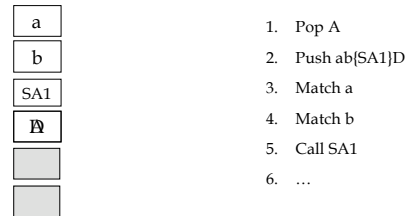


# Semantic Actions

## Top-down parsing and translation schemes

- Semantic actions in right-hand-sides can be pushed on the stack in the same way as terminals and non-terminals
- When on stack top, pop and call the action

**Example: Given A on stack top, A ::= ab {SA1} D to be applied:**

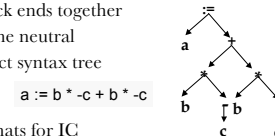


## Semantic action tasks

- **Check context-sensitive constraints** that are not handled in the CFG
  - Variable in declaration statement not previously declared
  - Variable in an executable statement previously declared
  - Type compatibility of operands
  - Formal and actual parameters match in number and type
  - Array has appropriate number of subscripts
- **Generate intermediate code**
  - Code can be generated as each executable statement is parsed (more or less)

## Intermediate code

- Ties the front and back ends together
- Language and machine neutral
- Represents the abstract syntax tree

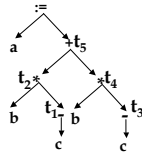


- Several common formats for IC
  - **Postfix notation**
    - Linearized (postorder) representation of the syntax tree  
a b c uminus b c uminus \* + assign
  - **Three address code**
    - Linearized representation of the syntax tree where explicit names ( $t_i$ ) are used for interior nodes
    - Four fields :  
OP ADDR1 ADDR2 ADDR3

## Three address code

`a := b * -c + b * -c`

`t1 := -c`  
`t2 := b * t1`  
`t3 := -c`  
`t4 := b * t3`  
`t5 := t2 + t4`  
`a := t5`



## Three address code

### Assignment statements

Regular	<code>x := y op z</code>	<code>x := op y</code>
Copy	<code>x := y</code>	
Indexed	<code>x := y[i]</code>	<code>x[i] := y</code>
Address	<code>x := &amp;y</code>	
Pointer	<code>x := *y</code>	

### Jumps (unconditional and conditional)

`goto L`  
`if x relop y goto L`

### Procedure statements

`param x`  
`return_address L`  
`return_value x`

## The Vassar Interpreter (TVI) code

<http://www.cs.vassar.edu/~cs331/tvi.opcodes>

Example : Simple arithmetic functions

`add op1, op2, op3`  
`sub op1, op2, op3`  
`div op1, op2, op3`  
`mul op1, op2, op3`  
computes `op1 + op2, op1 - op2, etc`  
result is stored in `op3` which must be an address (or alias)

`fadd, fsub, fdiv, fmul`  
floating point version of the above

`uminus op1, op2`  
negates the value of `op1` and stores the result in `op2`

## Processing declarations

### Declarations are non-executable

- No code generated
- Housekeeping:
  - Record information about names in the symbol table
    - Role
      - » Simple variable
      - » Array
      - » Procedure
      - » Function
    - Type
      - » integer
      - » Real
    - Special
      - » Array dimensions (upper and lower bound)
      - » Procedure and function : number and type of parameters
      - » Function : return value type

## Processing declarations

### Tasks

- Store information about names in the appropriate symbol table (global or local)
- “Allocate” space
  - Map out global storage and activation record for procedures and functions
  - Simple variables require 4 bytes (1 unit of storage in our scheme)
  - Array storage requirements computed from size

## Processing declarations

```

var a, b : integer; c : real;
    d : array [1..10] of integer;
2: <identifier-list> ::= identifier #13 <identifier-list-tail>
3: <identifier-list-tail> ::= , identifier #13 <identifier-list-tail>
4: ::=
5: <declarations> ::= var #1 <declaration-list> #2
6: ::=
7: <declaration-list> ::= <identifier-list> : <type> #3 ;
   <declaration-list-tail>
8: <declaration-list-tail> ::= <identifier-list> : <type> #3 ;
   <declaration-list-tail>
9: ::=
10: <type> ::= <standard-type>
11: ::= <array-type>
12: <standard-type> ::= integer #4
13: ::= real #4
14: <array-type> ::= #6 array [ constant #7 .. constant #7 ]
   of <standard-type>
    
```

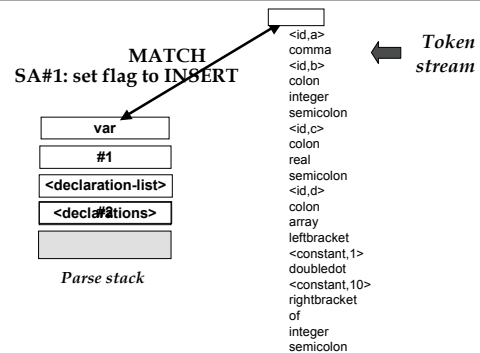
## Processing declarations

SEMANTIC ACTION #1 : INSERT/SEARCH = INSERT

SEMANTIC ACTION #2 : INSERT/SEARCH = SEARCH

*These actions set the flag indicating that id's are to be inserted into the symbol table in a declaration*

## How does it work?



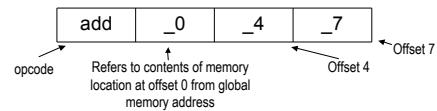
## Generating Code

- Initially, we need to generate code in actions **9** and **56**
  - Signal beginning of the main procedure (main)
- SA 9**

```
GEN(CODE)
GEN(call,main,0)
GEN(exit)
```
- Use **Gen** function
- Code is written to **quadruple array**

## Quadruple array

- Array in which the TVI code is stored as generated
- Each element in the array is a **quadruple**
  - Holds the **opcode** plus (up to) **3 addresses** for the instruction
    - Some instructions use only 1, 2 or 3 fields
  - Opcode and addresses are **strings**



## Signature for Quadruple and Quad array

```
struct Quadruple { string TVIOpCode;
                  string Param1;
                  string Param2;
                  string Param3; };
```

```
vector<Quadruple*> Quads;
```

**Gen** function takes the info to generate a quadruple

```
void Gen (const char* TVIOpCode, STEntry* TVIParam1=NULL,
          STEntry* TVIParam2=NULL, STEntry* TVIParam3=NULL);
```

– Takes symbol table entry pointers, retrieves names from these

**WriteCode** function fills the quad array

```
void WriteCode(string TVIOpCode, string TVIParam1=NULL,
               string TVIParam2=NULL, string TVIParam3=NULL);
```

## SA 9 (second part)

- For the code generation in SA 9, do

```
Gen("CODE");
WriteCode("call", "main", "0"); ← exception
Gen("exit");
```
- WriteCode** increments **NextQuad**
  - NextQuad points to the next open slot in the Quad array
    - Initialize to 1

## Semantic Actions 55 and 56

---

- These two SAs handle the generation of code signalling the beginning and end of the main procedure and allocating and deallocating global memory

```
#55 : BACKPATCH(GLOBAL_STORE, GLOBAL_MEM)
      : GEN(free GLOBAL_MEM)
      : GEN(PROCEND)

#56 : GEN( PROCBEGIN main)
      : GLOBAL_STORE = NEXTQUAD
      : GEN(alloc,_)
```

## Implementation

---

### #55:

```
BACKPATCH(GLOBAL_STORE, GLOBAL_MEM)
  i = intToChars ( GLOBAL_MEM );
  Quad[GLOBAL_STORE][2] = GLOBAL_MEM;
GEN(free GLOBAL_MEM)
  WriteCode( "free", i );
  Gen( "PROCEND" );
```

### #56:

```
GEN( PROCBEGIN main)
  WriteCode( "PROCBEGIN", "main" );
GLOBAL_STORE = NEXTQUAD
  (obvious!)
GEN(alloc,_)
  Gen( "alloc" );
```

## Simple arithmetic expressions

---

- Initially, ignore type checking
- For now, ignore handling relational operators
- Semantic actions **30, 53, 31, 34, 37, 42, 43, 44, 45, 46, 48**

## Relational Expressions

---

- parsed with same grammar rules as arithmetic expressions
- have to keep track of what type of expression we are parsing (relational or arithmetic)
  - note that arithmetic expressions can be embedded in relational expressions
    - $(a + b) < c$
- relational embedded in control flow structures (IF and WHILE)
- code generated involves **jumps** to different locations depending on truth or falsity of the expression

## Generating TVI Code

- Generate branch statements
  - **goto, blt, beq, bne, bgt, blt, bge, ble**
- Usually, we have not yet generated the code that is the target of jumps

Example: `if (a<b) then c:=d;`

When quadruple 24 is generated, 26 and 27 do not yet exist

```
[24] blt a,b,26
[25] goto 27
[26] move d,c
[27] ...
```

when we parse `a<b` need to generate code that will branch to the statement `c:=d` if `a<b` is true, branch to the statement after `c:=d` otherwise

## Solution

- Generate the quadruples for relational expressions with targets left empty
  - same strategy as with `ALLOC`
- Keep a **list** of the quads that need to be filled in with the target quad for the `TRUE` case, another list of quads to be filled in for `FALSE` case
  - **These lists will be called `E.TRUE` and `E.FALSE`**
    - “expression true” and “expression false”

## Simple Relational Expressions

- Handled by productions for `<expression>` and `<expression-tail>`
    - note grammar does not allow e.g. `a<b<c`
- `<expression-tail> ::= RELOP #38 <simple-expression> #39`
- **SA #38 : push RELOP**

## E.TRUE and E.FALSE

- Also in SA #39, make note of the quadruples just generated so their targets can be filled in later
- **SA #39 : generate code**

```
E.TRUE := makelist(NEXTQUAD)
E.FALSE := makelist(NEXTQUAD+1)
GEN(if ID op ID goto __)
GEN(goto __)
```
- Need to hang on to `E.TRUE` and `E.FALSE` empty targets on the stack