CS320 Handout 15
The INTEX Programming Language

Assaf Kfoury

February 10, 2006

Here are typical programs written in INTEX:

```
(program (a b)                    ;; computes the average of A and B
   (div (+ a b) 2)
   )

(program (cent)                   ;; converts Centrigade to Farenheit
   (+ 32 (div (* 9 cent) 5))
   )

(program (fahr)                   ;; converts Farenheit to Centigrade
   (div (* 5 (- fahr 32)) 9)
   )
```

1 The Syntax of INTEX

There are different ways of specifying the syntax of a programming language. We do it here using so-called BNF notation:

⟨literal⟩	::=	$\cdots \mid -3 \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \cdots$	one literal for every integer constant
⟨varref⟩	::=	$\cdots \mid$ `a` $\mid$ `b` $\mid \cdots \mid$ `x` $\mid$ `y` $\mid$ `avg` $\mid \cdots$	any Scheme symbol is a variable reference
⟨exp⟩	::=	⟨literal⟩ $\mid$ ⟨varref⟩ $\mid$ ⟨binapp⟩	expression
⟨binapp⟩	::=	(+ ⟨exp⟩ ⟨exp⟩) $\mid$	
		(− ⟨exp⟩ ⟨exp⟩) $\mid$	
		(∗ ⟨exp⟩ ⟨exp⟩) $\mid$	
		(`div` ⟨exp⟩ ⟨exp⟩)	
⟨formals⟩	::=	(⟨varref⟩ ⟨varref⟩ $\cdots$ ⟨varref⟩)	zero or more
⟨body⟩	::=	⟨exp⟩	
⟨program⟩	::=	(`program` ⟨formals⟩ ⟨body⟩)	

This BNF grammar – call it BNF-1 for later reference – is a precise statement of the syntax of INTEX programs. It says nothing about their semantics, i.e., how to execute them and how to assign meanings to them. As syntax, INTEX programs can be handled as *symbolic expressions* (S-expressions) of Scheme.

Our grammar BNF-1 is said to produce a *concrete syntax*. It is concrete in that it makes certain commitments about the final shape of INTEX programs that an *abstract syntax* may delay. For example, BNF-1 is already committed to enclose every binary applications between two matching parentheses (just as in Scheme), whereas a different BNF grammar for another, more abstract, syntax may be:

⟨literal⟩ ::=	⋯	as previous BNF
⟨varref⟩ ::=	⋯	as previous BNF
⟨exp⟩ ::=	⋯	as previous BNF
⟨binapp⟩ ::=	+ ⟨exp⟩ ⟨exp⟩ |	matching parentheses omitted
	− ⟨exp⟩ ⟨exp⟩ |	
	∗ ⟨exp⟩ ⟨exp⟩ |	
	`div` ⟨exp⟩ ⟨exp⟩	
⟨formals⟩ ::=	⋯	as previous BNF
⟨body⟩ ::=	⋯	as previous BNF
⟨program⟩ ::=	⋯	as previous BNF

According to this BNF – call it BNF-2 – the following is a legal expression:

`+` ⟨`exp-1`⟩ `*` ⟨`exp-2`⟩ ⟨`exp-3`⟩

Note the lack of parentheses in this expression. Is anything lost by omitting matching parenthesis pairs in binary applications? More specifically, are there legal expressions according to BNF-2 which are *ambiguous*? (An expression is ambiguous if it can be parsed in more than one way and thus can be interpreted in more than one way.) The answer is "no". By restricting the operators +, −, ∗ and `div` to be strictly binary, expressions are uniquely parsed – and the omitted parentheses are therefore superfluous. But the situation changes if we choose to make the operators +, −, ∗ and `div` have arbitrary arities (or if we add other features to the language).

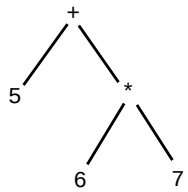
Exercise. Extend BNF-1 and BNF-2 by allowing the operators +, −, ∗ and `div` to have arbitrary arities, i.e., each can be applied to an arbitrary number $n \geqslant 0$ of arguments. Call the resulting grammars BNF-3 and BNF-4, respectively. Show that expressions which are legal according to BNF-3 are not ambiguous, whereas expressions which are legal according to BNF-4 are ambiguous.

According to the preceding exercise, if we make the BNF grammar "abstract" by omitting parentheses around applications, then we may turn it into an "ambiguous" grammar. In the case of a concrete BNF grammar, the parentheses dictate how an expression is uniquely parsed in the form of a tree (its so-called *abstract syntax tree*). In the case of an abstract BNF grammar, the omitted parentheses may allow an expression to be parsed into more than one tree.

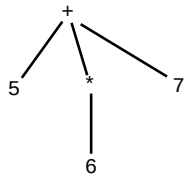
2 Abstract Syntax Trees

Given an expression generated by a BNF grammar, we can parse it in the form of an an *abstract syntax tree* (abbreviated *AST*). If the expression is not ambiguous, then the corresponding AST is uniquely determined, and vice-versa.

Let e be the expression "`+ 5 * 6 7`", which is legal according to both BNF-2 and BNF-4, but not according to BNF-1 and BNF-3 – make sure you agree with these assertions. According to BNF-2, the AST of e is uniquely determined and looks like the following:



According to BNF-4, there are two possible AST's of e, which implies that e is ambiguous (from BNF-4's point of view). One AST is the same as the preceding one (produced according to the rules of BNF-2), and another AST looks as follows:



Interpreting the first AST returns the value 47. Assuming that "`* 6`" (i.e., "`*`" applied to the single argument "`6`") returns 6, the second AST returns the value 18. These are two different values, 47 and 6, for the same expression e – which is another way of stating the ambiguity of BNF-4.

3 INTEX Programs as S-Expressions

Recall that an *abstract data types* (ADT) consists of a collection of objects together with appropriate operations on these objects. The collection of all INTEX programs as pieces of syntax, now viewed as S-expressions, together with appropriate operations to create them and break them apart, form an ADT.

A particular implementation of this ADT in Scheme is given in Handout # 16, entitled *Manipulating the syntax of INTEX programs*. Here is an example of how we can use the *constructors* of this ADT:

```
(define avg
   (make-program (list 'a 'b)
      (make-binapp 'div
                   (make-binapp '+
                                (make-varref 'a)
                                (make-varref 'b))
                   (make-literal 2))))
```

The result of the preceding code is to bind the name `avg` to the following INTEX program:

```
(program (a b)  (div (+ a b) 2) )
```

And here is an example of how to use the *selectors* of this ADT:

```
(define exp-size
   (lambda (exp)
      (if (literal? exp)
          1
          (if (varref? exp)
              1
              (if (binapp? exp)
                  (+ 1
                     (exp-size (binapp-rand1 exp))
                     (exp-size (binapp-rand2 exp)))
                  (error "EXP-SIZE: Unrecognized expression --" exp))))))

(define program-size
   (lambda (pgm)
      (+ 1 (exp-size (program-body pgm)))))
```