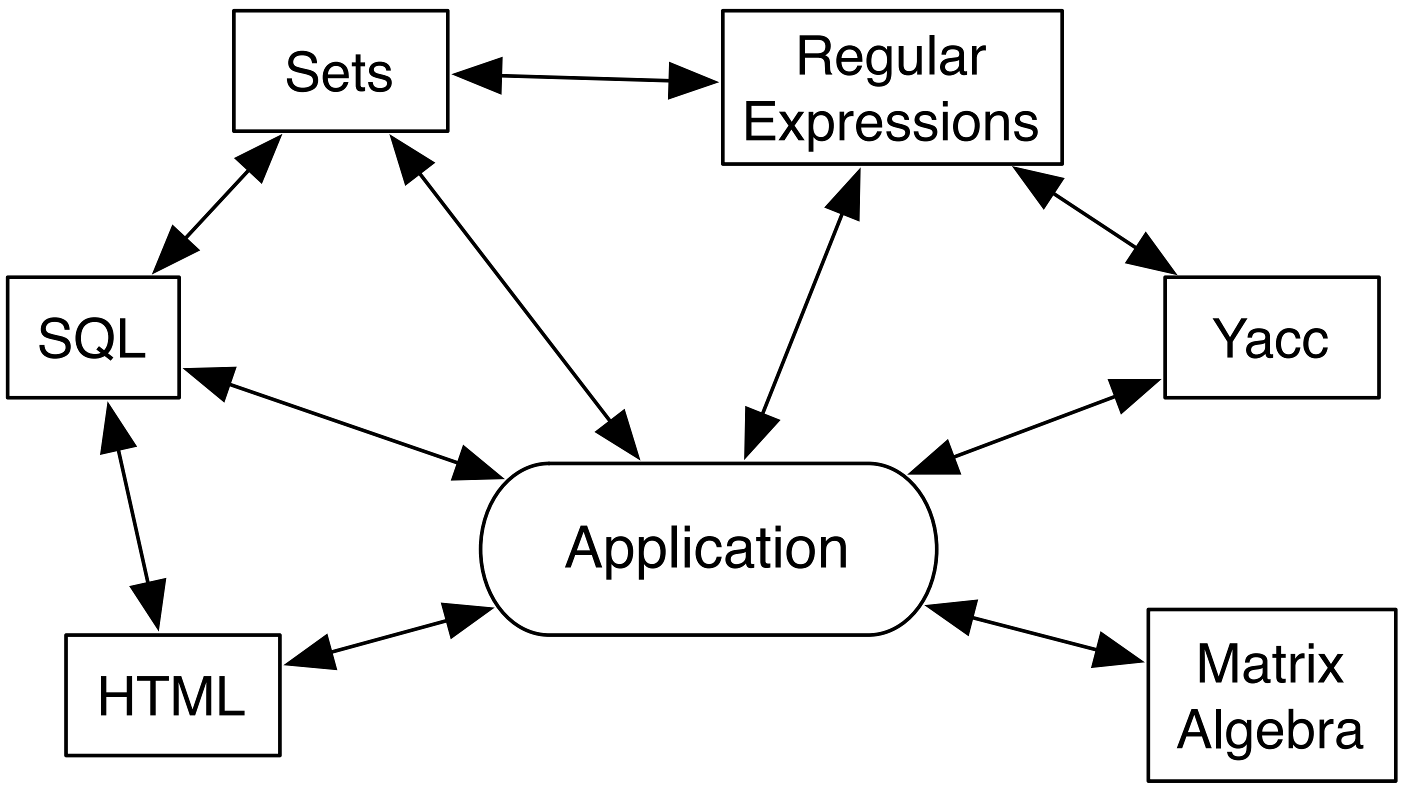
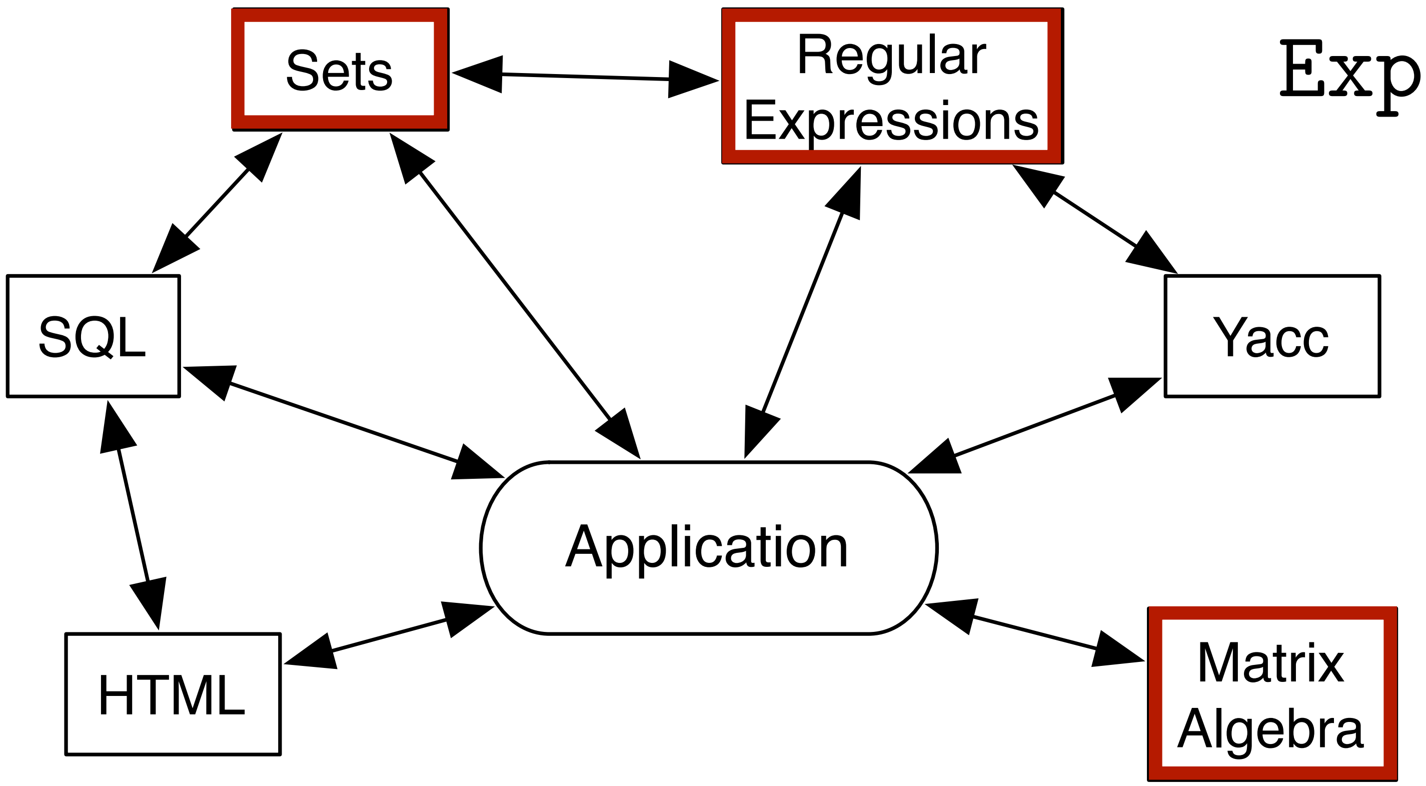
Well-typed Islands Parse Faster

Erik Silkensen and Jeremy Siek
University of Colorado

Composing DSLs



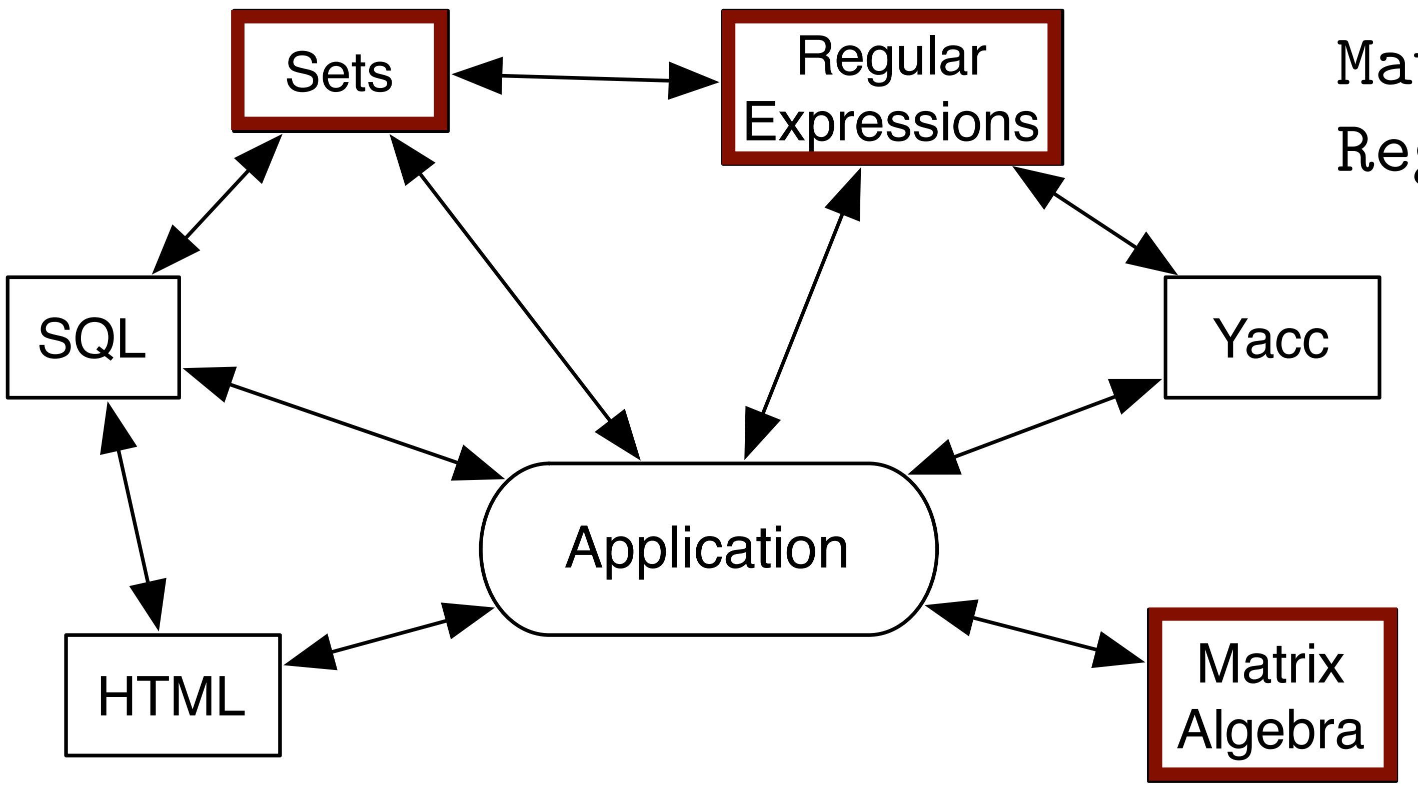
Composing DSLs




```
Expr ::= Expr "+" Expr
       | Expr "+" | ...
```

```
A + A + A
```

Composing DSLs

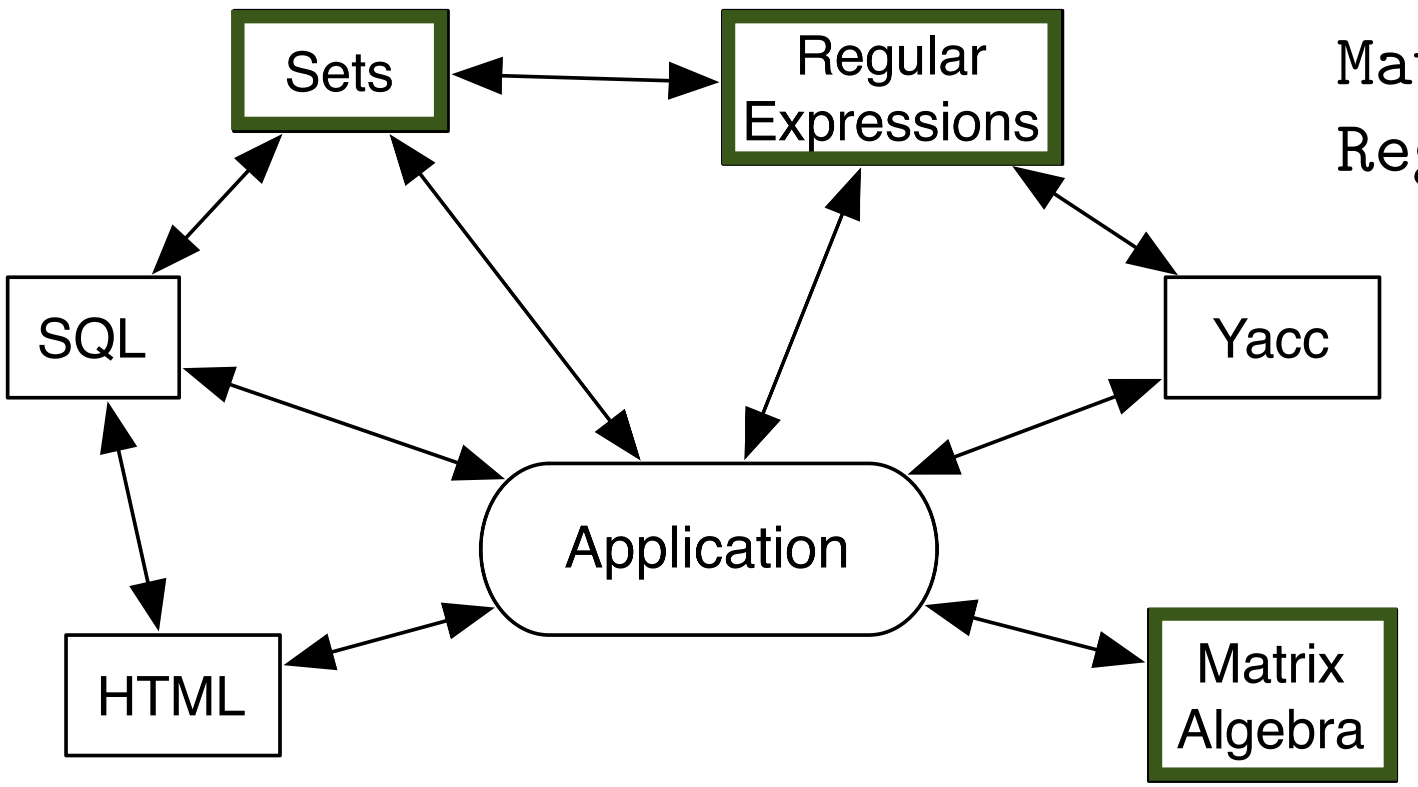



```
Matrix ::= Matrix "+" Matrix
Regexp ::= Regexp "+"
   Set ::= Set "+" Set
```

Type-Oriented Grammar

```
A + A + A
```

Composing DSLs




```
Matrix ::= Matrix "+" Matrix
Regexp ::= Regexp "+"
   Set ::= Set "+" Set
```

Type-Oriented Grammar

Type-based Disambiguation

declare `A` : `Matrix`;

`A + A + A`

Chart Parsing

[Kay 1986]

- CYK [1965, 1967, 1970]
- Earley [1968, 1970]
- Island [Stock et al. 1988]

$$O(|\mathcal{G}|n^3)$$

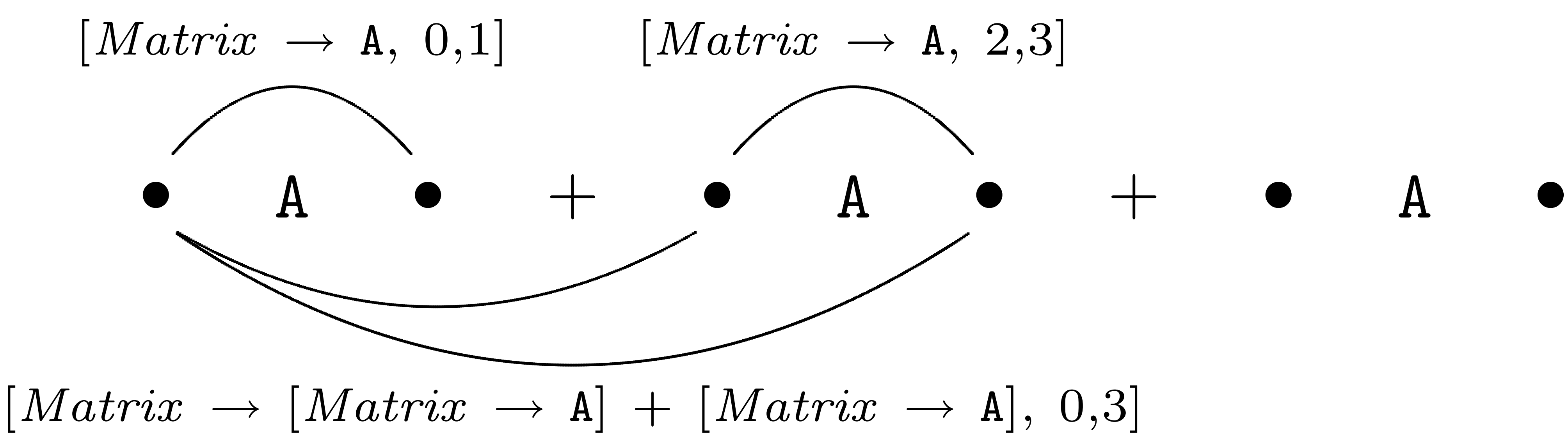


Chart Parsing

$$
(\mathrm{BU})\,\frac{(\mathrm{BU})\,\dfrac{\vdash [\mathtt{A}, 0, 1] \qquad \mathit{Matrix} \rightarrow \mathtt{A} \in \mathcal{P}}{\vdash [\mathit{Matrix} \rightarrow \mathtt{.A.}, 0, 1]} \qquad \mathit{Matrix} \rightarrow \mathit{Matrix}\ \mathtt{+}\ \mathit{Matrix} \in \mathcal{P}}{\vdash [\mathit{Matrix} \rightarrow \mathtt{.}\mathit{Matrix}\mathtt{.}\ \mathtt{+}\ \mathit{Matrix}, 0, 1]}
$$

$$
(\textsc{Compl})\,\frac{\vdash [\mathit{Matrix} \rightarrow \mathtt{.}\mathit{Matrix}\mathtt{.}\ \mathtt{+}\ \mathit{Matrix}, 0, 1] \qquad \vdash [\mathtt{+}, 1, 2]}{\vdash [\mathit{Matrix} \rightarrow \mathtt{.}\mathit{Matrix}\ \mathtt{+.}\ \mathit{Matrix}, 0, 2]}
$$

‘Type-Oriented’ Island Parsing

declare `A` : `Matrix`;

`A + A + A`

`A` – ‘well-typed island’

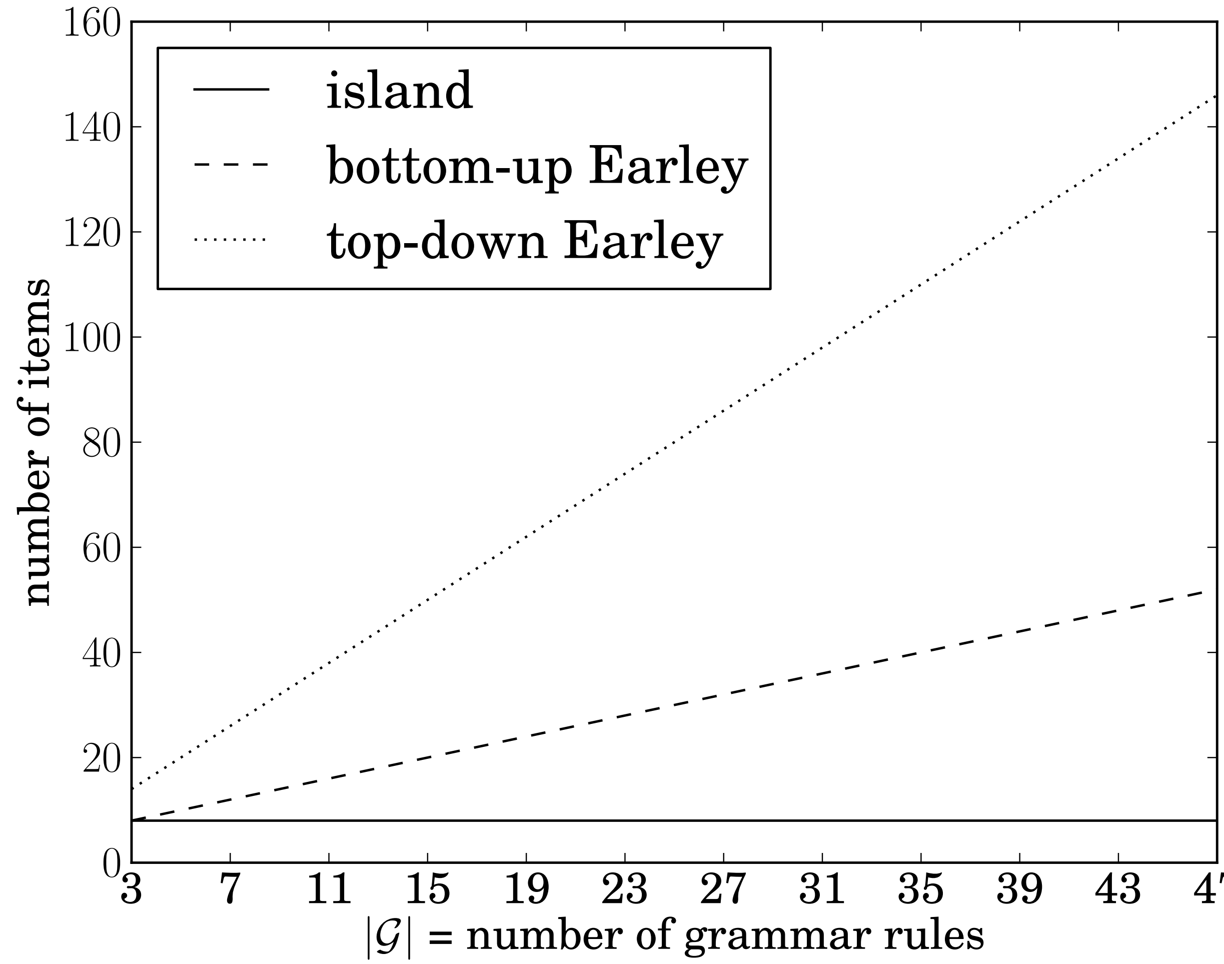
Don’t apply BU rule to ‘untyped islands’.

‘Type-Oriented’ Island Parsing

$$\textbf{module}\ \texttt{Typed}^0\ \{$$
```
  E ::= V;
  V ::= "-" V;
}
```
$$\textbf{module}\ \texttt{Typed}^i\ \{$$
```
  E ::= Mi;
  Mi ::= "-" Mi;
}
```

$\textbf{import}\ \mathcal{G}^0, \mathcal{G}^1, \ldots, \mathcal{G}^k;$
$\textbf{declare}\ \texttt{A:V};$

```
--A
```



A System for Extensible Syntax

- Variable Binders and Scope
- Rule-Action Pairs
- Structural Nonterminals

A System for Extensible Syntax

Variable Binders and Scope [Jim et al. 2010, Cardelli et al. 1994]

forall T1 T2.
  T2 ::= "let" x:Id "=" T1 { x:T1; T2 }

$\mathcal{G} \cup (\texttt{T1} \longrightarrow \texttt{x})$

let n = 7 { n * n }

Int ::= "n"

A System for Extensible Syntax

Rule-Action Pairs [Sandberg 1982]

```
Integer ::= "|" x:Integer "|" = (abs x);
```

$$(\mathbf{:}\ f\ \ (Integer\ \ \rightarrow\ \ Integer))$$
$$(\mathbf{define}\ \ (f\ \ x)\ \ (abs\ \ x))$$

A System for Extensible Syntax

Rule-Action Pairs [Sandberg 1982]

```
Integer ::= "|" x:Integer "|" = (abs x);
```

$(\textbf{:}\ f\ (Integer\ \rightarrow\ Integer))$
$(\textbf{define}\ (f\ x)\ (abs\ x))$

forall `T1 T2.`

```
  T2 ::= "let" x:Id "=" e1:T1 { x:T1; e2:T2 } ⇒
    (let: ([x : T1 e1]) e2);
```

$(\textbf{define-syntax-rule}\ (m\ x\ e_1\ e_2\ T_1\ T_2)$
$\quad (\textbf{let:}\ ([x\ :\ T_1\ e_1])\ e_2))$

A System for Extensible Syntax

Structural Nonterminals

```
forall T1 T2.
  T1 ::= p:(T1 × T2) "." "fst" = (car p);
```

A System for Extensible Syntax

Structural Nonterminals

$\mathbf{forall}$ `T1 T2.`
`T1 ::= p:` $\boxed{\texttt{(T1} \times \texttt{T2)}}$ `"." "fst"` $=$ `(car p);`

Let `Type` give the syntax of types (i.e., nonterminals) in a grammar,

```
Type ::= Id | "(" Type ")"
```

A System for Extensible Syntax

Structural Nonterminals

forall `T1 T2.`
  `T1 ::= p:(T1` $\times$ `T2) "." "fst"` $=$ `(car p);`

Let `Type` give the syntax of types (i.e., nonterminals) in a grammar,

```
Type ::= Id | "(" Type ")"
```

and map them to Typed Racket types with a third rule-action pair:

`Type ::= T:Id` $\equiv$ `T | "(" T:Type ")"` $\equiv$ `T`

A System for Extensible Syntax

Structural Nonterminals

```
forall T1 T2.
  T1 ::= p:(T1 × T2) "." "fst" = (car p);
```

Let Type give the syntax of types (i.e., nonterminals) in a grammar,

```
types {
  Type ::= T1:Type "×" T2:Type ≡
    (Pairof T1 T2);
}
```

and ... on pair:

```
Type ::= T:Id ≡ T | "(" T:Type ")" ≡ T
```

An Example

```
types {
  Type ::= T1:Type "->" T2:Type [right] ≡ (T1 -> T2);
}

forall T2.
  T1 -> T2 ::= "fun" x:Id ":" T1:Type { x:T1; e1:T2 } ⇒
    (λ: ([x : T1]) e1);

forall T1 T2.
  T2 ::= f:(T1 -> T2) x:T1 [left] ⇒ (f x);

forall T1 T2.
  T1 -> T2 ::= "fix" f:(T1 -> T2) -> (T1 -> T2) =
    ((λ: ([x : (Rec A (A -> (T1 -> T2)))])
        (f (λ (y) ((x x) y))))
    ((λ: ([x : (Rec A (A -> (T1 -> T2)))])
        (f (λ (y) ((x x) y)))));
```

An Example

```
let fact =
  fix fun f : Int -> Int {
        fun n : Int {
          if n < 2 then 1
          else n * f (n - 1)
        }
      }
{
  print fact 5
}
```

Related Work

- Earley and type inference:
  - Aasa et al. [1988], Missura [1997], Wieland [2009]
- Parsing Expression Grammars (PEGs) [Ford 2004]:
  - Fortress [Allen et al. 2009], Katahdin [Seaton 2007], Rats! [Grimm 2006]
- Scannerless GLR [Tomita 1985]:
  - MetaBorg [Bravenboer et al. 2005], SugarJ [Erdweg et al. 2011]

Implementation

http://extensible-syntax.googlecode.com