Lexical and Parse Structure of Programming Languages

CS152

Chris Pollett

Sep. 15, 2008.

Outline

- Introduction to Syntax
- Lexical Structure
- Context Free Grammars

Introduction to Syntax

- Recall there were two aspects of making a programming language machine understandable:
  - Giving it a syntax
  - Providing semantics to that syntax.
- We are now going to discuss the two main aspects of specifying the syntax of a programming language:
  - Indicating its lexical elements
  - Giving it a context free grammar
- We will then use the C we have just learned (not today, but next week probably) to talk about common tools for performing analysis of lexical elements and parsing of grammars.

Lexical Structure

- The **lexical structure** of a programming language is the structure of its **tokens**.
- For C, these tokens would include the words: if, while, do, for, functions names, etc.
- During the **scanning phase** of programming translation, the translator searches the through the program and as it finds a token it recognizes, passes it to the **parser** so that the parser can act on it to see how it matches against grammar rules.

Types of Tokens

- Reserved Words (keywords): Things like if, while, for …
  - Called reserved because they cannot be used for as one of the other kinds of tokens. I.e., we can't do things like declare a variable named "if".
- Literals (contants): Things like 42 or "hello"
- Special Symbols: ";", "<=", "+" …
- Identifiers: Things like variable names or function names.

More on Identifiers

- Some languages have a maximum length for identifiers or ignore characters after the first so many characters. For example, C64 Basic treated the first two characters of a variable as significant.
- Consider the two identifiers: foo and foobar. When tokenizing, if we see foo should we return the token? Or should we keep scanning to see if we really have foobar?
- The standard practice is to keep scanning in case we have foobar. This is called the **principle of longest substring**.
- We use **token delimeters** such as whitespace or structural entities to determine when a token ends.
- Some old languages like Fortran are so called **fixed-format** because they used to be input using things like punchcards. Rather than use the principle of longest substring, they might tokenize based on things like the tokens that have been seen so far on the line:

```
IF = 2
IF(IF. LT .  0) IF = IF  + 1
ELSE IF = IF +2
```

Specifying Tokens

- Tokens are often specified using **regular expressions**.
- Regular expressions are built out of characters together with three basic operations: concatenation, repetition, and choice (selection).
- **Concatenation** is usual indicating by justaposition two or more characters. For example, the expression ab means the character concatenated to b.
- **Repetition** is indicated using *, to indicate 0 or more occurrences of the pattern.
  - For example, a* means 0 or more a's. (aa)* means 0 or more aa's.
  - *Notice we can use parentheses in expressions*.
  - Although, it can be defined from concatenation and *, + is used to denote 1 or more occurrences of a pattern. For example, a+ is 1 or more a's.
- **Choice** is indicated with a |. So (a|b) means either the string a or the string b.
- As a more complicated example, (a|b)*c matches the tokens aabbc, c, babc, but not bca or aab.

Some Abbreviations

- We saw + for 1 or more on the last slide.
- You can indicate ranges of numbers/ letter using a hyphen: [0-9] or [a-z].
- A backslash can be used to escape some special characters \(, *, etc.
- A question mark ? can be used to indicate something is optional.
- So [0-9]+(\.[0-9]+)? Could be used to represent a floating point literal.
- Many Unix utilities use regular expressions for text searches. For example grep. We will use lex/flex for tokenizing our programming languages.

Brute Force Tokenizing

- If we weren't using a tool like lex or flex, how could we write a program which scans the input looking for tokens?
- We could make a finite automata for the regular expression (CS154 stuff) and implement that in C. This is essentially what lex and flex do.
- There would be one accepts state of the finite automata per distinct token.
- When an accept state is reached the parser is called with the given token, it computes something and we return to our automata's start state.
- To implement an a finite automata using a while loop: while(c= nextchar()){ switch(cur_state){switch(c){} }}

Context Free Grammars

- These are very much like the grammars you might have seen in grade school. For example:
  1. *sentence* --> *noun-phrase verb-phrase* .
  2. *noun-phrase* --> *article noun*
  3. *article* --> a | the.
  4. *noun* --> girl | dog.
  5. *verb-phrase* --> *verb noun-phrase.*
  6. *verb* --> sees | pets.
- We have a collection rules. The left hand side being the larger structural unit and the right hand side saying what it is made out of. Nonitalic items are tokens; italic items are structures.
- A computer program might be built from rules like: program --> subcomponent1 subcomponent2 …
- Often to denote structures we write them in < >. So would have <sentence>.
- Rather than use --> we use ::= or =. This gives one essentially BNF notation: <sentence> ::= <noun-phrase> <verb phrase> “.”

Derivations

- Let's look at how we could derive the string "the girls sees the dog" using our grammar.

  sentence => *noun-phrase verb-phrase* .

  ⇒*article noun verb-phrase* .

  ⇒the *noun verb-phrase* .

  ⇒the girl *verb-phrase* .

  ⇒the girl *verb noun-phrase* .

  ⇒the girl sees *noun-phrase* .

  ⇒the girl sees *article noun* .

  ⇒the girl sees a noun .

  ⇒the girl sees a dog .