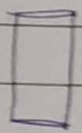
Camlin Page
Date 9 / 4 / 19

SDT for C-like Language (CFG)
(using CLR(1))

Area of cylinder



$2\pi r h + 2\pi r^2$

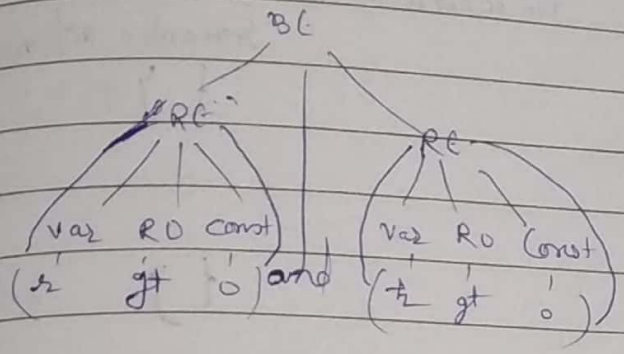
$= 2\pi r (h + r)$

C-like program	Intermediate Code
main {	
float r;	FLOAT r
float h;	FLOAT h
float v;	FLOAT v
float a;	FLOAT a
read(r);	a = READ() → macro
read(h);	h = READ()
~~read(v);~~	~~v = READ()~~
~~read(a);~~	~~a = READ()~~
if (r gt 0) and (h gt 0)	t_1 = r gt 0
then goto 10;	t_2 = h gt 0
goto 20;	t_3 = t_1 * t_2
	IF t_3 THEN GOTO 10 ~~IF t_3 THEN GOTO~~ 10
10: v = 22/7 * r * r * h;	10: ~~...~~ c_1
a = 2 * 22/7 * r * (r+h);	c_2
write(v);	WRITE v
write(a);	WRITE a
stop;	STOP
20: stop;	20: STOP
}	

left associative (note on the read statements)

here, if -ve we should give error message

22/7 can be taken as a symbol and computed only once.



C1

$t_4 = r$ inttofl 2
$t_5 = 22/7$
$t_6 = t_5 * t_4$ → Semantic error as int and float can't be multiplied. This error is given.
$t_7 = t_6 * r$
$a = t_7$

C2

$t_8 = r + r$
$t_9 = 2 * 22$
$t_{10} = t_9 / 7$
$t_{11} = t_{10} * r$
$t_{12} = t_1 * t_8$
$a = t_{12}$

At the end of the statement, reduction happens

Semantically incorrect ① multiplication float & int
② infinite loop
③ dividing by 0.

Camlin | Page
Date / /

• JC is loaded and
not just shown on the screen

CFG:

• It's not algorithmic
• When reduce happens, the actions taken

	Product	Semantic actions
P_0	$P' \rightarrow P$	{ }
	Augmented Product	
	$P \rightarrow main\ \{ STMTS \}$	{ }
	$STMTS \rightarrow STMTS\ S$	{ }
	$STMTS \rightarrow S$	{ }
	$S \rightarrow const : S$ |	{ }
	$S \rightarrow S$ |	{ }
	$S1 \rightarrow DS \mid RS \mid WS \mid AS \mid IETS \mid GS \mid SS$	{ }
	$DS \rightarrow int\ var;$	{ printf("INT", var.name) }
	$DS \rightarrow float\ var;$	{ printf("FLOAT", var.name) }
	$RS \rightarrow read(var);$	{ printf(var.name = "READ()") }
	~~$WS \rightarrow write(var_2);$~~	
	$WS \rightarrow write(var_2);$	{ printf("WRITE" = var.name) }
	$AS \rightarrow var := AE;$	{ printf(var.name = AE.exp) }
	~~$AE \rightarrow AE + T$~~	
	$var \rightarrow a \mid b \mid \ldots \mid z$	{ }
	$Const \rightarrow 0 \mid 1 \mid 2 \ldots \mid 100$	{ }
	$AE \rightarrow AE + T$	{ ~~gen~~ (AE.exp = ~~NewTemp()~~) ~~printf~~ (AE.exp + T.exp) }
	$AE \rightarrow AE - T$	{ gen (AE.exp = NewTemp() printf (AE.exp = T.exp) }
	$AE \rightarrow T$	{ ~~make~~ (AE.exp = T.exp) }

(a new variable) we need a place holder because this T is then reduced to T.

$T \rightarrow T * F$ { gen (T.exp = NewTemp()) }

$T \rightarrow T / F$

$T \rightarrow F$

$F \rightarrow (T)$ F

$F \rightarrow G$

$G \rightarrow var$

$G \rightarrow const$ { make (G.exp = const.

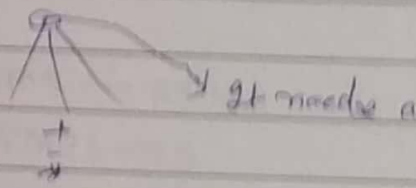
$G \rightarrow (AE)$ { make (G.exp = AE.exp) }

• 9-10 semantic analysis methods were discussed

Page
Date 23/01/19

→ Declarative statements need semantic actions
→ Assignment :
→ Arithmetic :
→

x = y this doesn't need a temporary variable



→ it needs a temporary variable

• Different flavours of languages came from compilation point of view.

→ Typechecking.
→ Default value for a variable

Code optimization

(Intermediate code is optimized) → generated on production

Benefits :

• Reduce the number of variable so as to reduce memory usage.
If swapping is used in memory to utilize it better thrashing will occur.

• The execution time is reduced.
Even 10% or 20% saving is huge because the programs run for several minutes.

• In case of CLR, if we don't get any action shift or reduce, give error message

Code optimization & semantic analysis done in compile time

Camlin | Page
Date / /

Techniques

(1) Compile-time evaluation (Constant Folding)

The program is not run in compile time so there is no evaluation in that time

- The compilers can be wrong sometimes.
  → Semantic actions can have mistakes
- This evaluation is semantic analysis & code opt.

⇒ In cases of calculation for cylinders and circles, we can calculate π once and store it for later usage.

$t_\pi = 22 \div 7$ (it's stored in temporary variable)

→ Getting a variable from variable vs. calculating the value (Trade-offs for storing the value)
(2-3 cycles) (41-5 cycles)

(2) Common sub-expression evaluation & propagation (within a block)

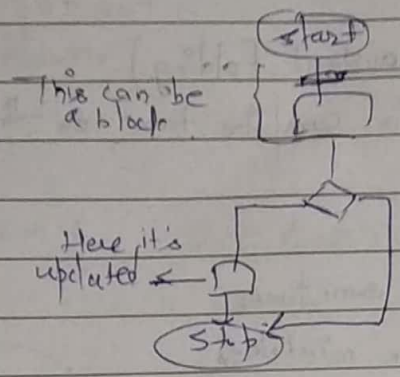
eg $x \uparrow 2$

When the program is examined we get to know that there is some sub-expression used many times
But here, we don't know the value of x.

$t_1 = x$

Note: In the nature of code, we read a value and use it in computations.
We check for a block of code in which the value of x is not changed and use this technique within the block.
⇒ The value of x can change if it comes on LHS of Arithmetic or Assignment statement

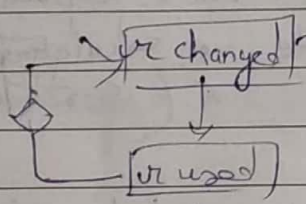
Control Flow Graph can be made by the compiler itself



Data Flow Graph.

As the data flows through the program
↓
variables

The ~~values~~ vars are initiated, assigned, reassigned, used in computation



↳ coz when the variable is declared no value is given to it, it stores garbage in memory.

(3) Replace costlier calculations by less costly calculations.
↳ takes more cycles.

```
r * 2 := r + r
r * 2 := r + r
```

(4) Shifting an "Invariant Code" outside a loop.

```
for (int i=0; i<100; i++)
{
    a := r * 2;        → It's an invariant statement.
    z[i] := i * a;     (An invariant variable is one whose value doesn't depend on loop variable)
}
```

CLR - short CLR of no. of states

Camlin Page

Date / /

(5) Reduce the number of temp variables.

x := r * 22/7 * h → one unit.

y := 22/7 * r * r * 2

→ Use temporary variables for storing values that are reused.

- Fetch insts from PSR
- Decode and get values
- Execute it

(6) We can run 2 blocks of code parallely

But we have to take care that assignment or declaration or arithmetic statements can't be broken down.

```
for ( int i = 0; i < 100; i++ )
{
    u[i] = i + n;
}
```

```
if (i < 100) then
{ . . . . }
```

→ un-executable code (there is no break st. in for-loop)

- Highlight it during semantic analysis.

Scanned by CamScanner