Concrete and Abstract Syntax

CS 131: Programming Languages
January 31, 2001

Describing Syntax

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

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

PL 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.

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)
• Definition
  - The symbol $\emptyset$ is a regular expression denoting the empty language.
  - The symbol $\varepsilon$ 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 "$a$".

Regular Expressions (continued)

• Definition (continued)
  - If $r_1$ and $r_2$ are regular expressions then $r_1 + r_2$ is a regular expression denoting the union of the languages given by $r_1$ and $r_2$.
  - If $r_1$ and $r_2$ are regular expressions then $r_1r_2$ is a regular expression containing all strings obtained by concatenating a string from $r_1$ and a string from $r_2$.
  - 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 $r$.

Regular Expression Examples

• Set of all binary numbers
• Set of all ways to write the keyword "if" in a language that is not case-sensitive.

Abbreviations

• It is often convenient to make some abbreviations:

  - $\{a,b,c,d,e\}$ The set \{"a","b","c","d","e"\}
  - $\{a,b,...,z\}$ The set \{"a","b","...","z"\}
  - $\{A,C,...,8\}$ The set \{"A","C","...","8"\}

  $r+$
  $r(r^*)$
  $r+$"
Big RE  [from RX library]

Mcu:7am{ae}z .{(AEae)l|]+ }7[Gk2b{aeu}+1{dtz|dhx}?afjy]

Moammar Qaddafi
Mu'ammar Gadhafi
Mu'ammar Qadafi
Moammar Qadhafi
Mu'ammar Gadhafi
Mo'ammar Qadafi
Moammar El Kaddafi
Mo'ammar El Kadhafi
Moammar Kaddafi
Mu'ammar al-Qadafi
Mo'ammar al-Qadafi
Mu'ammar al-Kadafi
Mo'ammar El Kaddafi
Mo'ammar El Kadhafi
Moammar al-Kadhafi
Mo'ammar al-Kadhafi
Mu'ammar al-Qadafi
Mu'ammar al-Qadafi
Mu'ammar Muhammad Abu Minyar al-Qadhafi

More RE 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

Limitations of RE's

• Regular expressions are very useful for describing "tokens" of a language
  - keywords
  - valid variable names
  - valid constants
  - pieces of punctuation

• But, not all languages are regular.

Limitations of RE's

• Consider language of balanced parentheses
  - "", ", ()", ",()", "(()())", ...

• Regular expressions correspond to finite automata, which have finite memories.

• Hence this language cannot be described by a regular expression.

• We want to be able to require correct "bracketing" in syntax
  - e.g., parentheses and let ... in ... end
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.

BNF Production Sequence

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

<number> → <number><digit>
          → <number><digit><digit>
          → <digit><digit><digit>
          → 3<digit><digit>
          → 34<digit>
          → 345
```

BNF Example: Nested Parens

```
<P> ::= ε
    | ( <P> )
    | <P><P>
```

```
<P> → (P)
→ (P)<P>
→ (()<P>)
→ ((()))
```
Representing Programs

- So far we have considered code simply as a string of characters.
- This is not an efficient representation for a compiler or an interpreter to use
  - Also not efficient if we want to reason about the code
- A first improvement is to consider the parse trees of the code

Parse Trees

- The parse tree for a program is a representation of a production sequence.
  - Leaves are terminals
  - Internal nodes are nonterminals
  - The children of each node are the items that replaced that nonterminal

```
<exp> ::= <exp> + <exp> | <exp> - <exp> 
| ( <exp> ) | <num>
```

Parse Tree for 2-3+5

```
2
   /   |
+    -
3    5
```

Ambiguity for 2-3+5

```
2
   /   |
-    +
3    5
```

```
<exp> ::= <exp> + <exp> | <exp> - <exp> 
| ( <exp> ) | <num>
```

```
2
   /   |
-    +
3    5
```````
An Unambiguous Grammar

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

\[
\begin{align*}
\text{<exp>} &::= \text{<exp>} + \text{<term>} \\
&\quad | \text{<exp>} - \text{<term>} \\
&\quad | \text{<term>} \\
\text{<term>} &::= \text{<term>} * \text{<factor>} \\
&\quad | \text{<factor>} \\
\text{<factor>} &::= ( \text{<exp>} ) \\
&\quad | \text{<num>} \\
\end{align*}
\]

2-3+5 Unambiguously

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 remember the parentheses?
Abstract Syntax

• Idea: remember the bare essentials

• We can describe abstract syntax via BNF as well:

\[ e ::= n \mid e + e \mid e - e \mid e * e \]

• It doesn't matter that this grammar looks ambiguous because we're concerned with trees, not strings.
  - A tree can't be ambiguous
  - No parentheses required in definition

Writing Abstract Syntax

• We will frequently want to write down a piece of abstract syntax, but trees are tedious.
• Therefore we will write abstract syntax as an ordinary expression, and the underlying tree is implicit
  - We throw in parentheses as needed
  - Use conventions like * having higher precedence than +, and - being left-associative
  - But we're always referring to a tree

Lexing and Parsing

• Modern compilers usually start with a lexer and a parser
  - Lexer: breaks input into tokens
  - Parser: turns tokens into trees
• 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 tokens
    - 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

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

fun fact(x) = if (x = 0) then 1 else x*fact(x-1)

(define (fact x)
  (if (eq x 0) 1 (* x (fact (- x 1)))))

Binding and Scope

- Most language have a notions of
  - Variable binding (declaration of new variable)
  - Scope of variables (where variables can be referenced)

let val x = 3 in x + x end

- Here X is a bound variable
- The scope of x is the expression x + x
Bound Variables

- Every use of a bound variable refers to a binding
  
  ```
  let val x = 3 in x + x end
  ```

- Nested bindings of same variable called “shadowing”
  - Rule: use of variable refers to nearest enclosing binder.

```
let val x = 10 in
  in (let val x = 11 in x + x end) + x end
```

Renaming Bound Variables

- In sane languages, choices of bound variables don’t matter:
  ```
  fn(x : int) => x + 1
  fn(y : int) => y + 1
  fn(### : int) => ### + 1
  ```

```
let val x = 3 in x + x end
let val y = 3 in y + y end
let val ### = 3 in ### + ### end
```

α-conversion

- Systematic renaming of bound variables is called α-conversion
- Shadowing can then always be avoided

```
let val x = 10 in
  in (let val x = 11 in x + x end) + x end
```

```
let val x = 10 in
  in (let val y = 11 in y + y end) + x end
```

α-equivalence

- Expressions that differ only in the names of bound variables said to be α-equivalent.
- If α-conversion does not change meaning, then it is often convenient to ignore names of bound variables.
- Formally: α-equivalent expressions are considered equal/equivalent/the same/indistinguishable.
- More formally: abstract syntax is equivalence classes of expressions under α-equivalence
Free Variables

- Variables used but not bound are said to be “free”

\[
\text{let val } x = 3 \text{ in } x + y \text{ end}
\]
(Here \(x\) is bound and \(y\) is free)

\[
x + \text{let val } x = 3 \text{ in } x + x \text{ end}
\]
(Here \(x\) occurs both bound and free)

Substitution

- Replacing free variables with terms
- Written \(e[x\leftarrow e']\)

\[
(x + \text{let } x = 3 \text{ in } x + y \text{ end})[y\leftarrow z+1]
\]
= \[
(x + \text{let } x = 3 \text{ in } x + (z+1) \text{ end})
\]

\[
(x + \text{let } x = 3 \text{ in } x + y \text{ end})[x\leftarrow z+1]
\]
= \[
((z+1) + \text{let } x = 3 \text{ in } x + y \text{ end})
\]

Substitution

- Typically need "capture-avoiding" substitution.
  - Particularly if we when identifying terms up to \(\alpha\)-equivalence
  - Free vars in substituted expression must stay free.

- Then,

\[
(x + \text{let } x = 3 \text{ in } x + y \text{ end})[y\leftarrow x+1]
\]

is not \[
x + \text{let } x = 3 \text{ in } x + (x+1) \text{ end}
\]