Concrete and Abstract Syntax

CS 131: Programming Languages
January 28, 2002

Syntax

• The legal “form” or “structure” of programs
  – How sub-constructs are put together to get larger constructs
  – Correct syntax is a precondition for being a valid program.

• Syntax is frequently distinguished from semantics, which relates to the meaning of programs.

Describing Syntax

• Computer languages need precisely-defined syntax
  – Otherwise, no way to make a program portable between implementations.
  – Can use this to automate program-analysis tools

• Can use results from formal language theory
  – Regular expressions
  – Context-free languages

Formal Languages

• A formal language is a set of finite strings of symbols
• Examples:
  – the set of all English words starting with “q”
  – the set of all natural numbers written in base 10
  – the set of all valid C programs
  – the set {"", "a", "b", "ab"}
• Given a finite string, we can ask whether or not it is in a given language.
Regular Expressions

• A regular expression is a way of denoting certain languages (called regular languages)
• Notation
  – The symbol ε is a regular expression denoting the language containing only the empty string "".
  – Any other symbol a is a regular expression denoting the language containing the single string "a".

Abbreviations

• It is often convenient to make some abbreviations:

  [abcde]  The set {"a","b","c","d","e"}
  [a-z]    The set {"a","b","c","d","e"}
  [AC-ES]  The set {"A","C","D","E","S"}

  r+         r(r*)
  r?         r+rε

Regular Expressions

• Notation (continued)
  – If r₁ and r₂ are regular expressions then r₁+r₂ is a regular expression denoting the union of the languages given by r₁ and r₂.
  – If r₁ and r₂ are regular expressions then r₁r₂ is a regular expression containing all strings obtained by concatenating a string from r₁ and a string from r₂.
  – If r is a regular expression then r* is a regular expression containing all strings formed by concatenating any finite number of strings (including zero) from the language denoted by r.

R. E. Examples

• SML (non-symbolic) identifiers, which must begin with a letter, and then may have any string of letters, digits, underscores, and primes

• Ada identifiers, which must begin with a letter and then may have any string of letters, digits and underscores, with the proviso that underscores may only occur one at a time and cannot be the last character.
Big RE [from RX library]

M\[ou\]'?am\[AEae\]l-[\AEae\]-\[AEae\]{\[AEae\]}\[AEae\]h?\[AEae\]+\[\AEae\][d\[AEae\]]\[AEae\]+a\[AEae\]iy\]

Limitations of RE's

- Consider language of balanced parentheses
  \{"", ",", "(()", "())", "(()())", ...
- Regular expressions correspond to finite automata, which have finite memories.
  - These cannot count arbitrarily high
  - Hence this language cannot be described by a regular expression.
- We need to be able to require correct "bracketing" in syntax
  - e.g., parentheses, or if ... then ... else ...

BNF Grammars

- The most common way to specify a language grammar is using Backus-Naur form, or BNF.
  - This corresponds to the formal-language definition of "context-free languages".

BNF Example: Simple Arithmetic

```
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<number> ::= <number><digit> | <digit>
<exp> ::= <exp> + <exp> | <exp> - <exp> | (<exp>) | <number>
```

::= specifies an "is-a" relationship

Alternatives are separated by vertical bars.

<digit> and <number> and <exp> are called nonterminals

actual digits, +, -, (, and ) are called terminal symbols.
Production Sequences

- A production sequence is
  - a sequence of strings (which may contain both terminals and nonterminals)
  - where each string is obtained from the previous one by expanding out a single nonterminal

Example Production Sequence

\[
\begin{align*}
&<\text{number}> : = 0 | 1 | 2 | 3 | 4 \\
&\quad \quad | 5 | 6 | 7 | 8 | 9 \\
&<\text{number}> : = <\text{number}><\text{digit}> | <\text{digit}> \\
&<\exp> : = <\exp> + <\exp> | <\exp> - <\exp> \\
&\quad \quad | ( <\exp> ) | <\text{number}>
\end{align*}
\]

\[
\begin{align*}
&<\text{digit}> : = 0 | 1 | 2 | 3 | 4 \\
&\quad \quad | 5 | 6 | 7 | 8 | 9 \\
&<\text{number}> : = <\text{number}><\text{digit}> | <\text{digit}> \\
&<\exp> : = <\exp> + <\exp> | <\exp> - <\exp> \\
&\quad \quad | ( <\exp> ) | <\text{number}>
\end{align*}
\]

Representing Programs

- The programmer treats code as simply a long sequence of characters.
  - This is not an efficient representation for compilers or interpreters
  - Does not directly represent the structure of the program.
- We can retain more information by remembering why we believe the program is syntactically valid.
  - We could remember a production sequence for the program, but this is even bigger and includes information we don’t care about.
  - But, we can summarize the production sequence by using parse trees.
Parse Trees

- The parse tree for a program is a representation of a production sequence (actually, many equivalent sequences)
  - Leaves are terminals
  - Internal nodes are nonterminals
  - The children of each node are the items that replaced that nonterminal

\[
\begin{align*}
\langle P \rangle & \rightarrow \langle P \rangle \\
& \rightarrow \langle P \rangle \langle P \rangle \\
& \rightarrow ((\langle P \rangle)\langle P \rangle) \\
& \rightarrow ((\langle P \rangle)\langle P \rangle) \\
& \rightarrow (\langle P \rangle\{\}) \\
& \rightarrow (\langle P \rangle\{\}) \\
\end{align*}
\]

- Parse Tree for 2-3+5

\[
\begin{align*}
\langle \text{exp} \rangle & \ ::= \langle \text{exp} \rangle + \langle \text{exp} \rangle \\
& \quad | \langle \text{exp} \rangle - \langle \text{exp} \rangle \\
& \quad | ( \langle \text{exp} \rangle ) \\
& \quad | \langle \text{digit} \rangle \\
\end{align*}
\]

- Ambiguity for 2-3+5

\[
\begin{align*}
\langle \text{exp} \rangle & \ ::= \langle \text{exp} \rangle + \langle \text{exp} \rangle \\
& \quad | \langle \text{exp} \rangle - \langle \text{exp} \rangle \\
& \quad | ( \langle \text{exp} \rangle ) \\
& \quad | \langle \text{digit} \rangle \\
\end{align*}
\]

- An Unambiguous Grammar

Claim: Every arithmetic expression has a unique parse tree according to the following grammar.

\[
\begin{align*}
\langle \text{exp} \rangle & \ ::= \langle \text{exp} \rangle + \langle \text{term} \rangle \\
& \quad | \langle \text{exp} \rangle - \langle \text{term} \rangle \\
& \quad | ( \langle \text{exp} \rangle ) \\
\langle \text{term} \rangle & \ ::= \langle \text{term} \rangle * \langle \text{factor} \rangle \\
& \quad | \langle \text{factor} \rangle \\
\langle \text{factor} \rangle & \ ::= \{ \langle \text{exp} \rangle \} \\
& \quad | \langle \text{digit} \rangle \\
\end{align*}
\]
Concrete vs. Abstract Syntax

- Concrete syntax is an API for the language
- Can choose very different concrete syntaxes which map to the same abstract syntax

\[
\text{fun fact}(x) = \begin{cases} 
1 & \text{if } (x = 0) \\
 x \times \text{fact}(x-1) & \text{else}
\end{cases}
\]

\[
\text{(define (fact x)}
\begin{cases} 
(\text{if (eq x 0) 1 (} \times \text{ (fact (- x 1)))})
\end{cases}
\]

Parse Tree Critique

- The parse tree is a better representation of the program than character strings
  - Shows separates subexpressions
  - Shows grouping
- But it contains a lot of junk
  - Who cares whether 3 is a num or a term or a factor or an exp?
  - Do we really have to keep around the parentheses?
Abstract Syntax

• Idea: remember the bare essentials

\[
\begin{array}{c}
\text{sum} \\
\text{diff} 2 \\
4 3 \\
\text{diff} \\
2 \text{sum} 3 5
\end{array}
\]

• We can describe abstract syntax as parse trees of a BNF grammar as well:

\[
\begin{align*}
\langle e \rangle & ::= n \\
& | \text{sum}\langle e \rangle,\langle e \rangle \\
& | \text{diff}\langle e \rangle,\langle e \rangle \\
& | \text{prod}\langle e \rangle,\langle e \rangle \\
& | \text{quot}\langle e \rangle,\langle e \rangle
\end{align*}
\]

Writing Abstract Syntax

• We will frequently want to write down a piece of abstract syntax
  – but trees are tedious.
  – and \text{sum}(\text{diff}(2,3),5) is hard to read.

• Therefore we will write abstract syntax as an ordinary expressions, and the underlying tree is implicit!
  – Free to throw in parentheses as needed
  – Use conventions like \(^*\) having higher precedence than \(\pm\), and \(-\) being left-associative
  – But we're always referring to an specific tree

Lexing and Parsing

• Modern compilers usually start with a lexer and a parser
  – Lexer: breaks input into words ("tokens")
    • variables, constants, keywords, symbols, ...
  – Parser: turns tokens into tree (AST)

• Tools exist for automatically generating these from a language description.
  – Lexer needs RE's describing tokens
  – Parser needs BNF describing grammar
Abstract vs. Concrete Syntax

• Concrete Syntax
  - What the user sees
  - Concerned with programs as strings of characters
  - How to resolve ambiguities (e.g., precedence and associativity of operators)
  - Spelling of keywords, punctuation, formatting, etc.

• Abstract Syntax
  - What the compiler needs to remember
  - Concerned with programs as structured data
    - No ambiguities remaining
    - Parsing details abstracted away