CS558 Programming Languages

Winter 2008

Lecture 2

Grammars

- Used for description, parsing, analysis, etc.
- Based on **recursive** definition of program structure.
- 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

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

- Nonterminals 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

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,

• Nonterminals 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	::=	...

GRAMMARS, FORMALLY

A **(context-free) Grammar** over a given **character set** consists of

• A set of **terminals**, which are strings of zero or more characters.

• A set of **nonterminals**, which are variables representing a set of terminals.

• A set of **productions**, each of which has a **left side** consisting of a single nonterminal and a **right side** consisting of zero or more terminals or nonterminals.

• A distinguished **starting nonterminal**.

We can **apply** a production to a string by replacing some instance of the left side nonterminal by the right side.

The **context-free language** $L(G)$ **generated** by a grammar G is the set of character strings that can be **derived** from the starting nonterminal by applying productions—in any order—until no nonterminals remain.

Note: Here we are using "**language**" in the technical sense meaning "a well-defined set of strings over the specified character set".

SIMPLE EXAMPLE GRAMMAR

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

Terminals: $\{\texttt{(},\texttt{)}\}$

Nonterminals: $\{S\}$

Productions:

S `::=` $\texttt{(}S\texttt{)}$

S `::=` SS

S `::=` ϵ (the empty string)

Starting nonterminal: S

Sample derivation:

$$S \rightarrow \texttt{(}S\texttt{)} \rightarrow \texttt{(}SS\texttt{)} \rightarrow \texttt{((}S\texttt{)}S\texttt{)} \rightarrow \texttt{(()}S\texttt{)} \rightarrow \texttt{(()(}S\texttt{))} \rightarrow \texttt{(()())}$$

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).

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.

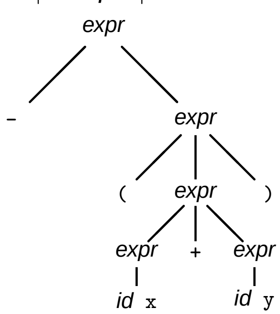
Parse Trees

Graphical representation of a derivation.

Given this grammar:

expr → *expr* + *expr* | *expr* ∗ *expr* | (*expr*) | -*expr* | *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.

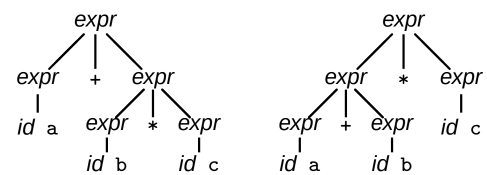
Ambiguity

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.

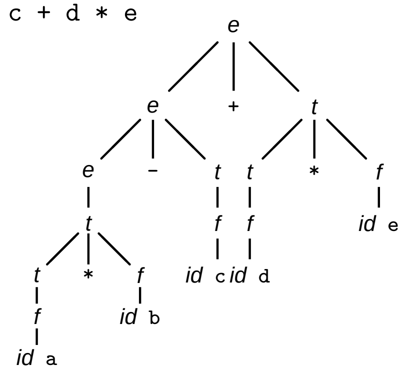
So, for example, `a - b - c * d` is equivalent to `(a - b) - (c * d)`. But this is a matter of **choice** when defining the language.

Rewriting Arithmetic Grammars

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

$$\begin{array}{lcl} \textit{factor} & \rightarrow & (\textit{expr}) \mid \texttt{id} \\ \textit{term} & \rightarrow & \textit{term} \texttt{ * } \textit{factor} \mid \textit{term} \texttt{ / } \textit{factor} \mid \textit{factor} \\ \textit{expr} & \rightarrow & \textit{expr} \texttt{ + } \textit{term} \mid \textit{expr} \texttt{ - } \textit{term} \mid \textit{term} \end{array}$$

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.

To show this formally, we can abstract the notion of "declaration before use" into a formal language

$$L = \{wcw \mid w \in (a \mid b)^*\}$$

It can be easily shown that no context-free grammar generates L.

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 illegal programs cannot be detected until runtime.

Parse Trees vs. Abstract Syntax Trees

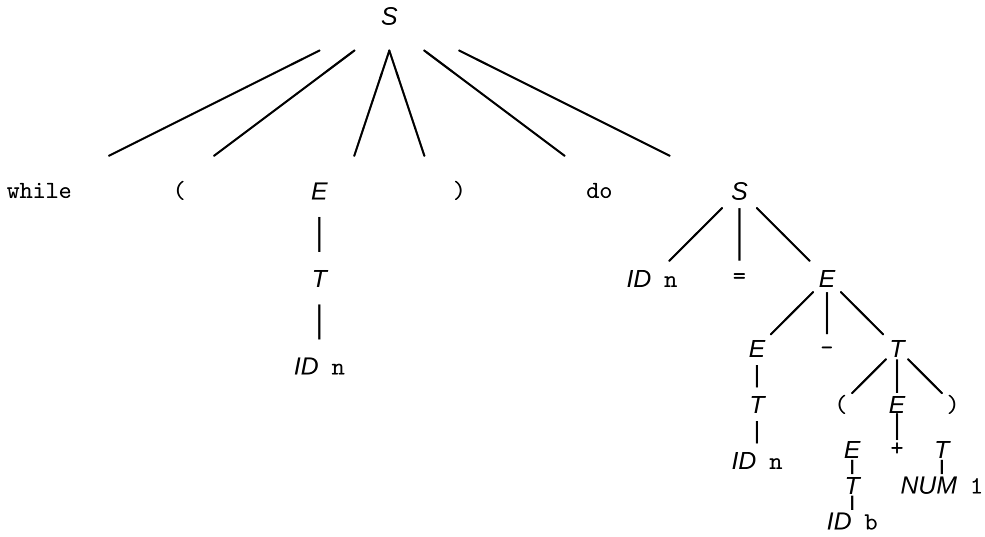
Parse trees reflect details of the **concrete** syntax of a program, which is typically designed for easy parsing.

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:

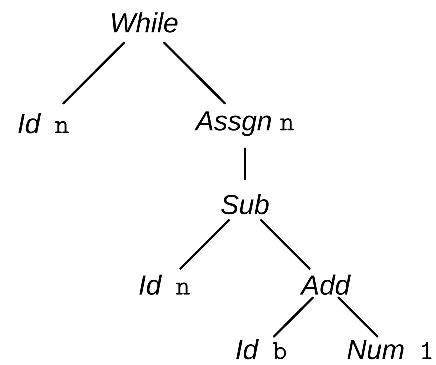
$$\begin{array}{l} S \rightarrow \texttt{while '(' } E \texttt{ ')' do } S \mid ID \texttt{ '=' } E \\ E \rightarrow E \texttt{ '+' } T \mid E \texttt{ '-' } T \mid T \\ T \rightarrow ID \mid NUM \mid \texttt{'(' } E \texttt{ ')'} \end{array}$$

`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)`



TREE GRAMMARS

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

label : kind $\longrightarrow$ *(attr*$_1$ ... *attr*$_m$*) kind*$_1$ ... *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 $\longrightarrow$ *Exp Stmt*
Assgn : Stmt $\longrightarrow$ *(string) Exp*
Add : Exp $\longrightarrow$ *Exp Exp*
Sub : Exp $\longrightarrow$ *Exp Exp*
Id : Exp $\longrightarrow$ *(string)*
Num : Exp $\longrightarrow$ *(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.

Internal Representation of ASTs in C

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 C, Pascal, Ada, etc., we might use **unions** or **variant** records for the different node labels of each node kind. E.g., in C:

```
struct stmt {
  enum { While, Assgn, ... } label;
  union {
    struct {
      struct exp *test;
      struct stmt *body;
    } while_s;
    struct {
      char *lhs; struct exp *rhs;
    } assgn_s;
    ...
  } u;
}
```

```
struct exp {
  enum { Add, ..., Num } label;
  union {
    struct {
      struct exp *left;
      struct exp *right;
    } add_e;
    ...
    struct {
      int value;
    } num_e;
  } u;
}
```


AST's in Java

In Java, heap records are **objects**. We define **classes** corresponding to the various kinds and a subclass for each label, e.g.

```
abstract class Stmt { }
class While extends Stmt {
  Exp test;
  Stmt body;
}
class Assgn extends Stmt {
  String lhs; Exp rhs;
}
...
abstract class Exp { }
class Add extends Exp {
  Exp left; Exp right;
}
...
class Num extends Exp {
  int value;
}
```


AST's in ML

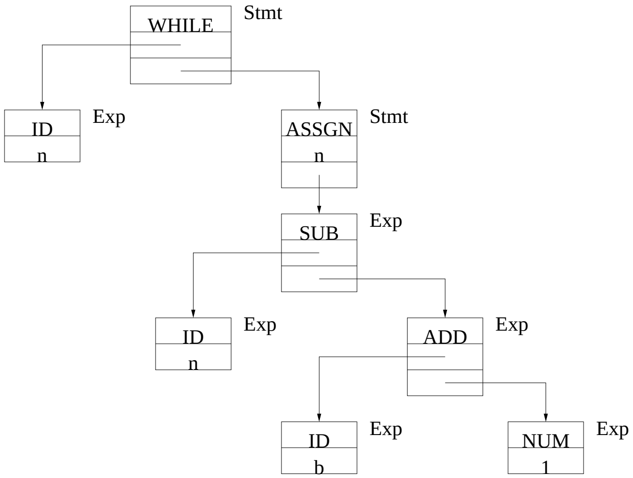
In ML, we can use **datatypes** to define a suitable set of variants.

```
datatype stmt = While of exp * stmt
              | Assgn of string * exp
              | ...
and exp = Add of exp * exp
        | ...
        | Num of int
```

Heap Structure

All these approaches generate roughly the **same** heap structures, 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!

One approach (deriving from the programming language LISP) is to use **parenthesized prefix notation** to represent trees.

Each node in the tree is represented by the expression

$$(\; label \;\; attr_1 \; \ldots \;\; attr_m \;\; child_1 \;\; child_n \;)$$

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`.

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.

Parsing this representation is easy!