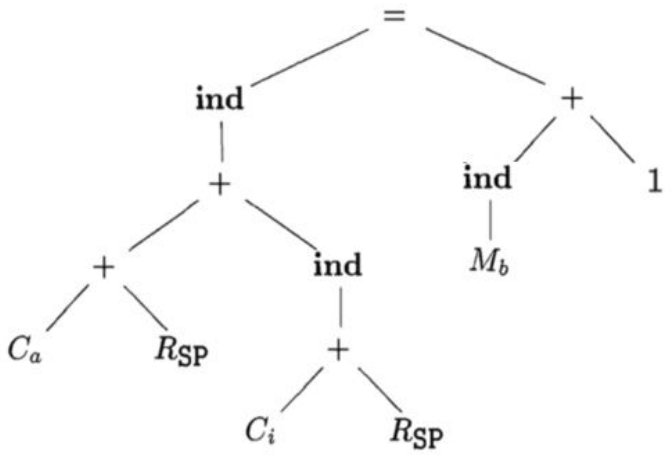
- Наредба a[i] := b + 1 се може, на пример, представити приказаним дрветом
- Mb - глобална меморијска адреса променљиве b
- Rsp - регистар који садржи показивач стека тренутне функције
- Ca - померај почетка низа a у односу на врх стека
- Ci - померај тренутне променљиве i у односу на врх стека




```
mov r0, rsp          ; upisujemo vrednost pokazivača rsp na početak stek okvira u registar r0
add r0, #Ca          ; uvećavamo r0 za pomeraj početka niza
mov r1, rsp          ; upisujemo vrednost pokazivača rsp na početak stek okvira u r1
add r1, #Ci          ; uvećavamo r1 za pomeraj promenljive i
mov r1, [r1]         ; učitavamo iz memorije sadržaj sa adrese određene vrednošću r1 u r1
add r0, r1           ; uvećavamo r0 za r1
mov r1, [Mb]         ; učitavamo sadržaj memorije sa adrese Mb u registar r1
add r1, 1            ; uvećavamo vrednost registra r1 za 1
mov [r0], r1         ; u memoriju na adresu određenu trenutnom vrednošću r0 upisujemo vrednost r1
```

Одабир инструкција - пример, варијанта 2

- Наредба a[i] := b + 1 се може, на пример, представити приказаним дрветом
- Mb - глобална меморијска адреса променљиве b
- Rsp - регистар који садржи показивач стека тренутне функције
- Ca - померај почетка низа a у односу на врх стека
- Ci - померај тренутне променљиве i у односу на врх стека




```
mov r0, #Ca          ; upisujemo pomeraj početka niza #Ca u registar r0
add r0, rsp          ; uvećavamo r0 za vrednost pokazivača okvira steka rsp
add r0, [rsp + #Ci]  ; uvećavamo r0 za vrednost upisanu u memoriji na adresi
                     ; određenoj zbirom rsp i pomeraja #Ci
mov r1, [Mb]         ; učitavamo sadržaj memorije sa adrese Mb u registar r1
inc r1               ; uvećavamo vrednost registra r1 za 1
mov [r0], r1         ; u memoriju na adresu određenu trenutnom vrednošću r0
                     ; upisujemo vrednost r1
```

Регистарска алокација

- Број регистара у процесору је ограничен
- Међукод користи неограничен број регистара
- За сваку променљиву међукода је потребно одредити неки регистар процесора или неку меморијску локацију на којој ће се та променљива чувати
- Циљ је што више смањити потребу за коришћењем меморије и премештањем података између меморије и процесора
- Две променљиве могу да се сместе у исти регистар ако се не користе истовремено (ако им се не преклапа “животни век”)
- Регистарска алокација се своди на проблем бојења графова који је NP комплетан, али се ефикасно решава хеуристикама

Распоређивање инструкција (енгл. instruction scheduling)

- Редослед извршавања неких инструкција се може заменити без утицаја на резултате израчунавања
- Тежи се редоследу у ком ће сва коришћења променљивих бити локализована, јер се тиме променљиве краће задржавају у регистрима, чиме се смањује потреба за коришћењем меморије и пребацивањем података између процесора и меморије

Распоређивање инструкција и проточна обрада

- Савремени процесор може да извршава више инструкција током једног откуцаја системског сата тј. има особину **проточне обраде** (енгл. *pipelining*)
- Док се једна инструкција довлачи из меморије у процесор, друга се декодира, трећа се већ извршава, а четвртој се већ складиште резултати
- Промена редоследа инструкција може утицати на то да се проточна обрада боље искористи тј. да се избегну застоји у проточној обради настали због зависности између суседних инструкција.
- Када је наредној инструкцији потребан резултат претходне, њено извршвање мора да причека да претходна инструкција заврши са уписом својих резултата. Распоређивач у тај застој може уметнути неку другу инструкцију, независну од ових (она ће бити извршена док би процесор практично чекао у празно).