CS323 Lecture - Specifying Syntax and Semantics Last revised 1/16/09

Objectives:

1. To review previously-studied methods for formal specification of programming language syntax, and introduce additional methods:
   a. The concept of metalanguages
   b. The hierarchy of formal languages.
   c. Regular languages/expressions
   d. BNF and EBNF

2. To discuss methods of handling rules that cannot be specified by a context-free grammar, or which may be difficult to specify in this way:
   a. Comments
   b. Declarations of Identifiers
   c. Reserved Words/Keywords/Predeclared Identifiers
   d. Operator precedence (if a language has many levels)

3. To briefly introduce alternative ways of specifying semantics

I. Introduction

A. In our discussion of programming language design and evaluation criteria, we saw that one important characteristic of a good language design is WELL-DEFINEDNESS: Both the syntax and the semantics of the language should be clearly and unambiguously defined. Today, we begin to consider ways of addressing the first of these issues.

B. It turns out that there are a number of very good tools and methodologies available to us for specifying the syntax of programming languages.

   1. This is important, because without such tools the writing of compilers would be impossible.

   2. The first phase of a compiler must be, in fact, an executable syntax specification for the language - any syntactically-correct program must pass through it unscathed, and any syntactically-incorrect program must be detected.

   3. In fact, for certain kinds of grammars, it is even possible to automate the process of producing the parser phase of the compiler - the part that analyzes the syntax of the program.

      For example, most Unix/Linux systems have a tool known as yacc (or the gnu version - bison):

      a. Yacc = yet another compiler compiler.

      b. The input to yacc is the grammar of a programming language, expressed in the metalanguage BNF.

      c. The output of yacc is a C program that parses programs in the language.

C. We will consider several tools which are commonly used for specifying syntax. These tools are what we call METALANGUAGES - i.e. languages for describing other languages.

D. Before we do so, though, we want to spend some time reviewing the theoretical underpinnings of these tools. These metalanguages are built on the foundation of theoretical work in the realm of FORMAL LANGUAGES - a discipline that predates Computer Science per se, and which undergirds both Computer Science and some aspects of the study of natural languages. This material was the subject of Models of Computation.

II. Review of Formal Languages and the Chomsky Hierarchy (Recall CPS220)

A. Some definitions:

1. Recall that A FORMAL LANGUAGE is a SET OF STRINGS built up of symbols from some ALPHABET according to the rules of a FORMAL GRAMMAR. We sometimes refer to the set of strings comprising a language as the "SENTENCES" of the language.

2. A formal language, can be characterized by three sets and a symbol:

   a. An alphabet, or set of terminal symbols.

   b. A set of non-terminal symbols.

   c. A set of productions.

   d. A designated non-terminal symbol called the start symbol.

3. For a given grammar, a DERIVATION is the process of starting with the start symbol and applying the rules of the grammar until a form is derived composed only of terminal symbols (which is then a member of the language). That is, the set of strings that can be derived by a given grammar is the language defined by that grammar.

   a. For the type of grammars we will be using, each step in the derivation will involve replacing one of the non-terminals in the evolving sentence with the right side of one of its productions. (For more complex grammars, the derivation process may also be more complex.)

   b. The inverse of derivation is PARSING. In parsing, we work backward from a member of the language, attempting to trace its derivation. (If this fails, then we conclude that the form we are working with is not a member of the language.)

      i. Thus, a programmer performs a derivation while writing a program.

      ii. The compiler parses the program in order to reconstruct this derivation, which then serves as the basis for translating the program.

      iii. A parse of a sentence can be represented by a PARSE TREE whose root is the start symbol and whose leaves are the terminal symbols of the sentence. (Hence their name, terminal symbols.)

B. One of the most important discoveries in theoretical computer science is the discovery of a hierarchical structure to categories of languages and grammars, and a related hierarchy of formal automata that can recognize (i.e parse) sentences described by the grammar. This hierarchy is called the Chomsky hierarchy:

Phrase Structure (or Type 0) Grammars	Turing Machine
Context-Sensitive Grammars	Linear bounded automata
Context-Free Grammars	Push-down automata
Right-linear or Regular Grammars	Finite-state machines

C. For the purpose of work in programming languages, two classes of grammars are of most interest.

1. Complete programming languages are typically described by context-free grammars. In fact, most of the metalanguages we will consider are designed for describing context-free grammars. (While context sensitive grammars or phrase structure grammars are more powerful in terms of the features they can describe, constructing a parser for such grammars is much more difficult; hence most programming languages are context free.)

2. However, certain features of a language can often be described by a simpler, regular grammar.

a. In particular, this is typically true of the rules for forming the TOKENS of the language - e.g.

- Numeric constants
- String constants
- Identifiers

etc.

b. Thus, for ease of parsing, programming language grammars are often defined on two levels.

i. The syntax of the tokens is described by a regular grammar, whose alphabet is the character set of the programming language and whose "sentences" are the tokens of the language.

Example: part of the regular grammar for Pascal tokens:

```
identifier:            (A..Z | a..z)(0..9 | A..Z | a..z | _)*

integer constant:      (0..9)(0..9)*
...
```

ii. The language itself then described by a context-free grammar, whose alphabet is the tokens described by the first grammar.

c. This leads to the parsing step of compilation being done in two phases:

i. Lexical analysis, whereby the incoming stream of characters is broken up into tokens, according to the rules of the token grammar (which is regular).

ii. Parsing proper, done according to the rules of a context-free grammar.

```
                  ___________________            __________
                 |                   |          |          |
----------->     | Lexical scanner   |  ------->| Parser   | ------->
Source text      |___________________|  Tokens  |__________| Parse tree
```

3. Unfortunately, not all aspects of programming language syntax can be handled by a context-free grammar. In particular, context-free grammars cannot handle declaration rules for identifiers.

    a. Thus, the following rules of a language like Java cannot be specified in its formal grammar:

        - Every identifier must be declared before it is used.

        - The same name cannot be declared twice in the same block.

        - An identifier must be used in a way that is consistent with its declaration.

    b. These rules could be handled by a context-sensitive grammar; but the resulting grammar would be too complex to serve as the basis for constructing a practical parser.

    c. Thus, the typical approach of compiler writers is to parse the program on the basis of a context-free grammar, and to handle context-sensitive features on an ad-hoc basis. (We will say more about this later.)

D. We now turn our attention to metalanguages for describing context-free grammars.

III. Backus-Naur Form and Extended Backus-Naur Form

A. One of the most commonly used tools for describing context-free grammars is Backus-Naur Form, or BNF.

1. Actually, two variants of BNF are in use - the original form, and an extended form called EBNF.

    a. The two variants are of equal power. Any grammar that can be described by one can be described by the other. Further, any context-free language can be defined by a BNF grammar (of either sort), and any language defined by a BNF grammar (of either sort) is context free.

    b. EBNF is more convenient to write and somewhat easier to read, and thus is often used in writing manuals, etc.

    c. However, plain BNF is more useful when a grammar is to be submitted to an automatic parser generator, such as yacc.

2. We will look at both forms - first regular BNF, then EBNF.

B. A BNF grammar consists of a series of productions, like the following example (which illustrates some of the major features of BNF):

```
<if-statement> ::= if ( <expression> ) <statement> |
                   if ( <expression> ) <statement> else <statement>
```

1. In BNF, non-terminal symbols are enclosed in angle brackets to distinguish them from terminals. Thus, in the above, <if-statement>, <expression> and <statement> are non-terminal symbols, while if, (,), and else are terminals. (Some notations use italics for non-terminals instead of angle brackets, but we'll stick with angle brackets for clarity)

2. A vertical bar is used to separate alternatives. The above production says that an <if-statement> can be produced in either of two ways. This could also have been written as two productions, as follows:

   ```
   <if-statement> ::= if ( <expression> ) <statement>
   <if-statement> ::= if ( <expression> ) <statement> else <statement>
   ```

3. The distinctive feature of BNF productions (and indeed of all context free grammars) is that the left hand side consists of a single non-terminal symbol. Further, every non-terminal in the grammar must appear on the left-hand size of at least one production.

4. One feature not illustrated by the above is repetition. For example, the following is part of a C-style compound statement (omitting the provision for declaring local variables):

   ```
   <compound-statement> ::= { } | { <statement-list> }
   <statement-list> ::= <statement> |
                        <statement-list> <statement>
   ```

   a. Notice that repetition is specified by recursion.

   b. For practical reasons, it is sometimes helpful to distinguish two kinds of recursion: LEFT RECURSION (as above) and RIGHT RECURSION.

      i. The above definition of statement-list could be rewritten using right-recursion, as follows:

      ```
      <statement-list> ::= <statement> |
                           <statement> <statement-list>
      ```

      ii. The issue of which form of recursion to use arises in conjunction with certain techniques for turning grammars into parsers. Some techniques require that recursion be limited to one or the other of these two forms.

      iii. Note that it is generally possible to transform a construct defined using left recursion into one using right recursion and vice-versa.

C. Extended BNF (EBNF) adds some notation to BNF to make certain constructs easier to write. In particular, it uses the meta-characters [] and {}.

   1. [] can be used to enclose an optional portion of a production. For example, our if-statement rule could be re-written as follows:

      ```
      <if-statement> ::= if ( <expression> ) <statement> [ else <statement> ]
      ```

2. {} can be used to enclose a portion of a production that can be repeated zero or more times.

   a. For example, our statement-list rule could be re-written as follows:

```
<statement-list> ::= <statement> { <statement> }
```

   b. In fact, compound-statement could now be defined using this notation without having to invent the non-terminal statement-list, as follows. (We quote the terminal use of { and } to distinguish from the metalanguage use)

```
<compound-statement> ::= "{" "}" | "{" statement { statement } "}"
```

IV. Handling Language Features that are not Context-Free

A. We have thus far been working exclusively with context free grammars. As we noted earlier, these cannot handle certain syntactic elements that are, in fact, context-sensitive.

   1. Because handling these elements by means of a context-sensitive grammar would be impractical, they are usually handled instead by ad-hoc means.

   2. In particular, recall that most compilers use parsers that actually operate in two stages:

      a. A lexical scanner breaks the incoming text up into units of meaning called TOKENS - e.g. constants, identifiers, reserved words, operators, etc.

      b. The stream of tokens serves as input to the parser proper.

```
                  ___________________          __________
                 |                   |        |          |
-----------> | Lexical scanner |  -------> | Parser | ------->
Source text      |___________________|  Tokens  |__________| Parse tree
```

      c. By doing some additional processing on the stream of tokens, we can handle many of the problem areas while still using context-free grammars in the parser.

   3. We now consider some of the problem areas and ways they can be handled.

B. One problem area is WHITESPACE

   1. By whitespace, we mean spaces and other similar characters - e.g. tabs, newlines.

   2. Most languages allow whitespace between any two tokens, and even require whitespace between certain kinds of tokens (e.g. between two identifiers or between an identifier and a reserved word.)

      Example: In Java, if one wishes to name a class Foo, one has to write it class Foo, not classFoo.

3. Formal grammars do not usually deal with whitespace; instead, the lexical scanner handles it in an ad-hoc fashion.

    a. When building a token, the encountering of whitespace (except within a quoted string) signals the end of the token.

    b. Outside of quoted strings, sequences of one or more whitespace characters are discarded by the lexical scanner and not passed on to the next phase.

C. Another problem area is COMMENTS. Almost all languages have them, but grammars seldom mention them.

1. The reason for this is apparent, when one considers what would be required to include comments in a grammar. For example, in Java a comment can occur anywhere whitespace can occur, so the grammar for a statement like the if-statement would have to look like this:

```
<if-statement> ::= [comment] if [comment] ( [comment] <expression>
                        [comment] ) [comment] statement [comment]
                    [ [comment] else [comment] <statement> [comment] ]
```

2. Rather than clutter the grammar this way, the lexical scanner simply recognizes and discards comments. This can be done by having a syntax for a comment token type, but then not passing this kind of token on to the next phase when one is encountered.

D. A more serious problem arises in conjunction with rules for the DECLARATION OF IDENTIFIERS.

1. Most languages require that an identifier be declared before it is used.

2. Further, most languages require that an identifier not be declared twice in two different ways in the same context (e.g. the same Pascal block.)

3. Since these rules cannot be modelled by a context-free grammar (and would even be very complex to model using a context-sensitive grammar), the formal grammar does not try to address them. For example, the following would be an acceptable program fragment according to the formal grammar of a C-like language:

```
{
   int x;
   double x;
   char x;
   y = 1;              /* No declaration anywhere for y */2
   ...
```

4. How, then, are declaration rules enforced? One ad-hoc approach is to handle them as follows:

    a. One of the most important data structures used by a compiler is its symbol table, in which information is recorded about the declarations of identifiers.

b. Whenever an identifier is recognized by the lexical scanner, it can be immediately looked up in the symbol table, so that information about its declaration (if any) can be passed along with it to the parser.

c. The parser can be augmented with action routines that work like this:

   i. If a declaration is recognized, then the information passed along with the identifier should indicate that it has no prior declaration in the current block. (If not, then an error message is generated.) If all is well, the action routine then adds it to the symbol table.

   ii. If a use of an identifier is recognized, then the information passed along with the identifier should indicate it was declared in a way that was compatible with the way it is being used. (If not, then an error message is generated.)

d. Thus, the structure of our parser becomes:

```
                 ___________________   Tokens   _________
                |                   |           |         |
-----------> | Lexical scanner |  -------> | Parser | ------->
Source text     |___________________|      ---> |_________| Parse tree
                                             |        |
                                             |        |
                                            _|________v______
                                           |                 |
                                           | Symbol table    |
                                           |_________________|
```

E. A related problem is the issue of RESERVED WORDS/KEYWORDS/PREDECLARED IDENTIFIERS

1. Most languages specify in their syntax that an identifier consists of any number of alphanumeric characters, of which the first must be alphabetic (as in Java.) Unfortunately, this definition fits the pre-defined "vocabulary" of the language, too. For example, Java reserved words such as class, int, if etc. qualify syntactically as identifiers.

2. How shall this be handled? Most languages associate with their grammer a list of reserved words, and enforce the rule that these words may not be used as identifiers, even though they otherwise fit the syntax of identifiers.

   a. Of course, there is no way to incorporate this in the formal grammar, per se.

   b. Instead, this can be handled in an ad-hoc way by having the lexical scanner incorporate a table of reserved words. Any time a token is recognized that might be an identifier, it must first be looked up in this table. If it is found there, it is a reserved word, rather than an identifier, and the information passed on to the parser will so indicate.

3. An alternate approach is taken by some languages, which allow keywords to also be used as identifiers.

   a. Example: the following is legal FORTRAN:

```
          IF (IF .EQ. IF) DO 1, DO = DO, DO
    1     WRITE (*,*) WRITE
```

   b. Note that the technical term RESERVED WORD does NOT apply in this case. FORTRAN is a language that has no reserved words.

F. One final syntax issue is the matter of OPERATOR PRECEDENCE

1. As we know, there are two possible interpretations of an expression like:

   2 * 3 + 4

   For a given language, only one interpretation is correct - though different languages may handle this differently.

2. How are operator precedence rules to be represented in a grammar?

   a. In some cases, it is possible to actually build the rules into the grammar.   For example, here is a simple BNF grammar for arithmetic expressions:

```
< expression > ::= < term > { < addop > < term > }
< addop >      ::= "+" | "-"
< term >       ::= < factor > { < mulop > < factor > }
< mulop >      ::= "*" | "/"
< factor >     ::= < constant > | < variable > | "(" < expression > ")"
```

   (This is simpler than even the simplest actual programming language, though Pascal is not much more complex. It is possible to construct a BNF grammar this way for a language like C, that has 16 different levels of operator precedence, ranging from the lowest (, and the various assignment operators) to the highest (including . and ->).

   b. However, some languages (especially those with many levels of operator precedence) handle this outside the grammar, by using a precedence table in the parser. (Java, for example)

V. Specifying Semantics

A. All of what we have discussed thus far has been concerned with specifying the syntax of a programming language - i.e. what constitutes a correct program? We now turn briefly to specifying semantics - what does a program mean? This is, in fact, a much harder question.

B. There are basically four approaches that can be taken to this issue.

1. The use of natural language

2. Operational semantics - the meaning

3. Denotational semantics

4. Axiomatic semantics

We will consider the last three just briefly

C. Operational semantics builds on a time-honored approach to understanding what a given language construct means: write a program incorporating the construct, run it, and see what it does. That is, the meaning of a construct is the effect it has on the state (memory and register contents) of the machine.

1. As a semantic tool, operational semantics formalizes this by defining a specific machine, with its own machine language, plus a translation algorithm for mapping constructs of the higher-level language into the machine language of that machine.

   a. Normally, the machine used is an ABSTRACT or VIRTUAL MACHINE, rather than an actual one. The instruction set of the abstract machine will be carefully designed to provide for straight-forward translation of the constructs of the language whose semantics it is to be used to define.

   b. Since the abstract machine does not physically exist, it is simulated by a software package running on a real machine.

   c. A simple example may clarify the idea:

      - Suppose we are interested in the semantics of a very simple language having only one data type - integer - and the following grammar. (We will confine ourselves to the semantics of one statement at a time):

```
<term> ::=       <variable> | <constant>
<expression> ::= <term> |
                 <term> <arithop> <term>
<arithop>    ::= + | - | * | /
<statement>  ::= <variable> := <expression> |
                 if <expression> then <statement> |
                 while <expression> do <statement>
```

      - We might define an abstract machine having a single stack plus the following stack-oriented machine language instructions:

```
PUSH    variable
PUSH    constant
POP     variable
ADD             -- pop two items, add them, push result
SUB             -- pop two items, add them, push result
MUL             -- pop two items, add them, push result
DIV             -- pop two items, add them, push result
BR      label
BEQ     label   -- pop one item; branch to address if it is 0
```

- The following rules suffice to map the higher-level constructs into this machine language:

    - Each variable is assigned its own location in the memory of the abstract machine.

    - A term maps to

      ```
              PUSH variable
      or      PUSH constant
      ```

    - The operators map to ADD, SUB, MUL, DIV respectively

    - An expression of the first kind maps to the code generated by its term

    - An expression of the second kind maps to

      ```
              Code for first term
              Code for second term
              Code for operator
      ```

    - An assignment statement maps to:

      ```
              Code for expression
              POP     variable
      ```

    - An if statement maps to

      ```
              Code for expression
              BEQ     label
              Code for statement
      label:
      ```

    - A while statement maps to

      ```
      label1: Code for expression
              BEQ label2
              Code for statement
              BR label1
      label2:
      ```

- Now, the semantics of any statement can be understood by mapping the statement to the machine language of the virtual machine and then determining the impact of the statement on the state of the virtual machine's memory.

```
Example:       while X - Y do
                   if Y - 3 then
                       X := X + 1
```

becomes

```
               L1:     PUSH    X
                       PUSH    Y
                       SUB
                       BEQ     L3
                       PUSH    Y
                       PUSH    3
                       SUB
                       BEQ     L2
                       PUSH    X
                       PUSH    1
                       ADD
                       POP     X
               L2:     BR      L1
               L3:
```

This code could now be executed by a simulator (or by hand) to determine the effect of the statement.

2. The best-known example of an operational definition of a language's semantics is the definition of PL/I, which was done using the Vienna Definition Language - an abstract machine and translator named for IBM's Vienna Laboratory, where it was developed. This provides an unambiguous definition for any PL/I statement or program - but is not particularly useful for someone who doesn't have access to the VDL software!

3. The operational approach is most useful in two areas

   a. An operational definition of the semantics of a language can help an IMPLEMENTOR of the language understand what to do. In fact, the virtual machine simulator plus the translator together constitute an implementation of the language, though probably not an efficient one.

   b. For ordinary users, the operational approach is more useful when applied to selected features of a language, possibly making use of other, more clearly understood features of the same language. For example, in the Pascal User Manual and report, the meaning of readln is defined as follows:

```
readln(f) is equivalent to

while not eoln(f) do get(f);
get(f)
```

D. The denotational approach was formulated by Dana Scott and Christopher Strachey, in an attempt to approach the issue of semantics with full mathematical rigor.

   1. In this approach, each construct of the language DENOTES some abstract mathematical entity.

      a. For example, a numeric constant denotes some number.

b. An expression or statement denotes some mathematical function that maps its inputs to its result.

2. The notation used by Scott, Strachey, and other writers in this field is quite complex and really readable only by a specialist. The following example of the semantics of binary integers will illustrate the approach:

```
N[[0]] = 0              -- the binary integer 0 denotes
                        the natural number 0
N[[1]] = 1              -- the binary integer 1 denotes
                        the natural number 1
N[[<bin-int> 0]] = 2*N[[<bin-int>]] -- a binary integer composed of
                        some binary integer followed
                        by 0 denotes the natural
                        number that is 2 times that
                        denoted by the first binary
                        integer

N[[<bin-int> 1]] = 2*N[[<bin-int>]] + 1 -- a binary integer composed of
                        some binary integer followed
                        by 1 denotes the natural
                        number that is 2 times that
                        denoted by the first binary
                        integer, plus 1
```

3. Obviously, applying this to a complete language would be a long and involved process, and the result would be of limited value to the average user or implementor. However, this could be useful to a language DESIGNER by forcing him to think rigorously about what the language means. Further, a language construct that is particularly hard to define the semantics of is one that is probably hard for humans to undertand and use well, too.

E. The Axiomatic approach to programming language semantics is closely related to work on formal proofs of program correctness. Under this approach, the meaning of a construct consists of the assertions(s) that can be made following its execution.

1. Programming language constructs thus have their meaning formalized by giving associated proof rules.

   a. Such proof rules are stated in terms of postconditions and weakest preconditions.

      - The postcondition of a statement is an assertion that one desires to have be true after the statement is executed.

      - The weakest precondition of a statement is the minimal condition that must hold before it is executed so that it will produce the desired postcondition.

        Example: suppose we want to have X = 1 after executing

        ```
        X := X + 1
        ```

        then the weakest precondition is X = 0 - giving

```
{ X = 0 }

X := X + 1

{ X = 1 }
```

b. The language is defined by associating with each construct a PREDICATE TRANSFORMER that derives the weakest precondition of a statement from the desired postcondition. For example, the predicate transformer for the assignment statement

```
V := E
```

with desired postcondition P is

P^{E}_{V} -- read P with every occurrence of V replaced by E

applying this to our previous example:

P is { X = 1 }

P^{E}_{V} is { (X+1) = 1 }

which simplifies to { X = 0 }

c. There is an important caveat in the preceeding example. The predicate transformer for assignment is valid iff the expression does not involve any functions with side effects. In fact, certain constructs like functions with side effects and goto's are particularly difficult to axiomize.

2. C.A.R. Hoare and Niklaus Wirth have developed a complete axiomatic specification of the semantics of Pascal, in a rather lengthy paper.

   a. In contrast to the other two methods of formal definition of semantics we have considered, this method is potentially of considerable usefulness to a user of the language - provided he is interested in proving his programs correct.

   b. On the other hand, this system is of lesser value to the language implementor or designer, relatively speaking.

F. When all is said and done, natural language (e.g. English) - with all its inadequacies - still turns out to be the best vehicle for describing the semantics of a language for most purposes.

1. We will, therefore, rely almost exclusively on English semantic descriptions in the rest of the course.

2. However, there are times when an ad-hoc "operational approach" is useful - testing an obscure construct using a believed-to-be-correct implementation of the language. (Of course, there's always a danger here if the implementation actually turns out to be incorrect - as I discovered with one version of Pascal when I tried to construct an

example for another lecture!