How to Create a Programming Language

L12B: INTERMEDIATE CODE GENERATION II

1

TOPICS

- INTERMEDIATE CODE
  - TRANSLATING EXPRESSIONS
  - TRANSLATING ARRAY ELEMENTS AND ARRAY REFERENCES
  - CONTROL FLOW
  - BOOLEAN EXPRESSIONS AND SHORT CIRCUITING
  - AVOIDING REDUNDANT GOTOS
  - BACKPATCHING

2

TRANSLATING EXPRESSIONS

- EXPRESSION EVALUATION CAN BE CODED BY ADDING 2 ATTRIBUTES TO NON-TERMINALS COMPRISING EXPRESSION:
  - .ADDR – ADDRESS OF RESULT
  - .CODE – CODE TO GENERATE RESULT
- ADD PRIMITIVE OPERATIONS TO HELP CREATE INTERMEDIATE CODE:
  - **GEN()** – THIS GENERATES AN INSTRUCTION (WHICH IS ADDED TO .CODE
  - **NEWTEMP()** – THIS CREATES A NEW TEMPORARY REGISTER FOR RESULTS
  - CONCATENATION OPERATOR - **||** - THIS IS USED TO APPEND .CODE ATTRIBUTES AND OTHER TEXT FOR CODE GENERATION

3

TRANSLATING EXPRESSIONS

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \mid\mid gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

E -> E_1 + E_2

1. Generate a new Temporary
2. Append code for E_1 (generated for some subexpression)
3. Append code for E_2
4. Generate an add instruction
   a. result field is E.addr
   b. operands are E_1.addr (the temp created for E_1) and E_2.addr (temp created for E_2)

4

INCREMENTAL TRANSLATION

- THE .CODE ATTRIBUTES IN THE PREVIOUS TRANSLATION SCHEME CAN GET VERY LONG
  - IT IS POSSIBLE TO GENERATE THE CODE 'ON THE FLY'
  - INSTEAD OF HAVING GEN() WRITE THE INSTRUCTION INTO A CODE ATTRIBUTE, WRITE IT…
    - DIRECTLY TO MEMORY THAT IS HOLDING GENERATED CODE OR…
    - TO A FILE

$$E \rightarrow E_1 + E_2 \quad \{\ E.addr = \textbf{new}\ Temp();\ gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \}$$

No code attributes needed since the instructions to compute the values in E_1.addr and E_2.addr have already been generated.

5

TRANSLATING ARRAY REFERENCES

- ARRAY ELEMENTS ARE STORED CONSECUTIVELY IN
  - ROW-MAJOR ORDER
    - ELEMENTS ACROSS A ROW ARE STORED IN ORDER, THEN THE ORDER MOVES TO THE NEXT ROW
    - RIGHTMOST SUBSCRIPT CHANGES QUICKEST (LIKE A CAR ODOMETER)
  - COLUMN-MAJOR ORDER
    - CONSECUTIVE ELEMENTS MOVE DOWN A COLUMN THEN CONTINUE AT THE TOP OF THE NEXT COLUMN
    - LEFT MOST SUBSCRIPT CHANGES THE QUICKEST
- ELEMENTS IN MOST LANGUAGES ARE NUMBERED 0->N-1 WHERE N IS DIMENSION SIZE

6

TRANSLATING ARRAY REFERENCES

- TO GENERATE THE LOCATION (L-VALUE) OF AN ELEMENT IN A 1 DIMENSIONAL ARRAY:
  - IF *BASE* IS THE LOCATION OF ELEMENT 0
  - *W* IS THE WIDTH OF AN ELEMENT IN BYTES
  - *I* IS THE ELEMENT INDEX
  - *EFF ADDR = BASE + I * W*
- EXAMPLE: A IS AN ARRAY OF INTEGERS (4 BYTES EACH) AND STARTS AT MEMORY LOCATION 0X800000. THE LOCATION OF ELEMENT WITH INDEX 5 IS:
- 0X800000 + 5 * 4 = 0X800014

7

TRANSLATING ARRAY REFERENCES (2 OR MORE DIMENSIONS)

- GIVEN:
  - IF *BASE* IS THE LOCATION OF ELEMENT 0
  - W_R IS THE WIDTH OF A ROW IN BYTES
  - W_E IS THE WIDTH OF AN ELEMENT IN BYTES
  - *I, J ARE THE INDICES FOR ROW/ELEMENT*
  - *EFF ADDR OF A[I][J] = BASE + I * W_R + J * W_E*
- EXAMPLE: A IS A 3X5 ARRAY OF INTEGERS AT MEMORY LOCATION 0X800000:
  - A[1][2] IS AT 0X800000 + 1 * 20 + 2 * 4 = 0X80001C

8

TRANSLATING ARRAY REFERENCES (2 OR MORE DIMENSIONS)

- GENERAL FORMULA FOR K-DIMENSIONAL ARRAY:
  - *EFF ADDR OF $A[I_1][I_2]...[I_K]$ = BASE + I_L X W_L + I_2 * W_2 + . . . + I_K * W_K*
- ALSO, FOR K DIMENSIONS, CAN USE ELEMENT COUNTS PER ROW/COLUMN RATHER THAN WIDTH.
  - N_I IS THE NUMBER OF ELEMENTS IN THE ROW OR PLANE
  - W IS THE WIDTH OF THE BASE ELEMENT
  - BASE IS THE BASE ADDRESS OF THE ARRAY
- *EFF ADDR OF $A[I_1][I_2]...[I_K]$ = BASE + ((...(I_1 * N_2 + I_2) * N_3 + I_3)...)*N_K + I_K) * W*

9

SEMANTIC ACTIONS FOR ARRAY REFERENCES

$S \rightarrow$ **id** = E ;	{ *gen(top.get(**id**.lexeme) '=' E.addr);* }
| L = E ;	{ *gen(L.addr.base '[' L.addr ']' '=' E.addr);* }
$E \rightarrow E_1$ + E_2	{ *E.addr = **new** Temp(); gen(E.addr '=' E_1.addr '+' E_2.addr);* }
| **id**	{ *E.addr = top.get(**id**.lexeme);* }
| L	{ *E.addr = **new** Temp(); gen(E.addr '=' L.array.base '[' L.addr ']');* }
$L \rightarrow$ **id** [E]	{ *L.array = top.get(**id**.lexeme); L.type = L.array.type.elem; L.addr = **new** Temp(); gen(L.addr '=' E.addr '*' L.type.width);* }
| L_1 [E]	{ *L.array = L_1.array; L.type = L_1.type.elem; t = **new** Temp(); L.addr = **new** Temp(); gen(t '=' E.addr '*' L.type.width); gen(L.addr '=' L_1.addr '+' t);* }

10

SEMANTIC ACTIONS FOR ARRAY REFERENCES

$S \rightarrow$ **id** = E ;	{ *gen(top.get(**id**.lexeme) '=' E.addr);* }
| L = E ;	{ *gen(L.addr.base '[' L.addr ']' '=' E.addr);* }
$E \rightarrow E_1$ + E_2	{ *E.addr = **new** Temp(); gen(E.addr '=' E_1.addr '+' E_2.addr);* }
| **id**	{ *E.addr = top.get(**id**.lexeme);* }
| L	{ *E.addr = **new** Temp(); gen(E.addr '=' L.array.base '[' L.addr ']');* }
$L \rightarrow$ **id** [E]	{ *L.array = top.get(**id**.lexeme); L.type = L.array.type.elem; L.addr = **new** Temp(); gen(L.addr '=' E.addr '*' L.type.width);* }
| L_1 [E]	{ *L.array = L_1.array; L.type = L_1.type.elem; t = **new** Temp(); L.addr = **new** Temp(); gen(t '=' E.addr '*' L.type.width); gen(L.addr '=' L_1.addr '+' t);* }

1. Get symbol info into L_1.array
2. Get type into L_1.type.elem
3. L.addr is a new temporary
4. Compute address into L.addr (note: L.type.width is size of whole row, plane, etc)

11

SEMANTIC ACTIONS FOR ARRAY REFERENCES

$S \rightarrow$ **id** = E ;	{ *gen(top.get(**id**.lexeme) '=' E.addr);* }
| L = E ;	{ *gen(L.addr.base '[' L.addr ']' '=' E.addr);* }
$E \rightarrow E_1$ + E_2	{ *E.addr = **new** Temp(); gen(E.addr '=' E_1.addr '+' E_2.addr);* }
| **id**	{ *E.addr = top.get(**id**.lexeme);* }
| L	{ *E.addr = **new** Temp(); gen(E.addr '=' L.array.base '[' L.addr ']');* }
$L \rightarrow$ **id** [E]	{ *L.array = top.get(**id**.lexeme); L.type = L.array.type.elem; L.addr = **new** Temp(); gen(L.addr '=' E.addr '*' L.type.width);* }
| L_1 [E]	{ *L.array = L_1.array; L.type = L_1.type.elem; t = **new** Temp(); L.addr = **new** Temp(); gen(t '=' E.addr '*' L.type.width); gen(L.addr '=' L_1.addr '+' t);* }

1. Copy L_1.array to L.array
2. Get type of subarray (from L_1.type.elem)
3. Generate a new temp (t)
4. Generate a new temp (L.addr)
5. Create code to calculate t and add it to L.addr (the offset from the base)

12

CONTROL FLOW

- BOOLEAN EXPRESSIONS CAN BE USED
  - TO COMPUTE A LOGICAL VALUE (TRUE/FALSE)
  - TO ALTER CONTROL FLOW
- HERE WE ARE CONCERNED WITH THE LATTER
- CONSIDER THE FOLLOWING GRAMMAR:
- B-> B||B | B&&B | !B | (B) | E REL E | TRUE | FALSE
- REL IS ONE OF THE RELATIONAL OPERATORS (<, <=, ==, !=, >, >=)
- || IS LOGICAL OR, && IS LOGICAL AND, ! IS NOT

13

CONTROL FLOW

- DEPENDING ON LANGUAGE SEMANTICS, BOOLEAN EXPRESSIONS MAY NOT NEED TO BE COMPLETELY EVALUATED
- MOST LANGUAGES ALLOW 'SHORT CIRCUIT' EVALUATION (QUIT ONCE YOU KNOW THE RESULT
  - B_1 || B_2 – IF B_1 IS TRUE, WE KNOW THE WHOLE EXPRESSION IS TRUE SO SKIP B_2 EVAL
  - B_1 && B_2 – IF B_1 IS FALSE, WE KNOW EXPRESSION IS FALSE SO SKIP B_2 EVAL

14

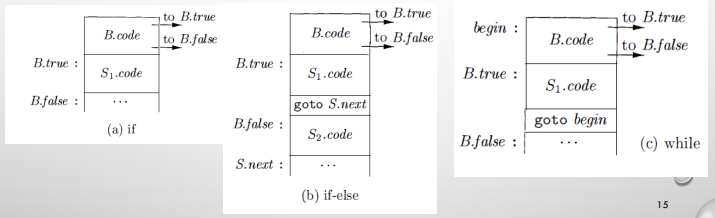
FLOW OF CONTROL STATEMENTS

Here is a small grammar of Flow of Control Statements

$S \rightarrow$ **if** (B) S_1
$S \rightarrow$ **if** (B) S_1 **else** S_2
$S \rightarrow$ **while** (B) S_1

B represents a Boolean expression
S represents statements in the language

Need to create semantic actions that generate the following patterns of code:



15

FLOW OF CONTROL STATEMENTS

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$S \rightarrow$ **if** (B) S_1 **else** S_2	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('goto'\ S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$

In this and following slides:

- B.true, B.false, S.next, S_1.next, etc are labels for branch transfers
- **B.true** – Branch here when B is true
- **B.false** – Branch here when B is false
- **S.next, S_1.next** – This is the target of the next statement after S, S1, etc

16

FLOW OF CONTROL STATEMENTS

PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **while** (B) S_1	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \mid\mid B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

17

BOOLEAN EXPRESSIONS

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1$ **||** B_2	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1$ **&&** B_2	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow$ **!** B_1	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr$ **rel**$.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$
$B \rightarrow$ **true**	$B.code = gen('\texttt{goto}'\ B.true)$
$B \rightarrow$ **false**	$B.code = gen('\texttt{goto}'\ B.false)$

B || B

1. Target true result of B_1 to overall result (short circuit)
2. False result needs a label
3. B_2.true goes to overall true result target
4. B_2.false goes to false target of B
5. Code is B_1 eval code, the label B1.false, and the B_2 eval code

B && B

Similar to B || B but reverse false and true evaluation targets

18

BOOLEAN EXPRESSIONS

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1$ **||** B_2	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1$ **&&** B_2	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow$ **!** B_1	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr$ **rel**$.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$
$B \rightarrow$ **true**	$B.code = gen('\texttt{goto}'\ B.true)$
$B \rightarrow$ **false**	$B.code = gen('\texttt{goto}'\ B.false)$

!B

Just reverse true and false labels from B_1!

19

AVOIDING REDUNDANT GOTOS

- CAN REDUCE GOTOS BY CLEVER REORGANIZATION OF TESTS AND CONTROL TRANSFERS
- CONSIDER: S -> IF (B) THEN S_1 => ACTIONS ARE:
- B.TRUE = NEWLABEL()
- B.FALSE = S_1.NEXT = S.NEXT
- B.CODE II LABEL(B.TRUE) II S_1.CODE
- NOW, USE A NEW OPERATOR ***FALL*** MEANING *DO NOT GENERATE A GOTO*
- S-> IF (B) THEN S_1 => ACTIONS ARE NOW:
- B.TRUE = FALL
- B.FALSE = S_1.NEXT = S.NEXT
- B.CODE II S_1.CODE

20

AVOIDING REDUNDANT GOTOS

- CAN REDUCE GOTOS BY CLEVER REORGANIZATION OF TESTS AND CONTROL TRANSFERS
- CONSIDER: S -> IF (B) THEN S_1 ELSE S_2 => ACTIONS ARE:
- B.TRUE = NEWLABEL()
- B.FALSE = NEWLABEL()
- S_1.NEXT = S_2.NEXT = S.NEXT
- S.CODE = B.CODE || LABEL(B. TRUE) || S_1.CODE || GEN('GOTO' S.NEXT) || LABEL(B.FALSE) || S_2.CODE
- NOW, USE A NEW OPERATOR ***FALL*** MEANING *DO NOT GENERATE A GOTO*
- S-> IF (B) THEN S_1 ELSE S_2 => ACTIONS ARE NOW:
- B.TRUE = FALL
- B.FALSE = NEWLABEL()
- S.NEXT = NEWLABEL()
- S.CODE = B.CODE || S_1.CODE || GEN('GOTO' S.NEXT) || LABEL(B.FALSE) || S_2.CODE

21

BACKPATCHING

- GENERATING JUMP INSTRUCTIONS ON THE FLY MAY REQUIRE A SECOND PASS TO DETERMINE THE ADDRESS OF LABELS GENERATED.
- **BACKPATCHING** ALLOWS 1 PASS TRANSLATION BY KEEPING LISTS OF JUMP TARGETS CREATED AS SYNTHESIZED ATTRIBUTES
- NEED 3 LISTS:
  - **B.TRUELIST** – INSTRUCTIONS THAT NEED A TARGET WHEN B IS *TRUE*
  - **B.FALSELIST** – INSTRUCTIONS THAT NEED A TARGET WHEN B IS *FALSE*
  - **S.NEXTLIST** – INSTRUCTIONS THAT NEED TO JUMP TO THE INSTRUCTION AFTER THE CODE IN S.
- 3 FUNCTIONS ARE USED:
  - **MAKELIST(I)** – CREATES A NEW LIST CONTAINING ONLY I, AN INDEX INTO THE INSTRUCTION LIST
  - **MERGE(P1,P2)** – MERGES TWO LISTS AND RETURNS THE MERGED LISTS
  - **BACKPATCH(P, I)** – PATCHES (FIXES) THE TARGETS OF ALL CONDITIONAL AND UNCONDITIONAL JUMPS IN THE LOCATIONS FOUND IN LIST P TO POINT AT INSTRUCTION I.

22

BACKPATCHING – BOOLEAN EXPRESSIONS

1) $B \rightarrow B_1$ || M B_2 { $backpatch(B_1.falselist, M.instr)$; $B.truelist = merge(B_1.truelist, B_2.truelist)$; $B.falselist = B_2.falselist$; }
2) $B \rightarrow B_1$ && M B_2 { $backpatch(B_1.truelist, M.instr)$; $B.truelist = B_2.truelist$; $B.falselist = merge(B_1.falselist, B_2.falselist)$; }
3) $B \rightarrow$! B_1 { $B.truelist = B_1.falselist$; $B.falselist = B_1.truelist$; }
4) $B \rightarrow$ (B_1) { $B.truelist = B_1.truelist$; $B.falselist = B_1.falselist$; }
5) $B \rightarrow E_1$ **rel** E_2 { $B.truelist = makelist(nextinstr)$; $B.falselist = makelist(nextinstr + 1)$; $emit(\texttt{'if'}\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ \texttt{'goto\ \_'})$; $emit(\texttt{'goto\ \_'})$; }
6) $B \rightarrow$ **true** { $B.truelist = makelist(nextinstr)$; $emit(\texttt{'goto\ \_'})$; }
7) $B \rightarrow$ **false** { $B.falselist = makelist(nextinstr)$; $emit(\texttt{'goto\ \_'})$; }
8) $M \rightarrow \epsilon$ { $M.instr = nextinstr$; }

Marker non terminal picks up location of next instruction generated (see production 8)

Backpatching done during compound expression evaluation

Lists are either reversed or copied in productions 3 and 4

makelist() is needed for productions 5, 6, and 7

23

BACKPATCHING – BOOLEAN EXPRESSIONS

- CONSIDER

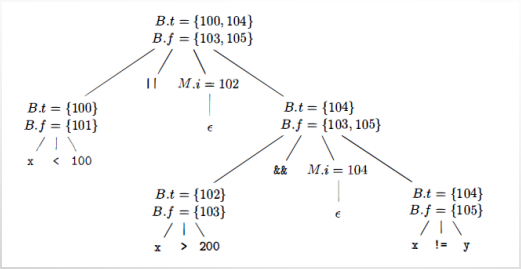
1) $B \rightarrow B_1$ || M B_2 { $backpatch(B_1.falselist, M.instr)$; $B.truelist = merge(B_1.truelist, B_2.truelist)$; $B.falselist = B_2.falselist$; }

- IF B_1 IS TRUE, CONTROL CAN JUMP PAST THE TEST IN B_2
- HOWEVER, IF IT IS FALSE, IT MUST JUMP TO THE TEST IN B_2 IN ORDER TO TEST THE COMPLETE CONDITIONAL.
  - SO THE BACKPATCH() OPERATION CAUSES ALL JUMPS ON THE FALSE LIST TO POINT AT M (M.INSTR) WHICH WILL BE THE START OF THE CODE IN B_2
  - MEANWHILE, THE TRUELIST IS SET TO THE COMBINATION OF TRUELISTS FOR B_1 AND B_2 SINCE BOTH OF THOSE MEAN THE OVERALL EXPRESSION IS TRUE.
  - FINALLY, B'S SYNTHESIZED FALSELIST SHOULD BE WHEREEVER THE FALSELIST FOR B_2 POINTS.

24

BACKPATCHING EXAMPLE

Parse tree for: x < 100 || x > 200 && x ! = Y



Generates:
100: if x < 100 goto –
101: goto –
102: if x > 200 goto –
103: goto –
104: if x != y goto –
105: goto –

25

BACKPATCHING FLOW OF CONTROL

1) $S \rightarrow \textbf{if}\,(B)\,M\,S_1$ { *backpatch*(*B.truelist*, *M.instr*); *S.nextlist* = *merge*(*B.falselist*, *S*$_1$*.nextlist*); }

2) $S \rightarrow \textbf{if}\,(B)\,M_1\,S_1\,N\,\textbf{else}\,M_2\,S_2$ { *backpatch*(*B.truelist*, *M*$_1$*.instr*); *backpatch*(*B.falselist*, *M*$_2$*.instr*); *temp* = *merge*(*S*$_1$*.nextlist*, *N.nextlist*); *S.nextlist* = *merge*(*temp*, *S*$_2$*.nextlist*); }

3) $S \rightarrow \textbf{while}\,M_1\,(B)\,M_2\,S_1$ { *backpatch*(*S*$_1$*.nextlist*, *M*$_1$*.instr*); *backpatch*(*B.truelist*, *M*$_2$*.instr*); *S.nextlist* = *B.falselist*; *emit*('goto' *M*$_1$*.instr*); }

4) $S \rightarrow \{\,L\,\}$ { *S.nextlist* = *L.nextlist*; }

5) $S \rightarrow A\;;$ { *S.nextlist* = **null**; }

6) $M \rightarrow \epsilon$ { *M.instr* = *nextinstr*; }

7) $N \rightarrow \epsilon$ { *N.nextlist* = *makelist*(*nextinstr*); *emit*('goto _'); }

8) $L \rightarrow L_1\;M\;S$ { *backpatch*(*L*$_1$*.nextlist*, *M.instr*); *L.nextlist* = *S.nextlist*; }

9) $L \rightarrow S$ { *L.nextlist* = *S.nextlist*; }

26

QUESTIONS?

27