BNF for Core

<prog> ::= *program* <decl seq> (1)
begin <stmt seq> *end*

<decl seq> ::= <decl> | <decl> <decl seq> (2)

<stmt seq> ::= <stmt> | <stmt> <stmt seq> (3)

<decl> ::= *int* <id list>; (4)

<id list> ::= <id> | <id>, <id list> (5)

<stmt> ::= <assign>|<if>|<loop>|<in>|<out>(6)

<assign> ::=<id> = <exp>; (7)

<if> ::= *if* <cond> *then* <stmt seq> *end;* (8)
|*if* <cond> *then* <stmt seq> *else* <stmt seq> *end;*

<loop> ::= *while* <cond> *loop* <stmt seq> *end;*(9)

<in> ::= *read* <id list>; (10)

<out> ::= *write* <id list>; (11)

BNF for Core (contd.)

<cond> ::= <comp>|!<cond> (12)
| [<cond> && <cond>] | [<cond> *or* <cond>]

<comp> ::= (<op> <comp op> <op>) (13)

<exp> ::= <trm>|<trm>+<exp>|<trm>−<exp> (14)

<trm> ::= <op> | <op> * <trm> (15)

<op> ::= <no> | <id> | (<exp>) (16)

<comp op> ::= != | == | < | > | <= | >= (17)

<id> ::= <let> | <let><id> | <let><no> (18)

<let>::=A | B | C | ... | X | Y | Z (19)

<no>::=<digit> | <digit><no> (20)

<digit>::=0 | 1 | 2 | 3 | ... | 9 (21)

- Notes:

  Productions (18)-(21) have no *semantic* significance;
  (19) and (21) are superseded by (19′) and (21′) on next page:

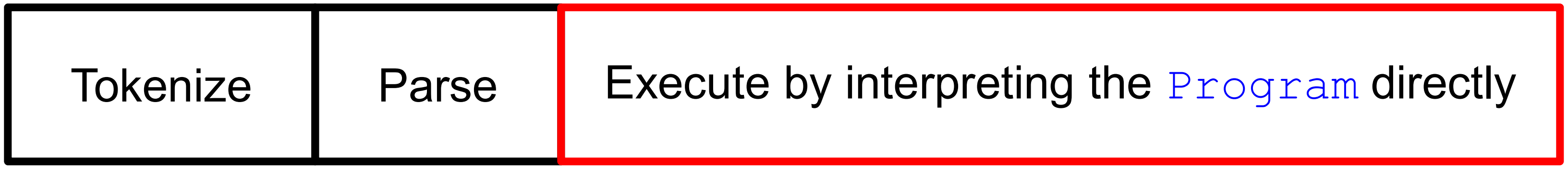
BNF for Core (contd.)

<let> ::= A | B | C | D | E | F | G | H | I | J | K | L | M
| N | O | P | Q | R | S | T | U | V | W | X | Y | Z (19′)

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 (21′)

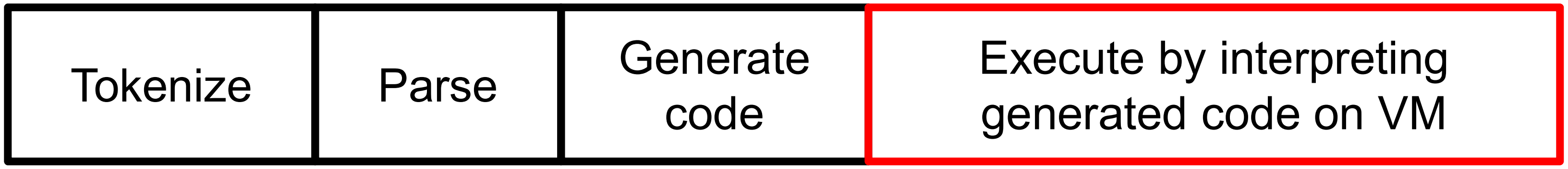
Timelines of Execution

- Directly interpreting a `Program`:



Tokenize | Parse | Execute by interpreting the `Program` directly

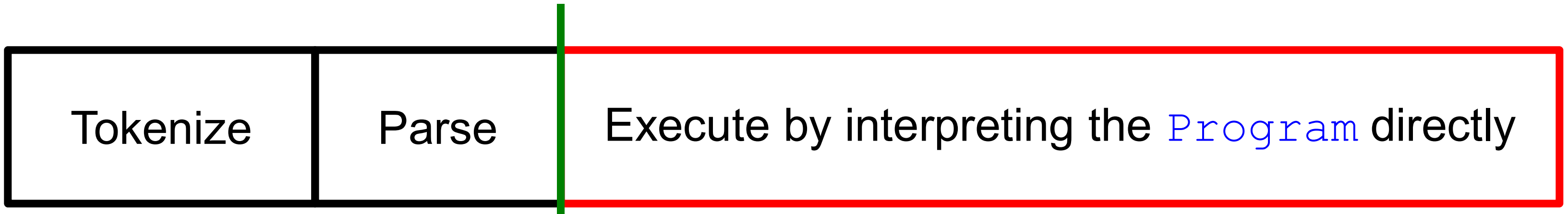
- Compiling and then executing a `Program`:



Tokenize | Parse | Generate code | Execute by interpreting generated code on VM

Timelines of Execution

- Directly interpreting a `Program`:

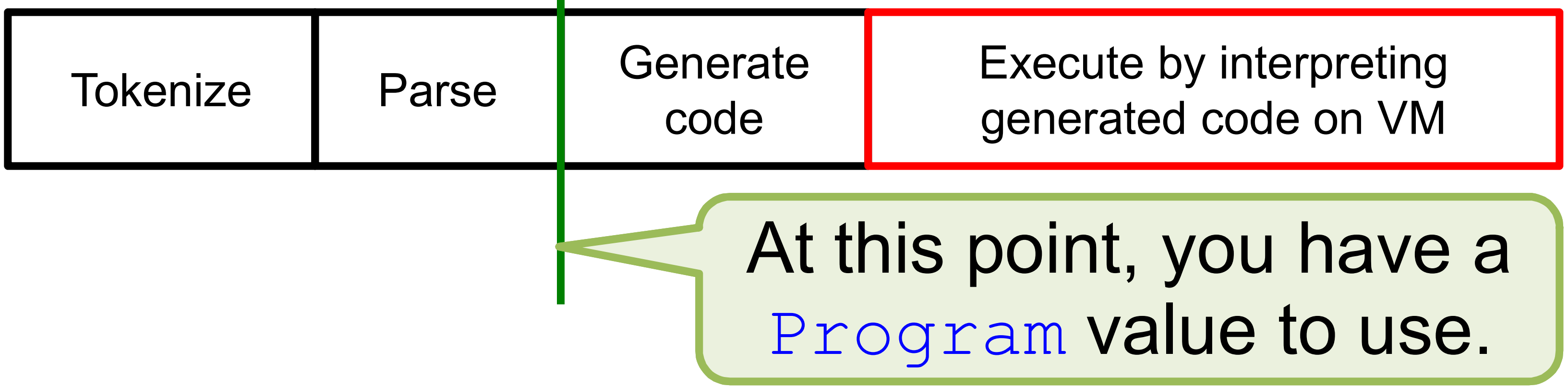


Tokenize

Parse

Execute by interpreting the `Program` directly

- Compiling and then executing a `Program`:



Tokenize

Parse

Generate code

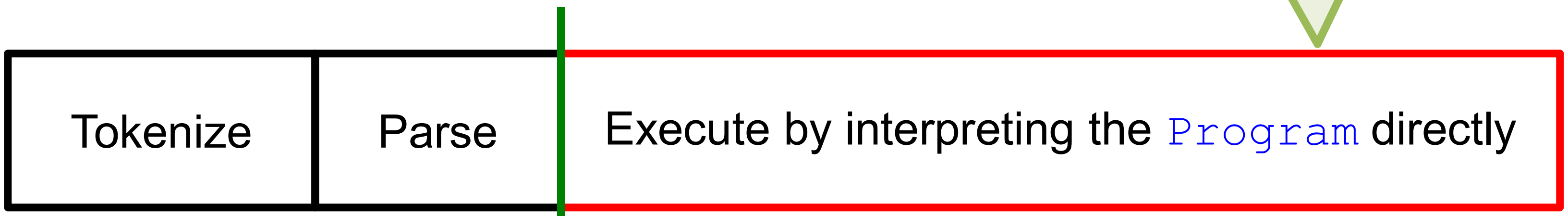
Execute by interpreting generated code on VM

At this point, you have a `Program` value to use.

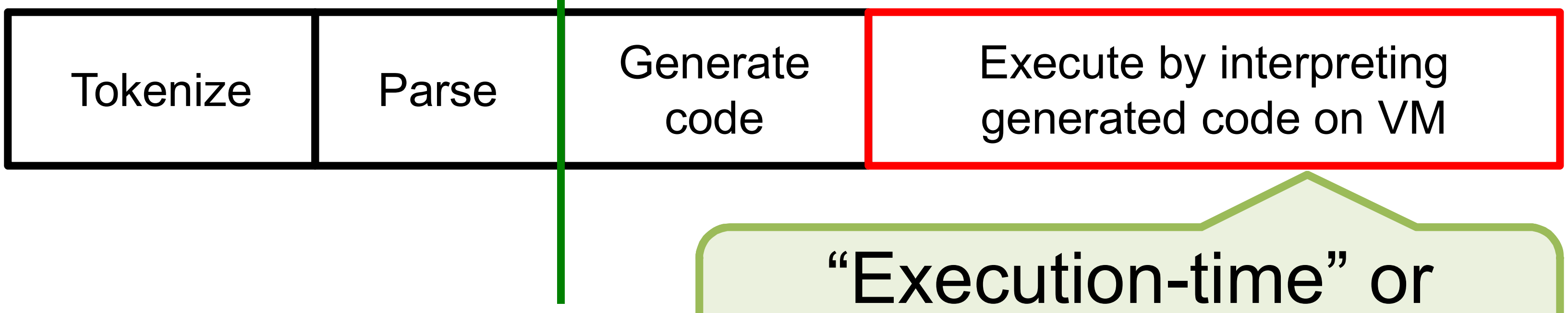
Timelines

“Execution-time” or “run-time” means here.

- Directly interpreting a `Program`:



- Compiling and then executing a `Program`:



“Execution-time” or “run-time” means here.

A Core program

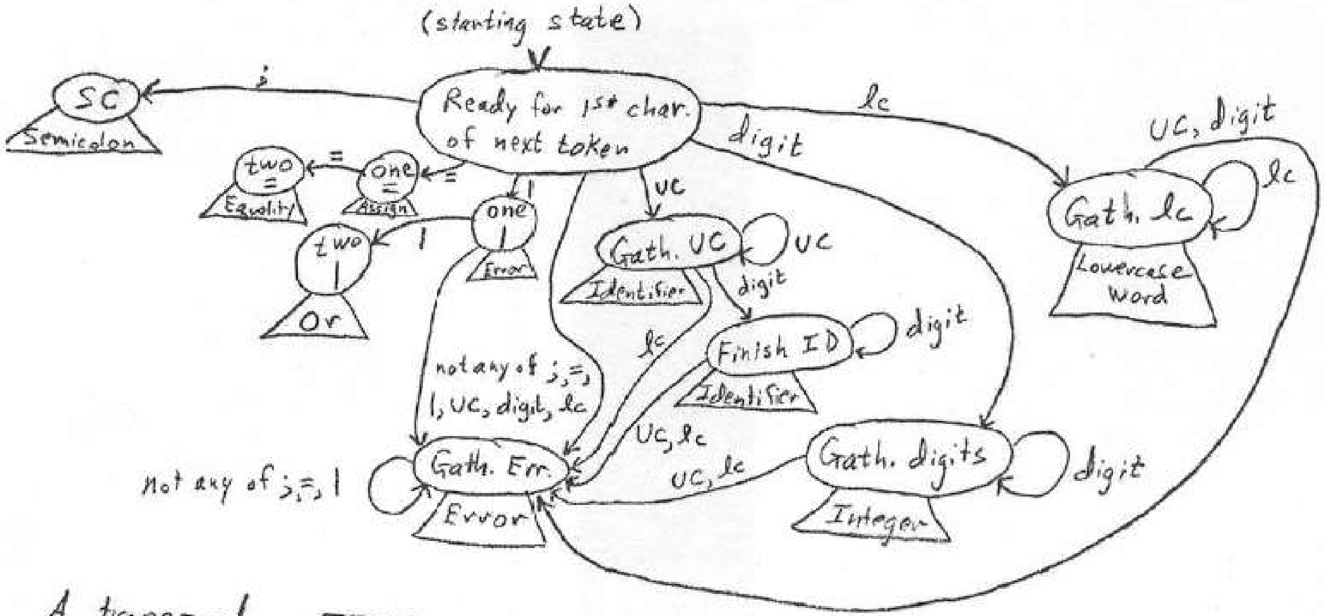
```
program
    int I, J;
begin
    read I;
    if (I == 3) then
        J = 4;
    end;
    J = J + 1;
    write J;
end
```

A Core program

```
program
   int I,J;begin
   read I;if(I==3)then
      J=4;end;J=J+1;write J;end
```

Lab 1 Finite State Automaton (FSA)

In your project, it is not necessary to gather a longest Error token in exactly the way done here; it is only important to recognize a tokenizing error at approximately that point.



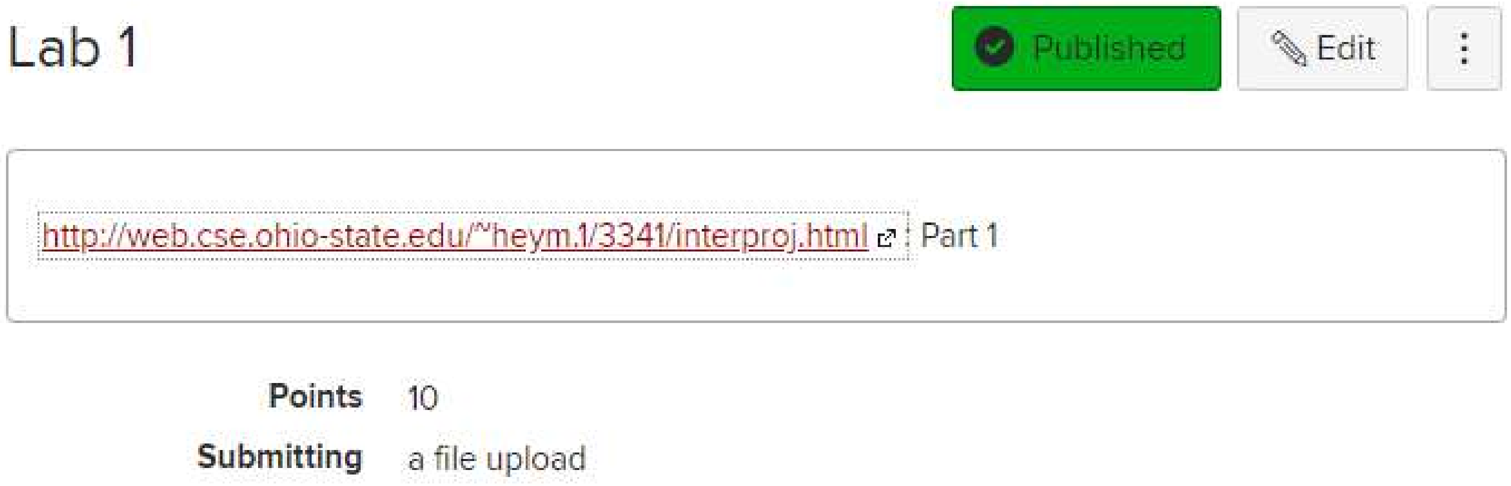
A trapezoid Label catches any next character that is not handled by a labeled transition out of the state. The result of the catch is as follows. The characters seen from the starting state, until this next character, are collected as a token (not including this next character), and the token's kind is what the Label in the trapezoid says. Processing will begin again with this next character, starting from the starting state.

Scanner Code from M. Scott’s Text

From Section 2.2.2 Scanner Code

```
state := 1                 --start state
loop
       read cur_char
       case state of
              1 : case cur_char of
                     ' ', '\t', '\n' :    ...
                     'a'...'z' :          ...
                     '0'...'9' :          ...
                     '>' :                ...
                     ...
              2 : case cur_char of
                     ...
                  ...
              n : case cur_char of
                     ...
```

Lab 1 and Its Skeleton Solution



Lab 1

http://web.cse.ohio-state.edu/~heym.1/3341/interproj.html Part 1

Points	10
Submitting	a file upload

http://web.cse.ohio-state.edu/~heym.1/3341/interproj.html