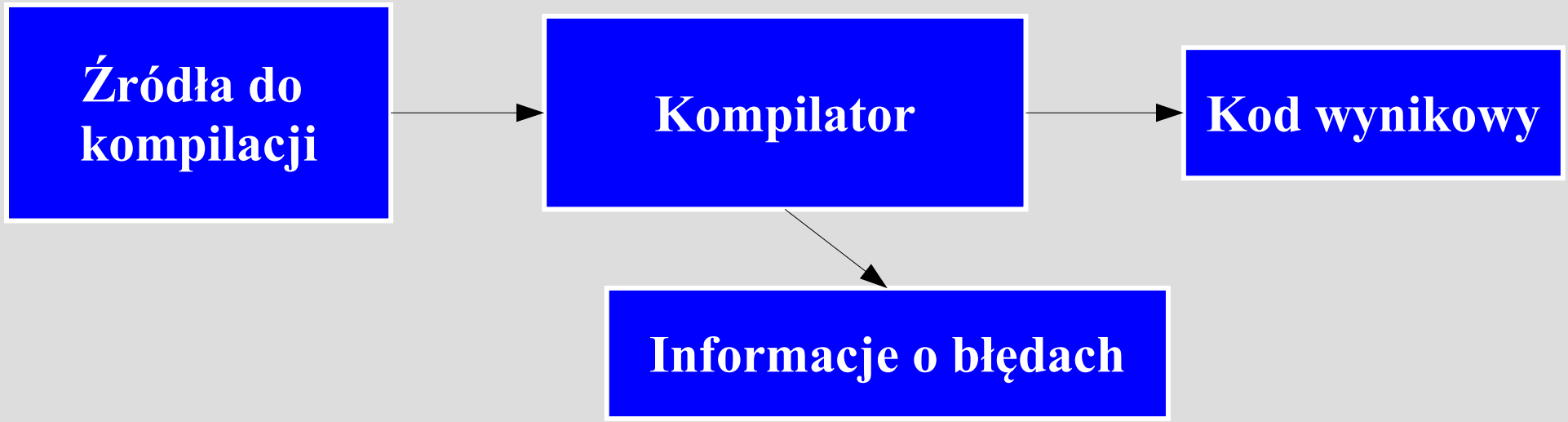
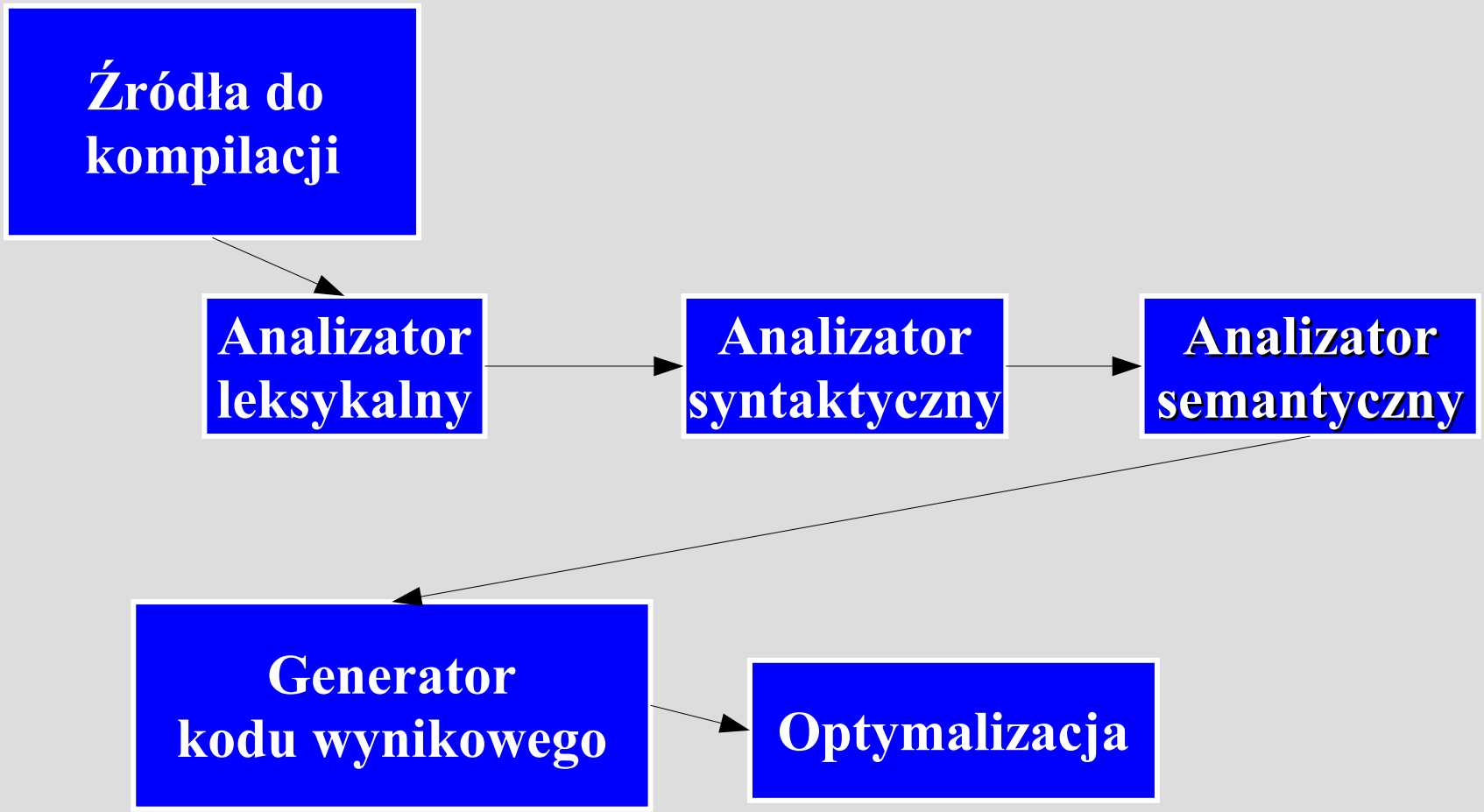
Języki formalne
i
metody kompilacji

Piotr Błaszyński

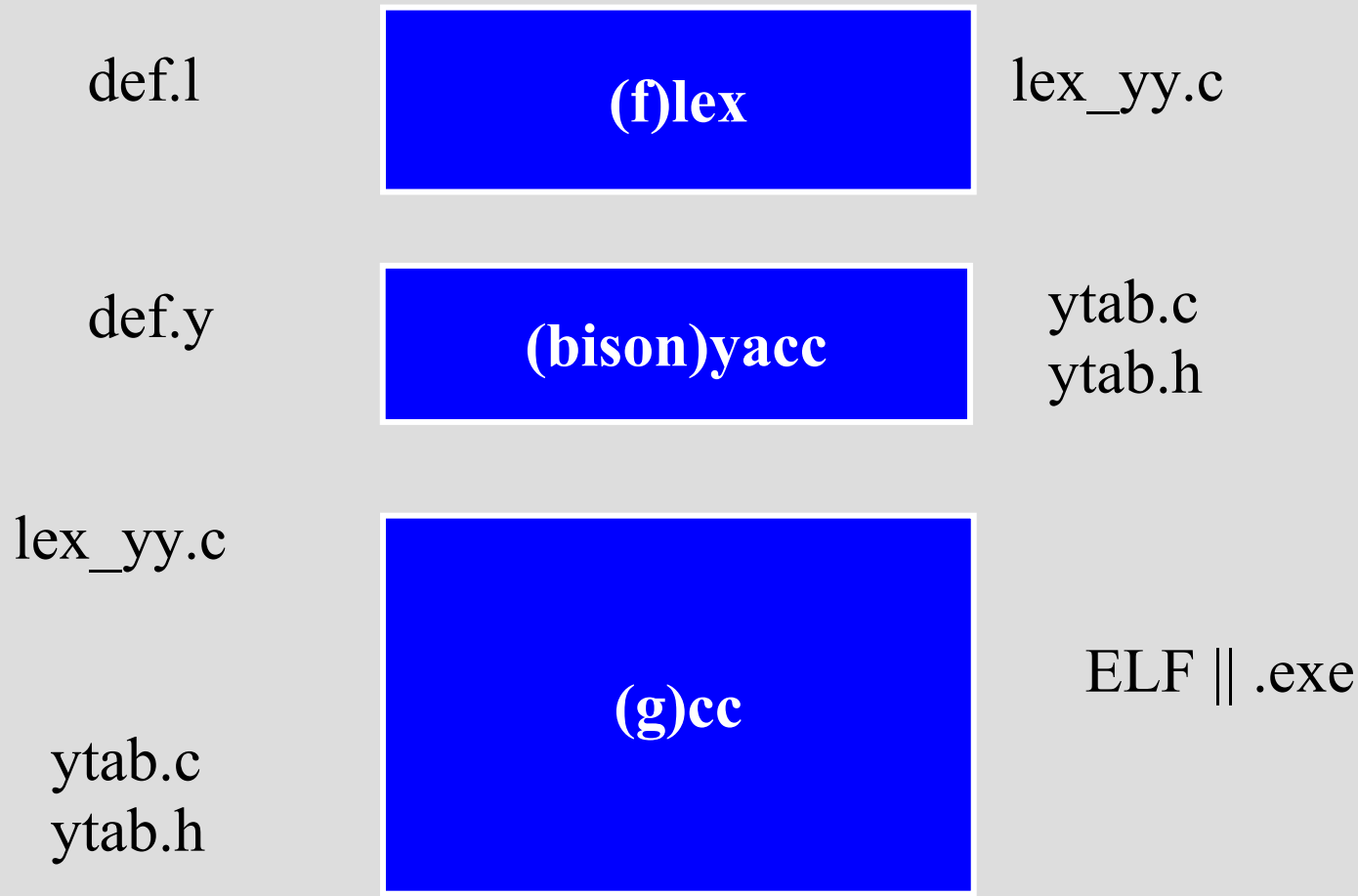
Budowa kompilatora:



Budowa kompilatora:



Budowa kompilatora:



Analizator semantyczny:

- Kontrola typów
- Właściwy przepływ sterowania
- Sprawdzenie niepowtarzalności nazw
- Różne inne testy związane z nazwami

Analizator semantyczny:

E→ literal { E.type := char }
E→ num { E.type := integer }
E→ id { E.type := lookup(id.entry) }
E→ E1 mod E2 { E.type := if E1.type = integer and E2.type = integer then integer else type_error }
E→ E1 [E2] { E.type := if E2.type = integer and E1.type = array(t) then t else type_error}

Kod pośredni, reprezentacje liniowe:

Zazwyczaj reprezentacje liniowe klasyfikuje się z uwagi na liczbę adresów, które mogą być argumentami instrukcji. Weźmy instrukcję: a := b * (c + 2)
i przyjrzyjmy się jej prawej stronie. Są tam używane dwa operatory arytmetyczne, + i *. Każdy z nich ma dwa argumenty oraz wynik, który trzeba gdzieś zapamiętać.

Kod pośredni, reprezentacje liniowe:

```
push y
mul
push x
sub
```

Kod jednoadresowy

```
t1 := 2
mul t1, y
t2 := x
sub t2, t1
```

Kod dwuadresowy

Okazuje się jednak, że definiując domyślne miejsce, z którego pobierane są argumenty i/lub domyślne miejsce dla wyniku, instrukcje nie muszą pobierać tak wielu argumentów.

Kod pośredni, reprezentacje liniowe:

Podstawowe rodzaje reprezentacji liniowych:

* Kod jednoadresowy, czyli zazwyczaj kod dla maszyn stosowych. Instrukcje pobierają z pamięci co najwyżej jeden adres, a pozostałe argumenty są pobierane ze stosu. Również wynik większości operacji jest umieszczany na stosie, a nie w pamięci. Kod jednoadresowy jest dobrą abstrakcją sprzętowych maszyn stosowych - procesorów, które nie mają dostępnych dla programisty rejestrów. Zaletą kodu jednoadresowego jest jego zwięzłość - instrukcje są krótkie, a wiele ich argumentów jest pobieranych z ustalonych miejsc. Wadą jest to, że liczba rozkazów potrzebnych do zapisania wybranej konstrukcji z kompilowanego języka bywa większa niż w przypadku kodu dwuadresowego czy trójadresowego.

Kod jednoadresowy

```
t1 := 2
mul t1, y
t2 := x
sub t2, t1
```

Kod dwuadresowy

```
t1 := 2
t2 := t1 * y
t3 := x - t2
```

Kod pośredni, reprezentacje liniowe:

* Kod dwuadresowy, w którym zazwyczaj instrukcje wymagające dwóch argumentów i wyniku zapisują swój rezultat w miejscu jednego z argumentów. Tę postać stosowano w kompilatorach generujących kod dla maszyn, których procesory miały tak działające rozkazy. Ponieważ wiele obecnie stosowanych procesorów obsługuje rozkazy odpowiadające lepiej opisanemu poniżej kodowi trójadresowemu, kod dwuadresowy jest rzadko spotykany.

Kod pośredni, reprezentacje liniowe:

```
push 2
push b
mul
push a
sub
```
Kod jednoadresowy

```
t1 := 2
mul t1, b
t2 := a
sub t2, t1
```
Kod dwuadresowy

```
t1 := 2
t2 := t1 * b
t3 := a - t2
```
Kod trójadresowy

Różne reprezentacje liniowe dla wyrażenia a - 2 * b

Kod pośredni, reprezentacje liniowe:

* Kod trójadresowy, w którym położenie obu argumentów instrukcji oraz miejsce na wynik są jawnie specyfikowane. Odpowiada to architekturze wielu procesorów RISC (Sparc, PowerPC, MIPS), a mniej - popularnej architekturze procesorów x86. Mimo tego, kod trójadresowy jest stosowany także w kompilatorach generujących kod dla procesorów Intela, np. w GCC. Najbardziej oczywistą implementacją kodu trójadresowego są czwórki - rekordy (lub coraz częściej obiekty) zawierające rozkaz (operację), jego dwa argumenty i wynik, czyli np. pola op,arg1,arg2 i wynik. Stąd pochodzi często spotykana nazwa tej implementacji - kod czwórkowy. Gdy któryś z rozkazów - np. operator logiczny not - potrzebuje tylko jednego argumentu, to pole arg2 pozostaje nieużywane.

Kod pośredni, reprezentacje liniowe:

Trzeba zauważyć, że w chwili generowania kodu pośredniego nie mamy jeszcze przydzielonych adresów w pamięci odpowiadających zmiennym. Wobec tego, jako arg1, arg2 i wynik zamiast adresów zmiennych używamy wskaźników na ich wpisy w tablicy symboli. Jednak, jak widać w przykładowym kodzie, użycie instrukcji dwuargumentowych w przypadku wyliczania skomplikowanych wyrażeń zmusza nas do przechowywania wyników fragmentów obliczeń w tymczasowo przydzielonym miejscu, nieodpowiadającym żadnej ze zmiennych z kodu źródłowego. Te zmienne, nazywane zmiennymi tymczasowymi, są automatycznie tworzone przez kompilator. Żeby zachować jednolitą zawartość czwórek, takie zmienne tymczasowe muszą być dopisywane do tablicy symboli.

Kod pośredni, reprezentacje liniowe:

Aby pominąć konieczność dopisywania zmiennych tymczasowych do tablicy symboli, można stosować trójki - reprezentację zbliżoną do kodu dwuadresowego, przy czym wynik rozkazu, zamiast zapamiętania pod podanym adresem, jest wpisywany do specjalnej zmiennej tymczasowej powiązanej z danym rozkazem (np. poprzez jego kolejny numer). Wówczas oczywiście niepotrzebne jest pole wyniku, a pola argumentów w rekordzie mogą zawierać albo wskaźnik na wpis w tablicy symboli, albo numer instrukcji. Oprócz takiej kłopotliwej zawartości pól argumentów, wadą trójek jest konieczność zmiany wszystkich odwołań do danego wyniku w przypadku zmiany kolejności instrukcji. Ponieważ utrudnia to bardzo optymalizację kodu, trójki nie są więc zbyt często stosowane.

Kod pośredni, NP, ONP:

Jeśli symbol jest operandem dodajemy go do kolejki wyjście.
Jeśli symbol jest operatorem, o1, wtedy:
1) dopóki na górze stosu znajduje się operator, o2 taki, że:
o1 jest łączny lub lewostronnie łączny i jego kolejność wykonywania jest mniejsza lub równa kolejności wyk. o2, lub
o1 jest prawostronnie łączny i jego kolejność wykonywania jest mniejsza od o2,
zdejmujemy o2 ze stosu i dokładamy go do kolejki wyjściowej;
2) wkładamy o1 na stos operatorów.
Jeżeli symbol to lewy nawias wkładamy go na stos.

Kod pośredni, NP, ONP:

Jeżeli symbol jest prawym nawiasem to zdejmujemy operatory ze stosu i dokładamy je do kolejki wyjście, dopóki symbol na górze stosu nie jest lewym nawiasem, kiedy dojdziemy do tego miejsca zdejmujemy lewy nawias ze stosu bez dokładania go do kolejki wyjście. Jeśli stos zostanie opróżniony i nie napotkamy lewego nawiasu, oznacza to, że nawiasy zostały źle umieszczone.
Jeśli nie ma więcej symboli do przeczytania, zdejmujemy wszystkie symbole ze stosu (jeśli jakieś są) i dodajemy je do kolejki wyjścia. (Powinny to być wyłącznie operatory, jeśli wystąpi jakiś nawias, znaczy to, że nawiasy zostały źle umieszczone.)
Przykład: 3+4*2/(1-5)^2

Kod pośredni, NP, ONP:

Jeżeli symbol jest prawym nawiasem to zdejmujemy operatory ze stosu i dokładamy je do kolejki wyjście, dopóki symbol na górze stosu nie jest lewym nawiasem, kiedy dojdziemy do tego miejsca zdejmujemy lewy nawias ze stosu bez dokładania go do kolejki wyjście. Jeśli stos zostanie opróżniony i nie napotkamy lewego nawiasu, oznacza to, że nawiasy zostały źle umieszczone.
Jeśli nie ma więcej symboli do przeczytania, zdejmujemy wszystkie symbole ze stosu (jeśli jakieś są) i dodajemy je do kolejki wyjścia. (Powinny to być wyłącznie operatory, jeśli wystąpi jakiś nawias, znaczy to, że nawiasy zostały źle umieszczone.)
Przykład: 3+4*2/(1-5)^2

Kod pośredni, istotne trójki:

- przypisania a = b *op* c, *op* to operator logiczny lub arytmetyczny
- a = *op* b, *op* to operator unarny (np. -, ~, !, konwersje)
- kopiowanie a = b,
- skok bezwarunkowy goto L (L – adres trójki, do której skacze)
- skok warunkowy if a *rop* b goto L *rop* – operator relacji (np. <, >, <=, ==, !=)
- param a_i , call p, n – przekazanie zestawu n parametrów i wywołanie procedury p

Kod pośredni, istotne trójki:

- przypisania indeksowane a = b[i] i a[i] = b,
- przypisanie adresu i wskaźnika a = &b i a = *b lub *a = b
- produkcja S-> while E do R

```
S.b = nowa etykieta //pierwszy element
S.e = nowa etykieta //zaostatni element
S.kod = gen(S.b:) |
                E.kod |
                gen(if E.wartosc = 0 goto S.e) |
                R.code |
                gen(goto S.b) |
                gen(S.e:)
```

Generowanie kodu, wstęp:

- Kod generowany jest dla różnych maszyn, trzeba pamiętać, że ten sam program źródłowy może się tłumaczyć na różne kody wynikowe,

Różnice:

- zużycie pamięci,
- szybkośc działania,
- liczba używanych rejestrów,
- ilość zużywanej energii,

- Istotne są też wyniki działania programu, powinny być takie same,

Generowanie kodu, wstęp:

- Generowanie kodu obejmuje:
- wyznaczanie kolejności operacji,
- przydzielanie rejestrów do przechowywania wartości,
- wybór odpowiednich instrukcji z docelowego języka (odpowiadającym operacjom z reprezentacji pośredniej),

Generowanie kodu, wstęp:

Problem generacji optymalnego kodu jest matematycznie nierostrzygalny, dodatkowo, jest utrudniony przez fakt istnienia różnych architektur docelowych.

Zagadnienia niezależne od architektury i systemu operacyjnego:
- zarządzanie pamięcią,
- wybór instrukcji,
- wybór rejestrów,
- kolejność wykonania.

Generowanie kodu, Wejście generatora:

Forma pośrednia, z informacjami potrzebnymi do określenia adresu (Uwaga! adres nie jest określany liniowo 1:1). Forma pośrednia powinna być przed generowaniem kodu sprawdzona przez poprzednie etapy kompilacji (frontend) i wykonane zostały na przykład odpowiednie operacje konwersji.
W niektórych przypadkach generowanie kodu może przebiegać współbieżnie z częścią analizy semantycznej.

Generowanie kodu, Wyjście generatora:

Program docelowy w:
- asemblerze (dodatkowy koszt koniecznej asemblacji),
- kodzie maszynowym z adresami względnymi (realokowalny) (dodatkowy koszt realokacji i linkowania),
- kodzie maszynowym z adresami bezwzględnymi (nie jest praktyczny, dobry do kompilatora "studenckiego"),

Generowanie kodu, Wyjście generatora:

Czemu konieczna jest odpowiednia alokacja rejestrów:
- operacje na nich są wydajniejsze ale jest ich mało
c=a+b
- ADD Rj, Ri – koszt 1, możliwe tylko wtedy, gdy Rj i Ri przechowują wartości a i b, Rj po wykonaniu operacji zawiera wartość c
- ADD c, Ri – koszt 2, możliwe przy b w Ri
- MOV c, Rj
  ADD Rj, Ri – koszt 3, ale c jest w Rj

Generowanie kodu, uproszczony algorytm:

Operujamy na trójkach, w jednym ruchu przekształcamy jedną trójkę, mając na uwadze fakt, czy operandy znajdują się w rejestrze i jeśli to możliwe wykorzystujemy ten fakt. Dodatkowo zakładamy, że każdy z operatorów ma swój odpowiednik na maszynie docelowej.

Generowanie kodu, uproszczony algorytm:

Kolejne założenie: wyniki obliczeń mogą być pozostawiane w rejestrach tak długo, jak to możliwe. Przenoszenie tych wartości do pamięci następuje tylko jeśli:

- rejestr je przechowujący jest potrzebny do innych obliczeń,
- przed samym wywołaniem procedury, skoku, albo wyrażenia, do którego odbywa się skok.

Generowanie kodu, uproszczony algorytm:

Konieczne jest przechowywanie tzw. deskryptorów rejestrów i deskryptorów adresów. Przechowują one co jest w danym rejestrze lub pod danym adresem.
- deskryptory adresów określają miejsce przechowywania wartości zmiennej (stos, pamięć, rejestr lub kilka z nich na raz) i pozwalają określić metodę dostepu do wartości. Przechowywać je można w tablicy symboli.

Generowanie kodu, uproszczony algorytm:

Trójka x:= y op z

1. Przy pomocji funkcji getreg określamy lokalizacji L, w której wynik y op z będzie przechowywany. L powinno być rejestrem, ale może być adresem w pamięci.
2. Sprawdzamy deskryptory adresów dla y, żeby znaleźć y' – jedno z miejsc przechowywania y. Jako pierwsze są wybierane miejsca wskazujące na rejestr. Jeżeli y' nie jest jeszcze w L, generujemy instrukcje MOV y', L

Generowanie kodu, uproszczony algorytm:

Trójka x:= y op z, lokalizacja L, y' w L
3. Generujemy instrukcje OP z', L, z' wyznaczamy podobnie jak punkcie 2. Uaktualniamy deskyptor adresu dla x, x jest teraz w L. Jeżeli L jest rejestrem, to zapisujemy w deskryptorze tego rejestru, że przechowujemy tam x i usuwamy x z innych deskryptorów rejestrów.
4. Jeżeli wartości y i/lub z nie mają kolejnego użycia a są przechowywane w rejestrze, usuwamy je z ich deskryptorów rejestrów.

Generowanie kodu, uproszczony algorytm:

Po przetworzeniu wszystkich trójek, dla wszystkich nazw, których wartości są aktywne, a nie są przechowane w pamięci generujemy odpowiednie instrukcje przeniesienia (MOV).

Generowanie kodu, algorytm, funkcja getreg:

1. Jeśli y jest w rejestrze, który nie przechowuje innych wartości (x:=y może powodować przechowywanie dwóch wartości jednocześnie) i y nie będzie później używane to zwracamy ten rejestr. Aktualizujemy deskryptor adresu dla y – L juz nie przechowuje y.
2. Nie udał się pkt. 1. - zwracamy pusty rejestr, jeśli jest jakiś,

Generowanie kodu, algorytm, funkcja getreg:

3. Nie udał się pkt. 2. - Jeśli x będzie dalej używany, albo op jest operatorem potrzebującym użycia rejestru (np. indeksowanie tablicy), znajdujemy używany rejestr R, składujemy jego zawartośc w pamięci M, jeżeli już nie jest we właściwej pamięci M (pamiętać o przechowywaniu wielu wartości w R). Aktualizujemy odpowiednio deskryptor adresu dla M. Zwracamy R. Wybór R nie jest optymalny, można stosować na przykład wybieranie rejestru, który zostanie najpóźniej użyty.
4. Jeśli x nie jest używany w danym bloku, albo nie ma pasującego rejestru wybieramy lokalizacje w pamięci dla x jako L

Generowanie kodu, instrukcja skoku:

Możliwe podejścia:
1. Rejestr określa warunek skoku przy pomocy jednego z 6 warunków: ujemny, 0, dodatni, nieujemny, nie 0, niedodatni, czyli wartość np: x<y określamy przez odjęcie x od y i sprawdzenie zawartości rejestru
2. Specjalne znaczniki określają (rejestr znaczników) warunek, podejście bardziej naturalne dla wielu maszyn.

Optymalizacja kodu wynikowego

- Optymalizacja kodu wynikowego polega na zmianie formy wyjściowej w celu minimalizacji lub maksymalizacji jednego z atrybutów uzyskanego programu. Najpopularniejsze cele
- minimalizacja czasu wykonania,
- zajętość pamięci,
- zużycie energii (urządzenia przenośne)

Optymalizacja kodu wynikowego

- Peephole optimization (przez wizjer)
  - eliminacja redundancji
  - optymalizacja przepływu
  - uproszczenia algebraiczne
  - użycie zamienników (dla danej architektury)

Optymalizacja kodu wynikowego

- eliminacja redundancji
  - (1) MOV R0, a
  - (2) MOV a,R0
  - usuwamy (2), bo (1) daje pewność, że zachodzi równość
  - jeśli przed (2) jest etykieta, to pewności nie ma
  - czyli (1) i (2) muszą być w tym samym bloku kodu (basic block)
  - część przypadków może być obsłużona przez alokacje rejestrów

Optymalizacja kodu wynikowego

- optymalizacja przepływu:
  - przy obsłudze instrukcji warunkowych itp., powstaje dużo skoków,
  - powstające skoki się często pokrywają
  - typowa optymalizacja przepływu sterowania:
    - goto LBL1
    - ....jakieś instrukcje....
    - LBL1: goto LBL2
  - zamieniamy na:
    - goto **LBL2**
    - ....jakieś instrukcje....
    - LBL1: goto LBL2

Optymalizacja kodu wynikowego

- uproszczenia algebraiczne
- Bardzo dużo możliwych przekształceń algebraicznych, nie wszystkie występują opłacalnie często
- typowe, często produkowane przez prosty generator kodu:
  - x=x+0 – brak efektu
  - x = x+1 – INC x
- podwyrażenia wspólne (a+b) – 4 * (a+b) : powyrażenie (a+b) można obliczyć raz,

Optymalizacja kodu wynikowego

- użycie zamienników (dla danej architektury)
  - x*x tańsze z reguły niż wywołanie potęgowania,
  - przesunięcie o bit tańsze od mnożenia i dzielenia przez 2,
  - zmiennoprzecinkowe mnożenie przez stałą (przy dopuszczalnym przybliżeniu) tańsze od dzielenia,
  - użycie trybu autoinkrementacji (dodaje 1 do operandu po jego użyciu) lub autodekrementacji
  - inne zależne od architektury

Optymalizacja kodu wynikowego

- Lokalne (int**ra**procedural optimizations)
  - Łatwiejsze i szybsze w wykonaniu, ale znacznie utrudnione przy uzyciu zmiennych globalnych (trzeba uwzględniać najgorsze przypadki)
- Globalne (int**er**procedural optimizations)
  - Więcej informacji, co za tym idzie nowe możliwości, np. umieszczanie kodu funkcji w miejscu jej wywołania

Optymalizacja kodu wynikowego

- Zmiana kolejności pętli (*ang. loop interchange*), dotyczy także pojedynczych instrukcji, jednak odczuwalne dla systemu znaczenie ma w przypadku pętli. Bloki kodu są zamieniane w celu uzyskania lepszego wykorzystania pamięci, w szczególność pamięci podręcznej. Może oznaczać to w na przykład zmniejszenie zużycia energii pobieranej na dostęp do pamięci. Zastosowanie tego algorytmu przy niekorzystnych układach pętli (złożone warunki kończenia pętli) może doprowadzić jednak do zwiększenia zużycia energii.

```
for (i = 0; i<100 ; i++)
    for (j = 0 ;j< 200 ;j++)
        a[i,j] = i + j
```

```
for (j = 0 ;j< 200 ;j++)
    for (i = 0; i<100 ; i++)
        a[i,j] = i + j
```

Optymalizacja kodu wynikowego

- Dzielenie pętli (*ang. loop fission (tiling, blocking*)) polega na takim podziale pętli (jak również innych bloków kodu), aby każdorazowo wykonywany blok kodu mieścił się w pamięci podręcznej. Dzięki zastosowaniu tego algorytmu możliwe jest również uzyskanie zmniejszenia zużywanej energii (na dostęp do pamięci), jak i znaczące przyspieszenie czasu wykonywania. Konieczne jest stałe określenie wielkości pętli lub definiowanie typowych rozmiarów pętli w danym programie albo obliczanie podziału pętli na podstawie wielkości pamięci podręcznej. W realnym świecie pierwszy przypadek zachodzi rzadko, konieczne jest podanie jako opcji typowego rozmiaru pętli lub rozmiaru pamięci podręcznej.

```
for (i=1 ; i< N ; i++)
(1){
(2)     A[i] = A[i] + B[i-1];
(3)     B[i] = C[i-1]*X + Z;
(4)     C[i] = 1/B[i];
(5)     D[i] = sqrt(C[i]);
(6) }
```

```
(1) for (i=0 ; i< N-1 ; i++)
(3)     B[ib+1] = C[ib]*X + Z
(4)     C[ib+1] = 1/B[ib+1]
(6) }
(1) for ib=0 to N-1 do
(2)     A[ib+1] = A[ib+1] + B[ib]
(1) for ib=0 to N-1 do
(5)     D[ib+1] = sqrt(C[ib+1])
(1) i = N+1
```

Optymalizacja kodu wynikowego

- Rozwijanie pętli (*ang. loop unrolling*) polega na powtarzaniu zawartości pętli i wykonywaniu odpowiednio mniejszej liczby iteracji w stosunku do oryginalnej liczby iteracji. Dzięki temu zyskuje się zmniejszenie liczby wykonywanych przez program skoków warunkowych. Zwiększana jest w tym przypadku również lokalność danych poprzez ograniczenie przesyłania danych z pamięci do rejestrów i z powrotem.

```
for (i = 0; i < 200; i++)
    g ();
```

```
for (i = 0; i < 200; i += 2)
{
    g ();
    g ();
}
```

Optymalizacja kodu wynikowego

- Łączenie pętli (*ang. loop fusion*) polega na łączeniu kilku pętli operujących na różnych elementach ale mających takie same warunki zakończenia. W przypadku poprawnego dobrania ograniczenia na rozmiar łączonych danych pozwala to na uzyskanie szybciej wykonującego się kodu bez zwiększania zużycia energii przez pamięć.

```
int i, a[100], b[100];
for (i = 0; i < 300; i++)
    a[i] = 1;
for (i = 0; i < 300; i++)
    b[i] = 2;
```

```
int i, a[100], b[100
for (i = 0; i < 300; i++)
{
    a[i] = 1;
    b[i] = 2;
}
```

Optymalizacja kodu wynikowego

- Rozszerzenie skalara (*ang. scalar expansion*) polega na zamienieniu zmiennej skalarnej na zmienną tablicową, która reprezentuje wartości pierwotnej zmiennej w poszczególnych iteracjach pętli. Tego rodzaju zmiana jest klasycznym przykładem szybszego wykonania kodu, ale przy zwiększonym użyciu pamięci. W niektórych jednak przypadkach taka zmiana może wpłynąć pozytywnie zarówno na szybkość i zużycie energii.

```
for (i=1 ; i< N ; i++)
{
    T = A[i] + B[i];
    C[i] = T + 1/T;
}
```

```
int Tx[N];
for (i=1 ; i< N ; i++)
{
    Tx[i] = A[i] + B[i];
    C[i] = Tx[i] + 1/Tx[i];
}
```

Optymalizacja kodu wynikowego

Źródła dla zainteresowanych:
http://www.nullstone.com/htmls/category.htm

http://www.agner.org/optimize/#manuals

http://www.redhat.com/en_us/USA/home/gnupro/technicaldetails/gc