Concrete and Abstract Syntax

CS 131: Programming Languages

September 18, 2001

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 ∅ is a regular expression denoting the empty language.
  - 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"`.

Regular Expressions

- Notation (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:

`[abcde]`	The set `{"a","b","c","d","e"}`
`[a-z]`	The set `{"a","b",...,"z"}`
`[AC-E8]`	The set `{"A","C","D","E","8"}`
r+	r(r*)
r?	r+""

More 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

Limitations of R. E.'s

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

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 `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.

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

```
<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><P>)                 → (<P><P>)
      → (()<P>)                  → (<P>())
      → (()())                   → (()())
```

Representing Programs

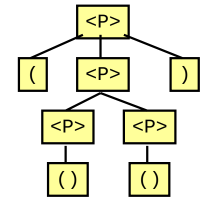
- The programmer treats code as simply a long sequence of characters.
  - This is not an efficient representation for compilers or interpreters
  - Does not directly represent the structure of the program.
- We can retain more information by remembering *why* we believe the program is syntactically valid.
  - 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

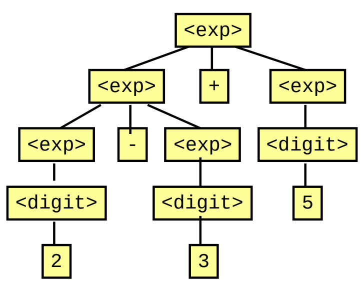
```
<P> → (<P>)
    → (<P><P>)
    → (()<P>)
    → (()())
```

```
<P> → (<P>)
    → (<P><P>)
    → (<P>())
    → (()())
```



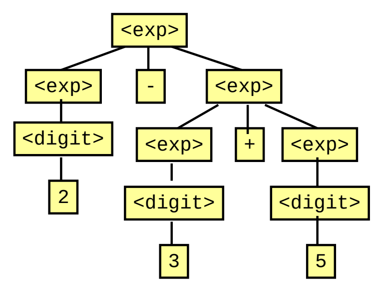
Parse Tree for 2-3+5

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



Ambiguity for 2-3+5

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

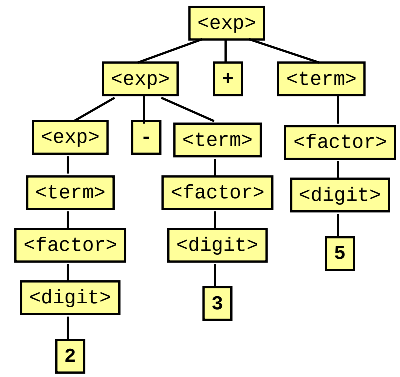


An Unambiguous Grammar

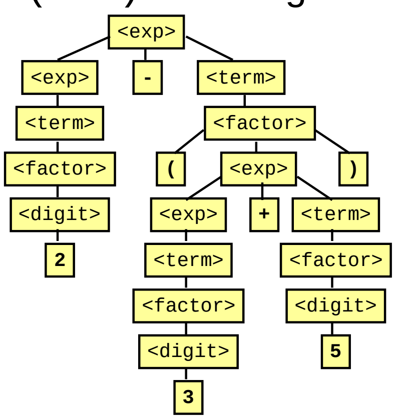
Claim: Every arithmetic expression has a unique parse tree according to the following grammar.

```
<exp>    ::= <exp> + <term>
           | <exp> - <term>
           | <term>
<term>   ::= <term> * <factor>
           | <factor>
<factor> ::= ( <exp> )
           | <digit>
```

2-3+5 Unambiguously



2-(3+5) Unambiguously

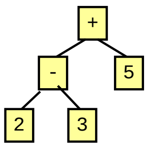


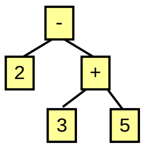
Parse Tree Critique

- 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





Abstract Syntax

- We can describe abstract syntax as parse trees in a BNF grammar as well:

  e ::= *n* | e + e | e - e | 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 an *specific* 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 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

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
```

```
let val x = 10
   in (let val x = x+1 in x + x end) + x end
```

- Nested bindings of same variable called "shadowing"
  - Usual rule: use of variable refers to nearest enclosing binder.

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 = x+1 in x + x end) + x end
```

↓

```
let val x = 10 in
   in (let val y = x+1 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"

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

(Here x is bound and y is free)

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

(Here x occurs both bound and free)

Free Variables

- Free-ness of variables is relative

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

Here x and y are free in the "body" of the let, but x is *not* free in the entire expression.

Substitution

- Replacing *free* variables with terms
- Written `e[x←e']`

```
(x + let x = 3 in x + y end)[y←z+1]
=
(x + let x = 3 in x + (z+1) end)
```

```
(x + let x = 3 in x + y end)[x←z+1]
=
((z+1) + let x = 3 in x + y end)
```

Substitution

- Unless otherwise specified, *capture-avoiding* substitution.
  - Particularly if we when identifying terms up to α-equivalence
  - Free vars in substituted expression must stay free.
- Then,

```
(y + let x = 3 in x + y end)[y←x+1]
```

is *not*

```
(x+1) + let x = 3 in x + (x+1) end
```

Substitution

```
(x + let x = 3 in x + y end)[y←x+1]
=
(x + let z = 3 in z + y end)[y←x+1]
=
x + let z = 3 in z + (x+1) end
```