Noisy Draft Language Reference, version 0.1

Phillip Stanley-Marbell
psm@mit.edu

MIT CSAIL,
Cambridge, MA 02139.

Hardware platforms such as Warp require appropriate programming abstractions. Noisy is a language designed for programming sensor-driven systems like Warp. This document describes the design of the language and provides examples of its use.

DRAFT April 1, 2016 - 15:38

Contents

1. INTRODUCTION

1.1 Overview

Noisy is a language for programming platforms with constrained memory and computation resources, such as the Warp platform, and platforms such at that illustrated in Figure 1. The language, its runtime system and the compiler, are designed to support making the communication between components of a program explicit. The language design is intended to facilitate three goals:

(1) *Language constructs for specifying precision, accuracy, reliability, and latency tolerance constraints, and constraint violation control flow.* This enables specification of the distribution of numeric errors (deviations) tolerable in individual program variables, and the change of control flow when these constraints are violated.

(2) *Program transformations that tradeoff performance and reliability* of applications, in the presence of errors and erasures in the underlying hardware, under numeric error tolerance constraints.

1.2 Organization of applications

Programs in Noisy are organized into units called *name generators (namegens)* (Figure 2). Name generators are collections of program statements (e.g., like functions or procedures in Algol family languages) which exchange information by explicit communication rather than transfer of control flow. They interact by communicating on *names* by which each name generator is represented in a runtime *name space*. Namegens and names are analogous to *Actors* and their *mail addresses* in the Actor system [1]. All interactions on names are expressed in terms of a small alphabet of operations, forming the *name interaction protocol*. Names abstract entries in the name space, which abstract channels, in much the same way that pointers

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

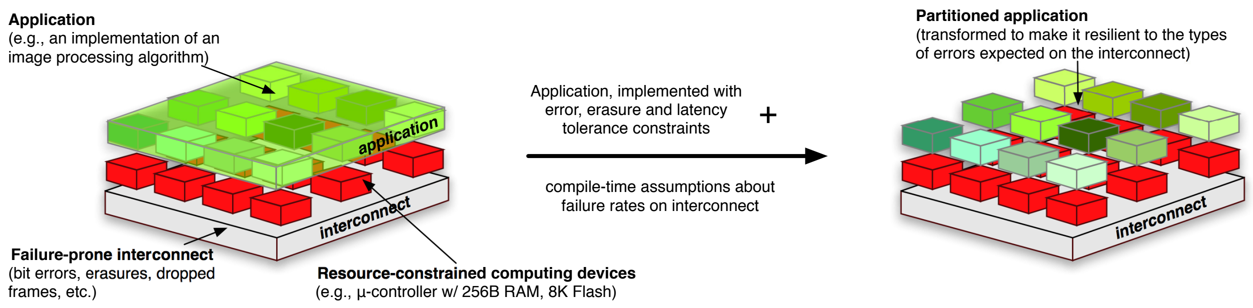
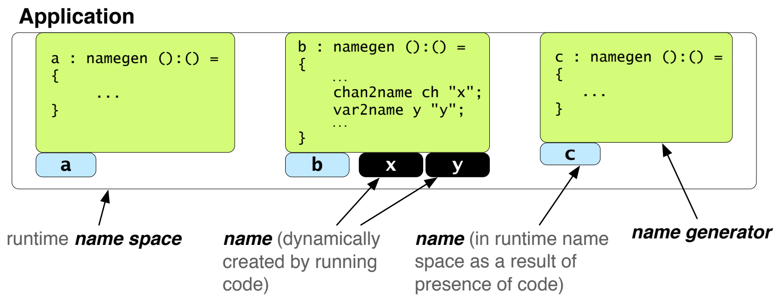

Application
(e.g., an implementation of an image processing algorithm)
application
interconnect
Failure-prone interconnect
(bit errors, erasures, dropped frames, etc.)
Resource-constrained computing devices
(e.g., µ-controller w/ 256B RAM, 8K Flash)
Application, implemented with error, erasure and latency tolerance constraints
+
compile-time assumptions about failure rates on interconnect
Partitioned application
(transformed to make it resilient to the types of errors expected on the interconnect)
interconnect

Fig. 1. Pictorial representation of motivating ideas in Noisy.

abstract memory addresses, which abstract memory cells. Decoupling the interactions between namegens through entries in the runtime name space, facilitates the easy migration of endpoints of communications. This is beneficial to application re-mapping in the presence of detected failures, for a programmable substrate with redundantly available resources.


Application
a
b
x
y
c
runtime name space
name (dynamically created by running code)
name (in runtime name space as a result of presence of code)
name generator

```
a : namegen ():() =
{
    ...
}
```

```
b : namegen ():() =
{
    ...
    chan2name ch "x";
    var2name y "y";
    ...
}
```

```
c : namegen ():() =
{
    ...
}
```

Fig. 2. Organization of applications into *namegens* which interact via *names* in a *runtime name space*, using a simple *name protocol*.

1.3 Error magnitude, loss (erasure) and latency tolerances

At the underlying hardware layer, there may be bit *errors* and *erasures*, which lead to erroneous changes in values. In what follows, the difference between the correct and erroneous values is termed the *error magnitude*. An error magnitude tolerance specifies the acceptable distribution on the error magnitude values. For example, a plain English specification of a simple error tolerance constraint could be "Ensure that the error magnitude only exceeds 2.0 at most one out of every million times that a value is read". If the error magnitude is denoted by a random variable M, with a specific instance m, the above statement could be written more precisely as

$$\Pr\{M > 2.0\} \quad \leq \quad \frac{1}{1,000,000}$$

If an application is partitioned for execution over a network of computing devices, its constituent components must communicate over a possibly failure-prone interconnect network, as illustrated in Figure 1. Values thus communicated could potentially be lost (e.g., the entire sent data packet is never received). Such loss will be referred to as *channel-level erasures*. A channel erasure tolerance constraint thus specifies the acceptable distribution on the *loss fraction* of values. Lastly, operations on channels, can have non-deterministic latencies, since they depend on the state of the underlying communication interconnect. Specifying *latency tolerance constraints* enable a program to specify how much latency on a communication is acceptable, and to take action when a constraint on latency is violated.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

These program-level annotations of tolerance constraints, together with empirically measured distributions of variable values and analytically derived expressions for the error-magnitude, enable the minimal message-level redundancy (encoding) to be determined [13].

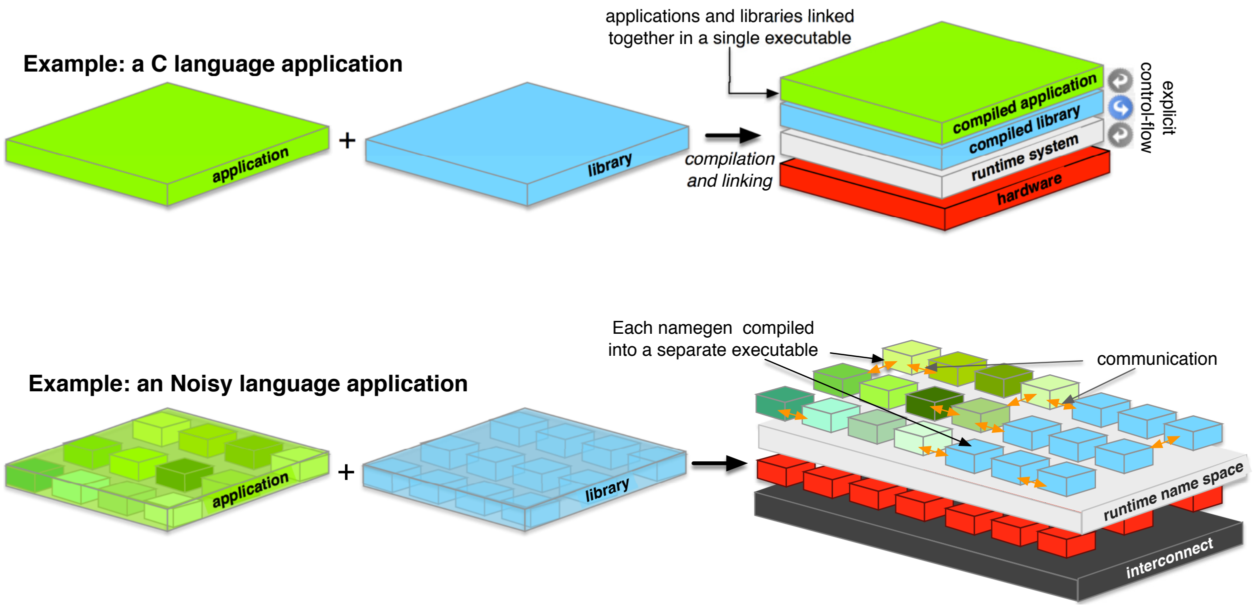

Example: a C language application
application
+
library
compilation and linking
applications and libraries linked together in a single executable
compiled application
compiled library
runtime system
hardware
explicit control-flow
Example: an Noisy language application
application
+
library
Each namegen compiled into a separate executable
communication
runtime name space
interconnect

Fig. 3. Illustration of the organization of applications into “name generators”.

The compiler takes as input program texts which implement *applications*, and produces a partitioned set of executables for execution over multiple hardware devices; together, these individual executables implement the application. The process is illustrated in Figure 3. The partitioning of programs is not in an attempt to *parallelize* them to achieve better performance, but rather, to facilitate the execution of programs larger than the memory resources of the individual computing devices in the system[1]. The ease of partitioning both at the level of namegens and within namegens, is facilitated by the *projection* of variables and channels in programs into the name space, which permits the entire interface to any sequence of program statements to be represented by entries in the runtime name space.

2. TERMINOLOGY AND DEFINITIONS

Declaration. The ascription of a type to an identifier without an associated initialization value is termed a *declaration*.

Definition. The assignment of a value along with type ascription is termed a *definition*.

3. LEXICAL ELEMENTS

Programs are sequences of Unicode [15] characters, organized according to the rules of the language grammar. Tokens are separated by whitespace, operators or other separators. Whitespace consists of `'\␣'`, `'\n'`, `'\r'` and `'\t'`. Separators and operators may abut one or more tokens. In addition to these separators and

[1]The target hardware platforms are systems composed of possibly hundreds of computing devices (e.g., microcontrollers) with total memory resources of the order of a few hundred bytes, in which the devices are intended to be unobtrusively embedded in some substrate or environment. For such systems, advances in semiconductor technology will be harnessed to make the devices *smaller*, and their memory resource constraints are not likely to be elided. The Sunflower hardware platform [11] uses 16-bit microcontrollers with 256 bytes of RAM and 8KB of Flash memory.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

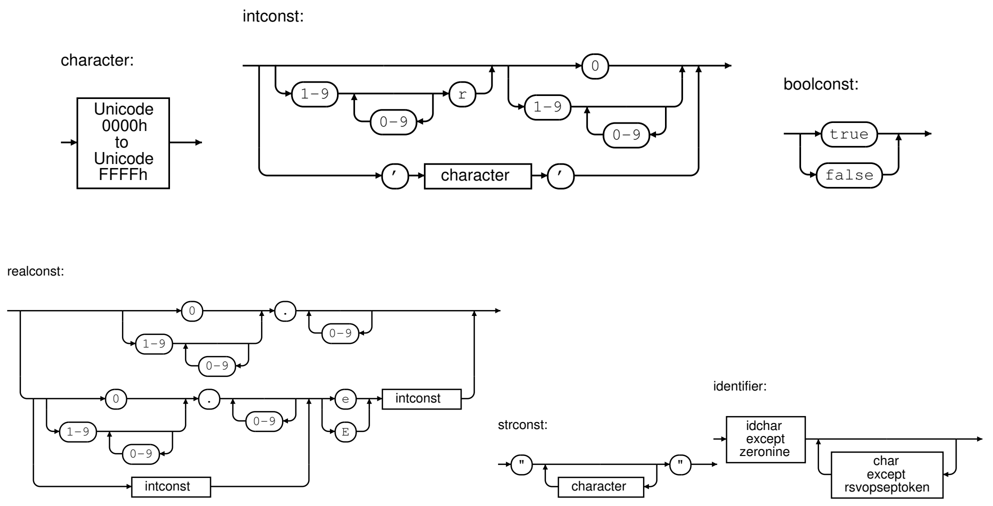


Fig. 4. Syntax diagrams for lexical elements.

operators, several tokens are reserved for use in the language and have special meaning; they may not be used as identifiers. Comments are introduced with the character #, and continue until the next newline, or the end of input:

```
#       This is a comment
```

3.1 Reserved tokens

A set of single and multiple character tokens are lexically reserved and may not be used in identifiers. The reserved operators and separators are:

~	!	%	^	&	*	(	)	-	+
=	/	>	<	;	:	'	"	{	}
[	]	|	<-	.	,	<=	>=	^=	|=
&=	%=	/=	*=	-=	+=	:=	!=	>>	>>=
<<	<<=	&&	||	::	==	--	++	<-=	=>
->	#								

The reserved identifiers in the language are:

adt	alpha	bool	byte	chan
chan2name	const	epsilon	erasures	errors
false	fixed	hd	int	iter
latency	len	list	match	matchseq
name2chan	namegen	nybble	nil	of
progtype	real	set	string	tau
tl	true	type	var2name	

The production rules employed in lexical analysis for determining the terminal symbols from sequences of Unicode characters, can be described by a *regular language* [2], whose productions are shown by the syntax diagrams in Figure 4.

4. SYNTACTIC ELEMENTS

A sequence of Unicode characters, which corresponds to a sequence of one or more tokens as described in the preceding section, is a valid program if it conforms to the rules of the language grammar. A complete language grammar in

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

EBNF [16] notation is provided in Appendix D.

4.1 Programs

A Noisy program is composed of a single program interface definition (*progtype definition*) and a collection of *name generators (namegens)*. The progtype definition specifies a set of constants, type declarations, and publicly visible namegens. All entries in the progtype definition become visible in the runtime names space.

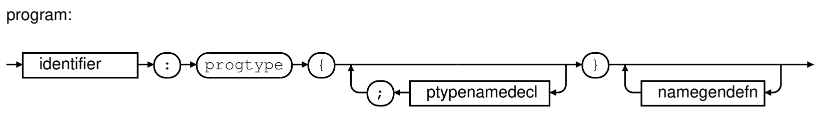


Fig. 5. Syntax diagram for overall program structure.

Example:

```
Hello : progtype
{
        init : namegen();
};
```

4.2 Types, type expressions, and type declarations

Noisy is statically checked and strongly typed. Every expression can be assigned a unique type, and the compiler will not accept as valid any program for which a type error can occur at runtime. The syntax of the basic and derived types are represented by the syntax diagrams in Figure 6. The basic data types are `bool`, `nybble`, `byte`, `string`, `int`, `real`, and `fixed`. A basic type may have associated with it one or more *error*, *loss* (i.e., erasure), or *latency* tolerance constraints. Variables of type `bool` are 1-bit values, `nybble` are 4-bits, `int` are 32-bit signed integers in two's complement format. Type `real` is a 64-bit double-precision floating point value in IEEE-754 format. The type `fixed` is a 16-bit fixed-point representation for real values.

The type collections are arrays, abstract data types (ADTs), lists, tuples and sets. Arrays are vectors of elements of a single type. ADTs and tuples are collections of elements of possibly different types; whereas an ADT collection has an associated *type name*, tuples are unnamed collections. Lists are collections of elements of a single type, with access to only the head of the list. Sets are unordered collections of elements, with primitive operations for union, intersection, relative complement and cardinality, with which idioms for other set operations (e.g., membership, subset) can be implemented[2]. The language types are discussed in more detail in Section 6, and a formal description of the type system is provided in Section B.

The types `bool` and `nybble` are motivated by the fact that there are many classes of applications in which single bit values are semantically popular, e.g., for use as flags, as are variables taking on a small range of values [10]. In memory constrained devices, it makes sense to provide language constructs that support these idioms. Particularly for small cardinality, sets provide a means for memory and computationally efficient implementation of collections. Example:

```
#       Variable 'pixel' is an integer with constraint that the
#       probability that the magnitude of error in its value being > 1
#       is less than 0.01

pixel : int, epsilon(1, 0.01);
```

[2]Wirth [17] attributes the idea of sets as a language data type to Hoare, circa 1972.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT

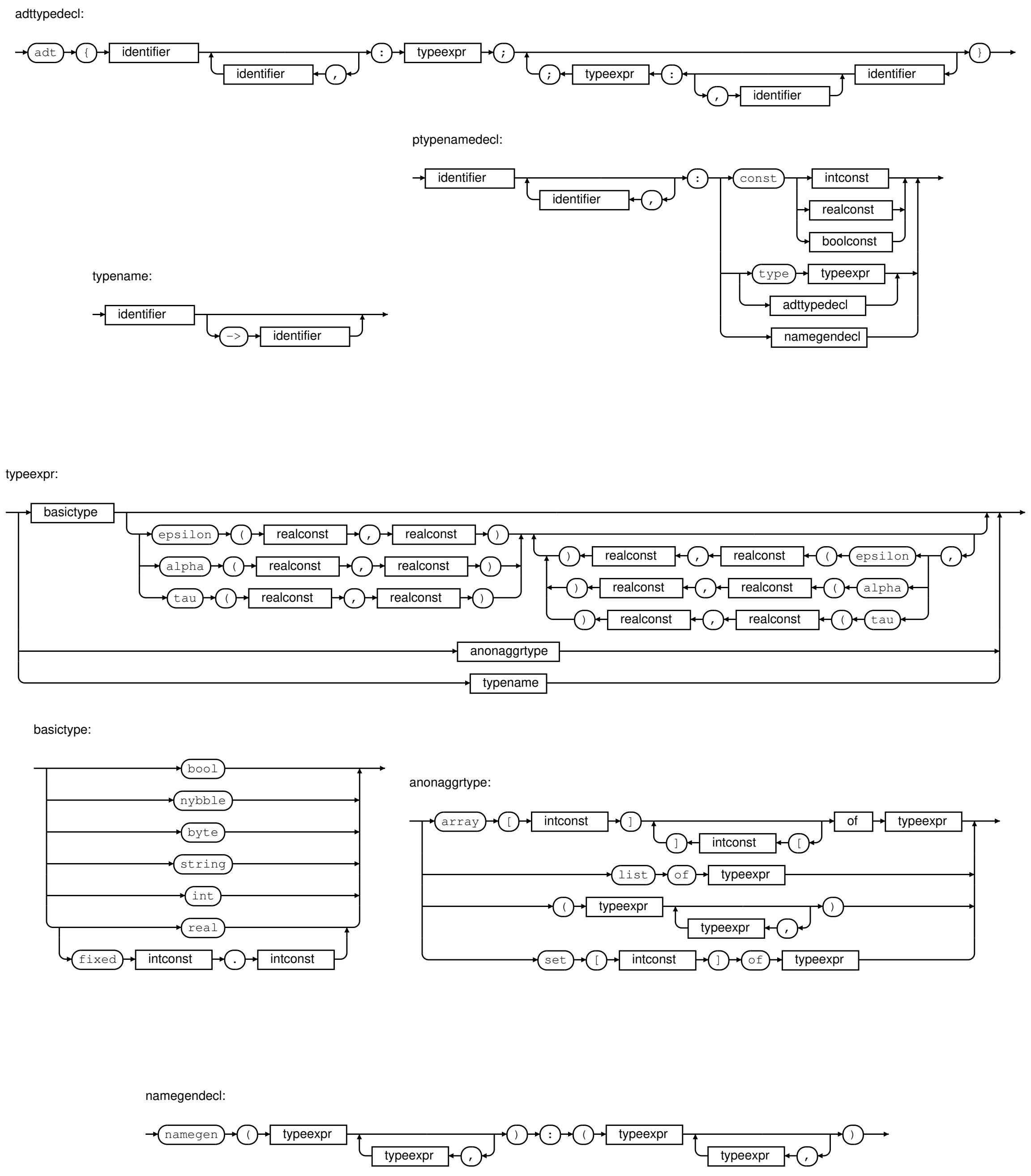


Fig. 6. Syntax diagrams for types.

4.3 Variable and Channel Identifiers

There are two kinds of program-defined identifiers in Noisy: *variables* and *channels*. Variables are identifiers that are used to refer to possibly structured data in memory. The simplest type of variable is a *pronumeral* that represents an item in memory with one of the basic arithmetic (numeric) types. Channels on the other hand are identifiers that are used to refer to communication paths between namegens. Channels as a programming language construct trace their roots to Hoare's *Communicating Sequential Processes (CSP)* [7]. All channels in Noisy are tied to a *name* in

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

the runtime name space.

While variables can be used in any expression (Section 4.9), channels can only be used in *send* and *receive* expressions. In a send expression, a value is written to a channel, while a values is read from a channel in a receive expression. The evaluation of a send expression does not complete until *another* namegen performs a receive operation on a channel that is associated with the same name in the runtime name space as the channel being written to. Likewise, evaluation of a receive expression will not complete until a send expression on a channel associated with the same name in the runtime name space is executed by another namegen.

Variables and channels must be defined before use, ascribing to the variable or channel a basic or collection type. A variable may also be declared in conjunction with assignment, in which case it is ascribed the type of the value it is being set to. Example:

```
#       Variable 'sensor' is a channel of real valued numbers,
#       with the constraint that the probability that the number
#       of lost values on the channel being > 1 is 0.1, and > 10 is 0.0003

sensor : chan of int, tau(1, 0.1), tau(10, 0.003);


#       Declare variable 'a' with type int
a       : int;

#       Declare variable 'b' with tuple type (int, int)
b       : (int, int);

#       Define variable 'c' with type being the type of its assigned value (real)
c       := 1.0;
```

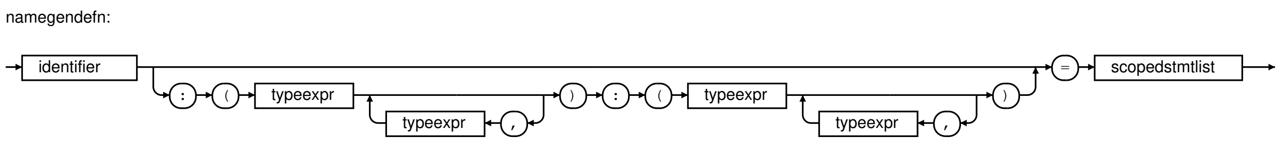


Fig. 7. Syntax diagram for name generator definition.

4.4 Channel declarations versus channel definitions

Channels are used to communicate between two namegens. Before a channel can be used, it must be associated with a name in the runtime name space via a `name2chan` expression. A channel declaration only serves to ascribe a type to the identifier that will eventually refer to a name in the runtime name space. A channel `definition` is the association of a name in the runtime to a channel identifier, i.e., assigning the result of a `name2chan` to an identifier — prior to a definition, channel identifiers have the value `nil`.

4.5 Structuring programs

Programs are composed of *program types (progtypes)* and their implementations, a collection of *name generators (namegens)*. A progtype is a unit of modularity that defines a set of types, constants, and name generators. The unit of compilation of programs is a single progtype and its implementation. This single complete program input to the compiler is used to generate one or more compiled outputs, corresponding to the pieces of the partitioned application. Partitioning occurs most easily at the level of individual namegens. Due to the structure of the language and its runtime system, it is however possible to further partition a single namegen into smaller pieces.

A *name generator* or *namegen* is a collection of type declarations, variable definitions and statements. It is the unit at which applications get partitioned into executable units. There is no shared state between namegens. Namegens are not functions: i.e., control flow does not pass between namegens. A namegen may bind a channel variable to a

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT

name. Syntactically, names are represented by strings. Such strings are part of a runtime *name space*[3]. The syntax diagrams for the grammar production for a valid namegen definition is shown in Figure 7.

All interaction between namegens is explicitly through *names* in the runtime name space, via the binding of names to channels. A namegen definition includes an interface tuple type. This is the type structure of the name for write and read operations on the name. Example:

```
Math : progtype
{
	sqrt	: namegen (real) : (real);
	exp	: namegen (real, real) : (real);
}

#	The actual implementation of the Math progtype must have
#	definitions for the namegen types sqrt and exp:
sqrt =
{
}

exp =
{
}

#	A namegen that does not appear in the interface is declared
#	and implemented in one step here
somefunc : (real, real) : (real) =
{
	#	Declaration of 'sqrt' in progtype introduced a new type name.
	#	We use that as the type of the variable bound to the namegen sqrt:
	s := name2chan sqrt "sqrt";
}
```

The definition of a namegen in a progtype introduces a new type name made up of a *write type* and a *read type*, that can be used in a name2chan expression. If a namegen is not defined in its progtype declaration, it is not visible within the name space once the progtype implementation is loaded. However, the definition of the namegen within the body of the program introduces a new type, as it would if it had been declared in the progtype. Within the body of a namegen, the namegen's name is a channel which when read from has the namegens *read type*, and which for the purposes of writes, has the namegen's *write type*. Applying the `name2chan` operator to a name that has type `namegen` causes the instantiation of a new thread of control / instance of the namegen. Example:

```

#	Defines a namegen mul.  An antry in the name space "mul"
#	with write type (int, int) and read type (int) will exist
#	at runtime

mul : (int, int) : (int) =
{
}
```

4.6 Scopes

The largest scope is the body of a namegen; there is no global scope. Within a namegen, a new scope is embodied by a collection of statements enclosed in braces (`{ ... }`). Variables defined within a scope have visibility only within the given scope. The following grammar productions introduce new scopes:

— `progtypebody` production

[3]By analogy, a computing system fashioned on the von Neumann model has the concept of *memory addresses* which are represented by integers, and are part of an *address space*.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

— body of ADT type declaration
— `scopedstmtlist` production
— `guardedscopelist` production

The complete language grammar is provided in Appendix D.

4.7 Statements

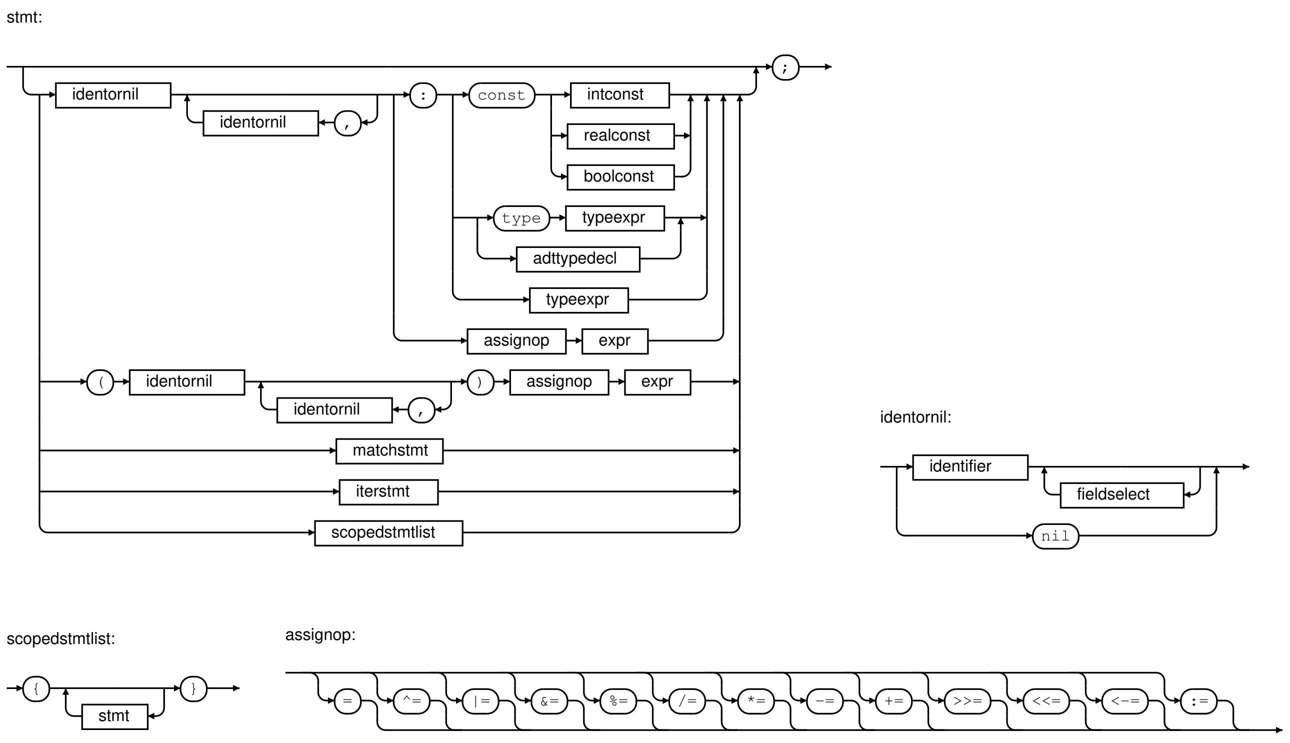


Fig. 8. Syntax diagrams for statements

The statements which make up the body of a namegen define how it computes, creates names in the runtime name space, and communicates with other namegens. The syntax diagrams for grammar productions corresponding to valid statements is shown in Figure 8.

4.8 Assignment statements

The *r-value* in an assignment must be of the same type as the *l-value*, with the exception of an *l-value* of `nil`, which can be assigned an expression of any type; the *r-value* in such an assignment to `nil` is first evaluated, but the assignment operation does not yield any further action. The basic assignment operator is `=`. The additional assignment operators which have the form *binop*`=` assign to the *l-value* the result of *l-value binop r-value*. The syntax diagrams for the associated grammar productions are illustrated in Figure 8. Example:

```

#       Bit-wise negation of variable 'a'
a ^= 1;
```

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

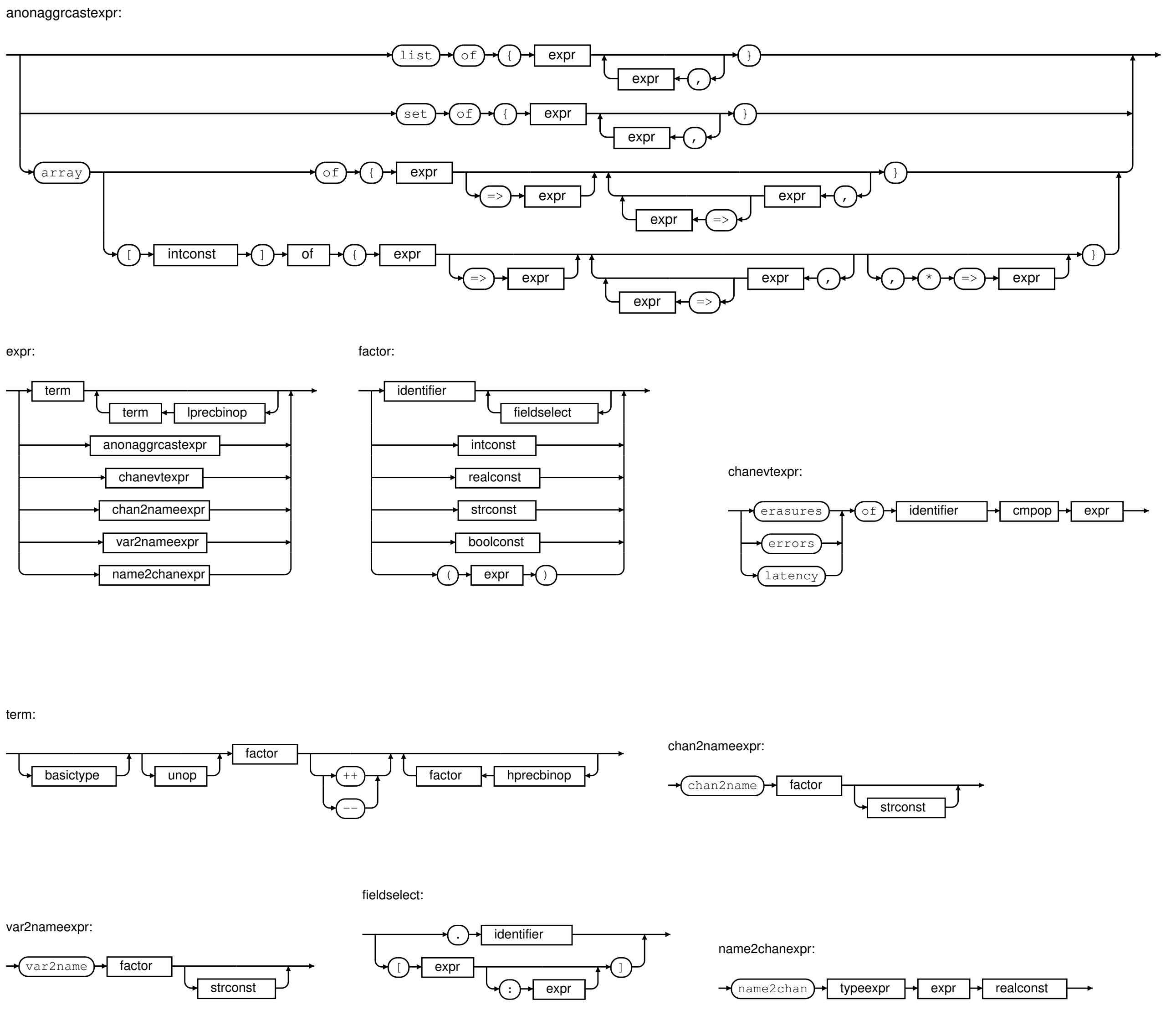


Fig. 9. Syntax diagrams for expressions

4.9 Expressions

The associativity of operators is implicit in the definition of the grammar for expressions: *expressions* are made up of *terms* and the low precedence binary operators; terms are made up of *factors* and the high precedence binary operators; factors are *l-values*, constants, expressions in parenthesis or unary operators followed by a factor. The unary operators thus have the highest precedence, followed by high precedence binary operators and then low precedence binary operators. This is illustrated by the grammar productions in Figure 9. Operators are discussed in more detail in Section 5.

4.10 The `match` and `matchseq` constructs

The `match` statement is a collection of guarded statement blocks. It executes *all* constituent statement blocks whose guards, which are Boolean expressions, evaluate to true. If multiple guards evaluate to true, the order in which the guarded statement blocks are evaluated is non-deterministic. The `match` statement is analogous to *guarded selection* in

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

Dijkstra's guarded commands [5]. The `matchseq` statement on the other hand evaluates its guards sequentially, until a guard that evaluates to true. The guarded statements are then executed, and the `matchseq` statement completes. The syntax of `match` and `matchseq` statements are illustrated by the syntax diagrams in Figure 10.

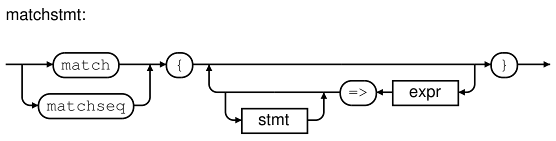


Fig. 10. Syntax diagrams for match statements

The `match` statement can be used to implement the equivalent of `if` statements (as in the Pascal and C family of languages), and multi-way selection statements such as Pascal `case` and C `switch` statements. Example:

```

#       If 'a' > 'b', 'max' = 'a',
matchseq
{
        a > b   =>      max = a;
        true    =>      max = b;
}
```

One possible implementation of a Fibonacci sequence generator namegen:

```

fibonacci : progtype
{
        fib : namegen (int):(int);
}

fib : (int) : (int) =
{
        v := <-fib;

        matchseq
        {
                (v == 0) =>
                {
                        fib <-= 0;
                }

                (v == 1) =>
                {
                        fib <-= 1;
                }

                true =>
                {
                        c1, c2 := name2chan int "fib" 0.0;
                        c1 <-= (v - 1);
                        c2 <-= (v - 2);
                        r <-= <-c1 + <-c2;
                }
        };
}
```

Another implementation of a Fibonacci sequence generator namegen:

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

```

parfib : (int) : (int) =
{
        c1, c2 : FibonacciType;

        v := <-parfib;
        matchseq
        {
                v == 0  => parfib <-= 0;
                v == 1  => parfib <-= 1;
                true    =>
                {
                        match
                        {
                        true    =>      c1 = name2chan FibonacciType "parfib";
                                        c1 <-= (v - 1);

                        true    =>      c2 = name2chan FibonacciType "parfib";
                                        c2 <-= (v - 2);
                        }
                        parfib <-= <-c1 + <-c2;
                }
        }
}
```

In the above example, the inner `match` statement is used to initiate two Fibonacci number computations without restriction on the ordering (i.e., they could potentially execute in parallel).

4.11 The `iter` construct

The `iter` statement is a repetitive collection of guarded statement blocks. It executes repeatedly while any of its guards, which are Boolean expressions, evaluate to true. The `iter` statement is analogous to *guarded iteration* in Dijkstra's guarded commands [5]. All statements whose guards evaluate to true execute. The syntax of `iter` statements is illustrated by the syntax diagrams in Figure 11.

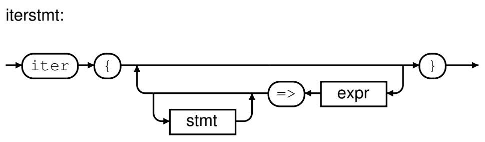


Fig. 11. Syntax diagrams for iter statements

5. OPERATOR DESCRIPTIONS

The semantics of the language-level operators are presented in the following sections. The language includes operators for operation on *values*, operators on *names* and operators on *channels*. The language-level operators and their semantics are listed in Table 1 and Figure 12.

5.1 Operators on names

The operators on names are `name2chan`, `chan2name`, and `var2name`. The `name2chan` operator applied to a name (string constant) yields a channel to an entry in the name space having the same type as the *l-value* of the statement, i.e., it matches on both the name as well as type of an entry in the runtime name space. If the name corresponds to a namegen, a new instance of the namegen is created, as described in Section 7. If multiple matches exist, the match is non-deterministic. A `name2chan` expression completes as soon as it either succeeds or times out. If no match exists (timed out), the result of the `name2chan` expression is `nil`. Example:

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT --

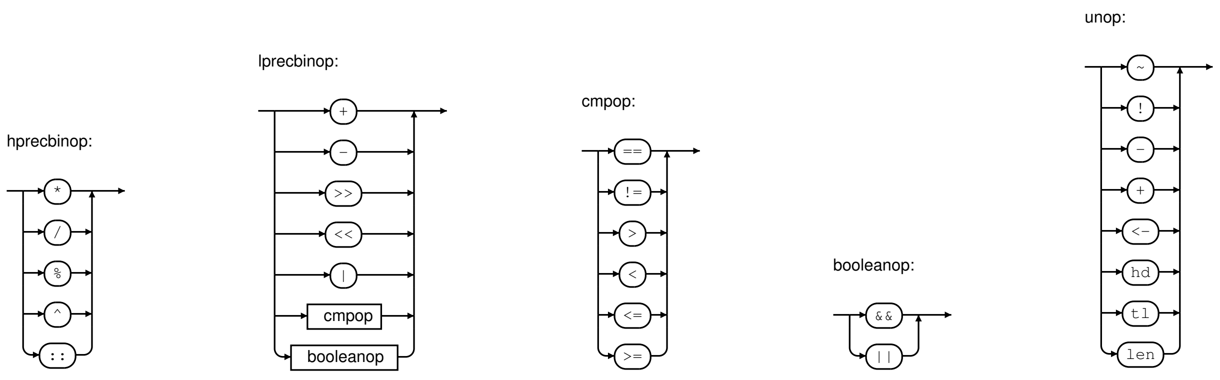


Fig. 12. Syntax diagrams for operators, and their classifications.

Table 1. Noisy language operators.

Operator	Description	Operator	Description
.	ADT member access	>=	Greater than or equal to
[]	Array or character string subscript	==	Equals
!	Logical not	!=	Not equals
~	Bitwise not	&	Bitwise AND, set intersection
++	Increment	^	Bitwise XOR
--	Decrement	|	Bitwise OR
hd	List head value	::	List append
tl	List tail value	&&	Logical AND
len	List, array, or string length; set size	||	Logical OR
name2chan	Bind name in name space to channel	=	Assignment
chan2name	Make channel visible as name in name space	:=	Declaration and assignment
var2name	Make variable visible as a name in name space	+=	Addition/concatenation/union assignment
type	Type cast	-=	Subtraction/difference and assignment
* / %	Multiplication, division, modulo	*=	Multiplication and assignment
+	Unary plus, addition, set union,	/=	Division and assignment
	string concatenation	%=	Modulo and assignment
-	Unary minus, subtraction, set difference	&=	Bitwise AND and assignment
<<	Logical left shift	|=	Bitwise OR and assignment
>>	Logical right shift	^=	Bitwise XOR and assignment
<	Less than	<<=	Logical left shift and assignment
>	Greater than	>>=	Logical right shift and assignment
<=	Less than or equal to	<-	Assignment to/from channel

```

#	Bind channel 'c' to the name "Fibonacci" having
#	type FibonacciType, timeout after 4 seconds
c	:= name2chan FibonacciType "Fibonacci" 4E6;
```

New entries in the runtime name space can be synthesized as interfaces to channels or variables within a namegen, via the `chan2name` and `var2name` operators. If there is an extant entry in the name space of the parent progtype with the same name *and* type, it is replaced with the new entry, but only within the scope of the initiating namegen and namegens which it has instantiated via `name2chan`. When a variable which has been made visible in the name space via `var2name` is written to, the write not only updates the memory cell on the local device, but also generates a write to any channel bound to the name.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT

5.2 Operators on channels

There are two operators on channels, a channel send (*channel* `<-=` *expr*) and a channel receive (*variable* or `nil <-` *channel*). Channel communication is unbuffered, rendezvous: a send (receive) completes when a matching receive (send) is performed at the other end of the channel.

All interaction between namegens is over a channel. Since namegens might be executing locally or remotely, the passage of references down a channel is not permitted. E.g., a channel cannot be passed down a channel. Instead, the reference must be converted to a name via chan2name. The value of chan2name and var2name expressions is the name (type `string`) that was posted for the chan, or `nil` if the posting failed. The chan2name and var2name operators can either be used to post an explicitly chosen name, e.g.

```
	chan2name mychan "name";
```

or might be used to generate an implicit name, in which case the name posted is chosen by the runtime and returned as the values of the `chan2name` or `var2name` expression.

6. TYPES

The syntax of type declarations and definitions were previously provided in Section 4. This section provides further detail on the language base types, as well as the construction of collection types.

The language includes a small set of basic types—`bool`, `nybble`, `byte`, `int`, `fixed` (fixed-point approximate real numbers), `real` (floating-point approximate real numbers) and `string`. The arithmetic types are `bool`, `nybble`, `byte`, `int`, `fixed` and `real`. The basic language data types may be employed in aggregate collections. The collection data types are arrays, tuples, ADTs (aggregate data types), lists and sets. Channels may be defined to carry any of the basic or aggregate data types, but these cannot include channels. Type definitions for the language base types may have an optional *error tolerance constraint*.

In addition to the basic types that can be used in type expressions, an identifier can also be of types `progtype`, `namegen` or act as an alias for a type expression. In the cases of progtypes and namegens, the type signatures are functions of the body of the progtype definition, and that of the namegen channel interface, respectively. ADTs and type aliases introduce new *type names*.

6.1 Reference versus value types, channels and `nil`

All the primitive and aggregate types of variables are value types. Channels are reference types (they have *implicit state*), and may not be passed down channels. A primitive or aggregate variable cannot be transmitted down a channel if it contains a channel within it. Since the equivalent of function calls are communications on channels, and as previously stated, these communications can only carry values, it could be said that the semantics for interaction between program components are *call-by-value*. Treating all types as value types should not be thought of as an inefficiency burden as an implementation can use a copy-on-write strategy to make computation within a namegen efficient.

6.2 Specifying tolerance constraints

For some variables in programs, it might be known that a certain degree of deviation from their correct values is acceptable. In the case of channels, a certain degree of erasures might be tolerable. Tolerance constraints on variables and channels, is key to being able to use the mathematical analysis of error magnitudes in programs [10] to implement forward error correction on the values exchanged between namegens. Conversely, it enables deliberate changes in functionality that improve performance but might induce errors.

For example, values to be transmitted on a communication channel might be occasionally purposefully dropped to reduce power consumption or improve *overall* network performance, while staying within prescribed constraints. As another example, in an application partitioned over devices that communicate over a wireless medium, the transmit power consumption might be purposefully reduced at the cost of a higher (tolerable) bit error rate. The low-level communication encodings are thus tuned to *application error tolerance properties*. We could think of the error tolerance constraints exposing semantic information that enables application-specific source and/or channel coding.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

The most flexible form in which constraints specifying *tolerable error* could be provided, would be as a *tail distribution* on error magnitude. For example, it might be desirable that the probability that the magnitude of error in a variable is greater than x should vary with x as $\frac{1}{x}$. The notational complexity of representing such error tolerance constraints would be significant: since it is logical for the error tolerance constraint to be placed in the type annotation, the type would then need to contain an expression in a variable (x in the above example). This approach is therefore avoided. Experience with the language and error tolerance constructs may necessitate revisiting this restriction.

Instead of such general error tolerance constraints, the language design includes a restricted form of the above in which x is constant. An error tolerance constraint is thus defined in terms of two variables, m and A, specifying that the probability the magnitude of error in a variable is greater than m, should be less than A, i.e., $\Pr\{M > m\} \leq A$. A type expression involving the basic types can be qualified with one or more error, erasure or latency tolerance constraints. For example, the following declares a variable `pixel`, of type `(byte, byte, byte, byte)`, with the error constraint that the probability that the magnitude of error (due to erasures or errors) in a member of the 4-tuple exceeds 4 should be 0.01:

```

t       : type int epsilon(4, 0.01);
pixel   : (t, t, t, t);
```

Latency tolerances are inherently at odds with error and erasure tolerances. With errors and erasures, one can introduce redundancy in the form of encoding to mitigate the effects of the undesirable phenomena. In the case of latency tolerances on the other hand, one must reduce redundancy to improve (decrease) latency (transmission time). On the other hand, the violation of latency tolerances can be easily checked in an implementation (e.g., via timers), whilst the violation of error magnitude tolerances is more involved: violation of the tolerance inherently means the decoding process wrongly decoded an erroneous codeword because it had more errors than that which could be corrected based on the minimum Hamming distance in the chosen code[4].

The goal of error and erasure tolerances in the current implementation is to address the problem of errors and erasures in the communication links between a partitioned application (between its constituent namegens). Tolerance constraints specified for channels apply to the values communicated on the channel. Tolerance constraints specified for variables are not used to encode the variables, but rather propagated upward through the dataflow graph to channels, reads from which reach the variables with tolerance constraints. Although a bit more difficult to reason about, the encodings also benefit values in programs, not only values communicated over a network (over channels): all variables can be thought of as channels to memory. This semantically exposes the fact that a read or a write from a “variable” might either mean reading from a local on-chip memory, or from a memory across the network, and the error tolerance associated with that “variable” might therefore either be used for bus encoding for error detection or forward error correction to a local memory, or for encoding a communication that happens over the network. A future implementation of the compiler could use the tolerance information on variables to encode the variables themselves, in order to mitigate the effect of soft-errors in memories, in an application-specific manner.

6.3 Types `bool`, `nybble`, `byte`, `int`, `real` and `fixed`

The primitive types `bool`, `nybble` and `byte` are unsigned quantities with sizes 1, 4 and 8 bits respectively. The type `int` is a signed 32-bit value in two’s complement format. The fixed point approximate real numbers, `fixed`, are signed 16-bit binary fixed point values in which 0 up to 15 bits can be used to represent the fractional component, and one bit is used to represent the sign. The floating point approximate real numbers, `real` are in 64-bit IEEE-754 double precision floating point format.

6.4 Type `string`

Strings are vectors of Unicode characters. An element in a string (a character), obtained using the notation *string_identifier[index]* is a 16-bit value. Strings can be concatenated using the `+` operator. The `len` operator,

[4]The minimum distance of a code, $d(C) \geq \delta \iff C$ is $\delta - 1$ error correcting. $d(C) \geq 2\epsilon + 1 \iff C$ is ϵ error correcting [4].

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

applied to a string yields the number of characters in the string. A substring or *slice* of a string is obtained using the notation:

string_identifier[start index (optional) ':' end index (optional)]

and yields a string. When the start (end) index is absent, it is implicitly the first (last) element of the string. The empty string `""` is represented by the value `nil`. Examples:

```

#       Define a new string 'name'
phrase  := "Mountains made of steam";

#       The variable c has type 'int', and holds the character 'M'
c       := phrase[0];

#       This sets the variable 'q' to "made of steam"
q       := phrase[len "Mountains ": ];
```

6.5 The array collection type

Arrays hold vectors of items of a single type. Multi-dimensional arrays are stored in memory in row-major order. The storage for arrays is conceptually allocated at the point of definition, thus there is no distinction between array declaration and array definition as there does in some languages. An element in an array is obtained using the notation *array_identifier[index]*. The `len` operator applied to an array yields the length of the array. A *slice* of an array is obtained using the notation:

array_identifier[start index (optional) ':' end index (optional)]

and yields an array of the same type. The type of an array includes its length, and thus all declarations of arrays include a specification of their length. However in an array definition from initializers, the dimensions may be omitted as they are implicit in the initializer. Example:

```

#       Declare 1-dimensional array 'a' of nybbles with 32 elements
a       : array [32] of nybble;

#       Declare a 3-dimensional array 'volume' of bits, and set one point
volume  : array [128][128][128] of bool;
volume[0][0][1] = 1;

#       Define a 3-dimensional array 'volume' of bits
volume2 := array [128] of {* => array [128] of {* => array [128] of bool}};

#       Definition of an array, type implicit from initializers,
first   := array of {"oil", "and", "water", "don't", "mix"};

#       Definition of an array, type implicit from initializers, with size 32
second  := array of {"oil", "and", "water", "don't", "mix", 31 => "."};

#       Definition of an array, type implicit from initializers, size 32;
#       the last 27 entries contain "."
third   := array [32] of {"oil", "and", "water", "don't", "mix", * => "."};
```

6.6 Tuple collection type

A tuple is an unnamed collection of items of possibly different type. A tuple type might be considered as a cartesian product of type expresisons. The elements in a tuple cannot be accessed individually; assignment to a tuple must be from a tuple expression, and assignment from a tuple type must be into a tuple of variables. Thus any assignment to a tuple, sets all fields. Example:

```

#       Declare color to be a tuple type
```

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

```
color	: (byte, byte, byte, byte);

color = (0, 6, 32, 8rFF);
```

6.7 Aggregate Data Type (ADT) collection type

ADTs are similar to tuples, with the only difference being that the members of an ADT may be named. An ADT declaration introduces a new type name, and subsequently, variables may be declared with that type name. Assignments from tuples with the same order and type of variables are permitted to ADT instance variables. Example:

```

#	Define an ADT type Color
Color : adt
{
	r : byte;
	g : byte;
	b : byte;
	a : byte;
};

#	Declare a variable with type Color
c : Color;

#	Assign a tuple to the ADT instance
c = (0, 6, 32, 8rFF);
```

6.8 The set collection type

The `set` collection data type is used to represent unordered collections of data items. A set can be cast to a list of the same type, yielding a non-deterministic ordered list. A list can likewise be cast as a set. The operations defined on set variables are *set union* ($A \bigcup B$ is denoted by `A + B`), *set intersection* ($A \bigcap B$ is denoted by `A & B`), *set difference or relative complement* ($A \bigcap \overline{B}$ is denoted by `A - B`) and *set cardinality* ($|S|$ is denoted by `len S`). The type of a set includes its cardinality, and thus all declarations of sets include a specification of their cardinality. However in a set definition from initializers, the cardinality may be omitted as it is implicit in the initializer. Multiplicity of elements is ignored (i.e., `set`s are not *multisets*). An empty set has the value `nil`. Examples:

```

#	Declare 's' as a set with cardinality 32, of integers
s	: set [32] of int;

#	Add a few elements to the set
s	+= set of {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41};

#	Check for set membership
u7p	:= s   &   set of {7};
```

6.9 Recursive list collection type

Recursive lists are lists of elements of a single type. The operations on `list` instances are `hd` *(head)*, which yields single datum, the first element in the list, `tl` *(tail)*, which yields a list representing all but the first element of the list, and `::` *(cons)*, which is used to append an element to the head of the list. Examples:

```

#	Define 'veggies' as a list of strings
veggies := list of {"carrot", "celery", "radish"};

#	'celery' will be a variable of type string initialized to the string "celery"
celery	:= hd tl veggies;

```

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

```
#	The cons (::) operator adds an item to a list
together := "rabbits" :: hd veggies :: nil
```

6.10 Dynamic data structures

There are no pointers *per se* in Noisy: this restriction ensures it is always possible to partition programs by using the runtime name space as an interface between partitions.

The only new instances that can be created at runtime are new instances of namegens. Namegens which embody data structures can be used to achieve dynamic data structures. A point of elegance in this approach is that, in much the same way that namegens may be instantiated on any hardware device on the network, the data structures embodied in namegens may reside on arbitrary devices in the network. As an example, consider a data structure used to implement nodes in the abstract syntax tree of a parser, implemented as a namegen, where `Node` is a previously defined ADT:

```

node : (Node):(Node) =
{
	n : Node;

	iter
	{
		match
		{
			<-node	=> node <-= n;

			node <- => n = <- node;
		}
		true	=>	;
	}
}

creategraph : ():() =
{
	n : Node;

	n.left = name2chan "node";
	n.right = name2chan "node";
}
```

Each time it is desired to dynamically create a new instance of the `Node` structure, a `name2chan` on the `node` namegen will create a new instance of the data structure, and the executor of the statement will obtain a channel to this new instance. In a sense, this approach bundles dynamically created data structures with their own access methods (through channel reads and writes). As a result, such data structures have a transactional interface.

7. THE RUNTIME SYSTEM

7.1 Runtime system abstraction

One way to look at the runtime system is by analogy to the memory system in a stored program computer. In traditional models of a stored program computer (e.g., the von Neumann / Princeton model), programs access *memory cells* by performing operations on *addresses*. These addresses form an *address space*. Two operations can be performed on a memory cell, given an address that represents it: memory read and memory write. When programs in a language such as C allocate memory dynamically, they are essentially communicating with the operating system to obtain a new address. We could call this operation on the address space new address, although it doesn't really *introduce* a new address, but just provides access to the address, to a program. We can thus say that the alphabet of operations on memories is {memory read, memory write, new address}. In a computer system, these operations are manifest as operations in hardware: a memread operation is embodied in instructions in the machine's instruction

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT

set architecture which perform a read from memory, e.g., an `ld` (load) instruction. Furthermore, the execution of such an instruction leads to a communication between the processor and the memory, yielding the result of the load.

By analogy, the runtime system in Noisy is made up of *names*, which are accessed through *channels*. The alphabet of operations pertinent to names is {nameread, namewrite, name2chan, chan2name, var2name}. The name2chan operation obtains a new channel through which programs can write to an entry in the name space. Conversely, chan2name (var2name) makes a channel (variable) in a program visible as an entry in the name space. Entries in the name space have the type structure of the variables or channels with which they are associated. As an example, a nameread operation is logically associated with a channel read expression in a program, just as a memread in the discussion above is logically associated with reading the value of a variable. Just as a memread leads to a communication on the address and data buses to memory, a nameread causes a device on which a namegen is executing to communicate with other computing devices which are part of the name space. It is at the level of these messages that error, loss (erasure) and latency tolerance constraints for channels are used to drive encoding. It is also from the framework implementing these low level messages that the tolerance constraint violations on channels are obtained. The nature and protocol of these communications is the subject of Section 7.5.4.

7.2 Target hardware model

The underlying hardware *abstraction* is assumed to be a collection of computing devices. More concretely, it is assumed that each namegen is mapped onto a *virtual processing device*, and the virtual processing devices are interconnected via a single *virtual network*. The virtual processing devices might have a one-to-one or many-to-one correspondence with actual processing devices (e.g., microcontrollers, processor cores, microprocessors, workstations). Similarly, the virtual network might be composed of a multi-hop wired or wireless network, on-chip bus, on-chip network, etc. The virtual processing devices will subsequently be referred to as *processing devices*, and the virtual network as *the network*, without discussing further details of either the nature of processing devices, or issues such as the underlying implementation of device-to-device communication, routing within the network, or optimization of placement/mapping of namegens to actual physical processing devices. The combination of virtual processing devices and the virtual network will be referred to collectively as the *runtime system*. The term *loading a program* will subsequently be used to refer to the act of placing the compiled Noisy program on a virtual device. It should be stressed that the above statements describe an *abstraction*; their purpose is to enable a description of the system without getting mired in implementation-specific details.

7.3 Structure of the runtime name space

All namegens exposed in an application's progtype definition appear as entries in the runtime name space with names of the form *progtype.namegen*. Entries in the runtime name space with distinct types associated with them are distinct. Duplicate entries (with identical names and types) may exist; this introduces nondeterminism.

A namegen is only *executing*, after its interface name has been bound to a channel. Each binding of a name to a channel creates a logically new instance of the namegen. Such a namegen instance so created shares the same name space as the namegen that bound its interface to a channel, and any changes to the runtime name space (e.g., creation of new entries via `chan2name` or `var2name`) are mutually visible. Besides entries corresponding to namegens, namegens may dynamically create entries in the runtime name space tied to channels. Such entries have the form *progtype.namegen.name*.

7.4 Enforcing access restrictions on names

The channels used by namegens to access names, can be used to implement the concept of *capabilities* from capability-based systems [8]. They provide an access path to software or hardware that is represented by a name in the name space. Access control can trivially be implemented by requiring the first value written down a channel to be an access right (e.g., a password)[5].

[5]Furthermore, a particular application might even define the sequence of accesses that must precede "real" interaction, to engage the accessor in, say, a Diffie-Hellman key generation, generating a shared secret that can be used to encrypt subsequent communications.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

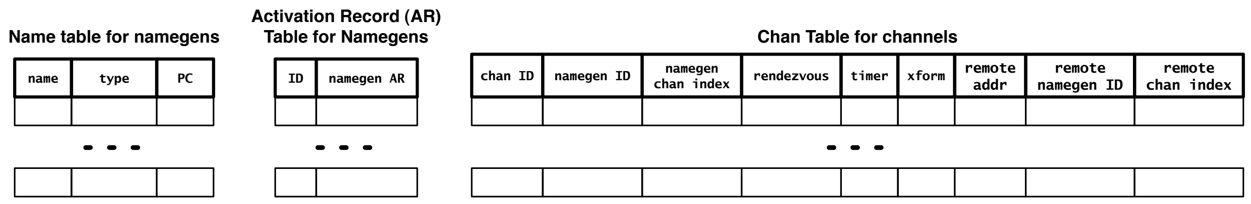


Fig. 13. Structures underlying the implementation of the runtime system.

7.5 Runtime system implementation

The *runtime system* is the collection of data structures, instantiated at runtime, that support the underlying operations being performed by executing programs. At the heart of the runtime system implementation is a set of three tables: the *name table*, *activation record table* and the *chan table*, illustrated in Figure 13. The implementation of these tables is described in Section 7.5.1.

7.5.1 *The name table.* The name table contains an entry for each namegen installed on a device, along with an entry representing the type structure of the namegen. The `Name` entries are strings representing the namegen, qualified by the progtype of which they are part (entries in the runtime name space are of the form *progtype.namegen[.name]*). At runtime, new entries are added to the name table whenever an executing namegen performs a `chan2name` or `var2name` operation. New entries in the name table are created whenever new code is installed on a device. By convention, a namegen with the identifier `init` immediately begins executing once loaded. Loading a namegen implementation with the same progtype as an extant one, into the runtime system (e.g., on a different device) should be thought of as being the same as overwriting code memory of a running application in a traditional program; undefined behavior will result.

When a namegen performs a `name2chan` operation, the local name table is first consulted. If no matching name and type is found locally, the name tables of all devices in the network are consulted[6]. If such an operation is successful, i.e., the name and type match an entry in the local name table, a new instance of the namegen (based on the `PC` entry) begins executing, with its own private stack. Such an instance is termed a namegen *activation*. The state for currently instantiated namegens is maintained in the *activation record table*.

7.5.2 *The activation record table.* The activation record table maintains the state corresponding to each namegen instantiation (as created by a `name2chan`, or an `init` namegen (Section 7.5.1)). The `ID` field uniquely identifies an instantiated namegen, and is used to identify namegens for all other operations. For example, all channels are associated with a particular namegen instance, and the instance's *ID* is used to track this correspondence. An instantiation of a namegen may create new entries in the runtime name space; these are only visible to the namegen that caused their instantiation (and to themselves).

7.5.3 *The chan table.* The chan table contains entries for all channels that are within the current scope of execution of all namegen activations. A namegen that performs a send or receive operation on a channel sleeps on a *rendezvous structure* in the chan table. When the channel communication operation completes (e.g., message successfully transmitted over network and an acknowledgment received), the sleeping namegen is awoken. The `xform` field contains a matrix (logically a part of the channel's type) representing the linear transformations that must be used to encode and decode the data exchanged between devices, representing the communication on a language-level channel. The messages on the network which are generated as a result of operations on channels are described in the next Section.

7.5.4 *Name communication protocol.* A small alphabet of messages may be exchanged between devices as a result of language-level constructs related to channels. The list of messages in this alphabet is provided in Table 2.

[6]Logically, the query is a broadcast, but an implementation may perform any number of optimizations to make this lookup more efficient.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

Table 2. Name communication protocol.

Message	**Description**	**Associated Language Construct**	**Parameters**
Tname2chan	Bind name to channel; if name is a namegen, create instance	`name2chan`	name, type
Rname2chan	Response: chan ID or nil		
Tnameread	Channel receive	Channel receive expression (`<-c`)	chan ID
Rnameread	Response: type structured data		
Tnamewrite	Channel send	Channel send expression (`c <-=`)	data, chan ID
Rnamewrite	Acknowledgment		

The following detail the effect of the receipt of messages in Table 2, on the runtime table structures, and on execution at the recipient node. The only language level channel operators that do not lead to the generation of messages are `chan2name` and `var2name`. They lead only to the creation of entries in the local name table.

Tname2chan / Rname2chan. Execution of a `name2chan` expression in a namegen will initiate the generation of a Tname2chan message on the network. A device which contains a matching entry (on both name and type) in its name table responds with a Rname2chan message. It is possible that no such device might exist, in which case the language-level expression will evaluate to `nil` after the language-level-specified timeout.

If the supplied name, in the name space, is a namegen, a new activation of the remote namegen is created, and corresponding entries for the send / receive interface channel tuple in the remote device's chan table are created. The ID of this allocated entry is returned to the initiating node in a `Rname2chan` message. A new entry is created in a local *chan table*, and this entry is used to store the received tag. Subsequent operations on the channel associated with this `name2chan` operation will occur with the specific instantiation of the remote namegen.

Tnamewrite / Rnamewrite. A channel send operation causes a Tnamewrite message to be generated on the network. The message target is determined by a lookup in the local chan table for (1) the destination network address and (2) the destination namegen `ID`, and (3) the destination namegen channel index, which identifies a channel in a specific instance of the remote namegen (with its private execution stack) that the values should be delivered to. The `timer` entry of the local chan table is updated with a timestamp, which is used to determine timeouts when the channel appears in a latency violation expression.

Tnameread / Rnameread. A channel receive operation causes a Tnameread message to be generated on the network. A lookup is performed in the local chan table to determine the remote device and the ID, to be included in the `Tnameread` message, designating which channel in a specific instantiation of a remote namegen the message should be delivered to. The `timer` entry of the local chan table is updated with a timestamp, which is used to determine timeouts when the channel appears in a latency violation expression.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

APPENDIX

A. A CALL-BY-VALUE LAMBDA CALCULUS WITH ERROR-TOLERANT TYPES

A central theme of the ideas presented in Noisy, is that, rather than adopting the traditional method of duplicating (or triplicating) computation in order to counteract the occurrence of failures, we introduce the concept of *error-tolerance constraints in programs*. To formalize this notion, we introduce a typed lambda calculus, λ_ε, in which the type ascriptions have error tolerance constraints. The role of λ_ε in this work is analogous to λ_{zap} in [9]. In contrast to [9], our goal is to enable programs to specify the amount of tolerable *numeric error magnitude*.

Type Inference Rules for λ_ε

$$\textsc{T-Int}\quad \frac{}{\Gamma \vdash n, \varepsilon : \texttt{int}, \varepsilon}$$

$$\textsc{T-True}\quad \frac{}{\Gamma \vdash \texttt{true}, \varepsilon : \texttt{bool}, \varepsilon}$$

$$\textsc{T-False}\quad \frac{}{\Gamma \vdash \texttt{false}, \varepsilon : \texttt{bool}, \varepsilon}$$

$$\textsc{T-ConstraintPreservationUnderError}\quad \frac{v : T_1, \varepsilon_1 \qquad K_{\varepsilon, f_t} : v \to v'}{\Gamma \vdash^{\langle t \rangle \sim f_t} v' : T, \varepsilon}$$

$$\textsc{T-Add}\quad \frac{\Gamma \vdash v : T, \varepsilon_1 \qquad \Gamma \vdash w : T, \varepsilon_2}{\Gamma \vdash v + w : T, q(\varepsilon_1, \varepsilon_2)}$$

$$\textsc{T-If}\quad \frac{\Gamma \vdash^{\langle t \rangle \sim f_t} \mathit{cond} : \texttt{bool}, \varepsilon \qquad \Gamma \vdash^{\langle t \rangle \sim f_t} a : T, \varepsilon_1 \qquad \Gamma \vdash^{\langle t \rangle \sim f_t} b : T, \varepsilon_2}{\Gamma \vdash^{\langle t \rangle \sim f_t} \ \texttt{if}\ \ \mathit{cond}\ \ \texttt{then}\ \ a\ \ \texttt{else}\ \ b : T, q(\varepsilon_1, \varepsilon_2)}$$

$$\textsc{T-Abs}\quad \frac{\Gamma, x : T_1, \varepsilon_1 \vdash y : T_2, \varepsilon_2}{\Gamma \vdash \lambda x : T_1, \varepsilon_1 . y : (T_1, \varepsilon_1 \to T_2, \varepsilon_2)}$$

$$\textsc{T-App}\quad \frac{\Gamma \vdash g : (T_1, \varepsilon_1 \to T_2, \varepsilon_2) \qquad \Gamma \vdash x : T_1, \varepsilon_1}{\Gamma \vdash g \ \ x : T_2, \varepsilon_2}$$

$$\textsc{T-Let}\quad \frac{v : T, \varepsilon_1 \qquad w : T, \varepsilon_2}{\texttt{let}\ \ v = w\ \ \texttt{in}\ \ e : T, r(\varepsilon_1, \varepsilon_2)}$$

The type inference rules for λ_ε are shown above. The first three inference rules are straightforward. For example, the value `true` with error tolerance constraint ε has type `bool`, with type error constraint ε. The fourth type inference rule, T-ConstraintPreservationUnderError is the key component that captures our proposal of error-tolerance transformations (encodings). The concept it embodies is that, if v has type T, and error tolerance

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

constraint ε, and K_{ε,f_t} is a transformation that takes as parameters the error tolerance constraint ε and a bit upset probability distribution, f_t, and encodes v to give v', then under the occurrence of a bit upset pattern $\langle t \rangle$, which follows distribution f_t, v' obeys the type and error constraint ascriptions of v.

B. FORMAL DEFINITION OF TYPE SYSTEM

The type inference rules specify the type resulting from the combination of expressions using the language level operators.

C. IMPLEMENTATION

The language grammar is listed in Appendix D. The language design is influenced most directly by the C (statement syntax) and Limbo (e.g., `adt` and `list` data type) programming languages , Hoare's CSP (channel concept) and Dijkstra's guarded commands (the `match` and `matchseq` statements), among others. Many of these influences were themselves influenced by earlier languages — Algol in the case of C, CSP, Alef and Newsqueak in the case of Limbo, etc. The overall *programming model* that Noisy provides bears some similarities to the Actors model [1]. The role of the runtime system and its associated language constructs (`name2chan`, `chan2name`, `var2name`) pervades the language structure. The `name2chan` construct enables both the facilities akin to function calls / process creation (function calls and `spawn` in Limbo). The `chan2name` and `var2name` language constructs were inspired by experience with the `sys->file2chan()` system library routine in the Inferno operating system [6]. The language level error, erasure and latency tolerance constraints as well as the constraint-violation driven control flow were borne out of our investigation of language-level transformations for error tolerance, and the programming language constructs which support them [13; 12].

The process of designing the grammar was driven by the desire to attain a language structure that was relevant to our goals of ease of partitioning and error-correctness tradeoffs, as well as a desire to simplify the implementation. The grammar is LL-1 to ensure that a recursive descent parser can be built for it, using only a single token of lookahead.

The compiler implementation was heavily influenced by Niklaus Wirth's Oberon compilers. Parsing is implemented with a recursive descent parser. Unlike Wirth's compilers which generate code in a single pass, the parser is used to build a tree-based intermediate representation, an *abstract syntax tree (AST)*. The AST is an abbreviated form of the parse tree, with nodes whose presence is implicit in a given subtree removed. Our use of the AST is essentially as an intermediate representation [7]. This in-memory intermediate representation is then traversed by one of the subsequent stages. There are currently two second-stage passes that use this intermediate representation — graph output, and C code generation. The graph output pass walks the AST and symbol table data structures, and outputs it in *dot* format, which is passed to the Dot [3] tool to render the graphs in one of many formats, including Postscipt. Figures C and 18 illustrate a small example program, its AST and the associated symbol table graphs generated by the compiler and rendered with Dot.

C.1 Building the AST

The lexical analyzer converts the input source into a sequence of *tokens*, corresponding to the reserved words, operators and separators in the language, and identifiers. The parser consumes these tokens from the lexer in the process of recognizing syntactic forms. The parser is structured as a set of routines, one per grammar production. Starting with the start symbol of a program, it recursively processes each valid production at that point in the parse. The decision of which production, and hence which routine to invoke next, is driven by the grammar $FIRST(X)$ and $FOLLOW(A)$ sets. During the language design, left recursion was eliminated from the grammar using Algorithm 4.1 in [2], and the grammar was left-factored using Algorithm 4.2 in [2]. The elimination of left-recursion is important as the $FIRST(X)$ sets of two alternatives in a grammar production may overlap in the presence of left-recursion. The top-level structure of the compiler is illustrate in Figure 16.

The AST is a binary tree; when nodes in the tree have greater than two children, the children are hung off a subtree of nodes used for chaining. Among other things, nodes contain references to entries in the symbol table, and to canonical trees representing their types, if relevant.

[7] Pascal P-code [14] was essentially a linearized form of the AST.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

C.2 Modifying the AST for easier code generation

There exist a few productions in the grammar which have as their result non-terminals. While this unclutters the grammar, it also leads in practice to AST subtrees that are unnecessarily "tall". For example, the grammar production for a constant delcaration is:

```
condecl                    ::= "const" (intconst | realconst | boolconst) .
```

The AST subtree built for a recognized `condecl` production will have as its root a node with type `condecl`, which will have as its only child (the token `const` is redundant in the AST) a node with type bing one of `intconst`, `realconst` or `boolconst`. The subtree of height two can however be substituted directly with one of these children. For lack of a better term, we may call this "tree height shortening". Tree shortening does not affect the functionality of the AST, and it greatly simplifies the task of routines that subsequently traverse the tree to generate code or perform other actions.

C.3 The symbol table

The symbol table stores context sensitive information gleaned from the input source during the process of parsing. Its structure is simple: it is a generalized tree with each subtree corresponding to a *scope*. Linked off each node in this tree is a list of identifiers, corresponding to the identifiers defined within that scope.

C.4 Types

Types are represented in the symbol table as trees built out of the primitive types and collection type constructors and correspond to the parse tree of a type expression. Two types are equivalent if their type trees are identical. Type equivalence is thus *structural* as opposed to *name equivalence*. To check types, the compiler performs a post-order walk of a type tree and generates a signature based on the nodes visited. Such a signature uniquely identifies a type. The graph generators include in the generated graphs textual renderings of these signatures on each node representing an identifier.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

D. Noisy LANGUAGE GRAMMAR

```
character               ::= Unicode-0000h-to-Unicode-FFFFh .
rsvopseptoken           ::= "~" | "!" | "%" | "^" | "&" | "*" | "(" | ")" | "," | "-" | "+" | "="
                            | "/" | ">" | "<" | ";" | ":" | "’" | "\"" | "{" | "}" | "[" | "]" | "|"
                            | "<-" | "." | "<=" | ">=" | "^=" | "|=" | "&=" | "%=" | "/=" | "*=" | "-="
                            | "+=" | ":=" | "!=" | ">>" | ">>=" | "<<" | "<<=" | "<-=" | "&&" | "||"
                            \| "::" | "=>" | "==" | "++" | "--" | "<-=" .
zeronine                = "0-9" .
onenine                 = "1-9" .
radix                   = onenine {zeronine} "r" .
charconst               = "’" character "’" .
intconst                ::= [radix] ("0" | onenine {zeronine}) | charconst .
boolconst               ::= "true" | "false" .
drealconst              = ("0" | onenine {zeronine}) "." {zeronine} .
erealconst              = (drealconst | intconst) ("e" | "E") intconst .
realconst               ::= drealconst | erealconst .
strconst                ::= "\"" {character} "\"" .
idchar                  = char-except-rsvopseptoken .
identifier              ::= (idchar-except-zeronine) {idchar} .

program                 ::= progtypedecl {namegendefn} .
progtypedecl            = identifier ":" "progtype" "{" progtypebody "}" .
progtypebody            = {ptypenamedecl ";"} .
ptypenamedecl           ::= identlist ":" (condecl | typedecl | namegendecl) .
condecl                 = "const" (intconst | realconst | boolconst) .
typedecl                = ("type" typeexpr) | adttypedecl .
adttypedecl             ::= "adt" "{" identlist ":" typeexpr ";" {identlist ":" typeexpr ";"} "}" .
namegendecl             ::= "namegen" tupletype ":" tupletype .
identornil              ::= (identifier {fieldselect}) | "nil" .
identornillist          = identornil {"," identornil} .
identlist               = identifier {"," identifier} .
typeexpr                ::= (basictype [tolerance {"," tolerance}]) | anonaggrtype | typename .
typename                ::= identifier ["->" identifier] .
tolerance               = errormagtolerance | losstolerance | latencytolerance .
errormagtolerance       = "epsilon" "(" realconst "," realconst ")" .
losstolerance           = "alpha" "(" realconst "," realconst ")" .
latencytolerance        = "tau" "(" realconst "," realconst ")" .
basictype               ::= "bool" | "nybble" | "byte" | "string" | "int" | realtype .
realtype                = "real" | fixedtype .
fixedtype               = "fixed" intconst "." intconst .
anonaggrtype            ::= arraytype | listtype | tupletype | settype .
arraytype               = "array" "[" intconst "]" {"[" intconst "]"} of typeexpr .
listtype                = "list" "of" typeexpr .
tupletype               = "(" typeexpr {"," typeexpr} ")" .
settype                 = "set" "[" intconst "]" "of" typeexpr .
initlist                = "{" expr {"," expr} "}" .
idxinitlist             = "{" element {"," element} "}" .
starinitlist            = "{" element {"," element} ["," "*" "=>" expr] "}" .
element                 = expr [ "=>" expr ] .
namegendefn             ::= identifier [":" tupletype ":" tupletype] "=" scopedstmtlist .
scopedstmtlist          ::= "{" stmtlist "}" .
stmtlist                = {stmt} .
stmt                    ::= [ identornillist ((":" (condecl | typedecl | typeexpr)) | (assignop expr))
                            | "(" identornillist ")" assignop expr | matchstmt | iterstmt
                            | scopedstmtlist ] ";" .
assignop                ::= "=" | "^=" | "|=" | "&=" | "%=" | "/=" | "*=" | "-=" | "+=" | ">>="
                           \| "<<=" | "<-=" | ":=" .
matchstmt               ::= ("match" | "matchseq") "{" guardbody "}" .
iterstmt                ::= "iter" "{" guardbody "}" .
```

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

```
guardbody               = {expr "=>" stmtlist} .
expr                    ::= (term {lprecbinop term}) | anonaggrcastexpr | chanevtexpr
                            | chan2nameexpr | var2nameexpr | name2chanexpr .
listcastexpr            = "list" "of" initlist .
setcastexpr             = "set" "of" initlist .
arrcastexpr             = "array" (("of" idxinitlist) | ("[" intconst "]" of starinitlist)) .
anonaggrcastexpr        ::= listcastexpr | setcastexpr | arrcastexpr .
chanevtexpr             ::= ("erasures" | "errors" | "latency") "of" identifier cmpop expr .
chan2nameexpr           ::= "chan2name" factor [strconst] .
var2nameexpr            ::= "var2name" factor [strconst] .
name2chanexpr           ::= "name2chan" typeexpr expr realconst .
term                    ::= [basictype] [unop] factor ["++" | "--"] {hprecbinop factor} .
factor                  ::= (identifier {fieldselect}) | intconst | realconst | strconst | boolconst
                            | "(" expr ")" .
fieldselect             ::= ("." identifier) | ("[" expr [":" expr] "]") .
hprecbinop              ::= "*" | "/" | "%" | "^" | "::" .
lprecbinop              ::= "+" | "-" | ">>" | "<<" | "|"  | cmpop | booleanop .
cmpop                   ::= "==" | "!=" | ">" | "<" | "<=" | ">=" .
booleanop               ::= "&&" | "||" .
unop                    ::= "~" | "!" | "-"  | "+" | "<-"  | "hd" | "tl" | "len" .
```

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

REFERENCES

[1] G. Agha. Actors.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[3] ATT. Dot.

[4] J. Baylis. *Error Correcting Codes: A Mathematical Introduction*. Chapman & Hall/CRC, 1997.

[5] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[6] S. Dorward. Inferno.

[7] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[8] H. M. Levy. *Capability-Based Computer Systems*. Digital Press, Bedford, Massachusetts, 1984.

[9] Reis. Faulty lambda calculus.

[10] P. Stanley-Marbell. Derivation of Probability Distributions of Data Values and Error Magnitudes in Program Variables Under Single and Multi-Bit Errors and Erasures (draft).

[11] P. Stanley-Marbell. The Sunflower Hardware Platform (draft).

[12] P. Stanley-Marbell and D. Marculescu. Pmup06.

[13] P. Stanley-Marbell and D. Marculescu. Tr-2008.

[14] UCSD. P.

[15] Unicode Consortium. *The Unicode Standard: Version 4.0*. Addison Wesley, Reading, Massachusetts, Aug. 2003.

[16] N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM*, 20(11):822–823, 1977.

[17] N. Wirth. *Grundlagen und Techniken des Compilerbaus*. Addison-Wesley, Bonn, 1 edition, 1996.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

Type Inference Rules for Operators

T-True
$$\frac{}{\texttt{true} : \texttt{bool}}$$

T-False
$$\frac{}{\texttt{false} : \texttt{bool}}$$

T-Var
$$\frac{}{x^{\tau} : \tau}$$

T-ADTmember
$$\frac{M : (\tau_1 \times \tau_2 \times \ldots \times \tau_n)}{M.k : \tau_k}$$

T-ArrayElem
$$\frac{M : (\tau \times \tau \times \ldots \times \tau)}{M[k] : \tau}$$

T-LogicalNot
$$\frac{M : \texttt{bool}}{!M : \texttt{bool}}$$

T-BitwiseNot
$$\frac{M : \tau \qquad \tau : \texttt{bool}|\texttt{nybble}|\texttt{byte}|\texttt{int}}{\texttt{~}M : \tau}$$

T-Increment
$$\frac{M : \tau \qquad \tau : \texttt{bool}|\texttt{nybble}|\texttt{byte}|\texttt{int}|\texttt{real}|\texttt{fixed}}{M\texttt{++} : \tau}$$

T-Decrement
$$\frac{M : \tau \qquad \tau : \texttt{bool}|\texttt{nybble}|\texttt{byte}|\texttt{int}|\texttt{real}|\texttt{fixed}}{M\texttt{- -} : \tau}$$

T-Head
$$\frac{M : \texttt{list of } \tau}{\texttt{hd } M : \tau}$$

T-Tail
$$\frac{M : \texttt{list of } \tau}{\texttt{tl } M : \texttt{list of } \tau}$$

T-Length
$$\frac{M : \tau \qquad \tau : \texttt{array}|\texttt{string}|\texttt{list}|\texttt{set}}{\texttt{len } M : \texttt{int}}$$

T-Name2Chan
$$\frac{M : \texttt{string}}{\texttt{name2chan } M : \texttt{chan of } \tau}$$

T-Chan2Name
$$\frac{M : \texttt{chan of } \tau}{\texttt{chan2name } M : \texttt{string}}$$

T-Var2Name
$$\frac{M : \texttt{chan of } \tau}{\texttt{var2name } M : \texttt{string}}$$

T-Multiplication
$$\frac{M_1 : \tau \qquad M_2 : \tau \qquad \tau : \texttt{bool}|\texttt{nybble}|\texttt{byte}|\texttt{int}|\texttt{fixed}|\texttt{real}}{M_1 \texttt{ * } M_2 : \tau}$$

T-Division
$$\frac{M_1 : \tau \qquad M_2 : \tau \qquad \tau : \texttt{bool}|\texttt{nybble}|\texttt{byte}|\texttt{int}|\texttt{fixed}|\texttt{real}}{M_1 \texttt{ / } M_2 : \tau}$$

T-Modulo
$$\frac{M_1 : \tau \qquad M_2 : \tau \qquad \tau : \texttt{bool}|\texttt{nybble}|\texttt{byte}|\texttt{int}|\texttt{fixed}|\texttt{real}}{M_1 \texttt{ \% } M_2 : \tau}$$

T-Add
$$\frac{M_1 : \tau \qquad M_2 : \tau \qquad \tau : \texttt{bool}|\texttt{nybble}|\texttt{byte}|\texttt{int}|\texttt{fixed}|\texttt{real}|\texttt{set}|\texttt{string}}{M_1 \texttt{ + } M_2 : \tau}$$

T-Sub
$$\frac{M_1 : \tau \qquad M_2 : \tau \qquad \tau : \texttt{bool}|\texttt{nybble}|\texttt{byte}|\texttt{int}|\texttt{fixed}|\texttt{real}|\texttt{set}}{M_1 \texttt{ - } M_2 : \tau}$$

T-Lshift
$$\frac{M_1 : \tau \qquad \tau : \texttt{nybble}|\texttt{byte}|\texttt{int}|\texttt{fixed}|\texttt{real} \qquad M_2 : \texttt{int}}{M_1 \texttt{ « } M_2 : \tau}$$

T-Rshift
$$\frac{M_1 : \tau \qquad \tau : \texttt{nybble}|\texttt{byte}|\texttt{int}|\texttt{fixed}|\texttt{real} \qquad M_2 : \texttt{int}}{M_1 \texttt{ » } M_2 : \tau}$$

T-LessThan
$$\frac{M_1 : \tau \qquad M_2 : \tau \qquad \tau : \texttt{bool}|\texttt{nybble}|\texttt{byte}|\texttt{int}|\texttt{fixed}|\texttt{real}|\texttt{string}}{M_1 \texttt{ < } M_2 : \texttt{bool}}$$

T-GreaterThan
$$\frac{M_1 : \tau \qquad M_2 : \tau \qquad \tau : \texttt{bool}|\texttt{nybble}|\texttt{byte}|\texttt{int}|\texttt{fixed}|\texttt{real}|\texttt{string}}{M_1 \texttt{ > } M_2 : \texttt{bool}}$$

T-LessThanEquals
$$\frac{M_1 : \tau \qquad M_2 : \tau \qquad \tau : \texttt{bool}|\texttt{nybble}|\texttt{byte}|\texttt{int}|\texttt{fixed}|\texttt{real}|\texttt{string}}{M_1 \texttt{ <= } M_2 : \texttt{bool}}$$

T-GreaterThanEquals
$$\frac{M_1 : \tau \qquad M_2 : \tau \qquad \tau : \texttt{bool}|\texttt{nybble}|\texttt{byte}|\texttt{int}|\texttt{fixed}|\texttt{real}|\texttt{string}}{M_1 \texttt{ >= } M_2 : \texttt{bool}}$$

Fig. 14. Type inference rules for language operators (part 1).

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

Type Inference Rules for Operators *(continued)*

T-Equals
$$\frac{M_1 : \tau \qquad M_2 : \tau \qquad \tau : \texttt{bool}|\texttt{nybble}|\texttt{byte}|\texttt{int}|\texttt{fixed}|\texttt{real}|\texttt{string}}{M_1 \;\texttt{==}\; M_2 : \texttt{bool}}$$

T-NotEquals
$$\frac{M_1 : \tau \qquad M_2 : \tau \qquad \tau : \texttt{bool}|\texttt{nybble}|\texttt{byte}|\texttt{int}|\texttt{fixed}|\texttt{real}|\texttt{string}}{M_1 \;\texttt{!=}\; M_2 : \texttt{bool}}$$

T-BitwiseAND
$$\frac{M_1 : \tau \qquad M_2 : \tau \qquad \tau : \texttt{bool}|\texttt{nybble}|\texttt{byte}|\texttt{int}}{M_1 \;\texttt{\&}\; M_2 : \tau}$$

T-SetIntersection
$$\frac{M_1 : \texttt{set of } \tau \qquad M_2 : \texttt{set of } \tau}{M_1 \;\texttt{\&}\; M_2 : \texttt{set of } \tau}$$

T-BitwiseXOR
$$\frac{M_1 : \tau \qquad M_2 : \tau \qquad \tau : \texttt{bool}|\texttt{nybble}|\texttt{byte}|\texttt{int}}{M_1 \;\texttt{\^{}}\; M_2 : \tau}$$

T-BitwiseOR
$$\frac{M_1 : \tau \qquad M_2 : \tau \qquad \tau : \texttt{bool}|\texttt{nybble}|\texttt{byte}|\texttt{int}}{M_1 \;\texttt{|}\; M_2 : \tau}$$

T-ListCons
$$\frac{M_1 : \tau \qquad M_2 : \texttt{list of } \tau}{M_1 \;\texttt{::}\; M_2 : \texttt{list of } \tau}$$

T-LogicalAND
$$\frac{M_1 : \texttt{bool} \qquad M_2 : \texttt{bool}}{M_1 \;\texttt{\&\&}\; M_2 : \texttt{bool}}$$

T-LogicalOR
$$\frac{M_1 : \texttt{bool} \qquad M_2 : \texttt{bool}}{M_1 \;\texttt{||}\; M_2 : \texttt{bool}}$$

T-Assignment
$$\frac{M_1 : \tau \qquad M_2 : \tau}{M_1 \;\texttt{=}\; M_2 : \tau}$$

T-DeclAssignment
$$\frac{M_1 : \tau \qquad M_2 : \tau}{M_1 \;\texttt{:=}\; M_2 : \tau}$$

T-AddAssignment
$$\frac{M_1 : \tau \qquad M_2 : \tau}{M_1 \;\texttt{+=}\; M_2 : \tau}$$

T-SubAssignment
$$\frac{M_1 : \tau \qquad M_2 : \tau}{M_1 \;\texttt{-=}\; M_2 : \tau}$$

T-MulAssignment
$$\frac{M_1 : \tau \qquad M_2 : \tau}{M_1 \;\texttt{*=}\; M_2 : \tau}$$

T-DivAssignment
$$\frac{M_1 : \tau \qquad M_2 : \tau}{M_1 \;\texttt{/=}\; M_2 : \tau}$$

T-ModAssignment
$$\frac{M_1 : \tau \qquad M_2 : \tau}{M_1 \;\texttt{\%=}\; M_2 : \tau}$$

T-ANDAssignment
$$\frac{M_1 : \tau \qquad M_2 : \tau}{M_1 \;\texttt{\&=}\; M_2 : \tau}$$

T-ORAssignment
$$\frac{M_1 : \tau \qquad M_2 : \tau}{M_1 \;\texttt{|=}\; M_2 : \tau}$$

T-XORAssignment
$$\frac{M_1 : \tau \qquad M_2 : \tau}{M_1 \;\texttt{\^{}=}\; M_2 : \tau}$$

T-LshiftAssignment
$$\frac{M_1 : \tau \qquad M_2 : \tau}{M_1 \;\texttt{«=}\; M_2 : \tau}$$

T-RshiftAssignment
$$\frac{M_1 : \tau \qquad M_2 : \tau}{M_1 \;\texttt{»=}\; M_2 : \tau}$$

T-ChanReadToLvalue
$$\frac{M_1 : \tau_1 \qquad M_2 : \texttt{chan of } \tau_1 \qquad M_3 : \texttt{chan of } \tau_2}{M_3 \;\texttt{=}\; M_1 \;\texttt{=}\; \texttt{<-}\; M_2 : \texttt{chan of } \tau_2}$$

T-ChanReadTest
$$\frac{M_2 : \texttt{chan of } \tau}{\texttt{<-}\; M_2 : \texttt{bool}}$$

T-ChanWriteTest
$$\frac{M_2 : \texttt{chan of } \tau}{M_2 \;\texttt{<-}\; : \texttt{bool}}$$

T-ChanWrite
$$\frac{M_1 : \tau_1 \qquad M_2 : \texttt{chan of } \tau_1 \qquad M_3 : \texttt{chan of } \tau_2}{M_3 \;\texttt{=}\; M_2 \;\texttt{<-=}\; M_1 : \texttt{chan of } \tau_2}$$

Fig. 15. Type inference rules for language operators (part 2).

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

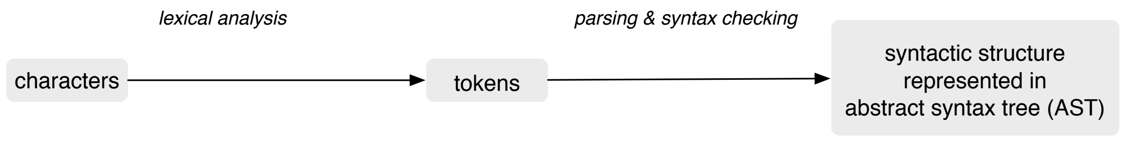

lexical analysis
parsing & syntax checking
characters
tokens
syntactic structure represented in abstract syntax tree (AST)

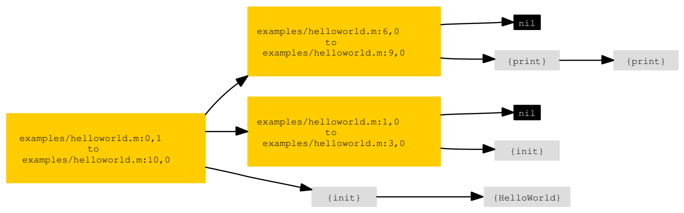
Fig. 16. Structure of the compiler.

```
HelloWorld : progtype
{
        init    : namegen ():();
}

init =
{
        print := name2chan System->print "system.print" 0.0;
        print <-= "Hello World!";
}
```

Fig. 17. An example program.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT


examples/helloworld.m:0,1 to examples/helloworld.m:10,0
examples/helloworld.m:6,0 to examples/helloworld.m:9,0
examples/helloworld.m:1,0 to examples/helloworld.m:3,0
nil
{print}
{print}
nil
{init}
{init}
{HelloWorld}

(a) Symbol table state after parsing the `HelloWorld` program. The Figure is generated automatically by the compiler.

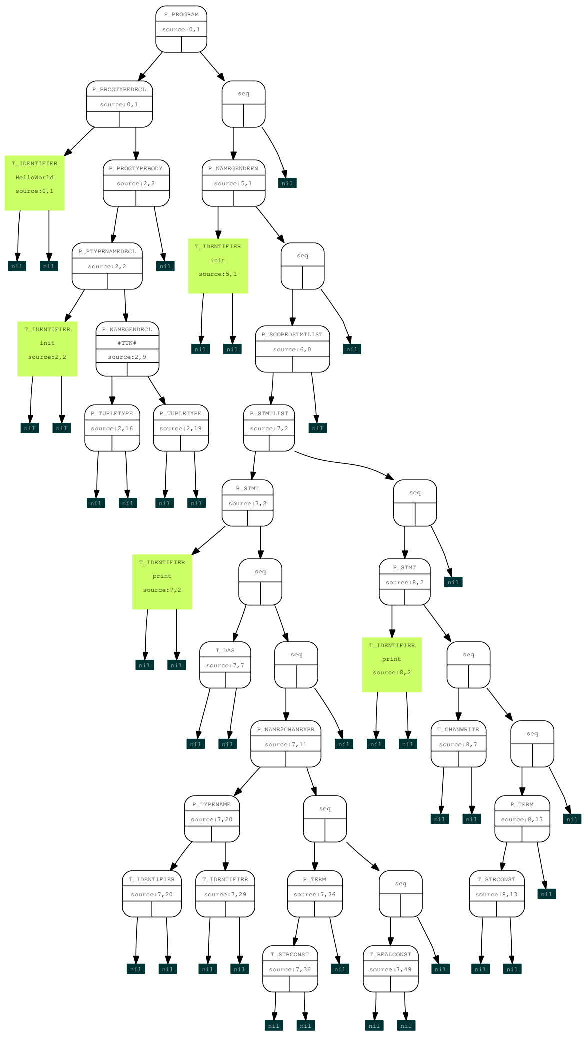


Fig. 18. Intermediate representation (Abstract Syntax Tree (AST)) of `HelloWorld` program. The Figure is generated automatically by the compiler.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -