Intermediate Code Generation

Dynamic storage allocation

1. Explicit allocation of fixed sized blocks
2. Explicit allocation of variable size blocks (First-fit method)

Implicit de allocation
Require cooperation between user program and run time package : to know when a block is no longer in use

Problems
Recognize block boundaries
If the block is in use
1. Reference count
2. Marking technique (Frozen pointer technique)

Intermediate Code Generation

- Translating source program into an “intermediate language.”
  - Simple
  - CPU Independent,
  - …yet, close in spirit to machine language.

- Or, depending on the application other intermediate languages may be used, but in general, we opt for simple, well structured intermediate forms.

- (and this completes the “Front-End” of Compilation).

Benefits

1. **Retargeting is facilitated**
2. **Machine independent Code Optimization can be applied.**

Intermediate Code Generation (II)

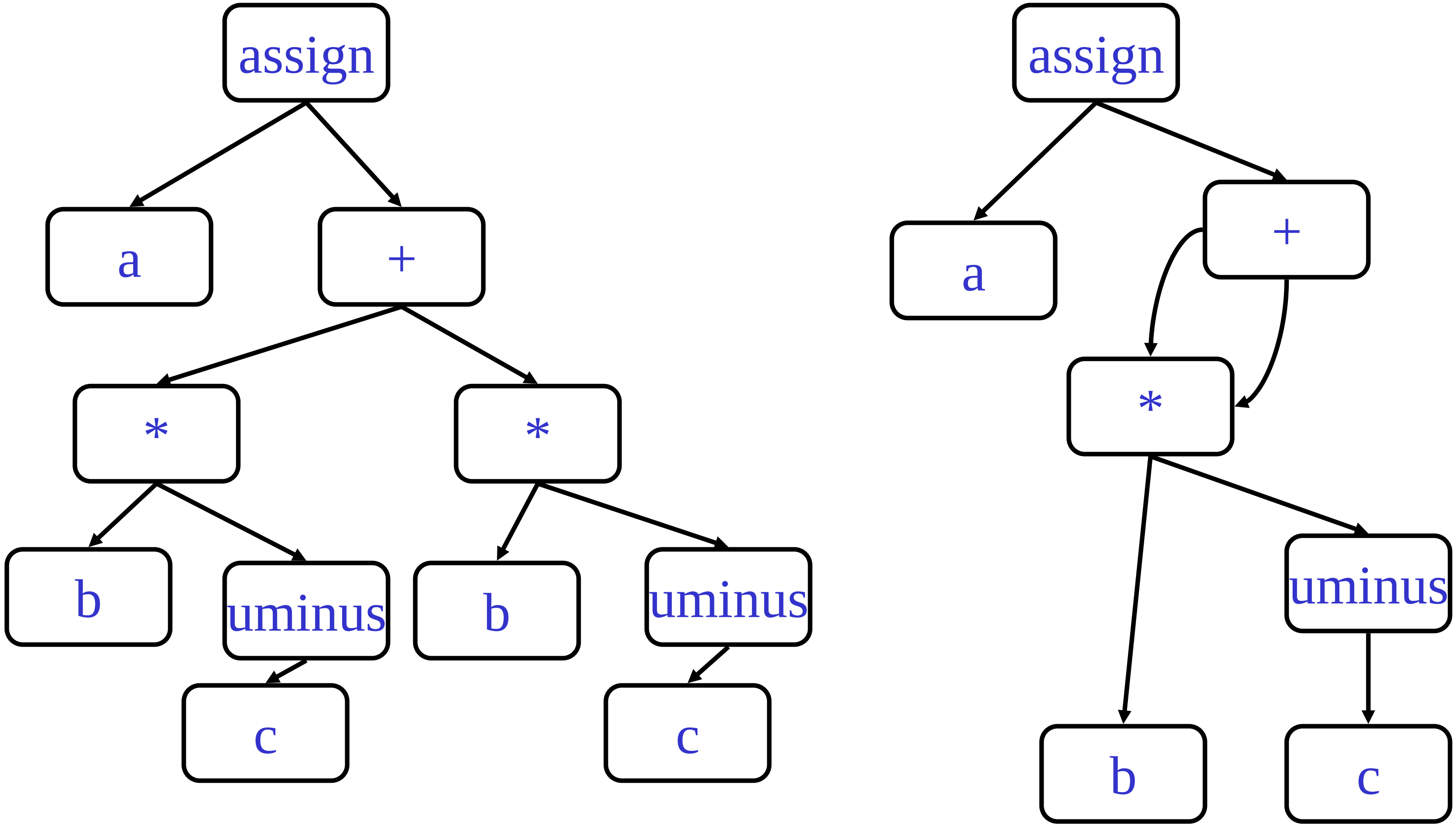
- *Intermediate codes* are machine independent codes, but they are close to machine instructions.

- The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator.

- Intermediate language can be many different languages, and the designer of the compiler decides this intermediate language.
  - syntax trees can be used as an intermediate language.
  - postfix notation can be used as an intermediate language.
  - three-address code (Quadraples) can be used as an intermediate language
    - we will use quadraples to discuss intermediate code generation
    - quadraples are close to machine instructions, but they are not actual machine instructions.
  - some programming languages have well defined intermediate languages.
    - java – java virtual machine
    - prolog – warren abstract machine
    - In fact, there are byte-code emulators to execute instructions in these intermediate languages.

Types of Intermediate Languages

- Graphical Representations.
  - Consider the assignment a:=b*-c+b*-c:



Syntax Dir. Definition for Assignment Statements

PRODUCTION	Semantic Rule
$S \rightarrow$ id := E	{ $S.nptr$ = *mknode* ('assign', *mkleaf*(id, id.*entry*), E.*nptr*) }
$E \rightarrow E_1 + E_2$	{E.*nptr* = *mknode*('+', E_1.*nptr*,E_2.*nptr*) }
$E \rightarrow E_1 * E_2$	{E.*nptr* = *mknode*('*', E_1.*nptr*,E_2.*nptr*) }
$E \rightarrow - E_1$	{E.*nptr* = *mknode*('uminus',E_1.*nptr*) }
$E \rightarrow (E_1)$	{E.*nptr* = E_1.*nptr* }
$E \rightarrow$ id	{E.*nptr* = *mkleaf*(id, id.*entry*) }

Three Address Code

- Statements of general form `x:=y op z`
- No built-up arithmetic expressions are allowed.
- As a result, `x:=y + z * w`
  should be represented as
  t_1:=z * w
  t_2:=y + t_1
  x:=t_2
- Observe that given the syntax-tree or the dag of the graphical representation we can easily derive a three address code for assignments as above.
- In fact three-address code is a linearization of the tree.
- Three-address code is useful: related to machine-language/ simple/ optimizable.

Example of 3-address code

t_1:=- c
t_2:=b * t_1
t_3:=- c
t_4:=b * t_3
t_5:=t_2 + t_4
a:=t_5

t_1:=- c
t_2:=b * t_1
t_5:=t_2 + t_2
a:=t_5

Types of Three-Address Statements.

Assignment Statement:	x:=y op z
Assignment Statement:	x:=op z
Copy Statement:	x:=z
Unconditional Jump:	goto L
Conditional Jump:	if x relop y goto L
Stack Operations:	Push/pop

More Advanced:

Procedure:

param x_1
param x_2
…
param x_n
call p,n

Index Assignments:

x:=y[i]
x[i]:=y

Address and Pointer Assignments:

x:=&y
x:=*y
*x:=y

Syntax-Directed Translation into 3-address code.

- First deal with assignments.
- Use attributes
  - E.*place:* the name that will hold the value of E
    - Identifier will be assumed to already have the place attribute defined.
  - E.*code:*hold the three address code statements that evaluate E (this is the `translation' attribute).
- Use function ***newtemp*** that returns a new temporary variable that we can use.
- Use function ***gen*** to generate a single three address statement given the necessary information (variable names and operations).

Syntax-Dir. Definition for 3-address code

PRODUCTION	Semantic Rule
$S \rightarrow$ **id := E**	{ S.*code* = E.*code*||*gen*(id.*place* '=' E.*place* ';') }
$E \rightarrow E_1 + E_2$	{E.*place*= *newtemp* ; E.*code* = E_1.*code* || E_2.*code* || || *gen*(E.*place*':='E_1.*place*'+'E_2.*place*) }
$E \rightarrow E_1 * E_2$	{E.*place*= *newtemp* ; E.*code* = E_1.*code* || E_2.*code* || || *gen*(E.*place*'='E_1.*place*'*'E_2.*place*) }
$E \rightarrow - E_1$	{E.*place*= *newtemp* ; E.*code* = E_1.*code* || || *gen*(E.*place* '=' 'uminus' E_1.*place*) }
$E \rightarrow (E_1)$	{E.*place*= E_1.*place* ; E.*code* = E_1.*code*}
$E \rightarrow$ **id**	{E.*place* = id.*entry* ; E.*code* = '' }

e.g. **a := b * - (c+d)**

What about things that are not assignments?

- E.g. while statements of the form "while E do S" (intepreted as while the value of E is not 0 do S)

Extension to the previous syntax-dir. Def.

<u>PRODUCTION</u>

S → while E do S_1

<u>Semantic Rule</u>

***S.begin* = *newlabel*;**
***S.after* = *newlabel* ;**
S.*code* = *gen*(S.*begin* ':')
|| E.*code*
|| *gen*('if' *E*.place '=' '0' 'goto' S.*after*)
|| S_1.*code*
|| gen('goto' S.*begin*)
|| gen(S.*after* ':')

Implementations of 3-address statements

- Quadruples

t_1:=- c
t_2:=b * t_1
t_3:=- c
t_4:=b * t_3
t_5:=t_2 + t_4
a:=t_5

Three address statement : abstract form of IC
3- such representations are

	op	*arg1*	*arg2*	*result*
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

Quadruples

Temporary names must be entered into the symbol table as they are created.

Implementations of 3-address statements, II

- Triples

t_1:=- c
t_2:=b * t_1
t_3:=- c
t_4:=b * t_3
t_5:=t_2 + t_4
a:=t_5

	op	*arg1*	*arg2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

Triples

Temporary names are not entered into the symbol table.

Other types of 3-address statements

- e.g. ternary operations like
  `x[i]:=y` `x:=y[i]`
- require two or more entries. e.g.

	op	*arg1*	*arg2*
(0)	[ ] =	x	i
(1)	assign	(0)	y

	op	*arg1*	*arg2*
(0)	[ ] =	y	i
(1)	assign	x	(0)

Implementations of 3-address statements, III

- Indirect Triples : Listing pointers to triples

	op		*op*	*arg1*	*arg2*
(0)	(14)	(14)	uminus	c	
(1)	(15)	(15)	*	b	(14)
(2)	(16)	(16)	uminus	c	
(3)	(17)	(17)	*	b	(16)
(4)	(18)	(18)	+	(15)	(17)
(5)	(19)	(19)	assign	a	(18)

Dealing with Procedures

P → procedure id ‘;’ block ‘;’

<u>Semantic Rule</u>

***begin* = *newlabel*;**

Enter into symbol-table in the entry of the procedure name the begin label.

P.*code* = *gen*(*begin* ‘:’) || block.*code* ||
gen(‘pop’ return_address) || gen(“goto return_address”)

S → call id

<u>Semantic Rule</u>

Look up symbol table to find procedure name. Find its begin label called proc_begin

return* = *newlabel;

S.*code* = *gen*(‘push’return)*;* *gen*(goto proc_begin) || *gen*(return “:”)

Declarations

Using a global variable *offset*

PRODUCTION	**Semantic Rule**
P $\rightarrow$ M D	**{ }**
M $\rightarrow$ ε	**{*offset*:=0 }**
D $\rightarrow$ id : T	**{ *addtype*(id.*entry*, T.*type*, *offset*) *offset*:=*offset* + T.*width* }**
T $\rightarrow$ char	**{T.*type* = *char*; T.*width* = 4; }**
T $\rightarrow$ integer	**{T.*type* = *integer* ; T.*width* = 4; }**
T $\rightarrow$ array [num] of T_1	**{T.*type*=*array*(1..num.*val*,T_1.*type*) T.*width* = num.val * T_1.*width*}**
T $\rightarrow$ ^T_1	**{T.*type* = *pointer*(T_1.*type*); T_1.*width* = 4}**

Nested Procedure Declarations

- For each procedure we should create a symbol table.

mktable(previous) – create a new symbol table where previous is the parent symbol table of this new symbol table

enter(symtable,name,type,offset) – create a new entry for a variable in the given symbol table.

enterproc(symtable,name,newsymbtable) – create a new entry for the procedure in the symbol table of its parent.

addwidth(symtable,width) – puts the total width of all entries in the symbol table into the header of that table.

- We will have two stacks:
  - **tblptr** – to hold the pointers to the symbol tables
  - **offset** – to hold the current offsets in the symbol tables in **tblptr** stack.

Keeping Track of Scope Information

Consider the grammar fraction:

P → D
D → D ; D | id : T | proc id ; D ; S

Each procedure should be allowed to use independent names.
Nested procedures are allowed.

Keeping Track of Scope Information

(a translation scheme)

P → M D { ***addwidth***(***top***(tblptr), ***top***(offset)); ***pop***(tblptr); ***pop***(offset) }

M → ε { t:=***mktable***(null); ***push***(t, tblptr); ***push***(0, offset)}

D → D_1 ; D_2 ...

D → proc id ; N D ; S { t:=***top***(tblpr); ***addwidth***(t,***top***(offset));
pop(tblptr); ***pop***(offset);
enterproc(***top***(tblptr), id.name, t)}

N → ε {t:=***mktable***(***top***(tblptr)); ***push***(t,tblptr); ***push***(0,offset);}

D → id : T {***enter***(***top***(tblptr), id.name, T.***type***, ***top***(offset);
top(offset):=***top***(offset) + T.***width***

Example: proc func1; D; proc func2 D; S; S