1. The Parse Stack looks like

Stack	Input	Apply Rule
S V	a b c b b b d d a c |-	
a S A V		1
S A V	b c b b b d d a c |-	
B b C A V		2
b B b b C A V		5
B b b C A V	c b b b d d a c |-	
c b C b b C A V		6
b C b b C A V	b b b d d a c |-	
C b b C A V	b b d d a c |-	
b b C A V		8
b C A V	b d d a c |-	
C A V	d d a c |-	
d C A V		7
C A V	d a c |-	
d C A V		7
C A V	a c |-	
A V		8
a c V		3
c V	c |-	
V	|-	ACCEPT

Where V stands for bottom of stack marker.

2. In Turing this expression has a lot of possibilities.
(Don't you just HATE it when language designers overload the syntax?)

2a. Static checks

A, I and B should all be checked for declaredness, visibility and accessibility.

The identifier I could be almost anything
e.g the name of a type, constant, variable, procedure, function, module
Type/kind of I need to be checked against type/kind of A in all cases.
If I is used to produce a value it should be checked:
- if it is a variable or constant check that it's scalar.
- if it is a function, check that it has no parameters and that it's return type is scalar.

The operand A(I) could be
array subscript
check A is 1-dimensional array,
check I produces a scalar value of suitable type
function call
check A is function with 1 parameter
check I is suitable actual parameter, e.g. var/const/func/proc and correct type
substring reference, i.e. A is a string variable/constant
check that I is a variable/constant of integer type
if I is a constant check that it is <= max length of A
collection dereference, i.e. A is the name of a collection,
I has to be checked that it is a pointer to the same collection.
set type constructor, i.e. A is the name of a set type and
I has to be checked that it is the correct type for a set member.

The operand B(7) could be
array subscript
check B is a 1-dimensional array,
integer 7 is a suitable type of subscript
check 7 is in range for subscript of B
(if upper bound of B is known at compile time)
function call
check B is function with 1 parameter
check 7 is suitable actual parameter, e.g. non-var, correct type
substring reference, i.e. B is a string variable/constant
check that 7 is <= max length of B.
set type constructor, i.e. B is the name of a set type and
7 has to be checked that it is the correct type and in range

for a set member.

The + operator could be
integer add
real add
string concatenation
set union
in each case the operands A(I) and B(7) have to be checked for type compatibility with each other and the operator.

2b. Dynamic Checks
If any of A, B, I are variables, check for use of uninitialized variables
If A is an array, check I is in range as a subscript.
If B is an array with dynamic upper bound, check that subscript 7 is in range.
If A is a string, check I is a valid substring reference
If B is a string check that 7 is a valid substring reference.
If A is a collection, check that I is a valid pointer, i.e. not null and pointing at data that has not been freed.
If A is a set type, then check that I is in range for the set constructor

3. There are a lot of steps that one can take to make a compiler for generated program compile quickly. The language being compiled will partially determine which of these speedups can be implemented.

a) tune lexical analysis for raw speed
- use hash function for reserved word determination, use perfect hash *if* the hash function isn't too expensive.
- while accumulating identifiers, record whether a digit is seen, if a digit has been found the identifier can't be a reserved word so skip the reserved word test.
- do all of the lexical analyzer optimizations on Slide 51
- try to make each lexical token as small as possible, possibly package tokens in blocks to save space.
- try to have the lexical analyzer recognize large constant tables and store them in special memory or disk tables, i.e. try to keep these large objects out of the lexical token stream.

b) tune for memory efficiency
- try to make all table entries as small as possible, especially the symbol table. Use auxiliary tables and special case code to keep the common case small.
- *if* the machine on which the compiler runs permits it, use main memory lavishly. 200 Mb of tables in real memory is faster than any alternative. Be prepared to back tables up to disk if the program being compiled gets too large.

c) use an O(n) parser tuned for speed, LL(1) and LALR(1) are *both* good bets depending on the language.

d) Optimize symbol lookup
Use a *very large* hash table. Tune hash function for the expected form of identifiers.

e) Tune processing for efficiency
- use a single pass compiler if the language allows it.
- try to avoid building large internal data structures (e.g. statement lists)
- make emitting of generated code as efficient as possible.

f) semantic analysis
- do the minimal checking required to ensure correctness. Question stated that this was required to check for bugs in the program generator.
- don't implement recovery strategies, an error halt is appropriate in this case.
- numeric error messages referencing a printed list of messages is sufficient. Source program coordinate of the error is very important
-design a special processor for checking initialization lists so they don't have to go through the full parse, semantic analysis mechanism.

g) optimization
A real toss up. Users want it but the size of programs will render almost all contemporary optimization techniques infeasible.

4. Successful Exits

T1 - integer consisting of a string of digits
T2 - the subrange separator ..
T3 - real number of the form .digits
T4 - real number of the form digits.
T5 - integer followed by the subrange separator ..

```
   T6 - real number with explicit exponent, any of
           digitsE+digits  digitsE-digits
           digits.E+digits digits.E-digits
           .digitsE+digits         .digitsE-digits
           digits.digitsE+digits   digits.digitsE-digits
   T7 - real number of the form  digits.digits

Error exits

   E1 - single period not followed by digits or period
        (this probably shouldn't be an error, since it
         could be part of   A . B  )
   E2 - sign in explicit exponent of a real number
        is not followed by a digit.
   E3 - E in explicit exponent of a real number is not
        followed by a sign or digit.
```

5.

5a. Follow sets

```
   follow( S )
       Add |- since S is the goal symbol
       Rule 3  add { d }
       Rule 5  add first( A ) and  { c }
       Rule 8  add { f }

       first( A ) =  { a, d }
       follow( S ) = { a, c, d, f, |- }

   follow( A )
       Rule 1  add first( B ) and { b }
       Rule 2  add first( S ) and { d }
       Rule 5  add { c }

       first( B ) = { a, c, d, e }
       first( S ) = { a }
       follow( A ) = { a, b, c, d, e }

   follow( B )
        Rule 1  add { b }
        Rule 11 add first( D ) and follow( D )

        first( D ) = { a }
        follow( B ) = { a, b } U follow( D )
                    = { a, b, c, d, f, |- }

   follow( C )
        Rule 1  add first( D ) and follow( S )
        Rule 6  add follow( B )
        Rule 9  add { g }

        follow( C ) = { a, g } U follow( S ) U follow( B )
                    = { a, b, c, d, f, g, |- }

   follow( D )
           Rule 1 add follow( S )
           Rule 11 add follow( D )

           follow( D ) = follow( S ) = { a, c, d, f, |- }
```

5b. Director Sets

1	{ a }	
2	{ a, c, d, f, |- }	follow(S)
3	{ a, d }	first(A)
4	{ a, b, c, d, e }	follow(A)
5	{ a, c, d }	first(S) U first(A) U { c }
6	{ e }	
7	{ a, b, c, d, f, |- }	follow(B)
8	{ a, f }	first(S) U { f }
9	{ a, f, g }	first(C) U { g }
10	{ a, b, c, d, f, g, |- }	follow(C)

```
        11      { a }
        12      { a, c, d, f, |- }

5c.  LL(1)ness

        NO WAY
                Left recursive in A and C
                Many director set conflicts.
```