lambda-flow - draft - version 0.3.1

Guilhem de Wailly

Institut

Preface

This document presents the **lambda-flow** language and its compiler. This is a functional synchronous data-flow language.

lambda-flow can model every application that can be described with a graph and that does not contain any fix-point definition. A graphical interface is proposed and it is currently rewriting with the Motif tool-kit.

The advantage of **lambda-flow** is its simplicity for the final user and its adaptability. **lambda-flow** does not contain complicated construction, which are difficult to learn, and with a confuse semantics (or a semantics that is difficult to understand).

The language has the minimal set of objects to model graph based application in a modular way. A module can be separately compiled. **lambda-flow** encourages polymorphic module definition in order to have a maximal code reuse.

It is a strongly typed language, but the type checking is as transparent as possible. It is not necessary to declare everywhere the type of the handled data: the compiler uses a powerful method to deduce these types from the data.

The language is extremely adaptable to any purpose because the handled data and their operators are not specified in the language. They form algebra that are text files dynamically loaded into the compiler. The only mandatory algebra is the integer algebra. Of course, the user can easily write an algebra and use mixed algebra in the same program.

In an other hand, **lambda-flow** is adaptable because the target code is not statically specified into the compiler: the target code definition is a text file dynamically loaded. The standard distribution issues three target code definitions for C, Scheme and i386 assembler. The user can easily add a target code.

lambda-flow was originated to allow signal processing application to be implemented onto parallel architecture. The underlined architecture is static because the language guarantees that all the memory accesses are known at compile-time. This is possible because **lambda-flow**programs are deterministic in time and in resources.

The compiler is available for major Unix systems and for DOS.

The main feature of the environment are:

- functional synchronous data-flow language, formal proofs easier,
- closed to the Z-formalism,
- data algebra independent, target code independent,
- implicit type-checking, no mandatory type declaration,
- modularity and separate compilation,
- polymorphic or typed abstraction,
- time and resource determinisms,
- low-level code optimization,
- parallelism easily accessible, cheap static parallel architecture proposed,
- could be a specialized chip (ASIC) specification language.

This document could be found in an HTML web page, where the curriculum vitae of the author and the related publications could also be obtained at the address:

`http://alto.unice.fr/ gdw`

Contents

Chapter 1. Introduction

lambda-flow is a general purpose **functional synchronous data-flow language**. It is general because the language does not specify the handled data and their operator (4). These definitions are **dynamically loaded** into the compiler, at compile-time (7). In addition, the target code is also defined dynamically at compile-time (5).

The language is **fully functional**. The word fully is important: the language is of course functional in the traditional meaning, where an expression has not side effect. But in addition, the whole solving process is described in a functional form. This strong feature emphasizes the semantics definition of the language. Making proofs is easier.

The data-flow property is the **functional translation of the state variables** in a program. **lambda-flow** belongs to the Lucid family. It was been shown that iterations in a program can be translated by stream definitions, with the advantage of the functional property. But **lambda-flow** is wanted to be deterministic in time and in resource. Some constraints have been added to the language, which provides the synchronous feature.

Due to the **determinisms** of the programs, their could be implemented onto **static parallel architecture**. All the memory accesses are known at compile time, so they can be resolved at this moment.

1.1. Existing tools

A dataflow program is a diagram with lines as data paths and boxes as operations. It exists two methods to run a dataflow program.

The first method to run a dataflow programs is the **data-driven** method. When a data is available on each input of an operator, the operator computes a new data that it puts on its output. The researches on dataflow parallel computers started with **Miller** and **Karp** in 1966. But this kind of dataflow suffers to the lack of a global semantic description of the program and the inefficiency of the implementations.

The second method to run a dataflow program is the **demand-driven** method. Here, a result is asked to an operator. The operator propagates the demand to its empty inputs. When all the inputs data are available, the operator computes a data and it returns it. This kind of dataflow is closed to the functional programming style.

Functional languages have all a valuable property: they are built on the mathematically based lambda-calculus. Functional programming languages can be efficiently implemented onto a classical **Von Neumann** architecture, which provides low cost specialized DSP processors and well known programming environments.

The first functional dataflow language is Lucid. It is the first to demonstrate that a dataflow programming style can replace iteration, with the advantage of the functional property. But **Lucid** contains several features not well adapted to DSP. Particularly, it is not **time**memory deterministic/.

The **Sisal** is introduced to implement general purpose FSD program onto parallel architecture. But it is not adapted for DSP for the same reasons than **Lucid**.

Lustre and **Signal** are two FSD languages well adapted for DSP. Their kernel is based on recurrent clocked equations. **Signal** does not define explicitly a root clock while the clocks in **Lustre** are all defined on a base clock.

Our language **lambda-flow** is a part of a CAD tool chain for implementing DSP application onto parallel architectures. It is more primitive than the languages cited above. It has less expressions and less concepts: it defines only one temporal operator, used to built a stream of values. The streams are updated in a synchronous way, so, the language could be used for **real-time** applications.

It provides the **alternative** construction (if-then-else). Associated to the stream object, the alternatives could be used to define some clocks. So, the clock concept is not explicitly defined in the core language.

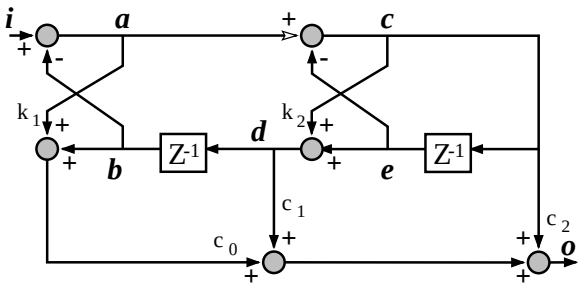
The integer **algebra** is predefined by default. All the other handled data are dynamically bound to an algebra. This feature gives to the language a great generality and adaptability: it is defined as a “thing to handle some things”, without specifying the nature of the handled things, such as the **Landin**’s language.

lambda-flow is a **typed language**, but its type checking has less constrains than the languages cited above: it encourages **polymorphic** abstractions. In addition, it supports a full lexical scope binding. **lambda-flow** has a

graphical interface where the program can be drawn as a dataflow graph.

1.2. Example

In this section, we use the **lambda-flow** language to specify a digital filter. The diagram of the filter is given in the following figure.



This is a recursive second order filter. It is very easy to program this filter with **lambda-flow**: give a name to each node, replace the *Z* operator by a stream, and you get the program:

```
filter := lambda i. begin
  o  ! a + b + c + d;
  a := i - b;
  b := 0 followed-by d;
  c := a - e;
  d := c + e;
  e := 0 followed-by c;
end;
```

Notice the great similarity between this specification and the Z-equations[1].

In order to simulate a multiple files program, let us define a `main` module which instantiates `filter`. We obtain:

```
main := lambda i:int. begin
 out  ! filter(i) extract o;
end;
```

As in the language C, a program must have one and only one `main` module. Its inputs must be typed. In addition, the outputs of the whole program is the outputs of the `main` module.

Now, this program has to be compiled with the **lambda-flow**compiler. **lambda-flow** accepts several options in the command line (7.1), and it has also a configuration file (7.2). Some options are loaded by default. So, the compiler invocation is simply:

```
$ flow filter.lf main.lf
```

This command produce an executable file `a.out`. If you want to keep the C auxiliary file, simply type:

```
$ flow -k filter.lf main.lf
```

that produces in addition the following `a.c` file:

```
#include <stdio.h>

int getint(int port) {
  int tmp;
  if (scanf ("%d", &tmp) == EOF) exit(0);
  return tmp;
}
#define putint(port,value) printf("%d ", value)
```

[1]The Z-formalism is a mathematics tool used to study the DSP filter.

```
static int d;
static int e;
static int i;
static int out_1;
static int a;
static int b;
static int c;

main() {
  start:

  init:
    i = getint (1);
    b = 0;
    e = 0;

  loop:
    a = (i) - (b);
    c = (a) - (e);
    d = (c) + (e);

    out_1 = putint (1, (a) + ((b) + ((c) + (d))));

  next:
    i = getint (1);
    b = d;
    e = c;

    goto loop;
}
```

The **`lambda-flow`** compiler could also produce Scheme code. Simply type:

```
$ flow -t scheme -k filter.lf main.lf
```

That produces a `a.scm` output file:

```
(define (getint port) (read))
(define (putint port value) (display value))
(define (bool->int bool) (if bool 1 0))
(define (int->bool  int) (if (zero? int) #f #t))


(let* (
      )
(let loop (
  (i (getint 1))
  (b 0)
  (e 0)
         )
(let* (
  (a (- i b))
  (c (- a e))
  (d (+ c e))
  (out_1 (putint 1 (+ a (+ b (+ c d)))))
       )
(loop
  (i (getint 1))
  (b d)
```

```
  (e c)
))))
```

Notice that the Scheme code does not contain any `set!`.

The last command produce Intel 386 code:

```
$ flow -t i386 -k filter.lf main.lf
```

That produces a `a.s` output file:

```
.data
 d       .word
 e       .word
 i       .word
 out_1_1 .word
 out_1_2 .word
 out_1_3 .word
 out_1   .word
 a       .word
 b       .word
 c       .word

.text
main:

init:
 in 1
 mov.w i, ax
 mov.w b, 0
 mov.w e, 0

loop:
 sub.w i, b
 mov.w a, ax
 sub.w a, e
 mov.w c, ax
 add.w c, e
 mov.w d, ax
 add.w c, d
 mov.w out_1_1, ax
 add.w b, out_1_1
 mov.w out_1_2, ax
 add.w a, out_1_2
 mov.w out_1_3, ax
 mov.w ax, out_1_3
 out 1
 mov.w out_1, ax

next:
 in 1
 mov.w i, ax
 mov.w b, d
 mov.w e, c

 jmp loop

end:
```

The power of **`lambda-flow`** is demonstrated with the code production os three very differents languages: C as

imperative language, Scheme as functional language and i386 assembler.

In fact, the target code is defined in a target code definition file that is a text file dynamically loaded into the compiler. The user could extent the standard target definition file very easily.

Chapter 2. Elements of the language

2.1. Comments

lambda-flow uses the C++ comment forms. A comment start from // to the end of the line. A comment could also start from /* to */, which allows multi-lines comments.

2.2. Identifiers

All the string of characters without blanks and special characters are identifiers. A blank is one or more mixed spaces, tab or newline. Special characters are `[]() ?:\.;,!` ; they are used as keywords in the language. for example:

```
toto fooéé #er 123
```

are four identifiers. Notice that `123` is not recognized as an integer, as in the traditional language, but as an identifier, as explained in the next section.

2.3. Data and operator

lambda-flow does not specify the data and their associated operators. These specifications are written in an algebra file, dynamically loaded in the compiler (4). Such an algebra defines an regular expression (4.2) to identify an identifier as its own data. **lambda-flow** supports several regular expressions modes (4.3. By default, it uses the mode used by the lexical parser `lex`. The regular expression used to identifier the integers is:

```
check = flow: "[+-]?[0-9]+"
```

This definition is a part an an algebra file. The way to build a new algebra is explained later, when the real algebra is defined (ref id=algebra t=X//).

lambda-flow defines by default the integer algebra, because it needs the integer. The type associated to this algebra is `int`. This algebra is described in a file `integer.alg` (4.6).

An operator in **lambda-flow** is also defined in the algebra file. It has a name, a signature and a comment. For example, the line that defines the addition integer operator in the integer algebra file is:

```
+ = int->int->int, integer addition
```

where + is the operator name, `int->int->int` the signature and `integer addition` the comment. The signature is given in the denotational form where the right-most type is the type of the value returned by the operator, and the others types are the type of the arguments.

Of course, **lambda-flow** can deals with operator with identical name. For example, the `real` addition operator could be defined as:

```
+ = real->real->real, real addition
```

When the + operator is encountered, **lambda-flow** checks the signature of the arguments, and chooses the good operator. **lambda-flow** does not defined a type-tower as in the other language. Such a mechanism could convert an integer into a real if it is added to a real.

2.4. Definition

A definition allows to associate a name to a value. Notice we use the verb associate and not assign: **lambda-flow** is a language with equations, not an imperative one. Everywhere in the current environment, the name becomes a synonym of the associated value. The environment mechanism is explained later. For example:

```
baz is foo;
```

```
foo is 3;
```

defines two identifiers `baz` and `foo`. `baz` is associated to the value `foo` and `foo` is associated to the value `3`.

Notice that the order of the definitions is not important: here, `foo` is defined *after* `baz` uses it. But in **`lambda-flow`**, the word *after* has no meaning.

The value of a definition could be whatever a **`lambda-flow`**expression. The name must be an identifier. In addition, a definition cannot occurs inside another expression such as `foo := baz := 3`. A definition is named a root object.

In **`lambda-flow`**, an expression must be ended with the character `;`, such as in the most popular languages.

If an identifier is defined twice, only the first definition is considered.

`lambda-flow` supports two syntax levels: a long-syntax mode and a short-syntax mode. The long mode uses explicit keyword, such as `is` while the short mode uses symbols. The symbol associated to `is` is `:=`. The example given above become:

```
baz := foo;
foo := 3;
```

2.5. Alternative

An alternative is a choice between two expressions called the *then-clause* and the *else-clause* according to a *condition*. Here, it is important to forget the notion of side effect, because **`lambda-flow`** has no side effect. So, it is not important to know if the *condition* is evaluated before the *then-clause*, or if only one of the *then-clause* or the *else-clause* is evaluated. For example:

```
if x then 1 else 2;
```

is an alternative. Its value depends on the value of the identifier `x`. The condition, and the two clauses could be whatever a non-root **`lambda-flow`** expression. The condition must have the integer signature: `0` plays the role of `false` and the other integers play the role of `true`.

The other syntax used for alternatives is:

```
x ? 1 : 2;
```

The alternative object allows to build multi-rate programs.

2.6. Application

An application applies some arguments to an operator. For example:

```
+ (1, 2);
```

is the application of the operator `+` to the argument `1` and `2`. In order to choose the good operator, **`lambda-flow`** first checks the signature of the arguments. In this expression, they are integer with the signature `int`. Then, it can choose among the loaded algebra an operator with the name `+` and with two integer arguments.

When an application has two arguments, it can be written in a more traditional way, such as:

```
1 + 2;
```

In this written, the two blanks between the operator are mandatory. Without any blank, **`lambda-flow`** understand the identifier `1+2`.

2.7. Stream

The stream allows the time to be handled in a program. It is a functional view of a state variable.

In imperative languages, the main elements is the storage cells and the way to access them. Generally, a cell is eccessed with the given name of the variable. A cell has a type, and only a value with the same type can be assigned.

With these two accesses mode, the imperative language naturally define the sequence, generally known as the control flow of the program. For an assignment instruction into a cell, there are two worlds: the world before the assignment, and the world after. The main problem with the assignment mechanism is the impossibility to know the state of a

big program at an instant, due to explosion of the state number.

The stream of data-flow languages is the functional expression of a state variable: it defines its initial value, and the way to compute all the next values. So it is possible to know the state of a program at each instant of its life. The gain is that a data-flow program supports formal proofs.

In **`lambda-flow`**, the stream expression is particularly simple: a stream is composed with an initial value named the *state*, and the expression of all the next values, named the *contract*. For example:

```
0 followed-by 1;
```

defines the stream of the integer `0` followed by the integer `1`. The authors of the language Lucid use the following syntax to deal with such a stream: they write `{0, 1, 1, ...}`. If the expression `x` is the stream `{1,2,3,..}`, the stream `y` defined as:

```
y := 0 followed-by x + 1;
```

is evaluated as `{0, 1, 2, 3, ...}`. The stream `x` could be defined with the expression:

```
x := 1 followed-by x + 1;
```

Streams can also be written with the short syntax, such as:

```
x := 1 x + 1;
```

2.8. Vector

A vector gathers some expression together in a same object. A vector starts with the keyword `begin` and ends with the word `end`. All the expressions it contains must be separated with a `;` (the last `;` could be omitted). For example:

```
begin
  1;
  f := 2;
  x := f + 1;
end;
```

is a vector. A vector plays the role of an environment frame: all the definition (2.4) it contains are visible in all the expressions it contains, even if they are vectors. Because a vector can contain a vector, a nested-environment mechanism is defined. In the vector above, the defined identifiers `f` and `x` are visible in all the expression the vector contains. In the following example:

```
begin
  a := 1;
  b := begin
    a := 2;
    c := 3;
    d := a + c;
  end;
end;
```

the evaluation of `d` is `5`. The identifier `a` is named a free variable of the vector `b`/. In this way, the free variables of a vector could be view as the inputs of a module. The vector object is the primitive tool for modularizing programs. This tool is greatly improved by the abstraction object.

Notice that the expression contained in a vector can be root expression, such the definition.

The other syntax for the vector is:

```
[
  f := 2;
  x := f + 1;
];
```

2.9. Output

An output is a root expression. It is used to explicitly export an expression outside of a vector. It is written:

```
foo output 1 + 3;
```

that exports the identifier `foo` outside of the vector that contains it. Of course, the variable of the the exported expression is bound in the environment where it is written. Notice this expression is not a definition: the identifier `foo` is not defined in the vector that contains it.

The other output syntax is:

```
foo ! 1 + 3;
```

2.10. Extraction

The extraction object is the counterpart of the output object: it can read an exported expression inside a vector. It is composed with an indexed expression and an index expressions, and it is written:

```
s := begin x!1; y!2; z!3; end;
t := s extract y;
```

Notice that **`lambda-flow`** does not attempt to bind the index identifier to the current environment: it tries to find a corresponding output in the indexed vector.

If there are two outputs with the same identifier in the indexed expression, only the first one is considered.

In this first version of **`lambda-flow`**, the indexed expression must be either a vector, or an identifier that refers to a vector or the instantiation (see below) of a vector.

`lambda-flow` supports numerical extraction where the index expression is an integer. The integer index is the numerical 1-based position of the indexed expression. In order to keep the functional property of the language, the first component of the indexed expression is returned if the index value does is not a valid index.

So, it is possible to write:

```
table := [123; 432; 234; 556;];
coef  := table extract x + 1;
```

In this case, **`lambda-flow`** generates the code:

The other syntax for the extraction is:

```
table := [123; 432; 234; 556;];
coef  :=      if (x + 1) = 1 then table extract 1
         else if (x + 1) = 2 then table extract 2
         else if (x + 1) = 3 then table extract 3
         else if (x + 1) = 4 then table extract 4
         else                     table extract 1;
```

As it can been see, if the index has not the good value, the default result is the first component of the table.

The short extraction syntax is:

```
coef := table . x + 1;
```

2.11. Abstraction

The abstraction object allows useful modularization of the applications. In order to keep the determinisms of the applications, the abstraction constructs are severely limited: they are not as powerful as in the lambda-calculus. Particularly, it is impossible to write the `Y` combinator, and all recursive construction. An abstraction is closest than the macro definition, such as in the C language. It is written:

```
lambda a, b, c. expr
```

where the identifier `a`, `b` and `c` are the parameters of the abstraction, and the `expr` the abstracted expression.

If `expr` has `a`, `b` or `c` as free variable, they are bound to the parameters that take their values where the abstraction is instantiated (2.13).

Notice the parameters could be typed or not. To add a type to a parameter, simply type:

```
lambda a:type1, b:type2, c:type3. expr
```

where `types` are the type names of loaded algebra. For example, `int` is a valid type name. Mixing typed and untyped

parameters is allowed.

The use of untypeded parameter encourages polymorphic definitions. For example, the following expression is defined whatever the type of the given arguments, if there is the operator `*` could be found in a loaded albegra.

```
sqr := lambda a. a * a;
```

The short syntax for an abstraction is:

```
\ a, b, c. expr
```

2.12. Abstraction declaration

Because **`lambda-flow`** allows the separate compilation of the source files, it is needed to declare the abstractions used in a file, without define them.

This is achieve with the writing :

```
lambda a, b, c
```

without any body. Generally, this object is used with a definition.

2.13. Instantiation

The instantiation syntax is identical than the application syntax. The `sqr` abstraction defined above is instantiated with the writing:

```
foo := sqr (3);
```

The underlined mechanism of the instantiation is closest to the macro-replacement than a function call. In fact, the body of the abstraction is put in place of the instantiation. Of course, this seems very simple. But **`lambda-flow`** keeps the static lexical binding, and this point is the main difficulty: the free variables of an abstraction are linked in the environment **where** the abstraction is defined instead of the environment where the abstraction is used. This kind of binding is very useful and currently used in almost the modern functional language.

2.14. Program

A **`lambda-flow`** program is a set of expressions which can be fit in several files. Among all these expressions, there is one module named the `main` module. It is a definition with `main` as identifier that defines an abstraction with or without parameters.

If they are, the parameters **must be have a type**: they play the role of the inputs of the program.

The body of the abstraction must be a vector. It could contain several outputs: in this case, they are the outputs of the application.

One of these outputs could play the role of the stopping condition: the program stops when the value of this output is reach a non-zero value. The stopping condition is specified as an option of the compiler command line (7.1).

Chapter 3. Semantics aspects

The compiler performs a strong verification of the programs before produces the target code. This verification checks if all the used identifiers are defined, if there is no-recursive cycles and then, it performs a type checking.

3.1. Closure

The closure deals with the variable definitions. An expression is closed if all the used identifiers are defined. Because **lambda-flow** allows a separate compilation, it is not an error if an identifier is not defined, but in this case, the code production is avoided.

lambda-flow has a static lexical scope strategy for the identifier. This way is more clearest for the final user than the dynamic lexical scope. Let us take this example:

```
a := 3;
f := lambda x. x + a;
begin
  a := 4;
  f (3);
end;
```

In this toy example, the free identifier `a` of `f` is linked to the topmost definition of `a`, and not to the `a` inside the vector: the identifiers are linked in the *definition place* instead of the *use place*. This rule is very natural for the final user, but it severely complicated the compiler.

lambda-flow allows the separate compilation of the programs. The separate compilation produce a **lambda-flow** program where all the known identifiers were bound to their value.

3.2. Calculability

The calculability deals with the cycle-detection in a program. The time determinism constraint imposes to forbid such cycles. There are two kind of cycle in **lambda-flow**: the fix-point equations, and the recurrent ones.

A fix-point occurs when an identifier needs the value it has to be evaluated. For example:

```
x := x + 1;
```

defines a fix-point equation on `x`. Some fix-point equations have a solution, and some have no solution. But they are all evaluated in an indeterministic duration. In a program, cycle are sometime as direct as in the example above, but sometime, they are difficult to find. In order to keep the time determinism of the programs, **lambda-flow** rejects programs that contain fix-point equations.

A recurrent equation has the same form than a fix-point equation, but the identifier is considered at two different instants. For example:

```
x := 0 followed-by x + 1;
```

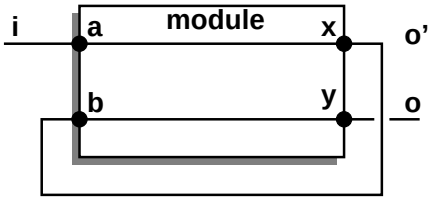
defined the stream of the natural integer, which can be written `{0, 1, 2, ...}`. The two `x` are considered at two different instants. This construction is the functional representation of a variable in imperative languages. The difference is that the evolution of the variable is perfectly known with a stream definition.

But **lambda-flow** has to check cycle in the stream contract itself. For example, the contract of the following stream cannot be evaluated:

```
x := [0; 0] followed-by [a:=a + 1; 2];
```

because the definition of `a` creates a fix-point equation.

lambda-flow uses a complex algorithm to find the cycles. For example, the following graph:



does not contains any cycle. It could be implemented with:

```
module := lambda a, b. begin
  x ! a;
  y ! b;
end;

instance := module (i, o');
o'       := instance . x;
o        := instance . y;
```

where `instance` is the instance of `module`. It is clear that the definition of `o` seems to create a cycle. The analysis of such instantiation is complex.

3.3. Constancy

The constancy deals with the type-checking. **`lambda-flow`** has a type strategy very natural for the final user: explicit type declarations can be entirely avoided for a program, except for the main inputs. The main inputs must be signed as `main := lambda in1 : type1, in2 : type2, ....`

The signatures[1] of the expressions are deduced from the type of the data, and when they exist, the type of the parameter. A data has only one type: **`lambda-flow`** does not implements the types tower, as in some other languages: the user has to convert explicitly the data on need.

The construction of the language allows the compiler to be always able to deduce the signature of the expressions. This encourage the polymorphic definitions. For example, the following abstraction:

```
sum := lambda init, unit. [
  state := init followed-by state + unit;
  out ! state;
];
```

is type independent. For example, `sum` could be instantiated with:

```
integers := sum(0, 1) . out;
```

or with:

```
complexes := sum(0+i0, 1+i0);
```

where `a+ib` is recognized by the complex algebra[2]. In these two instantiations, the `+` operator is not the same: in the first instantiation, it is the integer addition operator while in the second instantiation, it is the complex addition operator. The `sum` abstraction is polymorphic because it could be used with all the algebra that define the `+` operator.

If the `sum` abstraction is wanted to be defined only for the complex, its definition could be replaced with:

```
sum   := lambda init:complex, unit:complex. [
state := init followed-by state + unit;
out    ! state;
];
```

---

[1] the word *type* is prefered for the date, and the word *signature* is prefered for the other expressions.

[2] Notice that 0+i0 is not recognized as `0` followed by `+`, `i` and `0`, but as an alone string recognized by the complex algebra

With this new definition, the instantiation `sum(0, 1)` produces a compiler error because the given arguments have not the good type.

The polymorph feature is very useful for the designer that conceives general programs.

Chapter 4. Algebra

An algebra is composed by a data-type and some operators. The data-type is materialized with a regular expression (4.2) that recognizes or not the given string. The operators are composed with a name, a signature and a comment.

The number of algebra is not limited, and an algebra could use the type of other algebra.

In this section, we will discuss about the default integer algebra file which could be found in the distribution.

4.1. General form

The text file that describes an algebra has this form:

```
[type]
name    = int
comment = Basic algebra for integer arithmetic
check   = flow: "[+-]?[0-9]+"

[operators]
+/-     = int->int,      change sign
+       = int->int->int, addition
...
```

where, `int` is the type name. The `check` field is discussed above. The number of the operator is not limited. The signature of the operator could use type of other algebra. For example, it is possible to add an integer to real converter operator with:

```
[operators]
...
int2real = int->real, integer to real convert
...
```

The operator name is not reserved. For example, the `real` algebra also defines the addition operator with:

```
+        = real->real->real, addition
```

The integer addition and the real addition are distinguished with the type of their arguments.

It is important to notice that the algebra do not specify the meaning of the operator, but only the way to know them.

The more important part of an algebra definition is the regular expression, explained in the next section.

4.2. Regular expression

`lambda-flow` uses regular expressions loaded from an algebra file to recognizes the data. A kind of regular expression is the operating systems shells that replace the `*` character with all the files of the current directory.

More generally, a regular expression is a text string that describes some set of strings. A regular expression `r` *matches* a string `s` if `s` is in the set of strings described by `r`.

4.3. Modes

The **`lambda-flow`** regular implementation is based on the `regex` package of **Karthryn A. Hargreaves** and **Karl Berry** (`ftp:`/prep.ai.mit/).

`lambda-flow` allows several regular expression mode. The mode is the first world. In the integer regular expression, the mode is `flow`.
The mode could be either: `awk`, `grep`, `egrep`, `ed`, `sed`, `posix-awk`, `posix_egrep`, `posix_basic`, `posix-minimal`, `posix-extended`, `posix-minimal-extended` or `flow`.

All the mode names correspond to programs of the UNIX system. The simplest mode is `flow` that implements regular expressions closest to the lexical parser `lex`.

4.4. `lambda-flow` mode

The **`lambda-flow`** regular expression mode is closest to the lexical parser `lex`. A regular expression is a string where each character is an operator.

4.4.1. The match self operator (ordinary)

This operator matches the character itself. All ordinary characters represent this operator. For example, `f` is always an ordinary character, so the regular expression `f` matches only the string `f`. In particular, it does not match the string `ff`.

4.4.2. The match-any-character operator (`.`)

This operator concatenates two regular expressions `a` and `b`. No character represents this operator; you simply put `b` after `a`. The result is a regular expression that will match a string if `a` matches its first part and `b` matches the rest. For example, `xy` (two match-self operators) matches `xy`.

4.4.3. Repetition Operators

Repetition operators repeat the preceding regular expression a specified number of times.

4.4.3.1 The Match-zero-or-more Operator (`*`)

This operator repeats the smallest possible preceding regular expression as many times as necessary (including zero) to match the pattern. `*` represents this operator. For example, `o*` matches any string made up of zero or more `o`s. Since this operator operates on the smallest preceding regular expression, `fo*` has a repeating `o`, not a repeating `fo`. So, `fo*` matches `f`, `fo`, `foo`, and so on.

4.4.3.2 The Match-one-or-more Operator (+)

This operator is similar to the match-zero-or-more operator except that it repeats the preceding regular expression at least once for what it operates on, how some syntax bits affect it.

For example, `ca+r` matches, e.g., `car` and `caaaar`, but not `cr`.

4.4.3.3 The Match-zero-or-one Operator (?)

This operator is similar to the match-zero-or-more operator except that it repeats the preceding regular expression once or not at all to see what it operates on, how some syntax bits affect it.

For example, `ca?r` matches both `car` and `cr`, but nothing else.

4.4.3.4 Interval Operators (`{...}`)

- `{count}` matches exactly `count` occurrences of the preceding regular expression ;
- `{min,}` matches `min` or more occurrences of the preceding regular expression ;
- `{min, max}` matches at least `min` but no more than `max` occurrences of the preceding regular expression.

The interval expression (but not necessarily the regular expression that contains it) is invalid if either `min` is greater than `max`, or any of `count`, `min`, or `max` are outside the range zero to `65535`.

4.4.4. The Alternation Operator (`|`)

Alternatives match one of a choice of regular expressions: if you put the character(s) representing the alternation operator between any two regular expressions `a` and `b`, the result matches the union of the strings that `a` and `b` match. For example, `foo|bar|quux` would match any of `foo`, `bar` or `quux`.

The alternation operator operates on the *largest* possible surrounding regular expressions. (Put another way, it has the lowest precedence of any regular expression operator). Thus, the only way you can delimit its arguments is to use grouping. For example, if `(` and `)` are the open and close-group operators, then `fo(o|b)ar` would match either `fooar` or `fobar`. (`foo|bar` would match `foo` or `bar`.)

The matcher usually tries all combinations of alternatives so as to match the longest possible string. For example, when matching `(fooq|foo)*(qbarquux|bar)` against `fooqbarquux`, it cannot take, say, the first ("depth-first") combination it could match, since then it would be content to match just `fooqbar`.

4.4.5. List Operators (`[...]` and `[...]`)

Lists, also called *bracket expressions*, are a set of one or more items. An *item* is a character, a character class expression, or a range expression. The syntax bits affect which kinds of items you can put in a list. We explain the last two items in subsections below. Empty lists are invalid.

A *matching list* matches a single character represented by one of the list items. You form a matching list by enclosing one or more items within an *open-matching-list operator* (represented by `[`) and a *close-list operator* (represented by `]`).

For example, `[ab]` matches either `a` or `b`. `[ad]*` matches the empty string and any string composed of just `a` and `d` in any order. A regular expression with a `[` but no matching `]` is considered as invalid.

Non-matching lists are similar to matching lists except that they match a single character *not* represented by one of the list items. You use an *open-nonmatching-list operator* (represented by `[`[1]) instead of an open-matching-list operator to start a nonmatching list.

[1]The is not considered to be the first character in the list. If you put a character first in (what you think is) a matching list, you'll turn it into a nonmatching list.

For example, `[ ab]` matches any character except `a` or `b`.

Most characters lose any special meaning inside a list. The special characters inside a list follow.

- `]` ends the list if it's not the first list item. So, if you want to make the `]` character a list item, you must put it first ;
- `[:` represents the open-character-class operator if what follows is a valid character class expression ;
- `:]` represents the close-character-class operator if what precedes it is an open-character-class operator followed by a valid character class name ;
- `-` represents the range operator if it's not first or last in a list or the ending point of a range.

All other characters are ordinary. For example, `[.*]` matches `.` and `*`.

4.4.5.1 Character Class Operators (`[: ...:]`)

A *character class expression* matches one character from a given class. You form a character class expression by putting a character class name between an *open-character-class operator* (represented by `[:`) and a *close-character-class operator* (represented by `:]`). The character class names and their meanings are:

- `alnum` letters and digits
- `alpha` letters
- `digit` digits
- `lower` lowercase letters
- `punct` neither control nor alphanumeric characters
- `upper` uppercase letters
- `xdigit` hexadecimal digits: `0`–`9`, `a`–`f`, `A`–`F`

These correspond to the definitions in the C library's `ctype.h` facility. For example, `[:alpha:]` corresponds to the standard facility `isalpha`. character class expressions are recognized only inside of lists; so `[[:alpha:]]` matches any letter, but `[:alpha:]` outside of a bracket expression and not followed by a repetition operator matches just itself.

4.4.5.2 The Range Operator (`-`)

Range expressions represent those characters that fall between two elements in the current collating sequence. You form a range expression by putting a *range operator* between two characters[1]. `-` represents the range operator. For example, `a-f` within a list represents all the characters from `a` through `f` inclusively.

4.4.6. Grouping Operators (`( . . . )`)

A *group*, also known as a *subexpression*, consists of an *open-group operator*, any number of other operators, and a *close-group operator*. This sequence is treated as a unit, just as mathematics and programming languages treat a parenthesized expression as a unit.

Therefore, using *groups*, you can:

- delimit the argument(s) to an alternation operator or a repetition operator ;
- keep track of the indices of the substring that matched a given group. This lets you either use the back-reference operator or use registers.

4.5. `lambda-flow` interactive regular expression tester

The **lambda-flow** compiler offers a reasonable interactive tool to test the regular expressions. Simply type in the command line:

```
$ flow  –regExCheck
```

[1] You can't use a character class for the starting or ending point of a range, since a character class is not a single character.

The compiler enters in an interactive mode and waits on a prompt. There are three levels, where you can chose the mode, enter the regular expression and test some strings. To leave a level in order to go in the upper level, simply type `ENTER` (typing `ENTER` in the mode selection level leaves the compiler).

The mode selection level let you chose the regular expression mode among the one cited above. On the prompt, type the selected mode. For example, type `flow`.

Then you are in the regular expression selection level. Here, you can either return to the mode selection level with `ENTER` or enter a regular expression. For example, enter the regular expression of the integer algebra, `[+-]?[0-9]+`, and then, press `ENTER`.

Then, you are in the string test level where you can test some string. Here, you can return to the upper level with `ENTER` or test a string. For example, type `123`. The compiler displays `match` because the entered string matches to the entered regular expression. Now, type `toto` and the compiler responds `no match`.

Press the key `ENTER` three times to return to the operating system.

4.6. The integer algebra

Here, we show the complete integer algebra provided with **`lambda-flow`**:

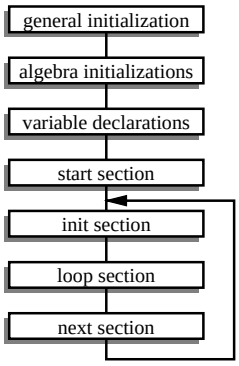
```
[type]
name    = int
comment = Basic algebra for integer arithmetic
check   = flow: "[+-]?[0-9]+"

[operators]
+/-     = int->int,      change sign
+       = int->int->int, add
-       = int->int->int, subtract
*       = int->int->int, multiply
/       = int->int->int, divide
%       = int->int->int, modulo
=       = int->int->int, equality
>       = int->int->int, greater
<       = int->int->int, smaller
>=      = int->int->int, greaterEq
<=      = int->int->int, smallerEq
<>      = int->int->int, different
<<      = int->int->int, shift left
>>      = int->int->int, shift left
&       = int->int->int, shift right
|       = int->int->int, shift right
&&      = int->int->int, shift right
||      = int->int->int, shift right
zero    = int->int,      is zero
```

Chapter 5. target code

lambda-flow is independent with the target code that is defined in a file called *target code definition*. Therefore, the produced code has always the same structure, as shown in the next section.

5.1. General format of the target code

The **lambda-flow** produced code has always the same structure. This structure is independent with the real produced code, specified in the *target code definition file*.



The code has an initialization phase, a temporary variable evaluation and a stream regeneration phase. Then it does a loop.

general initialization: initialization of the whole program. Here, some files could included, or some global variables initialized ;

algebra initialization: each used algebra could initialize itself here. An algebra could initializes a variable or defines some things ;

variables declarations: all the inputs, the stream states and the temporary variables are declared here;

start section: this section of code compute all the necessary values for stream initialization. Notice that the stream states are not initialized here;

init section: the inputs and the stream states are initialized here, possibly with the values computed in the `start section`;

loop section: the loop section compute all the temporary variables used to compute the outputs of the program and the next stream state values;

next section: the new values of the stream state are assigned here, as the new values of the inputs;

loop: this section contains the code to jump to the `init section`.

5.2. Target code definition file

A *target code definition file* (TCDF) is a text file that implement the operators of the algebra in a specific target code. It is organized with two main parts: a general definition part and an algebra implementations part.

The TCDF is based on template strings. A template string is a string where some value could placed. A well

know template string form is the C `printf()` function format string, where the `%x` templates are replaced with the corresponding values.

Each template in a TCDF has a name and a template string value. For example, a template could be:

```
command = "gcc %s %s -o %s"
```

that defines the `command` template as the string `“gcc %s -o %s”`. The TCDF supports several template string formats, as explained in the next section.

5.3. Template strings format

A template string is the value of a template. The TCDF has three template string format:

`simple string` : the value of the template is the given string without any replacement. The string is written as is, without any syntactic marker, as `template = the value`. This string could be empty if a value is optional;

`“printf()”` **format** : the value of the template is a string where some `%s` appear. The whole string is written into `“”`. The number of `%s` is determined by the considered template. For example, the `command` template has three values: the command options, the command input file and the command output file, given to the template string in this order. Using this format allows `\n` replacement in the string;

`slang()` **format** : the **`slang`** template string format has the following form: `slang (parameters) {slang function body}` where `parameters` are the name of the parameters, and `body` the **`slang`** function body. The `command` template string could be replaced with `slang (opt, in, out) {return Sprintf (“gcc %s %s -o”, opt, in, out, 3);}`. The **`slang`** language is explained in a next section (6).

5.4. General definitions in a target code definition file

The general definitions in a TCDF are grouped under the section `[target]`. Under this section there are several template and several options, as explained above.

5.4.1. `extension` template

This template is the extension of the file generated by the **`lambda-flow`** compiler when the user want to keep the auxiliary file. This template has no argument.

The `extension` template of the C-TCDF is:

```
extension = c
```

5.4.2. `command` template

The `command` template is the post-compiler to run after **`lambda-flow`** has produced the auxiliary file. Generally, it is either a compiler or an interpreter.

This template has three arguments, the command options, the input file which is the auxiliary file name and the output file name specified by the user.

The `command` template of the C-TCDF is:

```
command = "gcc %s %s -o %s"
```

5.4.3. `linear` option

The `linear` option is either `yes` or `no`. It is not a template, but an option. When it is set to `yes`, **`lambda-flow`**produces a linear code.

The `linear` template of the C-TCDF is:

```
linear = yes
```

With the C language, this option could be `no` because the C language supports recursive expression construction.

5.4.4. `comment` template

The `comment` template is the line-comment form of the target language. It has one argument, the string to be commented in the auxiliary file.

The `comment` template of the C-TCDF is:

```
comment = "/* %s */"
```

5.4.5. `width` option

The `width` option indicate the indentation used by **`lambda-flow`** in the auxiliary file to put its comments.

The `extension` template of the C-TCDF is:

```
width = 50
```

5.4.6. `init` template

The `init` template is a text to put on the top of the auxiliary file. It has no argument. It could be used to declare and initialize some **`slang`** variables.

The `init` template of the C-TCDF is:

```
init ="\
/*C TARGET CODE*/\
#include <stdio.h>\n"
```

This definition shows that the template string could be written in more than one line, with the escape character `\` at the end of the inner lines.

This template has the `printf()` format to allow the character `\n` to be replaced with a newline in the output file.

5.4.7. `identifier` template

The `identifier` template has one argument,an identifier name. It is used to allows the identifier name to be conform to the target language. **`lambda-flow`** generates identifier that contains numeric value and underscore character.

This template could be used to replace these characters if the target language does not support them in the identifier name.

The `extension` template of the C TCDF is:

```
identifier = "%s"
```

5.4.8. `alternative` template

The `alternative` template is the translation in the target language of the alternative. It has three argument, the condition which is an integer where 0 denotes the false value and the other values denote the true ones, the then-clause and the else-clause.

If the `linear` option is not set, the argument could be complex expression. If it is set, the argument are either simple data or identifier.

The `alternative` template of the C TCDF is:

```
alternative = (%s) ? (%s) : (%s)
```

5.4.9. `exit` template

The `exit` template is used to check if the stopping condition is reached. It has one argument, the stopping condition. Generally, this template call the `exit` command of the target language.

The `exit` template of the C TCDF is:

```
exit = "if (%s) exit(0);"
```

5.4.10. `pre-start`, `pre-init`, `pre-loop`, `pre-next` and `post-next` templates

These templates are placed on the top of the corresponding sections, except `post-next` placed on the bottom of the `next` section (5.1).

These templates have no argument.

These templates of the C TCDF are:

```
pre-start    = "\nmain() {\n  start:"
pre-init     = "\n  init:"
pre-loop     = "\n  loop:"
pre-next     = "\n  next:"
post-next    = "    goto loop;\n}"
```

5.5. Algebra dependent definitions in a target code definition file

For each algebra supported by the considered target code, a section must be created in the TCDF, with the type name of the algebra. In the C TCDF, it can be found the `[int]` section that correspond to the integer algebra.

This section defines some general template, and the templates of all the supported operators of the algebra.

5.5.1. `init` template

This template is used to initialize the algebra (4) in the considered target code. This template has no argument.

The `init` template of the C TCDF for the integer algebra is:

```
init = /* get/putint functions */\
int getint(int port) {\
  int tmp;\
  if (scanf ("%%d", &tmp) == EOF) exit(0);\
  return tmp;\
}\
#define putint(port,value) printf("%%d ", value)\n
```

5.5.2. `data` template

This template is used to convert a data recognized by the `check` regular expression of the corresponding algebra (4.1) into a data of the target code.

This template has one argument, the data. This template is generally used with the assembler languages that have to prefix the direct data.

The `data` template of the C TCDF for the integer algebra is:

```
data = "%s"
```

5.5.3. `declare` template

This template is used in the declaration section (5.1) to declare the identifier with the considered type.

It has two arguments, a type name and an identifier. The identifier is treated by the `identifier> template of the TCDF (5.4.7). The type name is the type used by the algebra (4.1).`

The `declare` template of the C TCDF for the integer algebra is:

```
declare = "%s %s;"
```

5.5.4. `assign` template

This template is used in all the sections of the auxiliary file to put a value in the variables. It has two arguments, the identifier name and the value.

The identifier is treated by the `identifier> template of the TCDF (5.4.7).`

The value is the production of one of the target templates.

The `assign` template of the C TCDF for the integer algebra is:

```
assign = "%s = %s;"
```

5.5.5. Operators templates

For each used operator of an algebra, **`lambda-flow`** must find a template in the TCDF.

The template is formed with the signature of the operator, followed by the template string itself. The number of argument of the template string depends on the considered operator.

For example, the template of the C TCDF for the integer algebra is for the `+` operator defined in the algebra (4.6) is:

```
+ = int->int->int, "(%s) + (%s)"
```

5.5.6. Input/output operators templates

For each algebra, the TCDF must contains two additional operators which are not defined in the algebra: they are the input/output operator.

`lambda-flow` uses the name `@in` and `@out` for these operators. The input operator has one argument, the input port number, as an integer. It returned value has the type of the considered algebra.

The output operator has two arguments, the output port number as an integer and the output value. The type of the second argument depends on the considered algebra.

The template of the C TCDF for the integer algebra is for the input/output operators are:

```
@in  = int->int,      "getint (%s)"
@out = int->int->int, "putint (%s, %s)"
```

Notice that the non-standard C function `getint()` and `putint()` are defined in the `init` template of the integer algebra of the C TCDF (5.5.1).

5.6. Samples of target code definitions

In this section, we present three TCDF for three very different language, such as C, Scheme and the 386 assembler.

5.6.1. The C target code definition file

```
[target]
extension    = c
command      = "gcc %s %s -o %s"
linear       = yes
comment      = "/* %s */"
width        = 50
init         = #include <stdio.h>\n

identifier   = "%s"
alternative  = "(%s) ? (%s) : (%s)"

pre-start    = \nmain() {\n  start:
pre-init     = "\n  init:"
pre-loop     = "\n  loop:"
exit         = "    if (%s) exit(0);"
pre-next     = "\n  next:"
post-next    = "    goto loop;\n}"

[int]
declare      = "static int %s;"
assign       = "    %s = %s;"
init         = "/* get/putint functions */\
int getint(int port) {\
  int tmp;\
  if (scanf ("%%d", &tmp) == EOF) exit(0);\
  return tmp;\
}\
#define putint(port,value) printf("%%d ", value)\n"

data         = "%s"

+/-          = int->int,        "- (%s)"
+            = int->int->int,   "(%s) + (%s)"
-            = int->int->int,   "(%s) - (%s)"
*            = int->int->int,   "(%s) * (%s)"
/            = int->int->int,   "(%s) / (%s)"
%            = int->int->int,   "(%s) % (%s)"
=            = int->int->int,   "(%s) == (%s)"
>            = int->int->int,   "(%s) > (%s)"
<            = int->int->int,   "(%s) < (%s)"
>=           = int->int->int,   "(%s) >= (%s)"
<=           = int->int->int,   "(%s) <= (%s)"
<>           = int->int->int,   "(%s) != (%s)"
<<           = int->int->int,   "(%s) << (%s)"
>>           = int->int->int,   "(%s) >> (%s)"
&            = int->int->int,   "(%s) & (%s)"
|            = int->int->int,   "(%s) | (%s)"
&&           = int->int->int,   "(%s) && (%s)"
||           = int->int->int,   "(%s) || (%s)"
zero         = int->int,        "(%s)"
@in          = int->int,        "getint (%s)"
@out         = int->int->int,   "putint (%s, %s)"
```

5.6.2. The Scheme target code definition file

```
[target]
extension    = scm
command      =
linear       = no
comment      = "; %s"
width        = 50
init         =

identifier   = "%s"
alternative  = "(if %s %s %s)"

pre-start    = "(let* ("
pre-init     = "      )\n(let loop ("
pre-loop     = "          )\n(let* ("
exit         = "(if (not (zero? %s)) (exit))"
pre-next     = "       )\n(loop "
post-next    = "))))\n"

[int]
declare      =
assign       = "  (%s %s)"
init         = "; get/putint functions\
(define (getint port) (read))\
(define (putint port value) (display value))\
(define (bool->int bool) (if bool 1 0))\
(define (int->bool  int) (if (zero? int) #f #t))\n"

data         = "%s"

+/-          = int->int,        "(- %s)"
+            = int->int->int,   "(+ %s %s)"
-            = int->int->int,   "(- %s %s)"
*            = int->int->int,   "(* %s %s)"
/            = int->int->int,   "(/ %s %s)"
%            = int->int->int,   "(modulo %s %s)"
=            = int->int->int,   "(bool->int (eq? %s %s))"
>            = int->int->int,   "(bool->int (> %s %s))"
<            = int->int->int,   "(bool->int (< %s %s))"
>=           = int->int->int,   "(bool->int (>= %s %s))"
<=           = int->int->int,   "(bool->int (<= %s %s))"
<>           = int->int->int,   "(not (eq? %s %s)))"
<<           = int->int->int,   "(* %s (* 2 %s))"
>>           = int->int->int,   "(inexact->exact (/ %s (* 2 %s))"
&            = int->int->int,   "(error)"
|            = int->int->int,   "(error)"
&&           = int->int->int,   "(bool->int (and (int->bool %s)(int->bool %s)))"
||           = int->int->int,   "(bool->int (or  (int->bool %s)(int->bool %s)))"
zero         = int->int,        "(zero? %s)"
@in          = int->int,        "(getint %s)"
@out         = int->int->int,   "(putint %s %s)"
```

5.6.3. The 386 assembler target code definition

This TCDF intensively uses the **`slang`** language to perfor proper assembly output.

```
[target]
extension    = s
```

```
command      = "gas %s %s -o %s"
linear       = yes
comment      = "; %s"
width        = 20
init         = slang () {\
                 return ".data";\
               }\
               variable label = 0;\
               define alternative (a1, _cond, a2, _then, _else) {\
                 label++;\
                 return Sprintf ("L_if_%d:       \n \
                                    cmp %s, %s    \n \
                                    j%s L_then_%d \n \
                                   L_else_%d:\n%s \n \
                                    jmp L_end_%d  \n \
                                   L_then_%d:\n%s \n \
                                    jmp L_end_%d  \n \
                                   L_end_%d:", \
                                  label, a1, a2, _cond, label,      \
                                         label, _else, label,       \
                                         label, _then, label, label,\
                                  12);\
               }
identifier   = "%s"
alternative  = slang (_cond, _then, _else) {\
                 variable __then = Sprintf (" mov.w ax, %s", _then, 1),\
                          __else = Sprintf (" mov.w ax, %s", _else, 1);\
                 return alternative (_cond, "ne", "0", __then, __else);\
               }
pre-start    = "\n\n.text\nmain:"
pre-init     = "\ninit:"
pre-loop     = "\nloop:"
exit         = "\nexit:\n cmp.w %s, 0\n je continue\n jmp end\n\ncontinue:"
pre-next     = "\nnext:"
post-next    = "\n jmp loop\n\nend:\n"

[int]
init         =
declare      = " %s .word"
assign       = slang (name, value){\
                 if (is_substr (value, " "))\
                   return Sprintf ("%s\n mov.w %s, ax", value, name, 2);\
                 else \
                   return Sprintf (" mov.w %s, %s", name, value, 2);\
               }
init         =
data         = "$%s"

+/-          = int->int,        " neg.w ax"
+            = int->int->int,       " add.w %s, %s"
-            = int->int->int,       " sub.w %s, %s"
*            = int->int->int,       " mov.w ax, %s\n imul.w %s"
/            = int->int->int,       " mov.w ax, %s\n idiv.w %s"
%            = int->int->int,       " mov.w cx, %s\n \
                                  mov.w ax, cx\n \
                                  mov.w bx, %s\n \
```

```
                                idiv.w bx\n    \
                                sub.w cx, ax"
=            = int->int->int,       slang (a1, a2) {
                                  return alternative (a1, "eq", a2, \
                                     " mov.w ax, 1", " mov.w ax, 0");
                                }
>            = int->int->int,       slang (a1, a2) {
                                  return alternative (a1, "g",  a2, \
                                     " mov.w ax, 1", " mov.w ax, 0");
                                }
<            = int->int->int,       slang (a1, a2) {
                                  return alternative (a1, "l",  a2, \
                                     " mov.w ax, 1", " mov.w ax, 0");
                                }
>=           = int->int->int,       slang (a1, a2) {
                                  return alternative (a1, "ge", a2, \
                                     " mov.w ax, 1", " mov.w ax, 0");
                                }
<=           = int->int->int,       slang (a1, a2) {
                                  return alternative (a1, "le", a2, \
                                     " mov.w ax, 1", " mov.w ax, 0");
                                }
<>           = int->int->int,       slang (a1, a2) {
                                  return alternative (a1, "ne", a2, \
                                     " mov.w ax, 1", " mov.w ax, 0");
                                }
<<           = int->int->int,       "   shl %s, %s"
>>           = int->int->int,       "   shr %s, %s"
&            = int->int->int,       "   and %s, %s"
|            = int->int->int,       "   or %s, %s"
            = int->int->int,       "   xor %s, %s"
&&           = int->int->int,       slang (a1, a2) {\
                                  variable _else = alternative (a2, "e", "0", \
                                     " mov.w ax, 0", " mov.w ax, 1");\
                                  return alternative (a1, "e", "0", " mov.w ax, 0",
_else);\
                                }
||           = int->int->int,       slang (a1, a2) {\
                                  variable _then = alternative (a2, "e", "0", \
                                     " mov.w ax, 0", " mov.w ax, 1");\
                                  return alternative (a1, "e", "0", \
                                     _then, "movw ax, 1");\
                                }
zero         = int->int,        slang (a1, a2) {
                                  return alternative (a1, "e", "0", \
                                     "  mov.w ax, 1", " mov.w ax, 0");
                                }
@in          = int->int,        " in %s"
@out         = int->int->int,       slang (port, value) {\
                                  return Sprintf (" mov.w ax, %s\n out %s",\
                                                  value, port, 2);\
                                }
```

Chapter 6. The `slang` language

The **`slang`** language comes from the package of **John E. Davis** which can be obtained in

`ftp://space.mit.edu/pub/davis/slang/`

In this section, we present a subset of this language useful for the TCDF template string.

6.1. Introduction

`slang` (pronounced "sssslang") is a powerful stack based interpreter that supports a C-like syntax. It has been designed from the beginning to be easily embedded into a program to make it extensible. **`slang`** also provides a way to quickly develop and debug the application embedding it in a safe and efficient manner. Since **`slang`** resembles C, it is easy to recode **`slang`** procedures in C if the need arises.

The **`slang`** language features both global variables and local variables, branching and looping constructs, as well as user defined functions. Unlike many interpreted languages, **`slang`** allows functions to be dynamically loaded (function auto-loading). It also provides constructs specifically designed for error handling and recovery as well as debugging aids (e.g., trace-backs).

The core language currently implements signed integer, string, and floating point data types. Applications may also create new types specific to the application (e.g., complex numbers). In addition, **`slang`** supports multidimensional arrays those types as well as any application defined types.

The syntax of the language is quite simple and is very similar to C. Unlike C, **`slang`** variables are untyped and inherit a type upon assignment. The actual type checking is performed at run time. In addition, there is limited support for pointers.

6.2. Variables

`slang` is an untyped language and only requires that an variable be declared before it is used. Variables are declared using the `variable` keyword followed by a comma separated list of variable names, e.g.,

```
variable larry, curly, moe;
```

As in C, all statements must end with a semi-colon. Variables can be declared to be either *global* or *local*. Variables defined inside functions are of the local variety and have no meaning outside the function.

It is legal to execute statements in a variable declaration list. That is,

```
variable x = 1, y = sin (x);
```

are legal variable declarations. This also provides a convenient way of initializing a variable.

The variable's type is determined when the variable is assigned a value. For example, in the above example, `x` is an integer and `y` is a float since `1` is an integer and the `sin` function returns a floating point type.

6.3. Functions

Like variables, functions must be declared before they may be used. The `define` keyword is used for this purpose. For example,

```
define factorial ();
```

is sufficient to declare a function named `factorial`. Unlike `variable` keyword, the `define` keyword does not accept a list of names. Usually, the above form is used only for recursive functions. The function name is almost always followed by a parameter list and the body of the function, e.g.,

```
define my_function (x, y, z) {
```

```
  <body of function>
}
```

Here `x`, `y`, and `z` are also implicitly declared as local variables. In addition, the function body must be enclosed in braces.

Functions may return zero, one or more values. For example,

```
define sum_and_diff (x, y) {
    variable sum, diff;

    sum = x + y;  diff = x - y;
    return sum, diff;
}
```

is a function returning two values.

Please note when calling a function that returns a value, the value returned cannot be ignored. See the section below on assignment statements for more information about this important point.

6.4. Statements and Expressions

A statement may occur globally outside of functions or locally within functions. If the expression occurs inside a function, it is executed only when the function is called. However, statements which occur outside a function context are evaluated immediately.

All statements must end in a semi-colon.

6.4.1. Assignment Statements

An assignment statement follows the syntax:

```
<ttiable name> = <expression>;
```

Whitespace is required on *both* sides of the equal sign. For example,

```
x = sin (y);
```

is correct but

```
x =sin(y);  x= sin(y); x=sin(y);
```

will generate syntax errors.

Often, functions return more than one value. For example,

```
define sum_and_diff (x, y) {
  return x + y, x - y;
}
```

returns two values. The most general assignment statement syntax is

```
(<var_1>,<var_2>,...,<var_n>) = <expression>;
```

e.g.,

```
(s, d) = sum_and_diff (10, 2);
```

To ignore one of the return values, simply omit the variable name from the list. For example,

```
(s, ) = sum_and_diff (10, 2);
```

may be used if one is only interested in the first return value.

Some functions return a variable number of values. Usually, the first value will indicate the actual number of return values.

For example, the `fgets` function returns either one or two values. If the first value is zero, there is no other return value. In this case, one must use another form of assignment since the previously discussed forms are inadequate. For example,

```
n = fgets (fd);
if (n != 0) {
```

```
   s = ();
   .
   .
}
```

In this example, the first value returned is assigned to `n` and tested. If it is non-zero, the second return value is assigned to `s`. The empty set of parenthesis is required.

Please note that RETURN VALUES CANNOT BE IGNORED. There are several ways of dealing with a return value when one does not care about it. For example, the function `fflush` returns a value. However, most C programs that call this function almost always ignore the return value. In **slang**, one can use any of the following forms:

```
variable dummy;
dummy = fflush (fd);

() = fflush (fd);

fflush (fd); pop ();
```

The second form is perhaps the most clear way of indicating that the return value is being ignored.

6.4.2. Binary Operators

slang supports a variety of binary operators. These include the usual arithmetic operators (`+`, `-`, `*`, /, and `mod`), the comparison operators (`>`, `>=`, `<`, `<=`, `!=`, and `==`) as well boolean operators (`or` and `and`) and bitwise operators (`|`, `\&`, `xor`, `shl` and `shr`). Like the assignment operator, these operators must also be surrounded by whitespace. That is,

```
x = y + z;
```

is a legal statement but `x = y+z;` is not legal.

To use these operators effectively, in addition to understanding the meaning of the operation, one must also understand the precedence level of the operator.

In **slang**, there are only three levels of precedence. The highest level consists of the `*`, /, and `mod` operators. The second level consists of the `+` and `-` operators. All other binary operators fall into the last level of precedence. Within a precedence level, operators are evaluated left to right. Parenthesis may be used to change the order of evaluation. For example, the expression:

```
a == b or c == d
```

IS NOT the same as:

```
(a == b) or (c == d)
```

since `==` and `or` share the same level of precedence. In fact, the expression without parenthesis is evaluated left to right and is equivalent to `((a == b) or a) == c`.

Finally, **slang** supports the increment and decrement operators `++` and `--`, and the arithmetic assignment operators `+=` and `-=`. Presently, these operators only work with integer types and a type mismatch error will result from the use of these operators with other types.

These following table shows the meaning of these operators.

```
Expression          Meaning
----------          -------
 ++x;                x = x + 1;
 x++;                x = x + 1;
 --x;               x = x - 1;
 x--;               x = x - 1;
 x += n;             x = x + n;
 x -= n;             x = x - n;
```

Note that **slang** does not distinguish between `x--` and `--x` since neither of these forms return a value as they do in C. With this in mind, do not use constructs such as:

```
while (i--) ....   % test then decrement
while (--i) ....   % decrement first then test
```

Instead, use something like

```
while (i, i–) .... % test then decrement
while (i–, i) .... % decrement first then test
```

These operators work only on simple scalar variables. In particular, `++(x)` is NOT the same as `++x` and will generate an error.

Whenever possible, these latter four operations should be used since they execute 2 to 3 times faster than the longer forms.

6.4.2.1 Short Circuit Boolean Evaluation

The boolean operators `or` and `and` ARE NOT SHORT CIRCUITED as they are in some languages. **`slang`** uses the `orelse` and `andelse` operators for short circuit boolean evaluation. However, these are not binary operators. Expressions of the form:

```
<expr_1> and <expr_2> and <expr_3> ... and <expr_n>
```

can be replaced by the short circuited version using `andelse`:

```
andelse {<expr_1>} {<expr_2>} {<expr_3>} ... {<expr_n>}
```

A similar syntax holds for the `orelse` operator. For example, consider the statement:

```
if ((x != 0) and (1 / x < 10)) do_something ();
```

Here, if `x` were to have a value of zero, a division by zero error would occur because even though `x != 0` evaluates to zero, the `and` operator is not short circuited and the `1 x/` expression would be evaluated. For this case, the `andelse` operator could be used to avoid this problem:

```
if (andelse
  {x != 0}
  {1 / x < 10})  do_something ();
```

6.4.3. Unary Operators

The UNARY operators operate only upon a single integer. They are defined by the following table below. In this table, the variable `i` is an integer type and `x` represents either a floating point or integer variable.

```
Unary Expr.  Meaning
———–   ——————————-
 not (i)      if i is non-zero return zero else return non-zero (i)         bitwise
not
 sqr(x)       the square of x
 mul2(x)      multiplies x by 2
 chs (x)      change the sign of x
 -x           same as chs (x)
 sign (x)     +1 if x > 0, -1 if x < 0, and 0 if x equals 0
 abs (x)      absolute value of x
```

Note the following points:

- All unary operators except `not` and operator on both integer and floating point types.
- The `!` operator used in C is not used in **`slang`**, `not` must be used instead.
- The bitwise not operator must enclose its argument in parenthesis. `i` will be flagged as a syntax error.
- Some applications which embed **`slang`** may overload these operators to work with application defined data types.

6.4.4. Data Types

Currently, **`slang`** only supports integer, floating point (double precision), and character string data types. It is possible for an application that embeds **`slang`** to define other, application specific, data types (e.g., complex numbers). In addition, the language supports arrays of any of these types (including application specific types).

6.4.4.1 Integers

Unsigned integers are not supported. An integer can be specified in one of several ways:

- As a decimal integer consisting of the characters `0` through `9`, e.g., `127`. The number cannot begin with a leading `0`. That is, `0127` is not the same as `127`.
- Using hexadecimal (base 16) notation consisting of the characters `0` to `9` and `A` through `F`. The hexadecimal number must be preceded by the characters `0x`. For example, `0x7F` is the same thing as decimal `127`.
- In Octal notation using characters `0` through `7`. The Octal number must begin with a leading `0`. For example, `0177` is the same thing as `127` decimal.
- Using character notation containing a character enclosed in single quotes as `'a'`. The value of the integer specified this way will lie in the range 0 to 256 and will be determined by the ASCII value of the character in quotes. For example,
  ```
  i = '0';
  ```
  results in a value of 48 for `i` since the character `0` has an ASCII value of 48.

  Strictly speaking, **slang** has no character type.

Any integer may be preceded by a minus sign to indicate that it is a negative integer.

6.4.4.2 Floating Point Numbers

Floating point numbers must contain either a decimal point or an exponent (or both). Here are examples of specifying the same floating point number:

```
12.,  12.0,  12e0,  1.2e1,  120e-1,  .12e2
```

Note that `12` is NOT a floating point number since it contains neither a decimal point nor an exponent. If fact, `12` is an integer.

6.4.4.3 Strings

A literal string must be enclosed in double quotes as in:

```
"This is a string".
```

Although there is no imposed limit on the length of a string, literal strings must be less than 256 characters. It is possible to go beyond this limit by string concatenation. Any character except a newline (ASCII 10) or the null character (ASCII 0) may appear in the *definition> of the string.*

The backslash is a special character and is used to include special characters (such as a newline character) in the string. The special characters recognized are:

```
\"    –  double quote
\'    –  single quote
\\    –  backslash
\a    –  bell character
\t    –  tab character
\n    –  newline character
\e    –  escape  (S-Lang extension)
\xhhh –  character expressed in HEXADECIMAL notation
\ooo  –  character expressed in OCTAL notation
\dnnn –  character expressed in DECIMAL (S-Lang extension)
```

For example, to include the double quote character as part of the string, it is to be preceded by a backslash character, e.g.,

```
"This is a \"quote\""
```

6.4.5. Mixing integer and floating point arithmetic

If a binary operation (+, -, * , /) is performed on two integers, the result is an integer. If at least one of the operands is a float, the other is converted to float and the result is float. For example:

```
11 / 2          -> 5   (integer)
```

```
11 / 2.0         -> 5.5 (float)
11.0 / 2         -> 5.5 (float)
11.0 / 2.0       -> 5.5 (float)
```

Finally note that only integers may be used as array indices, for loop control variables, shl, shr, etc bit operations. Again, if there is any doubt, use the conversion functions `int` and `float` where appropriate:

```
int (1.5)         -> 1 (integer)
float(1.5)        -> 1.5 (float)
float (1)         -> 1.0 (float)
```

6.4.6. Conditional and Branching Statements

`slang` supports a wide variety of looping (`while`, `do while`, `loop`, `for`, `forever`, and `_for`) and branching (`if`, `!if`, `else`, `andelse`, `orelse`, and `switch`) statements.

These constructs operate on code statements grouped together in *blocks*. A block is a sequence of **`slang`** statements enclosed in braces and may contain other blocks. However, a block cannot include function declarations; function declarations must take place at the top level. In the following, `statement` refers to either a single **`slang`** statement or to a block of statements and `{ block }` refers to a block of statements.

6.4.6.1 `if, if-else`

```
if (expression) statement;
```

Evaluates `statement` if the result of `expression` is non-zero. The `if` statement can also be followed by an `else`:

```
if (expression) statement; else statement;
```

6.4.6.2 `!if`

```
!if (expression) statement;
```

Evaluates `statement` if `expression` is evaluates to zero. Note that there is no `!if-else` statement.

6.4.6.3 `orelse, andelse`

These constructs were discussed earlier. The syntax for the `orelse` statement is:

```
orelse { block } { block } ... { block }.
```

This causes each of the blocks to be executed in turn until one of them returns a non-zero integer value. The result of this statement is the integer value returned by the last block executed. For example,

```
orelse { 0; } { 6; } { 2; } {3; }
```

returns `6` since the second block returns the non-zero result `6` and the last two block will not get executed.

The syntax for the `andelse` statement is:

```
andelse { block } { block } ... { block }.
```

Each of the blocks will be executed in turn until one of them returns a zero value. The result of this statement is the integer value returned by the last block executed. For example,

```
andelse { 6; } { 2; } { 0; } {4; }
```

returns `0` since the third block will be the last to execute.

6.4.6.4 `while`

```
while (expression) statement;
```

Repeat `statement` while `expression` returns non-zero. For example,

```
j = 20; i = 10; while (i) { j = j + i; i = i - 1; }
```

will cause the block to execute 10 times.

6.4.6.5 do-while

```
do statement; while (expression);
```

Execute `statement` then test `expression`. Repeat while `expression` is returns non-zero. This guarantees that `statement` will be executed at least once.

6.4.6.6 for

```
for (expr1; expr2; expr3) statement;
```

Evaluate `expr1` first. Then loop executing `statement` while `expr2` returns non-zero. After every evaluation of `statement` evaluate `expr3`. For example,

```
variable i, sum;
sum = 0;
for (i = 1; i <= 10; i++) sum += i;
```

computes the sum of the first 10 integers.

6.4.6.7 loop

```
loop (n) statement;
```

Evaluate `statement` `n` times. If `n` is less than zero, `statement` is not executed.

6.4.6.8 forever

```
forever statement;
```

Loop evaluating statement forever. Forever means until either a `break` or `return` statement is executed.

6.4.6.9 switch

The switch statement deviates the most from its C counterpart. The syntax is:

```
switch (x)
  { ... :  ...}
    .
    .
  { ... :  ...}
```

Here the object `x` is pushed onto the stack and the sequence of blocks is executed. The `:` operator is a **slang** special symbol which means to test the top item on the stack, if it is non-zero, the rest of the block is executed and control then passes out of the switch statement. If the test is false, execution of the block is terminated and the process repeats for the next block.

The special keyword `case` may be used to compare the value of objects. It returns non-zero if the objects correspond to the same object and zero otherwise.

For example:

```
variable x = 3;
switch (x)
  { case 1: print("Number is one.")}
  { case 2: print("Number is two.")}
  { case 3: print("Number is three.")}
  { case 4: print("Number is four.")}
  { case 5: print("Number is five.")}
  { pop(); print ("Number is greater than five.")}
```

Here `x` is assigned a value of 3 and the `switch` statement pushes the 3 onto the stack. Control then passes to the first block. The first block uses the `case` construct to compare the top top stack item (3) with 1. This test will result with zero at the top of the stack. The `:` operator will then pop the top stack item and if it is zero, control will be passed to the next block where the process will be repeated. In this case, control will pass to the second block and on to the third block. When the `:` operator is executed for the third block, a non-zero value will be left on the top of the stack and the `print` function will be called. Control then passes onto the statement following the last block of the

`switch` statement.

Note that, in this example, the last block does not test the value of `x` against anything. Instead, if this block is executed, the top stack item (the value of `x` in this case) will be removed from the stack by the `pop` function and the rest of the block executed.

Unlike most other languages with some form of switch statement, `x` does not have to be a simple integer. For example, the following is perfectly acceptable:

```
variable x;
x = "three";
switch (x)
  { case "one": print("Number is 1.")}
  { case "two": print("Number is 2.")}
  { case "three": print("Number is 3.")}
  { case "four": print("Number is 4.")}
  { case "five": print("Number is 5.")}
  { pop(); print ("Number is greater than 5.")}
```

Again, the `case` function is used to test the top stack item and the last block serves as a "catch-all".

6.4.6.10 `break, return, continue`

`slang` also includes the non-local transfer functions `return`, `break`, and `continue`. The `return` statement causes control to return to the calling function while the `break` and `continue` statements are used in the context of loop structures. Here is an example:

```
define fun () {
  forever
    {
       s1;
       s2;
       ..
       if (condition_1) break;
       if (condition_2) return;
       if (condition_3) continue;
       ..
       s3;
    }
  s4;
  ..
}
```

Here, a function `fun` has been defined that includes a `forever` loop which consists of statements `s1`, `s2`, ...,`s3` and 3 boolean conditions. As long as `condition_1`, `condition_2`, and `condition_3` return 0, statements `s1`, `s2`, ..., `s3` will be repeatedly executed. However, if `condition_1` returns a non-zero value, the `break` statement will get executed, and control will pass out of the `forever` loop to the statement immediately following the loop which in this case is `s4`. Similarly, if `condition_2` returns a non-zero number, `return` will cause control to pass back to the caller of `fun`. Finally, the `continue` statement will cause control to pass back to the start of the loop, skipping the statement `s3` altogether.

6.4.7. Arrays

Arrays are created using the function call `create_array`. The type of the array and the size of the array are specified by parameters to this function. The calling syntax is:

```
x = create_array (<type>, i_1, i_2 ... i_dim, dim);
```

Here a `dim` dimensional array of type specified by `type` is created. The size of the array in the *nth* dimension is specified by the parameters `i_1` ...`i_n` parameter. The `type` parameter may be any one of the values given in the following table:

```
Parameter Type of array
```

```
___         ____-
  'S'             array of strings
  'F'             array of floats
  'I'             array of integers
  'C'             array of characters
```

Other integer values for the type may be given for applications which defined application specific types to create arrays of those types.

In the current implementation, `dim` cannot be larger than 3. Also note that space is dynamically allocated for the array and that, upon assignment, copies of the array are NEVER used. Rather, references to the array are used by the assignment statements.

For example:

```
variable a = create_array ('F', 10, 20, 1);
variable b = a;
```

This creates a 2 dimensional 10 x 20 array of 200 floats and assigns it to `a`. The second statement makes the variable `b` also refer to the array specified by variable `a`.

Accessing a specific element of the array may be accomplished by placing the "coordinates" of the element in square brackets. For example, to access the (3, 4) element of the above array use `a[3, 4]`. Note that this differs from the way the C language specifies array access and that, like the C language, array subscripts start from 0.

Finally, array notation may also be used for extracting characters from a string. For example, if one has:

```
variable ch,  s = "Hello World";
```

then `ch = s[0]` could be used to extract the first character from the string `s`. However, this syntax cannot be used to replace characters in the string, i.e., `s[0] = ch` is illegal and will generate an error. For the latter case, one must either use the `strsub` function or use a character array.

Examples:

Here is a function that computes the trace (sum of the diagonal elements) of a square 2 dimensional n x n array:

```
define array_trace (a, n) {
  variable sum = 0, i;
  for (i = 0; i < n; i++) sum = sum + a[i, i];
  return sum;
}
```

This fragment creates a 10 x 10 integer array, sets its diagonal elements to 5, and then computes the trace of the array:

```
variable a, j, the_trace;
a = create_array('I', 10, 10, 2);
for (j = 0; j < 10; j++) a[j, j] = 5;
the_trace = array_trace(a, 10);
```

6.4.8. Stack Operators

The use of local variables greatly simplifies the task of maintaining the stack. Nevertheless, **slang** is really a stack based language and there are times when they are useful.

```
pop        % removes the top object from the stack
dup        % duplicates the top object on the stack
exch       % exchanges top 2 objects on the stack
```

These operators work on all data types – they are not limited to integers.

Chapter 7. Using the compiler

7.1. Compiler command line options

The compiler is invoked with the following command:

```
$ flow [option]* [files]*
```

where `[option]` is zero or more options of the list given above, and `[files]` is zero or mode source files.

All the options have a long name that begin with `–`. Certain options have also a short single character name that begin with `-`. The possible options of **`lambda-flow`** compiler are:

`[–algebra | -a] alg` load algebra 'alg' (4). The name could be a complete file name, such as \usr\/local\/lib\/flow\/integer.alg/, or more simply the name `integer` is the path of the file is added in the standard path with the `–path` option.

`–analysis` analysis only the input file in order to check the error.

`[–compile|-c]` compile only the input file.

`–convert` parse the inputs files. Used with the `–group` option, this allows the user to know how **`lambda-flow`** has grouped the expression. Used with the `–short` option, this allows the conversion between the short and the long syntax.

`–errors err` set the max error count to 'err'. If 'err' is 0, there is no error count limit. This is not recommended because if your program is too cyclic, the compiler could display a lot of errors.

`–group` groups expressions in parse result with parenthesizes.

`[–help | -h]` display the **`lambda-flow`** help.

`[–keep | -k]` keep auxiliary files. The auxiliary file are produced by **`lambda-flow`**and given to the post-compiler. With this option, **`lambda-flow`** do not erase the auxiliary file.

`–license` display the **`lambda-flow`** license.

`–linear` forces output code to be linearized.

`–log file` sets the log file to 'file'. If file is 'stderr', the standard error file is used. **`lambda-flow`** could be extremely verbose (see the `–verbose` option). Generally, the displayed messages are used to understand how **`lambda-flow`** work, or to debug it. If you do not understand an error display by the compiler, use the log file.

`[–output | -o] file` set the output file to 'file'. This file is used with the `–compile` option to produce a file named 'file'. It is also used as auxiliary file, with the target extension added (5.4.1), and as output file of the post compiler (5.4.2).

`[–path | -p] path` adds 'path' as a standard program path. **`lambda-flow`** keep a list of path. It uses this list when you try to open a file not in the current path. This list is used for the source files, the algebra definition files and the target code definition files. The other way to add a standard path is to define it with the environment variable `LAMBDA_FLOW_PATH` (several paths could be added this way, if they are separated with `:`).

`–pcos opts` post-compiler options set to 'opts'. This option allow to give some additional options to the post-compiler.

`-ppos opts` preprocessor options set to 'ppos'. If **lambda-flow** is compiled with the preprocessor support, this option allows to give some options to the preprocessor.

`-regExCheck` run the **lambda-flow** compiler as the regular expression interactive test (4.2). Then, the user can checks its regular expressions.

`-regress` regression tests. If **lambda-flow** is compiled with the regression test support, it could be run on the provided regressions test file. A regression file is a **lambda-flow** source file where the name has a special format. The name is composed with `xxx-y.z` where `xxx` could be `syn` (syntactic test), `sem` (semantics test), `com` (compilation test), `cod` (code production test) or `exe` (runtime test), `y` the test number and `z` the number of error found by **lambda-flow** (`x` for an undetermined number of error).

`-regressSet` with this option, **lambda-flow** set the error value to the given regression test file.

`-short` The compiler display the conversion result with the short syntax.

`-stop out` The stopping condition of the source program is set to the output (in the **lambda-flow** meaning) 'out'. The evaluation of the stopping condition is done with the `exit` template (5.4.9).

`[-target | -t] trg` Set the target output code to 'tgr'. The option could be a complete target file name such as `\usr\local\lib\flow\c.trg/` or simply the name of the target such as `c` is the corresponding target file path is added with the `-path` option.

`[-verbiage | -v] n` verbose operation level is set to 'n'. n is a logical or of the value given in the VERBIAGE file in the root directory of the distribution.

`-width n` set the indent width to 'n' for the comments that the compiler generate.

7.2. Initialization file

In order to avoid long command line, **lambda-flow** support some options to be put in an initialization file. The name of this file is `flow.ini`/.

The compiler first tries to load this file in the current working directory, then in the user home directory, and in the library installation directory, and at least in the directory where the compiler executable is installed.

This file has the following form:

```
[flow]
errors          = 50
path            = /usr/local/lib/flow
algebra         = integer boolean
target          = c
libraries       = default.lf
default         = debug.lf
verbiage        = 65535
preprocessor    = cpp -P %s %s -o %s
logfile         = flow.log
```

where:

`errors` set the maximal error count ;

`path` adds some blank separated paths as standard paths ;

`algebra` loads the blank separated algebra

`target` loads the target code definition file;

`libraries` load always the file specified in this list;

`default` if no source files are specified on the command line, loads this default file;

`verbiage` set the verbiage level ;

`preprocessor` set the preprocessor ;

`log file` set the log file.

7.3. Compile a single file program

The simplest form of programming in **`lambda-flow`** is to keep the sources in a single file.

In order to invoke the compiler on the file that contain our example (1.2), just type:

```
$ flow example.lf
```

The default option are set in the `flow.ini` file (7.2). By default, the C target code is chosen. You can see in your working directory a file `a.out` produced by the post-compiler of the C TCDF, which is set to `gcc` (5.6.1).

If you want to see the C auxiliary file produced by **`lambda-flow`**, use the `-k` command line option, that avoid the auxiliary file to be deleted. Just type:

```
$ flow example.lf -k
```

Notice the place of the option is not important with the compiler. This command produce a file `a.c`. If you want to have another target code, simply type:

```
$ flow -t scheme example.lf
```

which produce a file `a.scm` which is a Scheme source file. If you want to specify your own algebra, you could use:

```
$ flow -a /path/my-algebra.alg example.lf
```

7.4. Invoking the preprocessor

Before processes the source file, **`lambda-flow`** could use a preprocessor. You can specify the preprocessor in the `file.ini` file. Notice that the **`lambda-flow`** parser automatically detects the `#line` directive produced by the C preprocessor.

You could pass some options to the preprocessor with a command line option. Simply type:

```
$ flow -ppos "-I. -Dlow_noise" example.lf
```

and the preprocessor is invoked with the option `-I. -Dlow_noise`. Notice the use of `“”` around the options.

So you can include some header files with the `#include` directive of the C preprocessor.

7.5. Compile a multi-files program

`lambda-flow` is able to compile a source file separately. Due to the nature of the language, it is impossible to produce target code for partial source file. **`lambda-flow`** produce a **`lambda-flow`** flattened program, with the same interface.

If your project have `f_1.lf`, `f_2.lf`, ..., `f_n.lf` as source files, you can invoke the compiler for each source file with:

```
$ flow -c f_i.lf -o f_i.lo
```

where `-c` is compile-only option, and where `f_i.lo` is the produced file. Then, you can link all these file together with:

```
$ flow f_1.lo f_2.lo ... f_n.lo -o a.out
```

which produces the executable (according to the target option) `a.out`.

Chapter 8. Getting, compiling and installing `lambda-flow`

This chapter contains:

- Installation instructions and notes for lambda-flow
- Where to get more information on lambda-flow
- Common problems
- Information on porting the program
- Obtaining the missing pieces of lambda-flow

8.1. Installation instructions for lambda-flow

The `configure` shell script attempts to guess correct values for various system-dependent variables used during compilation, and creates the Makefile. It also creates a file `config.status` that you can run in the future to recreate the current configuration.

To compile this package:

8.1.1. Configuring

Normally, you just `cd` to the directory containing the package's source code and type `.`configure/. If you're using `csh` on an old version of System V, you might need to type `sh configure` instead to prevent `csh` from trying to execute `configure` itself (under AIX, you may need to use ksh instead of sh).

Running `configure` takes a while. While it is running, it prints some messages that tell what it is doing. If you don't want to see any messages, run `configure` with its standard output redirected to \dev/null/; for example, `.` \configure \>\/dev\/null/.

To compile the package in a different directory from the one containing the source code, you must use a version of `make` that supports the `VPATH` variable, such as GNU `make`. `cd` to the directory where you want the object files and executables to go and run the `configure` script. `configure` automatically checks for the source code in the directory that `configure` is in and in `..` If for some reason `configure` is not in the source code directory that you are configuring, then it will report that it can't find the source code. In that case, run 'configure/ with the option `–srcdir=DIR`, where DIR is the directory that contains the source code.

By default, `make install` will install the package's files in usr/local/bin/, usr/local/man/, etc. You can specify an installation prefix other than usr/local/ by giving `configure` the option `–prefix=PATH`. Alternately, you can do so by consistently giving a value for the `prefix` variable when you run `make`, e.g.,

```
$ make prefix=/usr/gnu
$ make prefix=/usr/gnu install
```

You need to have the package regex installed. You can specify the path of regex.o with the configure option `–with-regex-o=fullpath`filename/.

`lambda-flow` is recommended to be compiled with the slang package. You can specify the slang library path with the configure option `–with-slang-lib=filename`. `filename` will be added to the linker option as `-lfilename`. If the path is not the standard, use the configure option `–libdir=fullpath`.

`lambda-flow` can be compiled with the garbage collector from **Hans-J. Boehm and Alan J. Demers**. You can use the configure option `–with-gc-lib=filename`. `filename` will be added to the linker options as `-lfilename`. If the path is not the standard, use the configure option `–libdir=fullpath`. If you cannot get a proper GC library, the configure option `–without-gc` could be used. In this case, $lambdaFlow will `malloc()` memory allocator without any `free()` call. In fact, this is not a real problem because **`lambda-flow`** is designed to use carefully the memory. The DOS version does not have a garbage collector.

`configure` also recognizes the following options:

`elp` Print a summary of the options to `configure`, and exit.

`–quiet`

`–silent` Do not print messages saying which checks are being made.

`–verbose` Print the results of the checks.

`–version` Print the version of Autoconf used to generate the `configure` script, and exit.

`–includedir=PATH` PATH is an additional path for the c header files

`–libdir=PATH` PATH is an additional path for the c library files

`–with-cc=CC` uses another c compiler. This option is used to obtain a DOS binary with `–with-cc=gcc-dos`. The default c compiler is detected by `configure`.

`–without-gc` do not use the garbage collector GC. By default, GC is used.

`–with-gc-lib=LIB` use LIB as gc lib. If the GC lib is libgc-linux.a, the LIB option is gc-linux. The path of the library should specified with `–libdir`. The path of the gc.h header file should be specified with `–includedir`

`–without-slang` do not use the slang library. By default, slang is used.

`–with-slang-lib=LIB` specifies the slang library. If the slang library is libslang-linux.a, the LIB option should be slang-linux. The path of the library should specified with `–libdir`. The path of the gc.h header file should be specified with `–includedir`. The default slang library is `libslang.a`.

`–with-regex-o=FILE` FILE is the regex.o package. FILE must be the full path name. The default file is `regex.o`

`–enable-regression` enable regressions tests. The regression test will be not useful for the normal user. By default, regressions tests are not set.

`–without-preprocessor` do not use any preprocessor. Lambda-flow can process the source file with a specified preprocessor (the default preprocessor is cpp). This option disable the preprocessor usage. The default preprocessor is `“cpp -P %s %s -o %s”`.

`–with-preprocessor_cmd=CMD` preprocessor command. The command must have three %s, the first one will be replaced with the preprocessor options, the second by the source file, the last with the target file. For example, CMD could be `“cpp %s %s > %s”`. Don’t forget the “ that avoid the shell interpretation.”

`configure` also accepts and ignores some other options.

On systems that require unusual options for compilation or linking that the package’s `configure` script does not know about, you can give `configure` initial values for variables by setting them in the environment. In Bourne-compatible shells, you can do that on the command line like this:

```
CC='gcc -traditional' LIBS=-lposix ./configure
```

On systems that have the `env` program, you can do it like this:

```
env CC='gcc -traditional' LIBS=-lposix ./configure
```

Here are the `make` variables that you might want to override with environment variables when running `configure`.

For these variables, any value given in the environment overrides the value that `configure` would choose:

`Variable: CC` C compiler program. The default is `cc`.

`Variable: CFLAGS` The default flags used to build the program.

`Variable: INSTALL` Program to use to install files. The default is `install` if you have it, `cp` otherwise.

For these variables, any value given in the environment is added to the value that `configure` chooses:

`Variable: LIBS` Libraries to link with, in the form `-lfoo -lbar...`.

If you need to do unusual things to compile the package, we encourage you to figure out how `configure` could check whether to do them, and mail diffs or instructions to the address given in the README so we can include them in the next release.

8.1.2. building

Type `make` to compile the package.

8.1.3. Regression tests

If the package is configured with the `–enable-regression` option and you want to run the tests, type `make regression`.

The regression test are run with the following command:

```
$ flow –regress Regress/*.*
```

And the result should be like this :

```
 /flow/Regress>../src/flow –regress *.*
flow, 0.3 (Thu Jul 11 16:43:58 1996)-(c) Guilhem de Wailly - 1995-1996
        sem-001.0       – done
        sem-002.1       – done
        sem-003.1       – done
        sem-004.0       – done
        sem-005.2       – done
```

without any error message.

8.1.4. Installing

Type `make install` to install programs, data files, and documentation.

8.1.5. Cleaning

You can remove the program binaries and object files from the source directory by typing `make clean`. To also remove the Makefile(s), the header file containing system-dependent definitions (if the package uses one), and `config.status` (all the files that `configure` created), type `make realclean`. If you want to clean the source tree completely, so that it contains only those files that should be packaged in the archive, issue `make distclean`. If you've run configure in a different directory than the source tree, distclean won't remove your *.o and linked programs in that directory. If you want to desinstall the program, type `make uninstall`.

The file `configure.in` is used to create `configure` by a program called `autoconf`. You only need it if you want to regenerate `configure` using a newer version of `autoconf`.

8.2. Where to get more information on lambda-flow

You can see at the provided documentation issued in three format, postscript, man and html. Some publication pointers are provided here.

You can send an e-mail to `gdw@unice.fr`.

You can read the WWW page at

`http://alto.unice.fr: gdw/pub/lambda-flow`

8.3. Notes about lambda-flow

lambda-flow has been run in the following configurations:

- i386-linux-linux-1.2.13
- i386-msdos-msdos-6.2
- sparc-sun-sunos-4.1
- sparc-sun-solaris-2.3

It is a preliminary beta test version. If you have an error, please change the verbiage option to 65535 and the log file to `flow.log`. Then recopile yout source file and send to us the source file, the `flow.log`.

Since **lambda-flow** is configured via the GNU `autoconf` program, it's not difficult to run it in other operating systems.

Some configure command line:

```
i386-linux-linux1.0    configure –enable-regression
                                 –includedir=/usr/local/include
                                 –libdir=/usr/local/lib
                                 –with-gc

sparc-sun-sunos4.1    configure –enable-regression
                               –includedir=$HOME/usr/include
                               –libdir=$HOME/usr/lib
                               –with-gc
                               –with-slang-lib=slang-sun

i386-msdos-msdos6.2    configure –with-cc=gcc-dos
                                 –enable-regression
                                 –includedir=c:\usr\include
                                 –libdir=c:\usr\lib
                                 –without-gc
                                 –with-slang-lib=slang-dos
                                 –with-regex-o=regex-dos.o
```

8.4. Porting the program

The main difficulty to port **lambda-flow** is to port the GC garbage collector.

The own part of **lambda-flow** is written with one GNU facility, the possibility to define some function inside a function. This allows to share the function parameters in an easy way. To port this on some traditional c compiler, it is needed to rewrite these inner function on the top-level, and either add to them some parameters or add some global variable.

8.5. Obtaining the missing pieces of lambda-flow

Lambda-flow will build without requiring you to get any other software packages, however, you may be interested in enhancing lambda-flow environment with some of these:

```
LAMBDA-FLOW    author : Guilhem de Wailly
               site   : ftp://alto.unice.fr/ gdw/flow/flow-src-0.2.tgz
                        ftp://alto.unice.fr/ gdw/flow/flow-bin-0.2.tgz
```

`GC` This implements garbage collection of chunks of memory obtained through (its replacement of) malloc(3). It works for C, C++, Objective-C, etc.

```
author : Hans J\"urgen Boehm and Mark Weiser
site   : ftp://parcftp.xerox.com:/pub/gc/gc4.3.tar.gz
```

`SLANG` **slang** (pronounced `sssslang`) is a powerful stack based interpreter that supports a C-like syntax. It

has been designed from the beginning to be easily embedded into a program to make it extensible. slang also provides a way to quickly develop and debug the application embedding it in a safe and efficient manner. Since slang resembles C, it is easy to recode **slang** procedures in C if the need arises.

```
author : John E. Davis
site   : ftp://space.mit.edu/pub/davis/slang
```

```
Regex provides three groups of functions with which you can operate on regular
    expressions. One group—the GNU group—is more powerful but not completely
    compatible with the other two, namely the POSIX and Berkeley UNIX groups; its
    interface was designed specifically for GNU. The other groups have the same
    interfaces as do the regular expression functions in POSIX and Berkeley UNIX.
    authors : Richard Stallman, Karl Berry,
              Kathryn Hargreaves, Jim Blandy,
              Joe Arceneaux, David MacKenzie,
              Mike Haertel, Charles Hannum
    site    : ftp://prep.ai.mit
              ftp://sunsite.unc.edu
```

And the GNU C Compiler may be obtained from the following sites:

```
ASIA:   ftp.cs.titech.ac.jp, utsun.s.u-tokyo.ac.jp:/ftpsync/prep,
        cair.kaist.ac.kr:/pub/gnu
AUSTRALIA: archie.au:/gnu (archie.oz or archie.oz.au for ACSnet)
AFRICA: ftp.sun.ac.za:/pub/gnu
MIDDLE-EAST: ftp.technion.ac.il:/pub/unsupported/gnu
EUROPE: ftp.cvut.cz:/pub/gnu, irisa.irisa.fr:/pub/gnu,
        ftp.univ-lyon1.fr:pub/gnu, ftp.mcc.ac.uk,
        unix.hensa.ac.uk:/pub/uunet/systems/gnu,
        src.doc.ic.ac.uk:/gnu, ftp.win.tue.nl, ugle.unit.no,
        ftp.denet.dk, ftp.informatik.rwth-aachen.de:/pub/gnu,
        ftp.informatik.tu-muenchen.de, ftp.eunet.ch,
        nic.switch.ch:/mirror/gnu, nic.funet.fi:/pub/gnu, isy.liu.se,
        ftp.stacken.kth.se, ftp.luth.se:/pub/unix/gnu, archive.eu.net
CANADA: ftp.cs.ubc.ca:/mirror2/gnu
USA:    wuarchive.wustl.edu:/mirrors/gnu, labrea.stanford.edu,
        ftp.kpc.com:/pub/mirror/gnu, ftp.cs.widener.edu, uxc.cso.uiuc.edu,
        col.hp.com:/mirrors/gnu, ftp.cs.columbia.edu:/archives/gnu/prep,
        gatekeeper.dec.com:/pub/GNU, ftp.uu.net:/systems/gnu
```

8.6. Copyright

WAILLY HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Index