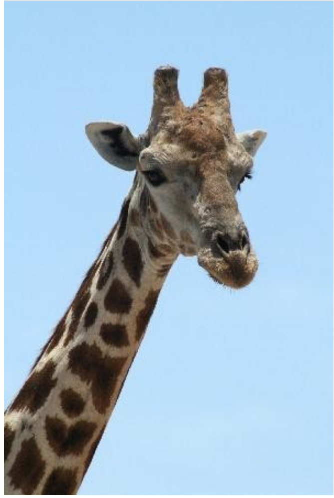

SETL, Smalltalk, and Agda

A Language Exploration Project

J Bolton

Overview

- SETL
  - Sets, tuples, and maps
  - Ada style procedural programming
  - Also includes objects
- Smalltalk
  - Purely object oriented
  - Dynamic and reflective
  - Untyped (sort of)
- Agda
  - Purely functional; Haskell-like syntax
  - Dependent types
  - Powerful pattern matching and mixfix

FIRST Set Construction Algorithm

- SETL
  - Readability and easy translation from pseudocode
- Smalltalk
  - Objects, state, and encapsulation
- Agda
  - Terseness, proof of correctness

SETL: Grammar Representation

- A context free grammar is a quadruple (V, Sigma, P, S) where V is a finite set of variables, Sigma (the alphabet) is a finite set of terminal symbols, P is a finite set of rules, and S is a distinguished element of V called the start symbol. The sets V and Sigma are assumed to be disjoint.

SETL: Grammar Representation

```
V := {"S", "A", "T", "Z", "B", "Y"};
Sigma := {"#", "+", "b", "(", ")", ""};
P := {["S", ["A", "#"]],
      ["A", ["T", "B"]],
      ["B", ["Z"]],
      ["B", [""]],
      ["Z", ["+", "T", "Y"]],
      ["Y", ["Z"]],
      ["Y", [""]],
      ["T", ["b"]],
      ["T", ["(", "A", ")"]]};
S := "S";
G := [V,Sigma,P,S];
```

SETL: Support Functions

$XY = \{uv \mid u \in X \text{ and } v \in Y\}$

$trunc_k(X) =$

$\{u \mid u \in X \text{ with } length(u) \le k$

$\text{or } uv \in X \text{ with } length(u) = k\}$

```
op .ct(a,b);
     return {x + y :
       x in a, y in b};
end op;
```

```
proc maptrunc(n, X);
     return {u(1..n)
          : u in X};
end proc;
```

SETL: The Algorithm

```
proc first(n, G);
  [V, Sigma, P, S] := G;
  F := F1 := {};
  for a in Sigma loop  $ 1.
    F1(a) := {a};
  end loop;
  for A in V loop    $ 2.
    if ([A, ""] in P)
      then F(A) := {""};
      else F(A) := {};
    end if;
  end loop;     $ 3.
  until forall A in V | F(A) = F1(A) loop
    for A in V loop    $3.1
      F1(A) := F(A);
    end loop;
    for R in P | #R > 1 loop    $3.2
      A := R(1);
      allfirst := .ct/[F1(u) : u in R(2)];
      F(A) +:= maptrunc(n,allfirst);
    end loop;
  end loop;
  return F;    $ 4.
end proc;
```

Input: context-free grammar $G = (V, \Sigma, P, S)$

1. for each a in Σ do $F'(a) := \{a\}$

2. for each A in V do $F(A) :=$
   $\{\lambda\}$ if $A \rightarrow \lambda$ is a rule in P
   $\varnothing$ otherwise

3. repeat

- 3.1 for each A in V do $F'(A) := F(A)$
  3.2 for each rule $A \rightarrow u_1u_2..u_n$ with $n>0$ do
  $F(A):=F(A) \cup trunc_k(F'(u_1)F'(u_2). . . F'(u_n))$
  until $F(A) = F'(A)$ for all A in V

4. $FIRST_k(A) = F(A)$

SETL: Conclusion

Advantages and Disadvantages

Smalltalk: The Data

```
Object subclass: Grammar [
    | Rules Start Variables Terminals |

    getVariables [^Variables]
    getSigma[^Sigma]
    getRules[^Rules]
    init [
        Rules := Bag new.
        Variables := Set new.
        Sigma := Set new.
    ]
    rule: l produces: r [
        Rules add: ((Rule new) lhs: l
rhs: r).
    ]
    setVariables: vs [Variables := vs]
    setTerminals: terms [Terminals :=
terms]
    setStart: st [Start := st]
]
```

```
Grammar subclass: Rule [
    | LHS RHS |

    getLHS [^LHS] getRHS[^RHS]
    lhs: l rhs: r [
        LHS := l.
        RHS := r.
    ]
    = x [
        ^(LHS = x getLHS & RHS = x
getRHS).
    ]
    hash [
       ^(LHS hash + RHS hash)
    ]
    printOn: stream [
        LHS printOn: stream.
        RHS printOn: stream.
    ]
]
```

Smalltalk: The Data

```
| grammar |
grammar := Grammar new init.
grammar setVariables: #( 'S' 'A' 'T' 'Z' 'B' 'Y').
grammar setTerminals: #('#' '+' 'b' '(' ')' '' ).
grammar rule: 'S' produces: #( 'A' '#').
grammar rule: 'A' produces: #( 'T' 'B').
grammar rule: 'B' produces: #( 'Z' ).
grammar rule: 'B' produces: #( '' ).
grammar rule: 'Z' produces: #( '+' 'T' 'Y' ).
grammar rule: 'Y' produces: #( 'Z' ).
grammar rule: 'Y' produces: #( '' ).
grammar rule: 'T' produces: #( 'b' ).
grammar rule: 'T' produces: #( '(' 'A' ')' ).
grammar setStart: 'S'.
(grammar getFirstSet: 1) printNl.
```

Smalltalk: The Algorithm (part 1)

```
Grammar extend [
getFirstSet: n [
  | aSet testrule |
  F := LookupTable new.
  F1 := LookupTable new.
  Terminals do: [ :t |
    aSet := Set new add: t; yourself.
    F1 at: t put: aSet.
    F at: t put: aSet.
   ].
   testrule := Rule new.
   Variables do: [ :v |
     testrule lhs: v rhs: #('' ).
     (Rules includes: testrule)
       ifTrue: [F at: v put:
         (Set new add: ''; yourself)]
       ifFalse: [F at: v put: Set new].
   ].
]
```

Input: context-free grammar $G = (V, \Sigma, P, S)$

1. for each a in Σ do $F'(a) := \{a\}$

2. for each A in V do $F(A) :=$

$$\begin{cases} \{\lambda\} & \text{if } A \rightarrow \lambda \text{ is a rule in P} \\ \varnothing & \text{otherwise} \end{cases}$$

Smalltalk: The Algorithm (part 2)

```
[F = F1] whileFalse: [
  | firsts fc |
  Variables do: [ :v |
    F1 at: v put: (F at: v).
  ].
  Rules do: [ :r |
    firsts := OrderedCollection new.
    (r getRHS) do: [ :x | firsts add:
(F1 at: x)].
    fc := firsts fold: [ :x :xs | x
concatLang: xs].
    fc := fc collect: [ :x | x
truncate: n ].
      F at: (r getLHS) put: ((F at: (r
getLHS)) + fc).
    ].
  ].
  ^F.
]
```

3. repeat

- 3.1 for each A in V do F'(A) := F(A)
  3.2 for each rule $A \rightarrow u_1 u_2 .. u_n$ with n>0 do
  $F(A) := F(A) \cup trunc_k(F'(u_1)F'(u_2)\ldots F'(u_n))$
  until F(A) = F'(A) for all A in V

4. $FIRST_k(A) = F(A)$

Smalltalk: Conclusion

Advantages and Disadvantages