Semantic Action notes

Dan Beatty

November 29, 2004

Generic Quad Generation: $E \rightarrow TE'$ $E' \rightarrow +TE'$ $E' \rightarrow \epsilon$ $T \rightarrow FT'$ $T' \rightarrow *FT'$ $T' \rightarrow \epsilon$ $F \rightarrow (E)$ $F \rightarrow i_d$

a := pop b := pop c := gentemp genquad (+, b,a,c) push (c)

a := pop b := pop c := gentemp genquad (*, b, a, c) push (c)

push (id)

(a + b) * 3

The task for this to generate semantic actions for this somewhat simple grammar and another example will be used to reinforce the principle. (Time index 2:39) Notice in this grammar several semantic actions are performed. The order of traversal and the actions taken matter. These semantic actions fill the semantic action stack (SAS) and fills temporary variables NQ (next quad), a,b, and c in these actions. The operation genquad has a side effect in advancing the NQ counter. Also t2 is left on the stack after all pushed and popped.

General Rules Every time that gen quad is called, there is an implicit incrementing of next quad (NQ). Unique integers must exist to provide quad-table operations. Integer locations for local variables must also exist, which are links to the symbol table (lexical analyzer). Negative integers may a good suggestion for temporary variables, or members of the symbol table. If it is a member of the symbol table, then gentemp must add that temporary variable reference to the symbol table. There is an important separation that exist between the op code of the quad table and the reserved words of the symbol table. A similar separation exist for the quad-table op code and the machine op code.

The operations include jump, jump equal, jump not equal, plus, and multiply.

Any type of expression will leave its result on the SAS. For N pops there are N+1 pushes for each expression evaluated. This forces an asymmetric relationship between push and pop.

Any type of statement will leave the stack empty as far as the statement is concerned. When any type of statement is processed (such read, write, for, while, etc), the processing starts with the

assumption that the stack is empty, regardless if it is, and finish processing with assumption that the stack is empty. The relationship on push pop is $N_{pop} \geq N_{push}$.

SAS a b

local variable a 1 11 local variable b I local variable c t1 NQ 10 11 12 13 33

Example: For Loop $S \rightarrow ...|fori_d := E\ to\ E\ do\ S\ od$

The state of the stack before semantic action C (before statement S) is the same as after statement S.

There needs to be a semantic action A, to identify the identifier for the loop statement. Semantic action needs to follow the first expression to collect the result from the expression, E.

Also to be noted, when genquad has an empty operand, that operand is represented with a zero.

Semantic action A push (id)

Semantic action B a := pop (result of expression RHS) b := pop (id LHS) c := genquad (:= , a, , b) push(b)

Semantic action C c := gentemp a := pop (result of expression) b := pop (id) push (b) push (NQ) genquad (¡= , a, b, c) genquad (jeq, c, false, _)

Semantic action D a := pop (NQ) b := pop (id) genquad (+, id, t, id) genquad (jmp, , , a) QUADS[a+1,4] := NQ

Sample for i := 1 to 10 do S: S ... od

a 1 11 b I c t1 NQ 10 11 12 13 33

Note: this is the NQ before the genquad statement which increments NQ.

10 11 12 13

$S \rightarrow ID\ :=\ E$

Semantic action A push (A)

Semantic action B a := pop b := pop genquad (:= , a, , b)

Another example *while C do S od*

Semantic action A a := pop push (NQ) genquad (jeq, a, false, _)

Semantic action B a := pop genquad (jmp, _ , _ , a) QUADS [a, 4] := NQ

a NQ

Example at 1:15:07

Parsing is started with E (the start symbol).

Given this expression from the grammar, $(E_L)(E_L)+$, try to make a semantic action

A at beginning, B inbetween)(, and C at)+. There are two stacks required to aid in this semantic action, in addition to the SAS. We use the marker # to indicate that results are about to be generated by an expression. These results can be lists of results, and they are pushed onto the SAS. The marker is used in subsequent semantic actions such as B to indicate a stop condition. The two addition stacks introduced as S_1 and S_2 are for storage of the results of each expression list. Pushes and pops are taken with regard to their respective stack.

Also are included local variables, A, and B.

Note that this mechanism allows EL to be a list, and for those elements to be added and stored as a list.

Semantic action A push (#)SAS

Semantic action B has

- A = pop(SAS)
- while A *neq* # do
  - push (A, S_1)
  - A := pop (SAS)
- push (#) SAS

Semantic action B A = pop(SAS) while A *neq* # do push (A, S_1) A := pop (SAS) push (#) SAS

Semantic action C has

- A := A := pop(SAS)
- while A $\neq$ # do
  - push(A,SZ)
  - A := pop (SAS)
- push (#) SAS

- A := pop(S1)
- B := pop (S2)
- while A $\neq$ empty do
  - C := gentemp
  - genquad (+, A, B, C)
  - push (C) SAS
  - A := pop (S1)
  - B := pop (S2)

Example: (1,2,3) (4,5,6) +

Note that a test of B in addition to A in semantic action C can indicate a semantic error.

Given: Given this statement from the grammar, $(I_L)(E_L)$:=, try to make a semantic action

(IL)(EL) :=

The first action pushes a marker on to the SAS to indicate a stop condition, which is semantic action A. The next step is to retrieve the results of EL off of the stack and preserve them in their own stack.

Presumption: IL puts its list of elements on the SAS. These elements can be retrieved as shown in semantic action B. In semantic action B, these elements are placed on their own stack, the results of EL are also placed on their own stack, and the results are popped together. As each element is popped, the element of EL is assigned to the element of IL and eventually the production is complete.

A at the beginning has Semantic Action A

- push (#)

B at the end of the production Semantic Action B

- A := pop (SAS)
- while A $\neq$ # do
  - push (A,S1)
  - A := pop (SAS)
- A := pop(SAS)
- while A $\neq$ # do

  - push (A, S2)
  - A := pop (SAS)
- A := pop (S1)
- B := pop(S2)
- while A $\neq$ # empty do
  - genquad (= A, _ , B)
  - A := pop(S1)
  - B := pop(S2)

(x,y,z) ((1,2,3)(4,5,6)+) :=

Another production: $\{SL\}$ id $from$ $[E$ .. $E]$ This production is found in S' which is called by production $S \rightarrow \alpha_{12} S_L \alpha_1 3 S'$. This production may produce some difficulty. The meaning of this production is execute the statements in this statement list with some identifier taking values from some expression to another expression stated in a bracketed list. There is an analogous to a for loop such that

for id := E1 to E2 SL This implies that this statement is handled the same way as a for loop.

Symbol Table - Quad Table

Production quality software often times approaches large data use in terms of iterative calls. Every recursive call places a number things on a stack, including activation records and their activation record stack. As a result, stack management becomes an issue if recursive calls dominate the software package. The example grammar has calls to many components and the semantic action stack and other stacks grow quickly.

There is a link between the symbol table and quad table. Constants and identifiers are pointers into the symbol table.

Question: How do we handle arrays?

(X(1)) ((a(1)) (a(2))) :=

Suggestions:

- Offsets. Note that offsets require a good deal of knowledge about the machine, and quads are supposed to be machine independent.
- A header to linked lists of arguments that make up operand 1 and 2.
- Add fields to the quad table. The obvious problem is that there is a limit to the number of subscripts that user can be allowed to use. There are languages that limit the number of

subscripts one can have to an array, and the reason may be how the quad table addresses them.

Offsets. Note that offsets require a good deal of knowledge about the machine, and quads are supposed to be machine independent. A header to linked lists of arguments that make up operand 1 and 2. Add fields to the quad table. The obvious problem is that there is a limit to the number of subscripts that user can be allowed to use. There are languages that limit the number of subscripts one can have to an array, and the reason may be how the quad table addresses them.

Memory segment/ block

Object Code

integer array

data

Activation records → Heaps Compiler Run Time

Code generation takes the Symbol Table and Quad Table and uses them to generate object and data sections. Quad tables support data flow optimization.

Error catch can occur in one of two places Compiler Run Time OS Run Time

Garbage collection serves two purposes:

- It covers for sloppy code
- It handles checkerboard type memory management within the process
- Prevents memory writes from overwriting object code or other critial sections.

Example The old scheme of IBM's database system included a system called IMS DB/DC. It was a hierarchical database, and did not survive in use much beyond 1985. It used COBOL queries which were translated into OS 370 operation code, and that code managed the database. One error noted occurred when a program was run that produced a boundary condition error. In the process the program overwrote sections of the object code itself. With no boundary protection for data, heap and object code sections, the program can otherwise destroy itself. The typical error for this at the time IBM was using this scheme was unrecognizable operation code, which was an assembler code error.

1 Semantic Actions Positions

$$P \rightarrow \alpha_0 D_1 \alpha_{12} S_L \alpha_{13}$$

$$D_1 \rightarrow I_L \alpha_{31} D | \epsilon$$

$$D \rightarrow \alpha_1 \alpha_{14} cons D_2 \alpha_{15} \alpha_{16} D_1 | \alpha_2 \alpha_{16} D_1$$

$$D_2 \rightarrow \alpha_{17} cons D_2 | \epsilon$$

$$I_L \rightarrow \alpha_{-1} I_L$$

$$I_{L_1} \rightarrow \alpha_{17} I_L | \epsilon$$

$$I_D \rightarrow i_d I'_D$$

$$I'_D \rightarrow [S_u] | \epsilon$$

$$T \rightarrow I_D | \alpha_{11} | cons$$

$$S_u \rightarrow i_d S'_u | cons S''_u$$

$$S'_u \rightarrow i_d \alpha_{17} S_u$$

$$S''_u \rightarrow \alpha_{-2} \alpha_{17} S_u$$

$$S \rightarrow \alpha_{12} S_L \alpha_{13} S' | \alpha_5 | \alpha_6 | \alpha_{18} I_L \alpha_{19} S''$$

$$S' \rightarrow \alpha_7 C | \alpha_8 C | i_d \alpha_9 \alpha_{14} E \alpha_{10} E \alpha_{15}$$

$$S'' \rightarrow \alpha_{18} E_L \alpha_{19} \alpha_{20} | \alpha_3 | \alpha_4$$

$$E \rightarrow \alpha_{18} E_L \alpha_{19} \alpha_{18} E_L \alpha_{19} E' | T$$

$$E' \rightarrow \alpha_{27} | \alpha_{28} | \alpha_{29} | \alpha_{30} | \alpha_{35}$$

$$E_L \rightarrow E E'_L | \epsilon$$

$$E'_L \rightarrow \alpha_{17} E_L$$

$$C \rightarrow (E_L)(E_L) C'' | \epsilon$$

$$C' \rightarrow \alpha_{16} C C'''$$

$$C'' \rightarrow \alpha_{21} C' | \alpha_{22} C' | \alpha_{23} C' | \alpha_{24} C' | \alpha_{25} C' | \alpha_{26} C'$$

$$C''' \rightarrow \alpha_{32} C' | \alpha_{33} C' | \alpha_{34} C'$$

$$S_L \rightarrow S S'_L$$
$$S'_L \rightarrow \alpha_{16} | \epsilon$$

Primary Semantic Actions

$$P \rightarrow program\ D_1 \{ S_L \}$$

$$S \rightarrow \{ S_L \} S' | rdln | wrln | (I_L) S''$$

$$S' \rightarrow when\ C | until\ C | i_d\ from\ [E \alpha_{10} E]$$

$$S'' \rightarrow (E_L) := | read | write$$

$$E \rightarrow (E_L)(E_L)E'|T$$

$$E' \rightarrow + \quad |- \quad |* \quad |/ \quad |mod$$

$$E_L \rightarrow EE'_L|\epsilon$$

$$E'_L \rightarrow, E_L|\epsilon$$

$$C \rightarrow (E_L)(E_L)C''$$

$$C' \rightarrow ; CC'''|not\ C'|\epsilon$$

$$C'' \rightarrow < C'| > C'| = C'| <= C'| >= C'| <> C'$$

$$C''' \rightarrow andC'|orC'$$

$$S_L \rightarrow SS'_L$$

$$S'_L \rightarrow ; |\epsilon$$

1.1 Semantic Actions on S'

Algorithm 1 Semantic action G_0

```
A := POP (SAS)   Work Section Quad
B := POP (SAS)   Next Quad from {S_L}S'
QT[B,4] := NQ
PUSH (A)
```

Algorithm 2 Semantic action G_1 (when C' decision)

```
c := POP (SAS)   result of C
b := POP (SAS)   work quad
d := POP (SAS)   exit quad
genquad (jeq, C, F, b)   jump to work
quads [d] results := NQ   adjust exit quad jump
```

Algorithm 3 Semantic action G_2 (until jump condition)

```
C := pop (SAS)   condition result
a := pop (SAS)   work quad
genquad (jeq, C, F, a)   jump condition for until
```

Algorithm 4 Semantic action G_3 (until test condition)

```
a := pop (SAS)  work quad
 b := pop (SAS)  start result
quads [b] results := NQ
push (a)  preserve work quad
```

Algorithm 5 Semantic action G_4

push (id)

Algorithm 6 Semantic action G_5

`c := pop(SAS)` (result of expression RHS)
`d := pop(SAS)` (id LHS)
`b := pop(SAS)` (work quad)
`a := pop(SAS)` (start quad)
`quads [a] result := NQ` (adjust start jump to initialization section)
`e := NQ` preserve jump to test location
`genquad (jmp, _ , _ , _)`
`genquad (:=, c, _ , d )` initialize id with the value of the expression (presumption of single element expression result).
`push (e, SAS)` preserve test jump
`push (d)` preserve id
`push(b)` preserve work jum

Algorithm 7 Semantic action G_6

`f := gentemp` Expression result
`b := pop (SAS)` work jump point
`d := pop (SAS)` id
`e := pop (SAS)` test jump point
`quads[e] result := NQ` adjust test jump point to this test
`g := gentemp` temporary variable
`genquad ( <= , d, f, g)`
`genquad ( + , d , 1, d )`
`genquad ( jeq, c, true , b )`

Algorithm 8 Semantic action G_7

```
a := NQ
genquad(jmp, _ , _, _ )
b := NQ
push (a, SAS)
push (b, SAS)
```

1.2 Condition and Expression Semantic Actions

Algorithm 9 Semantic action W_1

```
push (#)
```

Algorithm 10 Semantic action W_2

```
a := pop (SAS)
while a ≠ # do
   push (a, S1 )
    a := pop (SAS)
```

Algorithm 11 Semantic action X_o

Require: Op Code to be generated: O

```
a := pop (SAS)
while a ≠ # do
   push (a, S1 )
   a := pop (SAS)
 push (#)
 a := pop (S1)
 b := pop (S2)
while a ≠ ∅ do
   c := gentemp
   genquad (O, a, b, c)
   push (c, SAS)
   a := pop (S1)
   b := pop (S2)
```

Algorithm 12 Semantic action X_{not}

```
 a := pop (S1)
while a ≠ ∅ do
   c := gentemp
   genquad (O, a, b, c)
   push (c, SAS)
   a := pop (S1)
```

1.3 Declarations

Algorithm 13 Semantic Action for I_0

push (#, SAS)

Algorithm 14 Semantic Action for I_1

a := int
b := pop (SAS)
while ($b \neq \#$)
    change symbol table entry b's type to the value of a
    b := pop (SAS)

1.4 Constants and Lists

$$T \rightarrow I_D | \alpha_{11} | cons$$

$$S_u \rightarrow i_d S'_u | cons S''_u$$

$$S'_u \rightarrow ;\ i_d\ S_u$$

$$S''_u \rightarrow ;\ \alpha_{-2} S_u$$

$$S'' \rightarrow (E_L)\ :=\ |\ read\ |\ write$$

Algorithm 15 Semantic Action for I_2

```
a := id
push (a, SAS)
```

Algorithm 16 Semantic Action for I_3

```
push (# , SAS)
push ( cons, SAS)
```

Algorithm 17 Semantic Action for I_4

Require: A temporary dimension structure temp-dim

```
a := pop(SAS)
while ( a ≠ # )
    tempdim adddim: a
    pop a
b := pop (SAS)
while ( b ≠ #)
    ST setdimension: tempdim in_symbol:b
    b := pop (SAS)
```

Algorithm 18 Semantic Action for I_5

```
push ( cons, SAS)
```

Algorithm 19 Semantic Action for I_6

```
push ( cons, SAS)
```

Algorithm 20 Semantic Action for I_7

```
A := pop (SAS)
while A ≠ # do
    push (A,S1)
    A := pop (SAS)
A := pop(SAS)
while A ≠ # do
    push (A, S2)
    A := pop (SAS)
A := pop (S1)
B := pop(S2)
while A ≠ # empty do
    genquad ( = A, _ , B)
    A := pop(S1)
    B := pop(S2)
```