CS143 Handout 07
Summer 2009 June 30, 2009

Formal Grammars

Handout written by Maggie Johnson and Julie Zelenski.

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 of English grammar:

```
sentence      ->   <subject> <verb-phrase> <object>
subject       ->   This | Computers | I
verb-phrase   ->   <adverb> <verb> | <verb>
adverb        ->   never
verb          ->   is | run | am | tell
object        ->   the <noun> | a <noun> | <noun>
noun          ->   university | world | cheese | lies
```

Using the above rules or *productions*, we can derive simple sentences such as these:

```
This is a university.
Computers run the world.
I am the cheese.
I never tell lies.
```

Here is a *leftmost derivation* of the first sentence using these productions.

```
sentence      ->   <subject> <verb-phrase> <object>
              ->   This <verb-phrase> <object>
              ->   This <verb> <object>
              ->   This is <object>
              ->   This is a <noun>
              ->   This is a university
```

In addition to several reasonable sentences, we can also derive nonsense like "Computers run cheese" and "This am a lies". These sentences don't make semantic sense, but they are syntactically correct because they are of the sequence of subject, verb-phrase, and object. Formal grammars are a tool for syntax, not semantics. We worry about semantics at a later point in the compiling process. In the syntax analysis phase, we verify structure, not meaning.

Vocabulary

We need to review some definitions before we can proceed:

grammar a set of rules by which valid sentences in a language are constructed.

nonterminal a grammar symbol that can be replaced/expanded to a sequence of symbols.

terminal an actual word in 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 is possible.

production a grammar rule that describes how to replace/exchange symbols. The general form of a production for a nonterminal is:

$$X \rightarrow Y_1 Y_2 Y_3 \ldots Y_n$$

The nonterminal X is declared equivalent to the concatenation of the symbols $Y_1Y_2Y_3 \ldots Y_n$. The production means that anywhere where we encounter X, we may replace it by the string $Y_1Y_2Y_3 \ldots Y_n$. 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, a sentence is a syntactically correct and complete program.

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*.

start symbol a grammar has a single nonterminal (the start symbol) from which all sentences derive:

$$S \rightarrow X_1 X_2 X_3 \ldots X_n$$

All sentences are derived from S by successive replacement using the productions of the grammar.

null symbol ε it is sometimes useful to specify that a symbol can be replaced by nothing at all. To indicate this, we use the null symbol ε, e.g., $A \rightarrow B \mid \varepsilon$.

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 (The alphabet in each case is {a,b}.)

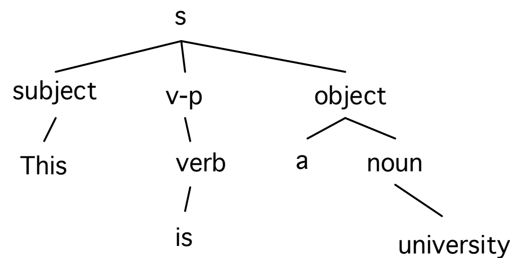
- Define a grammar for the language of strings with one or more a's followed by zero or more b's.
- Define a grammar for even-length palindromes.
- Define a grammar for strings where the number of a's is equal to the number b's.
- Define a grammar where the number of a's is not equal to the number b's. (Hint: think about it as two separate cases...)

(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 "This is a university" where the rules are applied step-by-step and we substitute for one nonterminal 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 nonterminal 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 "This is a university":



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 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: $\mathbf{V} = \mathbf{T} \cup \mathbf{N}$, and $\mathbf{T} \cap \mathbf{N} = \varnothing$.

The convention used in our lecture notes are 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 letters 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, of the general form: $\underline{u}$ -> $\underline{v}$. In a *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 $\in$ **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 $\underline{u}$ -> $\underline{v}$ is used to replace an occurrence of $\underline{u}$ by $\underline{v}$. Formally, if we apply a production p $\in$ **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: $\underline{w} =>^{p} \underline{z}$. We also use:

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

Grammar Hiearchy

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 $\underline{u}$ -> $\underline{v}$ 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 uXw –> uvw where u, v and w are arbitrary strings of symbols in **V**, with v non-null, and X a single nonterminal. In other words, X may be replaced by v but only when it is surrounded by u and w. (i.e., in a particular context).

Type 2: context-free grammars

Productions are of the form X–> v where v is an arbitrary string of symbols in **V**, and X is a single nonterminal. Wherever you find X, you can replace with v (regardless of context).

Type 3: regular grammars

Productions are of the form X–> a, X–> aY, or X–>ε 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 empty, 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

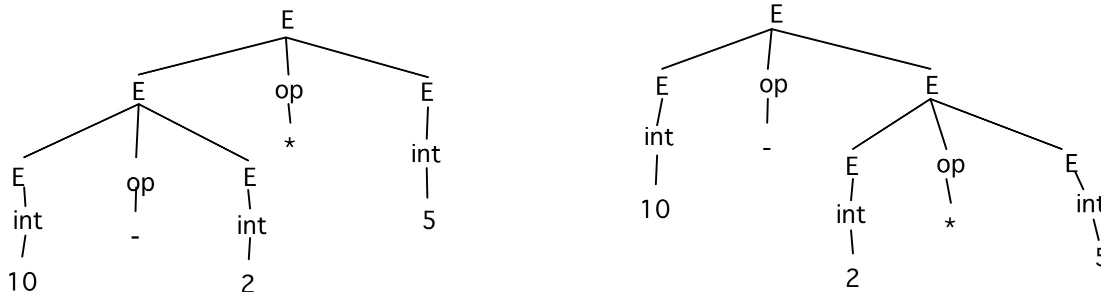
There are several efficient approaches to parsing most Type 2 grammars and we will talk through them over the next few lectures. However, there are some issues that can interfere with parsing that we must take into consideration when designing the grammar. Let's take a look at three of them: ambiguity,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

```
E      –>   E op E | ( E ) | int
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 `E –> E op E`. 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 (e.g., 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! Given 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 fairly 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. It is an undecidable problem to determine whether any grammar is ambiguous, much less to attempt to mechanically remove all ambiguity. However, that doesn't mean in practice that we cannot detect ambiguity or do something about it. For programming language grammars, we usually take pains to construct an unambiguous grammar or introduce additional disambiguating rules to throw away the undesirable parse trees, leaving only one for each sentence.

Using the above ambiguous expression grammar, one technique would leave the grammar as is, but add disambiguating rules into the parser implementation. We could code into the parser knowledge of precedence and associativity to break the tie and force the parser to build the tree on the right rather than the left. 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 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.

```
E        ->   E t_op E | T
t_op     ->   + | -
T        ->   T f_op T | F
f_op     ->   * | /
F        ->   (E) | int
```

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:

```
E        ->   E t_op T | T
t_op     ->   + | -
T        ->   T f_op F | F
f_op     ->   * | /
F        ->   (E) | int
```

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 (undecidable, remember?) 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 left of the right-side of the production, e.g. A -> u | Av, we call the production *left-recursive*. Similarly, we can define a *right-recursive* production: A -> u | vA. Some parsing techniques have trouble with one or the other variants of recursive productions and so sometimes we have to massage the grammar into a different but equivalent form. Left-

recursive productions can be especially troublesome in the top-down parsers (and we'll see why a bit later). Handily, there is a simple technique for rewriting the grammar 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' | ε
```

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 to avoid this situation. We 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 appropriate 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  |  ε
```

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 ε.

Hidden left-factors and hidden left recursion

A grammar may not appear to have left recursion or left factors, yet still have issues that will interfere with parsing. This may be because the issues are hidden and need to be first exposed via substitution.

For example, consider this grammar:

```
A    ->   da | acB
B    ->   abB | daA | Af
```

A cursory examination of the grammar may not detect that the first and second productions of B overlap with the third. We substitute the expansions for A into the third production to expose this:

```
A    ->   da | acB
B    ->   abB | daA | daf | acBf
```

This exchanges the original third production of B for several new productions, one for each of the productions for A. These directly show the overlap, and we can then left-factor:

```
A    ->   da | acB
B    ->   aM | daN
M    ->   bB | cBf
N    ->   A | f
```

Similarly, the following grammar does not appear to have any left-recursion:

```
S    ->   Tu | wx
T    ->   Sq | vvS
```

Yet after substitution of S into T, the left-recursion comes to light:

```
S    ->   Tu | wx
T    ->   Tuq | wxq | vvS
```

If we then eliminate left-recursion, we get:

```
S    ->   Tu | wx
T    ->   wxqT' | vvST'
T'   ->   uqT' | ε
```

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 around. 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 that 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 present in the original 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 subroutine was created. No stack was used. The parameters for the subroutine were copied into the activation record and that data area was used for subroutine 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 had a tremendous impact on programming language design, compiler construction, and language theory, but the language itself was a commercial failure.

Partly this was due to design decisions (overly complex features, no IO) along with the politics of the time (popularity of Fortran, lack of support from the all-powerful IBM, resistance to BNF).

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.