# Lecture 5

# Case study: lexical analysis

- **Lexical analysis converts**
  - **sequences of characters**
  - **into**
  - **sequences of tokens**
  - **tokens are also called words or lexemes**

- **For us, a token will be one of:**
  - **a number**
    **(sequence of digits)**
  - **an identifier**
    **(sequence of letters or digits starting with a letter)**
  - **a 'special symbol' such as `+`, `*`, `<`, `==>` or `++`**
    - **special symbols are specified by a table – see later**

# Numbers and letters

- **A number is a sequence of digits**

- **<= is overloaded and can be applied to strings**

  - **suppose $x$ and $y$ are single-character strings**

  - **then $x$<=$y$ just tests whether the ASCII code of $x$ is less then or equal to the ASCII code of $y$**

```
fun IsDigit x = "0" <= x andalso x <= "9";
> val IsDigit = fn : string -> bool
```

- **ASCII codes of lower case letters are adjacent**

- **ASCII codes of upper case letters are adjacent**

```
fun IsLetter x =
 ("a" <= x andalso x <= "z") orelse
 ("A" <= x andalso x <= "Z");
> val IsLetter = fn : string -> bool
```

# Separators

- **Separators are spaces, newlines and tabs**

```
fun IsSeparator x =
 (x = " " orelse x = "\n" orelse x = "\t");
> val IsSeparator = fn : string -> bool
```

- **Characters that are not digits, letters or separators are assumed to be special symbols**
  - **multi-character special symbols are considered later**

- **Input a list of single-charater strings**
  - **lexical analysis converts input to a token list**

# Special case: only numbers

- **Suppose input just consists of numbers separated by separators**

- **Lexical analysis for just this case needs to:**
  - **repeatedly remove digits until a non-number is reached**
  - **then implode the removed characters into a token**
  - **and add that to the list of tokens**

- `GetNumber` **takes a list, $l$ say, of single-character strings and returns a pair consisting of**
  - **a string representing a number consisting of all the digits in $l$ up to the first non-digit**
  - **the remainder of $l$ after these digits have been removed**

- `GetNum` **uses an auxiliary function** `GetNumAux`
  - `GetNumAux` **has an extra argument** `buf` **for accumulating a (reversed) list of characters making up the number**

# GetNumAux and GetNum

```
fun GetNumAux buf [] = (implode(rev buf), [])
 |  GetNumAux buf (l as (x::l')) =
     if IsDigit x then GetNumAux (x::buf) l'
                  else (implode(rev buf),l);
> val GetNumAux =
>  fn
>   : string list -> string list -> string * string list

GetNumAux ["a","b","c"] ["1","2","3"," ","4","5"];
> val it =
>  ("cba123",[" ","4","5"]) : string * string list
```

- **Then `GetNum` is simply defined by:**

```
val GetNum = GetNumAux [];
> val GetNum = fn : string list -> string * string list

GetNum ["1","2","3"," ","4","5"];
> val it = ("123",[" ","4","5"]) : string * string list

GetNum ["a","0","1"];
> val it = ("",["a","0","1"]) : string * string list
```

- **Anomalous return of `""` fixed later**
- **Could localise definition of `GetNumAux` using `local` $\cdots$ `in` $\cdots$ `end`**

# Special case: only identifiers

- **Analysis of identifiers similar to numbers**

```
fun GetIdentAux buf [] = (implode(rev buf), [])
 |  GetIdentAux buf (l as (x::l')) =
     if IsLetter x orelse IsDigit x
      then GetIdentAux (x::buf) l'
      else (implode(rev buf),l);
> val GetIdentAux =
>  fn
>  : string list -> string list -> string * string list

GetIdentAux ["a","b","c"]
            ["e","f","g","4","5"," ","6","7"];
> val it =
>  ("cbaefg45",[" ","6","7"]) : string * string list
```

- **An identifier must start with a letter**

```
exception GetIdentErr;
> exception GetIdentErr

fun GetIdent (x::l) =
 if IsLetter x then GetIdentAux [x] l
               else raise GetIdentErr;
> val GetIdent =
>  fn : string list -> string * string list
```

# Unified treatment

- **Can unify Analysis of numbers and identifiers**
    - **single general function** `GetTail`
    - **takes a predicate as argument**
    - **then uses this to test whether to keep accumulating characters or to terminate**
    - `GetNumAux` **corresponds to**

      ```
      GetTail IsDigit
      ```

    - `GetIdentAux` **corresponds to**

      ```
      GetTail (fn x => IsLetter x orelse IsDigit x)
      ```

```
fun GetTail p buf [] = (implode(rev buf),[])
 |  GetTail p buf (l as x::l') =
     if p x then GetTail p (x::buf) l'
            else (implode(rev buf),l);
> val GetTail = fn
>   : (string->bool)
>      -> string list
>       -> string list -> string * string list
```

# GetNextToken and Tokenise

```
fun GetNextToken [x] = (x,[])
 |  GetNextToken (x::l) =
     if IsLetter x
      then GetTail
            (fn x => IsLetter x orelse IsDigit x)
            [x]
            l
      else if IsDigit x
            then GetTail IsDigit [x] l
            else (x,l);
> val GetNextToken =
>  fn : string list -> string * string list
```

- **To lexically analyse a list of characters:**
  - **repeat** `GetNextToken` **& discard separators**

```
fun Tokenise [] = []
 |  Tokenise (l as x::l')  =
     if IsSeparator x
      then Tokenise l'
      else let val (t,l'') = GetNextToken l
           in t::(Tokenise l'') end;
> val Tokenise = fn : string list -> string list

Tokenise (explode "123abcde1][  ] 56a");
> val it =
>  ["123","abcde1","]","[","]","56","a"] : string list
```

# Multi-character special symbols

- `Tokenise` doesn't handle multi-character special symbols
  - these will be specified by a table
    - represented as a list of pairs
    - that shows which characters can follow each initial segment of each special symbol
    - such a table represents a FSM transition function
- For example, suppose the special symbols are

  <=, <<, =>, =, ==>, ->

  then the table would be:

```
[("<",  ["=","<"]),
 ("=",  [">","="]),
 ("-",  [">"]),
 ("==", [">"])]
```

- Not fully general
  - if ==> is a special symbol
    - then == must be also

# Utility functions

- **Test for membership**

```
fun Mem x [] = false
 |  Mem x (x'::l) = (x=x') orelse Mem x l;
> val Mem = fn : ''a -> ''a list -> bool
```

- `Get` **looks up the list of possible successors of a given string in a special-symbol table**

```
fun Get x [] = []
 |  Get x ((x',l)::rest) =
     if x=x' then l else Get x rest;
> val Get = fn : ''a -> (''a * 'b list) list -> 'b list

Get "=" [("<",["=","<"]),
         ("=",[">","="]),
         ("-",[">"]),
         ("==",[">"])];
> val it = [">","="] : string list

Get "?" [("<", ["=","<"]),
         ("=", [">","="]),
         ("-", [">"]),
         ("==",[">"])];
> val it = [] : string list
```

# GetSymbol

- `GetSymbol` **takes**
  - **a special-symbol table**
  - **and a token**
- **It extends the token by**
  - **removing characters from the input**
  - **until table says no further extension is possible**

```
fun GetSymbol spectab tok [] = (tok,[])
 |  GetSymbol spectab tok (l as x::l') =
     if Mem x (Get tok spectab)
       then GetSymbol spectab (tok^x) l'
       else (tok,l);
> val GetSymbol = fn
>   : (string * string list) list
>     -> string -> string list -> string * string list
```

# GetNextToken

- `GetNextToken` **can be enhanced to handle special symbols**
- **Special-symbol table supplied as an argument**

```
fun GetNextToken spectab [x] = (x,[])
 |  GetNextToken spectab (x::(l as x'::l')) =
     if IsLetter x
      then GetTail
            (fn x => IsLetter x orelse IsDigit x)
            [x]
            l
      else if IsDigit x
            then GetTail IsDigit [x] l
            else if Mem x' (Get x spectab)
                  then GetSymbol
                        spectab
                        (implode[x,x'])
                        l'
                  else (x,l);
> val GetNextToken = fn
>   : (string * string list) list
>       -> string list -> string * string list
```

# Tokenise

- `Tokenise` **can be enhanced to use the new** `GetNextToken`

```
fun Tokenise spectab [] = []
 |  Tokenise spectab (l as x::l')  =
     if IsSeparator x
      then Tokenise spectab l'
      else let val (t,l'') = GetNextToken spectab l
           in t::(Tokenise spectab l'') end;
> val GetNextToken = fn
>    : (string * string list) list
>       -> string list -> string * string list
```

# Example

```
val SpecTab = [("=",  ["<",">","="]),
               ("<",  ["<",">"]),
               (">",  ["<",">"]),
               ("==", [">"])];
> val SpecTab =
>   [("=",["<",">","="]),
>    ("<",["<",">"]),
>    (">",["<",">"]),
>    ("==",[">"])]
>   : (string * string list) list

Tokenise SpecTab (explode "a==>b c5 d5==ff+gg7");
> val it =
>  ["a","==>","b","c5","d5","==","ff","+","gg7"]
>  : string list
```

- `Lex` **is a lexical analyser**

```
val Lex = Tokenise SpecTab o explode;
> val Lex = fn : string -> string list

Lex "a==>b c5 d5==ff+gg7";
> val it =
>  ["a","==>","b","c5","d5","==","ff","+","gg7"]
>  : string list
```

# The $\lambda$-calculus

- **The $\lambda$-calculus is a theory of functions**
  - **originally developed by Alonzo Church**
  - **as a foundation for mathematics**
  - **in the 1930s, several years before digital computers were invented**

- **In the 1920s Moses Schönfinkel developed** *combinators*

- **In the 1930s, Haskell Curry rediscovered and extended Schönfinkel's theory**
  - **and showed it equivalent to the $\lambda$-calculus.**

- **About this time Kleene showed that the $\lambda$-calculus was a universal computing system**
  - **it was one of the first such systems to be rigorously analysed**

# Enter Computer Science

- **In the 1950s John McCarthy was inspired by the $\lambda$-calculus to invent the programming language LISP**

- **In the early 1960s Peter Landin showed how the meaning of imperative programming languages could be specified by translating them into the $\lambda$-calculus**
  - **he also invented an influential prototype programming language called ISWIM**
  - **ISWIM introduced the main notations of functional programming**
  - **and influenced the design of both functional and imperative languages**
  - **ML was inspired by ISWIM**

# Strachey & Turner

- **Building on this work, Christopher Strachey laid the foundations for the important area of denotational semantics**

- **Technical questions concerning Strachey's work inspired the mathematical logician Dana Scott to invent the** *theory of domains*
  - **an important part of theoretical computer science**

- **During the 1970s Peter Henderson and Jim Morris took up Landin's work and wrote a number of influential papers arguing that functional programming had important advantages for software engineering**

- **At about the same time David Turner proposed that Schönfinkel and Curry's combinators could be used as the machine code of computers for executing functional programming languages**

# Theory can be useful!

- **$\lambda$-calculus is an obscure branch of mathematical logic that underlies important developments in programming language theory, such as the:**
  - **study of fundamental questions of computation**
  - **design of programming languages**
  - **semantics of programming languages**
  - **architecture of computers**

# Syntax and semantics of the $\lambda$-calculus

- **$\lambda$-calculus is a notation for defining functions**
  - **each $\lambda$-expression denotes a function**
  - **functions can represent data and data-structures**
  - **details later**
  - **examples include numbers, pairs, lists**
- **Just three kinds of $\lambda$-expressions**
  - *Variables*
  - *Function applications* **or** *Combinations*
  - *Abstractions*

# Variables

- **Functions denoted by variables are determined by what the variables are bound to**
  - **binding is done by abstractions**
- $V$**,** $V_1$**,** $V_2$ **etc. range over arbitrary variables**

# Function applications (combinations)

- **If $E_1$ and $E_2$ are $\lambda$-expressions**
  - **then so is** $(E_1\ E_2)$
  - **it denotes the result of applying the function denoted by $E_1$ to the function denoted by $E_2$**
  - $E_1$ **is called the** *rator* **(from 'operator')**
  - $E_2$ **is called the** *rand* **(from 'operand')**

# Abstractions

- **If $V$ is a variable and $E$ is a $\lambda$-expression**
  - **then $\lambda V.\ E$ is an abstraction**
  - **with *bound variable* $V$**
  - **and *body* $E$**

- **Such an abstraction denotes the function that takes an argument $a$ and returns as result the function denoted by $E$ when $V$ denotes $a$**

- **More specifically, the abstraction $\lambda V.\ E$ denotes a function which**
  - **takes an argument $E'$**
  - **and transforms it into $E$[$E'$/$V$]**
    - **the result of substituting $E'$ for $V$ in $E$**
    - **substitution defined later**

- **Compare $\lambda V.\ E$ with `fn` $V$ => $E$**

# Summary of $\lambda$-expressions

$$
\begin{array}{lcl}
<\lambda\text{-}expression> & ::= & <variable> \\
 & | & (<\lambda\text{-}expression> <\lambda\text{-}expression>) \\
 & | & (\lambda <variable> . <\lambda\text{-}expression>)
\end{array}
$$

- **If** $V$ **ranges over** $<$ *variable* $>$
- **And** $E$**,** $E_1$**,** $E_2$**,** $\ldots$ **etc. range over** $<$ $\lambda$-*expression* $>$
- **Then:**

$$
E ::= \underbrace{V}_{\text{variables}} \mid \underbrace{(E_1\ E_2)}_{\substack{\text{applications}\\ \text{(combinations)}}} \mid \underbrace{\lambda V.\ E}_{\text{abstractions}}
$$