UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS 164 **P. N. Hilfinger**
Spring 2005

CS 164: Programming Languages and Compilers
Syntactic Analysis, Part I

1 Recap

The purpose of *syntactic analysis* is to analyze textual input so as to confirm that it is *syntactically well-formed*—that it obeys certain general structural rules dictated by the specification of the input language—and to convert it into a form that gives later parts of the compiler convenient access to this structure.

For example, we might say that in Java a conditional statement can have the form

if (*Expression*) *Statement* **else** *Statement*

In later parts of the compiler, the programmer might reasonably want a data structure that represents "an **if** statement" and that provides operations that return "the **then** clause," "the **else** clause," and "the test" from this statement. These operations would be awkward to implement if the data structure used were simply a copy of the original text of the statement (a string). Instead, a tree-like form is a better representation. This requires analyzing the original text into its constituent grammatical parts.

This task is traditionally broken down into *lexical analysis*— which breaks the text down into the smallest useful atomic units, known as *tokens*, while throwing away extraneous information, such as white space and comments—and *parsing*—which operates on tokens and groups them into useful grammatical structures. There is no sharp distinction between these two activities—I am happy to classify both under "syntactic analysis." A single monolithic subprogram could handle both simultaneously, as was done in very early compilers.

To a certain extent, we divide the tasks as we do to accommodate certain techniques and certain automatic or semi-automatic tools. For example, the parsing techniques we'll use in this class are designed to decide on what to do next on the basis of the next token of input. If tokens are single characters, they won't have enough information to decide. For example, suppose a program has seen the characters '`x+y`' and the next character is a blank. This is insufficient information to determine whether '`x+y`' is to be treated as a subexpression, since if the next non-blank character is '`*`', then `y` should be grouped with whatever is after the

asterisk. The lexer, on the other hand, can first eliminate whitespace, making the decision easier. Another example is '`x+y`' followed by a '`+`'. Here, the decision depends on whether the character immediately after the '`+`' is another '`+`'. If the lexer has previously grouped all '`++`'s into single tokens, the decision is easily made, with no *ad hoc* scanning ahead in special cases.

In previous lectures, we looked at the use of finite-state automata to recognize and classify significant atomic clumps (tokens) of source text. In brief outline,

- *Regular expressions* can describe a variety of languages (sets of strings), including the set of atomic symbols of a typical programming language.
- *Finite-state automata* (FSAs) are abstract machines that also recognize languages.
- *Deterministic finite-state automata* (DFAs) are a subset of finite-state automata that are easily converted into programs.
- There exists a translation from regular expressions into FSAs.
- There exists a translation from FSAs that happen to be nondeterministic into DFAs (and hence into programs).
- There exist techniques for improving DFAs by eliminating redundant states.
- The total process of conversion from regular expression to program is automatable.

Now we turn to the rest of the process.

2 We're not there yet

Regular expressions describe languages, but they have their limitations. The set of languages describable by a regular expression is the same as the set of languages recognizable by a NFA (or DFA), and these are easily seen to be limited. Intuitively, no NFA can correctly recognize a language that requires counting up to arbitrarily large numbers. For example, the language consisting of all strings of equal numbers of 0s and 1s can't be recognized by any FSA. The reason is simple. Consider the strings '01', '0011', etc. After scanning the 0's at the beginning of '01', the machine must be in than 0s." After scanning the 0's at the beginning of '0011', it must be in "the state that will accept exactly two more 1s than 0s." Obviously, all these states are different (they correspond to entirely different sets of strings), and there are an infinite number of such possible states. But finite-state automata must have finite numbers of states, so no such machine exists.

Another problem is that our regular expressions describe entire strings (sentences, programs, etc.) of a language well enough, but there is no obvious way to recover any *inner structure* from the mere fact that a regular expression has matched a language.

3 Production Rules

We can address these problems by moving to a different way of describing languages: using systems of rules that allow us to name constituent parts of a string. For example, here is a description of floating-point literals in a Java-like language:

```
digit →  '0'
digit →  '1'
digit →  '2'
   ⋮
digit →  '9'
int →  digit
int →  digit int
sign →  '+'
sign →  '-'
sign →
exponent →  'e' sign int
exponent →
literal →  sign int '.'  int exponent
```

Each line '$A \rightarrow \ \alpha_0 \cdots \alpha_n$' can be read as "an A may be formed from an α_0 followed by an α_1,..., followed by an α_n." When n is 0, there are no right-hand side symbols (see the last definitions of `sign` and `expon`, for example), so that we have "A may be formed from the empty string" in such cases. The symbols that are defined to the left of some arrow are called *nonterminal symbols* (or *nonterminals* or *metavariables*) and the other symbols are called *terminal symbols* (or *terminals*). Each nonterminal symbol thus stands for a language; we typically single out one of them—called the *start symbol*—to be the principal language described by the rules (in this case, the start symbol would be `literal`). We call an entire collection of rules a *grammar*.

To save vertical space, we often use a simple shorthand for multiple rules with the same left-hand side:

```
digit →  '0' | '1' | '2' | ...  | '9'
int →  digit | digit int
sign →  '+' | '-' |
expon →  'e' sign int |
literal →  sign int '.'  int expon
```

and I will often use ϵ to make empty right-hand sides more visible, as in

```
sign →  '+' | '-' | ϵ
```

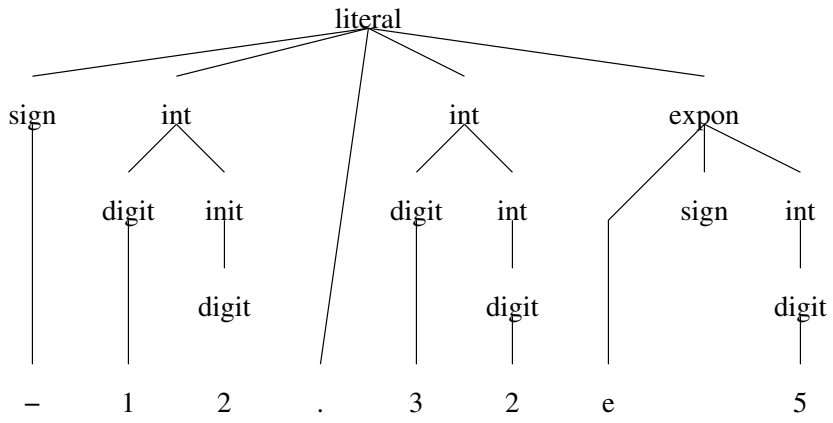

Figure 1: Parse tree for '-12.32e5'

Other variations are possible. For example, the BISON and YACC tools use ':' in place of '→', whereas the Algol 60 report used '::='. Likewise, BISON and YACC terminate each rule with (unquoted) semicolons.

These rules make it convenient to talk about "the sign part" of the literal, or "the exponent part." Given a string, say '-12.32e5', we can say the string is in the language described by "literal" because we can break it down according to the rule for "literal" like this:

$$\overbrace{-}^{\text{sign}}\ \overbrace{12}^{\text{int}}\ \overbrace{.}^{\text{'.'}}\ \overbrace{32}^{\text{int}}\ \overbrace{e5}^{\text{expon}}$$

and each of these five pieces matches either a matching terminal symbol or some right-hand side of the indicted nonterminal. For example,

$$\overbrace{e}^{\text{'e'}}\ \overbrace{}^{\text{sign}}\ \overbrace{5}^{\text{int}}$$

By repeating this process, we get the tree structure shown in Fig. 1, which matches the entire string and shows the entire breakdown. The process of performing this matching of rules with terminal symbols is called *parsing*. The resulting tree is called a *parse tree*. The leaves of this tree are nonterminals that produce the empty string and terminal symbols.

4 Derivations

Most compilers don't really deal with parses in the form of parse trees, but are more concerned with what rules were applied. The information content is actually exactly the same. Suppose we perform a preorder traversal of the parse tree in Fig. 1 and for each nonterminal we visit, print out the rule that was used to produce its children. We get the following list:

```
1.  literal →  sign int '.'  int expon
```

```
2.  sign →  '-'            9.   int →  digit
3.  int →  digit int       10.  digit →  '2'
4.  digit →  '1'           11.  expon →  'e' sign int
5.  int →  digit           12.  sign →
6.  digit →  '2'           13.  int →  digit
7.  int →  digit int       14.  digit →  '5'
8.  digit →  '3'
```

To see how much information is present here, let's go back the other way, and re-create the tree from this list of applied rules. The top node is the start symbol, 'literal'. We now apply step (1), which tells us that the children of the root node are instances of 'sign', 'int', the terminal symbol '.', 'int', and then 'expon'. Since we did a preorder traversal to get this tree, we know that step (2) has to apply to the leftmost unprocesses nonterminal, an instance of 'sign', telling us that its child is the terminal symbol '-'. Step (1) produced two instances of 'int', but since we know that we got this list of rules from a preorder traversal, we know that step (3) has to apply to the leftmost of those instances, giving children 'digit' and 'int'. If you continue this process, you should see that we end up with Fig. 1.

The list of productions 1–14 defines a *leftmost derivation* of the string '-12.32e5' from the grammar. The term "leftmost" refers to the fact that each step applies to the leftmost nonterminal instance in the partially completed tree that has not yet been assigned a rule. Another way of presenting the same derivation is as a sequence of *sentential forms:*

```
literal ⇒  sign int .  int expon ⇒  - int .  int expon
⇒  - digit int .  int expon ⇒  - 1 int .  int expon
⇒  - 1 digit .  int expon ⇒  - 1 2 .  int expon
⇒  - 1 2 .  digit int expon ⇒  - 1 2 .  3 int expon
⇒  - 1 2 .  3 digit expon ⇒  - 1 2 .  3 2 expon
⇒  - 1 2 .  3 2 e sign int ⇒  - 1 2 .  3 2 int
⇒  - 1 2 .  3 2 e digit ⇒  - 1 2 .  3 2 e 5
```

A sentential form, in other words, is just a sequence of symbols, both terminals and nonterminals. The '⇒ ' symbol may be read as "derives in one step." We keep going until there are no nonterminals left (a sentential form containing no nonterminals is also called a *sentence*). The symbol '$\stackrel{*}{\Rightarrow}$' means "derives in 0 or more steps", and '$\stackrel{+}{\Rightarrow}$' means "derives in 1 or more steps." Thus 'literal' $\stackrel{*}{\Rightarrow}$ '-12.32e5' and 'literal' $\stackrel{+}{\Rightarrow}$ '-12.32e5' and 'sign int . int expon' $\stackrel{+}{\Rightarrow}$ '-1 digit . int expon'.

As long as we are consistent in how we apply rules, we can always effect this transfer between parse trees and derivations. For example, if we perform a preorder tree traversal of the parse tree, except that we visit children from right to left, rather than left to right, we get what is called a *rightmost derivation*. For the example above, we would apply the same rules the same number of times, but in a different order:

```
1, 11, 13, 14, 12, 7, 9, 10, 8, 3, 5, 6, 4, 2
```

If this looks a little strange, consider the sequence in reverse (appropriately called a *reverse rightmost* or *canonical* derivation, which goes backwards, "unapplying" each production in turn.) The first rule in the reversed sequence handles the '-' at the *beginning* of the string. The next rule (step 4 in the leftmost derivation) handles the '1' digit—the second character of the string. The reverse rightmost derivation reconstructs the parse tree from the bottom up and from left to right, whereas the leftmost (forward) derivation constructs it top down from left to right.

In principle, all kinds of other derivations are possible, but we will be chiefly interested in the leftmost and (in its reverse form) the rightmost derivation. Furthermore, although technically the term "derivation" refers to a sequence of sentential forms, we will use both these and sequences of productions (under a fixed, specified derivation order) interchangeably.

5 Ambiguity

It may not be obvious, but the grammar for floating-point literals that we've been using has the property that there is a unique parse tree that matches any valid string. This isn't true of all grammars. For example, suppose we tried to describe a slightly different kind of floating-point literal, in which either the integer part or the fraction part, but not both may be empty. One way to do so would be to replace the definitions of 'int' and 'literal' as follows:

```
optint →  int optint |
int →  digit optint
literal2 →  sign optint '.'  int expon | sign int '.'  optint expon
```

(I use 'literal2' to distinguish this language from the preceding). But now there are two ways to match the string '1.2': either the '1' or the '2' can be an 'optint'. These two choices correspond to the two different parse trees shown in Fig. 2, with two different corresponding leftmost derivations:

```
(a) literal2 ⇒  sign optint .  int expon ⇒  optint .  int expon
             ⇒  digit optint .  int expon ⇒  1 optint .  int expon
             ⇒  1 .  int expon ⇒  1 .  digit optint expon
             ⇒  1 .  2 optint expon ⇒  1 .  2 expon ⇒  1 .  2
(b) literal2 ⇒  sign int .  optint expon ⇒  optint .  int expon
             ⇒  digit optint .  optint expon ⇒  1 optint .  optint expon
             ⇒  1 .  optint expon ⇒  1 .  digit optint expon
             ⇒  1 .  2 optint expon ⇒  1 .  2 expon ⇒  1 .  2
```

We say that the grammar is *ambiguous:* there exists at least one string for which there are two parses. For every parse tree there is still exactly one leftmost (or rightmost) derivation, but there are multiple parse trees, and hence multiple derivations.

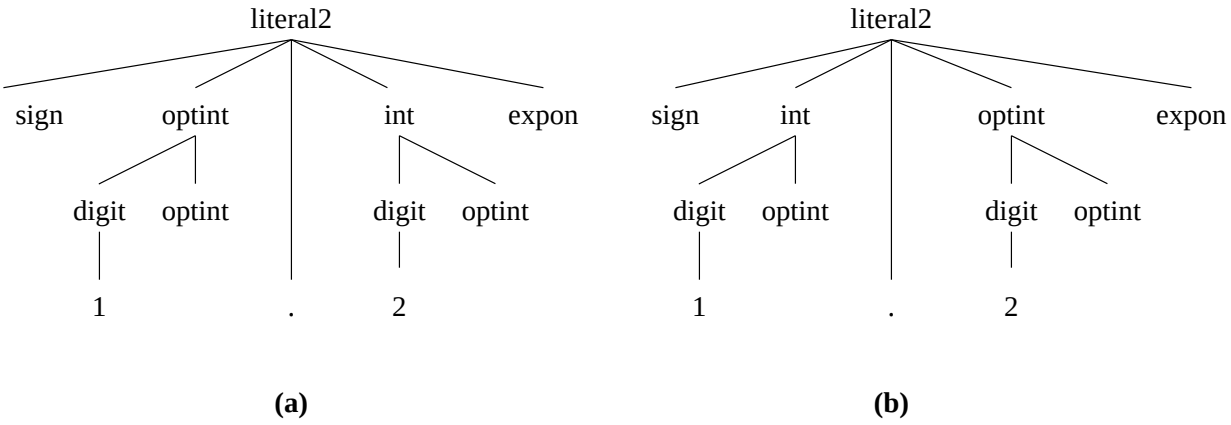

Figure 2: Two parses for '1.2' under the 'literal2' grammar

We say that the *grammar* is ambiguous, not the *language*, because there is an unambiguous grammar for this language: simply replace the definitions for 'literal2' with

```
literal2 →  sign int '.'  int expon
literal2 →  sign '.'  int expon | sign int '.'  expon
```

(There are, in fact, languages that have only ambiguous grammars, but in practice, one doesn't encounter them in compilers.)

6 Context-Free Grammars

So far, the grammars I've written have obeyed the following ordering restrictions:

1. Any nonterminal in the right-hand side of a rule, except possibly one that appears at the right end of the rule, is completely defined in previous rules.

2. If a nonterminal appears as the rightmost symbol on a right-hand side, it is either previously defined or it is the same as the left-hand side symbol of the rule.

That is, the following grammar would *not* qualify:

```
A →  'x' B
B →  'y' B 'z'
```

The first rule mentions a nonterminal that is not yet defined. That problem could be cured by making it the last rule. However, the second rule cannot be fixed: the nonterminal `B` appears in the middle of a rule with `B` on the left side.

Grammars that obey these restrictions I will call *regular* or *Type 3* grammars. It's fairly easy to see that they describe the same languages as do regular expressions—that any such

grammar can be converted to a regular expression and vice-versa. As such, they are subject to the same descriptive limitations as finite-state automata.

Unfortunately, common programming languages are clearly non-regular. For example, the languages you are probably familiar with require that parentheses be balanced. But that is just a slightly modified form of the "equal number of 0s and 1s" example from §2, above. Now, it's true that in practice, one could limit programmers to, say, 20 levels of parenthesis nesting, and that such a language could be described by a regular expression or regular grammar. However, you would find it a rather horrendous grammar.

If we remove the regular-grammar restrictions, we get a class of grammars known as the *Type 2* or *context-free* grammars, which will be our next main object of study. Such grammars can "count" arbitrarily high, at least under the right circumstances. For example, here is a language of correctly nested parentheses:

```
parens →  parens '(' parens ')' | ϵ
```

It recognizes the empty string, '()', '()()', '(())', '((()))', etc.

7 Syntax-Directed Translation

Finding a derivation would merely be an interesting academic exercise (and we haven't even gone into the "how" of it yet) were it not for the fact that we can use it for our Larger Purpose of translating programming languages. In some sense, a parse tree itself is a translation of a programming language into a form that makes it easy to get at the logical units of programs (the "then part" of a conditional, for example). However, we can use the parsing process to direct the formation of other kinds of translation (in fact, we usually do; parse trees are not usually produced directly). For example, suppose I wanted to translate floating literals into `doubles`. I could define how to do so by attaching *semantic actions* to the grammar rules that assign *semantic values* to the nodes of the parse tree.

```
digit →  '0'       { $$ = 0; }
   ⋮
digit →  '9'       { $$ = 9; }
int →  digit       { $$ = $1; }
int →  int digit   { $$ = 10*$1 + $2; }
sign →  '+'        { $$ = 1; }
sign →  '-'        { $$ = -1; }
sign →             { $$ = 1; }
exponent →  'e' sign int
                   { $$ = $2 * $3; }
exponent →         { $$ = 1; }
frac →  digit      { $$ = 0.1 * $1; }
frac →  digit frac { $$ = 0.1 * ($1 + $2); }
literal →  sign int '.'  frac exponent
```

```
{ $$ = $1 * ($2 + $4) * 10**$5; }
```

(Here, I have used the notation of YACC and BISON for the semantic rules (the things in curly braces). Another style you'll sometimes see is to use the names of the nonterminals in the semantic rules:

```
frac →  digit frac  { frac0 = 0.1 * (digit + frac1); }
```

Since we'll be using BISON, let's stick with its notation here to avoid confusion.)

Each semantic rule is to read as a rule assigning a semantic value to each node in the parse tree. Usually, the parse tree stays implicit, and we just use the values attached to it. For a rule of the form $A \rightarrow \alpha_1 \cdots \alpha_n$, the notation `$$` means "the value assigned to a node produced by this rule" and `$`k means "the value assigned to the instance of α_k that a node produced by the rule is attached to." Here, I have taken the liberty to introduce `**` as the exponentiation operator. I have also taken the liberty of re-arranging the definition of `int` according to our new freedom to write context-free grammars.