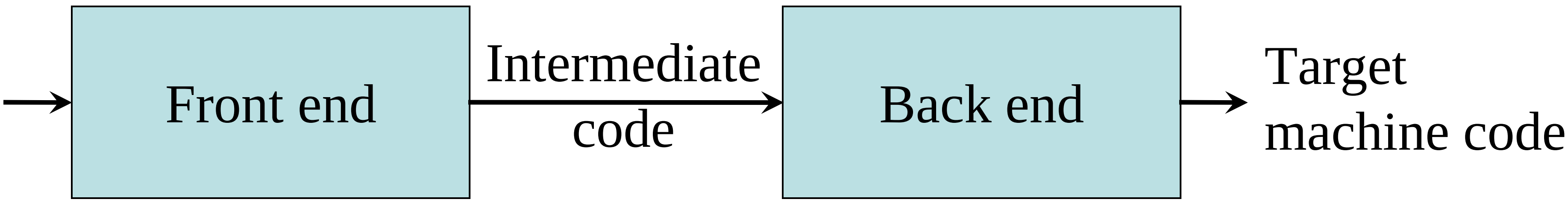
Intermediate Code Generation

Reading: chapter 8

How the syntax-directed methods can be used to translate into an intermediate form of programming language.

Intermediate Code Generation



The front end translate the source program into an intermediate representation from which the backend generates target code

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

Intermediate Representations

There are three kinds of intermediate representations:

1. *Graphical representations* (e.g. Syntax tree or Dag)

2. *Postfix notation*: operations on values stored on operand stack (similar to JVM bytecode)

3. *Three-address code*: (e.g. triples and quads)Sequence of statement of the form

$x = y \text{ op } z$

Note: we discuss only three-address code representation

Three-Address Code (Quadruples)

A quadruple is: $x = y$ ***op*** z

where x, y and z are names, constants or compiler-generated temporaries and ***op*** is any operator. (only one operator on the right side of the statement)

Postfix notation (much better notation because it looks like a machine code instruction)

op y,z,x apply operator *op* to y and z, and store the result in x.

We use the term "three-address code" because each statement usually contains three addresses (two for operands, one for the result).

Thus the source language like $x + y * z$ might be translated into a sequence

$t1 = y * z$

$t2 = x + t1$

where $t1$ and $t2$ are the compiler generated temporary name.

Three-Address Statements

- Assignment statements: *x* = *y op z*, *op is binary*
- Assignment statements: *x* = *op y*, *op is unary*
- Indexed assignments: *x* = *y*[*i*], *x*[*i*] = *y*
- Pointer assignments: *x* = **&***y*, *x* = *****y*, *****x* = *y*
- Copy statements: *x* = *y*
- Unconditional jumps: **goto** *label*
- Conditional jumps: **if** *x relop y* **goto** *label*
- Function calls: **param** *x*… **call** *p*, *n* **return** *y*

Syntax-Directed Translation into Three-Address Code

Productions	Synthesized attributes:	
$S \rightarrow$ **id** **=** E	S.code	three-address code for evaluating S
| **while** E **do** S	S.begin	label to start of S or nil
$E \rightarrow E$ **+** E	S.after	label to end of S or nil
| E ***** E	E.code	three-address code for evaluating E
| **-** E	E.place	a name that holds the value of E
| **(** E **)**		
| **id**		
| **num**		

To represent three address statements

gen(E.place '=' E_1.place '+' E_2.place)

Code generation

```
t3 = t1 + t2
```

Syntax-Directed Translation into Three-Address Code

Productions	Semantic rules
$S \rightarrow$ **id** = E	S.code = E.code $\Vert$ *gen*(**id**.place '=' E.place); S.begin = S.after = nil
$S \rightarrow$ **while** E **do** S_1	(*see next slide*) *Returns a new temporary name*
$E \rightarrow E_1$ + E_2	E.place = *newtemp*(); E.code = E_1.code $\Vert$ E_2.code $\Vert$ *gen*(E.place '=' E_1.place '+' E_2.place)
$E \rightarrow E_1$ * E_2	E.place = *newtemp*(); E.code = E_1.code $\Vert$ E_2.code $\Vert$ *gen*(E.place '=' E_1.place '*' E_2.place)
$E \rightarrow$ - E_1	E.place = *newtemp*(); E.code = E_1.code $\Vert$ *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**.name E.code = ''
$E \rightarrow$ **num**	E.place = *newtemp*(); E.code = *gen*(E.place '=' **num**.value)

Syntax-Directed Translation into Three-Address Code

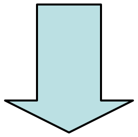
Returns a new label

Production	Semantic rule
$S \rightarrow$ **while** E **do** S_1	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 ':')

S.begin:	E.code
	if E.place **= 0 goto** S.after
	S.code
	goto S.begin
S.after:	…

Syntax-Directed Translation into Three-Address Code : Example

i = 2 * n + k
while i do
i = i - k



```
     t1 = 2
     t2 = t1 * n
     t3 = t2 + k
     i  = t3
L1:  if i = 0 goto L2
     t4 = i - k
     i  = t4
     goto L1
L2:
```

Declarations: Names and Scopes

Declarations list size units as bytes, in a uniform-size environment offsets and counts could be given in units of slots, where a slot (4 bytes on 32-bit machines) holds anything.

We need local symbol tables to record global declarations as well as local declarations in procedures, blocks, and structs to resolve names.

The address consist of an offset from the base of static data area or the field for local data in an activation record.

Before the first declaration the offset is set to 0. when new name is created, that name entered in the symbol table with offset equal to the current value of offset, and offset is incremented by the width of data object denoted by that name.

Declarations: Symbol Table Operations

- *mktable*(*previous*) returns a pointer to a new table that is linked to a previous table in the outer scope
- *enter*(*table*, *name*, *type*, *offset*) creates a new entry in *table*
- *addwidth*(*table*, *width*) accumulates the total width of all entries in *table*
- *enterproc*(*table*, *name*, *newtable*) creates a new entry in *table* for procedure with local scope *newtable*
- *lookup*(*table*, *name*) returns a pointer to the entry in the table for *name* by following linked tables

Grammar for Declarations & Statements

Productions

$P \rightarrow D$ **;** S

$D \rightarrow D$ **;** D

  | **id :** T

  | **proc id ;** D **;** S

$T \rightarrow$ **integer**

  | **real**

  | **array [num] of** T

  | ^ T

  | **record** D **end**

$S \rightarrow S$ **;** S

  | **id :=** E

  | **call id (** A **)**

Productions *(cont'd)*

$E \rightarrow E$ **+** E

  | E ***** E

  | **-** E

  | **(** E **)**

  | **id**

  | E ^

  | **&** E

  | E **. id**

$A \rightarrow A$ **,** E

  | E

Synthesized attributes:

T.type	pointer to type
T.width	storage width of type (bytes)
E.place	name of temp holding value of *E*

Global data to implement scoping:

tblptr	stack of pointers to tables
offset	stack of offset values

Grammar that generate Pascal like declarations and statements

Translation Schemes for Declarations

$P \rightarrow$ { t := $mktable$(nil); $push$(t, $tblptr$); $push$(0, $offset$) }
D **;** S

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

$D \rightarrow$ **proc id ;**
{ t := $mktable$(top($tblptr$)); $push$(t, $tblptr$); $push$(0, $offset$) }
D_1 **;** S
{ t := top($tblptr$); $addwidth$(t, top($offset$));
pop($tblptr$); pop($offset$);
$enterproc$(top($tblptr$), **id**.name, t) }

$D \rightarrow D_1$ **;** D_2

… …->

Translation Schemes for Declarations

… …

$T \rightarrow$ **integer** { T.type := '*integer*'; T.width := 4 }

$T \rightarrow$ **real** { T.type := '*real*'; T.width := 8 }

$T \rightarrow$ **array [num] of** T_1

{ T.type := *array*(**num**.val, T_1.type);

T.width := **num**.val * T_1.width }

$T \rightarrow$ ^ T_1

{ T.type := *pointer*(T_1.type); T.width := 4 }

$T \rightarrow$ **record**

{ *t* := *mktable*(nil); *push*(*t*, *tblptr*); *push*(0, *offset*) }

D **end**

{ T.type := *record*(*top*(*tblptr*)); T.width := *top*(*offset*);

addwidth(*top*(*tblptr*), *top*(*offset*)); *pop*(*tblptr*); *pop*(*offset*) }

Translation Schemes for Assignments Statements

$S \rightarrow S$ **;** S

$S \rightarrow$ **id :=** E

{ *p* := *lookup*(*top*(*tblptr*), **id**.name);
  if *p* = nil **then**
    error()
  else if *p*.level = 0 **then** *// global variable*
    emit(**id**.place ':=' *E*.place)
  else *// local variable in subroutine frame*
    emit(fp[*p*.offset] ':=' *E*.place) }

Translation Schemes for Expressions

$E \rightarrow E_1 + E_2$	{ E.place := *newtemp*(); *emit*(E.place ':=' E_1.place '+' E_2.place) }
$E \rightarrow E_1 * E_2$	{ E.place := *newtemp*(); *emit*(E.place ':=' E_1.place '*' E_2.place) }
$E \rightarrow - E_1$	{ E.place := *newtemp*(); *emit*(E.place ':=' 'uminus' E_1.place) }
$E \rightarrow (E_1)$	{ E.place := E_1.place }
$E \rightarrow$ **id**	{ p := *lookup*(*top*(*tblptr*), **id**.name); **if** p = nil **then** *error*() **else if** p.level = 0 **then** // *global variable* E.place := **id**.place **else** // *local variable in frame* E.place := fp[p.offset] }

……->

Translation Schemes for Expressions

… …

$E \rightarrow E_1$ ^ { E.place := *newtemp*();
emit(E.place ':=' '*' E_1.place) }

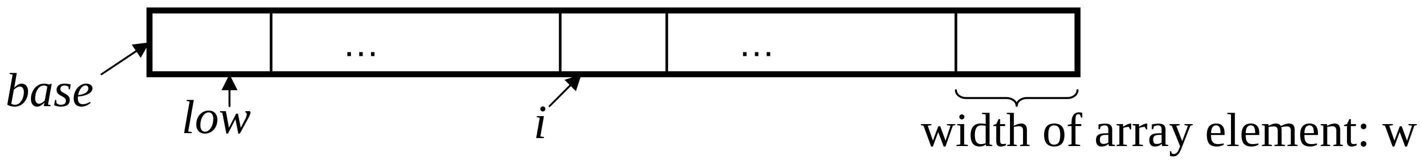
$E \rightarrow$ **&** E_1 { E.place := *newtemp*();
emit(E.place ':=' '&' E_1.place) }

$E \rightarrow$ $\mathbf{id}_1$ **.** $\mathbf{id}_2$ { p := *lookup*(*top*(*tblptr*), $\mathbf{id}_1$.name);
if p = nil **or** p.type != Trec **then** *error*()
else
q := *lookup*(p.type.table, $\mathbf{id}_2$.name);
if q = nil **then** error()
else if p.level = 0 **then** // *global variable*
E.place := $\mathbf{id}_1$.place[q.offset]
else // *local variable in frame*
E.place := fp[p.offset+q.offset] }

Addressing Array Elements

One-Dimensional Arrays

```
A : array [10..20] of integer;
```



`… := A[i]` $= base_{\mathbf{A}} + (i - low) * w$

$= i * w + c$

where $c = base_{\mathbf{A}} - low * w$

with $low = 10$; $w = 4$

```
t1 := c    // c = baseA - 10 * 4
t2 := i * 4
t3 := t1[t2]
…  := t3
```

Addressing Array Elements

Multi-Dimensional Arrays

`A : array [1..2,1..3] of integer;`

`… := A[i,j]` $= base_{A} + ((i_1 - low_1) * n_2 + i_2 - low_2) * w$

$= ((i_1 * n_2) + i_2) * w + c$

where $c = base_{A} - ((low_1 * n_2) + low_2) * w$

with $low_1 = 1;\ low_2 = 1;\ n_2 = 3;\ w = 4$

`t1 := i * 3`
`t1 := t1 + j`
`t2 := c`   **`// c =`** $base_{A} - (1 * 3 + 1) * 4$
`t3 := t1 * 4`
`t4 := t2[t3]`
`… := t4`

See on the book for translation scheme for addressing array elements – page 484 Ist ed

Translating Logical and Relational Expressions

a or b and not c

```
t1 := not c
t2 := b and t1
t3 := a or t2
```

a < b

```
    if a < b goto L1
    t1 := 0
    goto L2
L1: t1 := 1
L2:
```

See translation scheme on the book page 490, Fig 8.20

Translating Procedure Calls

$S \rightarrow$ **call id (** *Elist* **)**
Elist $\rightarrow$ *Elist* **,** *E*
| *E*

$S \rightarrow$ **call id (** *Elist* **)**	{ **for** each item *p* on *queue* **do** *emit*(‘**`param`**’ *p*); *emit*(‘**`call`**’ **id**.place |*queue*|) }
Elist $\rightarrow$ *Elist* **,** *E*	{ append *E*.place to the end of *queue* }
Elist $\rightarrow$ *E*	{ initialize *queue* to contain only *E*.place }

call foo(a+1, b, 7) ⇒

```
t1 := a + 1
t2 := 7
param t1
param b
param t2
call foo 3
```

Exercise

8.1(c), 8.2 (a), 8.3(c), 8.7, 8.12 & 8.14 from book