CS558
Programming Languages

Fall 2023
Lecture 1b

Andrew Tolmach
Portland State University

Describing Languages

Syntax

Easy!

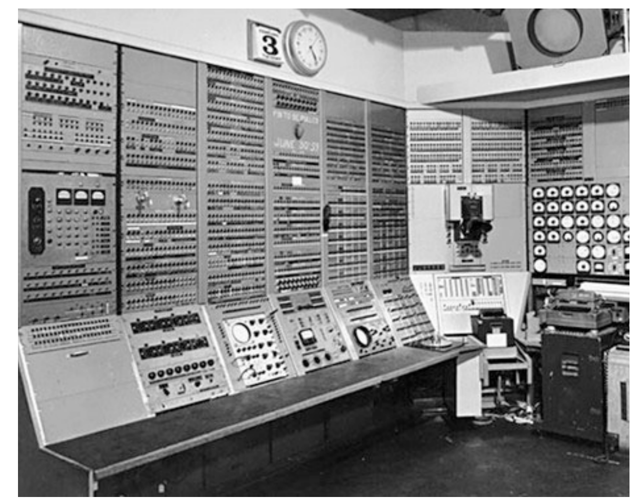
www.zipcodewilmington.com

Program	::=	`BEGIN` *Statement-seq* `END`
Statement-seq	::=	*Statement*
		[';' *Statement-seq*]
Statement	::=	[*While-statement* | *For-statement*]
While-statement	::=	`WHILE` *Expression*
		`DO` *Statement-seq* `END`
Expression	::=	*Factor* { (`AND` | `OR`) *Factor* }
Factor	::=	'(' *Expression* ')' | *Variable*

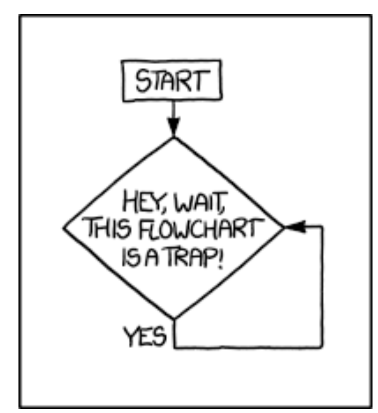
Form: what does program look like?

Semantics

Harder!



www.computerhope.com



imgs.xkcd.com/comics/flowchart.png

Meaning: what does program do?

Describing Syntax

Concrete

- Describes legal form and structure of programs
- Describes what programs look like on page or screen
- Usually defined by a context-free grammar (CFG)

Abstract

- Describes essential contents of programs as they might look internally (in a compiler or interpreter)
- Can be defined by a tree grammar

PL syntax before symbolic grammars

An amessage is	
either a *demand*, and has	[**Print** $a+2b$
a *body* which is an aexpression,	
or **else** a definition,	[**Def** $x=a+2b$
where rec	
an aexpression (aexp) is	
either *simple*, and has	[$CAth231''$
a *body* which is an identifier	
or a *combination*, in which case it has	[$sin(a+2b)$
a *rator*, which is an aexp,	or
and a *rand*, which is an aexp,	$a + 2b$
or *conditional*, in which case it is	
either *two-armed*, and has	[$p \rightarrow a+2b;\quad 2a-b$
a *condition*, which is an aexp,	
and a *leftarm*, which is an aexp,	
and a *rightarm*, which is an aexp,	
or *one-armed*, and has	[$q \rightarrow 2a-b$
a *condition*, which is an aexp,	
and an *arm*, which is an aexp,	
or a *listing*, and has	[$a+b,\ c+d,\ e+f$
a *body* which is an aexp-list,	
or *beet*, and has	[$x(x+1)$ **where** x = a + $2b$
a *mainclause*, which is an aexp,	or
and a *support*	**let** $x = a + 2b;\quad x(x+1)$
which is an adef,	

P.J. Landin, The Next 700 Programming Languages, CACM 9(3), Mar. 1966

PL syntax with CFGs

```
Program        ::=  BEGIN Statement-seq END
Statement-seq  ::=  Statement
                    [ ';' Statement-seq ]
Statement      ::=  [ While-statement | For-statement ]
While-statement ::= WHILE Expression
                    DO Statement-seq END
Expression     ::=  Factor { (AND | OR) Factor }
Factor         ::=  '(' Expression ')' | Variable
```

- Clear, compact, precise
- Single definition supports recognition, generation, analysis, …
- Captures recursive structure, which is very common in PLs
- Rich theory with connections to automatic parser generation, push-down automata, etc.

Context-free Grammars (CFGs)

- Formally defined by
  - a set of terminal symbols e.g. { (,) }
  - a set of nonterminal variables, which represent sets of strings of terminals

    e.g. {S,T}

    one of which is the start symbol (e.g. S)
  - a set of production rules that map nonterminals to strings of terminals and nonterminals

    e.g. $\{ S \rightarrow (S),\ S \rightarrow S\,S,\ S \rightarrow \varepsilon \}$

empty string

Derivations

The language L(G) defined by a grammar G is the set of sentences (strings of terminals) that can be derived by applying production rules, beginning from the start symbol.

A derivation can be represented as a parse tree.

The grammar specifies the shapes of possible parse trees.

Existence of a parse tree for a sentence shows that sentence is in L(G).

A parse tree imposes hierarchical structure on the parsed sentence.

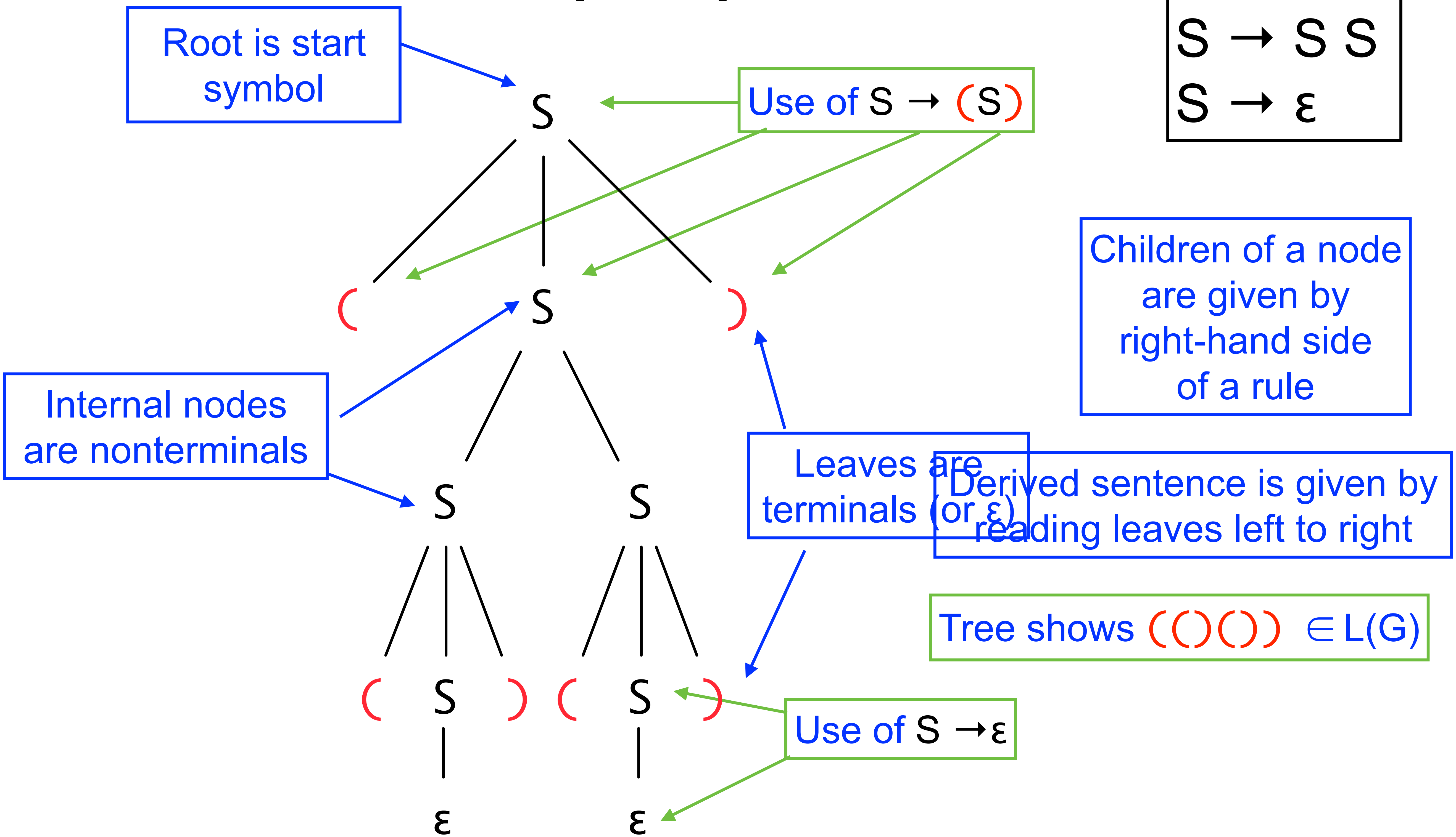
Example parse tree

G

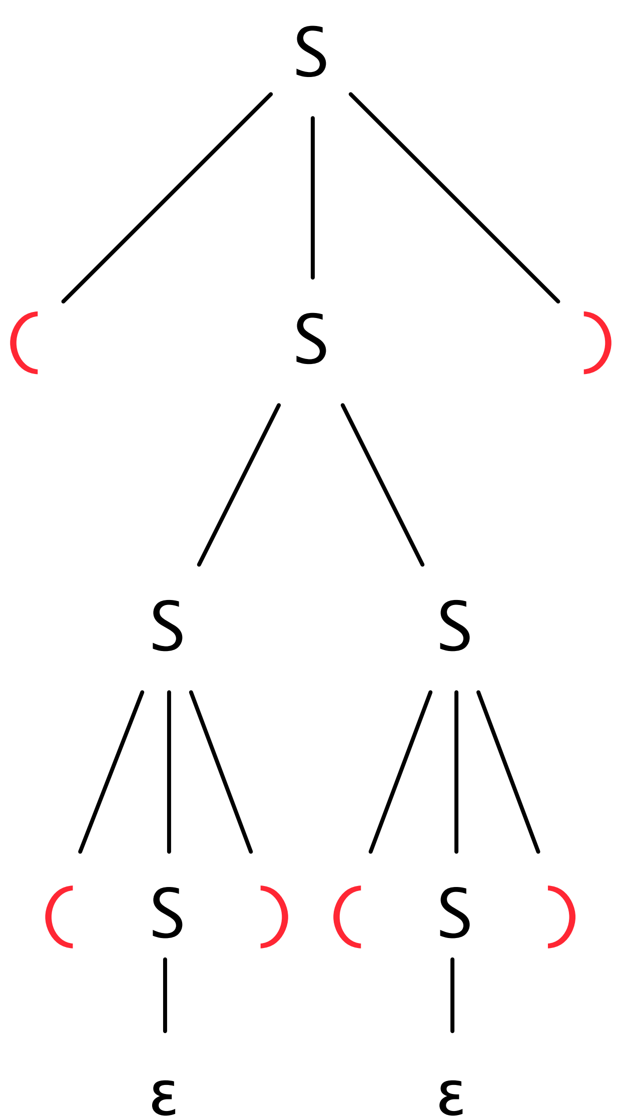
$S \rightarrow (S)$

$S \rightarrow S\ S$

$S \rightarrow \varepsilon$



Linear Derivation Sequences



We can capture the content of a parse tree as a linear sequence of individual derivation steps, where each step expands a single nonterminal by applying some grammar rule.

When there is more than one nonterminal, we can choose any one of them to expand next. This means there can be multiple different linear sequences for the same tree.

E.g. here are two of the possible sequences for the given parse tree:

$$\underline{S} \Rightarrow (\underline{S}) \Rightarrow (\underline{S}S) \Rightarrow ((\underline{S})S) \Rightarrow (()\underline{S}) \Rightarrow (()(\underline{S})) \Rightarrow (()())$$

$$\underline{S} \Rightarrow (\underline{S}) \Rightarrow (S\underline{S}) \Rightarrow (S(\underline{S})) \Rightarrow (\underline{S}()) \Rightarrow ((\underline{S})()) \Rightarrow (()())$$

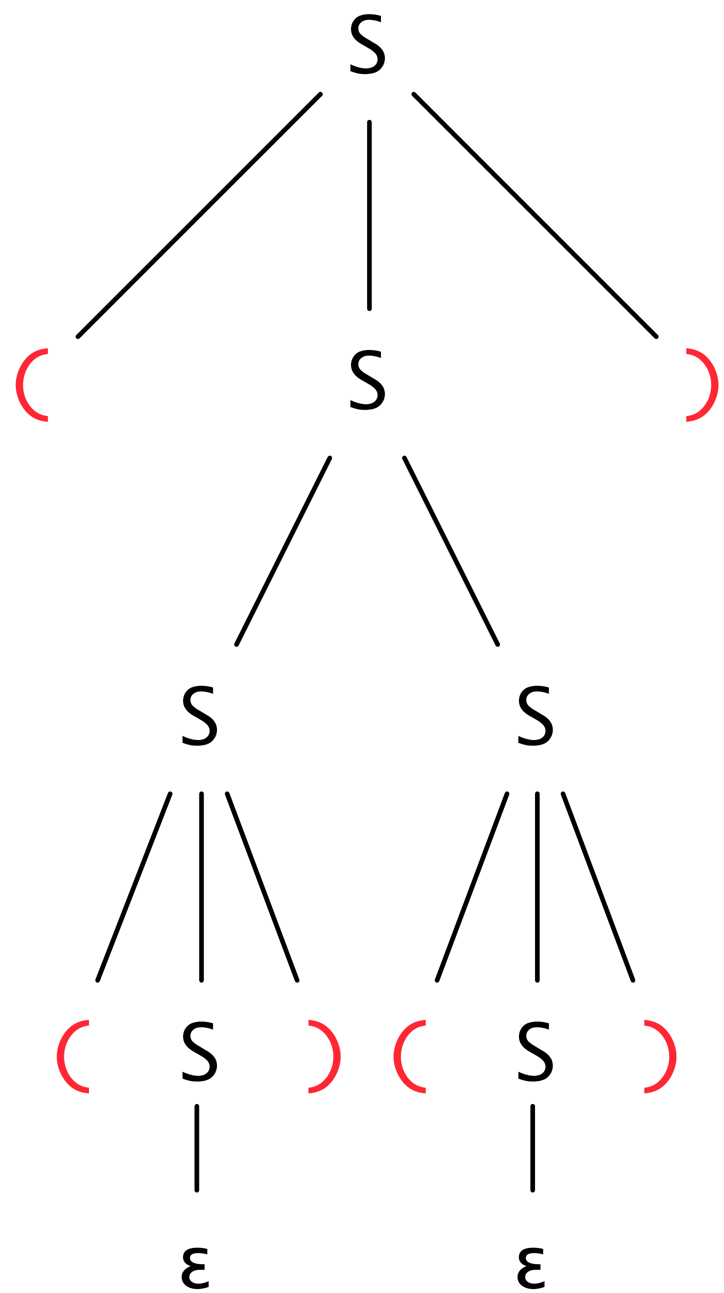
All sequences for a given tree illustrate the same parse structure.

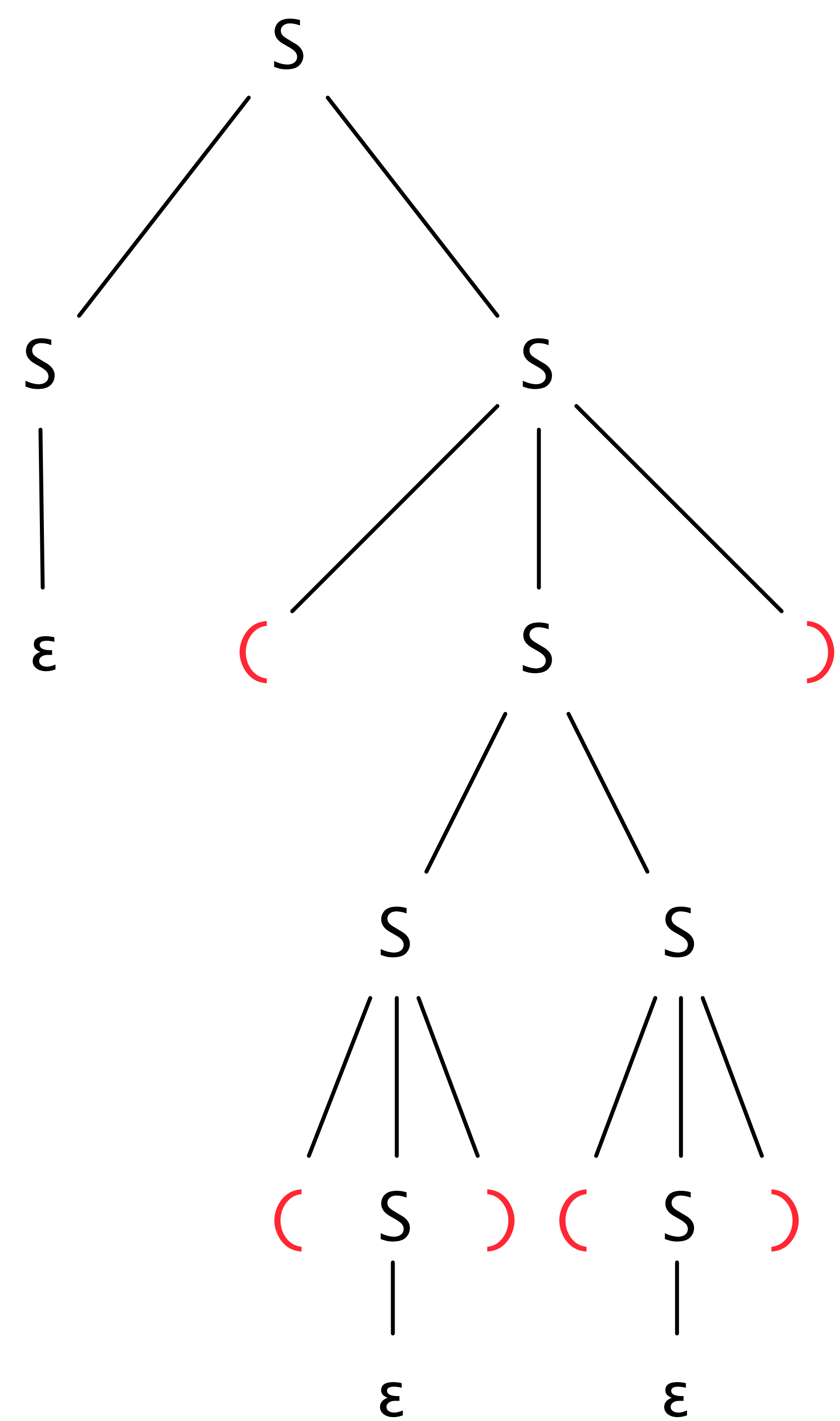
Ambiguous Grammars

G

$$
\begin{aligned}
S &\rightarrow (S) \\
S &\rightarrow S\ S \\
S &\rightarrow \varepsilon
\end{aligned}
$$

- Some grammars let us build multiple different parse trees for the same sentence.
- Such grammars are said to be ambiguous.





Both trees show (()()) $\in$ L(G), but they impose different structure.

Alternative Grammars

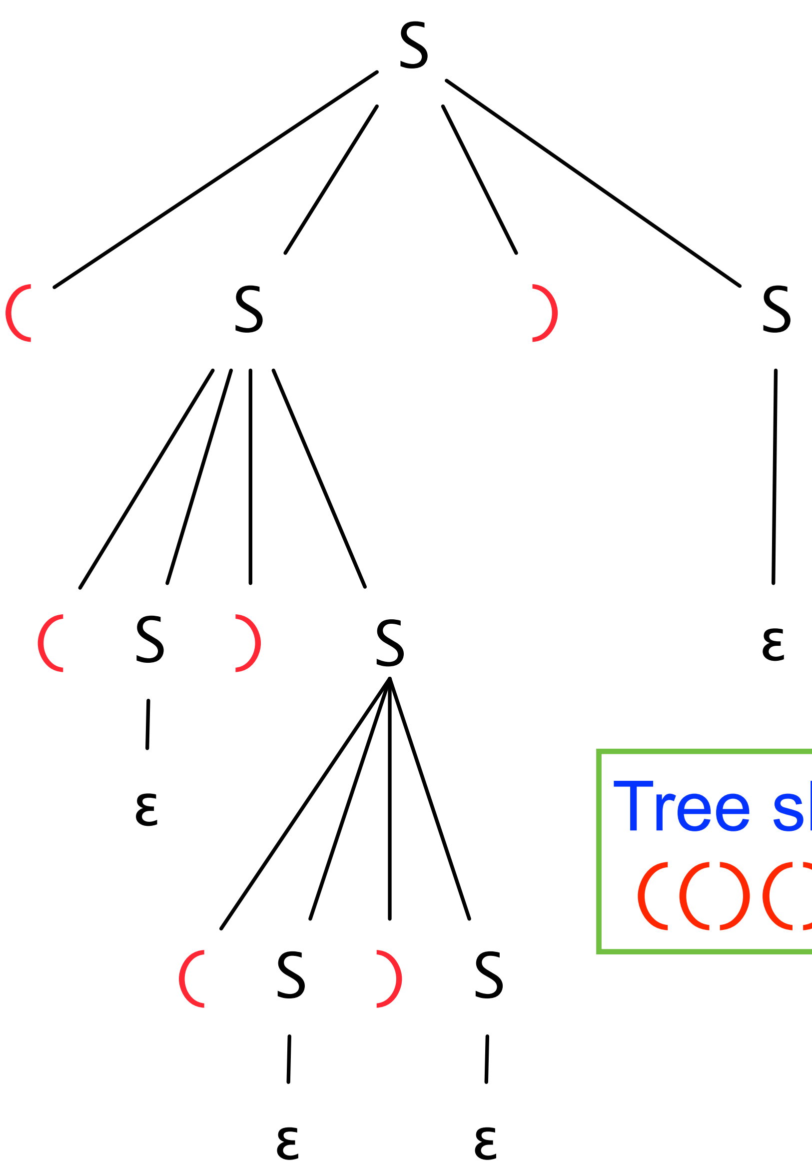
G

$$\boxed{\begin{array}{l} S \rightarrow (S) \\ S \rightarrow S\ S \\ S \rightarrow \varepsilon \end{array}}$$

How might we describe the entire language L(G) informally?

Another grammar for the same language is G'

$$\boxed{\begin{array}{l} S \rightarrow (S)S \\ S \rightarrow \varepsilon \end{array}}$$



Tree showing that
(()()) $\in$ L(G')

It is possible to prove that L(G) = L(G')

But the two grammars impose very different structures on sentences.

Recap of Key CFG Points

- In an ambiguous grammar, the same sentence can have multiple parse trees
  - and hence multiple alternative structures
  - This is generally a bad thing for programming language grammars
- The same parse tree can correspond to multiple linear derivations
  - Depends on the nonterminal we choose to expand first
  - Not important: all these derivations generate the same structure
- The same language can have multiple grammars
  - Each grammar may impose a different structure on a given sentence
  - For programming language grammars, choice of structure is important

Defining CFGs for real PLs

- Use a richer set of terminal symbols called tokens
  - Strings of ASCII or Unicode characters
  - May be generic, i.e. representing sets of strings
    - Generic token carries an attribute value
    - e.g., ID tokens carry the actual identifier string, NUM tokens carry the value of a numeric literal
- CFG is usually notated using some variant of BNF

BNF (Backus-Naur Form)

Invented ca. 1960 for Algol-60 language

nonterminals

terminals

empty string

instead of $\to$

<program>	::=	BEGIN *<statement-seq>* END
<statement-seq>	::=	*<statement>*
<statement-seq>	::=	*<statement>* ; *<statement-seq>*
<statement>	::=	*<while-statement>*
<statement>	::=	*<for-statement>*
<statement>	::=	*<empty>*
<while-statement>	::=	WHILE *<expression>* DO *<statement-seq>* END
<expression>	::=	*<factor>*
<expression>	::=	*<factor>* AND *<factor>*
<expression>	::=	*<factor>* OR *<factor>*
<factor>	::=	(*<expression>*)
<factor>	::=	*<variable>*
<for-statement>	::=	. . .
<variable>	::=	. . .

EBNF (Extended BNF)

More compact

nonterminals

terminal

Program	::=	BEGIN *Statement-seq* END
Statement-seq	::=	*Statement*
		[‘;’ *Statement-seq*]
Statement	::=	[*While-statement* | *For-statement*]
While-statement	::=	WHILE *Expression*
		DO *Statement-seq* END
Expression	::=	*Factor* { (AND | OR) *Factor* }
Factor	::=	‘(’ *Expression* ‘)’ | *Variable*
For-statement	::=	. . .
Variable	::=	. . .

optional

grouping

0 or more

terminal

alternative

Many different variants of EBNF exist

Syntax Analysis (Parsing)

- A parser recognizes syntactically legal programs (as defined by a grammar) and rejects illegal ones.
  - A successful parse captures the hierarchical structure of the program (expressions, blocks, etc.)
  - Tree produced by parsing is basis for further processing (e.g. type checking, interpretation, code generation,…)
  - Failed parse provides error feedback to the user showing where and why the program was illegal
- For most modern PLs, an efficient parser can be generated automatically from CFG
  - Only true for a restricted class of grammars

Expression Grammars

- Expressions are at the heart of most high-level languages, and illustrate important CFG issues

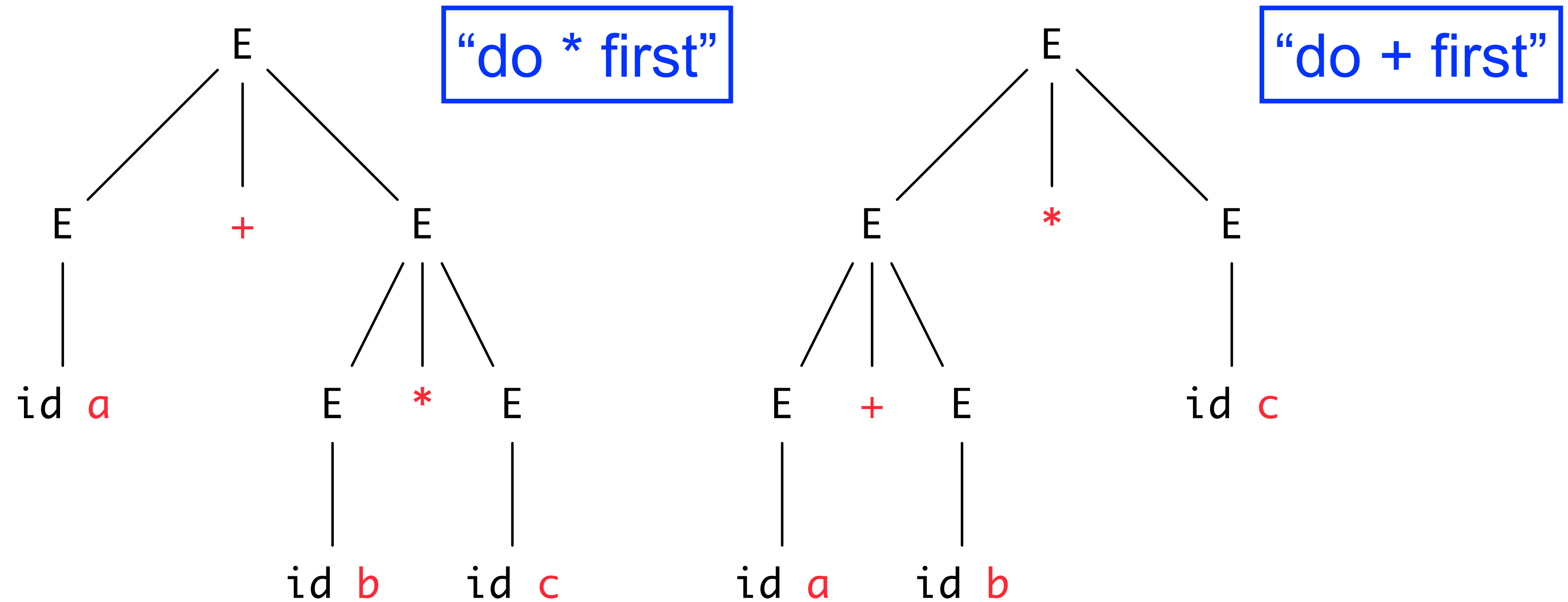
- A naive grammar for arithmetic expressions

$$E ::= E + E \mid E - E \mid E * E \mid E \,/\, E \mid (E) \mid id_x$$

generic token representing identifiers

Ambiguous! Here are two parse trees for a+b*c:

we might think left tree is "correct" one, but nothing in grammar says so



Arithmetic Ambiguity

$$E ::= E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid id_x$$

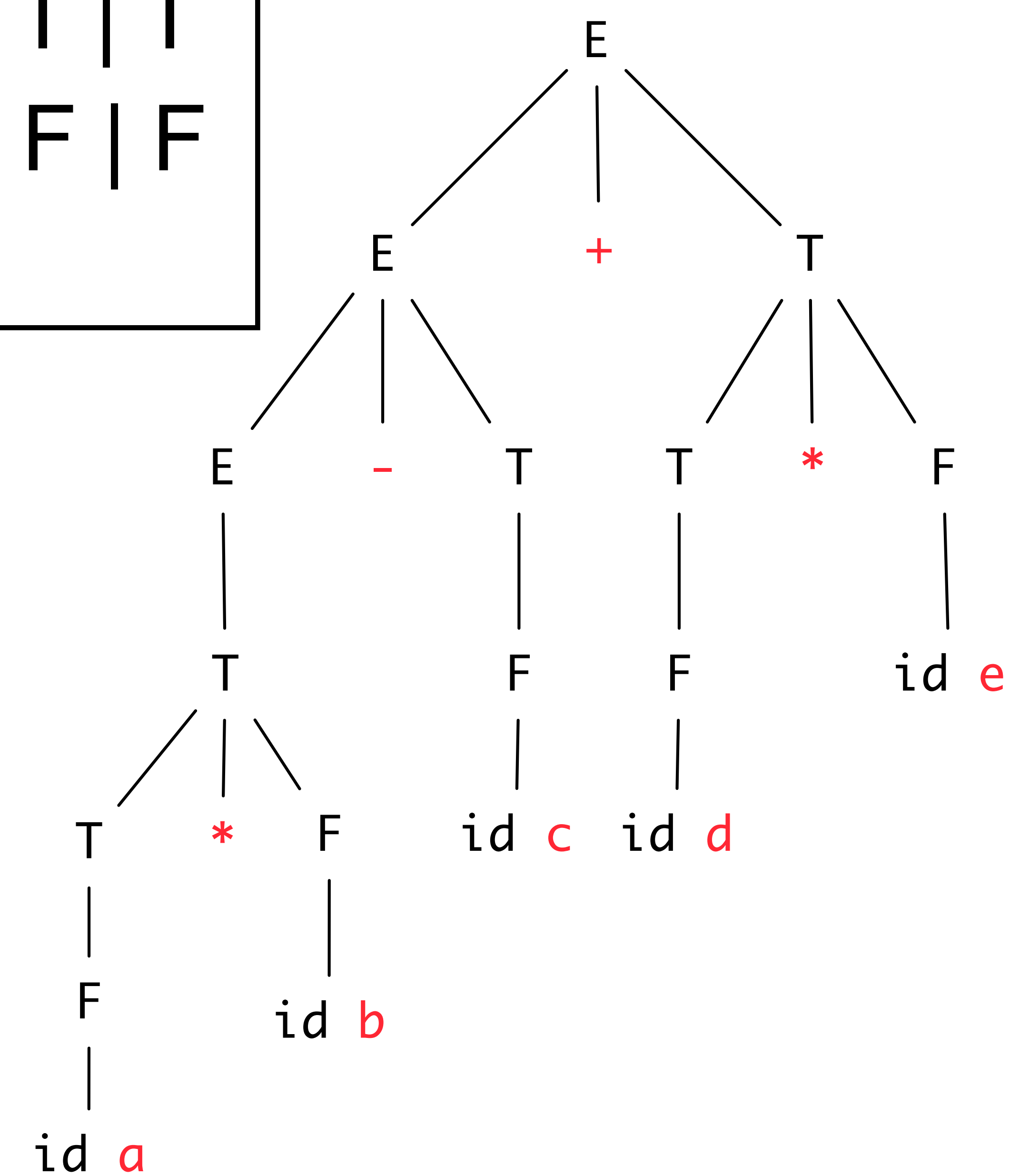
- To disambiguate grammars like this, we must choose desired order of operations for any expression of the form $E_a\ op_1\ E_b\ op_2\ E_c$
- Precedence: which operation (op_1 or op_2) is done first?
- Associativity: if op_1 and op_2 have the same precedence, does E_b "associate" with the operator to its left or to its right?
- I.e., is the expression equivalent to $(E_a\ op_1\ E_b)\ op_2\ E_c$ or to $E_a\ op_1\ (E_b\ op_2\ E_c)$?
- The "usual" rules (based on common usage in written math) give * and / higher precedence than + and - and make all these operators left-associative. But this is a matter of choice in language design.

Rewriting Arithmetic Grammars

- One way to enforce desired precedence/associativity is to build them into the grammar using extra nonterminals, e.g.

$$
\begin{aligned}
E &::= E + T \mid E - T \mid T \\
T &::= T * F \mid T / F \mid F \\
F &::= (E) \mid id_x
\end{aligned}
$$

Example parse for `a*b-c+d*e`:



Why does this work?

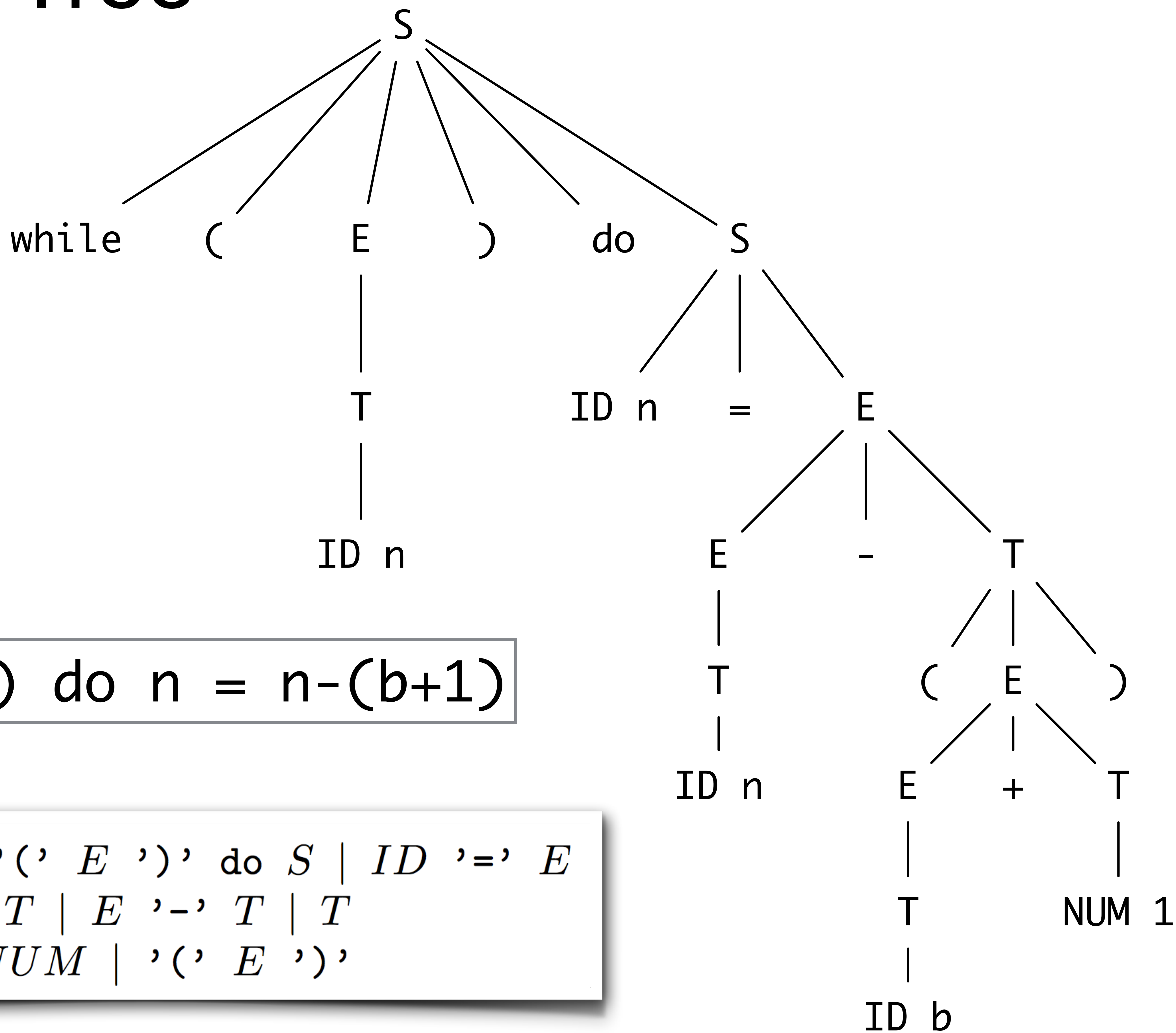
Limitations of CFGs

- CFGs do a great job at describing the syntactic structure of programming languages and identifying syntactically legal programs
- But there are many useful characteristics of legal programs that cannot be captured in a grammar (no matter how clever we are)
  - e.g. in many languages, variables must be declared before they are used, but this property cannot be captured in a CFG
- So checking program legality typically requires more than syntax analysis
  - Most compilers/interpreters use a secondary “semantic” analysis phase to check things like type-correctness.
  - Sometimes invalid programs cannot be detected until run-time

Parse trees vs. Abstract syntax Trees

- Parse trees reflect details of concrete syntax of program
  - Typically designed for easy parsing
- To process a program, we usually want a simpler, more abstract representation, the abstract syntax tree (AST)
  - No firm rules about AST design: matter of engineering taste

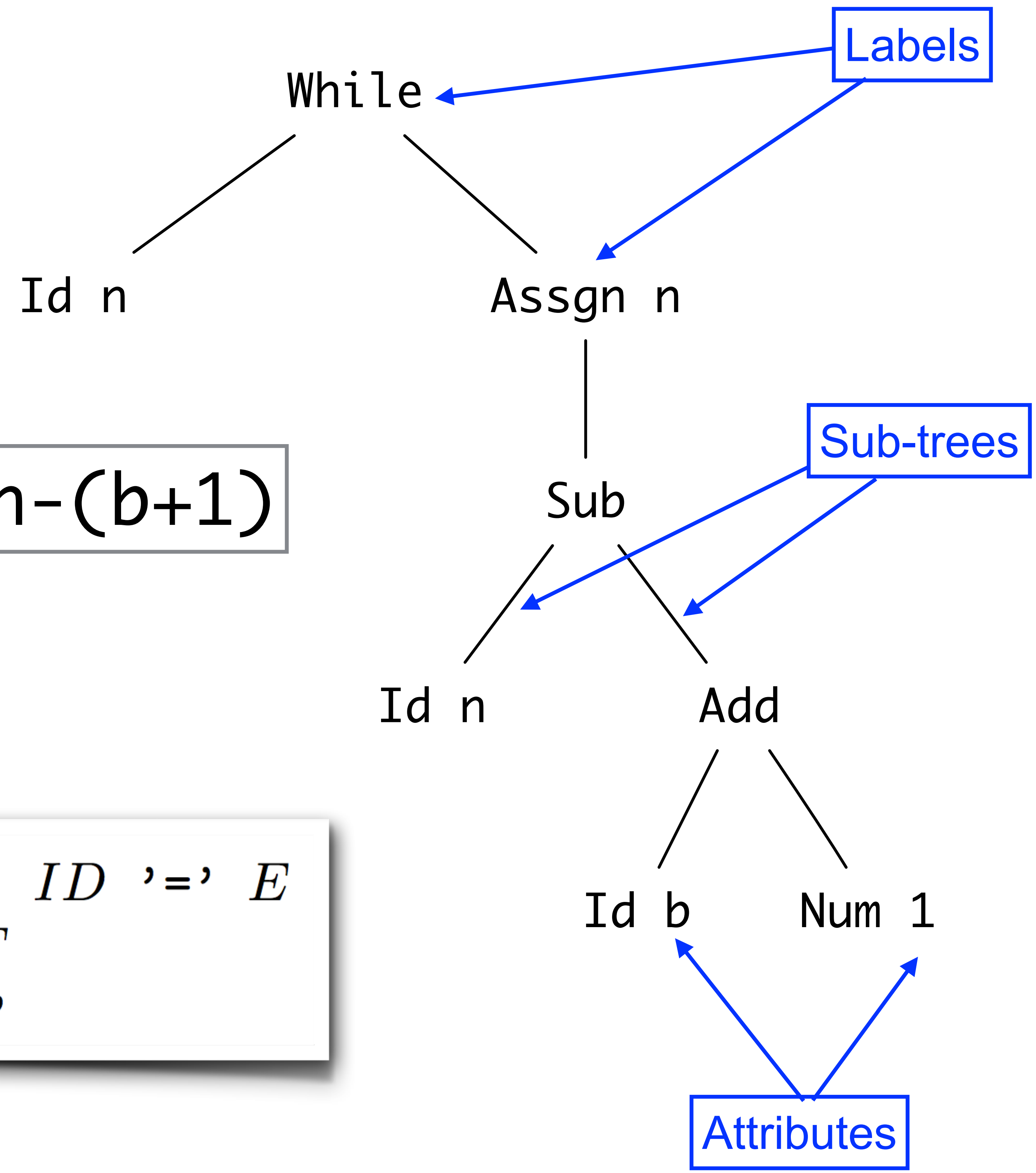
Parse Tree



while (n) do n = n-(b+1)

$$S \rightarrow \texttt{while}\ \text{'('}\ E\ \text{')'}\ \texttt{do}\ S \mid ID\ \text{'='}\ E$$
$$E \rightarrow E\ \text{'+'}\ T \mid E\ \text{'-'}\ T \mid T$$
$$T \rightarrow ID \mid NUM \mid \text{'('}\ E\ \text{')'}$$

Possible Abstract Syntax Tree




```
while (n) do n = n-(b+1)
```

$$S \rightarrow \texttt{while}\ \text{'('}\ E\ \text{')'}\ \texttt{do}\ S \mid ID\ \text{'='}\ E$$
$$E \rightarrow E\ \text{'+'}\ T \mid E\ \text{'-'}\ T \mid T$$
$$T \rightarrow ID \mid NUM \mid \text{'('}\ E\ \text{')'}$$

Tree Grammars

ASTs obey a tree grammar, with rules of the form

$label\text{: } kind \rightarrow (\; attr_1 \; \ldots \; attr_m \;) \; kind_1 \; \ldots \; kind_n$

where the LHS classifies the possible node labels into kinds, and the RHS gives the label's atomic attributes (if any) and the kinds of its subtrees (if any).

Example:

While : Stmt $\rightarrow$ *Exp Stmt*
Assgn : Stmt $\rightarrow$ *(string) Exp*
Add : Exp $\rightarrow$ *Exp Exp*
Sub : Exp $\rightarrow$ *Exp Exp*
Id : Exp $\rightarrow$ *(string)*
Num : Exp $\rightarrow$ *(int)*

Abstract syntax captures the essence

- Concrete syntax can have a big impact on the style and usability of a language…
- …and people love to argue about it…
- …but it is fundamentally superficial.
- The same abstract syntax can be used to represent many different concrete syntaxes…

Concrete alternatives

C-like:

```
while (n) do n = n - (b + 1);
```

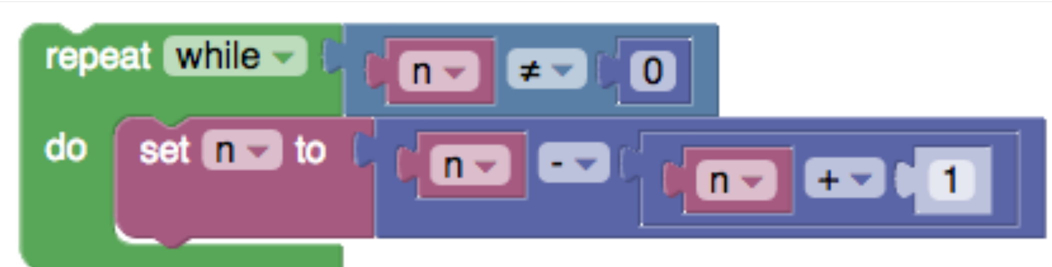
Fortran-like:

```
do while(n .NE. 0)
        n = n - (b + 1)
end do
```

COBOL-like:

```
      PERFORM 100-LOOP-BODY
        WITH TEST BEFORE
        WHILE N IS NOT EQUAL TO 0
 100_LOOP-BODY.
      ADD B TO 1 GIVING T
      SUBTRACT T FROM N GIVING N
```

Using a graphical notation:



developers.google.com/blockly/

ASTs in Scala

- ASTs have recursive structure and nodes have irregular size and shape, so we store them as dynamically-allocated heap records, one per tree node.

- In Scala, heap records are objects. We define classes corresponding to the various kinds, and a case class for each label, e.g.

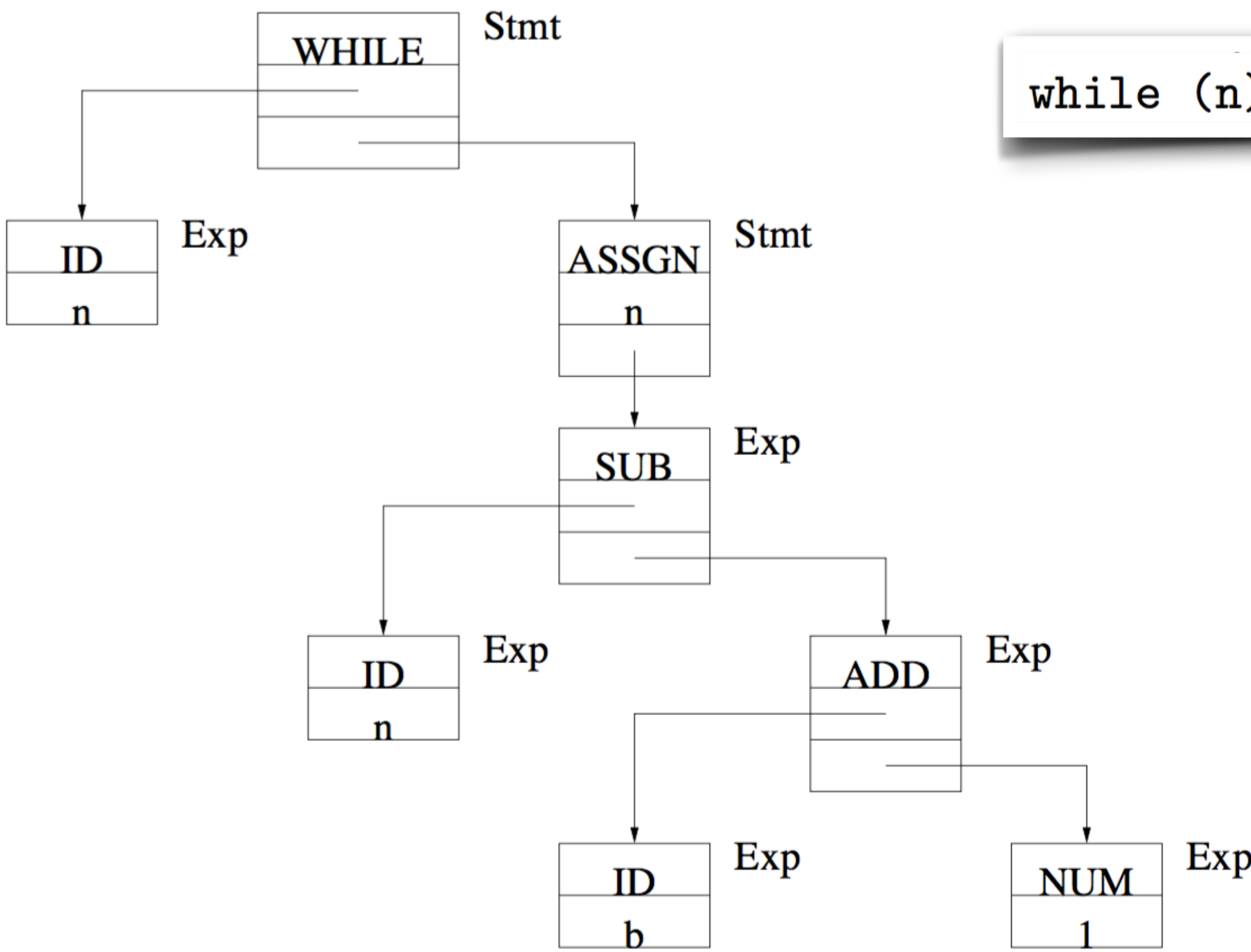
```
sealed abstract class Stmt
case class While(test:Exp,body:Stmt) extends Stmt
case class Assign(lhs:String,rhs:Exp) extends Stmt

sealed abstract class Exp
case class Add(left:Exp,right:Exp) extends Exp
case class Num(value:Int) extends Exp
```

- The case class declarations also define constructors, so we can write, e.g.

```
val e:Exp = Num(42)
```

AST Heap Structure

- Using these constructors, we generate a heap structure that is isomorphic to the AST




```
while (n) do n = n - (b + 1)
```

External Representation of ASTs

- Useful to give ASTs an external (concrete!) format so they can be read or written by programs or humans.
- External format must accurately record the internal tree structure, not just the sequence of leaves (“fringe”) of the tree.
- Can’t use the tree grammar to parse, since it is typically very ambiguous!
- We’ll represent trees using parenthesized prefix notation
  - also called s-expressions (from the LISP language)

s-expression example

Each AST node is represented by an expression

$$(\textit{label}\ \textit{attr}_1 \ldots \textit{attr}_m\ \textit{child}_1\ \ldots\ \textit{child}_n\ )$$

where *label* is the node label, the $attr_i$ are the label's attributes (if any) and the $child_i$ are the label's sub-trees (if any), each of which is itself a node expression

```
(While (Id n)
       (Assgn n (Sub (Id n)
                     (Add (Id b)
                          (Num 1)))))
```

line breaks and indentation have been added to improve readability

For readability, we can use abbreviations for common labels, such as `+` for `Add`. We can also omit labels on leaves, e.g. use `3` for `Num 3`, or `b` for `Id b`, if no confusion would arise.

```
(While n
       (Assgn n (- n
                   (+ b 1))))
```

Parsing s-expressions

- Parsing s-expressions into AST nodes is easy!
  - Everything is either an atom (keyword, symbol, numeric literal, etc.) or a parenthesized list of atoms
  - Can divide parsing into two parts:
    - Language-independent parse into generic s-expression type (atoms and lists)
    - Language-dependent analysis of generic s-expression to recover AST
      - First item in each list is a symbol that tells the node type and implies the kind and number of remaining list items