

*Using ParentheC to Transform Scheme
Programs to C or How to Write Interesting
Recursive Programs in a Stack-Based,
Imperative, Inhospitable Host (Registerization)*

Ronald Garcia, Jeremy G. Siek, Ruj Akavipat, William Byrd,
Samuel Chun, David W. Mack, Alex Platte, Heather Roinessad,
and Daniel P. Friedman

*Computer Science Department, Indiana University
Bloomington, IN 47405, USA*

1 Introduction

Many interesting Scheme programs can be transformed into equivalent C programs. The process involves applying provably correct transformations to a correct Scheme program, converting it step-by-step from one form to another. At each stage the program runs exactly as it did at the beginning. After applying these transformations, the result is the same Scheme program but in registerized form. The final step of the transformation to C is a relatively trivial rewrite of Scheme syntax into C.

Conversion to C is simple-minded, but unfortunately it is not easy. The resulting C code is generally quite verbose and relatively complex. The translation is mechanical, but mistakes in translation are easy to make and quite difficult to locate. Experience has shown in the presence of a single mistake, it is better to start the process from scratch.

The *ParentheC/reg* language, which contains only four simple forms, is one approach to avoiding the tedious translation from registerized Scheme to C. Manually converting Scheme to *ParentheC/reg* is easy and the rest of the work has been *automated*.

The problem, of course, would be much simpler if C did not blowup when cascading many recursive function calls. This is a situation that occurs when running an interpreter, for example. More importantly, even if the code is in first-order tail form, we still have to contend with cascading tail calls. Of course, these tail calls can be eliminated by either using trampolining or by using *gotos* with global register assignments. Here, we choose the registerization approach.

While transforming the Scheme program to registerized form, we methodically introduce places where we can use *ParentheC/reg*. When we have reached tail recursive and registerized forms, the entire program will have been transformed to *ParentheC/reg*, which is completely testable in Scheme. Finally, we can run the *pc2c* translator, which produces a valid C program that yields the same results as the original Scheme program.

The structure of the paper is in four sections. First, we introduce the language `ParentheC/reg`, which is a subset of Scheme plus four simple macros. Second, we work through a simple example starting with a recursive definition and taking it all the way to C. This process entails making the code be in first-order tail form. That means that all calls are tail calls to global functions and there are no lambda expressions. Once we have reached this stage, we registerize the program. Then we feed the `ParentheC/reg` file to `pc2c`, which creates a C program (in two files). We end this section by showing how the C code corresponds to the `ParentheC/reg` code. Next, we describe the operation of the four `ParentheC/reg` macros. In the last section, we offer some conclusions.

2 The Language `ParentheC/reg`

`ParentheC/reg` is a subset of Scheme with four special forms: `define-registers`, `define-label`, `define-union`, and `union-case`. Here is the grammar:

```

<Prog>      → <Expr> <Expr>*
<Expr>      → (define-registers <var>*)
              | (define-label <label> <Complex>)
              | (define-union <label> (<label> <var>*)*)
<Complex>   → <Simple>
              | (error <string>)
              | (if <Simple> <Complex> <Complex>)
              | (cond (<Simple> <Complex>)+)
              | (cond (<Simple> <Complex>)* (else <Complex>))
              | (begin <Complex>*)
              | (set! <var> <Simple>)
              | (union-case <var> <label>
                  ((<label> <var>*) <Complex>)*
              | (let ((<var> <Simple>)* <Complex>)
                  (<label>))
<Simple>    → #t
              | #f
              | <var>
              | <integer>
              | (<1-Prim> <Simple>)
              | (<2-Prim> <Simple> <Simple>)
              | (if <Simple> <Simple> <Simple>)
              | (<label> <Simple>*)
<1-Prim>    → zero? not sub1 add1 error
<2-Prim>    → + - * / and or < > <= >= random

```

It should be clear from this grammar that many features of standard Scheme are not available. The only immediate data types available are boolean and integer values. Pairs and vectors are *not* available. They are replaced by the record operators

`define-union` and `union-case`. The allowable primitives include a zero predicate, addition, subtraction, multiplication, division, `subtract-1`, and `add-1`.

2.1 Defining Functions (Labels)

We start with `define-label`, which is not strictly necessary in `ParentheC/reg`, since its use is the same as one of the syntaxes of `define`. We require its use, because it signals that the body of the function being defined is in first-order tail form and that it *requires* that the function consumes no arguments. Hence, it is better considered to be a *label* for some code rather than a function.

For example, the code for `add` would be defined (assuming that it has been registerized) in `ParentheC/reg` as

```
(define-label add (+ x y))
```

This is equivalent to the standard Scheme version

```
(define add (lambda () (+ x y)))
```

In fact, the `ParentheC/reg` macro package for Scheme expands `define-label` into precisely this format.

2.2 Unions

`ParentheC/reg` unions express those portions of a Scheme program represented using records. Making a set of Scheme program objects, such as continuations, be representation independent involves two transformations. First, the expressions that construct each *variant* are replaced with calls to *constructor* functions. Free variables become parameters to each constructor function, and the expression as a whole becomes its body. Then the locations in the source code where these objects are consumed are replaced with calls to an `apply_` function, a function that knows what to do with these variants. If the variants in question are procedures, for example, the function applies the procedure to its other parameters.

To transform this form to a record-based format, each of the trivial constructor functions is modified to return a tagged list containing a type tag, its variant tag, and the parameters to the function. Then the `apply_` function is modified to compare against each possible tagged list form. The action clause for each tagged list corresponds to the actions performed by the procedural form. The following Scheme code defines three constructor functions. The body of each constructor function returns a tagged list. The name of the constructor function is formed by concatenating the type to the variant separated by `_`.

```
(define record_a
  (lambda (item1 item2)
    '(record a ,item1 ,item2)))
```

```
(define record_b
  (lambda (item1)
    '(record b ,item1)))

(define record_c
  (lambda (v)
    '(record c ,v)))
```

These definitions are built using `define-union`. It creates trivial definitions like those given above.

```
(define-union record
  (a item1 item2)
  (b item1)
  (c v))
```

There are a couple restrictions not directly imposed by this definition. First, the tags must form a set of symbols, and second the variables must form a set of lexical variables represented with symbols. These restrictions will lead to an error if they are violated.

ParentheC/reg unions created with constructors built from `define-union` are consumed using `union-case`. Using `union-case` on a particular type before the associated type has been defined is strictly forbidden, the initial `<var>` position must be a lexical variable, the `<label>`s from each clause must form a set of symbols, the `<var>`s must form a set of lexical variables, and the bodies must be legitimate expressions.

Consider this usage of `union-case` for this example.

```
(define-label process_record
  (union-case r record
    [(a item1 item2) (+ item1 item2)]
    [(b item1) (sub1 item1)]
    [(c v) (* v 7)]))
```

Clearly this use of `define-union` and `union-case` is analogous to explicit tagged list constructors and a pattern matcher.

2.3 Registers

ParentheC/reg supports an additional construct `define-registers` to define the global registers used by the registerized functions. Its use is straightforward:

```
(define-registers reg1 reg2 reg3)
```

This is necessary in order to allow the C version to define global variables to be used as registers.

2.4 Properties

As characterized in the language grammar above, ParentheC/reg supports a limited subset of the Scheme language. There are a number of important properties that may not be obvious.

1. The program must be tail recursive and it must have been registerized.
2. All functions consume no arguments except for union constructors. This restriction is a function of the registerization of the program. Union constructors are created by `define-union` and may take zero or more arguments.
3. A function `main` of no arguments *must* be defined. This is where execution will start in the C program.
4. Certain Scheme functionality may only be used in certain contexts. This is characterized by the “Simple” and “Complex” contexts in the grammar above.
5. ParentheC/reg places no restrictions on variable and label names. Any valid Scheme symbol may be used. Symbols with special characters will be renamed in the C code.

3 A fully-worked example

The following example illustrates how to use ParentheC/reg to transform Scheme programs to C programs. Along the way, we demonstrate how to registerize while expressing the result as a ParentheC/reg program. Then we show the resulting C code. We use `factorial` below as our example program. These transformations are overkill for such a program. When a program is complicated, however, these manual transformations are a good way to maintain control over the host (in this case C) language’s underlying runtime architecture.

Finally, we show how to execute ParentheC/reg programs in Scheme and illustrate the conversion from ParentheC/reg to C.

3.1 Converting Scheme to ParentheC/reg

Here is the standard recursive definition of `factorial`.

```
(define factorial
  (lambda (n)
    (cond
      [(zero? n) 1]
      [else (* n (factorial (sub1 n)))])))

(define main
  (lambda ()
    (factorial 5)))
```

```
> (main)
120
```

The function `main` encapsulates the start of our computation. It begins program execution in the resulting C program. In order for ParentheC/reg programs to be translated to C, there *must* be a global definition of `main`.

Because `factorial` is not even in tail form, we transform it into continuation-passing style. Any approach that would transform it to tail form would suffice, and we are just using one such technique.

```
(define factorial
  (lambda (n)
    (factorial_cps n (lambda (v) v))))

(define factorial_cps
  (lambda (n k)
    (cond
      [(zero? n) (k 1)]
      [else (factorial_cps (sub1 n) (lambda (v) (k (* n v))))])))
```

Our next goal is to remove the higher-order attributes from these two definitions. Our eventual host is C, which only supports first-order global representations. Therefore, we use a two-step process to enforce this restriction. First we make these two definitions first-order, which makes them representation independent with respect to the introduced continuations. The code still uses higher-order values, but they are hidden in `apply_k` and its associated constructors.

```
(define factorial
  (lambda (n)
    (factorial_cps n (kt_empty))))

(define factorial_cps
  (lambda (n k)
    (cond
      [(zero? n) (apply_k k 1)]
      [else (factorial_cps (sub1 n) (kt_extend n k))])))

(define apply_k
  (lambda (k^ v)
    (k^ v)))

(define kt_empty
  (lambda ()
    (lambda (v) v)))

(define kt_extend
  (lambda (n k)
    (lambda (v)
      (apply_k k (* n v)))))
```

Now that the representation dependency is only in the `apply_k` function and its associated constructors, we can choose a different representation for the continuation values. Once the new representation is chosen, however, we must also change the definition of `apply_k`.

```
(define-union kt
  (empty)
  (extend n k))

(define apply_k
  (lambda (k^ v)
    (union-case k^ kt
      [(empty) v]
      [(extend n k) (apply_k k (* n v))])))
```

At this point, we know that our program is in first-order tail form. That is, there are no function values and all calls are tail calls. (Another example would be `factorial` written in accumulator-passing style.) Any time we have this property we can either turn tail calls into `gotos` while passing arguments through global registers or we can trampoline the program. Here, we choose to registerize it so we can use `gotos`.

We begin by registerizing the calls to `factorial` and `factorial_cps`.

```
(define factorial
  (lambda ()
    (begin
      (set! n n)
      (set! k (kt_empty))
      (factorial_cps))))

(define factorial_cps
  (lambda ()
    (cond
      [(zero? n) (apply_k k 1)]
      [else (begin
              (set! k (kt_extend n k))
              (set! n (sub1 n))
              (factorial_cps))])))

(define main
  (lambda ()
    (begin
      (set! n 5)
      (factorial))))
```

The ordering of `set!` statements in each call is critical. For example, in the “else” clause of the `cond` statement, `k` *must* be set before `n`. Otherwise, `n` would get its

new value and `k` would be assigned to the new value rather than the old (correct) one.

We finish this process by registerizing `apply_k`. We don't registerize the union constructors because they are all trivial functions that are guaranteed to terminate.

```
(define-union kt
  (empty)
  (extend n k))

(define apply_k
  (lambda ()
    (union-case k^ kt
      [(empty) v]
      [(extend n k) (begin
                      (set! k^ k)
                      (set! v (* n v))
                      (apply_k))])))

(define factorial
  (lambda ()
    (begin
      (set! n n)
      (set! k (kt_empty))
      (factorial_cps))))

(define factorial_cps
  (lambda ()
    (cond
      [(zero? n) (begin
                  (set! k^ k)
                  (set! v 1)
                  (apply_k))]
      [else (begin
              (set! k (kt_extend n k))
              (set! n (sub1 n))
              (factorial_cps))]))))

(define main
  (lambda ()
    (begin
      (set! n 5)
      (factorial))))
```

Finally, we are ready to translate this code into `ParentheC/reg` by replacing function definitions—which now have no arguments—with `define-label` and by adding the `define-registers` construct.

```
(define-registers n k k^ v)

(define-union kt
  (empty)
  (extend n k))

(define-label apply_k
  (union-case k^ kt
    [(empty) v]
    [(extend n k) (begin
                     (set! k^ k)
                     (set! v (* n v))
                     (apply_k))]))

(define-label factorial
  (begin
    (set! n n)
    (set! k (kt_empty))
    (factorial_cps)))

(define-label factorial_cps
  (cond
    [(zero? n) (begin
                 (set! k^ k)
                 (set! v 1)
                 (apply_k))]
    [else (begin
            (set! k (kt_extend n k))
            (set! n (sub1 n))
            (factorial_cps))]))

(define-label main
  (begin
    (set! n 5)
    (factorial)))
```

By CPSing, registerizing, and using the four ParentheC/reg constructs, we have produced a ParentheC/reg program ready to be transformed into C.

3.2 Executing ParentheC/reg programs from Scheme

The following is an execution of our ParentheC/reg definition of `factorial`.

```
> (load "ParentheC.ss")
> (load "fact5.pc")
> (main)
120
```

Loading `ParentheC.ss` prepares Scheme to execute ParentheC/reg programs. The file `fact5.pc` contains the implementation of `factorial` in ParentheC/reg shown above. Calling `main` runs the `factorial` program. After making changes to your ParentheC program, you must restart Scheme prior to running it with `ParentheC.ss` again.

3.3 Transforming ParentheC/reg programs to C with `pc2c`

The file `pc2c.ss` implements a translator that produces C source from ParentheC/reg programs. The function `pc2c` takes three strings as parameters. The first parameter is the name of the ParentheC/reg file to be translated to C. The next two parameters name the two output files that `pc2c` generates. The first output file names the C source file and should have the extension `.c`. The second output file names the C header file and should have the extension `.h`.

Here is the process by which a ParentheC/reg program is translated into C. Suppose the derived `factorial` program above is the file `fact5.pc`. Then it can be translated into the files `fact5.c` and `fact5.h` below.

```
> (load "pc2c.ss")
> (pc2c "fact5.pc" "fact5.c" "fact5.h")
```

The procedure `pc2c` reads the ParentheC/reg program from the file `fact5.pc` and creates two files `fact5.c` and `fact5.h`.

To run the resulting C program, first compile it. The following directions describe how to carry out this process from a Unix-based system. Compile the program from a shell prompt using

```
% cc fact5.c -o fact5
```

The command `cc` runs the C compiler. The file `fact5.c` is the C program to be compiled. The file `fact5.h` contains information that directs the C compiler to use the declarations in `fact5.h`. The parameters `-o fact5` tell the C compiler to name the resulting executable file `fact5`. Now, from the shell prompt, we can run the resulting executable.

```
% ./fact5
120
```

When `factorial` was executed using `ParentheC.ss`, the `main` function was called after loading `fact5.pc` into the Scheme system. The C translation of ParentheC/reg, however, automatically calls `main` when it is executed.

There is a function `compile/run` in `pc2c.ss` that performs all of these actions for you. It takes one argument, the name of your ParentheC/reg program without the `.ss` suffix.

```
> (load "pc2c.ss")
> (compile/run "fact5")
120
```

The Scheme code that accomplishes this task is quite simple. It converts the ParentheC/reg file to C code, compiles it, and, if compilation succeeded, runs the resulting program.

```
(define compile/run
  (lambda (base-name)
    (let ([pc-file (string-append base-name ".ss")]
          [c-file (string-append base-name ".c")]
          [header-file (string-append base-name ".h")])
      (pc2c pc-file c-file header-file)
      (let ([compile-command (string-append "gcc -o " base-name " " c-file)]
            [indent-command (string-append "indent -bad -bap -br -nbs -ce"
                                             "-cli1 -di1 -i2 -npro -npsl ")])
        (let ([compile-val (system compile-command)])
          (unless (zero? compile-val)
            (error 'compile/run "Compilation command '~s' returned ~s"
                   compile-command compile-val))
          (system (string-append indent-command c-file))
          (system (string-append indent-command header-file))
          (system (string-append "./" base-name))
          (void))))))
```

3.4 Factorial in C

The resulting C code for `factorial` is significantly longer and more verbose than the ParentheC/reg implementation. This code growth alone suggests that ParentheC/reg can save much effort when transforming registerized Scheme to C.

The C source generated by `pc2c` is not easy to read. On Unix systems, there exists a pretty printing tool for C called `indent` that reformats C files. By running this indenting tool over the C source, browsing the code becomes possible. The following command line formats the `fact5.c` file.

```
% indent fact5.c
```

We need to say just a few words about the header file below. The first line of declarations defines the global registers that were created with `define-registers`. Notice that the register `k^` has been replaced with `kr__ex__` because C does not support the `^` character.

The next declarations are for the two union types. Let's look closely at building

the union type `kt`. We can see that the number of lines in the `union` is the same as the number of lines in the `enum` (enumeration). This just separates the tags from the field names of the record. The entire record `kt`, itself, contains the `enum`, which is accessed through the name `tag` and the `union`, which is accessed through the name `u`. The most important thing to notice, however, is that every field within a union has type `void*`. This is a pointer to a type of void. This particular type in C allows us to pass an integer value or a pointer to a union type.

Finally, the header file defines the function `go` where all of the work of the registered program actually occurs.

Here is the header file, `fact5.h`, generated by `pc2c` from `fact5.pc`.

```
void *n, *k, *kr__ex__, *v;

struct kt;
typedef struct kt kt;
struct kt {
    enum {
        _empty_kt,
        _extend_kt
    } tag;
    union {
        struct { char dummy; } _empty;
        struct { void *_n; void *_k; } _extend;
    } u;
};

void *ktr_empty();
void *ktr_extend(void *n, void *k);

void *go();
```

We make a few observations that should aid in understanding the generated C function definitions below. The `{ ... }` wrapped around a sequence of commands acts as a Scheme `begin` expression. The character `=` is used as a binary assignment operator. The item on the left is a typed container for the item on the right. The `switch` statement dispatches on the kind of `tag` it finds. The `goto` command transfers execution to a particular label. Labels are specified on a line with an identifier and a colon (e.g., `main:`).

If we look closely at the definition of the function `main`, we see that it calls the function `go` and prints out the resulting value. The definitions of the two union constructors follow. They are simply functions that create data structures and return them to the calling program.

Everything else of interest lies in the function `go`. Our factorial function was registered, so we don't create individual functions for each of them. Each function is specified within `go` by a label denoting its name. The first instruction in `go` is to jump to the label `main`. The contents of each label is now enclosed by a set of curly

braces { ... }. You can see that all `set!` assignments have been changed into C style = assignments. All non-constructor procedure calls are now simple `gotos`.

And, here is the source file, `fact5.c`, generated by `pc2c` from `fact5.pc`.

```
#include <stdio.h>
#include "fact5.h"

int main(void) {
    printf("%d\n", (int)go());
    return 0;
}

void *ktr_empty() {
    kt* _data = (kt*)malloc(sizeof(kt));
    if(!_data) {
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }
    _data->tag = _empty_kt;
    return (void *)_data;
}

void *ktr_extend(void *n, void *k) {
    kt* _data = (kt*)malloc(sizeof(kt));
    if(!_data) {
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }
    _data->tag = _extend_kt;
    _data->u._extend._n = n;
    _data->u._extend._k = k;
    return (void *)_data;
}
```

```

void *go() {
    goto main;
}
applyr_k:
{
    kt* _c = (kt*)kr__ex__;
    switch (_c->tag) {
        case _empty_kt: {
            return((void *)v);
        }
        case _extend_kt: {
            void *n = _c->u._extend._n;
            void *k = _c->u._extend._k;
            kr__ex__ = (void *)k;
            v = (void *)((int)(void *)n * (int)(void *)v);
            goto applyr_k;
        }
    }
}
}
}
factorialr_cps:
{
    if(((void *)n == 0)) {
        kr__ex__ = (void *)k;
        v = (void *)1;
        goto applyr_k;
    } else {
        k = (void *)ktr_extend((void *)n, (void *)k);
        n = (void *)((int)(void *)n - 1);
        goto factorialr_cps;
    }
}
}
main:
{
    n = (void *)5;
    goto factorial;
}
factorial:
{
    n = (void *)n;
    k = (void *)ktr_empty();
    goto factorialr_cps;
}
}

```

4 Macros for ParentheC/reg

What follows is a discussion of the macros used to run ParentheC/reg programs within Scheme.

4.1 Defining Registers

We begin with `define-registers`, a ParentheC/reg primitive that defines the global registers added during the process of registerization.

```
(define-syntax define-registers
  (syntax-rules ()
    [(_ ids* ...) (begin (define ids* 0) ...)]))
```

This is equivalent to a series of `define` statements in Scheme. If we load `expand-only`, we can see exactly what this macro does.

```
> (load "expand-only.ss")
> (expand-only '(define-registers)
  '(define-registers a b c))

(begin (define a 0) (define b 0) (define c 0))
```

4.2 Defining Functions (Labels)

The `define-label` ParentheC/reg primitive is not strictly necessary, since it is equivalent to one of the forms of `define`. We require its use primarily because it enforces the restriction that all functions take zero arguments.

```
(define-syntax define-label
  (syntax-rules ()
    [(_ fn body body* ...)
     (define fn (lambda () body body* ...))]))
```

We return to our example of the usage of `define-label` above and again expand the syntax with `expand-only`.

```
> (expand-only '(define-label)
  '(define-label add (+ x y)))

(define add (lambda () (+ x y)))
```

4.3 Unions

As discussed above, unions allow us to use a record structure within our ParentheC/reg programs, since lists are not permitted. In general, records have *constructors* and functions that apply their results, such as `apply_k`. The macro `define-union` generates the constructors.

```
(define-syntax define-union
  (syntax-rules* ()
    [(_ type-name [tag f* ...] ...)
     (begin
       (define type-name_tag
         (lambda (f* ...)
           '(,type-name ,tag ,f* ...)))
       ...))])
```

This creates trivial definitions of record constructors. Again, we expand a simple example.

```
> (expand-only '(define-union)
  '(define-union record
    (a item1 item2)
    (b item1)
    (c v)))
(begin
  (define record_a
    (lambda (item1 item2)
      '(record a ,item1 ,item2)))
  (define record_b
    (lambda (item1)
      '(record b ,item1)))
  (define record_c
    (lambda (v)
      '(record c ,v))))
```

These constructor definitions and the record types created by `define-union` are consumed by the macro `union-case`, defined here.

```
(define-syntax union-case
  (syntax-rules ()
    [(_ var type-name [(tag f* ...) body body* ...] ...)
     (if (not (union-type? 'type-name))
         (error 'union-case "~s is not a union type." type-name)
         (case (cadr var)
           [(tag) (apply (lambda (f* ...) body body* ...) (caddr var))]
           ...))
```

Consider how `union-case` would expand on this example.

```
> (expand-only '(union-case)
  '(define-label process_record
    (union-case r record
      [(a item1 item2) (+ item1 item2)]
      [(b item1) (sub1 item1)]
      [(c v) (* v 7)])))
```

```
(define-label process_record
  (if (not (valid-variant? 'record r))
      (error 'union-case "~s is not a union type ~s." r 'record)
      (case (cadr r)
          [(a) (apply (lambda (item1 item2) (+ item1 item2)) (caddr r))]
          [(b) (apply (lambda (item1) (sub1 item1)) (caddr r))]
          [(c) (apply (lambda (v) (* v 7)) (caddr r))])))
```

5 Conclusion

There have been three goals in this tutorial. First, we have demonstrated a technique for writing interesting programs in an inhospitable host. Second, we have presented a small enough example that should encourage the study of the inhospitable host, itself. In our example, the host has been C, but any language, including even assembly language or Java, might be considered an inhospitable host. Third, we have shown how to write a few interesting macros to support this system.

We have observed this technology in action. This example should give a feel for how the header file declarations work with the other code. Studying this example thoroughly should clarify how to implement complicated programs, for example interpreters, in a stack-based, imperative, inhospitable language like C.

6 Acknowledgements

We would like to thank Oleg Kiselyov for commenting on and critiquing our approach. His observations resulted in important improvements to `ParentheC`.

7 Appendix

We redefine the macros but this time writing them in a nearly bullet-proof way. Most of these are easy to understand if there is focus on the input pattern and the last expression. Everything between those points are either to extend global data structures or to check for invalid input. It is nearly impossible to check for every possible syntactic error, so as an exercise try to come up with such errors.

The first definition is `define-label` below. It maintains a global list of function names `**pc-func-name-table**` that it has built, and it forces an error if the same function name is redefined. This error occurs at macro-expansion time through calls to `pc-error-check:---`. Otherwise, it has the same semantics as the previous version.

```
(define **pc-func-name-table** '())

(define pc-add-func-name!
  (lambda (func-name)
    (set! **pc-func-name-table**
      (cons func-name **pc-func-name-table**))))

(define pc-func-name-exists?
  (lambda (fn)
    (memv fn **pc-func-name-table**)))

(define-syntax pc-err
  (syntax-rules ()
    [(_ who code (str arg* ...))
     (begin
      (printf "%nParentheC Error - In Expression:%n%n")
      (pretty-print code)
      (error who str arg* ...))]))

(define-syntax define-label
  (lambda (x)
    (pc-error-check:define-label (syntax-object->datum x))
    (syntax-case x ()
      [(_ fn body* ...)
       (pc-add-func-name! (syntax-object->datum #'fn))
       #'(define fn (lambda () body* ...))]))))
```

```

(define pc-error-check:define-label
  (lambda (code)
    (match code
      [(define-label ,fn)
       (pc-err 'define-label code ("must have at least one body"))]
      [(define-label (,fn ,p* ...) ,body)
       (pc-err 'define-label code ("cannot have any parameters"))]
      [(define-label ,fn ,body ,body* ...)
       (if (pc-func-name-exists? fn)
           (pc-err 'define-label code
                    ("function name ~s already exists" fn))]
           [any (pc-err 'define-label code ("invalid syntax"))])]
    ))

(define pc-check-vars
  (let ([pc-identifier?
        (lambda (x)
          (and (symbol? x)
               (let ([ch* (string->list (symbol->string x))]
                     (and
                      (char-alphabetic? (car ch*))
                      (andmap
                       (lambda (c)
                         (or (char-alphabetic? c) (char-numeric? c) (eqv? c '#_)))
                       (cdr ch*)))))))]
        (lambda (who code vars)
          (for-each (lambda (id)
                      (if (not (pc-identifier? id))
                          (pc-err who code ("invalid ParentheC identifier used: ~s" id))))
                    vars))))

(define pc-check-set-of-vars
  (letrec ([set-of-vars?
            (lambda (ls)
              (or (null? ls)
                  (and (not (memv (car ls) (cdr ls))) (set-of-vars? (cdr ls))))))]
          (lambda (who code vars)
            (if (not (set-of-vars? vars))
                (pc-err who code ("duplicate variable used: ~s" vars))))))

```

In `define-union`, in addition to the naive macro-expansion-time checks, we also have some subtle ones. Specifically, we have to make sure that we are not recreating the same type. We do this by maintaining a global association list `**pc-union-type-table**` with each list entry list being of the form: `[union-type`

([tag . arity-of-constructor] ...)]. Then we can determine if we are recreating a type with `pc-union-type-exists?` below. We also need to create run-time tests to make sure that the right number of arguments are being passed to its constructors.

```
(define **pc-union-type-table** '())

(define pc-add-union-type!
  (lambda (union-type sub-tn* arg-count*)
    (set! **pc-union-type-table**
      (cons '(,union-type ,(map cons sub-tn* arg-count*))
        **pc-union-type-table**))))

(define pc-union-type-exists?
  (lambda (union-type)
    (assv union-type **pc-union-type-table**)))

(define-syntax define-union
  (lambda (x)
    (pc-error-check:define-union (syntax-object->datum x))
    (syntax-case x ()
      [(_ union-type [sub-tn* arg** ...] ...)
       (let ([ut-val (syntax-object->datum #'union-type)]
             [st*-val (syntax-object->datum #'(sub-tn* ...))]
             [arg-count*-val (map length (syntax-object->datum #'(arg** ...) ...))])
         (with-syntax
          ([[constructor-fn* ...]
            (datum->syntax-object #'_
              (map (lambda (st-val)
                     (string->symbol (format "~s_~s" ut-val st-val)))
                  st*-val))])
          [[arg-count* ...]
            (datum->syntax-object #'_ arg-count*-val)])
          (pc-add-union-type! ut-val st*-val arg-count*-val)
          #'(begin
              (define constructor-fn*
                (lambda n-arg
                  (if (eq? (length n-arg) arg-count*)
                      '(union-type sub-tn* ,@n-arg)
                      (pc-err 'constructor-fn* '(constructor-fn* ,@n-arg)
                        ("wrong number of arguments to constructor: expected ~s"
                          arg-count*))))))
              ...)))])))))
```

```

(define pc-error-check:define-union
  (lambda (code)
    (match code
      [(define-union ,union-type)
       (pc-err 'define-union code
        ("must have at least one sub-type in union-type: ~s" union-type))]
      [(define-union ,union-type [,sub-tn* ,arg** ...] ...)
       (pc-check-vars 'define-union code '(,union-type ,sub-tn* ...))
       (pc-check-set-of-vars 'define-union code sub-tn*)
       (for-each
        (lambda (arg*)
          (pc-check-vars 'define-union code arg*)
          (pc-check-set-of-vars 'define-union code arg*))
        arg**)
       (if (pc-union-type-exists? union-type)
           (pc-err 'define-union code
            ("union-type ~s already exists" union-type))]
           [,any (pc-err 'define-union code ("invalid syntax"))])]))))

```

The last macro `union-case` requires even more expansion-time tests. Most important of these tests, however, is that the order of clauses in a `union-case` corresponds to the order in the associated `define-union`. When the message `'no matching union-type exists'` appears either there are too few or too many clauses in the `union-case` expression. In addition, there is a run-time test using `pc-valid-variant?` that guarantees that the argument passed to the `union-case` is appropriate.

```

(define-syntax union-case
  (lambda (x)
    (syntax-case x ()
      [(_ exp union-type [(sub-tn* arg** ...) body** ...] ...)
       #'(general-union-case union-case exp union-type
          [(sub-tn* arg** ...) body** ...] ...)])))))

```

```

(define-syntax general-union-case
  (lambda (x)
    (let ([code (syntax-object->datum x)])
      (pc-error-check:general-union-case code (cadr code)))
    (syntax-case x ()
      [(_ label var union-type [(sub-tn* arg** ...) body** ...] ...)
       #'(let ([code '(label exp union-type [(sub-tn* arg** ...) body** ...] ...)])
           (if (not (pc-valid-variant? 'union-type var))
               (pc-err 'label code
                        ("invalid datum for union-type ~s: ~s" 'union-type var)))
           (case (cadr var)
             [sub-tn* (apply (lambda (arg** ...) body** ...) (caddr var))]
             ...
             [else (pc-err
                     'label code
                     ("It should never come here: ~s, ~s" var 'union-type))])))))]))

(define pc-valid-variant?
  (lambda (union-type variant)
    (and
      (list? variant)
      (>= (length variant) 2)
      (let ([ut (car variant)]
            [st (cadr variant)]
            [arg-count (length (caddr variant))])
        (and
          (eqv? union-type ut)
          (let ([type (assoc union-type **pc-union-type-table**)])
            (and type
                  (member '(,st . ,arg-count) (cadr type))))))))))

```

```

(define pc-error-check:general-union-case
  (lambda (code who)
    (match code
      [(general-union-case ,label ,var ,union-type)
       (pc-err who code ("all union-type must have at least one sub-type"))]
      [(general-union-case ,label ,var ,union-type
        [(,sub-tn* ,arg** ...) ,body** ...] ...)
       (pc-check-set-of-vars who code '(,var ,union-type))
       (pc-check-vars who code '(,union-type ,sub-tn* ...))
       (pc-check-set-of-vars who code sub-tn*)
       (for-each
        (lambda (arg*)
          (pc-check-vars who code arg*)
          (pc-check-set-of-vars who code arg*))
        arg**)
       (if (ormap null? body**)
           (pc-err who code
              ("all union-case clause must contain at least one body")))
       (pc-union-type-does-not-exist? who var union-type
        '(,sub-tn* ...) '((,arg** ...) ...) '((,body** ...) ...))]
      [,any (pc-err who code ("invalid syntax"))]))))

(define pc-union-type-does-not-exist?
  (lambda (who var union-type st* arg** body**)
    (let* ([arg-count* (map length arg**)]
           [sub-type* (map cons st* arg-count*)]
           [type '(,union-type ,sub-type*)])
      (if (not (member type **pc-union-type-table**))
          (begin
            (printf "%nParentheC Error - In Expression:%n%n")
            (pretty-print
             '(,who ,var ,union-type
              ,(map (lambda (st arg* body*)
                     (cons (cons st arg*) body*))
                   st* arg** body**)))
            (error who "no matching union-type exists"))))))))

```

Although these tests may seem rather expensive, there are two issues that we should consider. First, given that our goal is to create C programs, it hardly matters how expensive ParentheC/reg code is. Second, the expansion-time tests potentially save a lot of time for users of ParentheC/reg.