COMP 356
Programming Language Structures
Notes for Classes 1–3 (Chapter 3 of *Concepts of Programming Languages*)
Syntax and Semantics

Acknowledgment: The majority of these notes were written by Prof. Tim Wahls; however, some significant changes have been made.

1 Introduction: why study the structure of programming languages?

The theory of computability tells us that all modern programming languages are *Turing complete*. In other words, these languages can be used to solve exactly the same set of problems (the set of *computable* problems). In fact, some surprisingly minimal subsets of a language are Turing complete (e.g. `if` and `goto` are sufficient; `if` and `while` are sufficient).

So, (1) why would we ever need more than one programming language? And (2) why would we bother to study the implementation of and differences between programming languages?

Possible answers to (1) include:

- different languages make it easier or harder to achieve certain tasks. Examples?
- different languages achieve different performance for certain tasks. Examples?

Possible answers to (2) include:

- Understanding implementation of a language can help improve the quality of the programs you write in that language. Examples?
- Understanding different languages broadens your selection of problem-solving techniques. Examples?
- Various practical advantages e.g. Knowing several languages makes you more attractive to an employer; you can select an appropriate language for a given task.
- Some fundamental ideas in the theory of computer science (e.g. recursion, abstraction) are reflected in the design of some languages. Examples?

See Section 1.1 of the textbook for some more detailed discussion.

2 Syntax and semantics overview

Definitions:

- **syntax** is the form (structure, grammar) of a language
- **semantics** is the meaning of a language

Example:

```
if (a > b) a = a + 1;
else b = b + 1;
```

- syntax: `if-else` is an operator that takes three operands—a condition and two statements
- semantics: if the value of `a` is greater than the value of `b`, then increment `a`. Otherwise, increment `b`.

Both the syntax and semantics of a programming language must be carefully defined so that:

- language implementors can implement the language (correctly), so that programs developed with one implementation run correctly under another (portability)
- programmers can use the language (correctly)

3 Describing Syntax

A **language** is a set of strings of characters from some alphabet.

Examples:

- binary numbers (using the alphabet {0, 1})
- Java identifiers
- Java programs that compile with no errors
- English words (assuming we agree on a specific dictionary)
- English sentences (assuming we agree on a dictionary and all the grammatical rules of English, which probably isn't possible in practice)

The syntax rules of a language determine whether or not arbitrary strings belong to the language. The first step in specifying syntax is describing the basic units or "words" of the language, called **lexemes**. For example, some typical Java lexemes include:

- `if`
- `++`
- `+`
- `3.27`
- `count`

Lexemes are grouped into categories called **tokens**. Each token has one or more lexemes.

token	**sample lexemes**
identifier	`count`, `i`, `x2`, ...
if	`if`
add_op	`+`, `-`
semi	`;`
int_lit	`0`, `10`
double_lit	`-2.4`

Table 1: Example tokens and some associated lexemes.

Tokens are specified using regular expressions or finite automata.

The scanner/lexical analyzer of a compiler processes the character strings in the source program and determines the tokens that they represent.

Once the tokens of a language are defined, the next step is to determine which sequences of tokens are in the language (syntax).

3.1 Using BNF to Describe Syntax

BNF ≡ Backus-Naur Form

BNF is:

- a **metalanguage**—a language used to describe other languages
- the standard way to describe programming language syntax
- often used in language reference manuals

The class (set) of languages that can be described using BNF is called the **context-free languages**, and BNF descriptions are also called **context-free grammars** or just **grammars**.

symbol	meaning
$\rightarrow$	is defined as
|	or (alternatives)
⟨something⟩	a **nonterminal**—replace by the definition of something
something	(with no ⟨ ⟩) a **token** or **terminal**—a "word" in the language being defined (not replaced)
ϵ	the empty string (nothing)

Table 2: BNF notation.

Example (if statements in C):

⟨if-stmt⟩ → if (⟨expr⟩) ⟨stmt⟩
⟨if-stmt⟩ → if (⟨expr⟩) ⟨stmt⟩ else ⟨stmt⟩

Each definition above is called a **rule** or **production**. A full BNF definition would include definitions for all the nonterminals used—⟨expr⟩ and ⟨stmt⟩ are not defined above.

The two rules above can be abbreviated using | as follows:

⟨if-stmt⟩ → if (⟨expr⟩) ⟨stmt⟩
| if (⟨expr⟩) ⟨stmt⟩ else ⟨stmt⟩

Example (expressions in C):

⟨expr⟩ → id | num | (⟨expr⟩) | ⟨expr⟩⟨op⟩⟨expr⟩
⟨op⟩ → `+` | `-` | `*` | `/` | `==` | `<` | `<=` | `>` | `>=`

Note that BNF definitions can be recursive.

3.2 Derivations

A **derivation** is a sequence of replacements using the rules of a grammar. Derivations:

- are often used to show that a particular sequence of tokens belongs to the language defined by the grammar
- always begin with the **start symbol** for the grammar
  - by tradition, the nonterminal on the LHS of the first rule is the start symbol
  - o.w. the start symbol could have a special name such as ⟨start⟩ or ⟨program⟩

Example: A derivation to show that:

id + num

is a valid expression (⟨expr⟩):

⟨expr⟩ ⇒ ⟨expr⟩ ⟨op⟩ ⟨expr⟩
⇒ id ⟨op⟩ ⟨expr⟩
⇒ id + ⟨expr⟩
⇒ id + num

Example: A derivation to show that:

id < (num / num)

is a valid expression (⟨expr⟩):

⟨expr⟩ ⇒ ⟨expr⟩ ⟨op⟩ ⟨expr⟩
⇒ id ⟨op⟩ ⟨expr⟩
⇒ id < ⟨expr⟩
⇒ id < (⟨expr⟩)
⇒ id < (⟨expr⟩ ⟨op⟩ ⟨expr⟩)
⇒ id < (num ⟨op⟩ ⟨expr⟩)
⇒ id < (num / ⟨expr⟩)
⇒ id < (num / num)

Definitions:

- the symbol ⇒ is called **derives**
- each string of symbols derived from the start symbol (including the start symbol itself) is called a **sentential form**
- a **leftmost derivation** is a derivation in which the leftmost nonterminal is always chosen for replacement
- a **rightmost derivation** is a derivation in which the rightmost nonterminal is always chosen for replacement

Derivation order has no effect on the set of strings that can be derived.
BNF example: A subset of statements in C:

⟨stmt⟩ → ⟨if-stmt⟩ | ⟨loop-stmt⟩ | ⟨assign-stmt⟩ | ⟨cmpd-stmt⟩
⟨if-stmt⟩ → if (⟨expr⟩) ⟨stmt⟩
| if (⟨expr⟩) ⟨stmt⟩ else ⟨stmt⟩
⟨loop-stmt⟩ → while (⟨expr⟩) ⟨stmt⟩
⟨assign-stmt⟩ → id = ⟨expr⟩ ;
⟨cmpd-stmt⟩ → { ⟨stmt-list⟩ }
⟨stmt-list⟩ → ϵ | ⟨stmt⟩ | ⟨stmt⟩ ⟨stmt-list⟩
⟨expr⟩ → id | num | (⟨expr⟩) | ⟨expr⟩ ⟨op⟩ ⟨expr⟩
⟨op⟩ → `+` | `-` | `*` | `/` | `==` | `<` | `<=` | `>` | `>=`

Even more definitions:

- a BNF rule in which the nonterminal on the LHS is also the first (leftmost) symbol on the RHS is **left recursive**
- a BNF rule in which the nonterminal on the LHS is also the last (rightmost) symbol on the RHS is **right recursive**

For example:

- ⟨stmt-list⟩ → ⟨stmt⟩ ⟨stmt-list⟩
  is right recursive
- ⟨expr⟩ → ⟨expr⟩ ⟨op⟩ ⟨expr⟩
  is both left and right recursive

3.3 Parse Trees

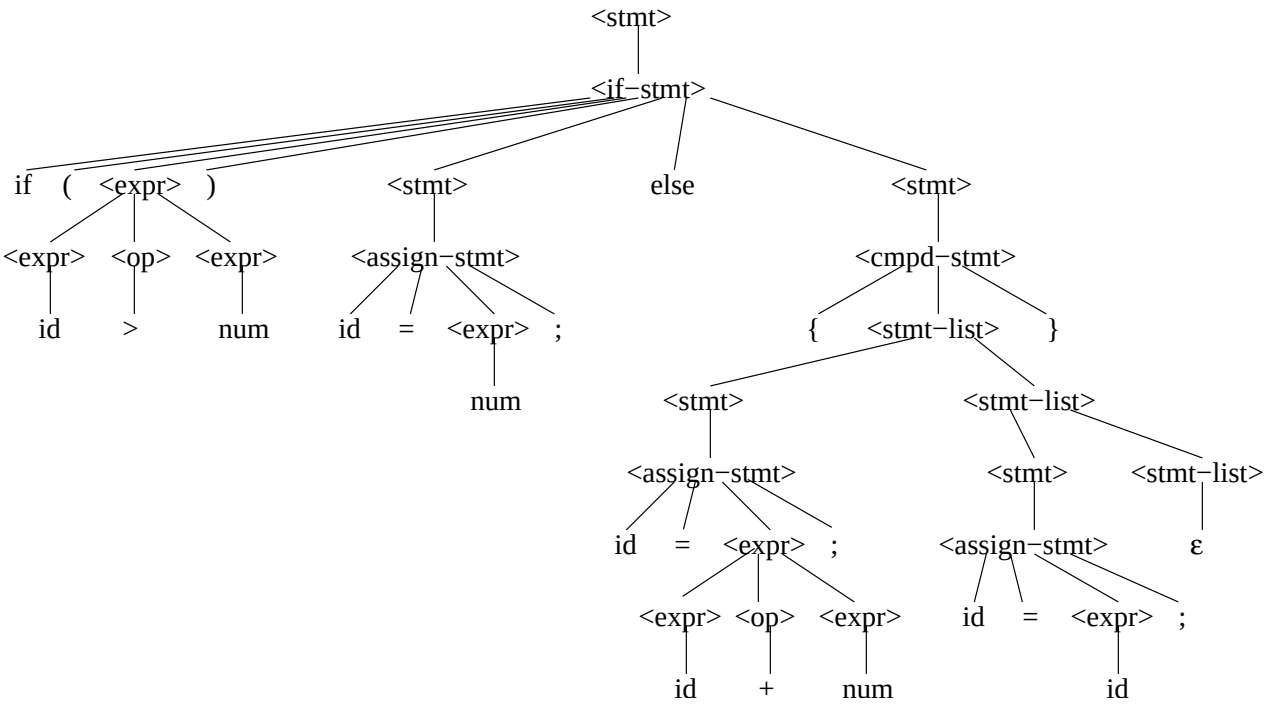
A **parse tree** is a graphical way of representing a derivation.

- the root of the parse tree is always the start symbol
- each interior node is a nonterminal
- each leaf node is a token
- the children of a nonterminal (interior node) are the RHS of some rule whose LHS is the nonterminal

For example, a parse tree for:

if (id `>` num) id = num; else { id = id `+` num; id = id; }

using the previous grammar is:



Notes on parse trees:

- the parser phase of a compiler organizes tokens into parse trees
- parse trees explicitly show the structure and order of evaluation of language constructs
- there is a one-to-one correspondence between parse trees and leftmost derivations for a string

A grammar is **ambiguous** if there are 2 or more distinct parse trees (or equivalently, leftmost derivations) for the same string.

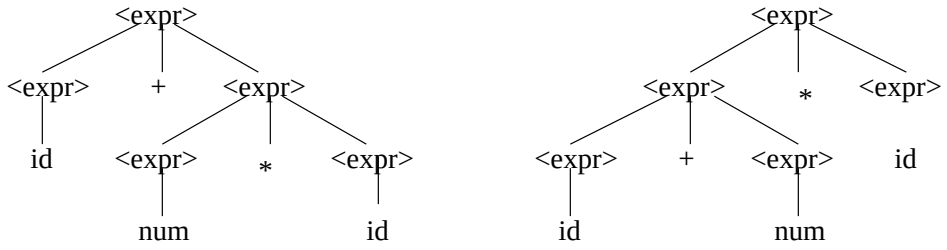
Consider the grammar:

$\langle \text{expr} \rangle \rightarrow \text{id} \mid \text{num} \mid (\langle \text{expr} \rangle) \mid \langle \text{expr} \rangle + \langle \text{expr} \rangle \mid \langle \text{expr} \rangle * \langle \text{expr} \rangle$

and the string:

id + num * id

The following parse trees show that this grammar is ambiguous:



Which parse tree would we prefer?

Grammars can often be modified to remove ambiguity—in this case, by enforcing associativity and precedence:

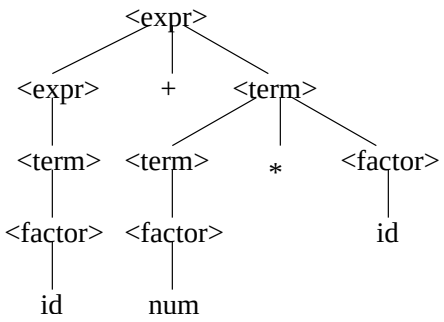
⟨expr⟩ → ⟨expr⟩ + ⟨term⟩ | ⟨term⟩
⟨term⟩ → ⟨term⟩ * ⟨factor⟩ | ⟨factor⟩
⟨factor⟩ → id | num | (⟨expr⟩)

The intuition behind this approach is to try to force the + to occur higher in the parse tree.

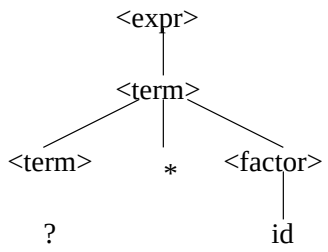
The parse tree for :

id + num * id

is:



Trying to find another tree:



This grammar modification gives rise to three proof obligations:

- the two grammars define the same language
- the second grammar always gives correct associativity and precedence
- the second grammar is not ambiguous

These proofs are omitted.

3.4 Extended BNF (EBNF)

EBNF provides some additional notation, but does not allow any additional languages to be defined (as compared to BNF). Hence, it is only a notational convenience.

notation	meaning
[x]	optional (0 or 1 occurrence of x)
{x}	repetition (0 or more occurrences of x)
(x | y)	choice (x or y)

Table 3: Additional EBNF notation.

Examples:

BNF: ⟨stmt-list⟩ → ϵ | ⟨stmt⟩ ⟨stmt-list⟩
EBNF: ⟨stmt-list⟩ → {⟨stmt⟩}

BNF: ⟨expr⟩ → ⟨expr⟩ `+` ⟨expr⟩ | ⟨expr⟩ `*` ⟨expr⟩
EBNF: ⟨expr⟩ → ⟨expr⟩ (`+` | `*`) ⟨expr⟩

BNF: ⟨expr⟩ → ⟨expr⟩ `+` ⟨term⟩ | ⟨term⟩
EBNF: ⟨expr⟩ → [⟨expr⟩ `+`] ⟨term⟩

4 Static Semantics and Attribute Grammars

Static semantics is any aspect of semantics (meaning) that can be checked at compile time (statically). Examples:

- checking that variables are declared before they are used
- type checking

A more specific example: consider the Java code

```
String x = 5;
```

Question (1): is this an element of the language generated by the Java BNF?
Question (2): is this legal Java code?

As we can see from this example, many language properties can be checked statically, but are impossible or difficult to specify with a BNF definition.

An **attribute grammar** is a grammar with attributes (properties) attached to each symbol, plus rules for computing and checking the values of attributes. Attribute grammars are useful for checking static semantic properties.

If parse trees are implemented with a record or class instance for each node, then the attribute is just an additional field in this representation.

Types of attributes:

- a **synthesized attribute** is an attribute whose value is computed based on the attribute values of child nodes (in a parse tree). Synthesized attributes pass information up the parse tree.
- an **inherited attribute** is an attribute whose value is computed based on the attribute values of parent and sibling nodes (in a parse tree). Inherited attributes pass information down the parse tree.
- an **intrinsic attribute** is an attribute whose value is not based on other attribute values.

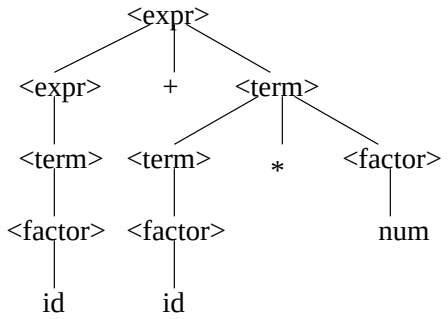
For example, consider the grammar:

⟨expr⟩ → ⟨expr⟩ + ⟨term⟩ | ⟨term⟩
⟨term⟩ → ⟨term⟩ * ⟨factor⟩ | ⟨factor⟩
⟨factor⟩ → id | num | (⟨expr⟩)

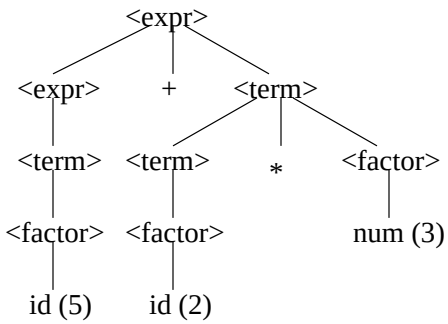
and expression:

x + y * 3

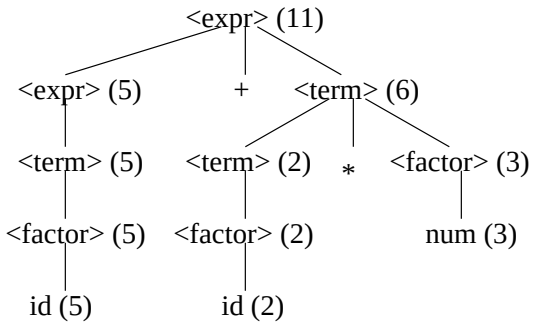
with parse tree:



Consider using an attribute named value to represent the mathematical result of an expression. If the value of x is 5 and y is 2, then the parse tree can be decorated/annotated with these intrinsic attribute values as follows:



Now synthesized attribute values can be computed for each of the nonterminals in the parse tree:



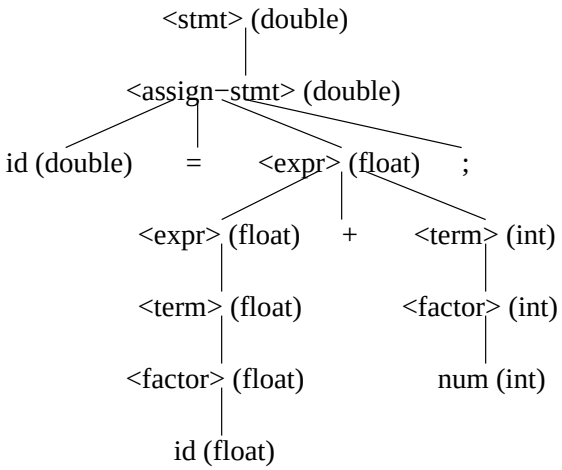
More practically, attributes can be used for type checking by adding a type attribute to each symbol. For example, consider the grammar:

⟨stmt⟩ → ⟨assign-stmt⟩
⟨assign-stmt⟩ → id = ⟨expr⟩ ;
⟨expr⟩ → ⟨expr⟩ + ⟨term⟩ | ⟨term⟩
⟨term⟩ → ⟨term⟩ * ⟨factor⟩ | ⟨factor⟩
⟨factor⟩ → id | num | (⟨expr⟩)

and assignment statement:

x = y + 5;

and suppose that y is of type float and x is of type double. The annotated parse tree is:



This simple example shows only the *actual* types of the nodes in the parse tree, inferred as synthesized attributes. A compiler would also use *expected* types—a completely separate attribute, inferred as an inherited attribute. The textbook example 3.6 goes over this in detail. Once both kinds of types have been inferred (actual and expected), a compiler can check the resulting parse tree to see if the actual types are the same as the expected types. Of course, it will report an error if there are any invalid mismatches. But note that some kinds of mismatches are okay, because many languages allow *coercion* of types: for example, in Java, an int value can be coerced into a float or double without causing a compiler error.

Parser generators such as YACC and CUP generate parsers from attribute grammar specifications of the language being parsed.