Predavanje 7

Prevajalniki in tolmači

Iztok Savnik, FAMNIT

April, 2023.

Prosojnice

Jan Karabaš, Lecture, Compilers and interpreters, Programming II, FAMNIT, 2015.

Torben Aegidius Mogensen, Basics of Compiler design, University of Copenhagen, 2010.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Compilers, 2nd Edition, Addison Wesley, 2007 (Red Dragon Book).

Vsebina

- Zgodovina
- Arhitektura prevajalnika
- Leksikalna analiza
- Slovnice & razčlenjevalnik
- Semantična analiza
- Vmesni jezik
- Generacija kode
- Optimizacija kode

Zgodovina

- 30-40’s: eno-namenski računalniki, strojna predstavitev podatkov in programa (ZuSe, ENIAC);
- 1952: G. Hopper, prevajalnik A-0, povezovalnik in nalagalnik;
- 1952: A. Glennie, autocode, enostaven zbirnik;
- 1957: J. Backus, Fortran, prvi pravi prevajalnik, BNF, prevedena koda boljša kot napisana v zbirniku;
- 1958: Bauer et al., Algol 58, prvi moderni prevajalnik;
- 60’s: IBM’s navzkrižni prevajalnik za IBM 7xx arhitekturo (npr. iz UNIVAC)
- 1962: Hart & Levin: LISP, prvi samogostiteljski prevajalnik;

Zgodovina

- 1968-1972: UNIX, C, lex, yacc, make, sh, grep;
- 70's: Pascal, Niklaus Wirth, Zürich, CDC Pascal
- 70's: SmallTalk, prvi sprotni (just-in-time) prevajalnik;
- 80's: Erlang, C++, Ocaml, Modula, Ada, Perl, Objective-C;
- 90’s: Haskell, Python, Ruby, Java;
- 00’s: C#, Scala, F#, Go, mobilne arhitekture, funkcijski jeziki;
- 10’s: TypeScript, Julia, Raku, Swift; podatkovni, funkcijski, objektno-usmerjeni.

Vsebina

- Zgodovina
- Arhitektura prevajalnika
- Leksikalna analiza
- Slovnice & razčlenjevalnik
- Semantična analiza
- Vmesni jezik
- Generacija kode
- Optimizacija kode

Zakaj prevajalniki in tolmači?

- Izvorna koda
  - Program je praviloma predstavljen z navadno tekstovno datoteko, ki se ne more izvajati na ciljnem stroju.
- Izvedljiva koda
  - Zapis programa, ki se lahko direktno izvaja na računalniku.
- Običajni cilji prevajanja/tolmačenja
  - Strojna koda (tudi oblika vmesne kode);
  - Zlogovna koda;
  - Izvedljiva datoteka;
  - Drug programski jezik;
  - Navzkrižno prevajanje.

Cilji prevajanja

- Strojna koda
  - Vezana na konkretno arhitekturo in na gostiteljski operacijski sistem, C, C++ (gcc), Fortran (gfortran), Pascal (fpc), OCaml (ocamlc),. . .
- Zlogovna koda
  - Pripravljena za izvajanje v okolju specifičnega virtualnega stroja, torej (v večini) neodvisna od stroja in operacijskega sistema, Java (javac), Python (python), Erlang (erl), C# (.NET), but also C and C++ (LLVM or .NET). . .
- Izvedljiva datoteka (izvorna koda)
  - Izvorna koda, ki je direktno izvedljiva, Unix lupine (bash, csh, zsh), BASIC (večina, vendar ne Visual Basic), Web (tangle);

Cilji prevajanja

- Programski jeziki
  - Fortran v C (f2c), Python v C (cpython);
- Navzkrižni prevajalniki
  - Aplikacije za mobilne telefone, in druge naprave se prevedejo z navzkrižnim prevajalnikom, ObjC za iOS, C++ za Android, Windows Phone, NVida CUDA, C za OpenCL. . .

Arhitektura prevajalnika

Prevajalnik je običajno sestavljen iz

i. Čelni del (front-end):
   - Rekonstrukcija vrstice,
   - Leksikalna analiza,
   - Predobdelava,
   - Sintaksna analiza,
   - Semantic analysis;

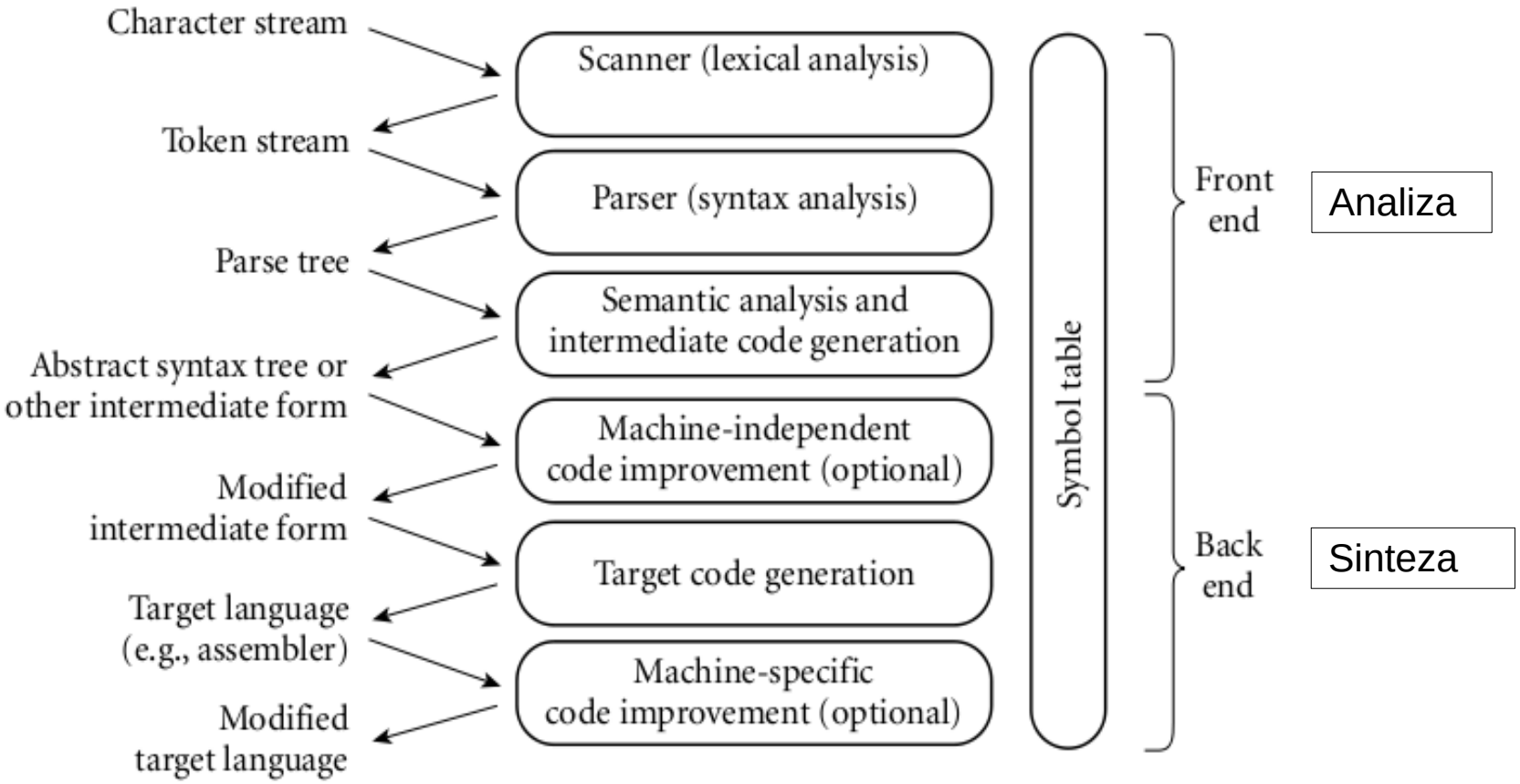
Analiza

ii. Zaledni del (back-end):
   - Vmesna optimizacija,
   - Tokovna (podatki in izvajanje) analiza,
   - Generiranje kode + optimizacija za ciljno arhitekturo;

Sinteza

iii. Povezovanje (opcijsko)

Faze prevajanja



Vsebina

- Zgodovina
- Arhitektura prevajalnika
- Leksikalna analiza
- Slovnice & razčlenjevalnik
- Semantična analiza
- Vmesni jezik
- Generacija kode
- Optimizacija kode

Rekonstrukcija vrstice

- UNICODE znake zamenjamo z ubežnimi sekvencami, npr. č → \x10D;
- Tabulatorji so zamenjani s presledki;
- Znaki za označitev konca vrstice so zamenjani s presledki;
- Znakovni nizi se združijo;
- Prazne vrstice
- se odstranijo.

Example (in C)

```
#include<stdio.h> int
main(
void)
{
                printf("Hello"
" world"
)
;
    return 0;}
```

Leksikalna analiza

- Normalizirana tekstovna oblika programa je razdeljena v seznam žetonov, leksikal. enot jezika.
- Običajni tipi žetonov so:
  - literali: 12345, 0xAB, "hello world", ’c’, true. . .
  - Ključne besede: if, else, for, function, return. . .
  - Identifikatorji: var1, a$, MyClass,. . .
  - Imena tipov: int, char*, ’a, void. . .
  - Operatorji: =, <-, +, :=, ==, >=,. . .
  - Oklepaji, navadni, zaviti, oglati;
  - Zamiki (stari Fortran, Python);
- Rezultat je sekvenca žetonov, kjer so odstranjeni vsi ločilni znaki.

Leksikalna analiza

- Leksikalna analiza prebere izvorno kodo po znakih. Vsak prepoznan žeton se izpiše v sekvenco žetonov;
- Natančnen, zapleten in utrujujoč postopek, veliko bolje je uporabiti regularne izraze za opis sintakse literalov;
- Regularni izrazi so vrsta abstraktnih računalnikov (DKA), ki prepoznajo predpisane vzorce v tekstu;
- Regularni izrazi so "lačni", ujamejo najdaljši možni tekst;
- Leksikalni analizatorji se zgradijo z uporabo posebnega prevajalnika – lekserja (lex, flex);
- Lekserji izdelajo izvorno kodo leksikalnega analizatorja v višje-nivojskem programskem jeziku (C, Java, OCaml, Pascal, Python).

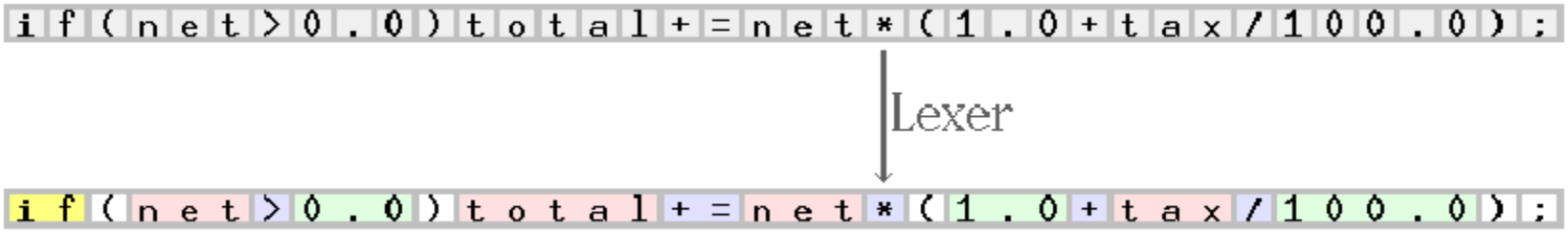
Leksikalna analiza: Primeri

Regularni izrazi:

```
      <id>   ::=   [_|__]?[a-zA-Z][_a-zA-Z0-9]*
<number>   ::=   [0-9]+ | 0x[0-9a-fA-F]+
     <kwd>   ::=    if | else | for | while | return
   <const>   ::=   <number> | <float> | <character>| \
                    <boolean> | <string>
```

Regularni izrazi so ’lačni’. Na primer, RE 0x[0-9a-fA-F]+ prepozna v nizu 0x109AB heksadecimalno konstanto namesto <const><id>.

Prepoznavanje žetonov:



Preprocesor

```
#ifndef SET_H
#define SET_H
#define max(a,b)  ((a) > (b) ? (a) : (b))
```

- Komentirani bloki in vrstice se izločijo
- Veliko jezikov uporablja preprocesorske makroje
  - Poseben jezik za procesiranje teksta
- Makroji so razširjeni pri preprocesiranju.
- Stavki za pogojno prevajanje se ovrednotijo in izpuščeni bloki simbolov se odstranijo
- Pomožne datoteke (vmesniki v C, C++) se dodajo namesto pripadajočih makrojev (#include)
- Leksikalna analiza se požene še enkrat.

Vsebina

- Zgodovina
- Arhitektura prevajalnika
- Leksikalna analiza
- Slovnice & razčlenjevalnik
- Semantična analiza
- Vmesni jezik
- Generacija kode
- Optimizacija kode

Slovnice programskih jezikov

Programski jeziki imajo enostavno sintakso, ki jo lahko opišemo s kontekstno neodvisno slovnico. Kontekstno neodvisna slovnica je matematična struktura naslednje oblike

$$G = (V, T, R, S)$$

- V je množica spremenljivk (neterminalov),
- T je množica simbolov (terminalov),
- R je množica produkcijskih pravil,
- S je začetni simbol.

.

Slovnice programskih jezikov

Primer:

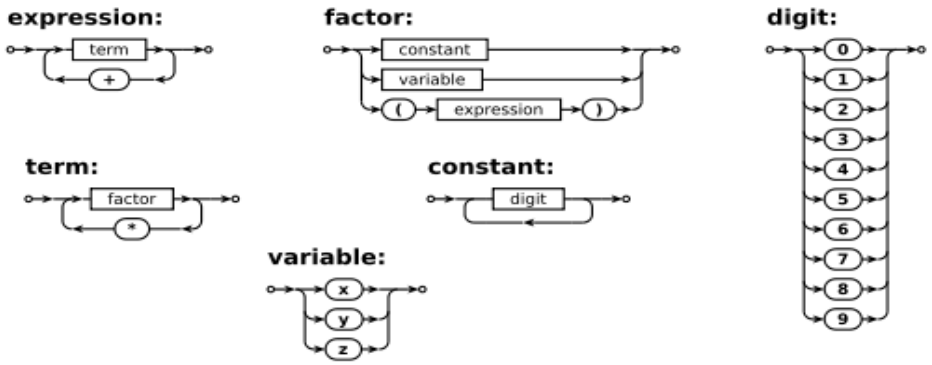
Jezik binarnih besed, ki so sestavljene iz enakega števila enic in ničel lahko opišemo z naslednjo slovnico.

G = ({S}, {0, 1}, R, S), kjer je R množica pravil

$$S \rightarrow 01 \mid 10 \mid 0S1 \mid 1S0$$

Backus-Naur oblika

Prikladen način opisa (kontekstno neodvisne) slovnice programskega jezika



Primer:

Enostaven jezik matematičnih izrazov:

```
<expression> ::= <term> | <term> "+" <expression>
<term>       ::= <factor> | <term> "*" <factor>
<factor>     ::= <constant> | <variable> | "(" <expression> ")"
<variable>   ::= "x" | "y" | "z"
<constant>   ::= <digit> | <digit> <constant>
```

Sintaksna analiza – razčlenjevanje

- Razčlenjevalnik je program, ki sprejme urejen seznam žetonov in prepozna pravila slovnice, ki tvorijo vhodni seznam žetonov.
- Kreira se konkretno sintaksno drevo vhodnega niza žetonov.
  - Koren drevesa ustreza začetnem neterminalnem simbolu.
  - Vozlišča ustrezajo pravilom (spremenljivke) in listi simbolom (terminali)
- Za vsak izraz prebran iz vhoda razčlenjevalnik vrne sintaksno drevo.

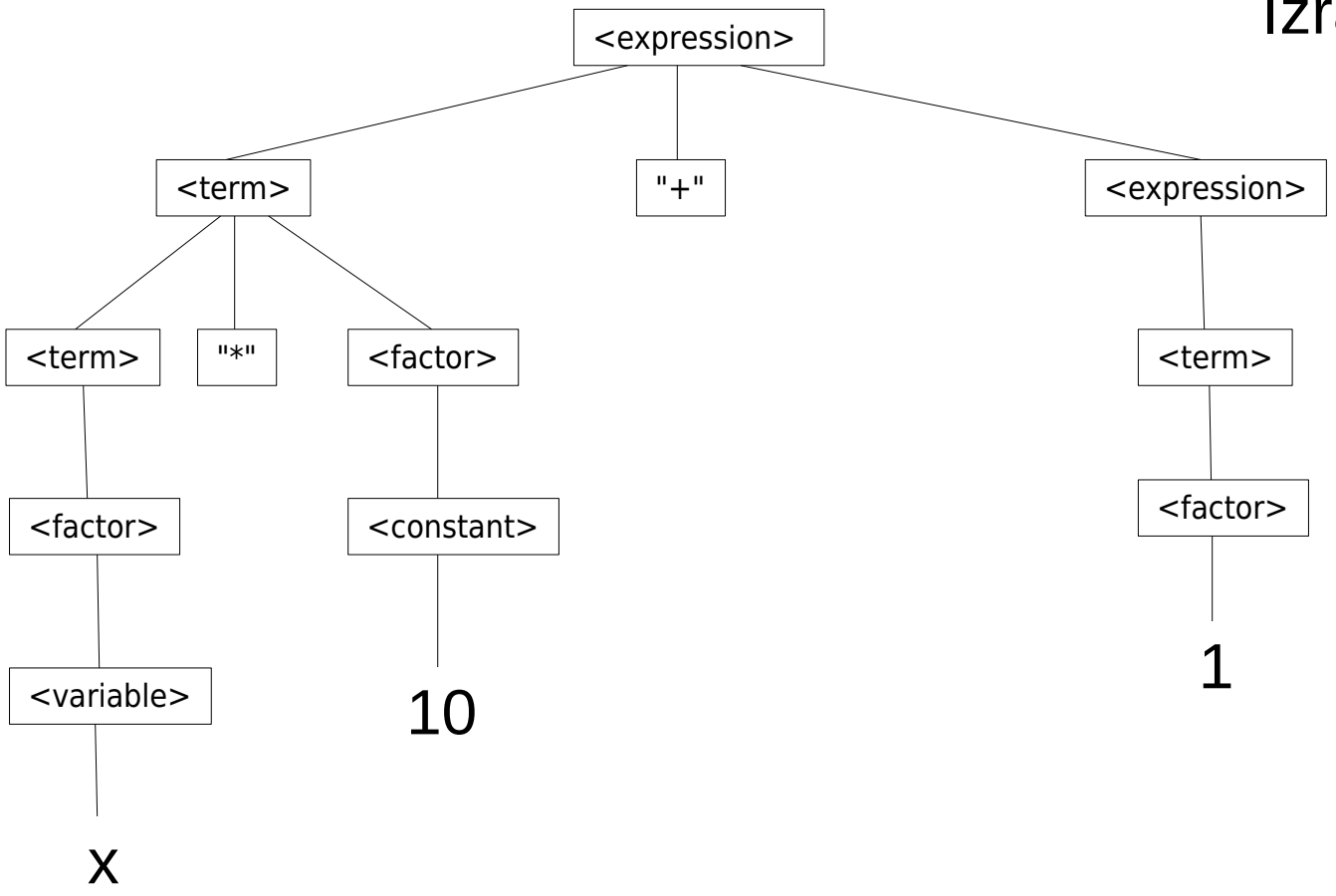
Sintaksna analiza – razčlenjevanje

- Proces konstrukcije sintaksnega drevesa je dvoumen ker so kontekstno neodvisne slovnice dvoumne.
- Poznamo več metod s katerimi se lahko izognemoo dvoumnosti:
  - ureditev pravil slovnice (prioriteta);
  - vnaprejšnji vmesnik (look-ahead buffer) za žetone (LL-razčlenjevalniki, LL(1));
  - branje iz leve, razčlenjevanje iz desne (LR, LALR parsers)

Primer: Sintaksno drevo

<expression> ::= <term> | <term> "+" <expression>
<term> ::= <factor> | <term> "*" <factor>
<factor> ::= <constant> | <variable> | "(" <expression> ")"
<variable> ::= "x" | "y" | "z"
<constant> ::= <digit> | <digit> <constant>

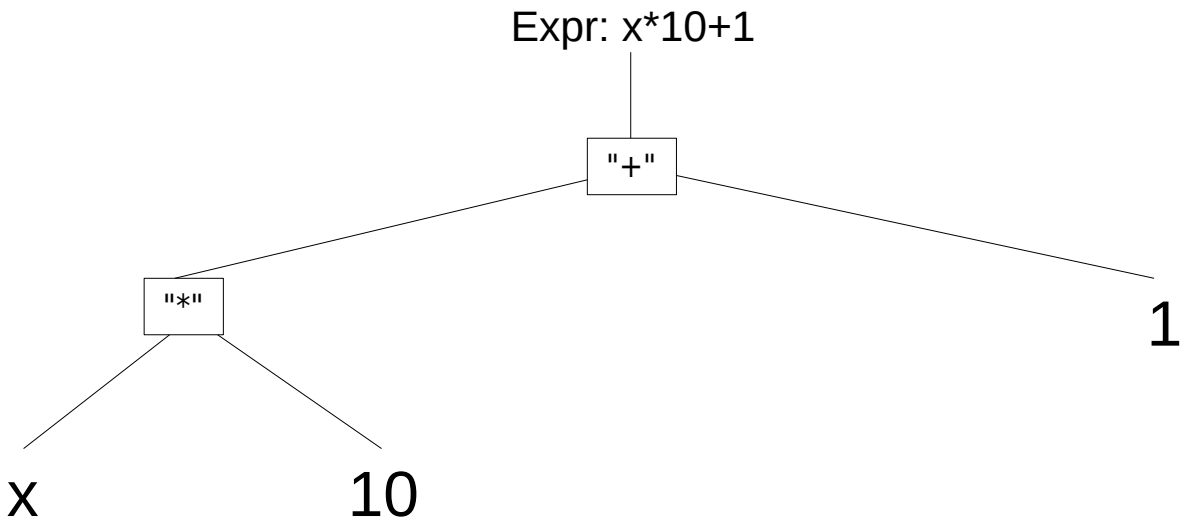
Izraz: x*10+1



Abstraktno sintaksno drevo (ASD)

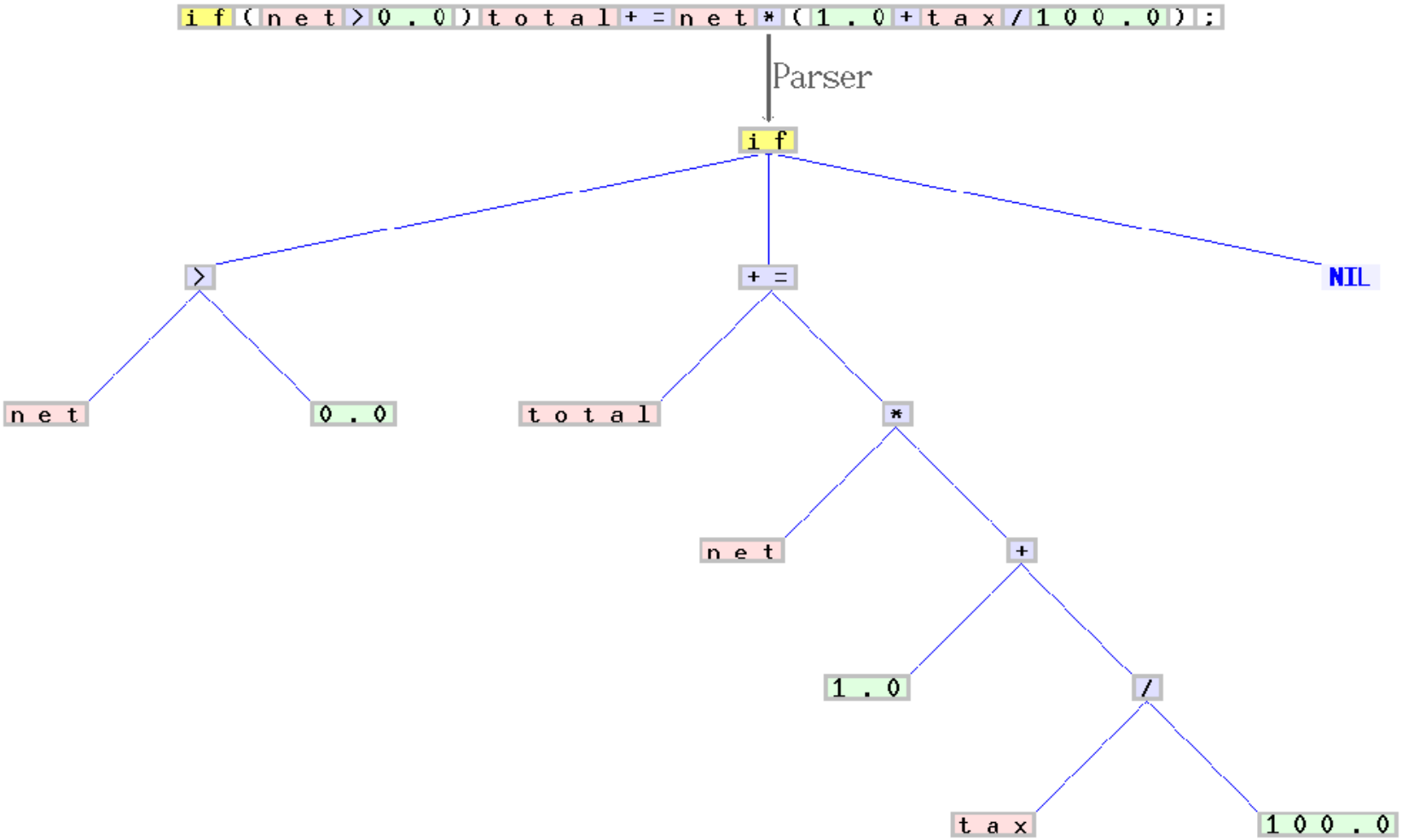
- ASD zadrži bistvena vozlišča sintaksnega drevesa in odvrže nepomembne podrobnosti.
  - Vsako vozlišče ustreza enemu ali večim vozliščem konkretnega sintaksnega drevesa.
- Dodatni podatki so pripeti na vozlišča
  - Za preverjanje tipov.
  - Za analize, ki se ne morejo implementirati med sintaksno analizo in preverjanjem tipov.
  - Za prevajanje na osnovi sintakse.

Primer: Abstraktno sintaksno drevo (1)




```
<expression> ::= <term> | <term> "+" <expression>
<term>       ::= <factor> | <term> "*" <factor>
<factor>     ::= <constant> | <variable> | "(" <expression> ")"
<variable>   ::= "x" | "y" | "z"
<constant>   ::= <digit> | <digit> <constant>
```

Primer: Abstraktno sintaksno drevo (2)



Vsebina

- Zgodovina
- Arhitektura prevajalnika
- Leksikalna analiza
- Slovnice & razčlenjevalnik
- Semantična analiza
- Vmesni jezik
- Generacija kode
- Optimizacija kode

Tabela simbolov

- Tabela simbolov vsebuje podatke o simbolih v programih.
  - Imena spremenljivk, tipov, funkcij, itd.
  - Podatki o simbolih so zbrani postopoma med fazo analize prevajalnika.
    - Uporabjajo se v fazi sinteze za generiranje ciljne kode.
  - Za vsak identifikator shranimo naslednje podatke.
    - Znakovni niz identifikatorja, tip, naslov v pomnilniku.
- Tabela simbolov je definirana za vsako definicijsko območje.
  - Imamo lahko različne simbole z istim imenom v različnih definicijskih območjih.
  - Definicijska območja so povezana hierarhično.

Preverjanje tipov

- Tipi kot abstraktni opisi vrednosti
  - Opis strukture in domen spremenljivk
- Preverjanje tipov je semantična analiza programa
  - Aspekt, ki ni pokrit s pravili slovnice
  - Preverjanje ali se tipi argumentov operacij ujemajo s tipom operacije
    - Najprej se izpelje tipe argumentov.
    - Potem se preveri kompatibilnost izpeljanih tipov; ni potrebno, da so tipi argumentov ekvivalentni tipom parametrov.
- Izpeljava tipov
  - Tipi izrazov so izpeljani iz tipov podizrazov.
    - Pogosto je uporabljeno ASD

Preverjanje tipov

  - Tipi v polimorfičnem sistemu tipov so izračunani kot rezultat sistema enačb.
- Izpeljava tipov in preverjanje tipov sta predstavljena na predavanju o tipih.
  - Enakost, konverzije in pretvorbe tipov
  - Izpeljava tipov
  - Preverjanje tipov na osnovi pravil
  - Izpeljava tipov v jezikih s polimorfizmom
- Tukaj si bomo pogledali primere pravil za preverjanje tipov izrazov.

Preverjanje tipov

- Primer
  - Preverjanje tipov jezika izrazov

$Check_{Exp}(Exp, vtable, ftable) =$ `case` Exp `of`	
num	`int`
id	$t = lookup(vtable, getname(\textbf{id}))$ *if* $t = unbound$ *then* **error**(); `int` *else* t
Exp_1 + Exp_2	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ *if* $t_1 =$ `int` *and* $t_2 =$ `int` *then* `int` *else* **error**(); `int`
Exp_1 = Exp_2	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ *if* $t_1 = t_2$ *then* `bool` *else* **error**(); `bool`
`if` Exp_1 `then` Exp_2 `else` Exp_3	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ $t_3 = Check_{Exp}(Exp_3, vtable, ftable)$ *if* $t_1 =$ `bool` *and* $t_2 = t_3$ *then* t_2 *else* **error**(); t_2
id ($Exps$)	$t = lookup(ftable, getname(\textbf{id}))$ *if* $t = unbound$ *then* **error**(); `int` *else* $((t_1, \ldots, t_n) \rightarrow t_0) = t$ $[t'_1, \ldots, t'_m] = Check_{Exps}(Exps, vtable, ftable)$ *if* $m = n$ *and* $t_1 = t'_1, \ldots, t_n = t'_n$ *then* t_0 *else* **error**(); t_0
`let` **id** = Exp_1 `in` Exp_2	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $vtable' = bind(vtable, getname(\textbf{id}), t_1)$ $Check_{Exp}(Exp_2, vtable', ftable)$

Semantična analiza

- Semantični analizator uveljavi vrsto pravil
  - Ta pravila se ne morejo preveriti med razčlenjevanjem (z uporabo kontekstno neovisnih slovnic) ali med preverjanjem tipov.
- Statična in dinamična semantična pravila
  - Statična pravila se preverijo v času prevajanja.
  - Za dinamična pravila se prevajalnik generira kodo, ki se preveri dinamično.

Semantična analiza

- Tipična statična pravila pokrita s semantičnim analizatorjem.
  - Vsak identifikator je definiran preden se uporabi.
  - Literali na rokah stavka switch so različne konstante (PJ C).
  - Nekateri jeziki dovoljujeo pretvorbo (angl. coerecion) vrednosti parametrov oziroma r-vrednosti pri prirejanju vrednosti spremenljivkam.
    - Vrednosti so automatično pretvorijo v vrednost pričakovanega tipa.
  - Vsaka funkcija, ki ima definiran tip rezultata (non-void) vrne vrednost eksplicitno.

Semantična analiza

- Primeri pravil, ki so uveljavljena v času izvajanja
  - Spremenljivke niso nikoli uporabljene, če jim ni bila prirejena vrednost.
  - Kazalci niso nikoli dereferencirani, če ne kažejo na veljaven objekt.
  - Vrednosti izrazov, ki predstavljajo indekse polj so v mejah polja.
  - Pri računanju aritmetičnih operacij ne pride do prelivanja.

Pred-vmesna koda

- Vsako vozlišče abstraktnega sintaksnega drevesa (pravilo KNS) je predstavljeno s ‘predlogo kode’
  - Razčlenjevalno drevo se transformira v linearno sekvenco inštrukcij (kodo);
  - Predloge (template) vozlišč se napolnijo z dejanskimi spremenljivkami in vrednostmi;
  - Koda vseh izrazov programa se združi v končno sekvenco instrukcij.
- Pred-vmesna koda
  - Oblika pred-vmesne kode
    - Sekvenca instrukcij
    - Označeno abstraktno sintaksno drevo

Vsebina

- Zgodovina
- Arhitektura prevajalnika
- Leksikalna analiza
- Slovnice & razčlenjevalnik
- Semantična analiza
- Vmesni jezik
- Generacija kode
- Optimizacija kode

Generiranje vmesne kode

- Na koncu bo moral biti program preveden v kodo konkretnega stroja.
- Prevajalniki običajno uporabljajo vmesne jezike.
  - Vmesni jeziki (okr. VJ) predstavljajo osnovo za prevajanje v strojne jezike.
- Prednosti:
  - Strukturiranje prevajalnika v manjša opravila
  - Več različnih visoko-nivojskih jezikov se lahko prevede v vmesni jezik
  - VJ se lahko prevede v različne ciljne arhitekture.
  - VJ lahko interpretiramo.
    - S programom, ki je implementiran na ciljni arhitekturi.

Generiranje vmesne kode

- Za prevod več različnih jezikov (N) v več različnih ciljnih arhitektur (M)
  - Direktno prevajanje: N*M
  - Vmesni jezik: N+M
- Tolmač vmesnega jezika
  - Uporabi jezik, ki je implementiran na veliko arhitekturah.
  - Isti tolmač lahko uporabimo za različne arhitekture.
  - Ista vmesna koda se lahko uporabi za vse konkretne stroje.
  - Vmesna oblika je lahko bolj kompaktna kot strojna koda.

Generiranje vmesne kode

- Slaba stran VJ je hitrost izvajanja
  - Interpretiranje bo (v večini primerov) precej počasnejše kot strojna koda
- Virtualni stroj Jave je velik uspeh.
- Prevajanje vmesne kode v strojni jezik med izvajanjem programa.
  - Just-in-time (JIT) prevajanje
  - Pogosto se uporablja za izvajanje vmesne kode za Javo
- Mi se bomo usmerili bolj v uporabo vmesne kode za prevajanje v strojne jezike,
  - Zaledni del prevajalnika.

Izbira vmesnega jezika

- Konfliktni cilji:
  - Enostavno prevajanje iz (različnih) visoko-nivojskih jezikov v vmesni jezik.
  - Enostavno prevajanje iz vmesnega jezika v raznolik nabor ciljnih arhitektur.
  - Vmesni jezik naj bo primeren za optimizacijo.
- Izbor nivoja jezika
  - Višje-nivojski jezik, večje breme na zalednem delu prevajalnika.
  - Nizko-nivojski vmesni jezik, večje breme na čelnem delu prevajalnika.

Izbor vmesnega jezika

- "Zrnatost" vmesnega jezika
  - Naj operacija operacija vmesnega jezika ustreza večji ali manjši količini dela?
- Kompleksne operacije
  - Pogosto uporabljene za tolmače (učinkovitost).
  - Preslikajo se v sekvenco strojnih inštrukcij.
- Preproste operacije
  - Sekvenca operacij se preslika na eno samo operacijo.

Vmesni jezik

- Bolj nizko-nivojski vmesni jezik
- Majhna zrnatost jezika

Program → [*Instructions*]

Instructions → *Instruction*
Instructions → *Instruction* , *Instructions*

Instruction → `LABEL` **labelid**
Instruction → **id** := *Atom*
Instruction → **id** := **unop** *Atom*
Instruction → **id** := **id binop** *Atom*
Instruction → **id** := *M*[*Atom*]
Instruction → *M*[*Atom*] := **id**
Instruction → `GOTO` **labelid**
Instruction → `IF` **id relop** *Atom* `THEN` **labelid** `ELSE` **labelid**
Instruction → **id** := `CALL` **functionid**(*Args*)

Atom → **id**
Atom → **num**

Args → **id**
Args → **id** , *Args*

Sintaksno usmerjeno prevajanje

- Generiranje kode s prevajalno funkcijo za vsako sintaktično kategorijo.
  - Sintaktična kategorija je definirana za vsako slovnično pravilo
  - Parametri prevajalne funkcije vsebujejo podatke o kontekstu (npr., tabela simbolov)
- Koda se generira lokalno za dano sintaktično kategorijo
  - Prevajanje ni optimalno glede na povezane kategorije.
  - Optimizacija (predstavljena kasneje) izloči nepotrebne spremenljivke, skoke, branja, itd.

Prevajanje izrazov

$$
\begin{array}{lcl}
Exp & \rightarrow & \textbf{num} \\
Exp & \rightarrow & \textbf{id} \\
Exp & \rightarrow & \textbf{unop}\ Exp \\
Exp & \rightarrow & Exp\ \textbf{binop}\ Exp \\
Exp & \rightarrow & \textbf{id}(Exps) \\
 & & \\
Exps & \rightarrow & Exp \\
Exps & \rightarrow & Exp\ ,\ Exps
\end{array}
$$

$Trans_{Exp}(Exp, vtable, ftable, place) =$ `case` Exp `of`	
num	$v = getvalue(\textbf{num})$ $[place := v]$
id	$x = lookup(vtable, getname(\textbf{id}))$ $[place := x]$
unop Exp_1	$place_1 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, place_1)$ $op = transop(getopname(\textbf{unop}))$ $code_1 ++ [place := op\ place_1]$
Exp_1 **binop** Exp_2	$place_1 = newvar()$ $place_2 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, place_1)$ $code_2 = Trans_{Exp}(Exp_2, vtable, ftable, place_2)$ $op = transop(getopname(\textbf{binop}))$ $code_1 ++ code_2 ++ [place := place_1\ op\ place_2]$
id$(Exps)$	$(code_1, [a_1, \ldots, a_n]) = Trans_{Exps}(Exps, vtable, ftable)$ $fname = lookup(ftable, getname(\textbf{id}))$ $code_1 ++ [place :=$ `CALL` $fname(a_1, \ldots, a_n)]$

$Trans_{Exps}(Exps, vtable, ftable) =$ `case` $Exps$ `of`	
Exp	$place = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, place)$ $(code_1, [place])$
$Exp\ ,\ Exps$	$place = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, place)$ $(code_2, args) = Trans_{Exps}(Exps, vtable, ftable)$ $code_3 = code_1 ++ code_2$ $args_1 = place :: args$ $(code_3, args_1)$

Prevajanje stavkov

$$
\begin{array}{lcl}
\mathit{Stat} & \rightarrow & \mathit{Stat}\ ;\ \mathit{Stat} \\
\mathit{Stat} & \rightarrow & \textbf{id} := \mathit{Exp} \\
\mathit{Stat} & \rightarrow & \texttt{if}\ \mathit{Cond}\ \texttt{then}\ \mathit{Stat} \\
\mathit{Stat} & \rightarrow & \texttt{if}\ \mathit{Cond}\ \texttt{then}\ \mathit{Stat}\ \texttt{else}\ \mathit{Stat} \\
\mathit{Stat} & \rightarrow & \texttt{while}\ \mathit{Cond}\ \texttt{do}\ \mathit{Stat} \\
\mathit{Stat} & \rightarrow & \texttt{repeat}\ \mathit{Stat}\ \texttt{until}\ \mathit{Cond} \\
 & & \\
\mathit{Cond} & \rightarrow & \mathit{Exp}\ \textbf{relop}\ \mathit{Exp}
\end{array}
$$

$Trans_{Stat}(Stat, vtable, ftable) = \texttt{case}\ Stat\ \texttt{of}$	
$Stat_1\ ;\ Stat_2$	$code_1 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_2, vtable, ftable)$ $code_1 ++ code_2$
$\textbf{id} := Exp$	$place = lookup(vtable, getname(\textbf{id}))$ $Trans_{Exp}(Exp, vtable, ftable, place)$
$\texttt{if}\ Cond$ $\texttt{then}\ Stat_1$	$label_1 = newlabel()$ $label_2 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_1, label_2, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_1 ++ [\texttt{LABEL}\ label_1] ++ code_2$ $\quad ++ [\texttt{LABEL}\ label_2]$
$\texttt{if}\ Cond$ $\texttt{then}\ Stat_1$ $\texttt{else}\ Stat_2$	$label_1 = newlabel()$ $label_2 = newlabel()$ $label_3 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_1, label_2, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_3 = Trans_{Stat}(Stat_2, vtable, ftable)$ $code_1 ++ [\texttt{LABEL}\ label_1] ++ code_2$ $\quad ++ [\texttt{GOTO}\ label_3, \texttt{LABEL}\ label_2]$ $\quad ++ code_3 ++ [\texttt{LABEL}\ label_3]$
$\texttt{while}\ Cond$ $\texttt{do}\ Stat_1$	$label_1 = newlabel()$ $label_2 = newlabel()$ $label_3 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_2, label_3, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_1, vtable, ftable)$ $[\texttt{LABEL}\ label_1] ++ code_1$ $\quad ++ [\texttt{LABEL}\ label_2] ++ code_2$ $\quad ++ [\texttt{GOTO}\ label_1, \texttt{LABEL}\ label_3]$
$\texttt{repeat}\ Stat_1$ $\texttt{until}\ Cond$	$label_1 = newlabel()$ $label_2 = newlabel()$ $code_1 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond, label_2, label_1, vtable, ftable)$ $[\texttt{LABEL}\ label_1] ++ code_1$ $\quad ++ code_2 ++ [\texttt{LABEL}\ label_2]$

Vsebina

- Zgodovina
- Arhitektura prevajalnika
- Leksikalna analiza
- Slovnice & razčlenjevalnik
- Semantična analiza
- Vmesni jezik
- Generacija kode
- Optimizacija kode

Generiranje strojne kode

- Izbrani vmesni jezik je precej nizko-nivojski
  - Podoben je strojni kodi modernih RISC procesorjev
  - Uporabljali bomo RISC MIPS (RISC V)
- Pogosto imamo enolično preslikavo iz VJ v nabor instrukcij RISC
  - Kompleksne RISC (MIPS) instrukcije so preslikane v vzorce inštrukcij VJ
  - In vice versa.
- Problemi pri prevajanju
  - Razlike v naboru inštrukcij
  - Zaseganje registrov
  - Klicne sekvence funcij

Generiranje strojne kode

- Razlike med operacijami RISC procesorja in operacijami vmesnega jezika.
  1) IF-THEN-ELSE instrukcija ima dve ciljni oznaki.
     - Instrukcija pogojni skok ima eno samo ciljno oznako.
  2) Poljubna konstanta je lahko operand inštrukcije .
     - RISC procesorji dovoljujejo majhne konstante kot operande.
  3) RISC procesorji (MIPS, ARM) imajo nekatere kompleksne operacije.
  4) VJ uporablja neomejeno število spremenljivk.
     - Procesor ima omejeno število registrov.
  5) VJ uporablja kompleksno operacijo za klic funkcije (CALL)

Pogojni skoki

- Inštrukcije pogojni skok imajo več različnih oblik na različnih procesorjih.
  - Preverjanje relacije med registri (if-then-else)
  - Instrukcije pogojni skok uporabljajo en sam ciljni naslov.
    - if c a1; jp a2
  - Pogosto tej inštrukciji sledi en izmed ciljnih naslovov of target addresses (poglej pravilo vmesnega jezika za IF)
    - Generator preveri kaj sledi
  - Koda se generira za pogoj; rezultat se shrani
    - V splošno namenske registre (MIPS, Alpha) + v poseben register (IA-64, PowerPC), + v zastavice (Sparc, IA-32) ...

IF c THEN l_t ELSE l_f

```
branch_if_c      l_t
jump             l_f
```

```
branch_if_not_c  l_f
```

Konstante

- VJ omogoča uporabo poljubne konstante kot operanda binarnih in unarnih operacij
  - Ne tudi v MIPS, ARM, …
  - Več inštrukcij je potrebno za izvedbo ene primerjalne operacije
- Generatorji kode morajo preveriti, če se konstanta lahko uporabi s katero izmed strojnih inštrukcij
  - Če se potem generator uporabi za prevod eno samo strojno instrukcijo
  - Če ne,
    1) Sekvenca inštrukcij zgradi konstanto v registru,
    2) Izbrana inštrukcija uporabi register namesto konstante

Kompleksne inštrukcije

- Večina inštrukcij v našem VJ je atomičnih
  - Uporaba RISC MIPS (RISC 5), ARM (mobilne naprave)
  - Vsaka inštrukcija VJ ustreza eni ali večim strojnim inštrukcijam
- Kompleksne RISC operacije (MIPS, ARM)
  - Preslikajo se v sekvenco inštrukcij VJ
- Primer:

$$t_2 := t_1 + 116$$
$$t_3 := M[t_2]$$

```
lw r3, 116(r1)
```

Pari vzorec/ zamenjava za podmožico nabora RISC inštrukcij

$t := r_s + k$, $r_t := M[t^{last}]$		`lw`	r_t, $k(r_s)$
$r_t := M[r_s]$		`lw`	r_t, $0(r_s)$
$r_t := M[k]$		`lw`	r_t, k(R0)
$t := r_s + k$, $M[t^{last}] := r_t$		`sw`	r_t, $k(r_s)$
$M[r_s] := r_t$		`sw`	r_t, $0(r_s)$
$M[k] := r_t$		`sw`	r_t, k(R0)
$r_d := r_s + r_t$		`add`	r_d, r_s, r_t
$r_d := r_t$		`add`	r_d, R0, r_t
$r_d := r_s + k$		`addi`	r_d, r_s, k
$r_d := k$		`addi`	r_d, R0, k
`GOTO` *label*		`j`	*label*
`IF` $r_s = r_t$ `THEN` $label_t$ `ELSE` $label_f$, `LABEL` $label_f$		`beq`	$r_s, r_t, label_t$
	$label_f$:		
`IF` $r_s = r_t$ `THEN` $label_t$ `ELSE` $label_f$, `LABEL` $label_t$		`bne`	$r_s, r_t, label_f$
	$label_t$:		
`IF` $r_s = r_t$ `THEN` $label_t$ `ELSE` $label_f$		`beq`	$r_s, r_t, label_t$
		`j`	$label_f$
`IF` $r_s < r_t$ `THEN` $label_t$ `ELSE` $label_f$, `LABEL` $label_f$		`slt`	r_d, r_s, r_t
		`bne`	r_d, R0, $label_t$
	$label_f$:		
`IF` $r_s < r_t$ `THEN` $label_t$ `ELSE` $label_f$, `LABEL` $label_t$		`slt`	r_d, r_s, r_t
		`beq`	r_d, R0, $label_f$
	$label_t$:		
`IF` $r_s < r_t$ `THEN` $label_t$ `ELSE` $label_f$		`slt`	r_d, r_s, r_t
		`bne`	r_d, R0, $label_t$
		`j`	$label_f$
`LABEL` *label*	*label*:		

Zaseganje registrov

- Ko generiramo kodo VJ lahko uporabljamo poljubno število spremenljivk
  - Procesorji nimajo neomejeno število registrov
- Potrebujemo postopek za zaseganje registrov
  - Preslikava velikega števila spremenljivk v majhno množico registrov
  - Dovolimo, da si več spremenljivk deli isti register
  - Včasih nimamo zadosti registrov v procesorju
- To imenujemo ’prelivanje’
  - Nekatere spremenljivke morajo biti začasno shranjene v spomin

Zaseganje registrov

- Kdaj si lahko dve spremenljivke delijo register?
- Živost (liveness) spremenljivke
  - Vrednost, ki jo vsebuje se bo uporabljala v prihodnosti
    - Formalna definicija (preko sprememb stanj)
- Alokacija registrov z barvanjem grafov
  - Graf sovpada (interference) spremenljivk
    - Dve spremenljivke sta povezani, če med njima obstaja sovpad
    - Vozlišča s skupno povezavo imata različni številki registrov
    - Številke registrov ne smejo biti večje kot število obstoječih reg.
    - Sicer morajo biti spremenljivke shranjene v spomin
  - NP kompleten problem

Klici funkcij

- Funkcijski kljic nismo obravnavali pri prevodu funkcij.
- Uporaba skladovnega pomnilnika
  - Ko je funkcija poklicana se vse žive spremenljivke shranijo (iz registrov) v spomin
    - Sklad se uporablja kot začasna shramba
  - Registri so zdaj prosti za uporabo v klicani funkciji
  - Sklad je uporabljen tudi za aktivacijske zapise funkcij
    - Spremenljivke in parametri klicane funkcije
    - Kontrolni podatki; naslov vrnitve, definicijsko območje, itd.
  - Aktivacijski zapisi bodo predstavljeni na predavanju o delu s spominom v PJ

Klici funkcij

- Kaj se zgodi pri klicu funkcije?
  - Prolog, epilog in klicne sekvence
    - Delo z registri, živimi spremenlj. parametri in aktivacij. zapisi
    - Kaj se zgodi pri klicu in kaj pri vračanu funkcij?
  - Kdo shrani registre? Funkcija iz katere se izvrši klic ali klicana funkcija?
    - Bolj kompleksno, če izvršitelj klica shranjuje žive spremenljivke in klicana funkcija shranjuje svoje spremenljivke.
  - Dostop do globalnih spremenljivk
    - Delo z definicijskimi območji preko aktivacijskih zapisov
  - Uporaba registrov za prenos parametrov
  - Interakcija z zaseganjem registrov

Vsebina

- Zgodovina
- Arhitektura prevajalnika
- Leksikalna analiza
- Slovnice & razčlenjevalnik
- Semantična analiza
- Vmesni jezik
- Generacija kode
- Optimizacija kode

Analiza in optimizacija

- Prepoznavanje specifičnih vzorcev v programu.
- Zamenjava prepoznanih vzorcev z manjšimi in/ali hitrejšimi vzorci.
  - Zamenjava sekvenc inštrukcij z drugimi sekvencami
  - Lahko apliciramo na vmesni jezik ali strojno kodo
- Optimizacija s kukalnim oknom (peephole)
  - Na kode gledamo preko majhnega okna.
  - Vidimo samo sekvence inštrukcij.
  - Bolj globalne lastnosti zahtevajo pogled na poljubno velik kontekst!

Optimizacija s kukalnim oknom

- Izogibanje odvečnim inštrukcijam load and store
- Izločitev nedostopne kode

```
LD a, R0
ST R0, a
```

```
    if debug == 1 goto L1
    goto L2
L1: print debugging information
L2:
```

- Optimizacija kontrole izvajanja
  - Večkratni skoki

```
  goto L1
  ...
L1: goto L2
```

- Poenostavitve operacij in redukcija v izrazni moči
  - Izloči stavke s tremi naslovi
  - Zamenjaj drage operacije s cenejšimi (x^2=x*x)

```
x = x + 0
x = x * 1
```

Analize podatkovnih tokov

- Odkrivanje in uporaba informacijskih tokov po programih
  - Prepoznavanje vzorcev v danem kontekstu
  - Npr. analiza živosti spremenljivk za vsak inštrukcijo
    - Imamo **in** in **out** množice spremenljivk za vsako inštrukcijo
- Podatki potrebni pri neki konkretni analizi
  - Predstavljajo en vidik izvajanja programa
  - Različni tipi analiz zahtevajo različne tipe in obravnave podatkovnih tokov

Analize podatkovnih tokov

- Analiza podatkovnih tokov za optimizacijo
  - Zamenjava skvence inštrukcij z drugo sekvenco
  - Npr., analiza živosti spremenljivk lahko prispeva k učinkovitosti zaseganja registrov
- Vzvratna in vnaprejšnja analiza
  - Analiza živosti je vzvratna analiza
  - Podatkovni tokovi tečejo nazaj: od uporabe proti prireditvam vrednosti spremenljivkam
- Nekateri primeri analiz podatkovnih tokov bodo predstavljeni v nadaljevanju

Primeri analize podatkovnih tokov

- Izločevanje mrtve kode
  - False veje pogojev
- Širjenje konstante
  - Ceneje je zaseganje fiksnega bloka spomina za konstante na začetku
- Skupni podizrazi
  - Identificirani in izračunani samo enkrat (postavljene so reference na vrednosti)
- Eliminacija večkratnih skokov
  - Sekvenca skokov je zamenjana z direktnim skokom

Primeri analize podatkovnih tokov

- Indeksi polj se preverijo samo enkrat
  - Nepotrebna preverjanja se odstrani.
- Transformacija zank
  - Vnaprejšnje branje spomina
  - Obrat: zamenjava notranje in zunanje zanke. Izboljšamo lokalnost referenc.
  - Vektorizacija: namera po izvajanju čim večjega števila iteracij hkrati.
  - Dviganje kode: premik invariante zanke izven zanke.

Primeri analize podatkovnih tokov

- Klici funkcij
  - Razširitev z vrivanjem: vstavljanje kode je cenejše od klica funkcije
  - Optimizacija na repu: transformiraj rekurzijo na repu v iteracijo
  - Specializacija: specializiraj specifične klice funkcij z odstranjevanjem nepotrebne kode
- Automatična paralelizacija
  - Procesorji imajo več jeder, neodvisni kodni segmenti se lahko izvajajo vzporedno