Chapter 8 Code Generation

- Study Goals
  - Master

    Intermediate code generation for basic structures
  - Understand

    Three-address code; Triple; Quadruple
  - Know

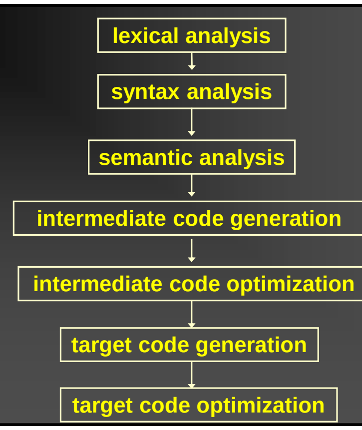
Overview of Code Generation

- The task of code generation is to generate executable code for a target machine that is a faithful representation of the semantics of the source code
- Code generation depends on
  - The characters of the source language
  - Detailed information about the target architecture
  - The structure of the runtime environment

Code generation is typically broken into several steps

1) Intermediate code generation
2) Generate some form of assembly code instead of actual executable code
3) Optimization

   To improve the speed and size of the target code



- Although a source program can be translated directly into the target language, some benefits of using a machine-independent intermediate code are:
  - Retargeting is facilitated: a compiler for a different machine can be created by attaching a back end for the new machine
  - A machine-independent code optimizer can be applied to the intermediate code

We will talk about general techniques of code generation rather than present a detailed description for a particular target machine

8.1 Intermediate Code and Data Structures for Code Generation

- Intermediate Representation(IR)
  - A data structure that represents the source program during translation is called an IR
  - For example: abstract syntax tree

- 8.1.1 Intermediate code
  - The need for intermediate code
    Abstract syntax tree does not resemble target code, particularly in its representation of control flow constructs
  - Intermediate code
    Representation of the syntax tree in sequential form that more closely resembles target code

- Advantage of intermediate code
  - It is particularly useful when the goal of the compiler is to produce extremely efficient code
  - It can also be useful in making a compiler more easily retargetable
- Popular forms of intermediate code
  Three-address code

intermediate code

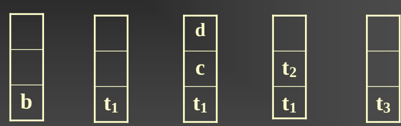
- Reverse Polish notation
- triples
- quadruples

Reverse Polish notation/ Postfix expression

expression in the language	RPN
a+b	ab+
a+b*c	abc * +
(a+b)*c	ab+c *

$-b+c*d$

b@cd*+



$t_1= - b$ $t_2= c*d$ $t_3= t_1+t_2$

triples

(op，ARG1,ARG2)

a :=b*c+b*d

(1) (* , b, c)

(2) (* , b, d)

(3) (+ , (1), (2))

(4) (:=, (3), a)

quadruples

(op，ARG1，ARG2，RESULT）

a:=b*c+b*d

1) (*, b, c, t_1)

2) (*, b, d, t_2)

3) (+, t_1, t_2, t_3)

4) (:=, t_3, —, a)

t_i（i=1,2,3） tempory variables

quadruples

quadruples	Intuitive form
（1） （ * , b , c , t_1）	（1） t_1:=b*c
（2） （ * , b , d , t_2）	（2） t_2:=b*d
（3） （ +, t_1 , t_2 , t_3）	（3） t_3:=t_1+t_2
（4） （:=, t_3 ,—, a）	（4） a:=t_3
(jump,—， —， L)	goto L
(jrop, B， C， L)	if B rop C goto L

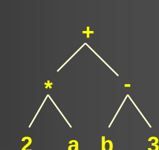
8.1.2 Three-Address Code

- The most basic instruction of three address code has the general form x=y op z
  - Which represents the evaluation of expressions
  - x,y,z are names, constants or compiler-generated temporary names
  - op stands for any arithmetic or logical operator, such as + ,'and'
  - "Three-address code" comes from this form of instruction, in general each of x,y and z represents an address in memory

Example

Computation of an expression is represented in three-address code

2*a+(b-3)



the corresponding

three-address code is

t1 = 2*a

t2 = b-3

t3 = t1+t2

where t1,t2,t3 are names for temporaries, they correspond to the interior nodes of the syntax tree and represent their computed values

Other instructions of three-address code

Three-address code for each construction of a standard programming language

1. Assignment statement has the form
   "x=y op z", where op is a binary operation
2. Assignment statement has the form
   "x=op y", where op is a unary operation
3. Copy statement has the form
   "x=y" where the value of y is assigned to x

4. The unconditional jump "goto L"
5. Conditional jumps ,such as "if B goto L" , "if_false B goto L" and "if A rop B goto L"
6. Statement "Label L" represents the position of the jump address
7. "read x"
8. "write x"
9. Statement "halt" serves to mark the end of the code

Example	Three-address code for it	
Sample TINY program	read x	
read x;	t1=0<x	
If 0<x then	if-false t1 goto L1	
fact:=1;	fact=1	
repeat	label L2	t4=x==0
fact:=fact*x;	t2=fact*x	if_false t4 goto L2
x:=x-1;	fact=t2	write fact
until x=0;	t3=x-1	Label L1
write fact	x=t3	halt
end		

8.2 Basic Code Generation Techniques

- Basic approaches to code generation in general (8.2)
- Code generation for individual language constructs (8.3)

Bottom-up Syntax-directed translation

1 Simple assignment statement translation
2 Boolean expression translation
3 Control statement translation
4 Declaration statement translation

1 Simple assignment statement translation

- Sementic analysis：
  1. Is id declared？
  2. operands's data type
  3. l = r
- translation object
  assignment quadruples

1-1. Variables, Procedures and functions in attributes and sementic rules

Attributes:

- id.name: token id's name
- E.place: the address of E in the Symbol table

Variables, functions, procedures:

- nextstat: the order of the next quadruple
- lookup（id.name）: check if id.name exits in the symbol table. If yes, return id's address, else return nil
- emit(): output a quadruple, nextstat+1
- newtemp(): temporaries, t_1, t_2 , ……
- error()

1-2. Translation（one data type）

Is id decared?

(1) S→id:＝E

{ p:＝lookup (id.name) ;
if p≠nil then emit (:＝, E.place , - , p)
else error }

(2) $E \to E^1 + E^2$

{ E.place:＝newtemp ;
emit (+ , E^1.place , E^2.place , E.place) }

(3) $E \to E^1 * E^2$

{ E.place:＝newtemp ;
emit (* , E^1.place , E^2.place , E.place) }

(4) $E \to -E^1$

{ E.place:＝newtemp ;
emit (@ , E^1.place , - , E.place) }

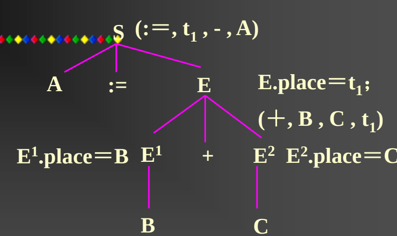
(5) $E \to (E^1)$

{ E.place:＝E^1.place }

(6) E→id

{ p:＝lookup (id.name) ;
if p≠nil then E.place:＝p
else error }

A:＝B+C



1-3 Translation（different data types）

- Sementic analysis:
  1. Is id declared?
  2. Operands' data type?
- id can be int or real.
- E.type: int or real。

$+^i$, $*^i$, $+^r$, $*^r$

itr: int -> real

$E \to E^1 + E^2$:

E.place:＝newtemp ;
if E^1.type＝int AND E^2.type＝int then
begin emit ($+^i$, E^1.place , E^2.place , E.place) ; E.type:＝int end
else if E^1.type＝real AND E^2.type＝real then
    begin emit ($+^r$, E^1.place , E^2.place , E.place) ;
        E.type:＝real
    end
  else if E^1.type＝int then
      begin t:＝newtemp ; emit (itr , E^1.place , - , t) ;
          emit ($+^r$, t , E^2.place , E.place) ; E.type:＝real
      end
      else begin t:＝newtemp ; emit (itr , E^2.place , - , t) ;
          emit ($+^r$, E^1.place , t , E.place) ; E.type:＝real
          end ;

- Attribute grammar
- Attributes：
- Rules：sementic rules, intermediate codes

2 Boolean expression translation

1. Boolean expression

- function：
  - the logic value
  - as the condition of control statement(if-then,while)
- grammar：

<BE>→<BE> or <BE> |<BE> and <BE> | not <BE> | (<BE>) |<RE>| true | false

<RE>→<AE> relop <AE> | (<RE>)

<AE>→<AE> op <AE> | -<AE> | (<AE>) | id | num

relop:(<=, < , =,≠, > , >=)

op: (+ , - , * , /)

E→E or E | E and E | not E | (E) | id rop id |true|false

- boolean operator: not>and>or
- arithmetic operator

> relational operator>boolean operator

2. Function1 of boolean expression：

direct computation & short-circuit computation

- direct computation

1 or 0 and 1=1 or 0=1

- short-circuit computation

A or B	if A then 1 else B
A and B	if A then B else 0
not A	if A then 0 else 1

3. translation of direct computation

A or B and not C

(	not,	C,	- ,	t1)
(	and,	B,	t1,	t2)
(	or,	A,	t2,	t3)

a<b:

(1) (	j<,	a,	b,	(4))
(2) (	:=,	0,	- ,	t_1)
(3) (	jump,	- ,	- ,	(5))
(4) (	:=,	1,	- ,	t_1)
(5)	…			

Translation of direct computation

(1)E→E^1 or E^2 { E.place :＝newtemp ;
emit (or , E^1.place , E^2.place , E.place) }

(2)E→E^1 and E^2 { E.place :＝newtemp ;
emit (and , E^1.place , E^2.place , E.place) }

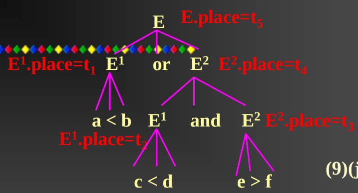
(3)E→not E^1 { E.place :＝newtemp ;
emit (not , E^1.place ,—, E.place) }

(4)E→(E^1) { E.place :＝E^1.place }

(5)E→id_1 rop id_2 { E.place :＝newtemp ;
emit (jrop , id_1.place , id_2.place , nextstat+3) ;
emit (:＝, 0 ,－, E.place) ;
emit (jump ,—,—, nextstat+2) ;
emit (:＝, 1 ,－, E.place) }

(6)E→true { E.place:＝newtemp;emit(:=,1,- ,E.place) }

(7)E→false {E.place:=newtemp;emit(:=,0,- ,E.place)} a<b or c<d and e>f

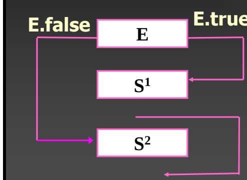


(1)(j<, a, b, (4))
(2)(:=, 0, - , t_1)
(3)(jump,- ,- ,(5))
(4)(:=, 1, - , t_1)

(5)(j<, c, d, (8))
(6)(:=, 0, - , t_2)
(7)(jump, - , - ,(9))
(8)(:=, 1, - , t_2)

(9)(j>, e, f, (12))
(10)(:=, 0, - , t_3)
(11)(jump,- ,- ,(13))
(12)(:=, 1, - , t_3)
(13)(and, t_2, t_3, t_4)
(14)(or, t_1, t_4, t_5)

4. Translation as the condition



if E then S^1 else S^2

- E.true:
- E.false:

E will be translated to

1. (jnz , A , - , p)
2. (jrop , A , B , p)
3. (jump , - , - , p)

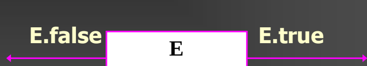
if A or B<D then S^1 else S^2

(1)	(jnz , A , - , 5)	A.true is 5
(2)	(jump , - , - , 3)	A.false is 3
(3)	(j< , B , D , 5)	B<D true 5
(4)	(jump , - , - , p+1)	B<D false (p+1)
(5)	(S^1......)	
(p)	(jump , - , - , q)	jump over S^2
(p+1)	(S^2....)	
(q)		

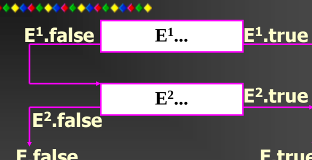
(1) - (4): A or B<D

- E: a rop b

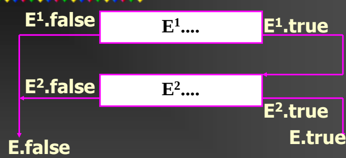
(jrop , a , b , E.true)
(jump ,—,—, E.false)



- E: E^1 or E^2 :



- E: E^1 and E^2



- E:not E^1

E : a<b or c<d and e>f

(1)	(	j<,	a,	b,	E.true)
(2)	(	jump,	- ,	- ,	(3))
(3)	(	j<,	c ,	d ,	(5))
(4)	(	jump,	- ,	- ,	E.false)
(5)	(	j>,	e ,	f ,	E.true)
(6)	(	jump,	- ,	- ,	E.false)

- Chaining and backpach of true and false

---

if <u>a<b or c<d and e>f</u> then S^1 else S^2

(1)	(j< ,	a ,	b ,	(7))
(2)	(jump, - ,		- ,	(3))
(3)	(j< ,	c ,	d ,	(5))
(4)	(jump, - ,		- ,	(p+1))
(5)	(j> ,	e ,	f ,	(7))
(6)	(jump, - ,		- ,	(p+1))
(7)	(S^1....)			
……				
(p)	(jump, - ,		- ,	q)
(p+1)	(S^2.....)			
……				
(q)				

- Chaining:

(10)…… goto E.true
……
(20)…… goto E.true
……
(30)…… goto E.true

(10)…… goto (0)
……
(20)…… goto (10)
……
(30)…… goto (20)

- (30) head of the chain, (10) tail of the chain
  0 is the label of chain tail
- E.true , E.false head of the true chain , the false chain

---

1) **merge (p_1, p_2) : P_1** and **p_2** are merged, return the head.
   if **P_2** is nil, return **P_1**, else the four section of **P_2** is modified to **P_1, return P_2**。
2) **backpatch (p , t) :**
3) **E.codebegin: the first order of E's quatruple**

- bottom - up parsing: boolean expression

1) E→E^1 or E^2

{ E.codebegin:=E^1.codebegin ;
backpatch (E^1.false , E^2.codebegin) ;
E.true:=merge (E^1.true , E^2.true) ;
E.false:=E^2.false }

2) E→E^1 and E^2

{ E.codebegin:=E^1.codebigin ;
backpatch (E^1.true , E^2.codebegin) ;
E.true:=E^2.true ;
E.false:=merge (E^1.fasle , E^2.false) }

3) E→not E^1

{ E.codebegin:=E^1.codebigin ;
E.true:=E^1.false ;
E.false:=E^1.true }

4) E→(E^1)

{ E.codebegin:=E^1.codebegin ;
E.true:=E^1.true ;
E.false:=E^1.false }

5) E→id_1 rop id_2

{ E.codebegin:=nextstat ;
E.true:=nextstat ;
E.false:=nextstat+1;
emit (jrop , id_1.place , id_2.place , 0) ;
emit (jump ,—,—, 0) }

6) E→true

{ E.codebegin:=nextstat ;
E.true:=nextstat ;
E.false:=0;
emit (jump ,—,—,0) }

7) E→false

{ E.codebegin:=nextstat ;
E.false:=nextstat ;
E.true:=0;
emit (jump ,—,—,0) }

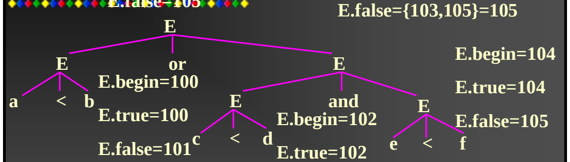
Translation

a<b or c<d and e<f

start from 100
nextstat=100

E.begin=100
E.true={100,104}=104
E.false=105

E.begin=102
E.true=104
E.false={103,105}=105



100 : (j< , a , b , 0)
101: (jump ,—,—,0)
102: (j< , c , d , 0)
103: (jump ,—,—, 0)
104: (j< , e , f , 0)
105: (jump ,—,—,0)

101(jump,—,—,102)
102 (j< , c , d ,104)
104 (j< , e , f, 100)
105 (jump ,—,—,103)

100: (j< , a , b , 0)
101: (jump,—,—,102)
102: (j< , c , d , 104)
103: (jump,—,—, 0)
104: (j< , e , f , 100)
105: (jump,—,—,103)
E.true=104 , E.false=105。

3. Translation of control statement

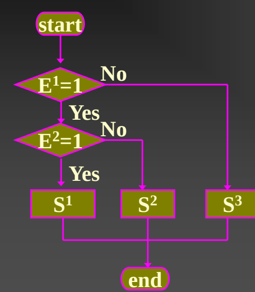
- if， while

S→　if E then S
　| if E then S else S
　| while E do S
　| begin L end
　| A

A →id:=E

L→ L ; S
　| S

if E^1 then if E^2 then S^1 else S^2 else S^3

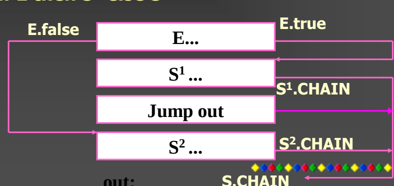


S : S.CHAIN

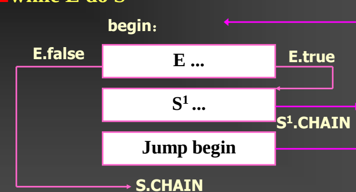
1. code structure

- if E then S^1

- if E then S^1 else S^2



- while E do S^1



2 Modified grammar

- reason：sementic actions could be executed.
- Separate the grammar

1) 把 S→if E then S^1
　C→if E then　(backpach E.true)
　S→C S^1

2) S→if E then S^1 else S^2
　C→if E then　(backpach E.true)
　T^p→C S^1 else　(jump,　backpach E.false)
　S→T^p S^2

3) S→while E do S^3
　W→while　(remember the position)
　W^d→W E do (backpach E.true)
　S→ W^d S^3

4) L→L ; S
　L^s→L ;　(backpach chain)
　L→L^s S

S→ if E then S

S→ if E then S else S

S→ while E do S

S→ begin L end

S→ A

L→ L ; S

L→ S

modified grammar

(1) S→ C S^1

(2) S→ T^p S^2

(3) S→ W^d S^3

(4) S→ begin L end

(5) S→ A

(6) L→ L^s S

(7) L→ S

(8) C→ if E then

(9) T^p→ C S^1 else

(10) W→ while

(11) W^d→ W E do

(12) L^s→ L ;

3. actions

C→if E then

{ backpatch (E.true , nextstat) ;

C.CHAIN:=E.false }

S→C S^1 /* if E then S^1 */

{ S.CHAIN:=merge (C.CHAIN , S^1.CHAIN) }

T^p→C S^1 else /* if E then S^1 else */

{ q:=nextstat ;

emit (jump, —, —, 0) ; //S^1 is executed，jump out of if

backpatch (C.CHAIN , nextstat) ;

T^p.CHAIN:=merge (q , S^1.CHAIN) }

S→T^p S^2 /* if E then S^1 else S^2 */

{ S.CHAIN:=merge (T^p.CHAIN , S^2.CHAIN) }

W→while

{ W.codebegin:=nextstat }

W^d→W E do /*while E do*/

{ W^d.codebegin:=W.codebegin ;

backpatch (E.true , nextstat) ;

W^d.CHAIN:=E.false }

S→ W^d S^3 /*while E do S^3 */

{ backpatch (S^3.CHAIN , W^d.codebegin) ;

emit (jump , —, —, W^d.codebegin) ;

/*S^3 is executed，jump to the beginning of While*/

S.CHAIN:=W^d.CHAIN) }

S→ begin L end

{ S.CHAIN:=L.CHAIN }

S→ A

{ S.CHAIN:=0 } /* assignment statement */

L→ S

{ L.CHAIN:=S.CHAIN }

L^s→L ;

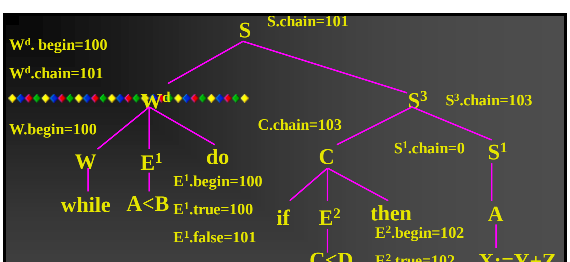
{ backpatch (L.CHAIN , nextstat) }

L→L^s S /* L;S */

{ L.CHAIN:=S.CHAIN }

Examle： while A<B do <u>if C<D then X:=Y+Z</u>

nextstat=100



100 (j< , A , B , 0) ⇒ 100 (j< , A , B ,102)

101 (jump, —, —,0)

102 (j< , C , D , 0) ⇒ 102 (j< , C , D , 104)

103 (jump, —, —,0) ⇒ 103 (jump, —, —, 100)

104 (+, Y , Z , t_1)

105 (:=, t_1 , —, X) 106 (jump, —, —, 100)

while A<B do <u>if C<D then X:=Y+Z</u>

100 (j< , A , B , 102)

101 (jump , —, —, 0)

102 (j< , C , D , 104)

103 (jump , —, —,100)

104 (+, Y , Z , t_1)

105 (:=, t_1 , —, X)

106 (jump , —, —,100)

S.CHAIN=101

while A<B do <u>if C<D then X:＝Y+Z</u> ：

(1) while is reduced to W，remember the position of while

(2) A<B is reduced to E^1：

100 (j< , A , B , 0)　　E^1.true＝100

101 (jump,－,－,0)　　E^1.false＝101

(3) W E do is reduced to W^d，E^1.true is backpached

100 （j< , A , B ,102)

W^d.CHAIN＝E^1.false＝101

(4) C<D is reduced to E^2：

102 (j< , C , D , 0)　　E^2.true＝102

103 (jump,－,－,0)　　E^2.false＝103

(5)if E^2 then is reduced to C，E^2.true is backpached

102 (j< , C , D , 104)

C.CHAIN＝E^2.false＝103

(6)X:＝Y+Z is reduced to S^1，

104 (＋, Y , Z , t_1)

105 (:＝, t_1 ,－, X)

S^1.CHAIN＝0

(7) C S^1 is reduced to S^2，S^2.CHAIN＝merge (103 , 0)＝103

(8) W^d S^2 is reduced to S，S^2.CHAIN is backpached

103 (jump,－,－, 100)

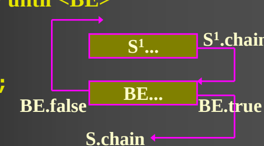
106 (jump,－,－, 100)

S.CHAIN:＝W^d.CHIAN＝101

top-down parsing-directed translation

recursive descent parsing

<S> → i:=<AE> | repeat <s¹> until <BE>




```
'repeat':
    begin
    R.codebegin:=nextstat;
    GETNEXT(TOKEN);
    S¹.CHAIN:=S(TOKEN);
    GETNEXT(TOKEN);
    if TOKEN ≠ 'until'  then  ERROR;
    backpatch(S¹.CHAIN,nextstat);
    GETNEXT(TOKEN);
    (BE.true,BE.false):=BE(TOKEN);//调用BE返回两个出口（真、假出口）
    backpatch(BE.false,R.codebegin);
    S.CHAIN:=BE.true;
    return(S.CHAIN);
    end
end case;
end;
```

LL(1) PARSING

L →D L | ε

D →a | b　　　L代表的由a和b组成的串或空串

要构造的是这样一个语义处理器，它将输入L串并将其中b的个数打印出来。

动作文法：
L → D L
L → {Out}
D → a
D → b {Add}

动作符{Add}和{Out}对应的动作子程序分别如下：
Add : S:=S + 1
Out : Print(S)

动作文法：
L → D L
L → {Out}
D → a
D → b {Add}

动作符{Add}和{Out}对应的动作子程序分别如下：
Add : S:=S + 1
Out : Print(S)

“bab”，LL(1)

步骤	分析栈	剩余输入	产生式	动作
1	#L	bab#	L→DL	
2	#LD	bab#	D→b{A}	
3	#L{A}b	bab#	匹配b	
4	#L{A}	ab#		S:=S+1
5	#L	ab#	L→DL	
6	#LD	ab#	D→a	

动作文法:
L → D L
L → {Out}
D → a
D → b {Add}

动作符{Add}和{Out}对应的动作子程序分别如下:
Add : S:=S + 1
Out : Print(S)

7	#La	ab#	匹配a	
8	#L	b#	L→DL	
9	#LD	b#	D→b{A}	
10	#L{A}b	b#	匹配b	
11	#L{A}	#		S:=S+1
12	#L	#	L→{O}	
13	#{O}	#		Print(S)
14	#	#	接受	

例：while 语句的动作文法和语义子程序

S → while $\{w_1\}$ E $\{w_2\}$ do S^1 $\{w_3\}$

$\{w_1\}$： /* 记住入口位置 */

W.codebegin:=nextstat;

push W.codebegin;

W.codebegin

语义栈

对应while {w1}匹配后

$\{w_2\}$:

/* E匹配后，E.true在栈顶，E.false在次栈顶 */

pop E.true;

backpatch(E.true,nextstat);

E.true
E.false
W.codebegin

语义栈

对应while {w1}E匹配后

S → while $\{w_1\}$ E $\{w_2\}$ do S^1 $\{w_3\}$

$\{w_3\}$： /* S^1匹配后，S^1.CHAIN在栈顶 */

pop S^1.CHAIN;

pop E.false;

pop W.codebegin;

backpatch(S^1.CHAIN,W.codebegin);

emit(jump, -, -, W.codebegin);

S.CHAIN:=E.false;

push S.CHAIN;

S^1.chain
E.false
W.codebegin

语义栈

对应while $\{w_1\}$E$\{w_2\}$do S^1匹配后

S.CHAIN

语义栈

对应while {w1} E {w2} do S^1 {w3}匹配后

8.4 A Survey of Code Optimization Techniques

- Code optimization is to improve code quality (speed and the size of target code) of a program
- A compiler writer must judge which technique are most likely to result in significant code improvement with the smallest increase in the complexity of the compiler

8.4.1 Principal Sources of Code Optimizations

1. Register Allocation

- Good use of registers is the most important feature of efficient code
- Instructors involving register operands are usually shorter and faster than those involving operands in memory
- The number of registers are limited, therefore, efficient utilization of registers is particularly important in generating good code

2. Unnecessary Operations

- Avoid generating code for redundant or unnecessary operation

Example 1:common sub expression elimination

Common sub expression is an expression that appears repeatedly in the code while it's value remains the same. Repeated evaluation can be eliminated by saving the first value for later use

(1) T_1 =4*I

(2) T_2 =addr(A)-4

(3) T_3 =T_2[T_1]

(4)T_4 =4*I → (4) T_4 :=T_1

Example 2:

Avoid storing the value of a variable or temporary that is not subsequently used

（1） I=1

（2） T_1=4

（3） T_3=T_2[T_1]

（4） T_4=T_1

（5） I=I+1

（6） T_1=T_1+4

（7） if $T_1 \le 80$ goto (3)

→

（2） T_1:=4

（3） T_3:=T_2[T_1]

（6） T_1:=T_1+4

（7） if $T_1 \le 80$ goto (3)

3. Costly Operations

- Reduce the cost of operations by implementing in cheaper way

  Example :

  Reduction in strength

  Replace multiplication by 2 with a shift operation

- Constant optimization

  Use information about constants to remove as many operations as possible or to precompute as many operations as possible

- Constant folding

  2+3, can be computed by the compiler and replaced by the constant value 5

- Constant propagation

  Determine if a variable might have a constant value for part or all of a program, and such transformations can then also apply to expressions involving that variable

8.4.2 Classification of Optimizations

- Two useful classifications of optimizations
- The time during the complication process when an optimization can be applied
- The area of the program over which the optimization applies

1. The time of application during compilation

   Optimizations can be performed at practically every stage of compilation. Typically, the majority of optimization are performed

- During intermediate code generation

  Optimizations are made by transformations on the syntax tree itself, where the appropriate subtrees are deleted or replaced by simpler ones

- After intermediate code generation

  For many optimizations, however, the syntax tree is an unsuitable structure for collecting information and performing optimization, these optimization are performed on intermediate code
- During target code generation

  These optimizations depend on the target architecture

2. The area of the program over which the optimization applies

The categories for this classification are:

- Local optimization
- Global optimization
- Interprocedural optimization

- Local Optimization

- Local optimization are applied to straight-line segments of code, that is, code sequences with no jumps into or out of the sequence
- A maximal sequence of straight-line code is called a basic block
- The local optimizations are those that are restricted to basic blocks

- Global Optimizations

  Optimizations that extend beyond basic blocks, but are confined to an individual procedure
- Interprocedural Optimizations

  Optimizations that extend beyond the boundaries of procedure to the entire program

8.4.3 Data Structures and Implementation Techniques for Optimizations

- Two data structures for optimizations

1. Flow graph

   Flow graph is a graphical representation of the intermediate code of a procedure, with which to perform global optimizations
2. DAG(directed acyclic graph)

   DAG is constructed for each basic block, with which to perform local optimizations

1 Flow Graph

- The flow graph is useful for representing global information about the code of each procedure
- The nodes of a flow graph are basic blocks
- The edges are formed from the conditional and unconditional jumps (which must have as their targets the beginnings of other basic blocks)

- The construction of flow graph

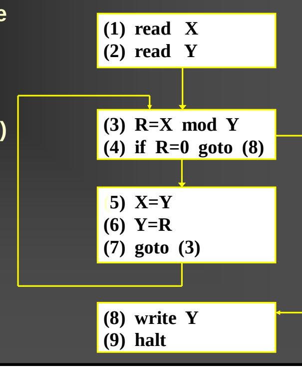
A flow graph, together with each of its basic blocks, can be constructed by a single pass over the intermediate code. Each new basic block is identified as follows:

1. The first instruction begins a new basic block
2. Each label that is the target of a jump begins a new basic block
3. Each instruction that follows a conditional jump begins a new basic block

Flow graph

Three-address code

(1) read X
(2) read Y
(3) R:=X mod Y
(4) if R=0 goto (8)
(5) X:=Y
(6) Y:=R
(7) goto (3)
(8) write Y
(9) halt



Explanation:

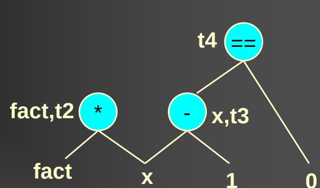
- The flow graph is the major data structure needed by data flow analysis, which uses it to accumulate information to be used for optimizations
- Different kinds of information may require different kinds of processing of the flow graph, and the information gathered can be quite varied, depending on the kinds of optimizations desired

2 DAG(directed acyclic graph)

- DAG of a Basic Block
  A DAG traces the computation and reassignment of values and variables in a basic block
- Leaf nodes are values that are used in the block that come form elsewhere
- Interior nodes are operations on values
- Assignment of a new value is represented by attaching the name of the target variable or temporary to the node representing the value assigned

Example　A basic block　DAG for this basic block

label L2
t2=fact*x
fact=t2
t3=x-1
x=t3
t4=x==0
if_false t4 goto L2



- Labels at the beginning of basic blocks and jumps at the end are usually not included in the DAG
- Copy operation do not create new nodes, but simply add new labels to the nodes representing the value copied
- Repeated use of the same value is also represented in DAG

- DAG Construction
- Process each statement of the block in turn
- For statement of the form x=y+z
  - Look for the nodes that represent the “current” values of y and z.
  - If y and z are constant, then create a node representing the result value of y+z, and label this node x, delete the node of y and z if they are created in this statement. otherwise create a node labeled + and give it two children y and z. Then we label this node x.
  - However, if there is already a node denoting the same value as y+z, we do not add the new node to the DAG, but rather give the existing node the additional label x

- Three details should be mentioned
  - For x=y+z, if y and z are constant, the interior node for + does not need to create, however execute y+z (constant folding)
  - For x=y+z, if there is already a node denoting the same value as y+z, we do not create new node, but rather give the existing node the additional label x. (local common subexpression elimination)
  - if x had previously labeled some other node, we remove the label, since the "current" value of x is the node just created(eliminate unnecessary assignment)

Example DAG of this block

(1) T_0:=3.14

(2) T_1:=2*T_0

(3) T_2:=R+r

(4) A:=T_1*T_2

(5) B:=A

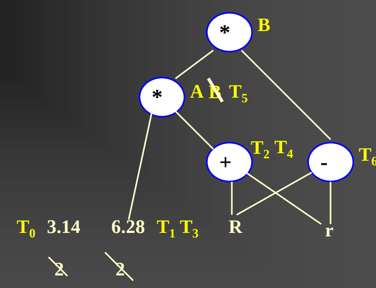
(6) T_3:=2*T_0

(7) T_4:=R+r

(8) T_5:=T_3*T_4

(9) T_6:=R−r

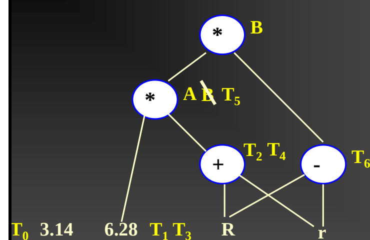
(10) B:=T_5*T_6



- Target code, or a revised version of intermediate code, can be generated from a DAG by a traversal according to any of the possible topological sorts of the nonleaf nodes

  A topological sort of a DAG is a traversal such that the children of nodes are visited before all of their parents

New sequence of three-address code



T_2 :=R+r

A :=6.28*T_2

T_6 :=R−r

B :=A*T_6

We have avoided the unnecessary use of temporaries