Semantic Actions

Notes by Dan Beatty

Semantic Actions are embedded with the syntax analysis functions capturing the semantics of the program.. As syntax is being analyzed, a semantic representation of the program is produced. There is such a thing as a quad table, and this table captures the meaning of the program in a concise way.

Semantic actions are associated with the syntax production rules they accompany. Semantics are an intricate part of a languages constructs. Furthermore, some semantics can define the very paradigm that a language represents.

The main semantic types shared in common with most languages are:

1. Expression statements
2. Conditional statements
3. Flow Control Statements
4. Declaration types
   (a) Identifier
   (b) Identifier Lists
   (c) Array Sub lists (indices).

Two benefits of using intermediate - machine independent form:

1. Retargeting is facilitated; a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2. A machine-independent code optimizer can be applied to the intermediate representation.

> Three address code is a linearized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph. The syntax tree and dag in Figure **??** are represented by the three-address code sequences in Figure [**?**]. Variable names can appear directly in three-address statements, so Figure **??** (a) has no statements corresponding to the leaves in figure **??**.

A quad table is a two dimensional table containing integers. The references are to the symbol table. References to quads which will be the row subscript for the quad in the quad table which may be any branch (loops or conditional branches).

Quad operations may be represented by 4 columns (operation, operand 1 and 2, and the result). The operands, and results may be references to the symbol table or the quad table. The operations must include Turing complete operations. The operations are referenced by integers mapped to specific operations.

Table 1: Turing Operations included in most Quad Tables

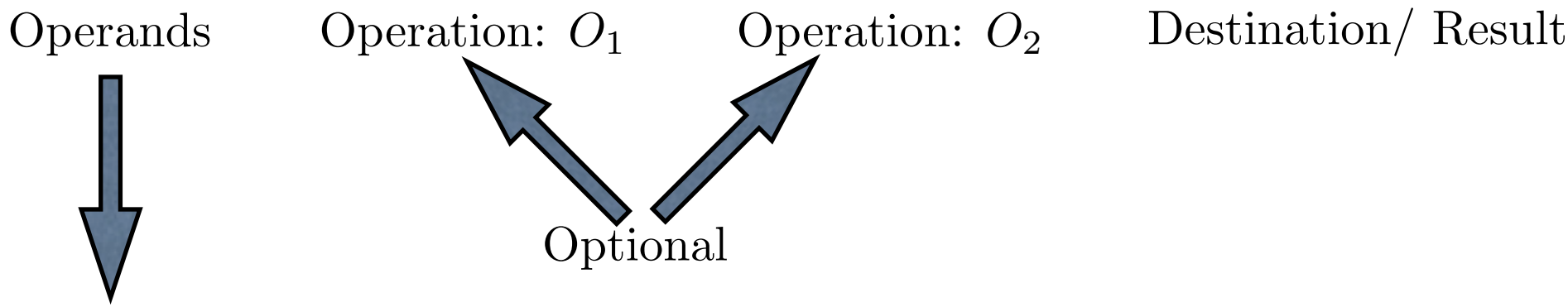
+	1
-	2
	3
/	4
<	5
>	6
>=	7
<=	8
read	9
write	10
readln	11
writeln	12
eq	13
neq	14
jmp	15
jmp-true	16
jmp-false	17
assign (:=)	18

The term "three address code" is defined by three addresses (operand 1, operand 2, and the result field). The other element is the operation code. Typically an user defined name is replaced by a symbol table reference. The operator is a type of Three-Address Statement.

1. Assignment (copy)
2. Unary operator
3. Binary operator
4. Unconditional jump
5. procedural calls (branches) and their parameters
6. indexed assignments
7. Address or pointer assignments

Semantic analysis

Byte code / Quads are machine independent representation of a language. An example of a quad type language is Sun's Java Byte code. Each quad has 4 parts to it. Operations, 2 operands, and Destination/ Result of quad operation.



Operations are equivalent to

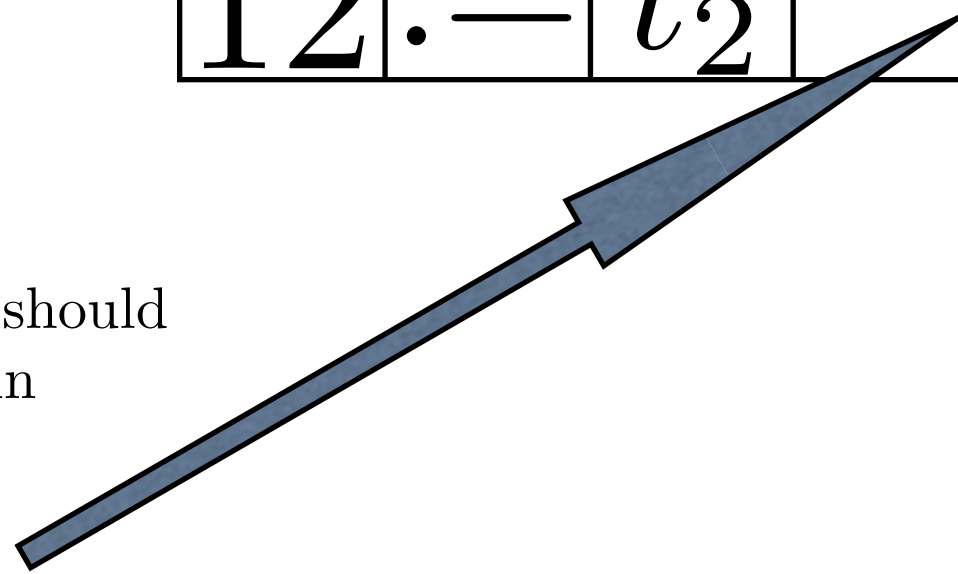
Languge Constructs → what the computer can do

Input / Output	read	write	readln	writeln				
Arithmetic Instructions	:=	-	+	*	/	mod		
Decision Flow	>	<	=	≤	≥	≠	and	or	not

Quad Example

$$\text{x} := (a + b) * 3$$

Line Number	Operation	O_1	O_2	Results / Destination
10	+	a	b	t_1
11	*	t_1	3	t_2
12	:=	t_2		X



Note that at line 12, a human should notice an optimization to be had in load directly the result into x.
Also, there are no registers.
All memory is symbolic.

This quad language allows the constructs of a language to captured in a machine independent and concise way.

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

Two basic attributes associated with either expression or statement statements to be encoded into quads are:

1. Place
2. Code

Generic Quad Generation Continued

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \quad (A)$$

$$E' \rightarrow \epsilon$$

$$T \rightarrow FT'$$

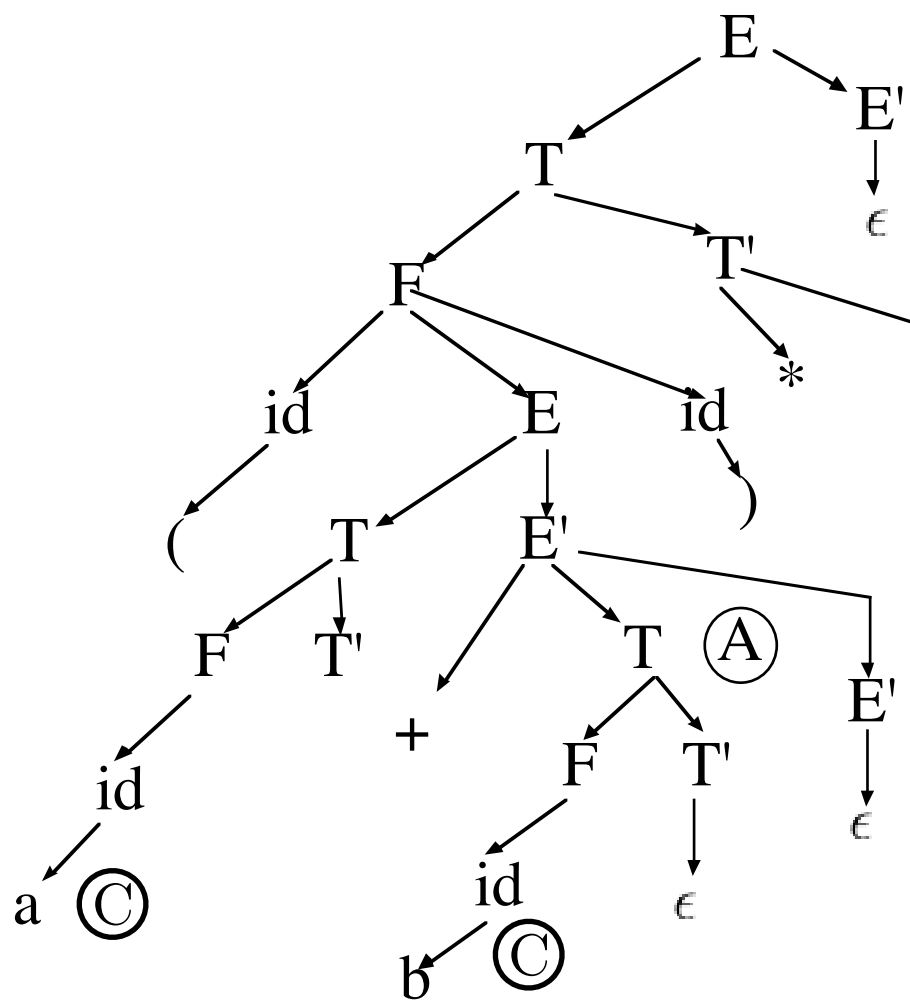
$$T' \rightarrow *FT' \quad (B)$$

$$T' \rightarrow \epsilon$$

$$F \rightarrow (E)$$

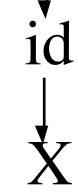
$$F \rightarrow i_d \quad (C)$$

(a + b) * X



Line Number	Operation	O_1	O_2	Results / Destination
10	+	a	b	t_1
11	*	X	t_1	t_2
12	:=	t_2		X

B



C

NQ	SAS
10	a
11	b
	t1
	X
	t2

local	b	X	
local	a	t1	
local	t1	t2	
NQ	10 -> 11	11	

The task here is 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.

Classic expression language example

$E \rightarrow E + T | T$

Ⓑ

$T \rightarrow T * F | F$

Ⓐ

$T \rightarrow (F) | i_d$

Ⓑ
a:= pop (F)
b:= pop (T)
c:= gentemp
genquad ($*$, a, b, c)
push (c)

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

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 $N_{pop} \geq N_{push}$.

The place attribute places a name that will hold the value of the statement/ expression. In Dr. Cooke's lingo, this is a gentemp command. The code attribute generates a quad, or sequence of quads. Statements have the additional attributes:

- **begin:** Start flow control statements
- **after:** End a previously generated flow control statement

Flow control generates labels which are reminscient of the labels in assembly.

1 Symbol Table

The symbol table was introduced in the lexical analyzer to store identifiers and constants. It is also used to differentiate reserved words and symbols from identifiers. In the syntax analyzer (parser), the token type is used to ascertain proper symbols where they should be in the productions. Now in the intermediate code generation, the majority of properties in the symbol table come to bear on the issue of translating a given grammar into a simple set of regular expressions.

A quad table is nothing more than a 2-D array of symbolic information. These symbols can be represented as integers. The operators themselves do not come from the symbol table. However, the operands such as identifiers and constants do. Jumps references are to rows in the table. A value has to be made available for true or false.

It is recommended that we use a 2-D integer array to represent the "Semantic routines" and that we have a routine to print out the semantic / quad tables. It is also recommended that we have a quad generation routine. Temporary variables DO NOT go into the symbol table. Quad Examples: arithmetic language. Consider using negative integers for the temporary variables.

genquad (_, _, _, _)

temp -

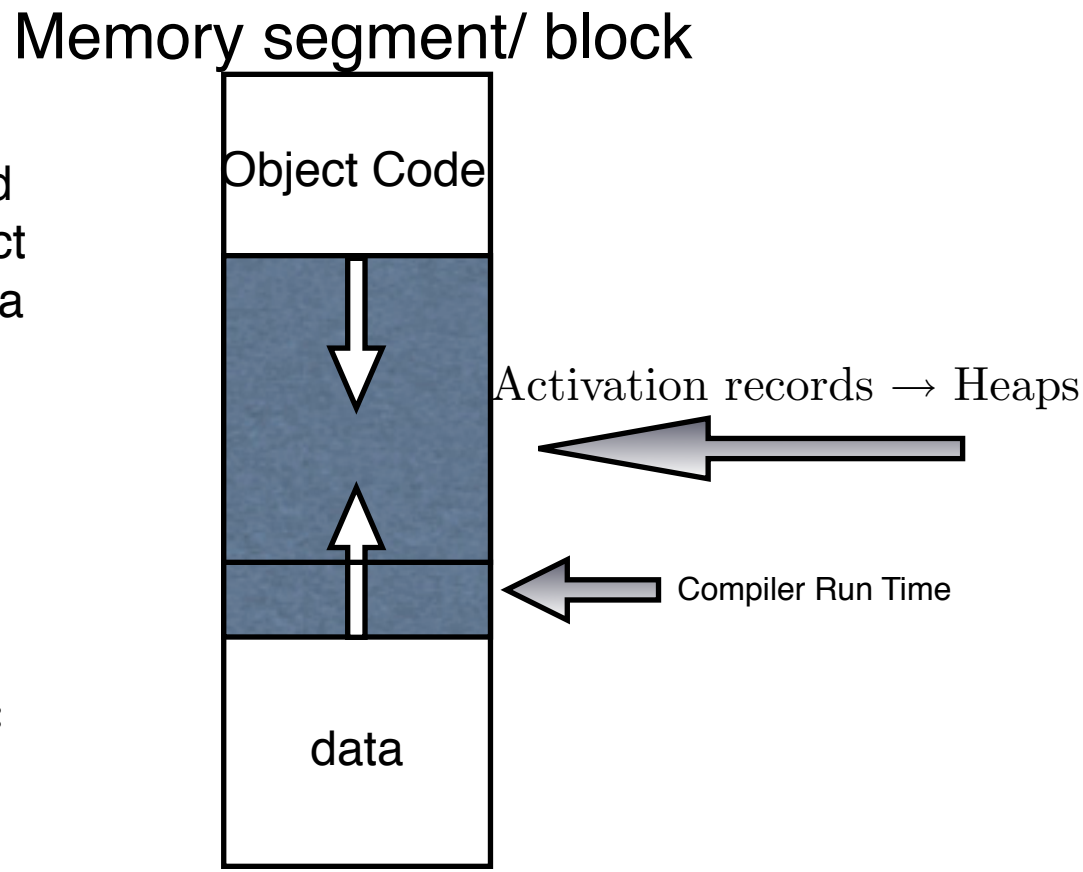
nextquad

semantic action stack

Symbol Table - Quad Table

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

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



Errors 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.

1.1 Names in the Symbol Table

1. "Leximes for all tokens are acquired into ST by the lexical analyzer. " [**?**] The ST has a operator called lookup necessary for finding these leximes.

2. Semantic actions has an operation called emit (gen_quad). This operation implants basic operations into the quad table.

3. When the statement forming a procedure body is examined, a pointer to the symbol table for the procedure appears on top of the table stack.

4. Given a production for a procedure:

$$D \rightarrow \textbf{proc id ;} N D_1 ; S$$

   "An names in an assignment generated by S must have been declared in either the procedure that S appears in, or in some enclosing procedure. " [**?**] In other words, the scope of statement determines the relevant symbol table paths.

5. There are critical semantic actions to take in the case of statements involving assignments, and Turing Essential Operations. Ordering is determined by syntax, and the calls then determine the semantics. (Reference figure 8.15 [**?**].

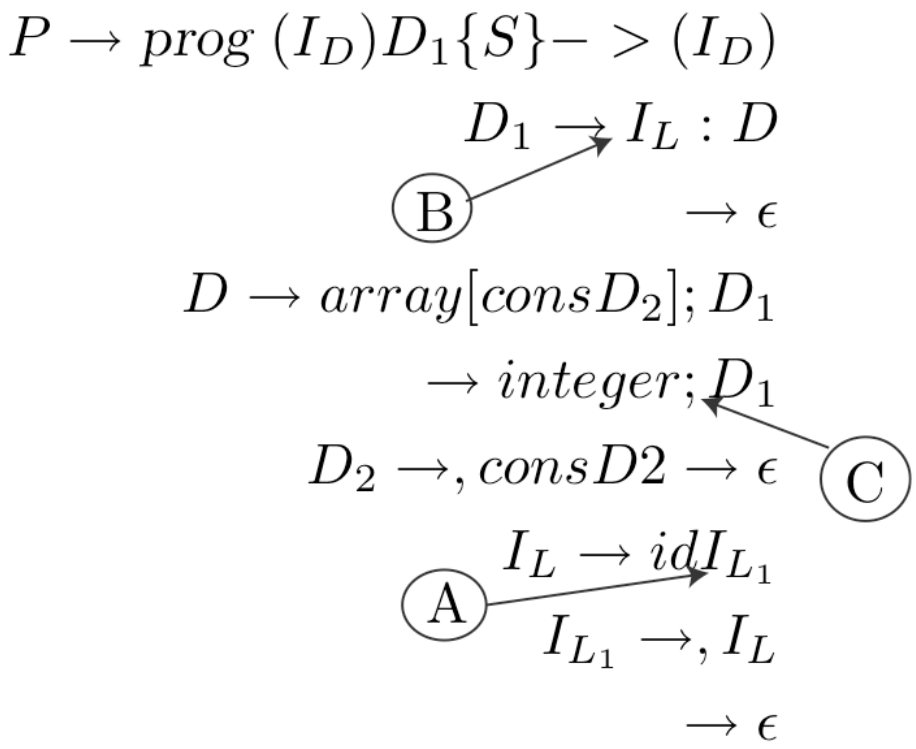


Figure 1: Declaration Semantics Example

Declaration statements primarily store information in the symbol table. The entries are initiated when a symbol is first detected in the tokenizer. The parser supplies the context of such a declaration. Declarations may include:

1. Identifiers
2. Identifier Lists
3. Array Sub lists (indices).

These three types have semantic actions that are required to obtain symbol entries from the parser. Once the parser indicates a enough information is acquired, then that information is used to augment the symbol table. Note that the quad table is not directly effected by a declaration statement.

1.3 Addressing Array Elements

1. "Elements of an array can be accessed quickly if the elements are stored in a block of consecutive locations." The equation is

$$A[i] = A_{base} + (i - \textit{lower-bound}) \times \textit{width}$$

Various means of computing this equation in the semantic actions can be performed depending on how the declaration of the array is handled. For example, the algebraic rewrite of $A[i]$ is

$$A[i] = A_{base} - \textit{lower-bound} \times \textit{width} + i \times \textit{width}$$

The first sum

$$C = A_{base} - \textit{lower-bound} \times \textit{width}$$

2. "Compile-time precalculation can also be applied to address calculations of elements of multi-dimensional arrays." This is a classic representation issue that must be addressed in most stages of language and compiler design. It can obviously be ignored in lexical analysis.

   In most cases, it is simply a matter of define the following parameters

   - Lower bound
   - Fastest Variance in Subscripts

3. A more significant issue in intermediate code generation is generating references for array(s) and their indices.

    - Left most expression semantic action association:
      This idea binds the semantic actions of the array to the first index of the list as opposed to the array identifier itself.
    - Number of dimensions and array of dimension values must be two properties of the symbol table record(s).
    - Included in the symbol table must be location and offset of the array.

1.5 Type Conversion within Assignment

> In practice, there would be many different types of variables and constants, so the compiler must either reject certain mixed-type operations or generate appropriate coercion (type conversion) instructions.

This issue is not in the grammar that is being presented in Cooke's class, but is common in procedural languages.

Figure 8.19 shows the algorithm of Type Conversion for simple addition expressions. The concept deals with converting arguments into compatible types. Obviously, there is not always a common type between two arguments. This issue is critical as data types are allowed to become more complex.

The example presented by Ullman is relatively simple and present in most commonly used procedural and object oriented languages.

1.6 Accessing Fields in Records

> The compiler must keep track of both the types and relative addresses of the fields of a record. An advantage of keeping this information in symbol-table entries for the field names is that the routine for looking up names in the symbol tables can also be used for field names.

Note that in all of this, words may have been stated about operations based on type. However, indirect memory access was not one of them.

It should be noted that procedural and logic programming have two separate concepts of the boolean or logic expressions in programming. In procedural, the concept of logic is for controlling the flow of the program. Although a procedural program can evaluate a set of boolean expressions to either true or false, and hence be used to evaluate a complex expression, it is not the same as a answer set or other resolution type concepts.

This section examines the procedural component of logic evaluations. Another section in the context of logic oriented semantics handles answer set semantics and reviews existing papers on the subject.

1.3 Declarations

$$D_1 \rightarrow I_L : D | \epsilon$$

I_0

Algorithm 12 Semantic Action for I_0

push (#, SAS)

$$D \rightarrow integer; D_1$$

I_1

Algorithm 13 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)

$$I_L \rightarrow id, I_{L_1}$$

I_2

$$I_{L_1} \rightarrow , I_L | \epsilon$$

$$D \rightarrow array[cons\ D_2]; D_1 | integer;\ D_1$$

I_3 I_4

Algorithm 14 Semantic Action for I_2

a := id
push (a, SAS)

$$D_2 \rightarrow , cons\ D_2 | \epsilon$$

I_5

Algorithm 15 Semantic Action for I_3

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

$$I_D \rightarrow id\ I'_D$$

I_2

Algorithm 16 Semantic Action for I_4

Require: A temporary dimension structure temp-dim
a := pop(SAS)
while ($a \neq \#$)
    tempdim adddim: a
    pop a
b := pop (SAS)
while ($b \neq \#$)
    ST setdimension: tempdim in_symbol:b
    b := pop (SAS)

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

$$I'_D \rightarrow \epsilon$$

Algorithm 17 Semantic Action for I_5

push (cons, SAS)

1.4 Constants and Lists

Algorithm 14 Semantic Action for I_2

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

$$T \rightarrow I_D | \alpha_{11} | cons \quad (I_2)$$

Algorithm 19 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)
```

$$S_u \rightarrow i_d S'_u | cons S''_u \quad (I_2) \quad (I_2)$$

$$S'_u \rightarrow ; \ i_d \ S_u \quad (I_2)$$

$$S''_u \rightarrow ; \ \alpha_{-2} S_u \quad (I_2)$$

$$S'' \rightarrow (E_L) \ := \ | \ read \ | \ write \quad (I_7)$$

Example: For Loop

$$S \rightarrow ... | for\ i_d := E\ to\ E\ do\ S\ od$$

Ⓐ Ⓑ Ⓒ Ⓓ

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

Example: For Loop

$$S \rightarrow ... | for\ i_d := E\ to\ E\ do\ S\ od$$

Ⓐ Ⓑ Ⓒ Ⓓ

Sample

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

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
b := pop (result of expression)
a := pop (id)
push (b)
push (NQ)
genquad (<= , a, b, c)
genquad (jeq, c, false, _)

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

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

SAS	a	1 10
I	b	I I
1	c	t1
I	NQ	10 11 12 13 32 33
10		
I		
11		

Line Numbers	Operators	Operands O_1	Operands O_2	Results Destinations
10	:=	1		I
11	≤	I	10	t_1
12	jpe	t_1	false	33
13				
14				
15				
16				
17				
18				
19				
20				
31	+	I	1	I
32	jp			11
33	}			

Another example $S \rightarrow ID := E$

A B

Semantic action A
push (A)

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

Another example $while\ C\ do\ S\ od$

A B

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

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

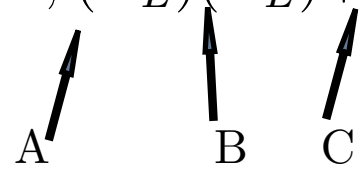
while X <= Y do

Another Example

```
if a < b then
    max := a
else
    max := b
p := 1
for i := 1 to n do
    p := p * i;
```

Line Numbers	Operators	Operands O_1	Operands O_2	Results Destinations	
10	<	a	b	t_1	
11	jpe	t_1	false	14	
12	:=	a		max	
13	jp			15	
14	:=	b		max	
15	:=	1		p	
16	:=	1		i	
17	≤	i	n	t_2	test
18	jpe	t_2	false	23	exit
19	*	p	i	t_3	
20	:=	t_3		p	
21	+	i	1	i	increment
22	jp			17	loop
23	}				

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.

Semantic action A
push (#)SAS

Semantic action B
A := pop(SAS)
while A <> # do
push (A, S1)
A := pop (SAS)
push (#) SAS

Semantic action C
A := pop(SAS)
while A <> # do
push(A,S2)
A := pop (SAS)
push (#) SAS
A := pop(S1)
B := pop (S2)
while A <> empty do
C := gentemp
genquad (+, A, B, C)
push (C) SAS
A := pop (S1)
B := pop (S2)

Also are included local variables, A, and B.

S_1	S_2	A	B	SAS	

Note that this mechanism allows EL to be a list, and for those elements to be added and stored as a list. Also, C should be done after the plus. Watch out for common prefixes, the elimination there after, and account for their semantic actions.

Semantic action A
push (#)SAS

Semantic action B
A := pop(SAS)
while A <> # do
push (A, S1)
A := pop (SAS)
push (#) SAS

Semantic action C
A := pop(SAS)

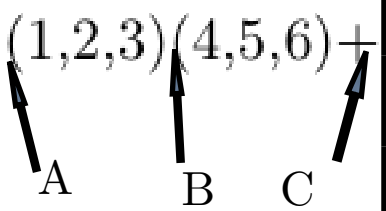
while A <> # do
push(A,S2)
A := pop (SAS)

push (#) SAS
A := pop(S1)
B := pop (S2)

while A <> empty do
C := gentemp
genquad (+, A, B, C)
push (C) SAS
A := pop (S1)
B := pop (S2)

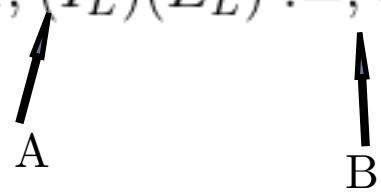
Note that a test of B in addition to A in semantic action C can test for a semantic error.

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



NO	S Actio	S_1	S_2	A	B	C	SAS	Operations	O_1	O_2	Result
	A						#				
	B						#, 1,2				
		3					#, 1,2				
		3,2 1					#				
	C	3,2 1					#, 4,5				
		3,2 1	6,5, 4				#				
10		3,2	6,5	1	4	t1	t1	+	1	4	t1
11		3	6	2	5	t2	t1,t 2	+	2	5	t2
12				3	6	t3	t1,t 2,t3	+	3	6	t3
13				em pty			t1,t 2,t3				

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



Semantic Action A
push (#)

Semantic Action B
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)

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, which is handled in semantic action B.

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.

Example of production

$(I_L)(E_L) :=$
(X,Y,Z)((1,2,3)(4,5,6)+) :=

A B

Semantic Action A
push (#)

Semantic Action B
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)

NQ	S Action	S_1	S_2	A	B	C	SAS	Operations	O_1	O_2	Result
10	A						#				
	P(IL)						#X, Y,Z				
	P(EL)						#X, Y,Z, #,				
	B			t3			#X, Y,Z, #, t1,				
		t3, t2,		t1			#X, Y,				
		"	Z,Y,	X			#				
30		t3,	Z,Y	t1	X			:=	t1		X
		t3	Z	t2	Y			:=	t2		Y
				t3	Z			:=	t3		Z
				empty							

Another production: $\{SL\}\ id\ from\ [E\ ..\ E]$ This production is found in S' which is called by production $S \rightarrow \alpha_{12} S_L \alpha_{13} S'$. 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.

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.

2 Methods of Translating Boolean Expressions

1. Principle means of encoding boolean expressions
   (a) Represent the states of boolean expressions numerically
      i. Operations become a form of boolean mathematics which inherent to the intermediate machine.
      ii. The logical operators $\wedge$, $\vee$ and $\neg$ (and, or, and not) can be represented by branching statements.
      iii. The logical operators can also be represented as actual quad operations.
   (b) Flow control representation: The method identify the boolean state by position in the program.
2. Optimizations in the flow control and numerical evaluation can be made in cases where by the outcome is determined with out evaluating the full condition.

2.1 Flow Control Statements

The three basic procedural control constructs:

1. if - then
2. if - then - else
3. while - do

In each of these cases, there is a conditional production part of these flow control statements which is evaluated true or false, and any branching can be based those two conditions.

2.2 Mixed-Mode Boolean Expresion

A semantic to be considered in procedural programming is

$$(a < b) + (b < a)$$

In this case, it is an arithmetic expression. Not all procedural syntax allows this sort of mixing.

1. Conditionals are special types of expressions.

2. The case of mixed expressions forces conditional expressions include type information. Reference Figure 8.25.

Until Loop

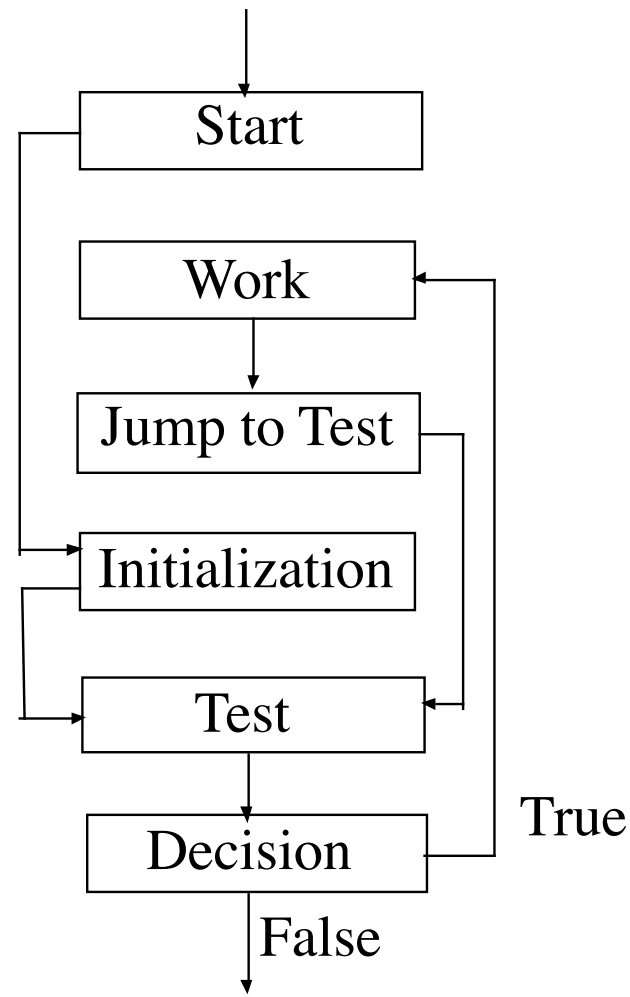
$$S' \rightarrow \underset{G_0}{when}\ C \underset{G_1}{|} \underset{G_3}{until}\ C \underset{G_2}{|} i_d \underset{G_4}{from}\ [E \alpha_{10} E]$$

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 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
```



When Test

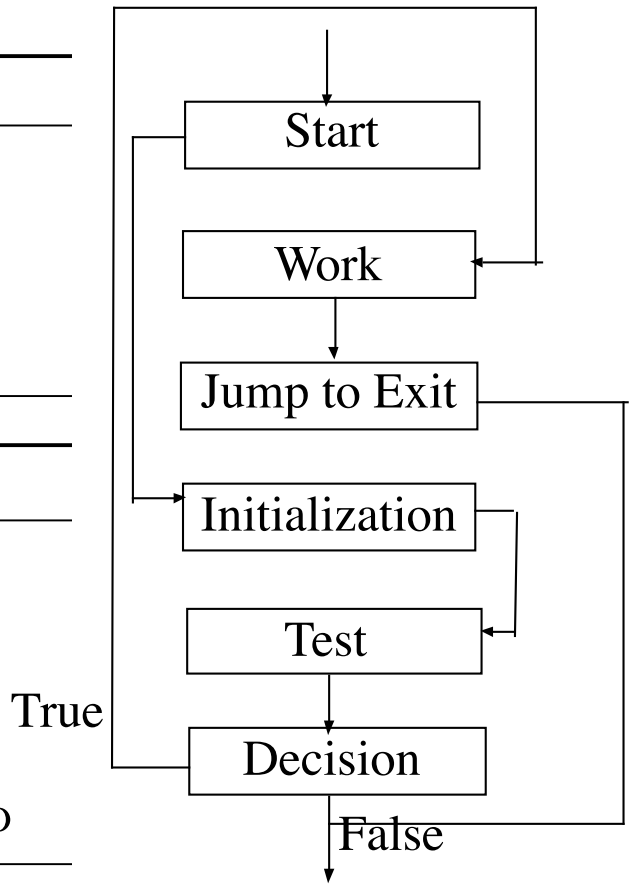
$$S' \rightarrow \underset{G_0}{when}\ C \underset{G_1}{|} \underset{G_3}{until}\ C \underset{G_2}{|} i_d \underset{G_4}{from}\ [E \alpha_{10} E]$$

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
```



$$S' \underset{G_0}{\rightarrow} when\ \underset{G_1}{C} \underset{G_3}{until}\ \underset{G_2}{C} \underset{G_4}{id\ from}\ [\underset{G_5}{E} \alpha_{10} \underset{G_6}{E}]$$

Algorithm 5 Semantic action G_4

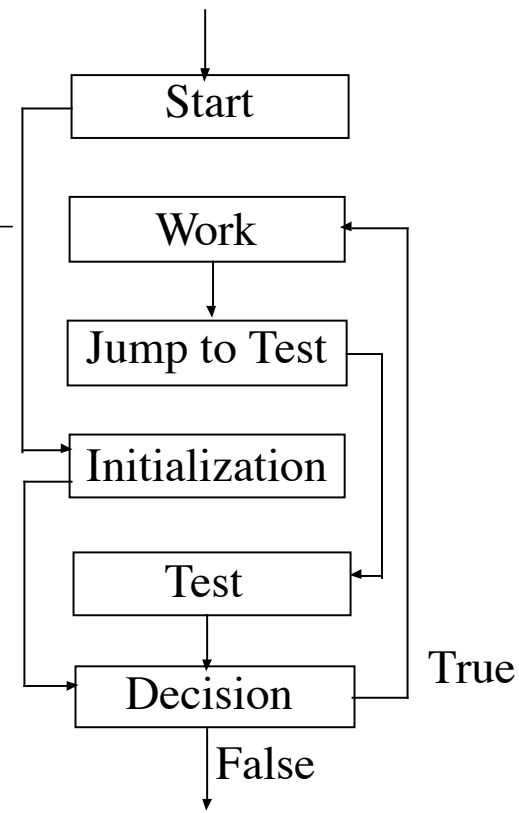
push (id)

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 )
```



$$S \rightarrow \underbrace{\{S_L\}}_{G_7} S' | rdln | wrln | \underbrace{(I_L)}_{I_0} S''$$

Algorithm 8 Semantic action G_7

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

Algorithm 13 Semantic Action for I_0

push (#, SAS)

Algorithm 7 Semantic action W_1

```
push (#)
```

Algorithm 8 Semantic action W_2

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

$$E \rightarrow (E_L)(E_L)E' | T$$
$$W_1 \quad W_2$$

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

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

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

$$C \rightarrow (E_L)(E_L)C''$$
$$W_1 \quad W_2$$

$$C' \rightarrow ; CC''' | not\ C' | \epsilon$$
$$X_{not}$$

$$C'' \rightarrow < C' | > C' | = C' | <= C' | >= C' | <> C'$$
$$X_< \quad X_> \quad X_= \quad X_\leq \quad X_\geq \quad X_\neq$$

$$C''' \rightarrow and C' | or C'$$
$$X_{and}\ X_{or}$$

Algorithm 9 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 10 Semantic action X_{not}

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