Notes on formal grammars

Handout written by Maggie Johnson and revised by me.

What is a grammar?

A *grammar* is a powerful tool for describing and analyzing languages. It is a set of rules by which valid sentences in a language are constructed. Here's a trivial example from the English grammar:

sentence ← **<subject> <verb-phrase> <object>**

Nick is a doofus.
Computers run the world.
I am the cheese.
I never tell lies.

To derive the simple sentences above, we need some other rules or *productions* in the grammar:

subject	←	Nick | Computers | I
verb-phrase	←	**<adverb> <verb>** | **<verb>**
adverb	←	never
verb	←	is | run | am | tell
object	←	the **<noun>** | a **<noun>** | **<noun>**
noun	←	doofus | world | cheese | lies

Here is a *left-most derivation* of the first sentence using these productions.

sentence	←	**<subject> <verb-phrase> <object>**
	←	Nick **<verb-phrase> <object>**
	←	Nick **<verb> <object>**
	←	Nick is **<object>**
	←	Nick is a **<noun>**
	←	Nick is a doofus

Of course, this grammar is clearly inadequate as a description of even the tiniest portion of English. In addition to the reasonable sentences above, we can also derive "I tell cheese", and "Nick run a lies". One important point to keep in mind as we work with grammars is we are most interested in syntax, not semantics. "I tell cheese" is syntactically correct based on the given grammar although semantically, it is incorrect. We worry about semantics at a later point in the compiling process. In the syntax analysis phase, we are only verifying the correctness of the structure.

Vocabulary

We need some definitions before we can proceed:

grammar	a set of rules by which valid sentences in a language are constructed.
nonterminals	the symbols of a grammar that can be replaced by other symbols or actual words
terminals	the actual words of a language; these are the symbols in a grammar that cannot be replaced by anything else. "terminal" is supposed to conjure up the idea that it is a dead-end— no further expansion or parsing is possible from here.
derivation	a sequence of applications of the rules of a grammar that produces a finished string of terminals. A *leftmost derivation* is where we always substitute for the

	leftmost nonterminal as we apply the rules (we can similarly define a rightmost derivation). A derivation is also called a *parse*.
production	another name for a grammar rule. The general form of a production is: $X \rightarrow Y_1Y_2Y_3...Y_n$ This defines a non-terminal X as being made up of the concatenation of non-terminals or terminals $Y_1Y_2Y_3...Y_n$. The production means that anywhere where we encounter X, we may replace it by the string $Y_1Y_2Y_3...Y_n$. Eventually we will have a string containing nothing that can be expanded further, i.e., it will consist of only terminals. Such a string is called a *sentence*. In the context of programming languages, syntactically correct programs are sentences derived from the productions defining the language.
start symbol	in a grammar we have a single entity (the start symbol) from which all syntactically correct programs derive: $S \rightarrow X_1X_2X_3...X_n$ A sentence must be derivable from S by successive replacement using the productions of the grammar.
null symbol ∂	it is sometimes useful to specify that a symbol can be replaced by nothing at all. To indicate this, we use the null symbol ∂, e.g., $A \rightarrow B \mid \partial$.
BNF	a way of specifying programming languages using formal grammars and production rules with a particular form of notation (Backus-Naur form).

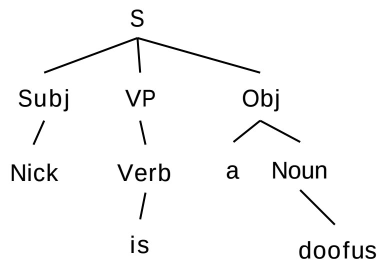
A few grammar exercises to try on your own:

- Define a grammar for the language over the alphabet {a,b} where the number of a's in a string is equal to the number b's in the string.
- Define a grammar where the number of a's is not equal to the number b's. (Hint: think about it as two separate cases...)
- Define a grammar for palindromes in the this alphabet.
- Can you write regular expressions for these languages? Why or why not?

Parse representation

In working with grammars, we can represent the application of the rules to derive a sentence in two ways. The first is a derivation as shown earlier for “Nick is a doofus” where the rules are applied step-by-step and we substitute for one non-terminal at a time. Think of a derivation as a history of how the sentence was parsed because it not only includes which productions were applied, but also the order they were applied (i.e. which non-terminal was chosen for expansion at each step). There can many different derivations for the same sentence (the leftmost, the rightmost, and so on).

A *parse tree* is the second method for representation. It diagrams how each symbol derives from other symbols in a hierarchical manner. Here is a parse tree for the “Nick is a doofus” string:



Although the parse tree includes all of the productions that were applied, it does not encode the order they were applied. For an unambiguous grammar (we'll define ambiguity in a minute), there is exactly one parse tree for a particular sentence.

More formal definitions

Here are some other definitions we will need, described in reference to this example grammar:

```
S    ← AB
A    ← Ax | y
B    ← z
```

alphabet

The alphabet is {S, A, B, x, y, z}. It is divided into two disjoint sets. The *terminal alphabet* consists of terminals, which appear in the sentences of the language: {x, y, z}. The remaining symbols are the *nonterminal alphabet*; these are the symbols that appear on the left side of productions and can be replaced during the course of a derivation: {S, A, B}. Formally, we use **V** for the alphabet, **T** for the terminal alphabet and **N** for the nonterminal alphabet giving us: **V** = **T**   **N**, and **T** ∩ **N** =   .

Our convention is to use a sans-serif font for grammar elements, lowercase for terminals, uppercase for nonterminals, and underlined lowercase (e.g., $\underline{u}$, $\underline{v}$) to denote arbitrary strings of terminal and nonterminal symbols (possibly null). In some textbooks, Greek symbols are used for arbitrary strings of terminal and nonterminal symbols (e.g., α, β)

context-free grammar

To define a language, we need a set of productions. In general terms, these take the form: $\underline{u} \leftarrow \underline{v}$. In a BNF or *context-free grammar*, $\underline{u}$ is a single nonterminal and $\underline{v}$ is an arbitrary string of terminal and nonterminal symbols. When parsing, we can replace $\underline{u}$ by $\underline{v}$ wherever it occurs. We shall refer to this set of productions symbolically as **P.**

formal grammar

We formally define a grammar as a 4-tuple {S, **P, N, T**}. S is the start symbol and S   **N**, **P** is the set of productions, and **N** and **T** are the nonterminal and terminal alphabets. A sentence is a string of symbols in **T** derived from S using one or more applications of productions in **P.** A string of symbols derived from S but possibly including nonterminals is called a *sentential form* or a *working string.*

A production $\underline{u} \leftarrow \underline{v}$ is used by replacing an occurrence of $\underline{u}$ by $\underline{v}$. Formally, if we apply a production p   **P** to a string of symbols $\underline{w}$ in **V** to yield a new string of symbols $\underline{z}$ in **V**, we say that $\underline{z}$ derived from $\underline{w}$ using p, written as follows: $\underline{w} \Rightarrow^{p} \underline{z}$. We also use:

$\underline{w} \Rightarrow \underline{z}$	$\underline{z}$ derives from $\underline{w}$ (production unspecified)
$\underline{w} \Rightarrow^{*} \underline{z}$	$\underline{z}$ derives from $\underline{w}$ using zero or more productions
$\underline{w} \Rightarrow^{+} \underline{z}$	$\underline{z}$ derives from $\underline{w}$ using one or more productions

equivalence

The language L(G) defined by grammar G is the set of sentences derivable using G. Two grammars G and G' are said to be *equivalent* if the languages they generate L(G) and L(G') are the same.

A hierarchy of grammars

We owe a lot of our understanding of grammars to the work of the American linguist Noam Chomsky (yes, the Noam Chomsky known for his politics). There are four categories of formal grammars in the *Chomsky Hierarchy*, they span from Type 0, the most general, to Type 3, the most restrictive. More restrictions on the grammar make it easier to describe and efficiently parse, but reduce the expressive power.

Type 0: free or unrestricted grammars
These are the most general. Productions are of the form $\underline{u} \leftarrow \underline{v}$ where both $\underline{u}$ and $\underline{v}$ are arbitrary strings of symbols in **V**, with $\underline{u}$ non-null. There are no restrictions on what appears on the left or right-hand side other than the left-hand side must be non-empty.

Type 1: context-sensitive grammars
Productions are of the form $\underline{u}X\underline{w} \leftarrow \underline{uvw}$ where $\underline{u}$, $\underline{v}$ and $\underline{w}$ are arbitrary strings of symbols in **V**, with $\underline{v}$ non-null, and `X` a single nonterminal. In other words, `X` may be replaced by $\underline{v}$ but only when it is surrounded by $\underline{u}$ and $\underline{w}$. (i.e. in a particular context).

Type 2: context-free grammars
Productions are of the form $X \leftarrow \underline{v}$ where $\underline{v}$ is an arbitrary string of symbols in **V**, and `X` is a single nonterminal. Wherever you find `X`, you can replace with $\underline{v}$ (regardless of context).

Type 3: regular grammars
Productions are of the form $X \leftarrow a$ or $X \leftarrow aY$ where `X` and `Y` are nonterminals and `a` is a terminal. That is the left-hand side must be a single nonterminal and the right-hand side can be either a single terminal by itself or with a single nonterminal. These grammars are the most limited in terms of expressive power.

Every type 3 grammar is a type 2 grammar, and every type 2 is a type 1 and so on. Type 3 grammars are particularly easy to parse because of the lack of recursive constructs. Efficient parsers exist for many classes of Type 2 grammars. Although Type 1 and Type 0 grammars are more powerful than Type 2 and 3, they are far less useful since we cannot create efficient parsers for them. In designing programming languages using formal grammars, we will use Type 2 or context-free grammars, often just abbreviated as CFG.

Issues in parsing context-free grammars

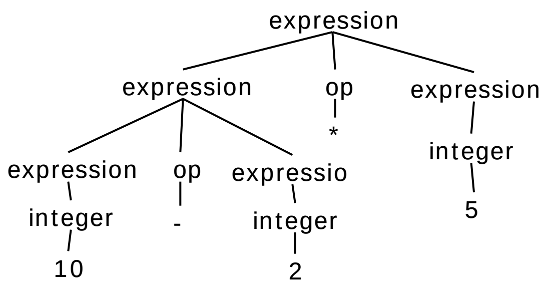
There are several workable and efficient approaches to parsing most Type 2 grammars and we will talk through them over the next few lectures. However, there are some issues that can interfere with efficient parsing that we must take into consideration when designing the grammar. Let's take a look at three of them: ambiguity, recursive rules, and left-factoring.

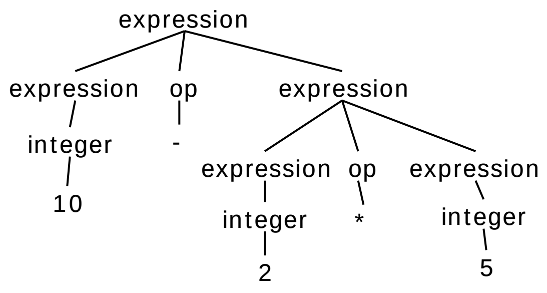
Ambiguity

If a grammar permits more than one parse tree for some sentences, it is said to be *ambiguous*. For example, consider the following classic arithmetic expression grammar:

```
expression      ←  expression op expression | ( expression ) | integer
op              ←  + | - | * | /
```

This grammar denotes expressions that consist of integers joined by binary operators and possibly including parentheses. As defined above, this grammar is ambiguous because for certain sentences we can construct more than one parse tree. For example, consider the expression `10 – 2 * 5`. We parse by first applying the production `expression ← expression op expression`. The parse tree on the left chooses to expand that first `op` to `*`, the one on the right to `-`. We have two completely different parse trees. Which one is correct?





Both trees are legal in the grammar as stated and thus either interpretation is valid. Although natural languages can tolerate some kind of ambiguity (puns, plays on words, etc.), it is not acceptable in computer languages. We don't want the compiler just haphazardly deciding which way to interpret our expressions! Givem our expectations from algebra concerning precedence, only one of the trees seems right. The right-hand tree fits our expectation that * "binds tighter" and for that result to be computed first then integrated in the outer expression which has a lower precedence operator.

It's actually rather easy for a grammar to become ambiguous if you are not careful in its construction. Unfortunately, there is no magical technique that can be used to resolve all varieties of ambiguity. One common means to handle it is to introduce some disambiguating rules into the parser implementation. For the above example, we could code into the parser knowledge of precedence and associativity to break the tie. The advantage of this is that the grammar remains simple and less complicated. But as a downside, the syntactic structure of the language is no longer given by the grammar alone.

Another common approach is to change the grammar to only allow the one tree that correctly reflects our intention and eliminate the others. For the expression grammar, we can separate expressions into multiplicative and additive subgroups and force them to be expanded in the desired order.

```
expression      ←   expression term_op expression | term
term_op         ←   + | -
term            ←   term factor_op term | factor
factor_op       ←   * | /
factor          ←   (expression) | integer
```

Terms are addition/subtraction expressions and factors used for multiplication and division. Since the base case for expression is a term, addition and subtraction will appear higher in the parse tree, and thus receive lower precedence.

After verifying that the above re-written grammar has only one parse tree for the earlier ambiguous expression, you might thing we were home-free, but now consider the expression `10 –2 –5`. The recursion on both sides of the binary operator allows either side to match repetitions. The arithmetic operators usually associate to the left, so by replacing the right-hand side with the base case will force the repetitive matches onto the left side. The final result is:

```
expression      ←   expression term_op term | term
term_op         ←   + | -
term            ←   term factor_op factor | factor
factor_op       ←   * | /
factor          ←   (expression) | integer
```

Whew! The obvious disadvantage of changing the grammar to remove ambiguity is that it may complicate and obscure the original grammar definitions. There is no mechanical means to change any ambiguous grammar into an unambiguous one—it is known to be an undecidable problem (in fact, even determining that a CFG is ambiguous is undecidable). However, most programming languages have only limited issues with ambiguity that can be resolved using ad hoc techniques.

Recursive productions

Productions are often defined in terms of themselves. For example a list of variables in a programming language grammar could be specified by this production:

```
variable_list      ← variable | variable_list , variable
```

Such productions are said to be *recursive.* If the recursive nonterminal is at the leftmost end of a right-side production, we call the production *left-recursive*. For example: A ← $\underline{u}$ | A$\underline{v}$. Similarly, we can define a *right-recursive* production: A ← $\underline{u}$ | $\underline{v}$A. Some parsing techniques have trouble with one or the other variants of recursive productions. Left-recursive productions can be especially troublesome in the parsers we build for compilers (we'll see why a bit later). Handily, there is a simple technique for removing left-recursion from a grammar. The basic idea is to rewrite it to move the recursion to the other side. For example, consider this left-recursive rule:

```
X      ← Xa | Xb | AB | C | DEF
```

To convert the rule, we introduce a new nonterminal X' that we append to the end of all non-left-recursive productions for X. The expansion for the new nonterminal is basically the reverse of the original left-recursive rule. The re-written productions are:

```
X      ← ABX'  | CX' | DEFX'
X'     ← aX' | bX' | ∂
```

It appears we just exchanged the left-recursive rules for an equivalent right-recursive version. This might seem pointless, but some parsing algorithms prefer or even require only left or right recursion.

Left-factoring

The parser usually reads tokens from left to right and it is convenient if upon reading a token it can make an immediate decision about which production from the grammar to expand. However, this can be trouble if there are productions that have common first symbol(s) on the right side of the productions. Here is an example we often see in programming language grammars:

```
Stmt        ← if Cond then Stmt else Stmt  | if Cond then Stmt |  Other   | ....
```

The common prefix is if Cond then Stmt. This causes problems because when a parser encounter an "if", it does not know which production to use. A useful technique called *left-factoring* allows us to restructure the grammar care of this problem. The basic idea is to rewrite the productions to defer the decision about which of the options to choose until we have seen enough of the input to make the right choice. We factor out the common part of the two options into a shared rule that both will use and then add a new rule that picks up where the tokens diverge.

```
Stmt        ← if Cond then Stmt OptElse  | Other  | ...
OptElse     ← else  S  |  ∂
```

In the re-written grammar, upon reading an "if" we expand first production and wait until if Cond then Stmt has been seen to decide whether to expand OptElse to else or ∂.

SOOP grammar

Here is a grammar for the SOOP language used in the programming projects.

```
Program                ->           ( Declaration )*

Declaration            ->           ClassDefinition
                       ->           ClassDeclaration
                       ->           FunctionDefinition
                       ->           FunctionDeclaration
                       ->           VariableDeclaration

VariableDeclaration    ->           Variable ';'

Variable               ->           Type Identifier ArrayModifiers

ArrayModifiers         ->           ( '[' IntConstant ']' )*

Type                   ->           int
                       ->           void
                       ->           bool
                       ->           double
                       ->           class Identifier

ClassDeclaration       ->           class Identifier ';'

ClassDefinition        ->           class Identifier (':' Identifier)?
                                      '{'  ClassDefinitionList '}'

ClassDefinitionList    ->           ( ClassDefinitionList
                                      (public|private)? ClassField )*

ClassField             ->           InstanceVariableDeclaration
                       ->           FunctionDefinition
                       ->           FunctionDeclaration

InstanceVariableDeclaration ->  Variable ';'

FunctionDeclaration    ->           Type Identifier
                                      '(' ParameterListOrVoid ')' ';'

ParameterListOrVoid    ->           Void
                       ->           ParameterList

ParameterList          ->           ( ParameterList ',' )* Parameter

Parameter              ->           Variable

FunctionDefinition     ->           Type Identifier '(' ParameterListOrVoid
')'
                                      CompoundStatement

CompoundStatement      ->           '{' ( VariableDeclaration )*
                                       ( Statement )* '}'

Statement              ->           SimpleStatement  ';'
                       ->           IfStatement
                       ->           WhileStatement
                       ->           ReturnStatement  ';'
                       ->           CompoundStatement
```

```
                      ->            PrintStatement ';'

SimpleStatement       ->            Designator '=' Expression
                      ->            Identifier '=' Expression
                      ->            Expression

Designator            ->            Expression ( '.' Identifier |
                                                 '[' Expression ']' )

Call                  ->            ( Expression '.' )? Identifier
                                      '(' ArgumentList ')'

Expression            ->            Designator
                      ->            Call
                      ->            Identifier
                      ->            Constant
                      ->            Expression '+' Expression
                      ->            Expression '-' Expression
                      ->            Expression '/' Expression
                      ->            Expression '*' Expression
                      ->            Expression '%' Expression
                      ->            Expression '==' Expression
                      ->            Expression '!=' Expression
                      ->            Expression '<' Expression
                      ->            Expression '>' Expression
                      ->            Expression '<=' Expression
                      ->            Expression '>=' Expression
                      ->            Expression '&&' Expression
                      ->            Expression '||' Expression
                      ->            '(' Expression ')'
                      ->            '-' Expression
                      ->            '!' Expression
                      ->            ReadInteger '(' ')'
                      ->            New '(' Identifier ')'
                      ->            NewArray '(' Expression ')'

Constant              ->            IntConstant
                      ->            BoolConstant
                      ->            DoubleConstant
                      ->            StringConstant
                      ->            Null

ArgumentList          ->            (Expression ',' )* Expression
                      ->

BooleanExpression     ->            Expression

WhileStatement        ->            while '(' BooleanExpression ')' Statement

IfStatement           ->            if '(' BooleanExpression ')' Statement
                                        ( else Statement )?

ReturnStatement       ->            Return ( Expression )?

PrintStatement        ->            Print '(' (Expression ',' )* Expression
')'
```

This grammar is not perfect. Can you find some things this grammar allows which should not be part of a valid program? As we will see, it is really hard to write a grammar for a programming language that is all-inclusive, and keeps out the things we don't want. Consequently, we end up doing some special processing in the semantic analysis phase to get rid of the unwanted constructs.

To give you and idea of how the grammar works, here is a simple derivation. You have to read it backwards, however, because of the parsing technique being used (more on this later).

```
void test(void) {
   int a;
   a = 12;
}
```

```
Type                    ->  Void
ParameterListOrVoid     ->  Void
Type                    ->  Int
ArrayModifiers          ->  ∂
Variable                ->  Type Identifier ArrayModifiers
VariableDeclaration     ->  Variable
Constant                ->  IntConstant
Expression              ->  Constant
SimpleStatement         ->  Identifier = Expression
Statement               ->  SimpleStatement ;
CompoundStatement       ->  { VariableDeclaration Statement }
FunctionDefinition      ->  Type Identifier ( ParameterListOrVoid )
CompoundStatement
Declaration             ->  FunctionDefinition
Program                 ->  Declaration
```

Programming Language Case Study: ALGOL

Algol is of interest to us because it was the first programming language to be defined using a grammar. It grew out of an international effort in the late 1950's to create a "universal programming language" that would run on all machines. At that time, FORTRAN and COBOL were the prominent languages, with new languages sprouting up all over the place. Programmers became increasingly concerned about portability of programs, and being able to communicate with one another on programming topics.

Consequently the ACM and GAMM (Gesellschaft für angewandte Mathematik und Mechanik) decided to come up with a single programming language that all could use on their computers, and in whose terms, programs could be communicated between the users of all machines. Their first decision was not to use FORTRAN as their universal language. This may seem surprising to us today, since it was the most commonly used language back then. However, as Alan J. Perlis, one of the original committee members, puts it:

> "Today, FORTRAN is the property of the computing world, but in 1957, it was an IBM creation and closely tied to IBM hardware. For these reasons, FORTRAN was unacceptable as a universal language."

ALGOL-58 was the first version of the language, followed up very soon after by ALGOL-60, which is the version which had the most impact. As a language, it introduced the following features:

- block structure and nested structures
- strong typing
- scoping

- procedures and functions
- call by value, call by reference
- side effects (is this good or bad?)
- recursion

It may seem surprising that recursion was not implemented in FORTRAN or COBOL. You probably know that to implement recursion we need a runtime stack to store the activation records as functions are called. In FORTRAN and COBOL, activation records were created at compile time, not runtime. Thus, only one activation record per sub-routine was created. No stack was used. The parameters for the sub-routine were copied into the activation record and that data area was used for sub-routine processing.

The ALGOL report was the first time we see BNF to describe a programming language. Both John Backus and Peter Naur were on the ALGOL committees. They derived this description technique from an earlier paper written by Backus. The technique was adopted because they needed a machine-independent method of description. If one looks at the early definitions of FORTRAN, one can see the links to the IBM hardware. With ALGOL, the machine was not relevant. BNF had a huge impact on programming language design and compiler construction. First, it stimulated a large number of studies on the formal structure of programming languages laying the groundwork for a theoretical approach to language design. Second, a formal syntactic description could be used to drive a compiler directly (as we shall see).

ALGOL was a commercial failure but as we have seen, it had a tremendous impact on programming language design, compiler construction and language theory. IBM was very powerful in those days, and the fact that they never bought into ALGOL really hurt its chances. There were, however, problems with the language too. For example, there were no IO statements provided in the language. This seems like a major flaw but the designers thought providing IO would tie the language to a particular machine. Their idea was a library of IO routines was to be provided which were specific to each machine, but no one every wrote them.

Bibliography

A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools. Reading, MA: Addison-Wesley, 1986.

J. Backus, “The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference,” Proceedings of the International Conference on Information Processing, 1959, pp. 125-132.

N. Chomsky, “On Certain Formal Properties of Grammars,” Information and Control, Vol. 2, 1959, pp. 137-167.

J.P. Bennett, Introduction to Compiling Techniques. Berkshire, England: McGraw-Hill, 1990.

D. Cohen, Introduction to Computer Theory. New York: Wiley, 1986.

J.C. Martin, Introduction to Languages and the Theory of Computation. New York, NY: McGraw-Hill, 1991.

P. Naur, “Programming Languages, Natural Languages, and Mathematics,” Communications of the ACM, Vol 18, No. 12, 1975, pp. 676-683.

J. Sammet, Programming Languages: History and Fundamentals. Englewood-Cliffs, NJ: Prentice-Hall, 1969.

R.L.Wexelblat, History of Programming Languages. London: Academic Press, 1981.