Concrete and Abstract Syntax
Syntax

• Most obvious feature of a language
  - How characters make up “tokens”
    • Keywords
    • Identifiers
    • Punctuation
  - How tokens make up program phrases
Lexing

• Breaking up characters into tokens
  - Automatic lexer generators: lex, flex
  - Generally tokens described via regexps

  • e.g., SML identifiers
    
    $((\text{[A-Z]}+\text{[a-z]})(\text{[A-Z]}+\text{[a-z]}+_\d+\d^\d)*$  
    $+$ 
    $[!\%\&\$\#+-/:<>?@\~^\d]*[!\%\$\#+-/:<=>?@\~^\d]*$  

  • e.g., [Tt][Hh][Ee][Nn] if keyword case does not matter
Lexing

if len >= 3 then
  len
else
  3

IF, ID(len), GE, NUM(3),
THEN, ID(len), ELSE, NUM(3)
Parsing

• Construction of program phrases out of the stream of tokens, generally in tree form.
  – Automatic parser generators: YACC, Bison, ...
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, etc.

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

\[
\begin{align*}
<\text{exp}> & ::= <\text{exp}> + <\text{term}> \\
& \quad | <\text{exp}> - <\text{term}> \\
& \quad | <\text{term}> \\
<\text{term}> & ::= <\text{term}> * <\text{factor}> \\
& \quad | <\text{term}> / <\text{factor}> \\
& \quad | <\text{factor}> \\
<\text{factor}> & ::= ( <\text{exp}> ) \\
& \quad | <\text{variable}> \\
& \quad | <\text{number}>
\end{align*}
\]
Abstract Syntax for Expressions

\[ E ::= E + E | E - E | E \times E | E / E | n | x \]

- Ambiguous grammar for parsing strings
- But at this point only care about trees!

- In practice, we will write abstract syntax as strings, but there’s always a unique tree implied.
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 (= 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)

• \texttt{let val x = 3 in x + x end}
  - \(x\) 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 = 11 in x + x end) + x
  end

• Nested bindings of same variable called “shadowing”
  - General rule: use of variable refers to nearest enclosing binder.
Renaming Bound Variables

• In sane languages, choices of bound variables don’t matter:

```plaintext
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
```
\(\alpha\)-conversion

- Systematic renaming of bound variables is called \(\alpha\)-conversion.
- Shadowing can then always be avoided.

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

let val x = 10 in
  (let val y = 11 in y + y end) + x
end
```
\( \alpha \)-equivalence

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

• Can always assume terms have no shadowing
• Can always assume bound variables in a term are different from some other finite set.
  – We will return to this point when discussing substitution.
Implementation Consequences

• There are at least three main ways to deal with bound variables in an implementation
  - Represent the term with a specific choice of bound variables, but do systematic renaming when necessary
  - Represent the term as a graph: uses of a variable "point" to to binding site they refer to
    • bound variables as generalized pronouns
  - Use deBruijn indices
deBruijn Indices

• Observation: scopes of variables are nested.
• deBruijn index of a variable: how many levels out the variable is bound.
  - Uniquely identifies variable
  - Never need to give the variable a name
deBruijn Examples

```
let x=10 in (let y = 5 in x + y)
```

```
let 10 in (let 5 in <#1> + <#0>)
```
deBruijn Examples

```
let x=10 in ((let y = 5 in x + y) + x)
```

```
let 10 in ((let 5 in <#1> + <#0>) + <#0>
```
Free Variables

• Variables used but not bound are said to be “free”
  - \( \text{let val } x = 3 \text{ in } x + y \text{ end} \)
  • \( x \) is bound, \( y \) is free.
Substitution

• Replacing variables with terms

\[ e[x\leftarrow e'] \]

\[
(x + (let\ x = 3\ in\ x + y))[y\leftarrow z+1] = (x + (let\ x = 3\ in\ x + (z+1)))
\]
Substitution

• Substitution affects only free occurrences of a variable

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

• Usually, want "capture-avoiding" substitution.
  - Particularly if we when identifying terms up to $\alpha$-equivalence

• Then,

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

  is not

  \[x + (\text{let } x = 3 \text{ in } x + (x+1))\]
Substitution

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

\[=\]

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

\[=\]

\[x + (\text{let } z = 3 \text{ in } z + (x+1))\]
Abstract Syntax Example

e ::= n      (* integer constant *)
    | x      (* variable *)
    | e + e
    | fn x => e
    | e(e)

Define FV and substitution