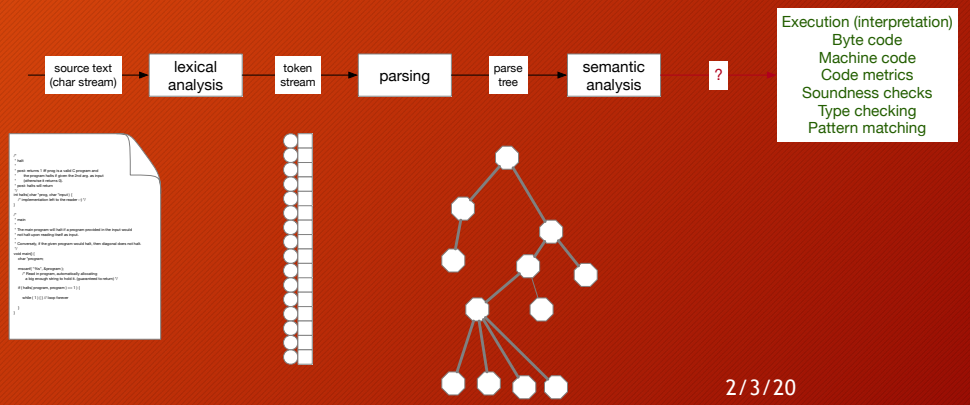
D.
PLCC Beginning Languages

Spring 2020

Some Terms

- *defined language*
  - the language to be processed
- *defining language*
  - the language to use for the processor

- "*processor*"?
  - compiler?
  - interpreter?
- Remember our diagram.



Interpreters vs. Compilers

- Any system that does not translate code in the defined language down to machine code does interpretation.
- Any system that transforms code in the defined language into another form before processing it does compilation.
- Which languages are purely one or the other?

V0

We start our V-series with V0, a language that recognizes simple expressions written in a prefix, function-calling style

- `5`
- `blah`
- `+(9,5)`
- `-(13,blah,89)`
- `-(+(1,2),+(3,4))`
- `add1(blah)`
- `sub1(15)`
- `+(+(+(+(0))))`

What it ain't

- V0 does not
  - know how to store variable values => no Environment
  - evaluate expressions
- Its legal stuff is
  - integer constants
  - identifiers
  - +(arg_1, arg_2, …, arg_n) and –(arg_1, arg_2, …, arg_n)
    - Any number of operands
  - add1(arg) and sub1(arg)
    - Just one operand, but checked at run-time

- It just displays the entered expression!

The Tokens


```
skip WHITESPACE '\s+'
skip COMMENT '%.*'
LIT '\d+'
LPAREN '\('
RPAREN '\)'
COMMA ','
ADDOP '\+'
SUBOP '\-'
ADD1OP 'add1'
SUB1OP 'sub1'
VAR '[A-Za-z]\w*'
```

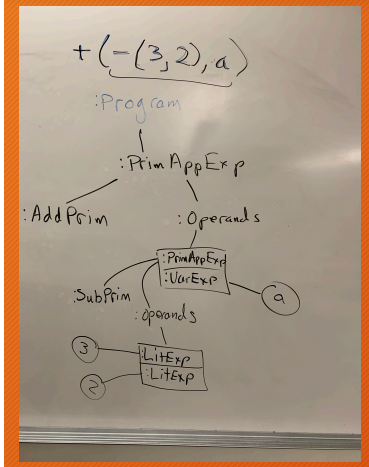
The Grammar


```
<program>          ::= <exp>
<exp>:LitExp       ::= <LIT>
<exp>:VarExp       ::= <VAR>
<exp>:PrimAppExp   ::= <prim> LPAREN <operands> RPAREN
<operands>         **= <exp> +COMMA
<prim>:AddPrim     ::= ADDOP
<prim>:SubPrim     ::= SUBOP
<prim>:Add1Prim    ::= ADD1OP
<prim>:Sub1Prim    ::= SUB1OP
```

The Semantic Stuff

- See v0.plcc.
- Note that, when the goal is to basically output what was input, a lot of the code seems rather silly.
- Demo

V0 Example from class: +(-(3,2),a)



:Program

:PrimAppExp

:AddPrim

:Operands

:SubPrim

:PrimAppExp

:VarExp

:Operands

:LitExp

:LitExp

:Token

2

:Token

:Token

3

a

```
<program> ::= <exp>
<exp>:PrimAppExp ::=
        <prim> '(' <operands> ')'
<operands> **= <exp> +','
<prim>:AddPrim ::= '+'
<prim>:SubPrim ::= '-'
<exp>:LitExp ::= <LIT>
<exp>:VarExp ::= <VAR>
```

V1: V0 + Evaluation

- We can now *apply* the primitive functions to literals.
- If we pre-load some bindings into an environment node, we can put variables in the function calls, too.
- Recall grammar
- To do
  - Create an `EnvNode` in `Program`
  - Add abstract `eval()` method to `Exp`
  - Add abstract `apply(Val[])` method to `Prim`
    - because `PrimAppExp` has the operands, not `Prim`

```
<program>          ::= <exp>
<exp>:LitExp       ::= <LIT>
<exp>:VarExp       ::= <VAR>
<exp>:PrimAppExp   ::= <prim> '(' <operands> ')'
<operands>         **= <exp> +','
<prim>:AddPrim     ::= ADDOP
<prim>:SubPrim     ::= SUBOP
<prim>:Add1Prim    ::= ADD1OP
<prim>:Sub1Prim    ::= SUB1OP
```

Solution and Example

- V1's EXPRESSION Handout

Solution and Example

- V2's V2_IF-EXPR Handout