

Code Generation

- Generating Code is essentially the same process as all the semantic analysis phases
 - We can undertake code generation only if we have a *valid* AST
 - Any compilation errors mean we must abort code generation!
- Code generation involves a tree walk of the AST, *after* we have completed address assignment
 - During this process, we may choose to re-arrange some of the AST
 - We must take great care to ensure that the resulting tree is semantically equivalent to the original!
- Code generation begins with the derivation of a series of *code generation templates* - the compiler writer normally needs to figure these out.
 - This requires familiarity with the target machine

Code Generation Templates

- We do this by looking (once again) at the syntax specification
- We go through the grammar, annotating it with the code we need to generate.
- Our target is the RPAL machine, which is stack based
- As a general rule, each of our code generation actions will do something with the top of the stack.

Pascal While loop:

$\langle \text{stmt} \rangle ::= \dots \mid \mathbf{while} \langle \text{expr} \rangle \mathbf{do} \langle \text{stmt} \rangle ;$

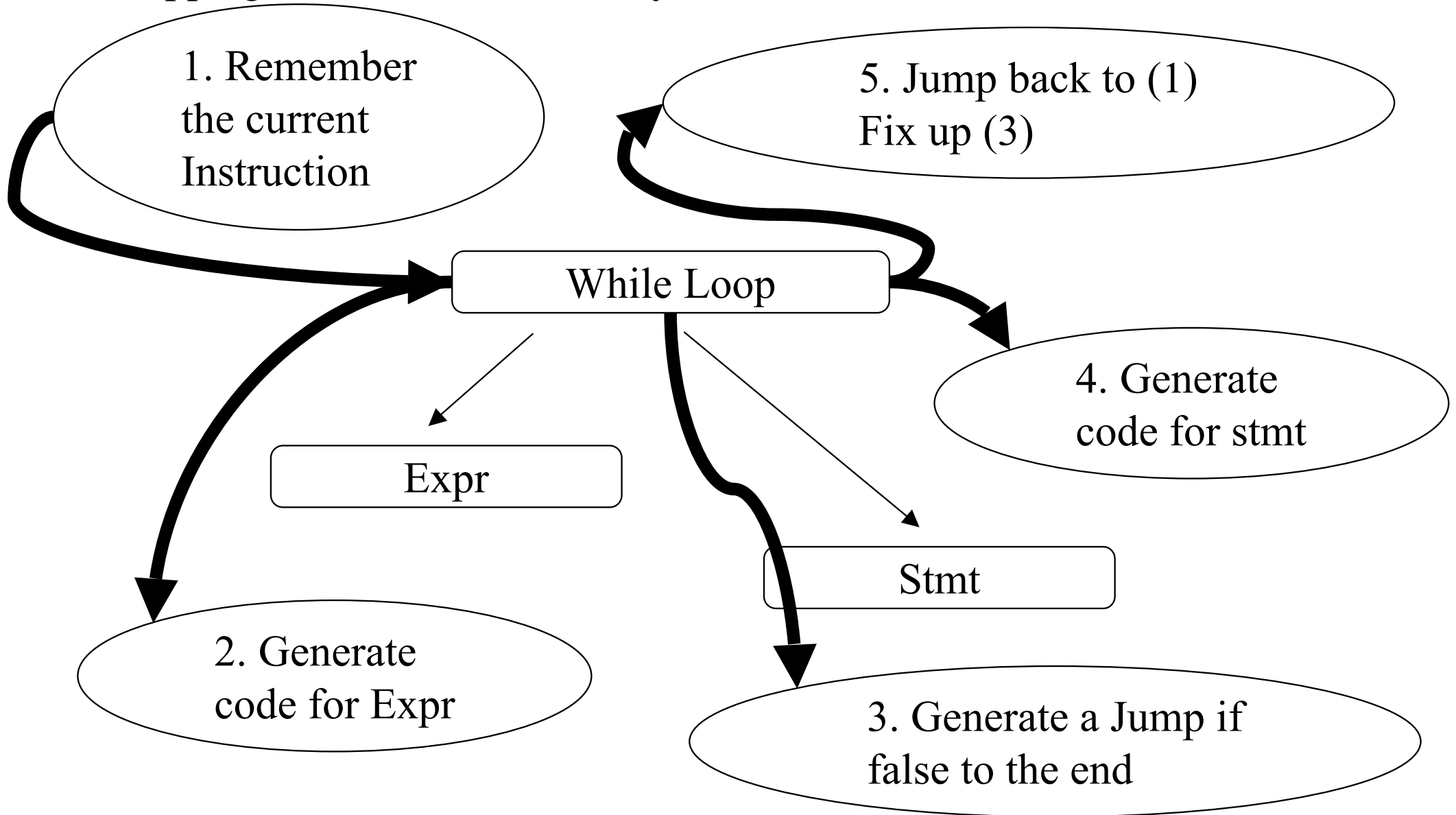
Expr will
leave the
boolean value
on the TOS

Stmt will
generate
code for body

We need to jump back to the start of the
loop and re-evaluate the $\langle \text{expr} \rangle$, if it is
false, we need to terminate the loop

Code Tree View

- Mapping to the code tree is easy:



Tree Walking Code

- The code generation for the while loop is straight forward:

....this code is from the "while" loop AST node class

```
startOfWhileLoop = code.currentLocation;
```

```
this.expression().genCode();
```

```
jumpToEnd = code.currentLocation;
```

```
code.gen(JIF,0,0);
```

```
this.statement().genCode();
```

```
code.gen(JMP, 0, startOfWhileLoop);
```

```
code.fixupJIF(jumpToEnd, code.currentLocation);
```

- Our Code generation class maintains methods that implement the fix-ups (fixupJIF)
- The Code generation class can simply use an array to store the code (this exactly mimics the RPAL machine)
- This fix-up is trivially implemented, but not all are!

Loop Exit statements

- Many programming languages have a loop *exit* statement
- These permit us to terminate a loop prematurely
- Implemented by jumping to the instruction *following* our loop body

while b do

begin

S1;

if b2 then exit;

S2;

if b3 then exit;

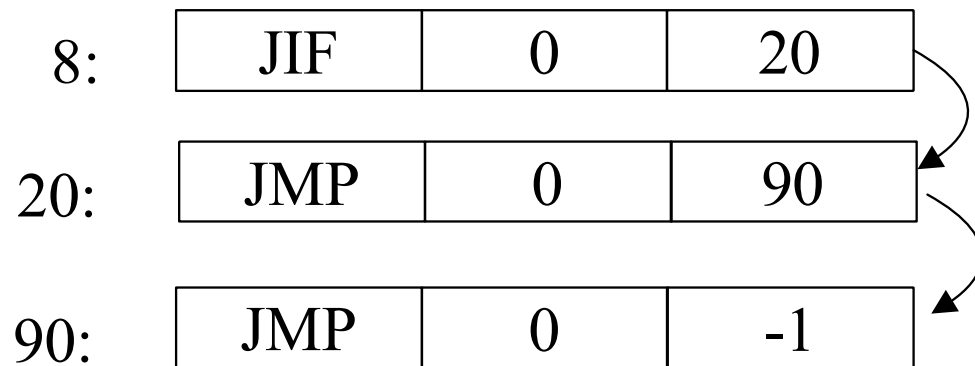
S3;

end;

- Note, of course, that Pascal has no exit mechanism!
- When we determine the point of the loop exit we need to fix up not only the JIF on the loop predicate, but an indeterminate number of exit points as well
 - We will need some kind of list of fix-ups!

Fix-ups

- Code fix ups can be trivially implemented using a list structure in the code *array*.
- The fix-ups required are always on branch (jump) instructions.
- The RPAL instruction format has the second operand as an integer value for jumps (a memory location)
 - Store a sentinel value there (e.g. -1)
 - Chain all the fix-ups by writing the index (code position) of the next fix-up there...



Example?

```
while b do  
begin  
  S1;  
  if b2 then exit;  
  S2;  
  if b3 then exit;  
  S3;  
end;
```

Start:

```
..Evaluate "b"  
JIF 0 $exit  
..Eval S1  
..Eval b2  
JIF 0 $end_1  
JMP 0 $exit
```

end_1:

```
..Eval S2;  
..Eval b3  
JIF 0 $end_2  
JMP 0 $exit
```

end_2:

```
..Eval S3;  
JMP 0 $Start
```

exit:

Misc. Code Gen

- Generating code requires a methodical template driven approach
 - Make *sure* you have code generation templates fully worked through before implementing!
- Implicit type conversions might require you to swap the TOS with TOS-1 to perform the conversion and then swap it back, unless you are careful
- “pointer” or “reference” variables cause problems
 - These contain memory references – think of Java “Object Reference” entities
 - They require *two* loads to get the actual value on the TOS
 - Be wary of these things as parameter transmission mechanisms!
- Implicitly declared loop control variables can cause difficulty too!

Code Generation Templates

- Recall: you need to build templates for each construct in the language
 - The RPAL machine is optimised to make this easy for the general cases

Method:

- For each construct, just write down what needs to be done!
 - Take care with differing *types*
 - Take care with different *kinds* of values (constants, variables, reference params etc)
- It is usual to do this by simply annotating the syntax
 - Note: you should do *all* the syntax – make sure you don't miss anything!
- The rest of this lecture will simply pick some simple and not so simple parts of the language – *it is not complete!*

Assignment

- Assume the variable “Id” is stored at (L,D)
- $\text{Id} := E$
 - Id is a *normal* variable or a *value* parameter:

E

STO L D

- Id is a *reference* parameter:

E

LDV L D

STI 0 0

- $\text{Id} := S$ (S is a *string* type)

LCS 0 S

STO L D

- $\text{Id} := S$ (Id is a *reference* parameter)

LCS 0 S

LDV L D

STI 0 0

- And similarly for constant integers and reals...

If Statements

- Simple!

If C_1 **then** S_1
 elsif C_2 **then** S_2
 ... **elsif** C_{n-1} **then** S_{n-1}
 else S_n

C_1			
JIF	0		L_1
S_1			
JMP	0		L_e
$L_1:C_2$			
JIF	0		L_2
S_2			
JMP	0		L_e
...			
$L_{n-2}:C_{n-1}$			
JIF	0		L_n
S_{n-1}			
JMP	0		L_e
$L_n:S_n$			
$L_e:$			

Relational Operators

- These have the general form: $E_1 \oplus E_2$
- Code generation is straight forward:
 E_1
 E_2
 \oplus
- Where \oplus is the appropriate “OPR” instruction
 - So for “=” we would do OPR 0 10
- We also have monadic operators (like **not**) which have the general form: $\oplus C$
- Code template is:
 C
 \oplus

Ranges

- Expressions like “ E in $lb .. ub$ ” require some care.
 - Convert it to the appropriate tests: $E \geq lb$ and $E \leq ub$.
 - Use the “if” code templates....

	E			
	OPR	0	23	; dup “E”
	lb			
	OPR	0	12	; <
	OPR	0	16	
	JIF	0	L_1	
	ub			
	OPR	0	14	; >
	OPR	0	16	
	JIF	0	L_2	
	OPR	0	17	; push T
	JMP	0	L_3	
L_1 :	OPR	0	24	; drop
L_2 :	OPR	0	18	; push F
L_3 :				

Procedure Calls

- General form: Mark stack top, push parameters and call

- $P(\text{val}_1, \dots, \text{val}_n, \text{ref}_1, \dots, \text{ref}_m)$ becomes:

MST	DL	0
LDV	$L_{\text{val}1}$	$D_{\text{val}1}$
...		
LDV	$L_{\text{val}n}$	$D_{\text{val}n}$
LDA	$L_{\text{ref}1}$	$D_{\text{ref}1}$
...		
LDA	$L_{\text{ref}m}$	$D_{\text{ref}m}$
CAL	$n+m$	L_p

- **Note carefully that this is incomplete!**

- What happens when we pass an (existing) reference parameter as a reference parameter?
- Oops!

Miscellaneous Extras

- We can have templates in our declarative regions too.
 - Declaration of n variables might become “INC 0 n ” for example
- The order in which the AST is traversed is important here
 - The first instruction in the CODE file is the first instruction executed
- Need to emit the code for the main program first, or generate jumps around the code generated by the other procedures
- Functions that don't execute a return and simply “fall through” to the end of the body can be trivially dealt with by planting a SIG instruction at the end of the function...