Final Year Project Report

R.E.A.L.

Eliott Bartley

A thesis submitted in part fulfilment of the degree of

BA/BSc (hons) in Computer Science

Supervisor: Dr. Joseph Kiniry

Moderator: Dr. Neil Hurley

UCD School of Computer Science and Informatics
College of Engineering Mathematical and Physical Sciences
University College Dublin

May 8, 2009

Table of Contents

Abstract

Real-time Execution and Analysis Language (REAL) is a small imperative programming language and IDE. REAL has the novel feature that it is compiled, statically-analysed, and executed by the IDE on every key-stroke. A REAL programmer sees the program's output in real-time as the program is written.

Acknowledgements

Thank you to Dr. Joseph Kiniry; it was fun talking about a topic I enjoy with someone who knows absolutely everything about it.

Chapter 1: **Introduction**

1.1 Project Specification

This project focuses on the creation of a simple imperative programming language, akin to the WHILE language [1], and its (correspondingly simple, but novel) development environment [2].

The novel feature of this research project is that this new language is compiled, executed, and (optionally) statically analysed on-the-fly as a developer writes new code and makes changes to existing code. The results of execution and static analysis are shown to the developer in real-time. These results focus on the program's dynamic state and static properties.

In particular, a program's dynamic state is summarised much like modern debuggers do today, but instead of summarising the current state of the stack and heap, REAL summarises the state of the stack and heap at each quiescent program state.

In addition, other semantic information is summarised to the user. The static properties of interest are data flow dependencies and assertions about program properties. For example, if the developer highlights a given variable 'v', all other variables that depend, either directly or indirectly on 'v' in the program are also highlighted.

The underlying aim of the project is to create an environment that promotes developer understanding and helps developers reason about their programs by giving immediate and visual feedback about how changes in a program's source code affect the rest of the program.

Mandatory:

- (Language Design) Design an imperative garbage-collected programming language with support for (at least) assertions, literals, types, variables, arithmetic operators, loops, conditionals, and procedures.
- (Parser) Implement a parser capable of compiling the language in real-time for an interpreter.
- (Interpreter) Implement an interpreter capable of executing the program in real-time.
- (IDE) Implement a simple development environment that permits a developer to edit program code and witness program state updating in real-time at each quiescent program point.

Discretionary:

- (Compound types) Add support for compound types to the language and environment.
- (Reference types) Add support for reference types to the language and environment.
- (Sub-types) Add support for sub-typing to the language and environment
- (Static checks) Add support for performing static checks of the program in real-time.
- Implement one of the following static checks:

  - data flow analysis,
  - information flow analysis,
  - assertion validity checking, or
  - weakest-precondition/strongest-postcondition generation.
- (Concurrency) Add support for execution speed optimisation by executing independent statements (or subsections of statements) concurrently.
- (Memory graph) Display a graph showing how memory (stack, heap, etc.) in used at any quiescent program state.

Exceptional:

- (Research paper) Write, submit, and publish a paper on the results.

1.2 Overview

REAL is an interpreted statically-typed imperative programming language. REAL supports literals, variables, arithmetic operators, arithmetic precedence, types, procedures, conditionals, loops, and assertions. REAL's basic types are Boolean, Integer, Float, and String.

REAL includes an IDE. The IDE automatically compiles and executes the REAL source at every keystroke. After executing, the IDE captures the results for each top-level statement. The IDE outputs each result next to its respective statement.

REAL's IDE also outputs the program's state as it is at the cursor position. The IDE passes the cursor position to the REAL compiler. The compiler flags the atomic operation nearest to the cursor position. The interpreter executes each atomic operation. The flagged atomic operation records the program state after it is executed. The state information contains the partial execution result and the visible binding stack. REAL's IDE outputs this state information below the program.

REAL and its IDE is implemented in the Java™ programming language. A Java™ Applet wrapper around the REAL IDE is runnable on-line at http://spaghetticode.net/real.htm.

Chapter 2: **Background Research**

2.1 Language Design

REAL's design follows from the principles in "The Principles of Programming Languages" as defined by MacLennan [8]. The Principles of Programming Languages describes several practices for good program design. These principles are:

- *"The Activation Record"* The activation record states "An activation record is an object holding all of the information relevant to one activation of an executable unit." REAL applies this principle through procedure scoping. Variables and sub-procedures defined inside a procedure's scope exist only with the procedure.
- *"The Automation Principle."* This principle states "Automate mechanical, tedious, or error-prone activities." REAL applies this principle where it does not conflict with other principles. All low-level operations in REAL are abstract in their syntax. This hides tedious error-prone activities. This principle is not applied in automating type conversion as this conflicts with the manifest interface principle.
- *"The Abstraction Principle."* The abstraction principle states "Avoid requiring something to be stated more than once; factor out the recurring pattern." A corollary to the automation principle. REAL applies this principle by supporting procedures.
- *"Defense in Depth Principle"* The defense in depth principle states "If an error gets through one line of defense (syntactic checking, in this case), then it should be caught by the next line of defense (type checking, in this case)." REAL supports this principle through several layers of error checking. Error checking is performed when parsing, examining AST structure, type-checking, and during execution.
- *"Information Hiding Principle"* The information hiding principle states "Modules should be designed so that: (1) The user has all the information needed to use the module correctly, and nothing more. (2) The implementor has all the information needed to implement the module correctly, and nothing more." REAL partially applies this principle through scoping. REAL supports scope in procedures and around a sub-section of statements. REAL does not support multiple modules so this principle is taken no further.
- *"The Labeling Principle"* The labeling principle states "We should not require the user to know the absolute position on an item in a list. Instead, we should associate labels with any position that must be referenced elsewhere." REAL applies this principle. All variables, procedures, and parameters must be named.
- *"The Localized Cost Principle"* The localized cost principle states "A user should only pay for what he uses; avoid distributed costs." REAL inherits this principle from its minimal design. All language constructs are minimal with no extensions.
- *"Manifest Interface Principle"* The manifest interface principle states "All interfaces should be apparent (manifest) in the syntax." REAL applies this principle. All actions in REAL must be explicitly specified.

- *"The Orthogonality Principle."* The orthogonality principle states "Independent functions should be controlled by independent mechanisms." REAL partially applies this principle. The grammar in REAL is specifically chosen so all independent functions use unique symbols or syntax insofar as standard ASCII symbols allow. The orthogonality principle is not applied to the following operators:
  - assignment = : conflicts with (equivalence ==, less-or-equal <=, and greater-or-equal >=
  - addition + : conflicts with unary absolute +
  - subtraction - : conflicts with unary inverse -
- *"The Portability Principle."* The portability principle states "Avoid features or facilities that are dependent on a particular machine or small class of machines." As an interpreted language, REAL inherits this principle.
- *"Preservation of Information Principle"* The preservation of information principle states "The language should allow the representation of information that the user might know and that the compiler might need." REAL applies this principle through requiring all actions to be explicit.
- *"The Regularity Principle."* The regularity principle states "Regular rules, without exceptions, are easier to learn, use, describe, and implement." REAL applies this principle through a shared operator set. All basic types share the set of operator symbols. As an example, the plus symbol (+) performs addition on Integers and Floats, concatenation on Strings, and logical OR on Boolean. This is possible as REAL does no automatic conversion and is explicitly typed.
- *"The Security Principle."* This principle states "No program that violates the definition of the language, or its own intended structure, should escape detection." REAL applies this principle through its deterministic parser and explicit statically analysed type-checker.
- *"The Simplicity Principle"* The simplicity principle states "A language should be as simple as possible. There should be a minimum number of concepts with simple rules for their combination." REAL inherits this principle from its minimalist design.
- *"The Structure Principle"* The structure principle states "The static structure of the program should correspond in a simple way with the dynamic structure of the corresponding computations." REAL applies this principle through structured loops and procedures. REAL does not support a GOTO statement.
- *"Syntactic Consistency Principle"* The syntactic consistency principle states "Things that look similar should be similar and things which look different should be different." REAL inherits this principle from applying the orthogonality principle and regularity principle.
- *"Zero-One-Infinity Principle"* The zero-one-infinity principle states "The only reasonable numbers in a programming language design are zero, one, and infinity." REAL does not support this principle. REAL does not support references, thereby not supporting NULL references (zero). REAL does not support arrays (infinity). REAL only supports defined single instances (one).

2.2 Parsing

In general, parsing is achieved as follows: A parser takes as input a source and grammar. The source is a text that is to be parsed. The grammar is a set of rules describing how it is parsed. The grammar rules are a set of mappings from one set of symbols to another. The resulting symbols may be mapped further (non-terminals). Symbols that cannot be mapped further are at a final state (terminals). The source is broken into symbols and recursively mapped by the grammar. Parsing is complete when only symbols that cannot be mapped further remain. The order of these mappings is the parsing for the given source. This order forms a tree structure called an abstract syntax tree (AST).

There are several different types of parser based on the complexity of the grammar's rules. These types are hierarchically ordered by the Chomsky hierarchy [5]. The Chomsky hierarchy is split into four types.

- Type 0: Type 0 grammars are known as unrestricted grammars. This type of grammar allows each rule to have any number of symbols map to any other number of symbols. This allows the symbol set to grow, adding significant complexity. Being able to map from several symbols also adds significant complexity, when the symbols match across arbitrary portions of the source.
- Type 1: Type 1 grammars are known as context-sensitive grammars. This type of grammar is similar to type 0, but in any mapping, the set of non-terminal symbols must map to a set of symbols that is the same, bar one symbol. This does not allow the symbol set to grow. Context-sensitive grammars are so called because the one symbol that is changed during mapping remains in the context of those that are not changed.
- Type 2: Type 2 grammars are known as context-free grammars. This type of grammar allows only a single symbol to be mapped to any number of other symbols. Context-free grammars are so called because they map from a single symbol, thereby ignoring any surrounding symbols. As type 2 grammars only map from a single symbol, the parser can be simply a finite-state automata (FSA) reading the source from left-to-right.
- Type 3: Type 3 grammars are known as regular grammars. This type of grammar allows only a single symbol to map to several terminal symbols followed by an optional single non-terminal symbol.

In choosing a grammar type for REAL, type 0 and type 1 are rejected due to their algorithm's time complexity. A type 3 grammar is also rejected due to being too restricted. A type 2 grammar is then chosen.

Several other types of parser also exist. Though they are not part of the Chomsky hierarchy, they do have a place in the hierarchy. One such parser is for a restricted type 2 grammar. Restricted type 2 grammars fit between type 2 and type 3 grammars in the Chomsky hierarchy. This restriction is that no two mappings be from the same symbol. Type 2 grammars allow the mappings $a \mapsto b$ and $a \mapsto c$ to exist together. These mappings require a non-deterministic FSA to parse. This is not allowed in a restricted type 2 grammar. A deterministic FSA is sufficient to parse restricted type 2 grammars.

Using a deterministic FSA that reads from left-to-right means a parsing can be generated with time complexity O(n). Due to REAL's real-time nature, this property is desired. Therefore, a restricted type 2 grammar is chosen for REAL.

2.3 Structural Operational Semantics

Structural Operational Semantics (SOS), introduced by Plotkin [9] and further discussed by Hennessy [6], is a mathematical notation for defining the operational semantics of computer programs. SOS is structural as it defines rules based on the structure of the programming language. Each rule defines an atomic operation in the language. Compound semantics come from recursively applying the SOS rules. This recursion follows the AST structure. Each recursion is the next level in the tree.

SOS is not only useful for operational semantics. SOS is useful for defining any semantics on a recursive structure. Therefore, SOS is also used to define the type semantics of computer programs.

Other approaches to defining operational semantics also exist, such as denotational semantics [10]. However, these other approaches only define operational semantics. SOS is chosen for REAL due to its application in not just operational semantics, and its clean layout.

Figure 2.1 shows a brief introduction to SOS.

Figure 2.1: Introduction to SOS

Name $\frac{a}{b}$ ^{c}	This is the general structure of an SOS rule. "Name" is a human readable name for this rule. This rule state that anywhere "a" occurs (the structure), "b" is the result (the semantics). An optional addendum to the rule is "c."
Γ	SOS uses Γ to represent the environment in which the program exists.
Γ,a	This states the environment set explicitly includes "a."
σ	SOS uses σ to represent a store. This is where bindings are stored.
$\sigma\{i\}$	The store is accessed by specifying a binding after the store.
$\sigma\{v/i\}$	The store is updated by specifying a value and a binding after the store.
$\vdash$	$\vdash$ is a turnstile to separate the environment from the program.
|	| is a turnstile to separate the store from the program.
$\diamond$	$\diamond$ is an identity. $\diamond$ is used when a rule is effected by nothing or affects nothing.
Assignment $\frac{\Gamma,i\to i_{loc}\|\sigma\vdash e\rightsquigarrow v\|\sigma}{\Gamma\|\sigma\vdash i=e\rightsquigarrow v\|\sigma\{v/i_{loc}\}}$	This rule states (above the line) if i can be found in the store ($\to i_{loc}\|\sigma$) and e reduces to a value, then (below the line) $i = e$ reduces to v (the value returned from this operation is the value v) and the store is updated, setting i to the value v.

Chapter 3: **Design**

3.1 Language

REAL supports the following basic types which all share a single kind [3]:

Type	Values
bool	true, false
int	signed 32bit integer
float	32bit IEEE float
string	double-quoted with escape \ for \" and \\

REAL supports the following operator set, sorted by precedence:

Operator	Inputs	Description	Type	Example
~	unary	scope access		~a
:	binary	type specifier		a:int
+	unary	absolute	int,float	+a
-	unary	not; negate	bool; int,float	-a
^	binary	xor; power	bool; int,float	a^b
\	binary	root	int,float	a\b
|	binary	log	int,float	a|b
*	binary	and; multiply	bool; int,float	a*b
/	binary	divide	int,float	a/b
%	binary	modulus	int,float	a%b
+	binary	or; add; cat	bool; int,float; string	a+b
-	binary	subtract; remove	int,float; string	a-b
<, <=, >=, >	binary	relative	int,float,string	a<b
==	binary	equal	bool,int,float,string	a==b
-=	binary	not equal	bool,int,float,string	a-=b
=	binary	assign		a=b

REAL also has the following syntax:

- *ID* : Identity value
- *a:int*=0 : Declare a new variable a with type int and value 0
- *a:int()*={0} : Declare a new procedure a with return type int and no parameters. The procedure returns a constant value 0
- *a:int(b:int)=b* : Declare a new procedure a with return type int and one parameter b of type int. The procedure returns the value passed to it
- *a* : Access the value a
- *a()* : Access the procedure a

- *a:int* : Convert the expression a to an integer. This differs from declaring a new variable a because there is no =
- *if(a){b}* : Execute and return b only if a is true, otherwise return identity
- *if(a){b}else{c}* : Execute and return b only if a is true, otherwise execute and return c
- *while(a){b}* : Repeat b until a is true
- *assert(a)* : Assert a to be true

All expressions in REAL return a value. As an example, the expression `if(a){1}else{2}` returns 1 if a is true and 2 if a is false. This poses an interesting question; what is returned from the expression `if(false){1}`? To account for this, REAL includes an identity value. All expressions that cannot return a known value return identity. All expressions have semantics to deal with an identity value and typically this is to simply return identity. Arithmetic binary operators, however, use identity to mean the identity of that operator. The identity of addition (+) is 0, the identity of multiplication (*) is 1, etc. REAL encapsulates identity as a special value known as "ID". REAL uses this to mean the identity value of the operator. As an example, the expression `2+ID` is 2. If both sides of a binary operator are identity, identity is returned.

3.2 Interface

REAL's interface is shown in figure 3.1. It is split into four parts. In the centre is the source input. The source is highlighted near the cursor. This is the current "focus" point. To the right of this is the output of each statement. Each output appears on the same row as the related input. Below the source input is the state of the program at the "focus" point. The first line is the result of the expression inside the "focus" point. The following lines show the scope stack and what is visible in that scope. Each line is another scope level further from the current "focus" point. To the left of the source input is an output of the AST. The AST is not visible in figure 3.1.

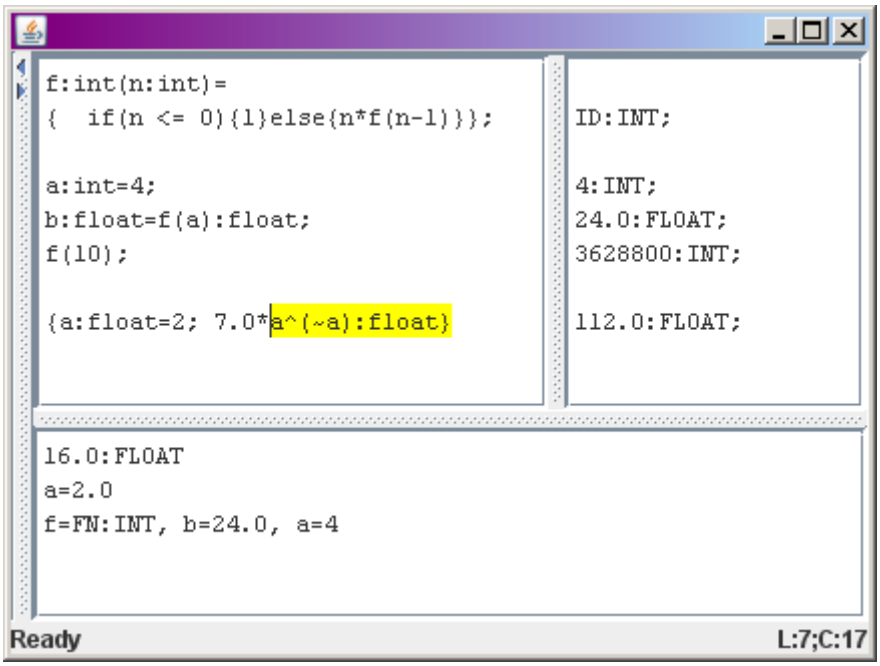
3.3 Parser

REAL is designed with a restricted type 2 grammar. However, one exception to this exists. Type conversion and type declaration have a similar syntax. Type conversion is stated as `identifer:type`. Type declaration is stated as `identifier:type=value`. This similarity is from the regularity principle. When parsing, the difference is only known when the = symbol is parsed. This requires a non-deterministic FSA to parse. To allow REAL to use a deterministic FSA, an extra ability is added. This ability allows the parser to modify the AST after it is generated. This allows the parser to change an AST conversion node to an AST declaration node without effecting the rest of the AST.

Figure 3.5 shows the deterministic FSA used by REAL. The format of this FSA is:

- *name*@=s : Define the start/final states where name is "start" or "final" and s is the state
- $s_0, c \rightarrow s_1, a_0, \ldots, a_N$: Define a transition from state s_0 to s_1 on reading a character from set c and performs actions a_0 to a_N

Figure 3.1: REAL's interface



- $s_0, c \rightarrow s_1 : s_2, a_0, \ldots, a_N$: Define a GOSUB transition. Go to transition s_1 and on return go to transition s_2
- $s_0, c \rightarrow :, a_0, \ldots, a_N$: Define a RETURN transition. Go to the return transition specified by the previous GOSUB transition
- $s_0, c \rightarrow, e$: Generate a parse error with reason e on reading character from set c
- $s_0, n, c \Rightarrow s_1, s_2, a_{i0}, \ldots, a_{iN}; a_{t0}, \ldots, a_{tN}$: Define a transition from state s_0 to s_1 on reading a sequence of characters n where the last character is not from set c. On reading the first character perform actions a_{i0} to a_{iN}. On reading the final character perform actions a_{t0} to a_{tN}. If any character is not read, go to state s_2

Special symbols used by the FSA include:

- # : Read an EOF symbol
- $: Define an `ELSE` transition. This is a transition that occurs if no other transition can be followed from the state. Symbols are not consumed during `ELSE` transitions
- * : Read any character
- *[a]** : Read any character except characters from set *a*

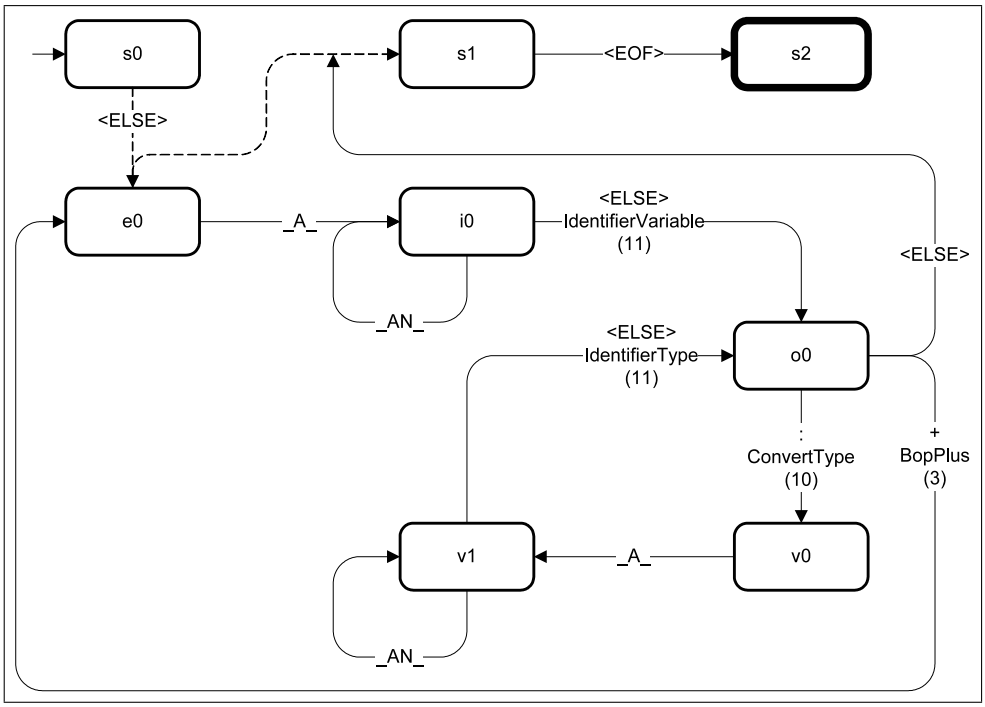
Actions performed by the FSA include:

- *name(level)* : Add a node to the AST at a specified level
- *name*(level)* : Add an identifier node to the AST at a specified level. This node stores a parsed identifier name

- *name(ilevel)* : Add a node to the AST at a specified level, but insert instead of append. The results in a left-to-right execution
- *name(olevel)* : Overwrite a node in the AST at a specified level. Used to modify the AST
- *start* : Start parsing an identifier name. Stored when name*(level) is performed
- *push* : Push the current AST node on to a stack to be recalled by pop or top
- *top* : Set the last AST node pushed to be the active AST node
- *pop* : Set the last AST node pushed to be the active AST node and remove it from the stack
- *nav.N* : Navigate to the N^{th} child and set it as the active node
- *nav.p* : Navigate to the parent and set it as the active node

A diagram of a section of the FSA is provided here 3.2 as an example. This example is used to describe a parsing of the REAL program `x:int+y`.

Figure 3.2: Section of deterministic FSA used by REAL



Below is a decription of each step through diagram in figure 3.2 to parse `x:int+y`. The parser begins in state `s0`.

1. start state `s0`, read input symbol `x`
2. `x` matches `ELSE`, go to `e0`
3. `x` matches `_A_` (alphabetic characters), go to `i0`
4. `:` matches `ELSE`, go to `o0`
5. transition `ELSE` constructs AST node Variable(11) 3.3 (A)

6. `:` matches `:`, go to `v0`
7. transition `:` constructs AST node ConvertType(10). Level(10) is less than active(11) node, so ConvertType becomes parent 3.3 (B)
8. `i` matches `_A_`, go to `v1`
9. `n` matches `_AN_` (alphanumeric characters), go to `v1`
10. `t` matches `_AN_`, go to `v1`
11. + matches `ELSE`, go to `o0`
12. transition `ELSE` constructs AST node Type(11). Level(11) is more than active node, so Type becomes child 3.3 (C)
13. + matches +, go to `e0`
14. transition + constructs AST node BopPlus(3). Level(3) is less than active (11) and parent(10) nodes, so Bop becomes grandparent 3.3 (D)
15. `y` matches `_A_`, go to `i0`
16. `EOF` matches `ELSE`, go to `o0`
17. transition `ELSE` constructs AST node Variable(11). Level(11) is more than active node, so Variable becomes a child 3.4
18. `EOF` matches `ELSE`, return from gosub, go to `s1`
19. `EOF` matches `EOF`, go to `s2`
20. final state `s2` completes the successful parsing

Figure 3.3: Building AST from x:int+y

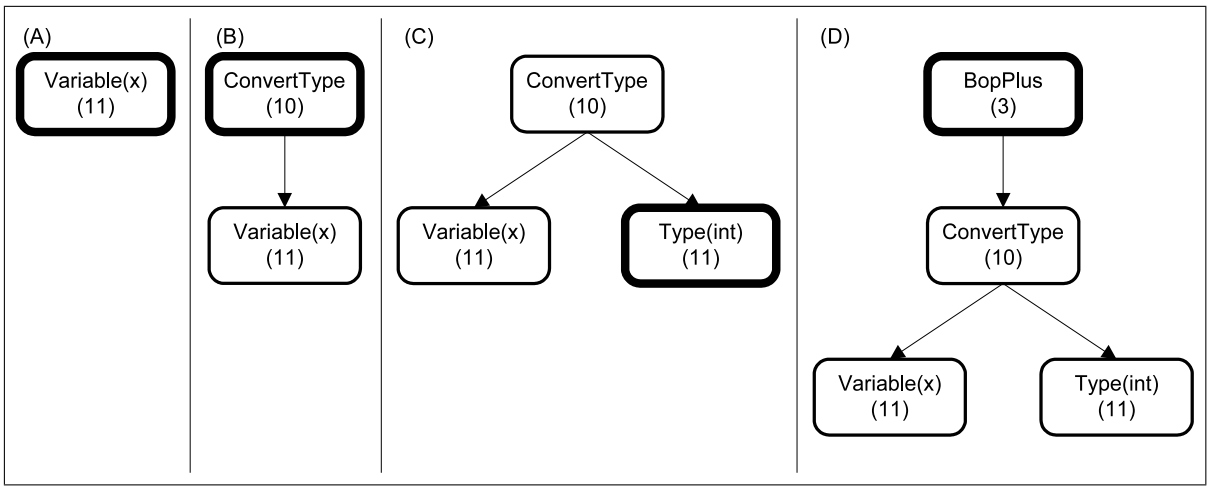


Figure 3.4: Complete AST from x:int+y

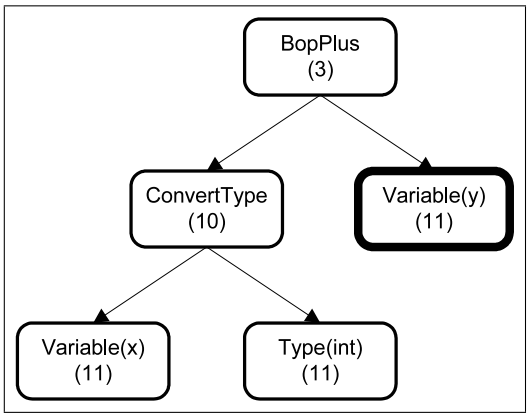


Figure 3.5: Deterministic FSA used by REAL

```
start                    @= s0
final                    @= s2
s0, $                    -> e0:s1
s1, #                    -> s2
e0, [afIitw]_!?a-zA-Z    -> i0   , start
e0, ~                    -> l0   , UopLevel(8)
e0, 0-9                  -> n0   , start
e0, .                    -> d0   , start
e0, })                   ->      , Unexpected bracket at this time
e0, ;                    -> e0   , top
e0, +                    -> e0   , UopPlus(8)
e0, \-                   -> e0   , UopMinus(8)
e0, (                    -> e0:b0, Bracket(-1), push
e0, {                    -> e0:b1, StatementList(-1), push
e0, "                    -> c0:o0, start, whitespace, parse
e0, :                    -> t0   ,
e0, assert, _!?a-zA-Z0-9 => A0,i0, start, Assert(-1), push
e0, if, _!?a-zA-Z0-9     => I0,i0, start; If(-1), push
e0, while, _!?a-zA-Z0-9  => W0,i0, start; While(-1), push
e0, true, _!?a-zA-Z0-9   => o0,i0, start; LiteralTrue(11)
e0, false, _!?a-zA-Z0-9  => o0,i0, start; LiteralFalse(11)
e0, ID, _!?a-zA-Z0-9     => o0,i0, start; LiteralID(11)
e0, $                    ->   :
t0, _!?a-zA-Z            -> t1   , start
t1, _!?a-zA-Z0-9         -> t1
t1, =                    -> t2   , DeclareType*(10)
t2, {                    -> t3   , DeclareTypeList(-1), push
t3, _!?a-zA-Z            -> t4   , start
t4, _!?a-zA-Z0-9         -> t4
t4, $                    -> t5   , IdentifierNew*(11)
t5, :                    -> t6   , DeclareTypeType(i10)
t6, _!?a-zA-Z            -> t7   , start
```

```
t7, _!?a-zA-Z0-9        -> t7
t7, $                   -> t8   , IdentifierType*(11)
t8, ;                   -> t3   , top
t8, }                   -> e0   , pop
n0, 0-9                 -> n0
n0, .                   -> d0
n0, $                   -> o0   , LiteralInt*(11)
d0, 0-9                 -> d1
d1, 0-9                 -> d1
d1, $                   -> o0   , LiteralFloat*(11)
c0, $                   -> c1   , start
c1, [\\\\"]*            -> c1
c1, "                   ->   :  , LiteralString*(11), whitespace, ignore
c1, \\                  -> c2
c2, "                   -> c1
c2, \\                  -> c1
b0, )                   -> o0   , pop
b1, }                   -> o0   , pop
o0, +                   -> e0   , BopPlus(i3)
o0, \-                  -> o2
o0, \*                  -> e0   , BopMultiply(i5)
o0, /                   -> e0   , BopDivide(i5)
o0, ^                   -> e0   , BopPower(i7)
o0, \\                  -> e0   , BopRoot(i7)
o0, \%                  -> e0   , BopModulus(i4)
o0, |                   -> e0   , BopLog(i6)
o0, =                   -> o1
o0, <                   -> o3
o0, >                   -> o4
o0, :                   -> v0   , ConvertType(i10)
o0, ;                   -> e0   , top
o0, .                   -> o5   , Dereference(9)
o0, $                   ->   :
o1, =                   -> e0   , CopEqual(1)
o1, $                   -> e0   , Assignment(0)
o2, =                   -> e0   , CopNotEqual(1)
o2, $                   -> e0   , BopMinus(i3)
o3, =                   -> e0   , CopLEqual(2)
o3, $                   -> e0   , CopLess(2)
o4, =                   -> e0   , CopGEqual(2)
o4, $                   -> e0   , CopGreat(2)
o5, _!?a-zA-Z           -> i0   , start
l0, ~                   -> l0   , UopLevel(8)
l0, _!?a-zA-Z           -> i0   , start
i0, _!?a-zA-Z0-9        -> i0
i0, (                   -> f0   , IdentifierFunction*(11), ParamList(-1), push
i0, $                   -> o0   , IdentifierVariable*(11)
f0, )                   -> o0   , pop
f0, $                   -> e0:f1, Param(-1), push
f1, ,                   -> f2   , pop
```

```
f1, )                   -> o0   , pop, pop
f2, $                   -> e0:f1, Param(-1), push
f2, ,                   ->      , Unexpected comma at this time
A0, (                   -> e0:A1, Bracket(-1), push
A1, )                   -> o0   , pop, pop
I0, (                   -> e0:I1, Bracket(-1), push
I1, )                   -> I2   , pop
I2, {                   -> e0:I3, StatementList(-1), push
I3, }                   -> I4   , pop
I4, else                => I5,I9
I4, $                   -> o0   , pop
I5, if                  => I0,I9, If(-1)
I5, $                   -> I6
I6, {                   -> e0:I7, StatementList(-1), push
I7, }                   -> o0   , pop, pop
I9, $                   ->      , else expected at this time
W0, (                   -> e0:W1, Bracket(-1), push
W1, )                   -> W2   , pop
W2, {                   -> e0:W3, StatementList(-1), push
W3, }                   -> o0   , pop, pop
v0, _!?a-zA-Z           -> v1   , start
v1, _!?a-zA-Z0-9        -> v1
v1, (                   -> p0:v2, IdentifierType*(11), DeclareParamList(-1), push
v1, =                   -> v9   , IdentifierType*(11)
v1, $                   -> o0   , IdentifierType*(11)
v2, )                   -> v3   , pop
v3, =                   -> v4   , DeclareFunction(0), nav.0 \
                                , DeclareFunctionType(o10), nav.0 \
                                , IdentifierNew*(o11), nav.p, nav.1 \
                                , IdentifierNewType*(o11), nav.p, nav.p
v4, {                   -> e0:v5, StatementList(-1), push
v5, }                   -> v6   , pop
v6, ;                   -> e0   , top
v6, #                   -> s2
v9, =                   -> e0   , CopEqual(1)
v9, $                   -> e0   , DeclareVariable(0), nav.0 \
                                , DeclareVariableType(o10), nav.0 \
                                , IdentifierNew*(o11), nav.p, nav.p
p0, _!?a-zA-Z           -> p1   , DeclareParam(-1), start, push
p0, $                   ->   :
p1, _!?a-zA-Z0-9        -> p1
p1, $                   -> p2   , IdentifierNew*(11)
p2, :                   -> p3   , DeclareParamType(i10)
p3, _!?a-zA-Z           -> p4   , start
p4, _!?a-zA-Z0-9        -> p4
p4, ,                   -> p5   , IdentifierType*(11), pop
p4, $                   -> p0   , IdentifierType*(11), pop
p5, _!?a-zA-Z           -> p1   , start, DeclareParam(-1), push
```

3.4 BNF & AST

Figure 3.6 describes REAL's AST in Backus-Naur Form [11] (BNF).

Figure 3.6: BNF used by REAL

```
<Program>               ::= <StatementList>
<StatementList>         ::= <Statement>
                          | <Statement> ; <StatementList>
<Statement>             ::= <DeclareVariable>
                          | <DeclareFunction>
                          | <Assignment>
                          | <Expression>
<Expression>            ::= true
                          | false
                          | ID
                          | <INTEGER>
                          | <FLOAT>
                          | <STRING>
                          | { <StatementList> }
                          | <Bracket>
                          | <BopPlus>
                          | <BopMinus>
                          | <BopMultiply>
                          | <BopDivide>
                          | <BopPower>
                          | <BopRoot>
                          | <BopModulus>
                          | <BopLog>
                          | <UopPlus>
                          | <UopMinus>
                          | <UopLevel>
                          | <CopEqual>
                          | <CopNotEqual>
                          | <CopGEqual>
                          | <CopGreat>
                          | <CopLEqual>
                          | <CopLess>
                          | <ConvertType>
                          | <IdentifierVariable>
                          | <IdentifierFunction>
                          | <Assert>
                          | <If>
                          | <While>
<DeclareVariable>       ::= <DeclareVariableType> = <Expression>
<DeclareVariableType>   ::= <IdentifierNew> : <IdentifierType>
<DeclareFunction>       ::= <DeclareFunctionType> { <StatementList> }
<DeclareFunctionType>   ::= <IdentifierNew> : <IdentifierNewType>
<IdentifierNewType>     ::= <IDENTIFIER> ( )
```

```
                          | <IDENTIFIER> ( <DeclareParamList> )
<DeclareParamList>      ::= <DeclareParam>
                          | <DeclareParam>, <DeclareParamList>
<DeclareParam>          ::= <DeclareParamType>
<DeclareParamType>      ::= <IdentifierNew> : <IdentifierType>
<Assignment>            ::= <IdentifierVariable> = <Expression>
<Bracket>               ::= ( <Expression> )
<BopPlus>               ::= <Expression> + <Expression>
<BopMinus>              ::= <Expression> - <Expression>
<BopMultiply>           ::= <Expression> * <Expression>
<BopDivide>             ::= <Expression> / <Expression>
<BopPower>              ::= <Expression> ^ <Expression>
<BopRoot>               ::= <Expression> \ <Expression>
<BopModulus>            ::= <Expression> % <Expression>
<BopLog>                ::= <Expression> | <Expression>
<UopPlus>               ::= + <Expression>
<UopMinus>              ::= - <Expression>
<UopLevel>              ::= ~ <UopLevel>
                          | ~ <IdentifierVariable>
                          | ~ <IdentifierFunction>
<CopEqual>              ::= <Expression> == <Expression>
<CopNotEqual>           ::= <Expression> -= <Expression>
<CopGEqual>             ::= <Expression> >= <Expression>
<CopGreat>              ::= <Expression> > <Expression>
<CopLEqual>             ::= <Expression> <= <Expression>
<CopLess>               ::= <Expression> < <Expression>
<ConvertType>           ::= <Expression> : <IdentifierType>
<Assert>                ::= assert <Bracket>
<If>                    ::= if <Bracket> { <StatementList> }
                          | if <Bracket> { <StatementList> }
                            else { <StatementList> }
<While>                 ::= while <Bracket> { <StatementList> }
<IdentifierNew>         ::= <IDENTIFIER>
<IdentifierType>        ::= <IDENTIFIER>
<IdentifierVariable>    ::= <IDENTIFIER>
<IdentifierFunction>    ::= <IDENTIFIER> ( )
                          | <IDENTIFIER> ( <ParamList> )
<ParamList>             ::= <Param>
                          | <Param> , <ParamList>
<Param>                 ::= <Expression>
<INTEGER>               ::= [0-9]
<FLOAT>                 ::= [0-9]*.[0-9]+
<STRING>                ::= "(.|\"|\\)*"
<IDENTIFIER>            ::= [a-zA-Z!?_][a-zA-Z0-9!?_]*
```

3.5 Type Checker

Figure 3.7 describes REAL's type semantics using SOS.

Figure 3.7: Type semantics used by REAL

Assert

$$\frac{\Gamma \vdash e{:}bool}{\Gamma \vdash assert(e){:}bool}$$

Assignment

$$\frac{\Gamma, T_0, T_1 \vdash T_1 = T_0}{\Gamma \vdash (i{:}T_0 = e{:}T_1){:}T_0}$$

BinaryOp1

$$\frac{\Gamma, T_0, T_1, T_2 \vdash T_2 = T_0 = T_1}{\Gamma \vdash (e_0{:}T_0 \odot e_1{:}T_1){:}T_2} \quad \odot = \{+\}$$

BinaryOp2

$$\frac{\Gamma, T_0, T_1, T_2 \vdash T_2 = T_0 = T_1}{\Gamma \vdash (e_0{:}T_0 \odot e_1{:}T_1){:}T_2} \quad \odot = \{-\} \quad (T_0, T_1) = \{\text{int,float,string}\}$$

BinaryOp3

$$\frac{\Gamma, T_0, T_1, T_2 \vdash T_2 = T_0 = T_1}{\Gamma \vdash (e_0{:}T_0 \odot e_1{:}T_1){:}T_2} \quad \odot = \{*, \hat{}\} \quad (T_0, T_1) = \{\text{bool,int,float}\}$$

BinaryOp4

$$\frac{\Gamma, T_0, T_1, T_2 \vdash T_2 = T_0 = T_1}{\Gamma \vdash (e_0{:}T_0 \odot e_1{:}T_1){:}T_2} \quad \odot = \{/, \%, \backslash, |\} \quad (T_0, T_1) = \{\text{int,float}\}$$

Bracket

$$\frac{\Gamma, T \vdash e{:}T}{\Gamma \vdash (e){:}T}$$

ConvertType

$$\frac{\Gamma, T_0, T_1 \vdash \diamond}{\Gamma \vdash e{:}T_0 \mapsto T_1} \quad (T_0, T_1) = \{\text{bool,int,float,string}\}$$

CompareOp1

$$\frac{\Gamma, T_0, T_1 \vdash T_0 = T_1}{\Gamma \vdash (e_0{:}T_0 \odot e_1{:}T_1){:}bool} \quad \odot = \{==, -=\}$$

CompareOp2

$$\frac{\Gamma, T_0, T_1 \vdash T_0 = T_1}{\Gamma \vdash (e_0{:}T_0 \odot e_1{:}T_1){:}bool} \quad \odot = \{¡, ¡=, ¿, ¿=\} \quad (T_0, T_1) \neq \{\text{bool}\}$$

DeclareFunction

$$\frac{\Gamma, f, T_0, ..., T_N, R \vdash \diamond}{\Gamma \vdash f{:}R(i_0{:}T_0, ..., i_N{:}T_N){:}R} \quad f = T_0 \mapsto \ldots \mapsto T_N \mapsto R$$

DeclareParam

$$\frac{\Gamma, T \vdash \diamond}{\Gamma \vdash e{:}T}$$

DeclareParamList

$$\frac{\Gamma, T_0, ..., T_N \vdash \diamond}{\Gamma \vdash (i_0{:}T_0, ..., i_N{:}T_N){:}f} \quad f = T_0 \mapsto \ldots \mapsto T_N$$

DeclareParamType

$$\frac{\Gamma, T, i \twoheadrightarrow i_{loc} | \sigma_0 \vdash \diamond \quad \Gamma, T, i \rightarrow i_{loc} | \sigma_1 \{T / i_{loc}\} \vdash \diamond}{\Gamma | \sigma_0 \vdash f(i{:}T) = \{e | \sigma_1\}} \quad Fresh \quad i_{loc} | \sigma_1$$

DeclareFunctionType

$$\frac{\Gamma, T \vdash \diamond}{\Gamma \vdash e{:}T}$$

DeclareVariableType

$$\frac{\Gamma, T, i \twoheadrightarrow i_{loc} | \sigma_0 \vdash \diamond \quad \Gamma, T, i \rightarrow i_{loc} | \sigma_1 \{T / i_{loc}\} \vdash \diamond}{\Gamma | \sigma_0 \vdash [e_0; ...; i{:}T; ...; e_N] | \sigma_1} \quad Fresh \quad i_{loc} | \sigma_1$$

DeclareVariable

$$\frac{\Gamma, T_0, T_1 \vdash T_0 = T_1}{\Gamma \vdash i{:}T_0 = e{:}T_1}$$

IdentifierFunction

$$\frac{\Gamma, T, i \rightarrow i_{loc} | \sigma \vdash \sigma\{i_{loc}\} \rightsquigarrow [T_0 \rightarrow \ldots \rightarrow T_N \rightarrow R] \quad \Gamma \vdash T_0 = t_0, ..., T_n = t_n}{\Gamma | \sigma \vdash i(t_0, ..., t_n){:}R}$$

IdentifierNew

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \diamond}$$

IdentifierNewType

$$\frac{\Gamma, T, i \rightarrow i_{loc} | \sigma \vdash \diamond}{\Gamma, i \rightarrow i_{loc} | \sigma \vdash \sigma\{i_{loc}\} | \sigma}$$

IdentifierType

$$\frac{\Gamma, T, i \rightarrow i_{loc} | \sigma \vdash \diamond}{\Gamma, i \rightarrow i_{loc} | \sigma \vdash \sigma\{i_{loc}\} | \sigma}$$

IdentifierVariable

$$\frac{\Gamma, i \rightarrow i_{loc} | \sigma \vdash \diamond}{\Gamma, i \rightarrow i_{loc} | \sigma \vdash i{:}\sigma\{i_{loc}\} | \sigma}$$

If

$$\frac{\Gamma, R \vdash e{:}bool}{\Gamma \vdash if(e)\{s{:}R\}{:}R}$$

IfEl

$$\frac{\Gamma, R \vdash e{:}bool}{\Gamma \vdash if(e{:}bool)\{s{:}R\} else\{s{:}R\}{:}R}$$

LiteralFalse

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash false{:}bool}$$

LiteralFloat

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash N{:}float}$$ N=32bit IEEE float

LiteralID

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \diamond}$$

LiteralInt

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash N{:}int}$$ N=32bit signed integer

LiteralString

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash A{:}string}$$ A=quoted string

LiteralTrue

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash true{:}bool}$$

Param

$$\frac{\Gamma,T\vdash\diamond}{\Gamma\vdash e{:}T}$$

ParamList

$$\frac{\Gamma,T_0,...,T_N\vdash\diamond}{\Gamma\vdash(i_0{:}T_0,...,i_N{:}T_N){:}f}\quad f = T_0\mapsto\ldots\mapsto T_N$$

Program

$$\frac{\Gamma,T,T\to T_{loc}|\sigma\vdash\diamond}{\Gamma|\sigma\{T/T_{loc}\}\vdash\diamond}\quad T = \{\text{bool,int,float,string}\}\quad \text{Fresh}T_{loc}$$

StatementList

$$\frac{\Gamma,T_0,...,T_N\vdash\diamond}{\Gamma\vdash\{e_0{:}T_0;...;e_N{:}T_N\}{:}T_N}$$

UopLevel

$$\frac{\Gamma,T\vdash\diamond}{\Gamma\vdash(\sim(e{:}T)){:}T}$$

UopMinus

$$\frac{\Gamma,T\vdash\diamond}{\Gamma\vdash(-(e{:}T)){:}T}\quad T\neq\{\text{string}\}$$

UopPlus

$$\frac{\Gamma,T\vdash\diamond}{\Gamma\vdash(+(e{:}T)){:}T}\quad T\neq\{\text{bool,string}\}$$

While

$$\frac{\Gamma,R\vdash e{:}bool}{\Gamma\vdash while(e)\{s{:}R\}{:}R}$$

3.6 Operational Semantics

Figure 3.8 describes REAL's operational semantics using SOS.

Figure 3.8: Operational semantics used by REAL

Assert

$$\frac{\Gamma \vdash e \rightsquigarrow ID}{\Gamma \vdash assert(e) \rightsquigarrow ID}$$

$$\frac{\Gamma \vdash e \rightsquigarrow true}{\Gamma \vdash assert(e) \rightsquigarrow true}$$

$$\frac{\Gamma \vdash e \rightsquigarrow false}{\Gamma \vdash assert(e) \rightsquigarrow \bot}$$

Assignment

$$\frac{\Gamma, i \rightarrow i_{loc} | \sigma \vdash e \rightsquigarrow v | \sigma}{\Gamma | \sigma \vdash i = e \rightsquigarrow v | \sigma\{v / i_{loc}\}}$$

BopID (All Bop* Rules)

$$\frac{\Gamma \vdash e_0 \rightsquigarrow ID \ \Gamma \vdash e_1 \rightsquigarrow v_1}{\Gamma \vdash e_0 \odot e_1 \rightsquigarrow v_1}$$

$$\frac{\Gamma \vdash e_0 \rightsquigarrow v_0 \ \Gamma \vdash e_1 \rightsquigarrow ID}{\Gamma \vdash e_0 \odot e_1 \rightsquigarrow v_0}$$

$$\frac{\Gamma \vdash e_0 \rightsquigarrow ID \ \Gamma \vdash e_1 \rightsquigarrow ID}{\Gamma \vdash e_0 \odot e_1 \rightsquigarrow ID}$$

CopID (All Cop* Rules)

$$\frac{\Gamma \vdash e_0 \rightsquigarrow ID \vee e_1 \rightsquigarrow ID}{\Gamma \vdash e_0 \odot e_1 \rightsquigarrow ID}$$

BopDivide

$$\frac{\Gamma \vdash e_0 \rightsquigarrow v_0 : T \ \Gamma \vdash e_1 \rightsquigarrow v_1 : T}{\Gamma \vdash e_0 / e_1 \rightsquigarrow div(v_0, v_1)} \quad T = \{int, float\}$$

BopLog

$$\frac{\Gamma \vdash e_0 \rightsquigarrow v_0 : T \ \Gamma \vdash e_1 \rightsquigarrow v_1 : T}{\Gamma \vdash e_0 | e_1 \rightsquigarrow div(log(v_0), log(v_1))} \quad T = \{int, float\}$$

BopMinus

$$\frac{\Gamma \vdash e_0 \rightsquigarrow v_0 : T \ \Gamma \vdash e_1 \rightsquigarrow v_1 : T}{\Gamma \vdash e_0 - e_1 \rightsquigarrow sub(v_0, v_1)} \quad T = \{int, float, string\}$$

BopModulus

$$\frac{\Gamma \vdash e_0 \rightsquigarrow v_0 : T \ \Gamma \vdash e_1 \rightsquigarrow v_1 : T}{\Gamma \vdash e_0 \% e_1 \rightsquigarrow mod(v_0, v_1)} \quad T = \{int, float\}$$

BopMultiply

$$\frac{\Gamma \vdash e_0 \rightsquigarrow v_0 : T \ \Gamma \vdash e_1 \rightsquigarrow v_1 : T}{\Gamma \vdash e_0 * e_1 \rightsquigarrow and(v_0, v_1)} \quad T = \{bool\}$$

$$\frac{\Gamma \vdash e_0 \rightsquigarrow v_0 : T \ \Gamma \vdash e_1 \rightsquigarrow v_1 : T}{\Gamma \vdash e_0 * e_1 \rightsquigarrow mul(v_0, v_1)} \quad T = \{int, float\}$$

BopPlus

$$\frac{\Gamma \vdash e_0 \rightsquigarrow v_0 : T \ \Gamma \vdash e_1 \rightsquigarrow v_1 : T}{\Gamma \vdash e_0 + e_1 \rightsquigarrow or(v_0, v_1)} \quad T = \{bool\}$$

$$\frac{\Gamma \vdash e_0 \rightsquigarrow v_0 : T \ \Gamma \vdash e_1 \rightsquigarrow v_1 : T}{\Gamma \vdash e_0 + e_1 \rightsquigarrow add(v_0, v_1)} \quad T = \{int, float\}$$

$$\frac{\Gamma \vdash e_0 \rightsquigarrow v_0 : T \ \Gamma \vdash e_1 \rightsquigarrow v_1 : T}{\Gamma \vdash e_0 + e_1 \rightsquigarrow cat(v_0, v_1)} \quad T = \{string\}$$

BopPower

$$\frac{\Gamma \vdash e_0 \rightsquigarrow v_0 : T \ \Gamma \vdash e_1 \rightsquigarrow v_1 : T}{\Gamma \vdash e_0 \hat{} e_1 \rightsquigarrow xor(v_0, v_1)} \quad T = \{bool\}$$

$$\frac{\Gamma \vdash e_0 \rightsquigarrow v_0 : T \ \Gamma \vdash e_1 \rightsquigarrow v_1 : T}{\Gamma \vdash e_0 \hat{} e_1 \rightsquigarrow pow(v_0, v_1)} \quad T = \{int, float\}$$

BopRoot

$$\frac{\Gamma \vdash e_0 \rightsquigarrow v_0 : T \ \Gamma \vdash e_1 \rightsquigarrow v_1 : T}{\Gamma \vdash e_0 \backslash e_1 \rightsquigarrow pow(v_0, div(1, v_1))} \quad T = \{int, float\}$$

Bracket

$$\frac{\Gamma \vdash e \rightsquigarrow v}{\Gamma \vdash (e) \rightsquigarrow v}$$

ConvertType

$$\frac{\Gamma \vdash e \rightsquigarrow v}{\Gamma \vdash e : T \rightsquigarrow v}$$

CopEqual

$$\frac{\Gamma \vdash e_0 \rightsquigarrow v_0 : T \ \Gamma \vdash e_1 \rightsquigarrow v_1 : T}{\Gamma \vdash e_0 == e_1 \rightsquigarrow v_0 = v_1} \quad T = \{bool, int, float, string\}$$

CopGEqual

$$\frac{\Gamma \vdash e_0 \rightsquigarrow v_0 : T \ \Gamma \vdash e_1 \rightsquigarrow v_1 : T}{\Gamma \vdash e_0 \text{¿} = e_1 \rightsquigarrow v_0 \geq v_1} \quad T = \{int, float, string\}$$

CopGreat

$$\frac{\Gamma \vdash e_0 \rightsquigarrow v_0 : T \ \Gamma \vdash e_1 \rightsquigarrow v_1 : T}{\Gamma \vdash e_0 \text{¿} e_1 \rightsquigarrow v_0 > v_1} \quad T = \{int, float, string\}$$

CopLEqual

$$\frac{\Gamma \vdash e_0 \rightsquigarrow v_0 : T \ \Gamma \vdash e_1 \rightsquigarrow v_1 : T}{\Gamma \vdash e_0 \text{¡} = e_1 \rightsquigarrow v_0 \leq v_1} \quad T = \{int, float, string\}$$

CopLess

$$\frac{\Gamma \vdash e_0 \rightsquigarrow v_0 : T \ \Gamma \vdash e_1 \rightsquigarrow v_1 : T}{\Gamma \vdash e_0 \text{¡} e_1 \rightsquigarrow v_0 < v_1} \quad T = \{int, float, string\}$$

CopNotEqual

$$\frac{\Gamma \vdash e_0 \rightsquigarrow v_0 : T \ \Gamma \vdash e_1 \rightsquigarrow v_1 : T}{\Gamma \vdash e_0 \text{-} = e_1 \rightsquigarrow v_0 \neq v_1} \quad T = \{bool, int, float, string\}$$

DeclareFunction

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash i() = \{e\} \rightsquigarrow ID}$$

DeclareParam

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash i \rightsquigarrow ID}$$

DeclareParamList
$$\frac{\Gamma\vdash\diamond}{\Gamma\vdash\diamond}$$

DeclareParamType
$$\frac{\Gamma\vdash\diamond}{\Gamma\vdash i\rightsquigarrow i}$$

DeclareFunctionType
$$\frac{\Gamma\vdash\diamond}{\Gamma\vdash T\rightsquigarrow ID}$$

DeclareVariableType
$$\frac{\Gamma\vdash\diamond}{\Gamma\vdash T\rightsquigarrow T}$$

DeclareVariable
$$\frac{\Gamma,i\rightarrow i_{loc}|\sigma_0\vdash e\rightsquigarrow v\ \Gamma,i\rightarrow i_{loc}|\sigma_1\vdash\sigma_1\{v/i_{loc}\}\vdash\diamond}{\Gamma|\sigma_0\vdash\{i{=}e;[e_0;...;e_N]\}|\sigma_1\}}\ \text{Fresh}\ \ i_{loc}$$

IdentifierFunction
$$\frac{\Gamma,i\rightarrow i_{loc}|\sigma\vdash\diamond}{\Gamma|\sigma\vdash i()\rightsquigarrow\sigma\{i_{loc}\}|\sigma}$$

IdentifierNew
$$\frac{\Gamma\vdash\diamond}{\Gamma\vdash i\rightsquigarrow i}$$

IdentifierNewType
$$\frac{\Gamma\vdash\diamond}{\Gamma\vdash T\rightsquigarrow ID}$$

IdentifierType
$$\frac{\Gamma\vdash\diamond}{\Gamma\vdash T\rightsquigarrow ID}$$

IdentifierVariable
$$\frac{\Gamma,i\rightarrow i_{loc}|\sigma\vdash\diamond}{\Gamma|\sigma\vdash i\rightsquigarrow\sigma\{i_{loc}\}|\sigma}$$

If
$$\frac{\Gamma\vdash e\rightsquigarrow true\ \Gamma\vdash s_0\rightsquigarrow v_0}{\Gamma\vdash if(e)\{s_0\}else\{s_1\}\rightsquigarrow v_0}$$

$$\frac{\Gamma\vdash e\rightsquigarrow false\ \Gamma\vdash s_1\rightsquigarrow v_1}{\Gamma\vdash if(e)\{s_0\}else\{s_1\}\rightsquigarrow v_1}$$

$$\frac{\Gamma\vdash e\rightsquigarrow false}{\Gamma\vdash if(e)\{s_0\}\rightsquigarrow ID}$$

$$\frac{\Gamma\vdash e\rightsquigarrow ID}{\Gamma\vdash if(e)\rightsquigarrow ID}$$

IfEl
$$\frac{\Gamma,R\vdash if\rightarrow R}{\Gamma\vdash if(e{:}bool)\{s{:}R\}else\{s{:}R\}{:}R}$$

LiteralFalse
$$\frac{\Gamma\vdash\diamond}{\Gamma\vdash false}$$

LiteralFloat
$$\frac{\Gamma\vdash\diamond}{\Gamma\vdash N}\ \text{N=32bit IEEE float}$$

LiteralID
$$\frac{\Gamma\vdash\diamond}{\Gamma\vdash ID}$$

LiteralInt
$$\frac{\Gamma\vdash\diamond}{\Gamma\vdash N}\ \text{N=32bit signed integer}$$

LiteralString
$$\frac{\Gamma\vdash\diamond}{\Gamma\vdash A}\ \text{A=quoted string}$$

LiteralTrue
$$\frac{\Gamma\vdash\diamond}{\Gamma\vdash true}$$

Param
$$\frac{\Gamma\vdash e\rightsquigarrow v}{\Gamma\vdash e\rightsquigarrow v}$$

ParamList
$$\frac{\Gamma\vdash e_0\rightsquigarrow v_0\ ...\ \Gamma\vdash e_N\rightsquigarrow v_N}{\Gamma\vdash(e_0,...,e_N)\rightsquigarrow[v_0,...,v_N]}$$

Program
$$\frac{\Gamma\vdash s_0\rightsquigarrow v_0\ ...\ \Gamma\vdash s_N\rightsquigarrow v_N}{\Gamma\vdash[s_0;...;s_N]\rightsquigarrow v_N}$$

StatementList
$$\frac{\Gamma\vdash s_0\rightsquigarrow v_0\ ...\ \Gamma\vdash s_N\rightsquigarrow v_N}{\Gamma\vdash\{s_0;...;s_N\}\rightsquigarrow v_N}$$

UopLevel
$$\frac{\Gamma,i\rightarrow i_{loc}|\sigma_{n-1}\vdash\diamond}{\Gamma|\sigma_n\vdash(\sim i)\rightsquigarrow i|\sigma_{n-1}}$$

UopMinus
$$\frac{\Gamma\vdash e\rightsquigarrow v{:}T}{\Gamma\vdash -e\rightsquigarrow not(v)}\ \ T=\{\text{bool}\}$$

$$\frac{\Gamma\vdash e\rightsquigarrow v{:}T}{\Gamma\vdash -e\rightsquigarrow -v}\ \ T=\{\text{int,float}\}$$

$$\frac{\Gamma\vdash e\rightsquigarrow ID}{\Gamma\vdash -e\rightsquigarrow ID}$$

UopPlus
$$\frac{\Gamma\vdash e\rightsquigarrow v{:}T}{\Gamma\vdash +e\rightsquigarrow abs(v)}\ \ T=\{\text{int,float}\}$$

$$\frac{\Gamma\vdash e\rightsquigarrow ID}{\Gamma\vdash +e\rightsquigarrow ID}$$

While
$$\frac{\Gamma\vdash e_0\rightsquigarrow true}{\Gamma\vdash while(e_0)\{s\}\rightsquigarrow while(e_1)\{s\}}$$

$$\frac{\Gamma\vdash e_0\rightsquigarrow false}{\Gamma\vdash while(e_0)\{s\}\rightsquigarrow ID}$$

$$\frac{\Gamma\vdash e_1\rightsquigarrow false\ s\rightsquigarrow v}{\Gamma\vdash while(e_1)\{s\}\rightsquigarrow v}$$

Chapter 4: **Implementation**

REAL is packaged into several major component. These packages are:

- *main* : An application to run REAL in
- *ui* : REAL's UI panel to be plugged into an application
- *parser* : Defines the deterministic FSA to parse REAL syntax
- *analyser* : Visitor that performs structure and type checking on the AST
- *interpreter* : Visitor that performs execution on the AST
- *utilities* : Additional visitors for debugging and calculating information about the AST
- *ast* : Defines all AST nodes
- *ast.environment* : Defines the environment in which the interpreter and type checker are run. This includes the binding store
- *ast.type* : Defines the basic and compound types
- *ast.visitor* : Defines the visitor pattern used to apply semantics to each AST node

REAL parses the input using a looped switch statement which mimics a deterministic FSA. All other processes in REAL are performed by applying the visitor pattern [4]. REAL executes a program by the following steps:

1. The parser is given the program and returns an AST
2. The type-checker visitor is applied to the AST and updates its type store
3. The interpreter visitor is applied to the AST and updates its binding store
4. The interpreter stores the final result of the program for later access

REAL's real-time UI executes the above actions on each key-stroke, adding the following steps:

1. After parsing, the AST is passed to a "focus" visitor which includes the cursor position
2. The focus visitor is applied to the AST and returns the node that is closest to the cursor position
3. The interpreter is given the "focus" AST node
4. The interpreter saves the program state after executing the "focus" node for later access

Using the source measuring tool JavaNCSS [13] the following complexity statistics are determined. NCSS (Non-Commenting Source Statements) is the number of executable statements. CCN (Cyclomatic Complexity Number) is the number of execution paths in the source. All functions have at least a value of one for CCN. Branching statements such as `if` or `switch` etc. add one to this count.

- *Packages* : 11
- *Classes* : 91
- *Functions* : 629
- *Average Functions per Class* : 6.91
- *NCSS* : 4453
- *Average NCSS* : 5.59
- *Average CCN* : 2.5

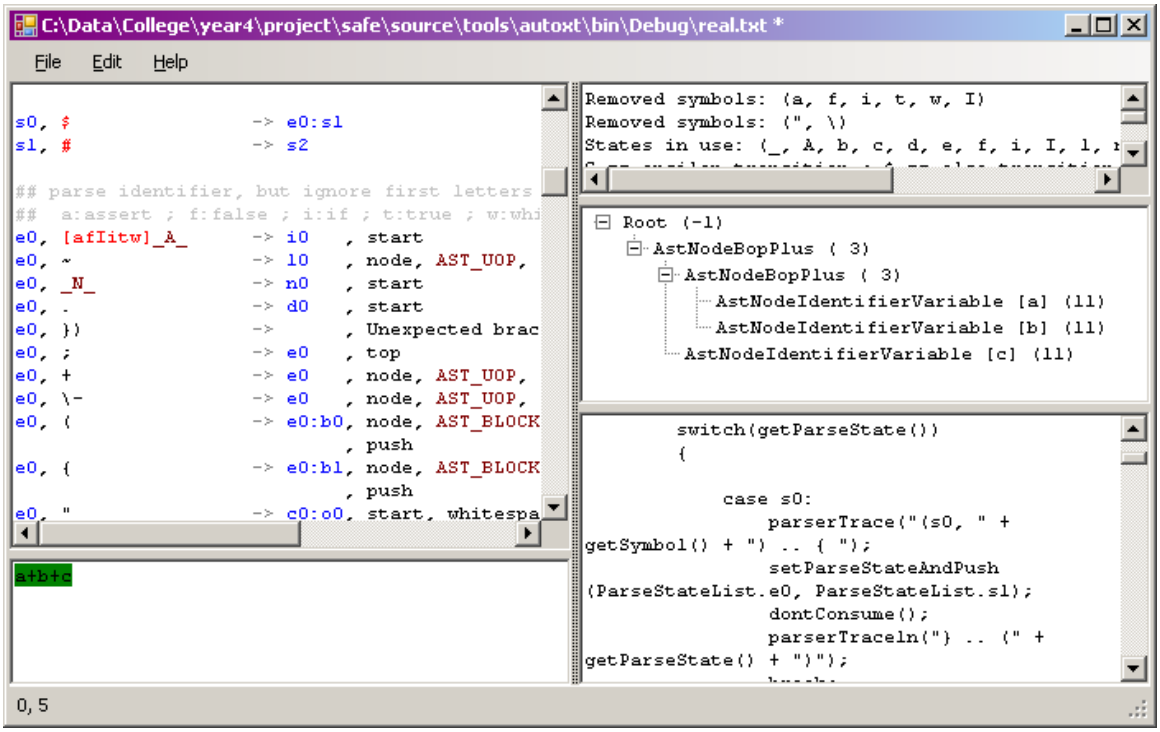
4.1 Parser

REAL is parsed using a form of push-down deterministic FSA [7]. Push-down automata push symbols on to a stack to later match those symbols. This allows brackets to be counted to ensure opened brackets are closed. REAL uses a modification of a push-down automata by means of a call stack. Instead of pushing symbols, REAL's parser pushes states. This allows REAL to break the FSA into procedures allowing the FSA definition to be simplified.

To support this parser, REAL includes a parser generator. This parser generator is a separate tool that constructs Java™ source code for parsing the input FSA. The interface for this tool is shown in figure 4.1. The interface is broken into five parts. On the top-left is an input for the FSA rules. On the bottom-left is an input where REAL programs can be written to test what AST they generate against the current FSA. On the top-right is an output log of the last parsing. On the middle-right is the AST generated from the REAL program against the FSA. On the bottom-right is the generated Java™ source code that performs the actions of the parser.

In keeping with the style of the project, the REAL parser generator also executes in real-time. This allows updates to be quickly tested, simplifying the task of designing the FSA.

Figure 4.1: Parser generator created for REAL



Chapter 5: **Testing**

Testing of REAL is done using a junit test case. This test consists of 829 hand-written mini programs. Each test includes with it an expected result. These tests cover all basic cases of REAL and build up to the minimum complex cases.

5.1 Testing results

Using the code coverage tool EMMA [12] tests are found to cover approximately 89.3 percent of the components used to compile and execute REAL programs. This testing does not cover the GUI components.

Figure 5.1 describes a list of errors uncovered by testing.

Figure 5.1: Errors in REAL uncovered by testing

- *Unexpected answers to compound expressions* : Expressions were being evaluated from right-to-left.
  - Corrected by introducing an "insert" action in the parser. Arithmetic operators are updated to use this "insert" action. This action causes child nodes to be inserted in reverse. Execution then runs from left-to-right
- *Unexpected exception* : Expressions such as `while(if(false){1}){a}` were failing. `if(false){1}` returns the value of the `else` expression. With no `else` expression, it returned nothing. `while` threw an exception not knowing what to do with the result.
  - Corrected by introducing an identity value. All undefined operations are given this identity value. All operational semantics are updated to account for identity as input.
- *Parameter count not tested* : Procedures defined could be called without giving the correct number of parameters in the call
  - Corrected by updating the type-checker to ensure parameter counts are correct
- *Redefinition allowed* : A variable defined twice would ignore the first definition and allow the second.
  - Corrected by updating the type-checker to ensure the type constructed doesn't already exist
- *Incorrect parsing allowed* : Specifying trailing commas in a procedure call or declaration was being parsed as successful
  - Corrected by updating the parser and disallow trailing commas
- *Scope operator error* : Specifying the scope operator on an l-value was throwing an exception
  - Corrected by updating the type and interpreter to allow the scope operator on an l-value

Chapter 6: **Conclusion**

In implementing the project, all mandatory requires are complete. Of the discretionary requirements, “reference types” and “concurrency” are not implemented. The discretionary requirement “subtypes” is only partially implemented but unusable. All other discretionary requirements are complete. ‘Compound types” is implemented in procedures. “Static check” is implemented in the type-checker. “Assertion validity” is implemented in the assert keyword. “Memory graph” is implemented in the “focus” output. The exceptional requirement is not done.

Bibliography

[1] The WHILE Language Syntax. http://pag.cs.uni-sb.de/while.html

[2] The WHILE Language IDE. http://www2.mta.ac.il/ tal/WIDE/

[3] Cardelli, L., (1991). *Typeful Programming. Formal Descriptions of Programming Concepts, E.J.Neuhold, M.Paul Eds., Springer-Verlag.* SRC Research Report 45, May 24, 1989. Revised January 1, 1993.

[4] Gamma, E. [et al.], (1994). *Design Patterns : elements of reusable object-oriented software.* Addison-Wesley professional computing series. ISBN: 0201633612.

[5] Grune, D., Jacobs C.J.H., (1990). *Parsing Techniques: A Practical Guide.* Ellis Horwood Series in Computers and Their Applications. ISBN: 0136514316.

[6] Hennessy, M., (1990) *The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics.* Selected chapters of the book by the same name. John Wiley & Sons, Inc.

[7] Lewis H.R., Papadimitriou C.H., (1981). *Elements of the Theory of Computation.* Prentice-Hall, Software Series. ISBN: 0132734176.

[8] MacLennan B.R., (1983). *Principles of Programming Languages: Design, Evaluation and Implementation.* Holt-Saunders International. ISBN: 4833701588.

[9] Plotkin, G.D., (2004). *A Structural Approach to Operational Semantics.* Journal of Logic and Algebraic Programming, Vol. 60–61, pp. 17-139.

[10] Schmidt, D.A., (1987). *Denotational Semantics: A Methodology for Language Development.* Wm. C. Brown Publishers. ISBN: 0697068498.

[11] Watt, D.A., (1991). *Programming Language Syntax and Semantics.* Prentice Hall International series in computer science. ISBN: 0137262663.

[12] *http://emma.sourceforge.net/*

[13] *http://www.kclee.de/clemens/java/javancss/*