CS558 Programming Languages

Fall 2015

Lecture 1b

Language Description and Documentation

For programmers, compiler-writers, and students . . .

Syntax (Easy)

What do programs look like?

- Grammars; BNF and Syntax Charts

Semantics (Hard)

What do programs do?

- Informal: Reference manuals, user guides, examples
- Formal: Operational, Denotational, Axiomatic
- Experimental: Try running the program and see what happens!

SYNTAX: CONCRETE & ABSTRACT

- Language syntax describes the legal form and structure of programs
- Concrete syntax describes is what a program looks like on the page or screen
- Abstract syntax describes the essential contents of a program as it might be represented internally (e.g. by an interpreter or compiler)
- In this course, we won't worry much about concrete syntax...
- ...but it is actually quite important to the **style** of a language
- Concrete syntax is defined by a **context-free grammar**

Simple Formal Example Grammar

Character set: $\{(,)\}$

Terminals: $\{(,)\}$

Non-terminals: $\{S\}$

Productions:

S ::= (S)

S ::= SS

S ::= ϵ (the empty string)

Starting non-terminal: S

Sample derivation:

$S \rightarrow (S) \rightarrow (SS) \rightarrow ((S)S) \rightarrow (()S) \rightarrow (()(S)) \rightarrow (()())$

This grammar generates the language of strings of properly matched parentheses.

It is often useful to think of a derivation as a **tree** (more shortly).

CONTEXT-FREE GRAMMARS

- Used for description, parsing, analysis, etc.
- Especially good for **recursive** definition of program structure
- A grammar is defined by:
  - a set of **terminal** symbols (strings of characters)
  - a set **non-terminal** variables, which represent sets of strings of terminals
  - a set of **production** rules that map non-terminals to strings of terminals and non-terminals
- The **language** $L(G)$ defined by a grammar G is the **set of strings of terminals** that can be derived by applying production rules beginning from a specified **start symbol** (a non-terminal).
- Grammars have rich theory with connections to automatic parser generation, push-down automata, etc.
- Many possible representations, including **BNF** (Backus-Naur Form), **EBNF** (Extended BNF), syntax charts, etc.

BNF was invented ca. 1960 and used in the formal description of Algol-60. It is just a particular notation for grammars, in which

- Non-terminals are represented by names inside angle brackets, e.g., <*program*>, <*expression*>, <*S*>.
- Terminals are represented by themselves, e.g., `WHILE`, `(`, `3`. The empty string is written as <*empty*>.

BNF Example...

<*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 is (any) extension of BNF, usually with these features:

- A vertical bar, $|$, represents a choice,
- Parentheses, (and), represent grouping,
- Square brackets, [and], represent an optional construct,
- Curly braces, $\{$ and $\}$, represent zero or more repetitions,
- Non-terminals begin with upper-case letters.
- Non-alphabetic terminal symbols are quoted, at least when necessary to avoid confusion with the meta-symbols above.

EBNF EXAMPLE

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	*::=*	*. . .*

SYNTAX ANALYSIS (PARSING)

Parser **recognizes** syntactically legal programs (as defined by a grammar) and **rejects** illegal ones.

- Successful parse also captures **hierarchical** structure of programs (expressions, blocks, etc.).

- Convenient representation for further semantic checking (e.g., typechecking) and for code generation.

- Failed parse provides error feedback to the user indicating where and why the input was illegal.

Any context-free language can be parsed by a computer program, but only **some** can be parsed **efficiently**. Modern programming languages can usually be parsed efficiently.

LEXICAL ANALYSIS

Programming language grammars usually take simple **tokens** rather than characters as terminals. Converting raw program text into token stream is job of the **lexical analyzer**, which

- Detects and identifies keywords and identifiers.
- Converts multi-character symbols into single tokens.
- Handles numeric and string literals.
- Removes whitespace and comments.

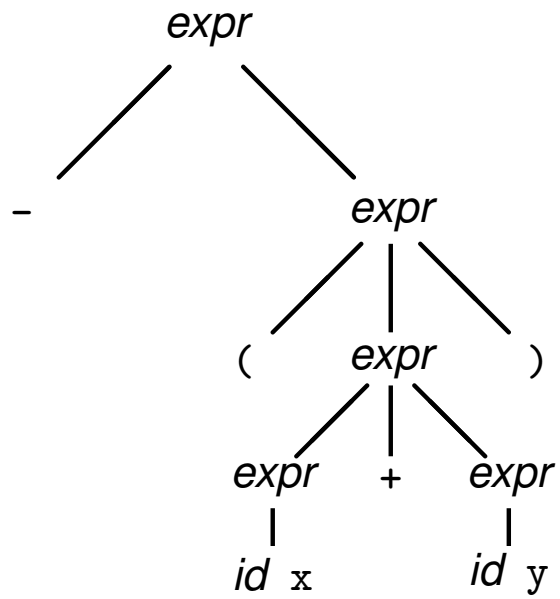
PARSE TREES

Graphical representation of a derivation.

Given this grammar:

$$expr \rightarrow expr + expr \mid expr * expr \mid (expr) \mid -expr \mid id$$

Example tree for derivation of sentence `-(x + y)` :



Each application of a production corresponds to an **internal** node, labeled with a **non-terminal**.

Leaves are labeled with **terminals**, which can have **attributes** (in this case the specific identifier name).

The derived sentence is found by reading leaves (or "fringe") left-to-right.

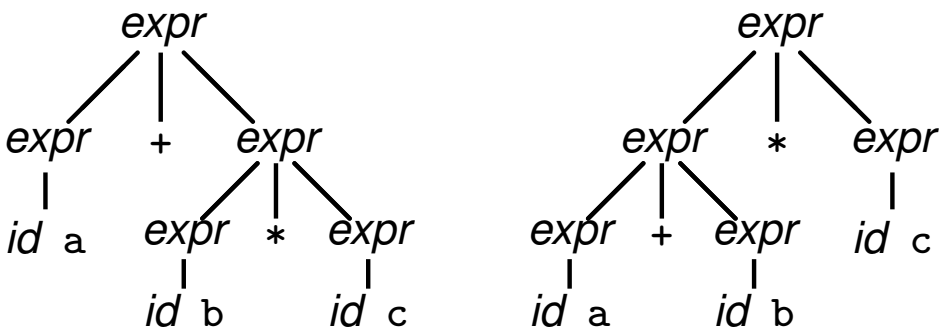
AMBIGUITY

A given **sentence** in $L(G)$ can have more than one parse tree.
Grammars G for which this is true are called **ambiguous**.

Example: given the grammar on the last slide, the sentence

```
a + b * c
```

has two parse trees:



We may think of the left tree as being the "correct" one, but nothing in the grammar says this.

To avoid the problems of ambiguity, we can:

- Rewrite grammar; or
- Use "disambiguating rules" when we implement parser for grammar.

AMBIGUITY IN ARITHMETIC EXPRESSIONS

To disambiguate a grammar like

$$E \rightarrow E \texttt{ + } E \mid E \texttt{ - } E \mid E \texttt{ * } E \mid E \texttt{ / } E \mid \texttt{(}E\texttt{)} \mid \texttt{id}$$

we need to make choices about the desired order of operations.

For any expression of the form `X` op_1 `Y` op_2 `Z` we must define:

- **Precedence** - which operation (op_1 or op_2) is done first?
- **Associativity** - if op_1 and op_2 have the same precedence, then does `Y` "associate" with the operator on the left or on the right?

In other words, we need rules to tell us whether the expression is equivalent to `(X` op_1 `Y)` op_2 `Z` or to `X` op_1 `(Y` op_2 `Z)`.

The "usual" rules (based on common usage in written math) give `*` and `/` higher precedence than + and −, and make all the operators left-associative.

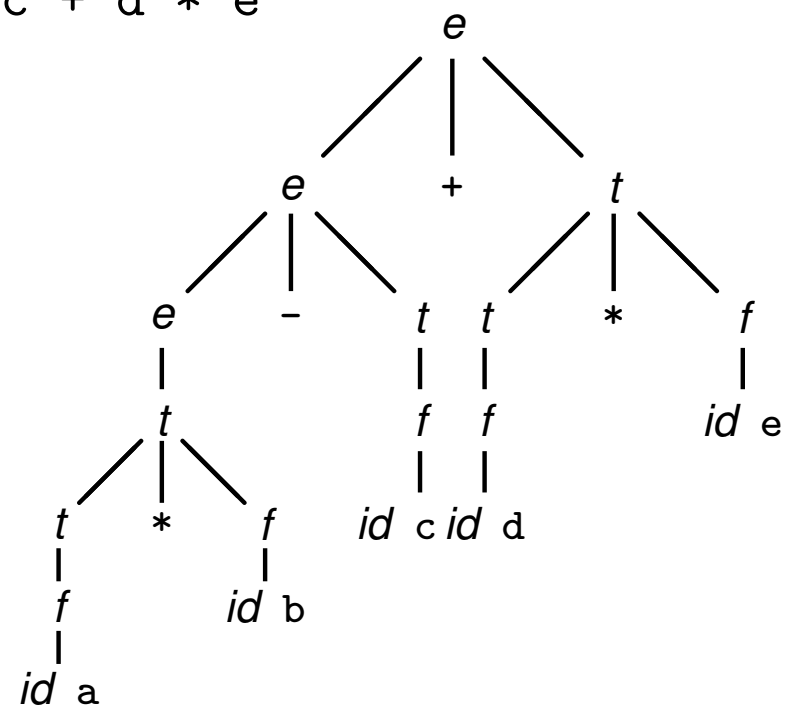
So, for example, `a - b - c * d` is equivalent to `(a - b) - (c * d)`. But this is a matter of **choice** when defining the language.

REWRITING ARITHMETIC GRAMMARS

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

factor	$\rightarrow$	(*expr*) | `id`
term	$\rightarrow$	*term* `*` *factor* | *term* `/` *factor* | *factor*
expr	$\rightarrow$	*expr* `+` *term* | *expr* `-` *term* | *term*

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



LIMITATIONS OF CONTEXT-FREE GRAMMARS

Context-free grammars are very useful for describing the structure of programming languages and identifying legal programs.

But there are many useful characteristics of legal programs that **cannot** be captured in a grammar (no matter how clever we are).

For example, in many programming languages, every variable in a legal program must be declared before it is used. But this property cannot be captured in a grammar.

So checking legality of programs typically requires more than syntax analysis. Most compilers use a secondary "semantic" analysis phase to check non-syntactic properties, such as type-correctness. Of course, sometimes invalid programs cannot be detected until runtime.

PARSE TREES VS. ABSTRACT SYNTAX TREES

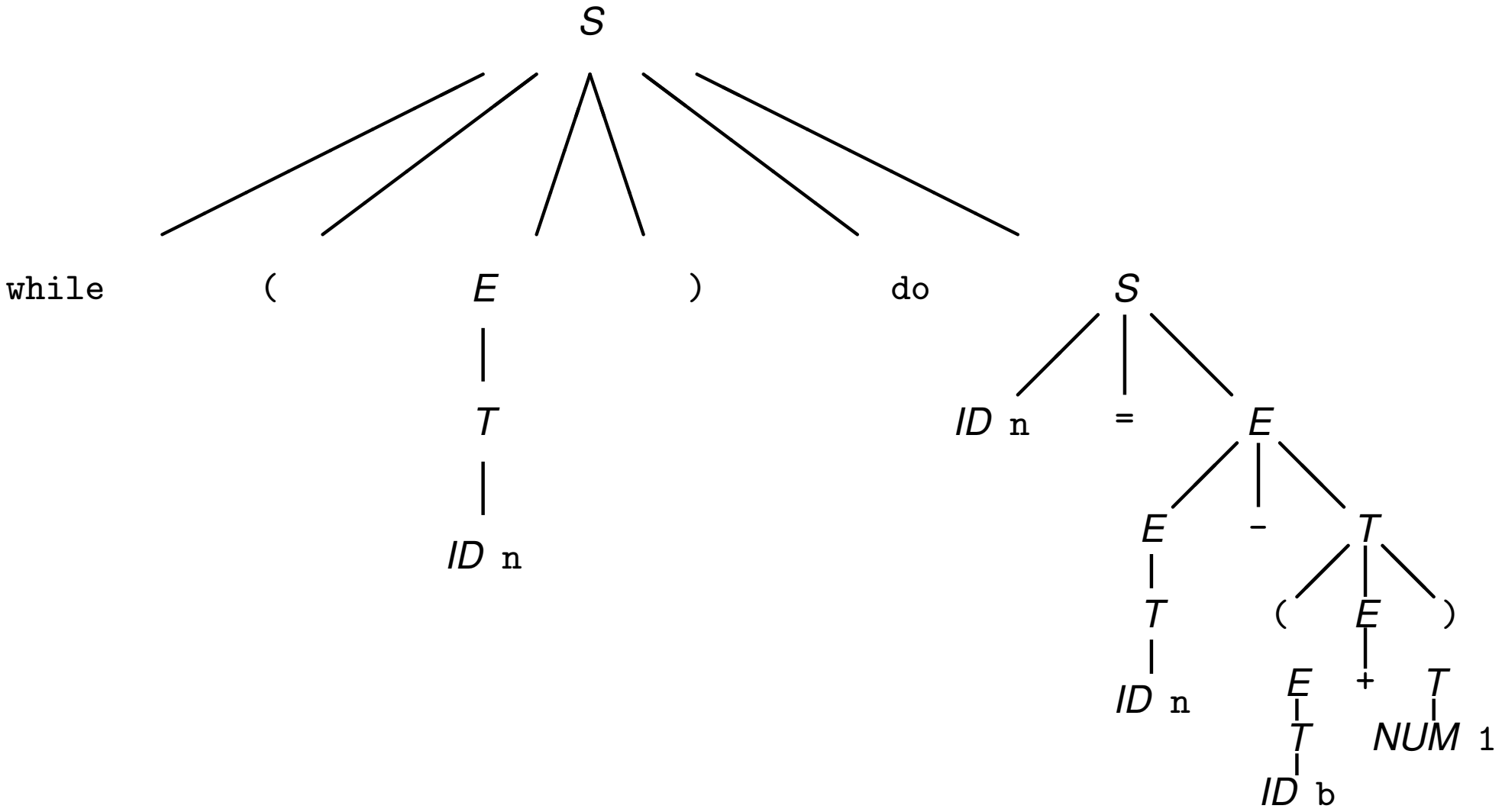
Parse trees reflect details of the **concrete** syntax of a program, which is typically designed for easy parsing.

For processing a language, we usually want a **simpler**, more **abstract** view of the program. (No firm rules about AST design: matter of taste, convenience.)

Simple concrete grammar:

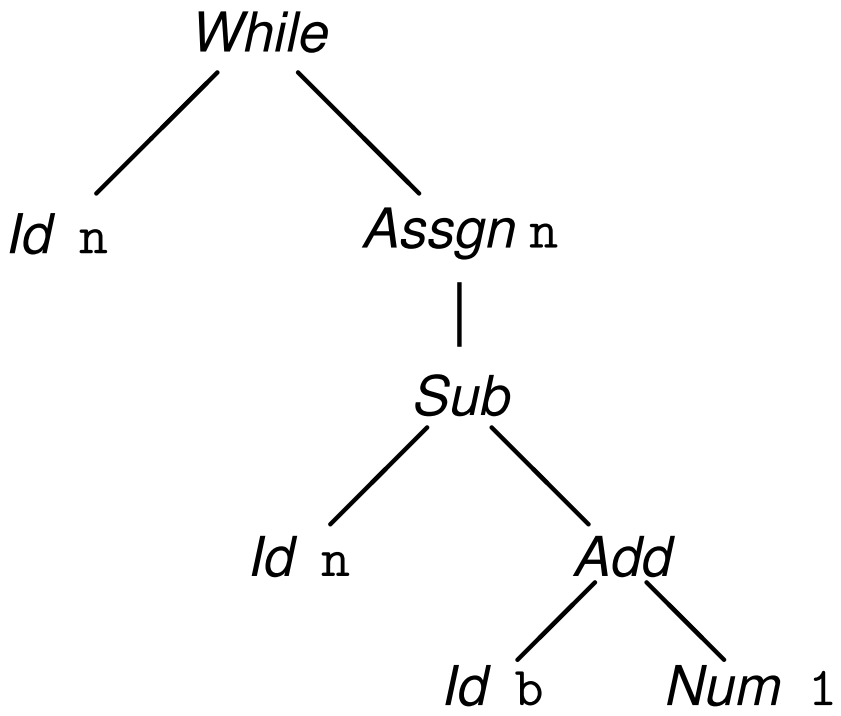
$$S \rightarrow \texttt{while '(' } E \texttt{ ')' do } S \mid ID \texttt{ '=' } E$$
$$E \rightarrow E \texttt{ '+' } T \mid E \texttt{ '-' } T \mid T$$
$$T \rightarrow ID \mid NUM \mid \texttt{'(' } E \texttt{ ')'}$$

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



PARSE TREES VS. ABSTRACT SYNTAX TREES (2)

Possible abstract syntax tree for `while (n) do n = n - (b + 1)`



Note that tree nodes may have **attributes** (such as the name of an *Id*) and/or **sub-trees**.

TREE GRAMMARS

AST's obey a **tree grammar**. Rules have form

label : kind $\rightarrow$ *(attr*$_1$ $\ldots$ *attr*$_m$*) kind*$_1$ $\ldots$ *kind*$_n$

where the LHS classifies the possible node **labels** into **kind**s, and the RHS describes the label's atomic **attributes** (if any, in parentheses) 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 is important for usability, but fundamentally superficial. The same abstract syntax can be used to represent many different concrete syntaxes.

Examples:

- C-like:

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

- Fortran-like:

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

Concrete Syntax Examples (2)

- 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
```

- Use Chinese keywords in place of `while` and `do`.
- Use a graphical notation.

AST'S IN SCALA

AST's have recursive structure and irregular shape and size, so it makes sense to store them as **heap** data structures using one record for each 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.

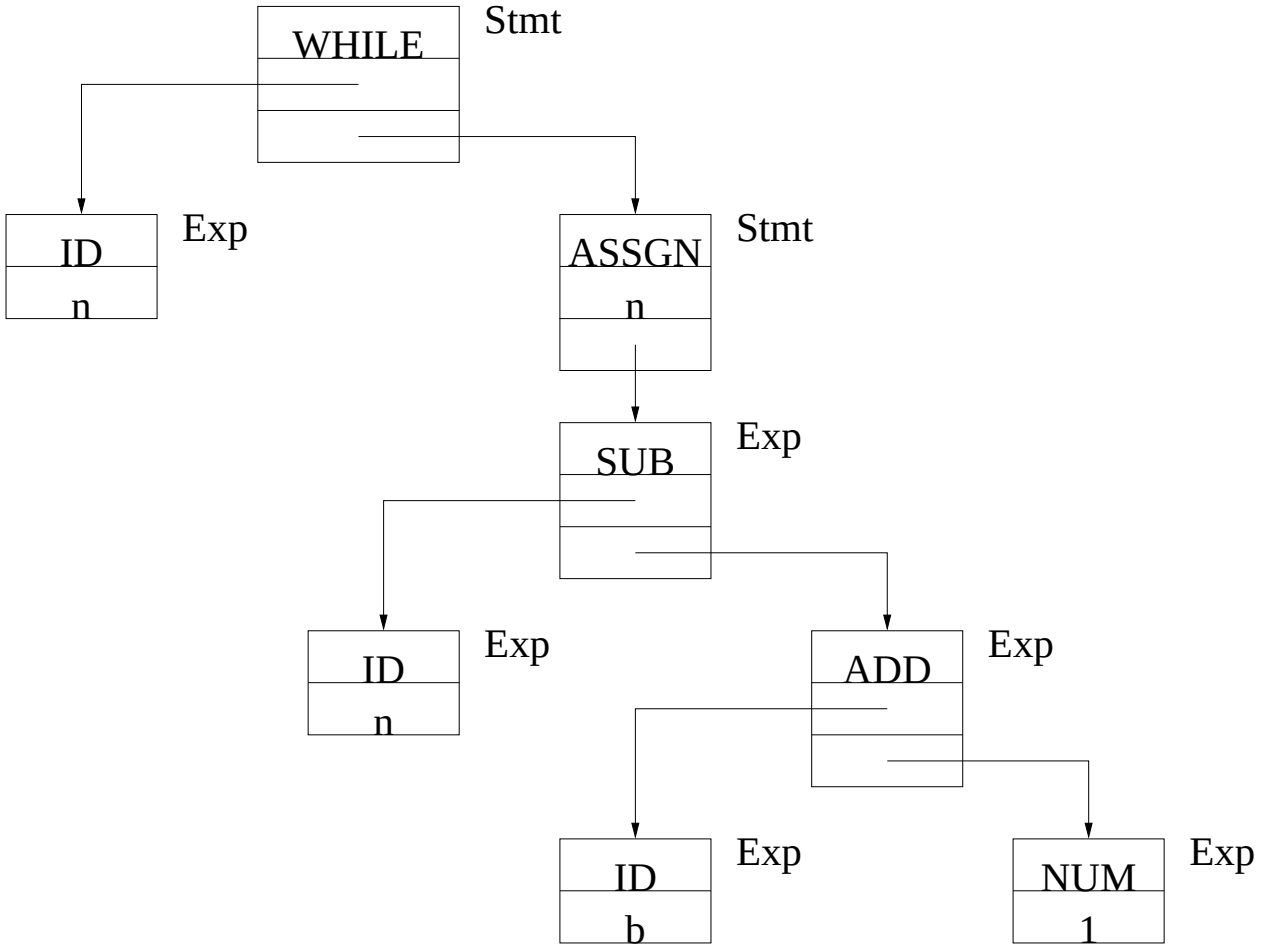
```
sealed abstract class Stmt
case class While(test:Exp,body:Stmt) extends Stmt
case class Assign(lhs:String,rhs:Exp) extends Stmt

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

The `case clase` declarations also define constructors, so we can say, e.g., `val e:Exp = Num(42)`.

HEAP STRUCTURE

Using these constructors, we generate a heap structure that is isomorphic to the AST tree, e.g. for `while (n) do n = n - (b + 1)`



External Representation of ASTs

Although ASTs are designed as an **internal** program representation, it can be useful to give them an **external** form too that can be read or written by other programs or by humans.

Any external representation of ASTs must accurately reflect the internal tree structure as well as the "fringe" of the tree. Can't use tree grammar to parse, since it is typically ambiguous!

We will represent trees using **parenthesized prefix notation**, also called **s-expressions** (the name comes from the programming language LISP).

Each node in the tree is represented by the expression

(*label* $attr_1$ ... $attr_m$ $child_1$ $child_n$)

where *label* is the node label, the $attr_i$ are the label's attributes (if any), and the $child_i$ are the labels sub-trees (if any), each of which is itself a node expression. To make things more readable, we might use abbreviations for common labels, e.g., + for `Add`. We may also represent simple leaf nodes by their bare attributes, e.g., use `3` for `(Num 3)`, as long as no confusion can arise.

External AST Example

So the representation of our AST example could be

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

where the indentation is optional, but makes the representation easier for humans to read. The BNF for this syntax can be given as:

```
<stmt> ::= (<While> <expr> <stmt>)
       |   (<Assgn> <name> <expr>)

<expr> ::= (+ <expr> <expr>)
       |   (- <expr> <expr>)
       |   <name>
       |   <int>
```

Concrete and abstract syntax are isomorphic.

PARSING PARANTHESIZED PREFIX NOTATION

Note that this BNF description contains essentially the same information as our AST internal datatype declarations (except for the descriptive field names).

Parsing this representation is easy! Why?

- Everything is either an atom (keyword, symbol, numeric literal, etc.) or a parenthesized list of atoms
- Each node in the AST corresponds to a list or atom.
- The first item in each list is always a symbol that identifies the node type and implies the kind and number of the remaining things in the list.
- The lexical analyzer just needs to identify the atoms and list delimiters.