Compiler Construction

Syntactic Sugar

Syntactic sugar & Desugaring

Syntactic Sugar

Additions to a language to make it easier to read or write, but that do not change the expressiveness

Desugaring

Higher-level features that can be decomposed into **language core of essential constructs**
$\Rightarrow$ This process is called ”desugaring”.

Pros & Cons for syntactic sugar

Pros

- More readable, More writable
- Express things more elegantly

Cons

- Adds bloat to the languages
- Syntactic sugar can affect the formal structure of a language

Syntactic Sugar in Lambda-Calculus

The term "syntactic sugar" was coined by Peter J. Landin in 1964, while describing an ALGOL-like language that was defined in term of lambda- calculus
$\Rightarrow$ goal: replace λ by ***where***

Curryfication

$$\lambda xy.e \Rightarrow \lambda x.(\lambda y.e)$$

Local variables

$$\text{let } x = e_1 \text{ in } e_2$$
$$\Rightarrow (\lambda x.e_2).e_1$$

List Comprehension in Haskell

```
qs []     = []
qs (x:xs) =
 qs lt_x ++ [x] ++ qs ge_x
   where lt_x = [y | y <- xs,
                     y <  x]
         ge_x = [y | y <- xs,
                     x <= y]
```

List Comprehension in Haskell

Sugared

```
[(x,y) | x <- [1 .. 6],
         y <- [1 .. x],
         x+y < 10]
```

Desugared

```
filter p
 (concat (map
   (\ x -> map (\ y -> (x,y))
           [1..x]) [1..6]
         )
 )
 where p (x,y) = x+y < 10
```

Interferences with error messages

”true” | 42

```
standard input:1.1-6:
    type mismatch
    condition type: string
    expected type: int
```

```
  function _main() =
    (
      (if "true"
        then 1
        else (42 <> 0));
      ()
    )
```

Sugar in Tiger

Light

- `if then`

Sugar in Tiger

Light
- `if then`

Regular
- Unary `-`
- `&` and `|`
- Beware of `( exp )` vs. `( exps )`

Sugar in Tiger

Light
- `if then`

Regular
- Unary `-`
- `&` and `|`
- Beware of `( exp )` vs. `( exps )`

Extra
- `for`
- `?:` as in GNU C
  `(a ?: b)`
- `where`
- Function overload

Dangerous Desugaring

Suppose we want to introduce an in-bound operator

$$\alpha \leq \beta \leq \gamma$$

Naive translation

```
let
in
  if    α ≤ β  &  β ≤ γ
  then
     1
  else
     0
 end
```

Dangerous Desugaring

Another translation

```
let var _beta :=  β
in
  if   α ≤  _beta  &
       _beta  ≤ γ
  then 1
  else 0
end
```

Dangerous Desugaring

Another (another) translation

```
let var _alpha :=  α
    var _beta  :=  β
    var _gamma :=  γ
in
  if  _alpha  ≤  _beta  &
      _beta   ≤  _gamma
  then 1
  else 0
end
```

Dangerous Desugaring

Final (and correct) translation

```
let var _alpha := α
    var _beta  := β
in
  if _alpha ≤ _beta
  then
     let var _gamma := γ
     in
       if _beta ≤ _gamma
       then 1
       else 0
  else 0
end
```

Basic desugaring

1. Walk the AST using a visitor
2. Focus on the type of node to be replaced
3. Build new ub-AST
4. Replace the nodes (and associated sub-trees) by the new sub-AST

Tweasts

Text With Embedded AST

Idea: Is it possible to desugar directly inside of the parser

Advantages:

- Reduce the number of AST classes
- Avoid many desugaring traversals
- Desugaring in concrete syntax

Desugaring

Desugaring in Abstract Syntax

```
exp: exp "&" exp {
  $$ = new IfExp(@$, $1,
           new OpExp(@$,$3, OpExp::ne,
                       new IntExp(@2, 0)),
           new IntExp(@2, 0));
}
```

Desugaring

Desugaring in Abstract Syntax

```
exp: exp "&" exp {
  $$ = new IfExp(@$, $1,
          new OpExp(@$,$3, OpExp::ne,
                      new IntExp(@2, 0)),
          new IntExp(@2, 0));
}
```

Desugaring in Concrete Syntax

```
exp: exp "&" exp {
  $$ = parse::parse(parse::Tweast() <<
     "if " << $1 << " then " << $3 << "<> 0 else 0");
}
```

Summary