IACL2[1]: an Infix Dialect of ACL2
Translations to ACL2

Michael K. Smith
mksmith@cli.com

Date: July 11, 1997

Computational Logic, Inc.
1717 West 6th Street, Suite 290
Austin, Texas 78703-4776
(512) 322-9951
Fax: (512) 322-0656

1 Introduction

Iacl2 is an infix dialect of ACL2. The intent is to provide a more conventional interface to the theorem prover, lowering some of the barriers to its adoption for specification as well as proof. This document describes the mapping between Iacl2 and ACL2. As such, it does not constitute an introduction to Iacl2 for an ACL2 novice.

Iacl2 wraps two components, a parser and a printer, around ACL2. The printer will print anything that ACL2 will produce. But the parser does not parse all of these forms. Some will most likely be permanently considered "expert" forms that can only be entered using the native s-expression syntax of ACL2.

The typical relation between Iacl2 and ACL2 forms is shown below:

Syntax	**Translation**
$x1$ OP $x2$	(op x1 x2)
foo($x1$, ..., xn)	(foo x1 ... xn)

[1]This work was supported in part by DARPA contract N66001-95-C-8634.

Some ACL2 functions are translated to infix operators, for example arithmetic operators like '+' and relations like '<'. Functions for which we have not provided an infix form print as the standard notation for a function call vs. the Lisp prefix notation.

The parser will parse forms that the printer will not print back identically. And the printer will print more forms than the parser will handle. But for syntacticly well formed input, the following relation should hold:

$$parse(print(parse(input))) = parse(input)$$

Note that Iacl2 relies on ACL2 to enforce many restrictions. For example, the requirement that constants begin and end with '*' is not checked by the Iacl2 parser, but by ACL2 itself. So while the stand-alone Iacl2 parser can perform a quick preliminary syntactic check, an Iacl2 file cannot be consider syntactically acceptable until its translation has passed through ACL2.

There is one use of ACL2 for which the desired parse/printer symmetry is likely to fail. If the user defines macros that use non-standard syntax, the printed result is likely to be unparsable. An example would be a macro with the sntax of `COND`.

2 Notation

The syntax rules honor the following conventions.

- Non-terminals are in *italics*.
- Reserved words and special symbols are in `typewriter font`.
- "a ::= b" means that non-terminal a is defined by the patterns of b.
- Alternatives are separated by | .
- Optional terms are bracketed, [*expr*] .
- The notation x^* indicates a possibly empty sequence of x.
- The notation x^+ indicates a non-empty sequence of x.
- The notation x c ... c x indicates a non-empty sequence of x separated by c.

The lexical rules are similar, with the distinction that their terminal character components are not separated by whitespace.

3 Reserved Words and Special Symbols

Reserved Words					
and	append	assoc	atom	axiom	binary-*
binary-+	binary-@	bit	boolean	case	case-match
certify	certify!	char	collect	complex	congruence
consp	constant	current	current-theory	difference	disable
e0-ord-<	elif	else	enable	enum	equivalence
evaluator	exists	for	forall	function	function-theory
guards	if	if*	iff	ignore	implies
in	in-package	include	incompatible	integer	intersect
label	lambda	load	local	macro	member
module	null	never	nil	not	mutual-recursion
of	on	or	package	quote	ratio
rational	record	return	signature	signed-byte	skip-proofs
t	string	symbol	standard-char	table	termination
then	theorem	theory	theory-invariant	thm	to
type	typedef	union	universal-theory	unless	unsigned-byte
verify	when	word			

Special Symbols				
=	==	:=	~=	/=
<	>	<=	>=	#
->	<->	&	|	~
'[..]	`[..]	,@	,	
`	'	@	:>	
-	+	*	/	

4 Literals, Identifiers, Constants

Syntax
literal ::= *boolean* | *string* | *number* | *character*
keyword ::= :*id*

4.1 Literals

4.1.1 Boolean

In Iacl2, `t` is a reserved word for boolean true, and `nil` is used for boolean false. Any non-NIL value is considered true.

Syntax
boolean ::= `t` | `nil`

4.1.2 Numbers

We support optionally signed decimal and octal integer literals and optionally signed decimal rationals. An octal literal begins with a '`0`'. Note that we do not support complex literals.

Syntax
number ::= *octal* | *decimal* | *rational*
octal ::= [`-`] `0` *digit*$^+$
decimal ::= [`-`] *non-zero-digit digit**
rational ::= *decimal*/*decimal*

The second *decimal* of a *rational* cannot be signed.

`01`, `0177`	*octal literals*
`1`, `-77`, `200`	*decimal literals*
`4/32`, `-13/100`	*rational literals*

4.1.3 Strings

We support standard string syntax. A `"` is quoted in a string by preceding it with a backslash, e.g. `"ab\"cd"` or `"\""`.

```
"This is a string"
"\""
"foo\"bar"
```

4.1.4 Characters

Characters are surrounded by single quotes, e.g. `'a'`. A number of special characters using slash notation are also supported.

`'a'`	
`'\n'`	*newline*
`'\t'`	*tab*
`'\b'`	*rub*
`'\f'`	*page*

4.1.5 Comments

Comments are indicated by the C /* ... */ convention. We depend on the C preprocessor to remove them. Note that the fact that we use CPP means that you can also use the other preprocessor supported conventions like #include, #define, etc.

Examples

```
/* This is a comment */

/* This is a multi-line comment,
   which can go on for some time.*/
```

4.2 Identifiers

An identifier may have a package prefix.

Syntax

id ::= *atom* | *atom*:*atom* | *atom*::*atom*
atom ::= *alpha char**
alpha ::= `a` .. `z` | `A` .. `Z`
char ::= *alpha* | *digit* | `-` | `_` | `?`

4.3 Constant Declarations (DEFCONST)

A constant name must begin and end with an asterisk (*).

Examples

```
constant *day*  = 'Mon ;
constant *temp* : integer = 24 ;
constant *digits* = '[0 1 2 3 4 5 6 7 8 9];
constant *three* : integer  = 3;
```

Examples

```
(const *digits* '(0 1 2 3 4 5 6 7 8 9))
```

Prints as:

```
constant *digits* = '[0 1 2 3 4 5 6 7 8 9];
```

Parses to:

```
(const *digits* '(0 1 2 3 4 5 6 7 8 9))
```

```
constant *three* : integer  = 3;
```

Parses to:

```
(const *three* 3 :type integer :lemma (integerp *three*))
```

Macroexpands to:

```
(progn (defconst *three* 3)
       (defthm constant-three-type (integerp *three*)
               :rule-classes nil))
```

Syntax	**Translation**
constant ::= `constant` `*`*id*`*` [: *type*] = *expr* ;	(constant *id* *expr* :TYPE type :LEMMA theorem doc-string)

5 Types, including Enumerations, Lists, Alists, Records

Any function of a single argument can function as a type. Such functions should not themselves have typed arguments (guards). E.g. '1+' is not a permissible type. A non-NIL return from $f(x)$ indicates that x satisfies the requirements of the type f.

Primitive Types	
Name	**Translation**
integer	INTEGERP
atom	ATOM
boolean	BOOLEANP
bit	(OR (EQUAL x 1) (EQUAL x 0))
character	CHARACTERP
complex	COMPLEX-RATIONALP
cons	CONSP
nil	NIL
null	(NULL x)
ratio	(AND (RATIONALP x)(NOT (INTEGERP x)))
rational	RATIONALP
string	STRINGP
symbol	SYMBOLP
signed-byte	INTEGERP
unsigned-byte	UNSIGNED-BYTEP
standard-char	STANDARD-CHARP

The parser accepts a number of so-called type definitions, which are converted to the appropriate ACL2 function definitions. We support special constructors for simple enumeration types, records, lists, and association lists. These are based on a set of libraries that are pre-loaded into Iacl2, as do the looping and quantified function constructors. We make sure that functions that may be used as types have their guards checked by inserting (DECLARE (XARGS :verify-guards T)) into the translation to ACL2. We have suppressed this in some of the examples below.

Examples

`type foo = integer ;`

Parses to:

`(DEFUN foo (x) (INTEGERP x))`

Prints as:

`function foo (x) { integerp(x) } ;`

`type foo = integer *;`

Parses to:

`(DEFLIST foo (l) INTEGERP)`

Prints as:

`type foo = integer *;`

`type foo = [integer . character] *;`

Parses to:

`(DEFALIST foo (l) (INTEGERP . CHARACTERP))`

Prints as:

`type foo = [integer . character] *;`

`type bar = enum [a, b, c];`

Parses to:

`(DEFUN bar (x) (MEMBER-EQUAL x '(a b c)))`

Prints as:

`function bar(x) { x in '[a b c] };`

```
type PERSON = record ( name : string := "John",
                       age : integer := 19,
                       sex : string := "male");
```

Parses to:

```
(DEFSTRUCTURE MAKE-person
```

```
        (name (:assert (STRINGP name) :rewrite)(:default "John"))
        (age (:assert (INTEGERP age) :rewrite)(:default 19))
        (sex (:assert (STRINGP sex) :rewrite)(:default "male"))
        (:options :guards (:predicate person)(:conc-name NIL)
        (:keyword-constructor key-person)
        (:keyword-updater update-person)))
```

Prints as:

```
type PERSON = record ( name : string := "John",
                       age : integer := 19,
                       sex : string := "male"  );
```

Syntax

typedef ::= *rename* | *enumeration* | *list* | *alist* | *record*

rename ::= `type` *id* = *id* ;
enumeration ::= `type` *id* = `enum` [*id* , ..., *id*] ;
list ::= `type` *id* = *id* * ;
alist ::= `type` *id* = [*id* . *id*] * ;
record ::= `type` *id* = (*field* , .. , *field*) ;

field ::= `id` [: *type*] [= *term*] ;
type ::= *id*

6 Expressions

Syntax

expr ::= *expr* op_1 *expr* | *relation*
relation ::= *relation* op_2 *relation* | $expr_2$
$expr_2$ ::= $expr_2$ op_3 $expr_2$ | *term*
term ::= *term* op_4 *term* | *unary* | *primary*
unary ::= op_5 *primary*
primary ::= (*expression*) | *primary* . *primary* |
<*exprs*> | *primary* [*expr*] |
[*key id*] | '*id* |
[*expr* , ... , *expr*] |
id (*expr*, ..., *expr*) |
statement |
id | *constant* | *literal*

op_1 ::= `&` | `|` | `->` | `<->` |
op_2 ::= `=` | `==` | `/=` | `~=` | `<` | `<=` | `>` | `>=` | `in` | `~in` | `member` |
op_3 ::= `+` | `-` | `:>` | `@` | `\` |

op_4 ::= `*` | `/` | `union` | `difference` | `intersect` | `incompatible` |
op_5 ::= $\sim$ | `not` | `-` | `/`

The infix operators in the table below are listed in order of increasing precedence, with equal precedence operators grouped together. Binary operators are left or right associative as indicated below.

Iacl2		**ACL2**	**Associates**
a `->` b	⇔	(implies a b)	right
a `<->` b	⇔	(iff a b)	right
a & b	⇔	(and a b)	right
$a \mid b$	⇔	(or a b)	right
$\sim a$	⇔	(not a)	
a `=` b	⇔	(equal a b)	left
a `==` b	⇔	(= a b)	left
a eq b	⇔	(eq a b)	left
a ∼`=` b	⇔	(not (equal a b))	left
a `/=` b	⇔	(/= a b)	left
a in b	⇔	(member a b)	left
a ∼in b	⇔	(not (member a b))	left
a `<` b	⇔	(`<` a b)	left
a `<=` b	⇔	(`<=` a b)	left
a `>` b	⇔	(`>` a b)	left
a `>=` b	⇔	(`>=` a b)	left
a `e0-ord-<` b	⇔	(`e0-ord-<` a b)	left
a `@` b	⇔	(append a b)	right
$b \backslash a$	⇔	(assoc a b)	left
a `:>` b	⇔	(cons a b)	right
a difference b	⇔	(difference a b)	left
a intersect b	⇔	(intersect a b)	left
a union b	⇔	(union a b)	left
a incompatible b	⇔	(incompatible a b)	left
$a + b$	⇔	(+ a b)	right
$a - b$	⇔	(- a b)	left
$a * b$	⇔	(* a b)	right
a/b	⇔	(/ a b)	left
'a	⇔	(quote a)	
[a, b, c, d]	⇔	(list a b c d)	
'[a b c d]	⇔	'(a b c d)	

Some examples:

Infix		**Translation**
a & b & c & d	⇔	(and a (and b (and c d)))
a & (b & (c & d))	⇔	(and a (and b (and c d)))
((a & b) & c) & d	⇔	(and (and (and a b) c) d)
(3 + 2) - (3 + 4)	⇔	(- (+ 3 2) (+ 3 4))
x * y * y = x * y * y	⇔	(equal (* x (* y y)) (* x (* y y)))
2 + 3 + 3 + 3	⇔	(+ 2 (+ 3 (+ 3 3)))
(2 + 3) + 3 + 3	⇔	(+ (+ 2 3) (+ 3 3))
((2 + 3) + 3) + 3	⇔	(+ (+ (+ 2 3) 3) 3)
2 + 3 + 4 + 5	⇐	(+ 2 (+ 3 (+ 4 5)))
2 + 3 + 4 + 5	⇐	(+ 2 3 4 5)
1 - 2 - 3 - 4	⇐	(- 1 2 3 4)
1 - 2 - 3 - 4	⇔	(- (- (- 1 2) 3) 4)
2 * 3 * 4 * 5	⇐	(* 2 3 4 5)
1 * 2 * 3 * 4	⇔	(* 1 (* 2 (* 3 4)))
1 / 2 / 3	⇔	(/ (/ 1 2) 3)
(1 / 2) * 3	⇒	(* (/ 1 2) 3)
1 / 2 * 3	⇔	(* (/ 1 2) 3))
2/3 * 1	⇔	(* 2/3 1))
1/2/3/4	⇒	(/ 1/2 3/4)
1/2 / 3/4	⇔	(/ 1/2 3/4)
1 / 2 / 3 / 4	⇐	(/ 1 2 3 4)
1 / 2 / 3 / 4	⇔	(/ (/ (/ 1 2) 3) 4)
∼∼a	⇔	(not (not a))
-x	⇔	(- x)
(- x)	⇒	(- x)
-4	⇔	-4

Some equivalences.

6.1 Arrays

ACL2's support for arrays is sufficiently unusual that we do not provide any special support for them. In fact, we use the syntax $a[i]$ to represent the ACL2 list operation `(nth a i)`. See the appendix.

7 Statements

Before we present the syntax of blocks we will describe multiple-valued forms and assignment, which are used in blocks.

7.1 Multiple Values

Functions normally return a single value. They can return multiple values. To create an expression returning multiple values use angle brackets as shown below.

Syntax
<*expr*, ..., *expr*>

7.2 Assignment

Assignments have the effect of '`let*`' variable introduction in ACL2. They can only appear in blocks that terminate in a non-assignment expression. In the case of multiple valued assignments (`mv-let`), the list of variables is enclosed in angle brackets.

Syntax
assignment ::= *id* `:=` *expr* `;` |
                `<`*id*, ..., *id*`>` `:=` *expr* `;`

7.3 Blocks

A block is an apparent sequence of statements, separated by ';' and enclosed in braces, '{ ' and '} '. It is in fact a `let*`. There may only be one expression following a sequence of assignments. The value of that expression is the value of the block.

Syntax
block ::= { [*assignment**] *expr*; }

Infix		**Translation**
{ <X, Y, Z> := <1, 2, 3>; *expr* } ;	⇔	(mv-let (x y z) (mv 1 2 3) *expr*)

{ a := b;
  { c := d;
    expr } } ;
	⇐	(let ((a b)
		        (c d))
		    expr)

{ a := b;
  { c := d;
    expr } } ;
	⇒	(let* ((a b)
		        (c d))
		    expr)

{ a := 1;
   c := 4;
   a + c } ;
	⇔	(let* ((a 1)
		         (c 4))
		    (+ a c))

<a, *expr*>	⇔	(mv a *expr*)

7.4 Flow of Control

Iacl2 depends on recursion for much of its control flow. We provide two conditional "statements" (actually expressions), `if` and `case`. In addition there is limited support for an iterative style of function definition (see iterative functions).

7.4.1 If statement

Syntax

`if` (*expr*) `then` *expr*
[`elif` (*expr*) `then` *expr*] *
`else` *expr*

Examples

```
if (a ~= b) then {
   c := 3;
   a * c }
else
   nil ;
```

Infix		Translation
if x then y else z	⇔	(if x y z)
if x then y elif a then b	⇔	(if x y (if a b))
if a then b elif c then d else x	⇔	(cond ((a b) (c d) (t x)))

7.4.2 Case statement

Syntax

`case` *expr* {
    val_1 : $expr_1$;
    ...
    val_k : $expr_k$;
    [`else` : $expr_{k+1}$] } ;

Example

```
case day {
 'mon : 1;
 'tue : 2;
```

```
 'wed : 3;
 'thu : 4;
 'fri : 5;
  else : 0; }
```

Parses to:

```
(CASE day ('mon 1 )
          ('tue 2 )
          ('wed 3 )
          ('thu 4 )
          ('fri 5 )
          (OTHERWISE 0 ))
```

8 Basic Events

8.1 Axioms (DEFAXIOM)

Example

```
(DEFAXIOM sbar
        (implies (bar x) (bar-zero y))
        :rule-classes (:rewrite)
        :doc "sbar testing")
```

Prints as:

```
/* sbar testing */
Axiom sbar
{ bar (x) -> bar-zero (y);}
 :rule-classes [:rewrite] ;
```

Parses to:

```
(DEFAXIOM sbar
        (implies (bar x) (bar-zero y))
        :rule-classes (:rewrite))
```

Syntax	**Translation**
axiom *id* { *expr* } :rule-classes rule-classes ;	(defaxiom name expr :rule-classes rule-classes)

8.2 Functions (DEF)

The formals of a function definitiona may be annotated with type requirements. In addition, the user can specify an arbitrary predicate that must hold between the inputs. The "| *expr*" syntax is used to assert such a relation.

The type and predicate requirements are translated into ACL2 *guards*. When introducing another function, f, that calls such a guarded function, say g, it must be proven that the f guarantees that the guards of g are satisfied.

Examples

```
function foo (x , y) {  x + y }   ;
```

Parses to:

```
(def foo (x y) (+ x y) :DISABLE T)
```

```
function foo (x : integer, y : integer  | bar(x)): integer
 {  if x < y then 1 + x else y - 1 ;}
 :measure p(x);
```

Parses to:

```
(def foo (x y)
     (DECLARE (TYPE INTEGER x)
              (TYPE INTEGER y)
              (XARGS :guard (bar x))
              (XARGS :measure (p x)))
              (IF (< x y)(+ 1 x)(- y 1))
     :TYPE INTEGER
     :LEMMA (IMPLIES (AND (INTEGERP x)(INTEGERP y))
                     (INTEGERP (foo x y)))
     :DISABLE T)
```

Macroexpands to:

```
(progn (defun foo(x y)
         (DECLARE (TYPE INTEGER x)
         (TYPE INTEGER y)
         (XARGS :guard (bar x))
            (XARGS :measure (p x)))
         (IF (< x y) (+ 1 x) (- y 1)))
       (defthm defun-foo-type
         (IMPLIES (AND (INTEGERP x)(INTEGERP y))
                  (INTEGERP (foo x y)))))
```

DEF is a macro which has **DEFUN** syntax. In addition, it takes keywords. :TYPE specifies the function type and :LEMMA is a constructed theorem asserting that if the arguments are of the right type then the function will return a value of the function type.

Syntax	**Translation**
`function` *id* (*vars* [| *expr*]) [: *type*]	(def id (*vars*)
{ *term* }	[dcl ... dcl]

[dcl ... dcl];

term
:TYPE *type*
:LEMMA theorem
:DISABLE [T | F])

vars ::= $\left[\right.$ *var* , ... , *var* $\left.\right]$
var ::= *id* $\left[\right.$: *type* $\left.\right]$
type ::= *id*

Syntax

dcl ::= `:measure` *expr* |
`:mode` $\left[\right.$ `PROGRAM` | `LOGIC` |
`:guard` *expr* |
`:verify-guards` *boolean* |
`:guard-hints [` *hint** `]` |
`:hints [` *hints** `]` |
`:otf-flag` *boolean* |

hints::= `[` *string hint** `]` |
hint ::= `:do-not-induct` *expr* |
`:do-not` *id* |
`:do-not [` *id** `]` |
`:in-theory` *expr* |
`:expand` *expr* |
`:cases` *expr* |
`:induct` *expr* |
`:by` *lmi* |
`:use` *lmi* |
`:use [` *lmi** `]` |
`:hands-off` $\left[\right.$ *id* | `[` id^+ `]` $\left.\right]$ |
`:restrict [` *rune sub* `,` ... `,` *sub rune* `]` |
`:bdd` *bdd*

lmi ::= *rune* |
`[ :theorem` *expr* `]` |
`[ :instance` *lmi sub* `]` |
`[ :functional-instance` *lmi sub* `]` |

rune ::= `[` *key id* `]` |
`[ :instance` *id* `]` |
`[ :functional-instance` *id* `]` |
`[ :theorem` *id* `]` |
id|

(*id*)

sub ::= [*set* , ... , *set*]

set ::= *id* := *expr*

We do not handle IGNORE dcls, e.g (declare (ignore x)).

8.3 Functions with Quantifiers (DEFUN-SK)

Example

```
function some-x-y-p-and-q(z)
{  exists x, y  | p(x, y, z) & q(x, y, z) }   ;
```

Parses to:

```
(DEFUN-SK some-x-y-p-and-q (z)
 (exists (x y)
        (and (p x y z)
             (q x y z))))
```

Syntax	**Translation**
`function` *id* (*vars*) { *quantifier id* , ... , *id* | *term* } ;	(DEFUN-SK *id* (*vars*) (*quantifier id* term*) &key doc quant-ok skolem-name thm-name)

quantifier ::= `exists` | `forall`

We don't support types on the arguments of quantified functions.

8.4 Iterative Functions (DEFLOOP)

This form of function definition is convenient for writing executable Iacl2 and for expressing some concepts in a familiar, procedural fashion. But there is little or no proof support for functions defined this way.

Example

```
function all-keyword(l)
{  for x in l, when keywordp(x), collect x }   ;
```

Parses to:

```
(DEFLOOP all-keyword (l)
  (FOR ((x IN l)) (WHEN (keywordp x) (COLLECT x))))
```

```
(defloop keyword-listp (l)
  "Recognizes lists of keywords."
  (declare (xargs :guard t))
  (for ((x in l)) (always (keywordp x))))
```

Prints as:

```
 /* Recognizes lists of keywords. */
Function keyword-listp(l)
{ for x in l, forall keywordp(x) }
  :guard t;
```

Syntax	Translation
`function` *id* (*id* , ... , *id*)	(defloop *id* (*id**)
{ `for` *id from expr* [`by` *expr*] ,	(for ((*id from* expr [`by` expr]))
[*condition expr*]	[(*condition* expr)]
[[`return` *expr*]	[[(return expr)]
[op expr]	[*(*op *expr)*]
[test expr]] } ;	[*(*test *expr)*]] *))*

from ::= `in` | `on`
condition::= `when` | `if`
op ::= `collect` | `append`
test ::= `forall` | `exists` | `never`

8.5 Macros (DEFMACRO)

We support two forms of macro definition in Iacl2. A "substitution macro" is one that can be expressed as a simple back-quoted form, substituting the bindings of the macro variables for their occurence within the body (e.g. like a CPP macro). If the body is a substitution macro, then the syntax is "macro foo(a,b) = ...". If you need to create a macro that is other than a simple substitution macro, an arbitrary constructive function, use "macro foo(a,b) # f(a,b)". '&' keywords only make sense in # macros.

Examples

```
macro key-name(name) = name[1];
```

Parses to:

```
(DEFMACRO key-name(name) `(NTH 1 ,name))
```

```
macro static-name(name) # name[1];
```

Parses to:

```
(DEFMACRO static-name(name) (NTH 1 name))
```

```
Thus

key-name(x) = x[1];

static-name('[a, b, c]) = 'b;
```

Syntax	**Translation**
`macro` id $(id^*) = term;$	(defmacro id (id^*) ` $term$) *With appropiate ",id" substitutions for variable occurences.*
`macro` id (id^*) # $term;$	(defmacro id (id^*) $term$)

8.6 Signature (DEFSTUB)

Examples

```
signature test1 (x, y, state) ;
```

Parses to:

```
(defstub-signature test1 (x y state) T)
```

Prints as:

```
signature test1(x, y, state) : t;
```

```
signature subr1 (x, y, state) : < T, state>;
```

Parses to:

```
(defstub-signature subr1 (x y state) (MV T state))
```

```
signature subr2(x, y, state) : foo;
```

Parses to:

```
(defstub-signature subr2 (x y state) foo :AXIOM (foo (subr2 x y state)))
```

```
signature subr3 (x : integer, y : boolean) : < integer, character>;
```

Parses to:

```
(defstub-signature subr3 (x y)
  (mv integer character)
  :axiom (implies (and (integerp x) (booleanp y)) (mv (subr3 x y))))
```

Macroexpands to:

```
(progn
  (defstub subr3 (x y) (mv integer character))

  (defaxiom signature-subr1-axiom (x y)
```

```
(implies (and (integerp x) (booleanp y)) (mv (subr3 x y)))))
```

Syntax	**Translation**
`signature` *id* (*vars*)	(defstub-signature id (id^*)
[: *id* | < *id*, ..., *id* >]	[*id* | (mv id^*)]
	:axiom axiom)

8.7 Theorems (DEFTHM)

Example

```
theorem eval-exp-list-val-type-prescription
{ flg -> true-listp (eval-exp-plus-val (flg, x, vs)) ; }
 :rule-classes :type-prescription;
```

Parses to:

```
(DEFTHM eval-exp-list-val-type-prescription
 (implies flg
         (true-listp (eval-exp-plus-val flg x vs)))
 :RULE-CLASSES :TYPE-PRESCRIPTION)
```

Syntax	**Translation**
`theorem` *id* { *term* }	(defthm *id term*
`:rule-classes` *rule-classes*	:rule-classes *rule-classes*
`:instructions` *instruction*	:instructions *instructions*
`:hint` *hints*	:hint *hints*
`:otf-flag` *otf-flag* ;	:otf-flg *otf-flg*
	:doc doc-string)

9 Complex Events

9.1 Modules (ENCAPSULATE)

Syntax	**Translation**
`module` [*id*]	(module ($signature^+$)
{ $signature^+$ }	($event^+$)
{ $event^+$ }	[:label *id*])

Example

```
module
{ an-element(lst) : t; }
```

```
{ local function an-element(lst)
        { if consp(lst) then car(lst)
              else nil }  ;
  local theorem member-equal-car
         { lst & true-listp(lst) -> car(lst) in lst }  ;
  theorem thm1 { null(lst) -> null(an-element(lst)) }  ;
  theorem thm2
  { true-listp(lst) & ∼(null(lst)) -> an-element(lst) in lst }  ; }  ;
```

Parses to:

```
(MODULE ((an-element (lst) t))
  (local (def an-element (lst)
           (if (consp lst) (car lst) nil)))
  (local (defthm member-equal-car
           (implies (and lst (true-listp lst))
                    (member-equal (car lst) lst))))
  (defthm thm1
     (implies (null lst) (null (an-element lst))))
  (defthm thm2
     (implies (and (true-listp lst)
                   (not (null lst)))
              (member-equal (an-element lst) lst))))
```

```
Module foo
{ an-element(lst) : t; }
{ local
  Function an-element(lst)
  { IF consp(lst) then car(lst)
         else nil }  ;
  local
  Theorem member-equal-car
  { lst & true-listp(lst) -> car(lst) in lst }  ;
  Theorem thm1  null(lst) -> null(an-element(lst)) }  ;
  Theorem thm2
  { true-listp(lst) & ~(null(lst)) -> an-element(lst) in lst ; }  ;
```

Prints as:

```
(MODULE ((an-element (lst) T))
     (local (def an-element (lst)
              (if (consp lst)(car lst)nil)
                   :disable t))
     (local (defthm member-equal-car
               (implies (and lst (true-listp lst))
                  (member-equal (car lst)lst))))
     (defthm thm1 (implies (null lst)
          (null (an-element lst))))
     (defthm thm2 (implies (and (true-listp lst)
                            (not (null lst)))
```

```
                  (member-equal (an-element lst)lst)))
     :label foo)
```

Macroexpands to:

```
(progn (DEFLABEL foo-start)
       (MODULE ((an-element (lst) T))
               (local (def an-element (lst)
                         (if (consp lst)(car lst)nil)
                       :disable t))
               (local (defthm member-equal-car
                            (implies (and lst (true-listp lst))
                                 (member-equal (car lst)lst))))
               (defthm thm1 (implies (null lst)
                              (null (an-element lst))))
               (defthm thm2 (implies (and (true-listp lst) (not (null lst)))
                              (member-equal (an-element lst)lst))))
        (DEFTHEORY foo
                   (set-difference-theories (current-theory :here)
                         (current-theory 'foo-start))))
```

9.2 Theory, In-Theory

9.2.1 Theory (DEFTHEORY)

Syntax	**Translation**
`theory` *id* := *term* ;	(DEFTHEORY *id term* :doc *string*)

Examples

```
theory ava-op-fns :=
           current-theory(:here)
 difference union-theories(function-theory('ava-op-fns-start), disable(moo));
```

Parses to:

```
(deftheory ava-op-fns
  (set-difference-theories
    (current-theory :here)
    (union-theories (function-theory 'ava-op-fns-start )(disable moo))))
```

9.2.2 In-theory

Syntax	**Translation**
In-Theory (*term*);	(IN-THEORY *term* :doc *string*)

Example

```
(in-theory (set-difference-theories
             (universal-theory :here)
             '(flatten (:executable-counterpart flatten))))
```

Prints as:

```
in-theory(            current-theory(:here)
           difference '[flatten [:executable-counterpart flatten]]);
```

9.3 Labels (DEFLABEL)

Syntax	**Translation**
label *id* ;	(deflabel *id* :doc *string*)

Example

```
(deflabel interp-section :doc "text")
```

Prints as:

```
/* Text */
label interp-section ;
```

Parses to:

```
(deflabel interp-section)
```

Note that we lose the :doc field, having treated it as a comment in Iacl2.

9.4 Packages

9.4.1 Package (DEFPKG)

Syntax	**Translation**
`package` *id* = *term* ;	(defpkg *id* *term* *string*)

The package *id* must be a string.

Example

```
(defpkg "my-pkg"
        (union-eq *acl2-exports*
                  *common-lisp-symbols-from-main-lisp-package*))
```

Prints as:

```
package "my-pkg" = union-eq(*acl2-exports*,
                            *common-lisp-symbols-from-main-lisp-package*);
```

9.4.2 IN-PACKAGE

Example

```
(in-package "ACL2")
```

Prints as:

```
in-package("ACL2");
```

Parses to:

```
(in-package "ACL2")
```

9.5 Mutual Recursion

9.5.1 Mutual-Recursion

Syntax	**Translation**
`mutual-recursion`	(mutual-recursion
{ $function_1$;	$function_1$
...	...
$function_n$; }	$function_n$)

Example

```
(mutual-recursion
 (defun evenlp (x)
   (if (consp x) (oddlp (cdr x)) t))
 (defun oddlp (x)
   (if (consp x) (evenlp (cdr x)) nil)))
```

Prints as:

```
mutual-recursion {

 function evenlp(x)
 { if consp(x) then oddlp(cdr(x))
     else t }    ;

 function oddlp(x)
 { if consp(x) then evenlp(cdr(x))
     else nil }    ;
 }   ;
```

9.5.2 DEFUNS

Syntax	**Translation**
See Mutual recursion above.	(DEFUNS def1 ... defn)

Example

```
(defuns
 (evenlp (x)
   (if (consp x) (oddlp (cdr x)) t))
 (oddlp (x)
   (if (consp x) (evenlp (cdr x)) nil)))
```

Prints as:

```
mutual-recursion {
  function evenlp(x)
        { if consp(x) then oddlp(cdr(x))
                else t }  ;
  function oddlp(x)
        { if consp(x) then evenlp(cdr(x))
                else nil }  ;
 }  ;
```

9.6 Include (INCLUDE-BOOK)

Example

```
(include-book "sets")
```

Prints as:

```
include("sets");
```

Parses to:

```
(include-book "sets")
```

10 Commands

10.1 Verify Guards

Syntax	**Translation**
`verify guards` *id* [`dcl .. dcl`] ;	(verify-guards *id* :hints hints :otf-flag otf-flag :doc *string*)

Example

```
(verify-guards flatten
              :hints (("Goal" :use (:instance assoc-of-app)))
```

```
               :otf-flg t
               :doc "string")
```

Prints as:

```
/* string */
verify guards flatten
 :hints [["Goal" :use [:instance assoc-of-app ]]]
 :otf-flag t;
```

10.2 Verify Termination

Syntax	**Translation**
`verify termination` *id* [`dcl .. dcl`] ;	(verify-termination (fn dcl ... dcl))

Example

```
(verify-termination  fact
 (declare (xargs :guard (and (integerp x) (>= x 0)))))
```

Prints as:

```
verify termination fact
      :guard integerp(x) & x >= 0;
```

10.3 Other Commands

Commands typed to Iacl2 must be terminated by a ;. A *cd* is a *command descriptor*.

Iacl2	ACL2
:*cmd cd*;	:*cmd cd*
:$cmd_2(id)$;	:*cmd2 id*

cmd ::= `PBT` | `PC` | `PCB` | `PCB!` | `PR` | `PUFF` | `PUFF*` | `UBT` | `UBT!`
cmd2 ::= `PE` | `PE!` | `PL` | `PR`

cd ::= `:N` | `:MAX` | `:MIN` | `:X` | `fn` | `[:X -k]` | `[`*cd* `n]` |
`[:SEARCH pat cd1 cd2']` | `[:SEARCH pat]`

Examples

```
:u;
:oops;
:pbt 12;
```

```
:pbt [:x -3];
:ubt 0;
:ubt [:search foo];

:pe fn;
:pf fn;
:pf (:definition,fn);
:pf (:rewrite,fn);
:pcs 12 14;
```

Iacl2	ACL2
:*cmd cd*;	:*cmd cd*
:$cmd_2(id)$;	:*cmd2 id*

cmd ::= `PBT` | `PC` | `PCB` | `PCB!` | `PR` | `PUFF` | `PUFF*` | `UBT` | `UBT!`
cmd2 ::= `PE` | `PE!` | `PL` | `PR`

cd ::= `:N` | `:MAX` | `:MIN` | `:X` | `fn` | `[:X -k]` | `[`*cd* `n]` |
`[:SEARCH pat cd1 cd2']` | `[:SEARCH pat]`

Examples

```
:u;
:oops;
:pcs 12 14;

:pf [fn];
:pf [:definition fn];
:pf [:rewrite fn];

:pbt [:x -3];
:ubt [:search foo];
```

A Arrays

Array operations just translate into standard function calls.

A.1 Aref

Functions suffixed with a '1' deal with one dimensional arrays and functions suffixed with a '2' deal with two dimensional arrays.

Syntax	**Translation**
aref1(*id*, *alist*, *i*)	(aref1 in alist i)
aref2(*id*, *alist*, *i*, *j*)	(aref2 id alist i j)

Example

```
(aref1 'demo (@ a) 0)
```

Prints as:

```
aref1(demo, @a, 0)
```

A.2 Arrayp

Syntax	**Translation**
array1p(*id*, *alist*)	(array1p id alist)
array2p(*id*, *alist*)	(array2p id alist)

Example

```
(array1p 'demo (@ a))
```

Prints as:

```
Array1p(demo, @a)
```

A.3 Aset

Syntax	**Translation**
aset1(*id*, *alist i*, *value*)	(aset1 id alist i val)
aset2(*id*, *alist*, *i*, *j*, *value*)	(aset2 id alist i j val)

Example

```
(assign b (aset1 'demo (@ a) 1 'one))
```

Prints as:

```
cb := Aset1(demo, @a, 1, 'one)
```

A.4 Compress

Syntax	**Translation**
compress1(*id*, *alist*)	(compress1 id alist)
compress2(*id*, *alist*)	(compress2 id alist)

Example

```
(assign a (compress1 'demo
                    '((:header :dimensions (5)
                               :maximum-length 15
                               :default uninitialized
                               :name demo)
                      (0 . zero))))
```

Prints as:

```
a := Compress1(demo, '[[:header :dimensions[5]
                      :maximum-length 15
                      :default uninitialized
                      :name demo]
                      [0 . zero]]
              );
```

A.5 Other Array Functions

Iacl2:	ACL2 Syntax:	ACL2:
`default(`*demo*`, @a)`	`(default name alist)`	`(default 'demo (@ a))`
`dimensions(`*demo*`, @a)`	`(dimensions name alist)`	`(dimensions 'demo (@ a))`
`header(`*demo*`,@a)`	`(header name alist)`	`(header 'demo (@ a)`
`maximun-length('demo, @a)`	`(maximum-length name alist)`	`(maximun-length 'demo (@ a))`

B Printed, But Not Parsed

The following events cannot be input directly into Iacl2. You need to create ACL2 books to support the the use of these advanced features.

B.1 Equivalence relation

B.1.1 DEFCONG

Syntax	**Translation**
(defcong equiv1 equiv2 *term* k :rule-classes rule-classes :instructions instructions :hints hints :otf-flg otf-flg :event-name event-name :doc doc)	

Example

```
(defcong set-equal iff (memb x y) 2
        :rule-classes (:rewrite))
```

Prints as:

```
Equivalence relation: SET-EQUAL preserves IFF
 for argument position 2 of MEMB(X, Y)
 :rule-classes [:REWRITE];
```

Does not parse.

B.1.2 DEFEQUIV

Syntax	**Translation**
(defequiv fn :rule-classes rule-classes :instructions instructions :hints hints :otf-flg otf-flg :event-name event-name :doc doc)	

Example

```
(defequiv set-equal
```

```
        :rule-classes (:rewrite))
```

Prints as:

```
Equivalence relation: SET-EQUAL
 :rule-classes [:REWRITE];
```

Does not parse.

B.2 Evaluator

Syntax	Translation
Evaluator (ev, ev-list)	(defevaluator ev ev-list
$[g_1 x_1 ... x_{n1},$	$((g_1 x_1 ... x_{n1})$
...	...
$g_k x_1 ... x_{nk}$]	$(g_k x_1 ... x_{nk})))$

Example

```
(defevaluator evl evl-list
  ((length x) (member x y)))
```

Prints as:

```
Evaluator (evl, evl-list)  [ length(x), x in y  ];
```

Does not parse.

B.3 Refinement (DEFREFINEMENT)

Syntax	Translation
(defrefinement equiv1 equiv2 :rule-classes rule-classes :instructions instructions :hints hints :otf-flg otf-flg :event-name event-name :doc doc)	

Example

```
(defrefinement equiv1 equiv2)
```

Prints as:

```
Refinement EQUIV1 refines EQUIV2;
```

B.4 CHOOSE (DEFCHOOSE)

Syntax **Translation**

(DEFCHOOSE fn (bound-v_1 ... bound-v_n)
(free-v_1 ... free-vark)
body)

Example

```
(defchoose choose-x-for-p-and-q (x) (y z)
  (and (p x y z)
       (q x y z)))
```

Prints as:

```
Choose choose-x-for-p-and-q(x) (y, z)
{ p(x, y, z)  &  q(x, y, z) } ;
```

B.5 SET and TABLE Events

All these events are printed and parsed as function calls.

Infix		Translation
set-compile-fns(*term*);	⇔	(set-compile-fns *term*)
set-ignore-ok(flg);	⇔	(set-ignore-ok flg)
set-inhibit-warnings(s1 s2);	⇔	(set-inhibit-warnings s1 s2)
set-invisible-fns-alist(alist);	⇔	(set-invisible-fns-alist alist)
set-irrelevant-formals-ok(flg);	⇔	(set-irrelevant-formals-ok flg)
set-measure-function(name);	⇔	(set-measure-function name)
set-verify-guards-eagerness(n);	⇔	(set-verify-guards-eagerness n)
set-well-founded-relation(rel);	⇔	(set-well-founded-relation rel)

B.5.1 Tables

Syntax **Translation**

(table table-name key-term value-term op *term*)

Examples

INFIX ACL2

```
table(tests, 25);                         (table tests 25)
table(tests);                             (table tests)
table(tests, nil, nil, :clear);           (table tests nil nil :clear)
table(tests, nil, foo(7),:clear);         (table tests nil (foo 7) :clear)
```

`table(tests, nil, nil,:guard);`	`(table tests nil nil :guard)`
`table(tests, nil, nil,:guard` *term*`);`	`(table tests nil nil :guard` *term*`)`