COMS22201: 2015/16

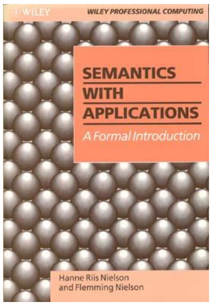
Language Engineering (Syntax)

Dr Oliver Ray
(csxor@Bristol.ac.uk)

Department of Computer Science
University of Bristol

Thursday 11th February, 2016

Course Textbook: Nielson



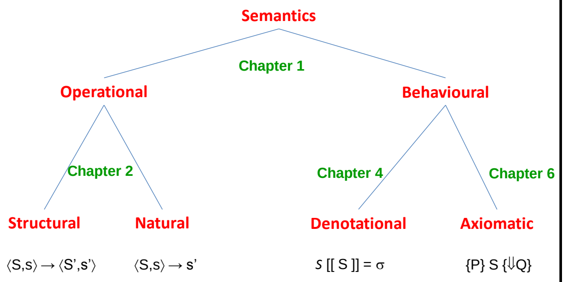
Semantics with Applications: A Formal Introduction

H. Nielson and F. Nielson

Revised online edition (click link below for pdf)

https://www.cs.bris.ac.uk/Teaching/Resources/COMS22201/nielson.pdf

Course Topics: Chapters 1, 4, 6 and 2



today: arithmetics
next week: Booleans
following week: Statements

The "While" Language: p7

Key semantic concepts will be illustrated using the simple (but Turing-complete) imperative language "While":

- **Numerals:** …, -1, 0, 1, 2, …
- **Variables:** …, "x", "y", "z", "temp", "v1", …
- **Arithmetics:** + , * , -
- **Booleans:** = , ≤ , ¬ , ∧
- **Statements:** := ,
  ; ,
  skip ,
  if … then … else … ,
  while … do …

Abstract Syntax: p7

a ::= n | x | a1 + a2 | a1 * a2 | a1 - a2

b ::= true | false | a1 = a2 | a1 ≤ a2 | ¬ b | b1 ∧ b2

S ::= x := a | S1 ; S2 | skip |
if b then S1 else S2 |
while b do S

where $n \in$ Num, $x \in$ Var, $a, a_i \in$ Aexp, $b, b_i \in$ Bexp, $S, S_i \in$ Stm

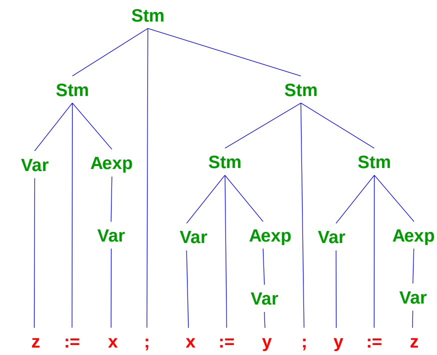
Haskell (Miranda) Encoding: p217

```
type Num = Integer

type Var = String

data Aexp = N Num | V Var
  | Add Aexp Aexp | Mult Aexp Aexp | Sub Aexp Aexp

data Bexp = TRUE | FALSE
  | Eq Aexp Aexp | Le Aexp Aexp
  | Neg Bexp | And Bexp Bexp

data Stm = Ass Var Aexp | Skip | Comp Stm Stm
  | If Bexp Stm Stm
  | While Bexp Stm
```

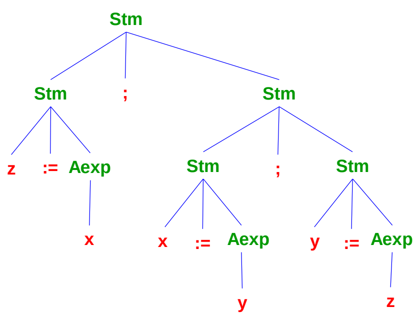
Equivalent BNF

```
⟨Aexp⟩ ::= ⟨Num⟩ | ⟨Var⟩
         | ⟨Aexp⟩ '+' ⟨Aexp⟩ | ⟨Aexp⟩ '*' ⟨Aexp⟩ | ⟨Aexp⟩ '−' ⟨Aexp⟩
⟨Bexp⟩ ::= 'true' | 'false'
         | ⟨Aexp⟩ '=' ⟨Aexp⟩ | ⟨Aexp⟩ '≤' ⟨Aexp⟩
         | '¬' ⟨Bexp⟩ | ⟨Bexp⟩ '∧' ⟨Bexp⟩
⟨Stm⟩  ::= ⟨Var⟩ ':=' ⟨Aexp⟩ | 'skip' | ⟨Stm⟩ ';' ⟨Stm⟩
         | 'if' ⟨Bexp⟩ 'then' ⟨Stm⟩ 'else' ⟨Stm⟩
         | 'while' ⟨Bexp⟩ 'do' ⟨Stm⟩
```

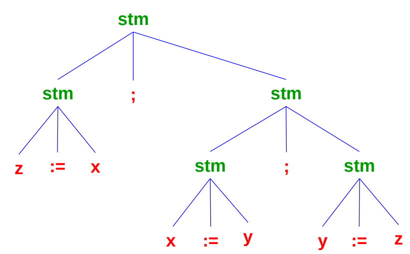
Parse Tree



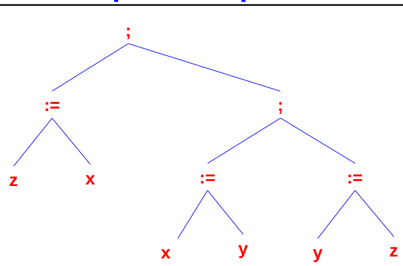
Simplified Tree



Abstract Syntax Tree: p8



Graphical Representation



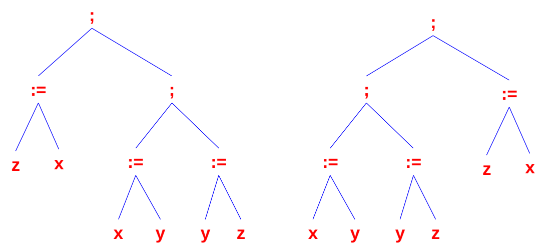
Linear Form

z := x ; (x := y ; y := z)

Semantics (of ASTs)

- The job of semantics is to assign meaning to ASTs
- The linear form retains all of the relevant syntactic information and is often easier to work with than a graphical form
- The linear form amounts to annotating the original expression with appropriately nested brackets
- The use of ASTs resolves any ambiguity that may be present in the abstract syntax of the language
- For example the two possible parses z := x ; (x := y ; y := z) and (z := x ; x := y) ; y := z of the same statement are distinguished at the level of abstract syntax

(Un)Ambiguity



z := x ; (x := y ; y := z)

(z := x ; x := y) ; y := z

Concrete Syntax

- In practice, the resolution of ambiguity would be resolved by an assumed underlying concrete syntax
- This is typically a more complex grammar with extra categories to ensures each expression can be uniquely parsed (by enforcing agreed precedence and associativity conventions)
- For example the production S ::= S_1 ; S_2 might be replaced by the more concrete productions S ::= Seq and Seq ::= S ; Seq to ensure right associativity of statement composition
- But semantics is not concerned with this detail as the concrete syntax merely serves to select one AST or another

Semantic Functions: p9

- The denotational style of semantics requires us to define a semantic function for each syntactic category
- These should be compositional: defining the meaning of an expression in terms of the meaning of (strict) sub-expressions
- A special notation with curly letters and square brackets is used so that we would write $\mathcal{F}$[[.]] : X → Y instead of f (.) : X → Y where X is a syntactic category and Y is a semantic class
- Anything inside syntactic brackets [[.]] is syntax (e.g. the numeral "0"); anything outside is semantics (e.g. the integer 0)
- Generally we try avoid non-semantic brackets as much as possible (as in functional programming)

Binary Numerals: p10

Syntax: n ::= 0 | 1 | n 0 | n 1 // Num

Semantics: $\mathcal{N}$: Num → Z // type signature

$\mathcal{N}$ [[0]] = 0 // basis elements

$\mathcal{N}$ [[1]] = 1

$\mathcal{N}$ [[n 0]] = 2 * $\mathcal{N}$ [[n]] // compound elements

$\mathcal{N}$ [[n 1]] = 2 * $\mathcal{N}$ [[n]] + 1

Example: $\mathcal{N}$ [[1 0 1]] = 2 * $\mathcal{N}$ [[1 0]] + 1

= 2 * (2 * $\mathcal{N}$ [[1]]) + 1

= 2 * (2 * 1) + 1

= 5

Program State: p12

- The value denoted by more complicated expressions (arithmetic, Boolean or statement) will typically depend on the program state
- We formalise a program state as a function from variables to integers. The set of all possible states is the set of such functions, denoted

  State = Var → Z
- For convenience, we can use a subscript notation to denote a state which maps particular variables x_k to particular integers i_k so that

  $S_{x_1=i_1 \; \ldots \; x_n=i_n} = \{ (x_1 , i_1) \ldots (x_n , i_n) \}$
- Strictly speaking states assign values to all variables, but we typically don't care about those variables not explicitly mentioned
- This leads to a more complex set of semantic functions of the form

  $\mathcal{F}$[[.]] : X → (State → Y)

Arithmetic Expressions: p13

Syntax: $a ::= n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2$

Semantics: $\mathcal{A} : \mathbf{Aexp} \rightarrow (\mathbf{State} \rightarrow \mathbf{Z})$

$$\mathcal{A}[[n]]\,s = \mathcal{N}[[n]]$$

$$\mathcal{A}[[x]]\,s = s\,x$$

$$\mathcal{A}[[a_1 + a_2]]\,s = \mathcal{A}[[a_1]]\,s + \mathcal{A}[[a_2]]\,s$$

$$\mathcal{A}[[a_1 * a_2]]\,s = \mathcal{A}[[a_1]]\,s * \mathcal{A}[[a_2]]\,s$$

$$\mathcal{A}[[a_1 - a_2]]\,s = \mathcal{A}[[a_1]]\,s - \mathcal{A}[[a_2]]\,s$$

Example:

$$\begin{aligned}\mathcal{A}[[x+1]]\,s_{x=3} &= \mathcal{A}[[x]]\,s_{x=3} + \mathcal{A}[[1]]\,s_{x=3} \\ &= s_{x=3}\,x + \mathcal{N}[[1]] \\ &= 3 + 1 = 4\end{aligned}$$