CSE443
Compilers

Dr. Carl Alphonse
alphonce@buffalo.edu
343 Davis Hall

© 2020 Carl Alphonse - Reproduction of this material is prohibited without the author's consent
Phases of a compiler

Intermediate Representation (IR): specification and generation

Figure 1.6, page 5 of text
backpatching for 
switch
Revisiting and expanding on translation of 'for' statement
Actions in 'for' production

for L_PAREN S1 SEMI_COLON
M1 B SEMI_COLON
M2 S2 R_PAREN
M3 S3
{...}
Actions in 'for' production

numbered from left to right

\texttt{for} \texttt{L\_PAREN} \texttt{S1} \texttt{SEMI\_COLON} \texttt{M1} \texttt{B} \texttt{SEMI\_COLON} \texttt{M2} \texttt{S2} \texttt{R\_PAREN} \texttt{M3} \texttt{S3} \{\ldots\}
Actions in 'for' production

\[
\text{for} \ L\_\text{PAREN} \ S1 \ \text{SEMI\_COLON} \ S3 \ \text{SEMI\_COLON} \ R\_\text{PAREN} \\
\{ \text{M}1.\text{instr} = \text{nextInstr}; \} \ S1 \ \text{SEMI\_COLON} \\
\{ \text{M}2.\text{instr} = \text{nextInstr}; \} \ S2 \ \text{R\_PAREN} \\
\{ \text{gen('goto', M}1.\text{instr}); \text{M}3.\text{instr} = \text{nextInstr}; \} \ S3 \\
\{ \text{t} = \text{mergelist}(\ S1.\text{nextList}, \ S2.\text{nextList}); \text{backpatch(t, M}1.\text{instr}); \text{backpatch(B.trueList, M}3.\text{instr}); \text{backpatch(S3.nextList, M}2.\text{instr}); \text{S.nextList} = \text{B.falseList}; \text{gen('goto', M}2.\text{instr}); \} 
\]
Actions in 'for' production

\{
  M1.instr = nextInstr;
  \}
\{ M2.instr = nextInstr; \}
\{ gen('goto', M1.instr); M3.instr = nextInstr; \}
\{ t = mergelist( S1.nextList, S2.nextList);
  backpatch(t, M1.instr);
  backpatch(B.trueList, M3.instr);
  backpatch(S3.nextList, M2.instr);
  $$\text{.nextList} = \text{B.falseList};$
  gen('goto', M2.instr);
\}
Actions in 'for' production

```
for L_PAREN S1 SEMI_COLON
{
  $<lbl>.instr = nextInstr;
}

B SEMI_COLON

S2

R_PAREN

{ $<lbl>.instr = nextInstr;
  gen('goto', $5.instr);
  $<lbl>.instr = nextInstr;
}

S3

t = mergelist( $3.nextList, $9.nextList);
backpatch(t, $5.instr);
backpatch($6.trueList, $11.instr);
backpatch($12.nextList, $8.instr);
$$nextList = $6.falseList;
gen('goto', $8.instr);
}
```
Exercise from last time
6.7.3 Backpatching Flow-of-Control statements

Exercise: show how to translate a generic switch statement:

```c
switch (E) {
    case C1 : sblock1
    case C2 : sblock2
    ...
    case Cn-1 : sblockn-1
    otherwise : sblockn
}
```
6.7.3 Backpatching Flow-of-Control statements

Exercise: show how to translate a generic switch statement:

```c
switch (E) {
    case C1 : sblock1
    case C2 : sblock2
    ...
    case Cn-1 : sblockn-1
    otherwise : sblockn
}
```

See also the discussion on pages 420 and 421 in the textbook.
Switch (6.8.2)
[read p.420-421]

```java
switch (E) {
    case C₁ : sblock₁
    case C₂ : sblock₂
    ...
    case Cₙ₋₁ : sblockₙ₋₁
    otherwise : sblockₙ
}
```
Switch (6.8.2)
[read p.420-421]

```java
switch (E) {
    case C₁ : sblock₁
    case C₂ : sblock₂
    ...
    case Cₙ₋₁ : sblockₙ₋₁
    otherwise : sblockₙ
}
```
Switch (6.8.2)
[read p.420-421]

```java
switch (E) {
    case C₁ : sblock₁
    case C₂ : sblock₂
    ...
    case Cₙ₋₁ : sblockₙ₋₁
    otherwise : sblockₙ
}
```

<table>
<thead>
<tr>
<th>start</th>
<th>E.code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>goto test</td>
</tr>
<tr>
<td>L₁</td>
<td>sblock₁.code</td>
</tr>
<tr>
<td></td>
<td>goto next</td>
</tr>
</tbody>
</table>

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent
Switch (6.8.2)
[read p.420-421]

switch (E) {
    case C_1 : sblock_1
    case C_2 : sblock_2
    ... 
    case C_{n-1} : sblock_{n-1}
    otherwise : sblock_n
}
Switch (6.8.2)  
[read p.420-421]

switch (E) {
    case C_1 : sblock_1
    case C_2 : sblock_2
    ...
    case C_{n-1} : sblock_{n-1}
    otherwise : sblock_n
}

<table>
<thead>
<tr>
<th>start</th>
<th>E.code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>goto test</td>
</tr>
<tr>
<td>L_1</td>
<td>sblock_1.code</td>
</tr>
<tr>
<td>L_2</td>
<td>sblock_2.code</td>
</tr>
<tr>
<td></td>
<td>goto next</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>L_{n-1}</td>
<td>sblock_{n-1}.code</td>
</tr>
<tr>
<td></td>
<td>goto next</td>
</tr>
<tr>
<td>L_n</td>
<td>sblock_n.code</td>
</tr>
<tr>
<td></td>
<td>goto next</td>
</tr>
</tbody>
</table>
Switch (6.8.2) [read p.420-421]

switch (E) {
    case $C_1$ : sblock$_1$
    case $C_2$ : sblock$_2$
    ...
    case $C_{n-1}$ : sblock$_{n-1}$
    otherwise : sblock$_n$
}
exercise 1

How will you modify your grammar rules to generate intermediate code for switch statements?

Exercise 1: add the necessary actions to the switch productions from your grammar; here’s my rule set for the switch statement - your formulation may be different. Where do the actions go, and what are the actions?

compound_statement :
  SWITCH L_PARENTHESIS expression R_PARENTHESIS case_list OTHERWISE COLON sblock

case_list :
  case_statement | case_statement case_list ;

case_statement :
  CASE constant COLON sblock ;
function calls
Function calls

```c
int foo(int x, int y) {
    int temp;
    temp = 2 * x + 3 * y;
    return temp;
}

int main() {
    ...  
    int a = ...  
    int b = ...  
    int c = foo(a * b, a + b);
    ...  
}
```

What happens during function call?
Function calls

Basic form: id(e₁, e₂, ..., eₖ)
Function calls

- Basic form: \text{id}(e_1,e_2,...,e_k)
- General form: \text{assignable}(e_1,e_2,...,e_k)

- If \( f \) is a function, \( g(4,5) \) yields a function, and \( r.h \) yields a function, then the following are legal:

\[
\begin{align*}
& f(3) & g(4,5)(3) & r.h(3)
\end{align*}
\]
How is function call carried out?

1. evaluate each of the argument expressions
2. mark the resulting values as parameters
3. invoke the function
How is function call carried out?

1. evaluate each of the argument expressions
   use compiler-generated temporaries

2. mark the resulting values as parameters
   use 'param' IR instruction

3. invoke the function
   use 'call(f,n)' IR instruction:
   - f is a function
   - n is arity of function

# of parameters
examples

f(x+1)
examples

Remember that the function call has structure.

\( f(x+1) \)
examples

\[ f(x+1) \]

\[ t1 = x + 1 \]

Generate code for the argument expression
examples

\[ f(x+1) \]

\[ t1 = x + 1 \]

\textit{param} \ t1

Mark the result as a parameter of the function call
examples

f(x+1)

t1 = x + 1
param t1

Call the function. The second argument of the call indicates the arity of the function (i.e. how many parameters it has)
examples

\[ f(x+1) \quad f(x+1,2*\text{y}) \]

\[ t1 = x + 1 \]

\text{param } t1

\text{call}(f,1)
Remember that the function call has structure.
examples

\[ f(x+1) \quad f(x+1,2*y) \]

\[ t1 = x + 1 \quad t1 = x + 1 \]

param t1
call(f,1)

Evaluate the first argument expression.
examples

\[ f(x+1) \quad f(x+1,2y) \]
\[ t1 = x + 1 \quad t1 = x + 1 \]
\[ \text{param } t1 \quad \text{param } t1 \]
\[ \text{call}(f,1) \]

Mark the result as a parameter.
examples

\[ f(x+1) \]
\[ f(x+1,2*y) \]

\[ t1 = x + 1 \]
\[ param t1 \]
\[ call(f,1) \]

\[ t1 = x + 1 \]
\[ param t1 \]
\[ t2 = 2 * y \]

Evaluate the second argument expression.
examples

\[ f(x+1) \quad \quad f(x+1,2\cdot y) \]

\[ t1 = x + 1 \quad \quad t1 = x + 1 \]
param \( t1 \)
param \( t1 \)
call(\( f,1 \))

t2 = 2 \cdot y
param \( t2 \)

Mark the result as a parameter.
Call the function.
An alternative to intermingling the 'param' instructions with the argument evaluation is to gather them in a queue, then place them between the argument evaluations and before the 'call' instruction.
examples

\[ f(x+1) \]
\[ t_1 = x + 1 \]
\[ \text{param } t_1 \]
\[ \text{call}(f,1) \]

\[ f(x+1,2\times y) \]
\[ t_1 = x + 1 \]
\[ t_2 = 2 \times y \]
\[ \text{param } t_1 \]
\[ \text{param } t_2 \]
\[ \text{call}(f,2) \]

\[ f(g(3\times z),h(a+b,a\times b)) \]

A slightly more involved example.
exercise 2

What intermediate code do you come up with for this example?

\( f(g(3z), h(a+b, a*b)) \)
exercise 3

- How will you modify your grammar rules to generate intermediate code for function calls?

- Assume that type checking and argument list length checking has already been accounted for in the semantic actions attached to productions:
  
  - type checking of each argument with corresponding parameter declaration (remembering that there is no coercion allowed in either an explicit or an implicit assignment)
  
  - checking that the number of arguments and the number of parameters is the same