Class 7: Specifying Syntax and Scanning

SI 413 - Programming Languages and Implementation

Dr. Daniel S. Roche

United States Naval Academy

Fall 2011

Specifying a Programming Language

Programming languages provide a medium to describe an algorithm so that a computer can understand it.

But how can we describe a programming language so that a computer can understand it?

Specifying a Programming Language

Programming languages provide a medium to describe an algorithm so that a computer can understand it.

But how can we describe a programming language so that a computer can understand it?

We need to specify both:

- Syntax: the rules for how a program can look
- Semantics: the *meaning* of syntactically valid programs

Syntax vs. Semantics: English examples

Consider four English sentences:

- Burens mandneout exhastrity churerous handlockies audiverall.
- Feels under longingly shooting the darted about.
- Colorless green ideas sleep furiously.
- It's like all the big stories were stitched together into dead tiny sisters.

Syntax vs. Semantics: English examples

Consider four English sentences:

- Burens mandneout exhastrity churerous handlockies audiverall.
  Not syntatically valid — invalid words.
- Feels under longingly shooting the darted about.
- Colorless green ideas sleep furiously.
- It's like all the big stories were stitched together into dead tiny sisters.

Syntax vs. Semantics: English examples

Consider four English sentences:

- Burens mandneout exhastrity churerous handlockies audiverall.
  Not syntatically valid — invalid words.
- Feels under longingly shooting the darted about.
  Not syntatically valid — parts of speech not properly structured.
- Colorless green ideas sleep furiously.
- It's like all the big stories were stitched together into dead tiny sisters.

Syntax vs. Semantics: English examples

Consider four English sentences:

- Burens mandneout exhastrity churerous handlockies audiverall.
  Not syntatically valid — invalid words.
- Feels under longingly shooting the darted about.
  Not syntatically valid — parts of speech not properly structured.
- Colorless green ideas sleep furiously.
  Not semantically valid (thanks Noam Chomsky).
- It's like all the big stories were stitched together into dead tiny sisters.

Syntax vs. Semantics: English examples

Consider four English sentences:

- Burens mandneout exhastrity churerous handlockies audiverall.
  Not syntatically valid — invalid words.
- Feels under longingly shooting the darted about.
  Not syntatically valid — parts of speech not properly structured.
- Colorless green ideas sleep furiously.
  Not semantically valid (thanks Noam Chomsky).
- It's like all the big stories were stitched together into dead tiny sisters.
  Totally fine. (From poem by Jeffrey Harrison.)

Syntax vs. Semantics: Code examples

What do the following code fragments mean?

-
  ```
  int x;
  x = 2^3;
  ```

Syntax vs. Semantics: Code examples

What do the following code fragments mean?

-
  ```
  int x;
  x = 2^3;
  ```

-
  ```
  if (x < y < z) {
    return y;
  }
  else return 0;
  ```

Syntax feeds semantics!

Consider the following grammar:

$$
\begin{aligned}
\mathit{exp} &\rightarrow \mathit{exp}\ \mathit{op}\ \mathit{exp} \mid \texttt{NUM}\\
\mathit{op} &\rightarrow + \mid - \mid * \mid /
\end{aligned}
$$

Syntax feeds semantics!

Consider the following grammar:

$$\begin{aligned} \mathit{exp} &\rightarrow \mathit{exp}\ \mathit{op}\ \mathit{exp} \mid \texttt{NUM} \\ \mathit{op} &\rightarrow \texttt{+} \mid \texttt{-} \mid * \mid / \end{aligned}$$

This correctly defines the syntax of basic arithmetic statements with numbers. But it is *ambiguous* and confuses the semantics!

Better syntax specification

Here is an unambiguous syntax for basic arithmetic:

Terminals (i.e., *tokens*)

$$
\begin{aligned}
\texttt{OPA} &= \texttt{+} \mid \texttt{-} \\
\texttt{OPM} &= * \mid / \\
\texttt{NUM} &= \texttt{DIGIT}+ \\
\texttt{LP} &=
\end{aligned}
$$

Valid constructs (i.e., *grammar*)

$$
\begin{aligned}
\textit{exp} &\rightarrow \textit{exp}\ \texttt{OPA}\ \textit{term} \mid \textit{term} \\
\textit{term} &\rightarrow \textit{term}\ \texttt{OPM}\ \textit{factor} \mid \textit{sfactor} \\
\textit{sfactor} &\rightarrow \texttt{OPA}\ \textit{factor} \mid \textit{factor} \\
\textit{factor} &\rightarrow \texttt{NUM} \mid \texttt{LP}\ \textit{exp}\ \texttt{RP}
\end{aligned}
$$

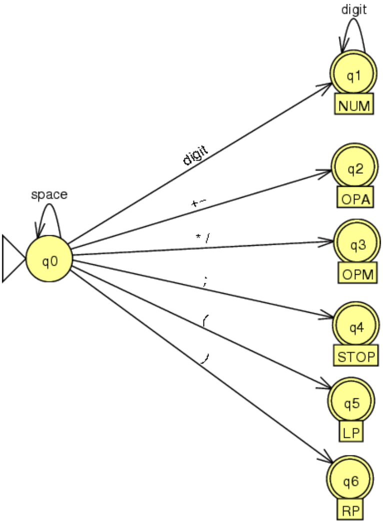
Scanner and Parser Specification

Recall that compilation begins with *scanning* and *parsing*.

- Scanning turns a raw character stream into a stream of tokens. Tokens are specified using *regular expressions*.
- Parsing finds larger syntactic constructs and turns a token stream into a parse tree. Grammar is specified in *Extended Backus-Nauer Form*. (EBNF allows the normal constructs plus Kleene +, Kleene *, and parentheses.)

Hand-rolled Scanner FA

Here is a finite automaton for our basic tokens:



What is a token?

When our FA accepts, we have a valid token.

We always want to return the terminal symbol or “type” of that token. This usually comes right from the accepting state number.

For some tokens, we may need some more information as well, such as the value of the number, or which operation was seen.

Code for hand-rolled scanner

The file `calcScanner.cpp` implements the FA above. Check it out!

This scanner is used by the Bison parser specified in `calcParser.ypp`. This contains:

- Datatype definition for the "extra" information returned with a token
- Grammar production rules, using token names as terminals
- A main method to parse from standard in

Extending our syntax

Some questions:

- What if we wanted ** to mean exponentiation?
- How about allowing comments? Single- or multi-line?
- How about strings delimited with "?
- What about delimiters?
- Can we allow negative and/or decimal numbers?

Maximal munch

How does the C++ scanner know that "`/*`" starts a comment, and is not a divide and then a multiply operator?

How does it know that "`-5`" is a single integer literal, and not the negation operator followed by the number 5?

How does it even know if "`51`" is two integers or one?

Maximal munch

How does the C++ scanner know that "`/*`" starts a comment, and is not a divide and then a multiply operator?

How does it know that "`-5`" is a single integer literal, and not the negation operator followed by the number 5?

How does it even know if "`51`" is two integers or one?

Maximal munch rule: always take the token that matches the most characters, starting from the current position.

Looking ahead

The code we referenced uses `cin.putback()` to return unneeded characters to the input stream.

But this only works for a single character. In general, we need to use a buffer. Implementing this requires a circular, dynamically-sized array, and is a bit tricky.

For example, consider the language with `-` and `-->` as valid tokens, but not `--`. This requires 2 characters of "look-ahead".