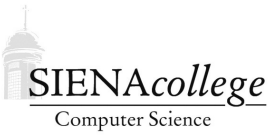


Computer Science 340
Programming Languages
Siena College
Fall 2019

Topic Notes: Syntax and Semantics

We now turn our attention to the general topic of describing the syntax and semantics of a programming language.

A language's *syntax* is the form or structure of the expressions and statements. It includes symbol and grammar rules. The syntax should be easy to learn and intuitive to use.

For example, this is valid C or Java:

```
int x = 7 + 3 - 8;
```

but this is not:

```
int x = 7 + 3 - * 8;
```

The syntax of a Java `while` statement would look like this:

```
while ( <boolean_expr> ) <statement>
```

The partial syntax of an `if` statement:

```
if ( <boolean_expr> ) <statement>
```

These mean that it is legal in the language to have the word "`while`" or "`if`" followed by an open parenthesis, followed by something that represents a "boolean expression", followed by a close parenthesis, followed by something that represents a "statement".

Its *semantics* determines the meaning of the expressions, statements, and program units.

For example, what does it mean when we encounter the following?

```
while ( <boolean_expr> ) <statement>
```

It means we execute `<statement>` zero or more times as long as `<boolean_expr>` evaluates to true.

Together, syntax and semantics define the language, forming the *language definition*. A language definition is the complete description of the language. It can be of use to

1. other language designers
2. implementers of the language
3. programmers, who are the users of the language

Errors in syntax are detected and reported by a compiler – the program just had an invalid sequence of characters or words. For example, a "`while`" that is not followed by a "`(`".

Errors related to semantics are defects in program logic that cause incorrect results or program crashes. For example, you are using a valid `while` loop but your boolean condition is incorrect, causing an infinite loop.

Describing Syntax

We start by focusing on syntax. First, some terminology:

- A *sentence* is a string of characters over some alphabet.
- A *language* is a set of sentences.
- A *lexeme* is the lowest level syntactic unit of a language, such as operators, punctuation, keywords, literals, or identifiers.
  - note: this is one step above individual characters
- A *token* is a set or category of lexemes (*e.g.*, identifier, integer literal).

Here are some examples of lexemes and tokens from a few languages. We have not looked at the language definitions in great detail to know the actual names of the tokens, but most are self-explanatory.

The BASIC statement

```
20 LET X = 2037
```

would be broken down as

Lexemes	Tokens
`20`	`integer_literal` (or: `line_number`)
`LET`	`let_keyword`
`X`	`identifier`
`=`	`equal_sign`
`2037`	`integer_literal`

The C or Java statement fragment (it's missing its body):

```
while ( xPos > 300 )
```

would be broken down as

Lexemes	Tokens
`while`	`while_keyword`
`(`	`open_paren`
`xPos`	`identifier`
`>`	`greater_than`
`300`	`integer_literal`
`)`	`close_paren`

The Java statement:

```
System.out.println( "Number is " + 9 + x );
```

would be broken down as

Lexemes	Tokens
`System`	`identifier` (or: `className`)
`.`	`dot_operator`
`out`	`identifier`
`.`	`dot_operator`
`println`	`identifier`
`(`	`open_paren`
`"`	`double_quote`
`Number is`	`string_literal`
`"`	`double_quote`
`+`	`string_concat_operator`
`9`	`integer_literal`
`+`	`string_concat_operator`
`x`	`identifier`
`)`	`close_paren`
`;`	`semicolon`

Language Recognizers and Generators

A *language recognizer* reads an input string and determines whether it belongs to the given language (*i.e.*, the string is *accepted*) or not (*i.e.*, the string is *rejected*).

This is the *syntax analysis* part of a compiler or interpreter.

A *language generator* produces syntactically acceptable strings of a given language.

It is not practical to generate *all* valid strings. Instead, we would inspect the generator rules (the *grammar*) to determine if a sentence is acceptable for a given language.

Grammars are a formal language-generation mechanism that are often used to describe syntax in programming languages.

In the mid-1950s, linguist Noam Chomsky developed four classes of generative grammars, two of which are useful for us:

- *Context-free grammars* (CFGs) are useful for describing programming language syntax.
- *Regular grammars* are useful for describing valid tokens of a programming language.

In 1960, John Backus and Peter Naur developed a formal notation for specifying programming language syntax. Their *Backus-Naur Form* (BNF) is nearly identical to Chomsky's CFGs.

The syntax of an assignment statement in BNF:

```
<assign> => <var> = <expression> ;
```

This is a BNF rule, or *production* that defines the `<assign>` *abstraction*.

The *definition*, in this case `<var> = <expression>` may consist of other abstractions, lexemes and tokens.

In BNF, abstractions are used to represent classes of syntactic structures. The names of the abstractions, called *nonterminal symbols*, or simply *nonterminals*, act like syntactic "variables". These are often denoted in angle brackets.

Terminals are lexemes or tokens.

A rule has a *left-hand side* (LHS), which is a single nonterminal, and a *right-hand side* (RHS), which is a string of terminals and/or nonterminals.

A set of such rules form the grammar.

For example, let's consider this grammar of BNF rules.

```
<program> => begin <stmts> end
<stmts> => <stmt> | <stmt> ; <stmts>
<stmt> => <var> = <expr>
<var> => a | b | c | d | e
<expr> => <term> + <term> | <term> - <term>
<term> => <var> | literal-integer-value
```

Everywhere we see `|`, it indicates "OR", meaning that the production can use one of the options.

The terminal `literal-integer-value` indicates a token that can be any of a set of lexemes – in this case any valid integer literal.

Here are three sentences that are in this language:

```
begin a = c + d end
```

```
begin a = b + c ; d = a + 7 end
```

```
begin a = c + 100 end
```

But this one is not:

```
begin a = c + d ; end
```

If we tried this, we would like to see a message like:

```
  syntax error! expected end but found ';'
```

But how do we know?

How can we generate a sentence that conforms to this grammar? We can *derive* one.

A *derivation* is a repeated application of rules that convert (eventually) all nonterminals to terminals. We start with a *start symbol* and end with a sentence in the language.

For our example, one possibility:

```
<program>  =>  begin <stmts> end
           =>  begin <stmt> end
           =>  begin <var> = <expr> end
           =>  begin b = <expr> end
           =>  begin b = <term> + <term> end
           =>  begin b = <var> + <term> end
           =>  begin b = c + <term> end
           =>  begin b = c + 123 end
```

Each intermediate form is also called a *sentential form*.

If we can find a derviation for a sentence, then it is in the language. So the sentence:

```
begin b = c + 123 end
```

is in our language.

There can be many (often infinitely many) possible derivations for a given sentence using a given grammar.

Often, we will want a *leftmost (or, rightmost) derivation* is one in which the leftmost (or, rightmost) abstraction is always the next one expanded.

For the sentence

```
begin d = 10 - a end
```

we can generate the leftmost derivation as follows:

```
<program> => begin <stmts> end
          => begin <stmt> end
          => begin <var> = <expr> end
          => begin d = <expr> end
          => begin d = <term> - <term> end
          => begin d = 10 - <term> end
          => begin d = 10 - <var> end
          => begin d = 10 - a end
```

or the rightmost derviation as follows:

```
<program> => begin <stmts> end
          => begin <stmt> end
          => begin <var> = <expr> end
          => begin <var> = <term> - <term> end
          => begin <var> = <term> - <var> end
          => begin <var> = <term> - a end
          => begin <var> = 10 - a end
          => begin d = 10 - a end
```

Why is the leftmost (or rightmost) derivation important? It is the one that would likely be used by a program attempting to parse its input.

For practice, consider this simple grammar:

```
<S> => <A> <B> <C>
<A> => a <A> | a
<B> => b <B> | b
<C> => c <C> | c
```

Which of the following sentences are generated by this grammar?

- `baaabbccc`
- `abc`
- `abcabc`
- `bbaabbaabbaabbaac`

- `aabbbbccccccccccccccccccc`

Parse Trees

A *parse tree* represents the structure of a derivation.

- Every internal node is a non-terminal abstraction.
- Every leaf node is a terminal symbol.

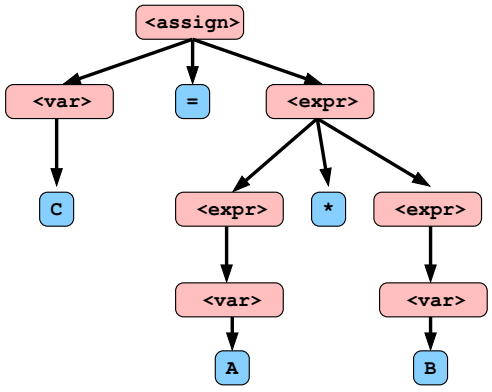
For the grammar:

```
<assign> => <var> = <expr>
   <var> => A | B | C | D
  <expr> => <expr> + <expr>
          | <expr> * <expr>
          | ( <expr> )
          | <var>
```

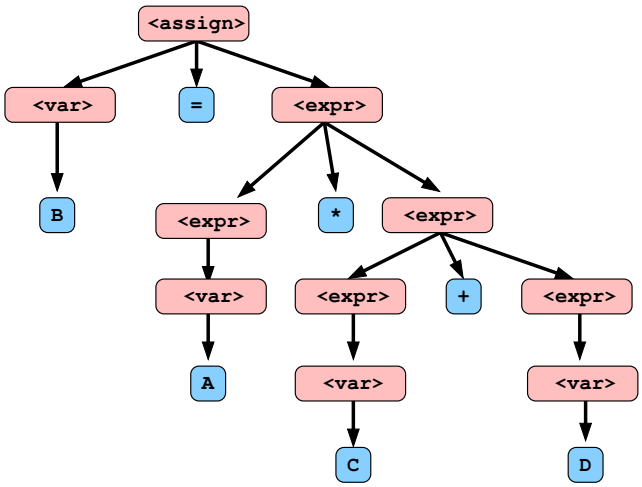
We can derive the sentence `C = A * B` with the following:

```
<assign> => <var> = <expr>
         => C = <expr>
         => C = <expr> * <expr>
         => C = <var> * <expr>
         => C = A * <expr>
         => C = A * <var>
         => C = A * B
```

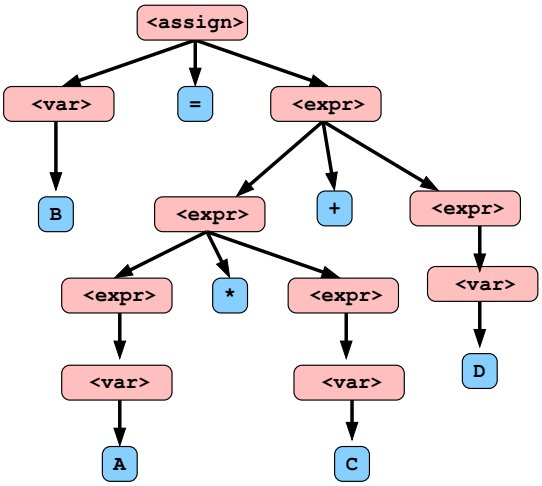
which corresponds to this parse tree:



Next, we draw a parse tree for `B = A * C + D`



Very nice, but why that instead of:



Is one better than the other? If so, why?

A grammar that generates a sentential form for which there are two or more distinct parse trees is an *ambiguous grammar*.

Ambiguity in a grammar leads to problems because compilers often base *semantics* on parse trees.

- operator precedence and associativity
- if-else

An *unambiguous grammar* has exactly one derivation and parse tree for each unique sentential form.

For the above ambiguous grammar, the following unambiguous grammar generates the same language:

```
<assign> => <var> = <expr>
   <var> => A | B | C | D
  <expr> => <expr> + <term>
          | <term>
  <term> => <term> * <factor>
          | <factor>
<factor> => ( <expr> )
          | <var>
```

This grammar enforces the precedence of multiplication over addition.

We also need to consider *associativity* of operations that are indicated by a grammar.

Consider this assignment statement:

```
A = B + C + A
```

A leftmost derivation will result in a parse tree that will cause `B + C` to be computed first, then the result added to `A`. This is what we would expect from our usual left-to-right evaluation of operations that are of equal precedence.

Mathematically, it would not matter if we had a grammar that resulted in `C + A` being computed first.

But what if the statement was

```
A = B / C * A
```

Here, even though the two operations are at the same precedence level, it is important that they are evaluated left to right.

With integer addition associativity would not matter, but note that with floating point addition, it could.

The Dangling `else`

If you write the following code:

```
if x > 0 then
  if y > 0 then
    y++;
  else
    z++;
```

Does the `else` go with the first `if` or the second? It would be excellent if this is not ambiguous (likely want it attached to the second, as the indentation above suggests).

Consider this grammar for an `if` statement:

```
<if_stmt> => if <logic_expr> then <stmt>
           | if <logic_expr> then <stmt> else <stmt>
```

We can construct two parse trees, one of which attaches the `else` to the first, the other to the second.

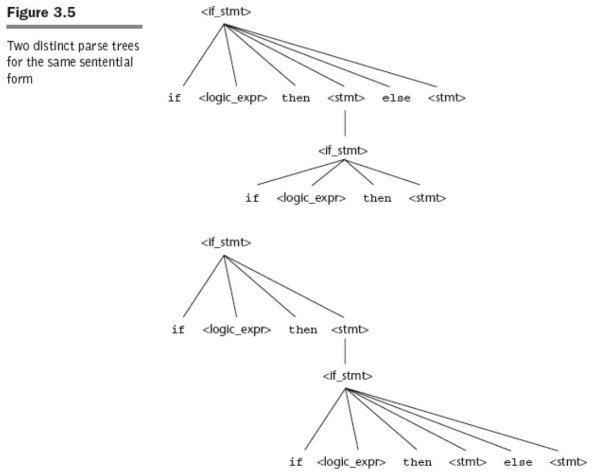
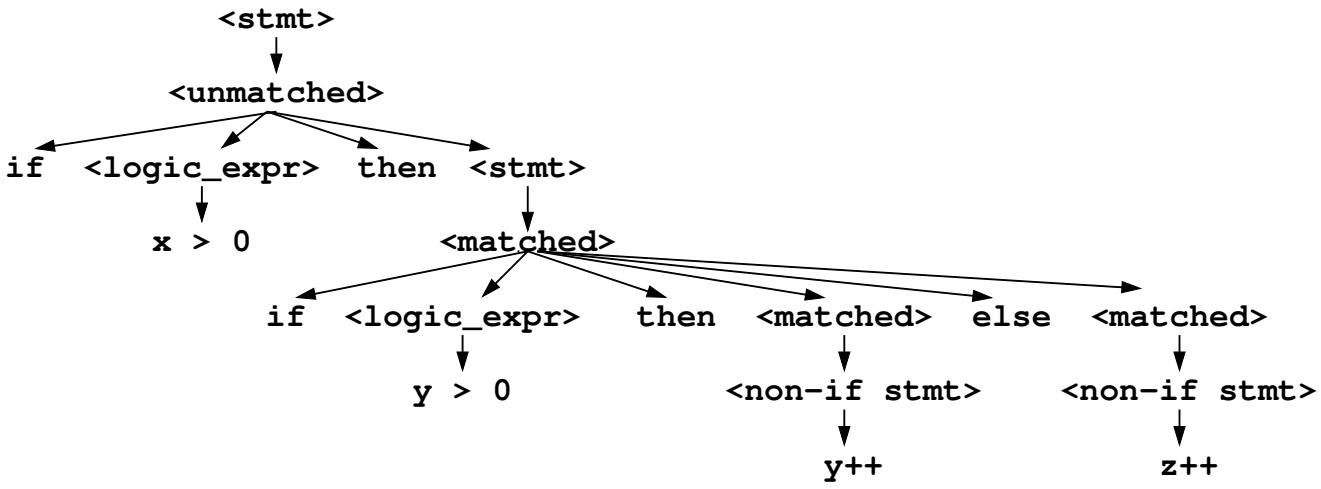


Figure 3.5 from Sebesta 2012.

We can create a more complex, but unambiguous grammar to ensure the else gets matched as we intend:

```
<stmt> => <matched> | <unmatched>
<matched> => if <logic_expr> then <matched> else <matched>
           | <non-if stmt>
<unmatched> => if <logic_expr> then <stmt>
             | if <logic_expr> then <matched> else <unmatched>
```

This gives us a unique parse tree for the program snip with the dangling else.



Note: Sebesta describes some enhancements to BNF that we will introduce as needed.

Static Semantics

Some aspects of syntax cannot be represented easily, and in some cases not at all, using CFGs.

For example, restrictions on the types of operands in expressions can be represented, but would require a much more cumbersome grammar. In Java, if we have a variable `x` of type `int` and a variable `y` of type `double`, the assignment `y = x;` is legal, but `x = y;` is not.

A syntactic restriction that cannot be represented in a context-free grammar (and hence not using BNF), is the enforcement that a variable must be declared before it is used.

The term for these types of language rules fall under the category of *static semantics*. That is, the parts of the language whose semantics can be checked at compile time, or "statically."

One approach to enforcing static semantics is to use *attribute grammars (AGs)*. With AGs, some additional semantic information is stored with nodes of a parse tree that was created with a CFG.

The text defines AGs more formally, but we will consider an example (based on Example 3.6 in Sebesta) to see the idea.

Consider this simple CFG:

```
<assign> => <var> = <expr>
<expr> => <var> + <var> | <var>
<var> => A | B | C
```

We want to enforce consistency of data types, which can either be `int_type` or `real_type`. There are two *attributes* introduced on our nonterminals: `actual_type` and `expected_type`.

- The `actual_type` is a *synthesized attribute* that, as its name suggests, stores the actual type of data to be represented by that nonterminal.

  - For a <var> nonterminal, we look up the variable's name to see what type it represents (ignoring for this example the details of this lookup).
  - For a <expr> nonterminal, the actual_type is determined by the actual_types of the RHS.

- The expected_type is an *inherited attribute*.
  - This only applies to the <expr> nonterminal, and is determined by the variable to which its value is being assigned.

The full attribute grammar is shown in Example 3.6 of Sebesta. A few things to note:

- Each syntax rule has a corresponding *semantic rule* that sets the actual_type or expected_type of a nonterminal in the syntax rule.
- Two of the syntax rules have a *predicate* that is used to enforce situations where the actual_type must match the expected_type of one of the <expr> nonterminals.

The text shows a parse tree for the sentence A = A + B (and then *decorates* it with information from the semantic rules), assuming that A is of real_type and B is of int_type. This is shown in Figures 3.6, 3.7, and 3.8.

Dynamic Semantics

The remainder of Chapter 3 deals with *dynamic semantics* – determining the meaning of programs and their components.

This topic is beyond the scope of our course, though some of the issues will arise during our later discussions.

Lexical Analysis

We now leave syntax analysis and parse trees for a bit to consider *lexical analysis* – the process of identifying the small-scale language constructs.

Here, we identify the lexemes – names, operators, numeric literals, punctuation, line numbers (BASIC), etc.

In many ways, lexical analysis is similar to syntax analysis, but it is generally a easier problem.

So lexical analysis is usually performed separately from syntax analysis. Why?

- Simplicity: simpler approaches are suitable for lexical analysis
- Efficiency: focuses optimization efforts on lexical analysis and syntax analysis separately

- Portability: a lexical analyzer might not always be portable (due to file I/O), whereas syntax analyzer may remain portable

The lexical analyzer is simply a *pattern matcher*.

- Identifies and isolates lexemes
- Is a "front-end" for the parser, which can then deal strictly with tokenized input
- Lexemes are logical substrings of the source program that belong together
- Lexical analyzer assigns codes called tokens to the lexemes
  - *e.g.*, `sum` is a lexeme; and `IDENT` is the token

Before we look at specifics of how a lexical analyzer works, let's think about what some of these lexemes look like.

First, consider integer constants in C/C++. These include:

- an optional unary minus sign
- digits
- optional e notation
- different prefixes for octal and hexadecimal

See Example:
`/home/cs340/examples/intconstants`

To create a formal definition of an integer with the restriction that it must be in base 10 and that it does not use e notation:

$$(\epsilon \bigcup -) \cdot (1 \bigcup 2 \bigcup 3 \bigcup 4 \bigcup 5 \bigcup 6 \bigcup 7 \bigcup 8 \bigcup 9) \cdot (0 \bigcup 1 \bigcup 2 \bigcup 3 \bigcup 4 \bigcup 5 \bigcup 6 \bigcup 7 \bigcup 8 \bigcup 9)*$$

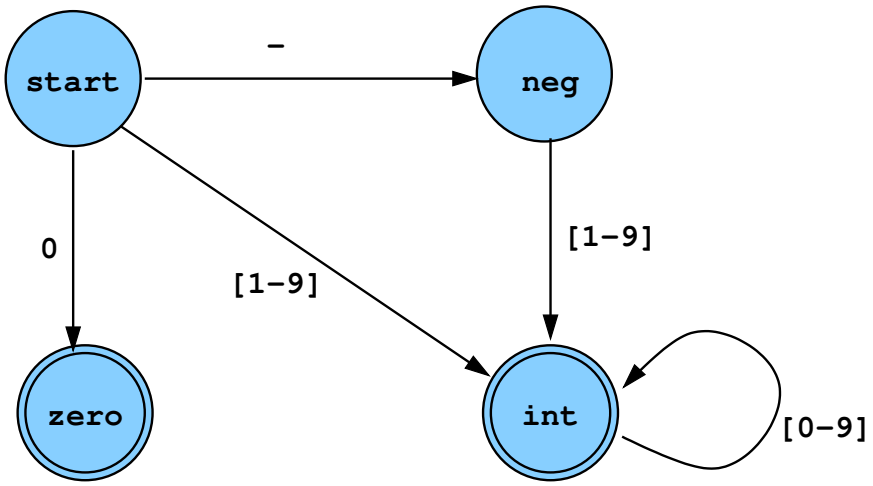
this means either nothing or a unary -, followed by one digit in the 1-9 range, then 0 or more copies of digits 0-9. The "any number" is indicated by the $\star$ at the end.

Alternately, we could use a Unix-like *regular expression*:

```
(-?[1-9][0-9]*|0)
```

Again, an optional -, one digit 1-9, zero or more digits 0-9, OR a single 0.

We can also see this as a *deterministic finite automaton (DFA)* or *state diagram*.



This can also be described by a grammar.

```
<int-literal> => -<unsigned-int>
               | <unsigned-int>
               | 0
<unsigned-int> => [1-9]
                | [1-9]<one-or-more-digits>
<one-or-more-digits> => [0-9]
                      | [0-9]<one-or-more-digits>
```

A language is *regular* if

- It can be represented by a regular expression.
- It can be represented by a deterministic finite automaton (DFA).
- It can be represented by a regular grammar.

These are all equivalent statements.

We have seen grammars. A *regular grammar* is one that has a very restricted form for its productions:

- a production's RHS may be a single terminal
- a production's RHS may be a single terminal followed by a single nonterminal

A grammar is regular iff it produces a regular language.

The grammar given above for integer literals is not a valid regular grammar because of the second rule (its RHS is a single nonterminal). We can rewrite it a bit to eliminate this.

```
<int-literal> => -<unsigned-int>
               | [1-9]
               | [1-9]<one-or-more-digits>
               | 0
<unsigned-int> => [1-9]
                | [1-9]<one-or-more-digits>
<one-or-more-digits> => [0-9]
                      | [0-9]<one-or-more-digits>
```

We've basically put a copy of the productions for `<unsigned-int>` into the productions for `<int-literal>` to come up with an equivalent grammar which now does satisfy the requirements for a regular grammar.

A Lexical Analyzer

Our textbook has a demonstration of a simple lexical analysis program for arithmetic expressions in Section 4.2.

It is worth some time to understand the relation between the state diagram below (from Sebesta) with the program, and to understand how the program works.

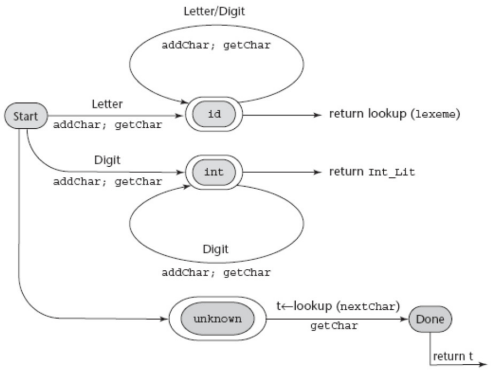


Figure 4.1 from Sebesta 2012.

An improved version of the C program from the text:

See Example:
`/home/cs340/examples/front`

See the extended comments in the code for more.

The Parsing Problem

We now turn our attention back to the more complicated problem of *parsing* a program in a given language.

The parser should be able to:

- Find syntax errors and report them with appropriate messages.
- Produce the parse tree for the program.

There are two major categories of parsers: *top down* and *bottom up*.

- A top down parser builds the tree from the root, matching a leftmost derivation.
  - The parser must choose the correct production of the leftmost nonterminal in a sentential form to get the next sentential form in the leftmost derivation, using only the first token produced by that leftmost nonterminal.
  - The most common top-down parsing algorithms are *recursive descent* and *LL parsers*.
- A bottom up parser starts at the leaves, matching a rightmost derviation.
  - Given a sentential form, determine what substring of the form that is the right-hand side of the rule in the grammar that must be reduced to produce the previous sentential form in the right derivation. (Yikes!)
  - The most common bottom-up parsing algorithms are in the *LR parser* family.

In order to be useful, a parser should look ahead only a single token in the input.

The Complexity of Parsing:

- Parsers that can be used for an arbitrary unambiguous grammar are complex and inefficient ($O(n^3)$, where n is the length of the input).
- A parser for a programming language compiler needs to be much more efficient ($O(n)$), so programming languages must have much more restrictive grammars to make this possible.

Recursive Descent Parsing

A *recursive descent parser* is a top down parsing technique that consists of a collection of procedures which mimic the RHS of all productions for each nonterminal.

- It is often easy to generate from EBNF representations.
- It can use backtracking (trial and error, essentially) to try multiple options when it is not clear which rule must be applied next, but this is inefficient and we strive to avoid it.

Consider this unambiguous grammar:

```
<expr> => <expr> + <term> | <expr> - <term> | <term>
<term> => <term> * <factor> | <term> / <factor> | <factor>
<factor> => ( <expr> ) | id | int-constant
```

We first convert it to the *Extended Backus Naur Form (EBNF)*, which permits some shorthand notations in our grammar:

```
<expr> => <term> { ( + | - ) <term> }
<term> => <factor> { ( * | / ) <factor> }
<factor> => ( <expr> ) | id | int-constant
```

The items inside the `{` and `}` are items in those rules that can be repeated (or left out). The options inside the parens separated by `|` represent a choice of either of those.

Each rule in the grammar becomes a function in the recursive descent parser.

See Example:
`/home/cs340/examples/recdescent`

As written, this program parses only expressions (not full assignment statements), so we begin by calling `lex` and then `expr`.

The `expr` function matches a term followed by any number of + or - tokens followed by another term. When it needs to match another nonterminal, we make a call to that nonterminal's function. When we need to match a terminal, we need to find it in `nextToken`, then call `lex` to advance to the next.

The `term` function is very similar to `expr`.

The `factor` function has more work to do.

A nonterminal that has more than one RHS requires an initial process to determine which RHS it is to parse

- The correct RHS is chosen on the basis of the next token of input (the lookahead)
- The next token is compared with the first token that can be generated by each RHS until a match is found
- If no match is found, it is a syntax error

This is demonstrated by the `factor` function.

Let's look at what an `if` statement's recursive descent parser might look like (assuming we had lots of other functionality added to support this):

The production this implements is

```
<ifstmt> => if ( <boolexpr> ) <statement> [ else <statement> ]
```

```
void ifstmt(){

  if (nextToken != IF_CODE) {
    error("expected if");
  }
  else {
    lex();  // match the if
    if (nextToken != LEFT_PAREN) {
      error("expected (");
    }
    else {
      lex(); // match the (, (Note: error in text; this was omitted)
      boolexpr();
      if (nextToken != RIGHT_PAREN) {
        error("expected )");
      }
      else {
        lex(); // match the ), (Note: error in text; this was omitted)
        statement();
        if (nextToken == ELSE_CODE){
          lex(); // match the else
          statement();
        }
      }
    }
  }
}
```

This is more complex than the ones we saw, but the idea remains the same.

Restrictions of Recursive Descent

Not all grammars can be immediately parsed by a recursive descent method, but rules may be rewritten in order for recursive descent to work.

If we have a grammar with a rule like:

```
<expr> => <expr> + <term>
```

This production has *left recursion*, which would lead to an infinitely recursive `expr` method.

The grammar needs to be rewritten to eliminate the left recursion. This process can be done with *Paull's Algorithm*.

To remove immediate left recursion – a nonterminal with productions that have the same nonterminal on the left:

A => Aα_1 | ... | Aα_m | β_1 | ... | β_n

where none of the β_i begins with A, becomes

A => β_1A$'$ | ... | β_nA$'$
A$'$ => α_1A$'$ | ... | α_mA$'$ | ϵ

See the example in Sebesta Section 4.4.2 for an application of this to an actual grammar.

Another restriction is that we should be able to choose the correct RHS of a production with multiple rules based only on the next token on the input. To do this, the grammar must pass the *pairwise disjointness test*.

For each nonterminal A with more than one RHS, it must be the case that for each pair of rules A => α_i and A => α_j,

$$FIRST(\alpha_i) \cap FIRST(\alpha_j) = \emptyset$$

where

$$FIRST(\alpha) = \{a | \alpha -> ^* a\beta\}$$

and if $\alpha -> ^* \epsilon$, then ϵ is in $FIRST(\alpha)$.

There are algorithms to compute the $FIRST$ sets, but for the grammars we will consider, we can determine these by looking at the rules.

There are also algorithms to "left factor" a grammar to allow them to pass the pairwise disjointness test.

Grammar Classes

A grammar is said to be *LL(k)* if parsing decisions require only k tokens of lookahead.

- First L stands for Left to Right scanning of token input
- Second L stands for producing a leftmost derivation

An LL(1) grammar lends itself to recursive descent parsing.

Other grammar classes include LR(k), LALR(k) – topics for a compilers course.

We will not consider these other classes in detail, nor will we look in detail at bottom up parsing.

Other Parsing Issues

There are a number of other parsing-related issues that are worth mentioning, but which are all beyond the scope of this course.

We have only considered correctness issues beyond construction of a parse tree with a brief look at attribute grammars. But how do we ensure that variables are declared before use? How do we ensure operations are on the correct type (such as the attribute grammar approach)? How do we take a valid integer or floating point literal and turn it into a usable binary representation?

These and other issues are discussed in the parts of chapter 3 we did not cover in class. These topics are interesting and useful, but you will not be responsible for that material.

Tools: lex and yacc

You are or soon will be experienced in writing a recursive descent parser. Parsers for real programming langauges are significantly more complicated. Someone designing a new language is unlikely to be interested in developing a full tokenizer and parser, especially if it cannot be done as recursive descent.

Since this is a relatively common task, tools have been developed to simplify the tasks. Perhaps the most common tools are *lex* and *yacc*, and variants (such as *flex* and *bison*). These tools are used for full-fledged compilers for programming languages, but also in many other contexts where structured input needs to be transformed.

As an example, we will take a look at a program I wrote many years ago that uses *lex* and *yacc* to process scripts in a very simple language.

See Example:
`/home/cs340/examples/pmdbtool`

The details of what the language does are not important, but we will examine how the *lex* input file (`pt_lex.l`) defines the tokens in the language, and the *yacc* input file defines the syntax and what the program should do when various rules match the sequence of input tokens.