Two Problems

- **Type checking**
  - Question: Does exp. e have type \( \tau \) in env \( \Gamma \)?
  - Answer: Yes / No
  - Method: Type derivation

- **Typability**
  - Question: Does exp. e have some type in env. \( \Gamma \)? If so, what is it?
  - Answer: Type \( \tau \) / error
  - Method: Type inference

### Type Inference - Outline

- Begin by assigning a **type variable** as the type of the whole expression
- **Decompose the expression** into component expressions
- **Use typing rules to generate constraints** on components and whole
- **Recursively find substitution** that solves typing judgment of first subcomponent
- **Apply substitution to next subcomponent** and find substitution solving it; compose with first, etc.
- **Apply composition of all substitutions** to original type variable to get answer

### Type Inference - Example

**What type can we give to**

\[
(\text{fun } x \to \text{fun } f \to f(f\ x))
\]

- Start with a type variable and then look at the way the term is constructed

---

**Type Inference - Example**

- **First approximate:** Give type to full expr
  
  \[
  \{ \} \vdash (\text{fun } x \to \text{fun } f \to f(f\ x)) : \alpha
  \]

- **Second approximate:** use fun rule
  
  \[
  \{ x : \beta \} \vdash (\text{fun } f \to f(f\ x)) : \gamma
  
  \{ \} \vdash (\text{fun } x \to \text{fun } f \to f(f\ x)) : \alpha
  \]

- **Remember constraint** \( \alpha \equiv (\beta \to \gamma) \)

---

**Type Inference - Example**

- **Third approximate:** use fun rule
  
  \[
  \begin{align*}
  &\{ x : \beta \} \vdash f(f\ x) : \epsilon \\
  &\{ \} \vdash (\text{fun } f \to f(f\ x)) : \gamma \\
  &\{ \} \vdash (\text{fun } x \to \text{fun } f \to f(f\ x)) : \alpha
  \end{align*}
  \]

- \( \alpha = (\beta \to \gamma); \ \gamma = (\delta \to \epsilon) \)
Type Inference - Example

- Fourth approximate: use app rule

\[
\{ f : \delta ; x : \beta \} |- f : \varphi \rightarrow \varepsilon \quad \{ f : \delta ; x : \beta \} |- f \times : \varphi
\]

\[
{ f : \delta ; x : \beta } |- (f (f x)) : \varepsilon
\]

\[
{x : \beta } |- (\text{fun } f \rightarrow f (f x)) : \gamma
\]

\[
\{ \} |- (\text{fun } x \rightarrow \text{fun } f \rightarrow f (f x)) : \alpha
\]

\[
\alpha \equiv (\beta \rightarrow \gamma) ; \gamma \equiv (\delta \rightarrow \varepsilon)
\]

Type Inference - Example

- Fifth approximate: use var rule, get constraint \( \delta \equiv \varphi \rightarrow \varepsilon \), Solve with same

\[
\{ f : \delta ; x : \beta \} |- f : \varphi \rightarrow \varepsilon \quad \{ f : \delta ; x : \beta \} |- f x : \varphi
\]

\[
{ f : \delta ; x : \beta } |- (f (f x)) : \varepsilon
\]

\[
{x : \beta } |- (\text{fun } f \rightarrow f (f x)) : \gamma
\]

\[
\{ \} |- (\text{fun } x \rightarrow \text{fun } f \rightarrow f (f x)) : \alpha
\]

\[
\alpha \equiv (\beta \rightarrow \gamma) ; \gamma \equiv (\delta \rightarrow \varepsilon)
\]
Type Inference - Example

- **Current subst:** \{\varepsilon, \beta, \delta \equiv \rightarrow \varepsilon\}
- **Apply to next sub-proof**

  (done) ... \{f : \varepsilon \rightarrow \beta; \beta : \varepsilon\} - \beta : \varepsilon

  ... \{f : \delta ; \beta : \beta\} - f x : \varepsilon

  \{x : \beta\} - (f f x) : \gamma

  \{\} - (fun x -> fun f -> f (f x)) : \alpha

  \alpha \equiv (\beta \rightarrow \gamma); \gamma \equiv (\delta \rightarrow \varepsilon)

10/16/2018
Type Inference - Example

- Current subst:
  \[
  \{ \gamma \equiv ((\beta \to \gamma), \varepsilon \equiv \beta, \zeta \equiv \beta, \varphi \equiv \beta, \delta \equiv \beta \to \beta) \}
  \]
- Solves subproof; return one layer

\[
\begin{array}{l}
\{ f : \delta ; x : \beta \} \mid (f (f x)) : \varepsilon \\
\{ x : \beta \} \mid (\text{fun } x \to f (f x)) : \gamma \\
\{ \} \mid (\text{fun } x \to \text{fun } f \to f (f x)) : \alpha
\end{array}
\]

- \( \alpha \equiv (\beta \to \gamma); \gamma \equiv (\delta \to \varepsilon) \)

Type Inference Algorithm

Let \( \text{infer } (\Gamma, e, \tau) = \sigma \)

- \( \Gamma \) is a typing environment (giving polymorphic types to expression variables)
- \( e \) is an expression
- \( \tau \) is a type (with type variables)
- \( \sigma \) is a substitution of types for type variables
- Idea: \( \sigma \) represents the constraints on type variables necessary for \( \Gamma \mid e : \tau \)
- Should have \( \sigma(\Gamma) \mid e : \sigma(\tau) \) valid

- Slight abuse of notation: \( \sigma(\Gamma) \) is substitution \( \sigma \) applied to all terms in the environment \( \Gamma = \{ x : \tau \ldots \} \) (i.e., \( \sigma(\Gamma) = \{ x : \sigma(\tau) \ldots \} \)).

Type Inference - Example

- Current subst:
  \[
  \{ \alpha \equiv ((\beta \to (\beta \to \beta)) \to \beta), \varepsilon \equiv \beta, \zeta \equiv \beta, \varphi \equiv \beta, \delta \equiv \beta \to \beta \}
  \]
- Solves subproof; return on layer

\[
\begin{array}{l}
\{ x : \beta \} \mid (\text{fun } x \to f (f x)) : \gamma \\
\{ \} \mid (\text{fun } x \to \text{fun } f \to f (f x)) : \alpha
\end{array}
\]

Type Inference Algorithm (All in one!)

\[
\begin{array}{l}
\text{infer } (\Gamma, e, \tau) = \sigma \\
\text{Case } \text{exp of} \\
\quad \text{Var } x \rightarrow \text{return } \text{Uniform}(e(x), \tau) \\
\quad \text{Const } c \rightarrow \text{return } \text{Uniform}(c, \tau) \\
\quad \text{App } (\text{fun } \alpha) \rightarrow \text{return } \text{Uniform}(\alpha, \tau) \\
\quad \text{Fun } \Gamma \mid e : \tau \to \sigma \\
\quad \text{Let } (\alpha, \beta) \rightarrow \text{return } \text{Uniform}(\alpha(\beta), \tau) \\
\quad \text{Return } \alpha(\beta) \rightarrow \text{return } \text{Uniform}(\alpha(\beta), \tau) \\
\end{array}
\]
Type Inference Algorithm

\[
\text{infer } (\Gamma, \text{exp}, \tau) =
\]

- Case \(\text{exp}\) of
  - Var \(v\) --> return \(\text{Unify}(\tau = \text{freshInstance}(\Gamma(v)))\)
    - Replace all quantified type vars by fresh ones
  - Const \(c\) --> return \(\text{Unify}(\tau = \text{freshInstance } \varnothing)\)
    where \(\Gamma \vdash c : \varnothing\) by the constant rules
  - Fun \(x \to e\) -->
    - Let \(\alpha, \beta\) be fresh variables
    - Let \(\sigma = \text{infer } ((\lambda : \alpha) + \Gamma, e, \beta)\)
    - Return \(\text{Unify}((\sigma(\tau) = \sigma(\alpha \to \beta))) \circ \sigma\)

Inference Example (Repeat)

- Fifth approximate: use var rule, get constraint \(\delta = \varnothing \to \varnothing\), Solve with same

- Apply to next sub-proof

\[
\{f : \delta; x : \beta\} \vdash f : \varnothing \to \varnothing
\]

Inference Example (Repeat)

- What do we do here?

\[
\{f : \forall \delta, \delta \to \delta ; x : \beta\} \vdash f : \varnothing \to \varnothing
\]

- And here?

\[
\{f : \forall \epsilon, \epsilon \to \epsilon ; x : \beta\} \vdash f : \varnothing \to \varnothing
\]

Inference Example (Repeat)

- Third approximate: use fun rule

\[
\{f : \exists \delta. \delta \to \delta\} \vdash f : \varnothing \to \varnothing
\]

\[
\{\} \vdash (\text{fun } x \to f(f x)) : \epsilon
\]

- \(\alpha \equiv (\beta \to \gamma)\);

Type Inference Algorithm (cont)

- Case \(\text{exp}\) of
  - App \((e_1, e_2)\) -->
    - Let \(\alpha\) be a fresh variable
    - Let \(\sigma_1 = \text{infer } (\Gamma, e_1, \alpha \to \tau)\)
    - Let \(\sigma_2 = \text{infer } (\sigma_1(\Gamma), e_2, \sigma_1(\alpha))\)
    - Return \(\sigma_2 \circ \sigma_1\)

Inference Example (Repeat)

- Fourth approximate: use app rule

\[
\{f : \delta; x : \beta\} \vdash f : \varnothing \to \varnothing\quad \{f : \delta; x : \beta\} \vdash f : \varnothing \to \varnothing
\]

\[
\{f : \delta \to \delta ; x : \beta\} \vdash f(f x) : \epsilon
\]
Type Inference Algorithm (cont)

- **Case** `exp` of
  - If `e_1` then `e_2` else `e_3` -->
    - Let `σ_1 = infer(Γ, e_1, bool)`
    - Let `σ_2 = infer(σ_1(Γ), e_2, σ_1(τ))`  
    - Let `σ_3 = infer(σ_2 o σ_1(Γ), e_2, σ_2 o σ(τ))`  
    - Return `σ_3 o σ_2 o σ_1`


- **Case** `exp` of
  - Let `x = e_1` in `e_2` -->
    - Let `α` be a fresh variable
    - Let `σ_1 = infer(Γ, e_1, α)`  
    - Let `σ_2 = infer(Δ:GEN(σ_1(α), σ_1(Γ)))`  
      + `σ_1(Γ), e_2, σ_1(τ))`  
    - Return `σ_2 o σ_1`


Reminder: Type Terms

- **Terms** made from constructors and variables

  **Reminder:**
  - Monomorphic Types (τ):
    - Basic Types: int, bool, float, string, unit, ...
    - Type Variables: α, β, γ, δ, ε
    - Compound Types: α → β, int * string, bool list, ...
  - Polymorphic Types:
    - Monomorphic types τ  
    - Universally quantified monomorphic types  
      ∀τ_1,...,τ_n: τ  
    - Can think of τ as same as ∀τ: τ


- **To infer a type**, introduce **type_of**
  - Let `α` be a fresh variable  
  - `type_of(Γ, e) =  
    let `α` be a fresh variable in  
    let `σ = infer(Γ, e, α)`  
    in `σ(α)`

  - Need substitution!
  - Need an algorithm for **Unif**!
Substitution Implementation

type term = Variable of string |
| Constructor of (string * term list)

let rec subst var_name residue term =
match term with
| Variable name ->
| if var_name = name then residue else term
| Constructor (c, tys) ->
| let newt = List.map (subst var_name residue) tys in Constructor (c, newt);;

Unification Problem

Given a set of pairs of terms ("equations")
\{(s_1, t_1), (s_2, t_2), \ldots, (s_n, t_n)\} *
(the unification problem) does there exist
a substitution \(\sigma\) (the unification solution)
of terms for variables such that
\(\sigma(s_i)\) is the same as \(\sigma(t_i)\),
for all \(i = 1, \ldots, n\)?

- Think of these pairs as \{("s_1\ =\ t_1"), ("s_2\ =\ t_2"), \ldots, ("s_n\ =\ t_n")\}
- This is the notation we're going to use in the example

Uses for Unification

- Type Inference and type checking
- Pattern matching as in OCaml
  - Can use a simplified version of algorithm
- Logic Programming - Prolog
- Simple parsing

Unification Algorithm

- Let \(S = \{(s_1 = t_1), (s_2 = t_2), \ldots, (s_n = t_n)\}\) be a unification problem.
- Unif(S) returns a substitution

  - Case \(S = \{\}\) : Unif(S) = Identity function
    - (i.e., no substitution)

  - Case \(S = \{(s = t)\} \cup S'\) : Four main steps
    - Delete, Decompose, Orient, Eliminate

Unification Algorithm for \(S = \{(s = t)\} \cup S'\)

- **Delete**: if \(s\) is \(t\) (\(s\) and \(t\) are the same term) then
  \(\text{Unif}(S) = \text{Unif}(S')\)
- **Decompose**: if \(s\) is \(f(q_1, \ldots, q_m)\) and \(t\) is \(f(r_1, \ldots, r_m)\)
  (same \(f\), same \(m\)), then
  \(\text{Unif}(S) = \text{Unif}(((q_1 = r_1), \ldots, (q_m = r_m)) \cup S')\)
- **Orient**: if \(t\) is \(x\) (a variable), and \(s\) is not a variable,
  \(\text{Unif}(S) = \text{Unif} (((x = s)) \cup S')\)

Unification Algorithm for \(S = \{(s = t)\} \cup S'\)

- **Eliminate**: if \(s\) is \(x\) (a variable), and \(x\) does not occur in \(t\) (use "occurs (x, t)" check!) then
  - Let \(\psi = \{x \rightarrow t\}\)
  - Let \(\psi = \text{Unif}(\psi(S'))\)
  - \(\text{Unif}(S) = \{x \rightarrow \psi(t)\} \circ \psi\)

  - Be careful when composing substitutions:
    - \(\{x \rightarrow a\} \circ \{y \rightarrow b\} = \{y \rightarrow ((x \rightarrow a)(b))\} \circ \{x \rightarrow a\}\) if \(y\) not in \(a\)
Tricks for Efficient Unification

- Don’t return substitution, rather do it incrementally
- Make substitution be constant time
  - Requires implementation of terms to use mutable structures (or possibly lazy structures)
  - We won’t discuss these

Example

- \( x,y,z \) variables, \( f,g \) constructors

| Unify \( \{(f(x) = f(g(f(z),y))), (g(y,y) = x)\} = ? \)

Example

- \( x,y,z \) variables, \( f,g \) constructors

| Unify \( \{(f(x) = f(g(f(z),y))), (g(y,y) = x)\} = ? \)

Example

- \( x,y,z \) variables, \( f,g \) constructors

Pick a pair: \( (g(y,y) = x) \)

| Unify \( \{(f(x) = f(g(f(z),y))), (g(y,y) = x)\} = ? \) by Orient

Example

- \( x,y,z \) variables, \( f,g \) constructors

Pick a pair: \( (g(y,y) = x) \)

| Unify \( \{(f(x) = f(g(f(z),y))), (g(y,y) = x)\} = ? \) by Orient

Example

| Unify \( \{(f(x) = f(g(f(z),y))), (g(y,y) = x)\} = ? \) by Orient
Example

- x,y,z variables, f,g constructors

- Unify \{ (f(x) = f(g(f(z), y))), (x = g(y,y)) \} = ?

Example

- x,y,z variables, f,g constructors

- \{ (f(x) = f(g(f(z), y))), (x = g(y,y)) \} is non-empty

- Unify \{ (f(x) = f(g(f(z), y))), (x = g(y,y)) \} = ?

Example

- x,y,z variables, f,g constructors

- Pick a pair: (x = g(y,y))

- Unify \{ (f(f(x)) = f(g(f(z), y))), (x = g(y,y)) \} = ?

Example

- x,y,z variables, f,g constructors

- Pick a pair: (x = g(y,y))

- Eliminate x with substitution \{ x \rightarrow g(y,y) \}

- Check: x not in g(y,y)

- Unify \{ (f(x) = f(g(f(z), y))), (x = g(y,y)) \} = ?

Example

- x,y,z variables, f,g constructors

- Pick a pair: (x = g(y,y))

- Eliminate x with substitution \{ x \rightarrow g(y,y) \}

- Check: x not in g(y,y)

- Unify \{ (f(f(g(y,y))) = f(g(f(z), y))) \}

- \{ x \rightarrow g(y,y) \} = ?
Example

- **x, y, z** variables, **f, g** constructors
- {f(g(y,y)) = f(g(f(z),y))} is non-empty

Unify {f(g(y,y)) = f(g(f(z),y))}

  o {x → g(y,y)} = ?

Example

- **x, y, z** variables, **f, g** constructors
- Pick a pair: (f(g(y,y)) = f(g(f(z),y)))
- Decompose: (f(g(y,y)) = f(g(f(z),y))) becomes
  {g(y,y) = g(f(z),y)}

Unify {f(g(y,y)) = f(g(f(z),y))}

  o {x → g(y,y)} = ?

Example

- **x, y, z** variables, **f, g** constructors
- (g(y,y) = g(f(z),y)) is non-empty

Unify {g(y,y) = g(f(z),y)}

  o {x → g(y,y)} = ?

Example

- **x, y, z** variables, **f, g** constructors
- Pick a pair: (g(y,y) = g(f(z),y))
- Decompose: (g(y,y) = g(f(z),y)) becomes
  {y = f(z); y = y}

Unify {g(y,y) = g(f(z),y)}

  o {x → g(y,y)} = ?

Unify {y = f(z); y = y} o {x → g(y,y)}
Example

- $x, y, z$ variables, $f, g$ constructors

- Unify $\{(y = f(z)); (y = y)\} \circ \{x \rightarrow g(y, y)\} = ?$

Example

- $x, y, z$ variables, $f, g$ constructors

- $\{(y = f(z)); (y = y)\} \circ \{x \rightarrow g(y, y)\}$ is non-empty

- Unify $\{(y = f(z)); (y = y)\} \circ \{x \rightarrow g(y, y)\} = ?$

Example

- $x, y, z$ variables, $f, g$ constructors

- Pick a pair: $(y = f(z))$

- Unify $\{(y = f(z)); (y = y)\} \circ \{x \rightarrow g(y, y)\} = ?$

Example

- $x, y, z$ variables, $f, g$ constructors

- Pick a pair: $(y = f(z))$

- Eliminate $y$ with $\{y \rightarrow f(z)\}$

- Unify $\{(f(z) = f(z))\} \circ \{y \rightarrow f(z)\} \circ \{x \rightarrow g(y, y)\} = \text{Unify } \{(f(z) = f(z))\}$

- $\circ \{y \rightarrow f(z)\} \circ \{x \rightarrow g(y, y)\} = \text{Unify } \{(f(z) = f(z))\}$

- $\circ \{y \rightarrow f(z)\}; \{x \rightarrow g((f(z), f(z)))\}$

Example

- $x, y, z$ variables, $f, g$ constructors

- Unify $\{(f(z) = f(z))\}$

- $\circ \{y \rightarrow f(z); x \rightarrow g((f(z), f(z)))\} = ?$

Example

- $x, y, z$ variables, $f, g$ constructors

- $\{(f(z) = f(z))\}$ is non-empty

- Unify $\{(f(z) = f(z))\}$

- $\circ \{y \rightarrow f(z); x \rightarrow g((f(z), f(z)))\} = ?$
Example

- $x, y, z$ variables, $f, g$ constructors

- Pick a pair: $(f(z) = f(z))$

- Unify \{$(f(z) = f(z))$\}
  \(\circ\) $(y \rightarrow f(z); x \rightarrow g(f(z), f(z)))$ = ?

Example

- $x, y, z$ variables, $f, g$ constructors

- Pick a pair: $(f(z) = f(z))$

- Delete

- Unify \{$(f(z) = f(z))$\}
  \(\circ\) $(y \rightarrow f(z); x \rightarrow g(f(z), f(z)))$ =

  Unify $\{} \circ \{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\}$

Example

- $x, y, z$ variables, $f, g$ constructors

- Unify $\{}$

- Unify $\{} \circ \{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\}$ =

  $\{} \circ \{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\}$

Example

- $x, y, z$ variables, $f, g$ constructors

- $\{}$ is empty

- Unify $\{}$ = identity function

- Unify $\{} \circ \{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\}$ =

  $\{} \circ \{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\}$

Example

- Unify \{(f(x) = f(g(f(f(z), y))), (g(y, y) = x))\} =

  $\{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\}$

  \begin{align*}
  f(\times) &= f(g(f(z), y)) \\
  &= f(g(f(f(z), f(z)))) = f(g(f(z), f(z)))
  \\
  g(y, y) &= \times \\
  &= g(f(z), f(z)) = g(f(z), f(z))
  \end{align*}

Example

- Unify \{(f(x) = f(g(f(f(z), y))), (g(y, y) = x))\} =

  $\{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\}$

  $y \rightarrow \text{int list}, x \rightarrow \text{(int list * int list)}$

  \begin{align*}
  f(\times) &= f(g(f(z), y)) \\
  &= f(g(f(f(z), f(z)))) = f(g(f(z), f(z)))
  \\
  (\text{int list * int list}) \text{ list} &= (\text{int list * int list}) \text{ list}
  \\
  g(y, y) &= \times \\
  &= g(f(z), f(z)) = g(f(z), f(z))
  \end{align*}
Example of Failure: Decompose

- Unify\((f(x,g(y)) = f(h(y),x))\)
  \[\text{Decompose:} \quad (f(x,g(y)) = f(h(y),x))\]
  - Unify \((x = h(y)), \ (g(y) = x))\)
  - Orient: \((g(y) = x)\)
  - Unify \((x = h(y)), \ (x = g(y)))\)
  - Eliminate: \((x = h(y))\)
  - Unify \((h(y) = g(y))) \circ \{x \rightarrow h(y)\}

- No rule to apply! Decompose fails!

Example of Failure: Occurs Check

- Unify\((f(x,g(x)) = f(h(x),x))\)
  \[\text{Decompose:} \quad (f(x,g(x)) = f(h(x),x))\]
  - Unify \((x = h(x)), \ (g(x) = x))\)
  - Orient: \((g(x) = x)\)
  - No rules apply.

Course Objectives

- New programming paradigm
  - Functional programming
  - Environments and Closures
  - Patterns of Recursion
  - Continuation Passing Style
- Phases of an interpreter / compiler
  - Lexing and parsing
  - Type systems
  - Interpretation
- Programming Language Semantics
  - Lambda Calculus
  - Operational Semantics
  - Axiomatic Semantics

Major Phases of a Compiler

| Source Program | \(\text{Lex}\) | Tokens |
| Source Program | \(\text{Parse}\) | Abstract Syntax |
| \(\text{Semantic Analysis}\) | Environment |
| Translate | Intermediate Representation |
| \(\text{Optimized IR}\) | \(\text{Instruction Selection}\) |
| \(\text{Unoptimized Machine-Specific Assembly Language}\) | \(\text{Instruction Optimize}\) |
| Optimized Machine-Specific Assembly Language | Emit code |
| Assembly Language | \(\text{Assembler}\) |
| Relocatable Object Code | \(\text{Linker}\) |
| Machine Code |

Modified from "Modern Compiler Implementation in ML", by Andrew Appel

Programming Languages & Compilers

Three Main Topics of the Course

<table>
<thead>
<tr>
<th>I</th>
<th>II</th>
<th>III</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Programming Paradigm</td>
<td>Language Translation</td>
<td>Language Semantics</td>
</tr>
</tbody>
</table>

Programming Languages & Compilers

Order of Evaluation

I
II
III

New Programming Paradigm
Language Translation
Language Semantics

Specification to Implementation
Language Syntax

- Syntax is the description of which strings of symbols are meaningful expressions in a language
- It takes more than syntax to understand a language; need meaning (semantics) too
- Syntax is the entry point

Syntax of English Language

- Pattern 1
  - David sings
  - The dog barked
  - Susan yawned

- Pattern 2
<table>
<thead>
<tr>
<th>Subject</th>
<th>Verb</th>
<th>Direct Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>David</td>
<td>sings</td>
<td>ballads</td>
</tr>
<tr>
<td>The professor</td>
<td>wants</td>
<td>to retire</td>
</tr>
<tr>
<td>The jury</td>
<td>found</td>
<td>the defendant guilty</td>
</tr>
</tbody>
</table>
Elements of Syntax

- Character set – previously always ASCII, now often 64 character sets
- Keywords – usually reserved
- Special constants – cannot be assigned to
- Identifiers – can be assigned to
- Operator symbols
- Delimiters (parenthesis, braces, brackets)
- Blanks (aka white space)

Expressions

if ... then begin ... ; ... end else begin ... ; ... end

Type expressions

typexpr₁ -> typexpr₂

Declarations (in functional languages)

let pattern₁ = expr₁ in expr

Statements (in imperative languages)

a = b + c

Subprograms

let pattern₁ = let rec inner = ... in expr

Lexing and Parsing

- Converting strings to abstract syntax trees done in two phases
  - Lexing: Converting string (or streams of characters) into lists (or streams) of tokens (the “words” of the language)
    - Specification Technique: Regular Expressions
  - Parsing: Convert a list of tokens into an abstract syntax tree
    - Specification Technique: BNF Grammars

Formal Language Descriptions

- Regular expressions, regular grammars, finite state automata
- Context-free grammars, BNF grammars, syntax diagrams
- Whole family more of grammars and automata – covered in automata theory

Grammars

- Grammars are formal descriptions of which strings over a given character set are in a particular language
- Language designers write grammar
- Language implementers use grammar to know what programs to accept
- Language users use grammar to know how to write legitimate programs