Gont Manual

September 2, 2002

`$Rev: 855 $ $LastChangedDate: 2002-09-01 23:39:07 +0200 (Sun, 01 Sep 2002) $`

Contents

Part I

Basics

1 Intro

Gont is programming language similar to C in some aspects, to Java in some other, and to ML in remaining.

This document is meant to give a brief introduction into Gont for someone with basic C experience.

1.1 Error tolerance

Gont always tries to ensure, you won't do anything harmful. However, as it isn't very bright, it might be sometimes annoying. Generally, it is something at the level of `gcc -Wall -Werror`, sometimes more, sometimes less.

1.2 Typographic conventions

Keywords are written `like this`.

Program examples and terminal output, is written using:

```
        monospaced font, indented in separate line.
```

1.3 Other notes

Words `foo`, `bar`, `baz` and `qux` used in examples are metasyntactic variables, they mean "any name".

2 Hello world

What can be first program we will write? There is only one choice... :) So startup your favorite editor and create file named *`hello.g`* with the following contents:

```
section init
{
        print_string("Hello world!\n");
}
```

Gont requires each module to have an interface. For example:

```
[malekith@roke gont-tut]$ ls
hello.g
[malekith@roke gont-tut]$ gontc hello.g
hello.g:0:1: cannot find module interface for Hello: file not found
[malekith@roke gont-tut]$
```

Interface should be in file under the same name as implementation file, but with *`.gi`* suffix. We will talk about modules more in Section 19 below. For now, we don't need anything special in interface, so we create it empty:

```
[malekith@roke gont-tut]$ echo > hello.gi
[malekith@roke gont-tut]$ gontc -c hello.gi
[malekith@roke gont-tut]$ gontc hello.g
[malekith@roke gont-tut]$ ./a.out
Hello world!
[malekith@roke gont-tut]$
```

It works! :)

3 Lexical conventions

Following example is list of valid Gont tokens (although without any sense :)

```
// comment
/* /* Unlike in C, */ comments can be nested. */
123     // decimal number
0x12a   // hexadecimal number
0o666   // octal number
0666    // *decimal* number
0b0110  // binary number
// one string
"string\nwith newline"
// second string
"string li" "terals can be"
  "split, and separated with whitespace."
// identifiers
ident_1
___ident123_4
// characters:
'x' '\n'
```

Code is not passed through any preprocessor by default.

Following strings cannot be used as identifiers, because they are keywords:

```
_ as bool break case continue def do else exception extern false finally
float for fun if in int let null open opt_struct raise return section
skip string struct switch true try type typedef union void while with
```

4 Control structures

Basic control structures include `while`, `do ... while` and `for` loops, `if` conditionals, and `break` and `continue` jump statements.

Gont control structures are in few places more restricted then these found in C. This is to satisfy safety paradigm. You might find this annoying, upon first reading.

4.1 Empty instruction

There is no empty instruction (written as ';' in C). `skip` keyword is introduced instead. For example:

C:

```
while (f(b++));
```

Gont:

```
while (f(b++))
        skip;
```

4.2 Conditions

Type of condition in control structures has to be bool. Therefore, C code:

```
int i = 10;
while (i--)
        f();
```

needs to be written as:

```
int i = 10;
while (i-- != 0)
        f();
```

4.3 Dangling else

In C `else` is associated with nearest else-free `if`. The grammar of C is said to be ambiguous because of this. This can be real problem sometimes, following code, due the indentation used, is misleading:

```
if (foo)
        if (bar)
                baz();
else
        qux();
```

In Gont one needs to use { } when putting one `if` into another.

4.4 Labeled loops

In Gont, similarly to Ada or Perl, loops can be given names, in order to later on tell `break` and `continue` which exactly loop to break. This looks as:

```
foo: while (cond) {
        bar: for (;;) {
                if (c1)
                        break foo;      // break outer loop
                else
                        continue bar;   // continue inner loop
                for (;;) {
                        if (c2)
                                break;  // break enclosing loop
                }
        }
}
```

Part II

Types

5 Basic Types

Gont includes following basic types:

- `int`

  signed integers, 32 or 64 bit wide, depending on architecture

- `float`

  IEEE double precision (64 bit) floating point numbers

- `string`

  character strings

- `bool`

  booleans, `true` and `false` are the only possible values

- `void`

  empty type, or more accurately, type of which just one value lives. While it cannot be spelled in current version of Gont, it is implicitly created with `return;` statement.

5.1 Pointers

There is no explicit notation of pointers. Structures and unions are always treated as pointers (as in Java). Strings are built in type. Pointers to functions are discussed in Section 8 below.

6 Structures

They are defined with `struct` and `opt_struct` keywords.

```
struct s {
        int a;
        string b;
}
```

is roughly equivalent of C's:

```
typedef struct {
        int a;
        char *b;
} *s;
```

So, name `s` can be used (only) without any `struct` or `opt_struct`. For example, this piece of code uses structure definition above:

```
string f(s foo)
{
        foo.a = 5;
        return foo.b;
}
```

You should note, that structures are passed by pointer (or by reference if you prefer C++ naming style), therefore changes made to struct inside function are reflected in state of it on the caller side, hence:

```
s x;
...
x.a = 10;
f(x);
// x.a is 5 here
```

Fields of structure can be accessed with ‘.’ operator, there is no ‘-¿’ operator.

6.1 What’s opt_struct?

`struct` value is always valid, i.e. it has to be initialized with object instance, before it is used, and you cannot assign `null` pointer to it.

`opt_struct` can be `null`. You still have to initialize it before, it is used, but you can do it with `null` keyword. You can also assign `null` to it later on.

This involves runtime check on each access to `opt_struct` value. When you try to access `opt_struct` value, that is `null`, `Null_access` exception is raised. This behavior can be controlled with compiler switch.

6.2 Assignment to structures

If you do:

```
void f(s1 x)
{
        s1 y;
        y = x;
        y.fld = 0;  // here you also modify x.fld
}
```

In general assigning values other then `int`’s, `bool`’s and `float`’s copies pointer, not content, i.e. makes an alias of an object.

6.3 Structure initializers

When initializing structures one has to spell field name along with expression initializing it. For example:

```
struct foo {
        int bar;
        string baz;
}

void f()
{
        foo qux = { bar = 1, baz = "quxx" };
}
```

{ ... } is also normal expression, for instance:

```
void f()
{
        foo({ bar = 1, baz = "qux" });
}
```

7 Polimorphism

This is probably the neatest thing in Gont. Structures, as well as functions can be parameterized over types. Thus it is possible to write generic types and functions, like list of anything, or stack of anything.

For example to define list of anything you write:

```
opt_struct <'a>list {
        'a data;
        <'a>list next;
}
```

`'a` is alpha. It is type variable, it stands for any type (similarly `'b` is beta, but I really don't know what `'foo` is... :-) `<'a>list` is read as "list of alphas" – list of data of any type. `<int>list` is "list of integers".

Then you use it as:

```
<int>list l;            // list of ints
<<int>list>list ll;     // list of lists of ints
<string>list s;         // list of strings
```

Now, when we have our polimorphic list, we note, that for example `length()` function doesn't need to know type of elements stored in the list:

```
int length(<'a>list l)
{
        int n;

        for (n = 0; l != null; l = l.next)
                n++;

        return n;
}
```

Gont standard library includes modules with ready-to-use generic datatypes.

7.1 Polimorphism vs templates

At the abstract level polimorphism is similar to templates in C++. However, in Gont, code for given polimorphic function is generated just once. Additionally polimorphism becomes really funny in conjunction with functional values. We'll discuss it later.

You might have seen generic datatypes implementations in C. They most often use `void*` pointers as placeholders for user data. Gont does essentially the same. However it ensures, this `void*` is always assigned and read from variables of the same type.

If you are familiar with C++ you know, that following C++ template instance is not valid:

```
List<List<int>> l;
```

whereas in Gont, following is valid:

```
<<int>list>list l;
```

'¡¡' and '¿¿' are lexed as bitwise shift operators in both Gont and C++. In C++ one have to write '¡ ¡', but in Gont parser handles this case specially, just for convenience.

8 Functional values

Functions are first class citizens. This functional programming slogan means that you can write a function, that takes function as argument and returns function. This is called higher order function. As this might sound a bit magic, let's make it look like real voodoo: :)

8.1 Example

```
opt_struct <'a>list {
        'a data;
        <'a>list next;
}

*(void (<'a>list)) mapper(*(void ('a)) f)
{
        void for_all(<'a>list lst)
        {
                while (lst != null) {
                        f(lst.data);
                        lst = lst.next;
                }
        }

        return for_all;
}
```

This function takes function `f` operating on items, and returns function that applies `f` to all elements of list.

Ok, now some explanations. The key thing to note is functional type notation. `*(void ('a))` is function that takes single argument, of any type, and returns no value. Similarly `*(string (int, <'a>list))` is function, that takes `int` and `<'a>list` parameters, and returns `string`.

8.2 That's weird, I can't get it!

First step is to use higher order functions, next is to write them. Gont standard library includes several modules involving higher order functions. One such example is `List::iter()` function. It is defined as:

```
void List::iter(*(void ('a)) f, <'a>List::t lst);
```

Let's say you have list of integers and want to print it out:

```
<int>List::t some_list;
// ... here it is assigned some value ...

void print_element(int el)
{
        print_int(el);
        print_newline();
}

List::iter(print_element, some_list);
```

`List::iter()` applies given function to each element of list.

8.3 More examples

```
// Call passed function f on each element of the list l
// starting from head.
void iter(*(void ('a)) f, <'a>list l)
{
        while (l != null) {
                f(l);
                l = l.next;
        }
}

// Call function f on each element of the list l,
// collect results as a list (of possibly different type)
// and return it.
<'b>list map(*('b ('a)) f, <'a>list l)
{
        <'b>list o = null;

        while (l != null) {
                o = {data = f(l.data), next = o};
                l = l.next;
        }

        return o;
}
```

Following functions are defined in Gont standard library:

```
string itoa(int);
void print_string(string);
```

We can use them with our `iter` and `map`:

```
<int>list il = { data = 1, next = { data = 2, next = null }};
<string>list sl = map(itoa, il);
iter(print_string, sl);
```

This will print "12".

8.4 MLish variations about defining functions

Nested function definitions (for example the `for_all` function) are shortcut to defining functional variables and initializing them, so our example could look as:

```
*(void (<'a>list)) mapper(*(void (int)) f)
{
        *(void (<'a>list)) for_all;

        for_all = fun void (<'a>list lst) {
                while (lst != null) {
                        f(lst.data);
                        lst = lst.next;
                }
        };

        return for_all;
}
```

Or even:

```
*(void (<'a>list)) mapper(*(void (int)) f)
{
        return fun void (<'a>list lst) {
                while (lst != null) {
                        f(lst.data);
                        lst = lst.next;
                }
        };
}
```

`fun` is keyword. After `fun` comes return type, then list of parameters, along with their types, just like in regular function definition in C. Then comes body of function – either block or expression in parenthesis.

8.5 Omitting return keyword

Special extension is supported, useful when dealing with functional values. Whenever function body (sequence of statements within `{ }`) should appear, single expression within `( )` can be supplied. It is passed to return statement, which is only statement in function body. So:

```
fun int (int a, int b) (a + b)
```

is equivalent of:

```
fun int (int a, int b) { return a + b; }
```

8.6 Closures

This section is for curious folks, who always want to know, how things work.

Local functions needs to remember variables from enclosing function, at the time they were defined. For example `for_all` function from our example needs `f` parameter of `mapper`. However `for_all` cannot rely on being called when `mapper` activation record is still on the stack.

Therefore local function are stored as pair of pointer to function and pointer to special closure structure. Closure of a function holds all variables defined in it, that might need to be accessed by local functions. Local functions, in generated code, are always passed closure to enclosing function as first argument.

9 Tuples

Tuple is datatype, that consists of two or more components, which are anonymous. Tuples are defined as:

```
*(int, int) t1;                 // pair of integers
*(int, string, bool) t2;
*(ctx, ty) t3;
```

And used as:

```
t1 = (1, 2);
int a, b;
(a, b) = t;
// swap
(a, b) = (b, a);
t3 = (1, "hello", true);
// true and false are keywords
return (1, "one");
```

They might come in handy, when you need to return more then one value from function. More on tuples in Section 13 below.

10 Unions

Union are more closely related to ML's datatypes then to C's unions. The basic difference is that Gont compiler remembers which member is currently stored in union.

Unions are defined as:

```
union exp {
        int Const;
        void Var;
        *(exp, exp) Add;
        *(exp, exp) Sub;
        *(exp, exp) Mul;
        *(exp, exp) Div;
}
```

This union can be later on used for processing symbolic expressions. For example:

```
// f = (x / 10) + x
exp f = Add(Div(Var, Const(10)), Var);
exp g = Var;            // both forms
exp h = Var();          // are correct
```

You can access union components only using pattern matching (there is no '.' notation). Pattern matching is discussed in Section 14.

11 Type inference

When writing programs in Gont, you will soon note, that type expressions can get quite large. To save your keyboard Gont compiler can guess types of variables for you. You can instruct it to do so, by using underscore (`_`) instead of type. Example:

```
_ s = "Hello world";    // string s = "Hello world";
_ make_tuple = fun _ (_ x, _ y) ((x, y));
_ make_tuple2 = fun (x, y) ((x, y));
```

`make_tuple2` definition is identical to `make_tuple`. If all types in lambda expression are `_`, then you can omit them all.

11.1 It's guessing? Is it safe?

Well... guessing isn't best description of type inference. It simply knows. If there are any problems, it will report them. For example in: `fun (x, y) (x + y)` type of `x` and `y` can be `float`, `int` or `string`. Therefore Gont compiler will reject it. However it won't complain about `fun _ (int x, _ y) (x + y)`.

11.2 Caveats

Current algorithm is somewhat limited, it will lose given complex enough example.

Part III

Pattern matching

12 Pattern matching basic types

Pattern matching is technique used to decomposite complex datatypes and made decisions based on their content.

In Gont one might pattern-match ints, strings, bools, tuples and unions. Pattern matching ints and bools looks like `switch` in plain C:

```
string my_itoa(int i)
{
        switch (i) {
        case 1:   return "one";
        case 2:   return "two";
        case _:   return "infinity";    // it's set rather low...
        }
}
```

`case _:` is Gont way to say `default:`. It is called match-all pattern.

As a consequence of `string` being built in, basic type, strings can also be pattern matched:

```
int my_atoi(string s)
{
        // note lack of () around s, they can be omitted
        switch s {
        case "one": return 1;
        case "two": return 2;
        case _:     return 3;
        }
}
```

As you probably guessed, `bool`'s are pattern matched with `true` and `false`.

13 Pattern matching tuples

Patterns can include variables. `case` clause assigns values found in given place in pattern to them. For example, following function return first element of tuple passed to it:

```
int first(*(int, int) t)
{
        switch t {
        case (x, y): return x;
        }
}
```

We can mix variables and constants (often called **type constructors** in this context) together:

```
int special_sym(int a, int b)
{
        switch (a, b) {
        case (_, -1):
                return -1;
        case (-1, x):
                return x;
        case (x, y):
                return x + y;
        }
}
```

As said before, “_” is match-all pattern. It matches any value. One might note, that `x` also matches any value. However, it is matter of good style, to use `_` when you are not going to use value matched.

Important thing to note about the example above, is that we construct tuple, just to pattern-match it. The same result could be achieved with few `if`’s, but pattern matching is often more readable (at least when one get used to it :-)

Variables inside patterns are not l-values, therefore tuples are immutable, i.e. one cannot modify them after they have been constructed.

14 Pattern matching unions

Similarly as with tuples, pattern matching is the only way to get to the value of union. Union patterns consists of field name followed by nothing, `()`, pattern enclosed in `( )` or tuple pattern. This mimics ways, one can construct union value.

Following example utilizes pattern matching in simple symbolic expression computations environment.

```
union exp {
        int Const;
        void Var;
        *(exp, exp) Add;
        *(exp, exp) Sub;
        *(exp, exp) Mul;
        *(exp, exp) Div;
}

int compute(exp e, int v)
{
        switch e {
        case Const(x)    : return x;
        case Var         : return v;
        case Add(e1, e2) : return compute(e1, v) + compute(e2, v);
        case Mul(e1, e2) : return compute(e1, v) * compute(e2, v);
        case Div(e1, e2) : return compute(e1, v) / compute(e2, v);
        case Sub(e1, e2) : return compute(e1, v) - compute(e2, v);
        }
}

exp diff(exp e)
{
        switch e {
        case Const(_)    : return Const(0);
        // both Var and Var() are legal
        case Var()       : return Const(1);
        case Add(e1, e2) : return Add(diff(e1), diff(e2));
        case Sub(e1, e2) : return Sub(diff(e1), diff(e2));
        case Div(e1, e2) :
                exp up = Sub(Mul(diff(e1), e2), Mul(e1, diff(e2)));
                exp down = Mul(e2, e2);
                return Div(up, down);
        case Mul(e1, e2) :
                return Add(Mul(diff(e1), e2), Mul(e1, diff(e2)));
        }
}
```

14.1 What happen to C's switch and enums?!?

Hmm... they are still there:

```
union color {
        void Red;
        void Green;
        void Blue;
}
```

This is roughly equivalent of C's:

```
typedef enum {
        Red,
        Green,
        Blue
} color
```

Then we do:

```
string color_name(color c)
{
        string r;

        switch (c) {
        case Red:
                r = "red";
        case Green:
                r = "green";
        // Blue() is also legal
        case Blue:
                r = "blue";
        }

        return r;
}
```

This looks almost like C, modulo one little issue. **There is no fall through.** Each `case` is separate branch or program execution. One doesn't need to write `break` statements. In fact, `break` would break loop enclosing `switch`, if there is any!

14.2 Fall through disclaimer

If you don't like the no-fall-through clause, please first consider following snippet of code:

```
switch e {
case Mult(e1, e2):
        // here we do fall through
case Const(x):
        // hmm... what does x mean, if we fall through from Mult(...)?
        // and what can e1 and e2 mean here, if we didn't fall
        // through?
}
```

Because `case` can bind new variables, it has to introduce new scope. So fall through is impossible.

14.3 But I want fall through!

Despite disclaimer above limited form of fall-through is available, if there are no statements after `case ...:`. Only variables bound by all `case` clauses are available in clause body. Also each occurrence of given variable needs to have the same type, thus:

```
union foo {
        int Int;
        int Int2;
        int Int3;
        string String;
        bool Bool;
}

foo x;

// this is legal
switch x {
case Bool(i):     skip;         // don't fall through
case Int(i):
case Int2(i):
case Int3(i):     print_int(i);
case String(s):   print_string(s);
case _:           skip;
}
```

```
// this is not legal
switch x {
case Bool(i):
case Int(i):      skip;  // i cannot be int and bool at the same time
case _:           skip;
}

// nor this:
switch x {
case Bool(_):
case Int(i):
case Int3(i):     print_int(i); // unbound variable i
case _:           skip;
}

// but this *is* legal
switch x {
case Bool(_):
case Int(i):
case Int3(i):     skip;
case _:           skip;
}
```

15 Case guards

`case` allows additional condition to be checked if pattern matches. This is called case guard. Example:

```
switch x {
case Int(i) if (i > 0): print_int(i)
case Int(i):            print_int(-i)
case _:                 skip;
}
```

is the same as:

```
switch x {
case Int(i): if (i > 0) print_int(i)
             else print_int(-i);
case _:      skip;
}
```

Although it might not seem very bright at the first glance it comes in handy in more complex cases, like:

```
switch x {
// special case negative Ints
case Int(i) if (i < 0):  print_string("small")
// don't special case negative Int2s and Int3s, also treat
// non-negative Ints ordinarily
case Int(i):
case Int2(i):
case Int3(i):             print_int(-i)
// ignore the rest
case _:                   skip;
}
```

15.1 Name binding and case guards

All variables bound by pattern are available in scope of case guard, for example:

```
        switch x {
        case Int(i) if i > 0:   // one can use i in condition
        case Bool(_):
                // but using i here would be error
                print_string("positive or bool");
        case _:
                skip;
        }
```

However beware cases like:

```
        switch x {
        case Int(y) if y > 0:
        case String(y) if String::length(y) == 3:
                // Compiler thinks y should be bound here, but this code
                // fails to typecheck.
                skip;
        case _:
                skip;
        }
```

which should be:

```
        switch x {
        case Int(i) if i > 0:
        case String(s) if String::length(s) == 3:
                // all ok
                skip;
        case _:
                skip;
        }
```

15.2 Case guards and pattern exhaustiveness

Compiler assumes each case guard can fail for purposes of exhaustiveness warnings, for example it will complain about:

```
        int i;
        // ...
        switch i {
        case x if x > 0:        return 1;
        case x if x <= 0:       return 2;
        }
```

that it is not exhaustive, although it is. The correct way to write this is:

```
int i;
// ...
switch i {
case x if x > 0:        return 1;
case x:                 return 2;
}
```

(although `if` would be surly more convenient in this case :-).

16 Pattern matching structures

It has been just implemented, so probably it's buggy ;)

```
switch e {
case {next = {next = null}, data = a}:
        return a;
case {next = null, data = a}:
        return a + 1;
case _:
        return 0;
}
```

17 Exhaustive matching

In previous examples we have checked for each possibility, in our `switch` statements, but we don't have to:

```
string var_name1(exp e)
{
        switch e {
        case Var(x): return x;
        // matches any x
        case x: return "not variable";
        }
}

string var_name2(exp e)
{
        switch e {
        case Var(x): return x;
        }
}
```

`var_name2` would raise `Match_failure` exception if any pattern didn't match.

However it is matter of good style, to always check each possibility. Compiler will warn you, if you fail to do it. This warning comes in very handy in some situations. For example, suppose you have union similar to `exp` (from example above) in your symbolic algebra package. Now, you want to add power operator. So you add:

```
*(exp, int) Pow;
```

to `exp` definition. Now you have to change each pattern matching of `exp` to check for power function. **gontc** will help you, complaining about non-exhaustive pattern matching. I, personally, found this feature very useful during Gont compiler development. Whenever I want to add new feature to Gont, I add it to syntax tree definition and fix compiler, until there are no warnings about non-exhaustive pattern matching.

18 let statement

`let` statement allows limited form of pattern matching, with terser syntax, for example patterns can be used to decomposite tuples, like this:

```
*(int, int) t = (1, 2);

...
switch t {
case (i1, i2): return i1 + i2;
}
```

This can be abbreviated to:

```
*(int, int) t = (1, 2);

...
let (i1, i2) = t in return i1 + i2;
```

There can be more then one assignment, like this:

```
let (t1, t2) = t,
    (s1, s2) = s in {
        // ...
}
```

The let assignment and binding names with case just creates new name for an object.

There is also second form of, that does not introduce new scope, but binds names until end of current scope. It's called "flying let", and is written as:

```
let (t1, t1) = t;
let x = foo();

bar(t1, x);
baz(t1, x);
```

Part IV

Other stuff

19 Module system

Modules provide way of separate compilation along with possibility of avoiding namespace conflicts. Current module system is based on ideas from OCaml. However, compared to OCaml, it is very limited, we only support one level of modules, there are no functors and so on.

19.1 Interface

Interface part of module `Foo` goes to file *`foo.gi`*. This is similar to .h header in C. It contains function prototypes, types definitions, variables declarations and similar stuff.

The main purpose of interface file is to hide implementation details.

Hiding information about defined functions and variables is easy – you simply do not provide prototype or declaration, and no one, outside *`foo.g`*, will be able to call this function, or reference variable.

You can also hide information about types, by writing `type bar` in foo.gi, you say: "I will define type named `bar` in *`foo.g`*, but users of this module doesn't need to know what exactly this type is". Of course if there is no need to even say, that `bar` is defined, you can omit `type bar`, and just define it in *`foo.g`*.

It is also possible to disclosure types in interface. It is done exactly the same way as in implementation file.

Now some example:

```
// interface of Foo

// reset module to initial state
void reset();

// don't disclosure actual type of file
type file;

// but provide definition for type used as interface
union open_flag {
        void Open_rdonly;
        void Open_rdwr;
        void Open_wronly;
        int Open_to_append;     // int argument specifies at which point
                                // to start appending (I know it's
                                // stupid :-)
}

file open(open_flag flag);

// global variable
int open_files_cnt;
```

19.2 Implementation

Implementation part of module `Foo` goes to file *`foo.g`*. Actual function, variable and type definitions are here.

All symbols defined in `Foo` are prefixed with `Foo::`. This allows having function (or type, or variable) named `foo` in both `Bar` and `Baz` modules.

Now we might look at some more realistic example:

File mylist.gi:

```
        // this is to output information,
        // that we implement type `t', but not to
        // disclosure what it is.
        type <'a>t;
```

```
// return first element (head) of the list
'a hd(<'a>t x);

// return all but first element (tail) of the list
<'a>t tl(<'a>t x);

// create new empty list
<'a>t create();

// apply f to all elements of l, return list of results
<'b>t map(*('b ('a)) f, <'a>t l);

// call f on all elements of the list
void iter(*(void ('a)) f, <'a>t l);
```

File mylist.g:

```
// this is local datatype.
opt_struct <'a>t {
        'a data;
        <'a>opt_struct next;
}

// this will be exported out (`public')
'a hd(<'a>t x) { return x.data; }
<'a>t tl(<'a>t x) { return x.next; }

// this is local, can't be called from outside the module
void helper(<'a>t x) { ... }

// and more publics
<'a>t create() { return null; }
// we already wrote these
<'b>t map(*('b ('a)) f, <'a>t l) { ... }
void iter(*(void ('a)) f, <'a>t l) { ... }
// ...
```

Then if you want to use the module, it can be done with `::` notation, like this:

```
<int>Mylist::t l = Mylist::create();
...
int k = Mylist::hd(l);
```

In case of some modules it might be useful to open them, i.e. import all symbols from module into current namespace, so you no longer have to use `::` notation (but you still can, it is often suggested for readability):

```
open Mylist;
...
<int>Mylist::t l = Mylist::create();
...
int k = hd(l);
// it might be hard to tell what <int>t refers to here,
// so using Mylist::t is recommended
<int>t rest = tl(l);
```

19.3 Hey, where did #include go?

In Gont, at present, there is no `#include`. Also you cannot prototype functions from other modules in implementation files. Modules are more abstract, higher level mechanism, that should be used instead. It is much harder to make a mistake when using modules, and it's not all that hard, once you used to them.

`#include` (and `#define` for that matter) might be implemented in some unspecified future, if there are some good reasons to. However `#include` won't be used for modularization.

19.4 Core module

`Core` module is open at the beginning of each compilation unit. It includes few primitive functions, exceptions and types.

20 Initialization and finalization of modules

You often need to initialize module before it is used. For example to global variables often needs some initial value that is not constant and so on. Gont uses `section` declaration to deal with such cases. `section` is followed by name of section. You can use `section` several times. Bodies of `section` under the same name in one module are concatenated, in order, in which they are given in source code.

Currently just two sections are provided:

- `init`

  Code in `init` section is executed at the beginning of the program.

  Order in which `init` sections of modules are executed is specified in command line during link. However Gont linker ensures you do not use any module, that has `init` or `fini` section before it is initialized.

- `fini`

  Code in `fini` section is executed just before leaving the program.

  Order of execution of `fini` sections is order of `init` reversed.

The `Core::at_exit` function registers given function to be run before program exits. However content of all `fini` sections are executed after all functions registered with `Core::at_exit` has completed.

There is no `main` function in Gont program. One uses `section init` for this purpose.

Following example:

```
section init { print_string("1 "); }
section fini { print_string("-1 "); }
section fini { print_string("-2\n"); }
void bye()
{
        print_string("bye ");
}
void main()
{
        at_exit(bye);
        print_string("main ");
}
section init { print_string("2 "); main(); }
```

prints: `1 2 main bye -1 -2`.

20.1 Greedy linking

You may tell **gontc** to link given library in greedy mode. It means that all files from this library are linked, and their `init` and `fini` sections are executed.

You don't want to do this with regular libraries, like Gont standard library, because you do not need all files from them. **gontc** is wise enough to tell that you need, let's say `List` module, if you used it in your program.

However greedy linking might come in handy, modules are not explicitly referenced from main program, but their `init` sections registers them somewhat with the main program.

20.2 Mutually recursive modules

Unlike in Caml, it is possible to have two modules that reference each other. However none of them can have `init` nor `fini` section. If they have, they cannot be mutually recursive (reason: what should be the order of initialization?)

21 Exceptions

Exceptions are sophisticated form of jump instruction. They are used to inform about unusual situation in a program, in order to transfer control to block, that can handle it. They can be thought of as member of a huge union:

```
union exn {
        // system defined
        void Core::Null_access;
        void Core::Match_failure;
        void Core::Not_found;
        void Core::End_of_file;
        string Core::Invalid_argument;
        string Core::Failure;
        void Core::Exit;

        // user defined, Compiler is name of user module
        string Compiler::Syntax_error;
        void Compiler::ICE;
}
```

New members are added with exception keyword, for example one can say in *`compiler.g`* or *`compiler.gi`*:

```
exception string Syntax_error;
exception void ICE;
```

In order to signal unusual situation, you use raise keyword:

```
raise Syntax_error("parse error");
raise ICE;
raise Compiler::ICE();
```

At the place, where you know how to handle it, you use try:

```
try {
        open_file();
        ...
        parse();
        ...
} with {
case Syntax_error(s):
        print_msg(s);
case ICE:
        print_msg("Internal Compiler Error");
} finally {
        close_file();
}
```

Statements in `try { ... }` are executed. If exception occurs during this process control is transfered to `with { ... }` part. Exception is then pattern matched against `case` clauses there. If it didn't match anything, it is raised again (to be possibly caught by another enclosing `try` block).

No matter how control leaves `try` block, statements in `finally { ... }` are executed upon exit. This can be used to release some allocated resources (for example to close a file, like in our example).

One can omit `with { ... }` part and use only `try { ... } finally { ... }`, or omit `finally { ... }` and use only `try { ... } with { ... }`.

22 Function prototypes

You cannot prototype functions from other modules (module interfaces are used for this). Functions from current modules need not be prototyped, **gontc** parses entire file and records all function definitions before proceeding with compilation.

Part V

Appendixes

23 Intro

This part gives some more formal (but still far from completely formal, working on it :) descriptions of some aspects of Gont language, and documents some other stuff that didn't fit anywhere :)

24 Type system

t, t1, t2, ... are type expressions. v1, v2, ... are type variables ('a, 'b, 'foo, ...).

24.1 Basic types

`int`, `float`, `string`, `bool`, `void`, written as is.

24.2 Function type

Function type is written as follows:

```
*(t (t1, t2, ..., tn))
```

where n ¿= 0. One might note that this is somewhat different from ML where all functions take just one parameter (which can be tuple or another function, but this is not the point). This is closer to C's function notation.

24.3 Tuples

```
*(t1, t2, ..., tn)
```

where n ¿= 2.

24.4 Structures

```
struct <v1, v2, ..., vn> NAME {
        t1 f1;
        t2 f2;
        ...
        tm fm;
}
```

Structures that can be invalid (i.e. null):

```
opt_struct <v1, v2, ..., vn> NAME {
        t1 f1;
        t2 f2;
        ...
        tm fm;
}
```

24.5 Unions (datatypes)

```
union <v1, v2, ..., vn> NAME {
        t1 f1;
        t2 f2;
        ...
        tm fm;
}
```

where n ¿= 0, m ¿= 1. If n == 0, ¡¿ can be omitted. fi are field names.

24.6 Named types

Structures and unions can be later on referred with:

```
<t1, t2, ..., tn> NAME
```

where n ¿= 0. If n == 0, ¡¿ can be omitted.

24.7 Foreword

* in front of tuple and function types is (crude) way to convince Bison, that there are none reduce/reduce conflicts in grammar. I guess there aren't even without '*', but tell it to yacc... (this problem is referred to in Bison manual, there are some workarounds, however I was unable to make use of it).

25 Wish list

There should be way to omit types, and have them reconstructed.

I would like to have some support for named arguments, so functions can be called as:

```
Window::create(width => 20, height => 23, color => red);
```

This should come with default values.