A PROPOSED UNIFIED VISUAL PROGRAMMING LANGUAGE

By
NABOU DIENG
Bachelor of Science in Computer Science
Oklahoma State University
Stillwater, Oklahoma
2005

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2010

A PROPOSED UNIFIED VISUAL
PROGRAMMING LANGUAGE

Thesis Approved:

Dr. Blayne Mayfield

Thesis Adviser

Dr. John Chandler

Dr. Johnson Thomas

Dr. Mark Payton

Dean of the Graduate College

ACKNOWLEDGMENTS

The successful preparation of this research would not have been possible without the invaluable contribution of my adviser, Dr Blayne Mayfield, nor would it have been possible without the great educational experience received from the professors of the Computer Science department of Oklahoma State University. For these reasons I would like to express my deepest gratitude to the members of my committee: Dr Blayne Mayfield, Dr John Chandler and Dr Johnson Thomas. I also would like to extend my sincere thanks to Dr Mansur Samadzadeh and Dr Nohpill Park, who have helped me tremendously during my university curriculum.

I further would like to thank my mother Dr Khadijhatou Seck for her unconditional encouragements in my studies, my dear brother Ababacar Dieng for his absolute support and my brother Madior Dieng for always motivating me.

Last but not least, I would like to take this opportunity to acknowledge the support and understanding throughout my life of my late farther Dr Papa Amath Dieng, who is not going to read these lines but who would have been proud to see me reach this milestone.

Table of Contents

Chapter Page

LIST OF FIGURES

LIST OF TABLES

Chapter I

INTRODUCTION

1.1 Visual Programming Languages

Visual programming languages (VPLs) are the class of programming languages with which users build programs by manipulating visual objects. The semantics of the program are thus expressed by graphical tokens as opposed to textual tokens used in textual programming languages (TPLs), and visual programming constructs as opposed to textual programming constructs in TPLs.

Integrated development environments such as Microsoft Visual Studio are visual programming environments (VPEs), and the languages they support for development, such as Visual C#, are not VPLs, since all the tokens of these languages are textual.

It is important to note that the term visual programming language, as known today, refers to a hybrid language that lies between a pure TPL and a pure VPL. Pure VPLs might not be a practical alternative to TPLs.

The main goals of VPLs are defined by Burnett [1]. She states that the three goals of VPLs are: to make programming easier to understand for audiences other than programmers, to reduce error proneness when programming and to help users program faster.

1.2 Issues of Visual Programming Languages

The most successful of the currently-available VPLs are domain specific; such languages include LabView, used for industrial automation or instrument control, and OpenMusic, used for musical composition. The other uses of VPLs generally are limited to teaching or research. The main issue faced by VPLs is their limited ability to produce a complex program while preserving a reasonable level of readability and maintainability. These issues of scale are a result of the presentation of a visual program. Since the program has text and graphics, it is visually bulkier than a TPL. The fact that most VPLs do not have a static representation – that is, a complete (unabridged) representation of the program—introduces readability issues. A high level of abstraction should be attained without sacrificing details that aid in the understanding of a program, as a whole. Again, because programming in a VPL is synonymous with manipulating visual objects to build a program, the management of the screen area poses a problem in building large programs efficiently.

Another concern with VPLs is the visual presentation of proper documentation, so that it is in line with the graphical nature of VPLs, while at the same time not adding more visual clutter to the program.

The last issue addressed in Burnett's paper—as well as in this section—is the readability of VPL programs. For instance, VPLs developed with arrows to direct the flow of data, or to represent the notion of ‘next statement’ have the advantage of showing visually the different segments of a program that could be executed concurrently; however, reading such programs is often very difficult because of the clutter added by the arrows.

1.3 Objectives and limits of this research

The first objective of this research is to analyze the grokens (graphical tokens) and viprocons (visual programming constructs) of a few selected VPLs in order to identify how issues related to the scaling up of VPLs are addressed in those languages, and also to identify weaknesses that preferably should not be part of a VPL.

The second objective is to design a general purpose VPL that could be used for complex programs, so that these programs can be reviewed and maintained more effectively than similar programs written in the VPLs analyzed in the first part of this research. The design of this "Unified visual programming language" or UVPL focuses on the visual features that could contribute to better scalability in visual programming, by using the analysis that results from the first objective.

Because this research focuses on the visual aspect of VPLs and its implications on readability and maintainability rather than on performance, an interpreter or a compiler is not developed for UVPL.

1.4 The Approach

In an attempt to fulfill the first objective, some popular, general-purpose and domain-specific VPLs are analyzed. The analysis is based on principles of programming languages and on strategies used in VPLs. The results of this analysis are used as a starting point to design the grokens and viprocons of UVPL. The last phase of this research consists of implementing a test program in each of the selected VPLs and in UVPL in order to gather metrics that allow a conclusion to be drawn about the goals attained by UVPL.

1.5 Chapters overview

This thesis first presents a review of background and previous work in VPLs relevant to this study. Then, the methodology adopted to conduct the research—which ranges from the selection of VPLs used in this research to the comparison techniques of these languages with UVPL—is described. Following the chapter on methodology, the results chapter presents a comparison between the selected VPLs and UVPL, and the comparison is used to evaluate the goals achieved by UVPL.

Chapter II

REVIEW OF LITERATURE

2.1 Background of VPLs

Margaret Burnett, whose primary research focus is on end-user programming, presents a thorough description of VPLs and their motivation [1]. To begin with, she explains the essential differences between TPLs and VPLs. Her major point is that the semantics of a program in a TPL can be conveyed only through text, whereas in a VPL the semantics of a program are conveyed at multiple levels, such as text, graphics, color, animation, etc.

In her paper Burnett addresses the history of VPLs by describing the precursory works related to the development of *programming by demonstration* and *programming via executable flowcharts*. Even though these first attempts seem very interesting, these languages could not be scaled up for programs of more conventional size, therefore they were less useful than their TPL counterparts.

Later on, the designs of VPLs took a new direction, and research was oriented towards domain-specific VPLs. These systems proved to be more successful than the earlier ones, since the target was a single, specific domain. As a result, it became possible to narrow down the collection of visual artifacts, operations, data structures, etc. to just those entities that are needed for a particular domain.

In her research Burnett identifies four strategies that could help achieve the most important goals of VPL research, which are making programming more understandable to non-programmers, increasing productivity of programmers and increasing correctness of programs. The four strategies used to achieve these goals are:

Concreteness: getting away from abstractness. An example would be to display automatically the effects of a program on a variable as the program runs.

Directness: directly manipulating objects. As an example, instead of describing semantics to be applied to an object, the programmer specifies the semantics by directly manipulating the object.

Explicitness: directly stating aspects of semantics rather than inferring them. For instance, using edges in a dataflow to express explicitly the relationships between variables or actions, or to direct explicitly the flow of data.

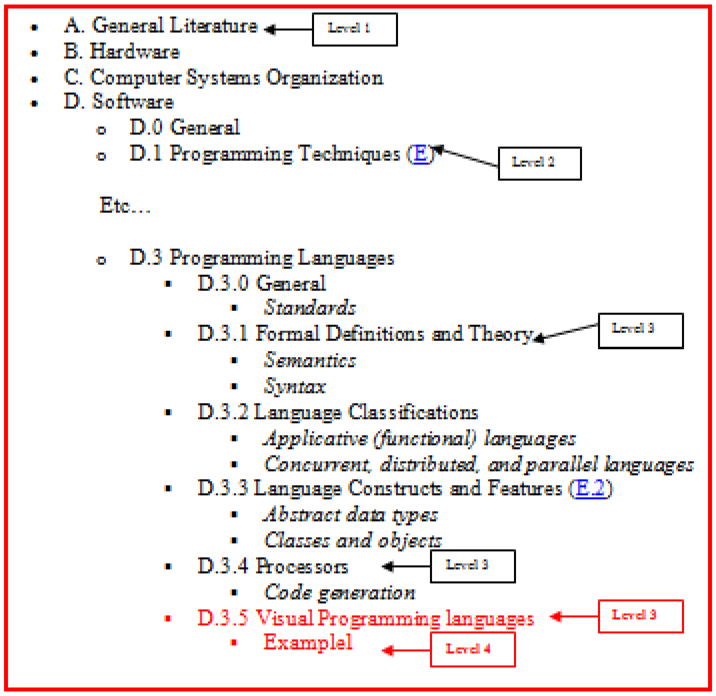
Immediate visual feedback: providing a livelier aspect of the programming experience. As programs are edited, the modifications to variables and objects are displayed automatically.

In her description of VPLs, Burnett also addresses the issue of abstraction in VPLs. The ability to reach some level of abstraction remains important, because it plays a major role in scalability. This statement is not in contradiction to strategy 1, because she refers here to the use of data and procedural abstraction, rather than the type of abstraction described in strategy 1. Data and procedural abstraction are possible in VPLs, since several current VPLs support these concepts. An example of procedural abstraction for VPL would be the ability to iconify a section of a dataflow. However, there is still room for improvement in this regard.

Among other important issues, Burnett discusses language specification for VPLs (this subject will be developed later in this chapter) and the cognitive dimension of VPLs, since the aim of these languages is to improve the programming experience of humans.

2.2 VPL Classification System

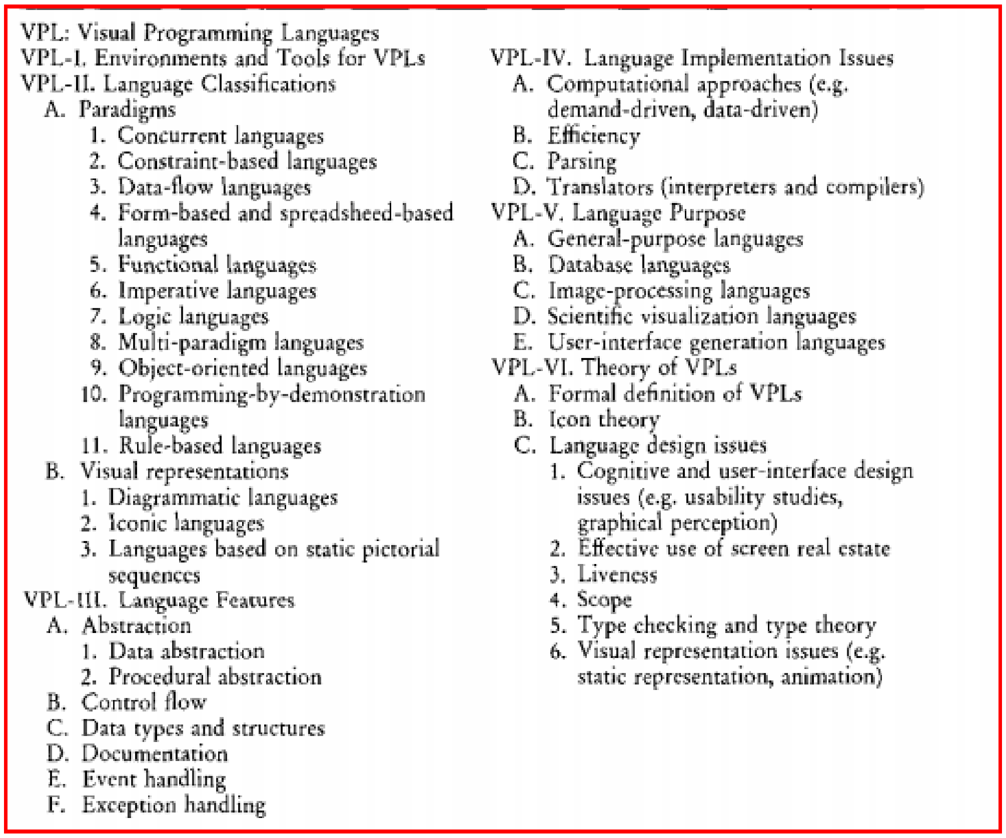
In 1993, Burnett and Baker proposed a classification system for visual programming languages [2]. As the literature of VPLs was broadening, they sensed that the development of a classification system to help researchers find the right material was a necessity. Although a similar computing reviews system already was designed by the Association for Computing Machinery (ACM), Burnett and Baker came to the conclusion that this system was not suitable for classifying VPLs. The ACM computing classification system is a four-level tree; placing VPLs under classification *D3 (Programming Languages)* would mean that only one more level could be added underneath VPLs. But, defining VPLs is more complex, and therefore, more than one subsection is needed to classify VPLs properly. However, for Burnett and Baker, this limitation could not satisfactorily classify the work in VPLs. Figure 2.2.1 shows an explanation of the levels in the ACM computing review system and the limitation for adding VPLs as a level-3 leaf in the tree, and figure 2.2.2 shows the classification of VPLs that Burnett and Baker proposed.


- A. General Literature ← Level 1
- B. Hardware
- C. Computer Systems Organization
- D. Software
    - D.0 General
    - D.1 Programming Techniques (E) ← Level 2

    Etc…

    - D.3 Programming Languages
        - D.3.0 General
            - *Standards*
        - D.3.1 Formal Definitions and Theory ← Level 3
            - *Semantics*
            - *Syntax*
        - D.3.2 Language Classifications
            - *Applicative (functional) languages*
            - *Concurrent, distributed, and parallel languages*
        - D.3.3 Language Constructs and Features (E.2)
            - *Abstract data types*
            - *Classes and objects*
        - D.3.4 Processors ← Level 3
            - *Code generation*
        - D.3.5 Visual Programming languages ← Level 3
            - Example1 ← Level 4

Figure 2.2.1: Example of adding VPLs to the ACM CR system. Modified from figure 2.2.2



VPL: Visual Programming Languages

VPL-I. Environments and Tools for VPLs

VPL-II. Language Classifications

- A. Paradigms
  1. Concurrent languages
  2. Constraint-based languages
  3. Data-flow languages
  4. Form-based and spreadsheed-based languages
  5. Functional languages
  6. Imperative languages
  7. Logic languages
  8. Multi-paradigm languages
  9. Object-oriented languages
  10. Programming-by-demonstration languages
  11. Rule-based languages
- B. Visual representations
  1. Diagrammatic languages
  2. Iconic languages
  3. Languages based on static pictorial sequences

VPL-III. Language Features

- A. Abstraction
  1. Data abstraction
  2. Procedural abstraction
- B. Control flow
- C. Data types and structures
- D. Documentation
- E. Event handling
- F. Exception handling

VPL-IV. Language Implementation Issues

- A. Computational approaches (e.g. demand-driven, data-driven)
- B. Efficiency
- C. Parsing
- D. Translators (interpreters and compilers)

VPL-V. Language Purpose

- A. General-purpose languages
- B. Database languages
- C. Image-processing languages
- D. Scientific visualization languages
- E. User-interface generation languages

VPL-VI. Theory of VPLs

- A. Formal definition of VPLs
- B. Icon theory
- C. Language design issues
  1. Cognitive and user-interface design issues (e.g. usability studies, graphical perception)
  2. Effective use of screen real estate
  3. Liveness
  4. Scope
  5. Type checking and type theory
  6. Visual representation issues (e.g. static representation, animation)

Figure 2.2.2: VPLs Classification system [3]

For the purpose of this research Burnett and Baker's classification system is used to categorize VPLs, even though this classification originally was designed to help researchers find proper research materials in the VPL areas. A given VPL can be categorized at the same time under the section VPL II – Language classification (by paradigm or visual representation) – and also under the section VPL V – Language Purpose. The other sections are engaged more specifically with visual programming language features than with the taxonomy, and thus could be disregarded if one’s purpose is to find some sort of hierarchical taxonomy.

2.3 VPL Grammar

Describing a textual programming language in Backus Naur Form (BNF) is possible because only one type of relationship is allowed between symbols: the relationship *next to* [4]; thus there is no need to define the specific type of relationship. However, formally specifying a VPL is more challenging, since there is more than one relationship that needs specifically to be added to the grammar.

In 1994, Kim Marriot presented a framework to formally define visual languages—the constraints multiset grammar (CMG) [4]. He proposed a theoretical foundation to generate a parser from a grammar describing a visual language. The parser takes as input a multiset of strings, lines, arcs, circles etc. Marriot states that for visual languages, grammars and parsers use multisets instead of sequences, because in general, people do not follow the same order when drawing complex diagrams.

Marriot explains that CMGs differ from traditional string grammars in two ways:

1. String grammars rewrite sequences of tokens, but multiset constraint grammars rewrite multisets of tokens.
2. String grammars have only one type of relationship, which is "*next to*", but multiset constraint grammars have a wider number of relationships, such as intersection, next to, above, below etc.

Constraints are used in a CMG to define the relationship between components. A CMG over a computation domain D is defined formally by Marriot as being composed of:

- a set of terminal type symbols, T_T

- a set of non-terminal type symbols, T_{NT}

- a distinguished start type symbols, $S_T \in T_{NT}$

- a set of productions

The language of the grammar will be the set of all sentences that can be generated from the start symbol using the productions in the grammar.

Marriott defines that in a constraint multiset grammar, a production is of the form:

$S ::= S_1,\ldots,S_n \exists C \text{ on } S'_1,\ldots,S'_m$

where S is a non-terminal symbol that can be rewritten to the multiset of symbols $S_1,\ldots,S_n$ and C is a set of constraints on the attributes of other symbols $S'_1,\ldots,S'_m$. Marriott defines the constraints C as elements that enable the encodement of spatial layouts and relationships between a diagram and its components in the grammar.

Marriott gives the following production example:

P:state ::= Q:circle, T:text

where

Q .midpoint = T .midpoint,

2 * Q.radius >= T.height,

2 * Q.radius >= T.width,

P.midpoint = Q.midpoint,

P.radius = Q.radius,

P.name = T.string,

P.kind = normal.

In this production:

- *Q .midpoint = T .midpoint* constrains the midpoint of the text so that it is the same as the midpoint of the circle; therefore the text and the circle share a common area.
- *2* Q.radius >= T.height* informs that the text height fits in the circle
- *2 * Q.radius >= T.width* informs that the text width fits in the circle

It can be deduced that the text is entirely in the circle, and that the text is perfectly centered in the circle.

- P.midpoint = Q.midpoint →the center of the production is the center of the circle.
- P.radius = Q.radius → the radius of the production is the radius of the circle.
- P.name = T.string → the name attribute of the production is the text value of T.
- P.kind = normal → the production is of the type or kind *normal*.

In his study Marriot unfortunately found out that parsing a sentence to find if it belongs to the language of a CMG is an undecidable problem because CMGs can emulate two-counter machines. Indeed, this is based on the fact that the halting problem for two-counter Turing machines is unsolvable, as proved by Pierce from the Carnegie Melon School of Computer Science [5].

The details of the formal description of these CMGs are outside the scope of this research; therefore, this section presents only the result of Marriot's studies. After investigating CMGs that are cycle free, Marriot came to the conclusion that the complexity of parsing a cycle-free CMG is not polynomial but exponential, but parsing a fixed deterministic

CMG has a polynomial complexity. The analysis of his results determined that the complexity of CMGs is in between that of string grammars and constraint logic grammars.

The research results presented by Marriott give a sense of the difficulty in formally specifying a VPL using a grammar, thus, the formal specification of UVPL will not be covered in this research.

2.4 Cognitive Dimension of VPLs

The primary purpose for the development of VPLs is to provide usability. However, development of VPLs seldom includes tests to show whether or not a VPL is usable.

T.R.G. Green proposes a method based on *cognitive walkthrough* to help designers of VPLs detect the level of usability they have achieved [6]. His paper elaborates on the human computer interaction (HCI) technique known as cognitive walkthrough. This technique is used to detect and correct usability problems on a user interface.

Cognitive walkthrough is a tool that was designed originally for testing usability in the engineering field. Green states the four phases of this approach:

1. Set a goal to be accomplished
2. Search the interface for available actions
3. Select an action that seems likely to make progress toward the goal
4. Perform the action and check to see if progress is made towards the goal.

Green declares that cognitive walkthrough is a good method to evaluate the use of VPLs for the following reasons:

- The development of a program using a VPL usually is done through a GUI. The cognitive walkthrough method focuses on a user's ability to figure out how to use a new UI; therefore, it is beneficial to use a cognitive walkthrough method to test the usability of VPLs.
- Usually computer scientists do not have a background in cognitive science; however, the cognitive walkthrough method—unlike other HCI approaches—seems more easily usable by computer scientists that are not familiar with cognitive science.

In his paper, Green describes a method he calls the WYSIWYT (what you see is what you test) methodology that he uses to test the VPL *Forms/3*; this visual language was developed by Burnett and Ambler in 1991 [9]. Green shows that this method did not yield good results, and that refining this method with cognitive walkthroughs produced better results.

Finally, Green concludes that cognitive walkthrough is a method with limitations, since it cannot evaluate the cost of making an error, for instance. Nevertheless, cognitive walkthroughs perform faster than *pilot analysis* or *protocol analysis*, and the focus of this method is on specific areas in a subtask, which helps to target specific design issues.

2.5 Short VPL Survey

The following section presents a brief survey of different types of domain specific VPLs.

Alternate Reality Kit (ARK) [7]: implemented in Smalltalk-80, ARK was developed around 1986 by Randall Smith. It is a virtual world programming environment and can be classified as a domain-specific VPL, since its sole purpose is to aid in the simulation of the fundamental laws of nature via a 2D animated environment. ARK is a system developed for a non-programmer audience that needs to understand the laws of nature, like gravity or friction. This VPL enables the users to grasp the concepts of physical laws by allowing them to apply the simulated laws to physical objects via virtual simulation. In ARK, objects are images that have a position and velocity, and to which forces can be applied. A user manipulates a given object with another object, a hand, which is controlled using a mouse. ARK allows the user to simulate the physical laws in their very basics, whereby the full details of reality are not implemented; instead, the user directly simulates the effect of an action, rather than all the different small reactions that lead to the final action. Smith gives the example of the implementation of an electrical switch; the user does not simulate the physical installation of a button where electrical lines are connecting the switch to the power supply, but the button is visualized, and pushing it on or off will have a simulated reaction.

In ARK, users interact with the objects through a GUI; they also can create new kinds of objects and add them to the library of built-in objects in the ARK warehouse.

ARK has three types of users:

1. The application level user who typically just runs a simulation
2. The simulation builder who builds a simulation application
3. The lowest level user who builds tools to be used by the simulation builder.

An important issue to point out about ARK is the use of the mouse to operate what is called the hand object. It has been observed that use of the mouse to operate the hand is not intuitive, and confuses a lot of users [7]. Indeed, many computer mice have only two buttons (left and right), yet a hand can grab, pull, push, release etc., which means that the mouse cannot, in an easy manner simulate all the different capabilities of the physical hand. However it is very easy to learn the idea behind ARK and its concepts.

Visual Imperative Programming (VIPR)[8]: VIPR was developed at the University of Colorado by Wayne Citrin. VIPR is not an iconic VPL; instead of text or icons or graphs it uses nested concentric rings to convey the semantic of a program. From one step of the program to another, inner rings are being merged while the outermost ring is connected to the state. Figure 2 shows how VIPR represents an "if" statement.

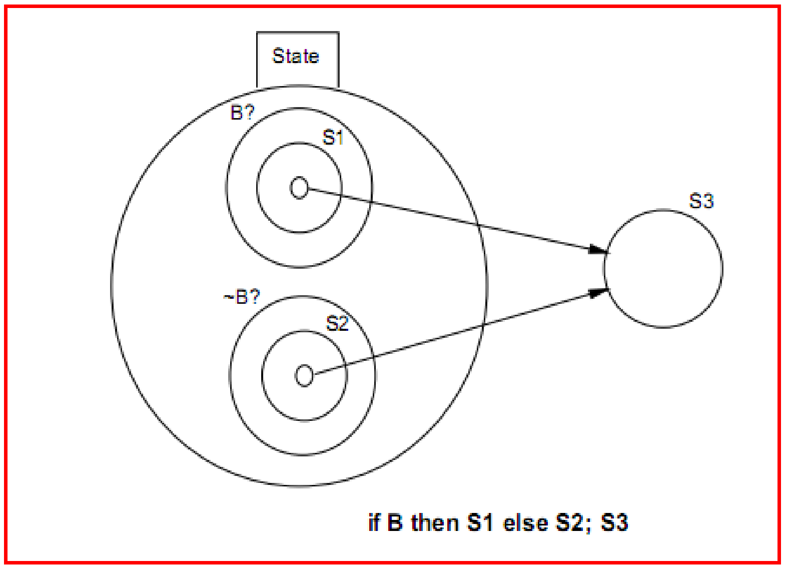


Figure 2.5.1: static VIPR representation of an if statement [8].

The development of VIPR was motivated by the desire to have an object-oriented language that is easy to learn and use; therefore, VIPR has all the features of an object-oriented language: inheritance, polymorphism, and dynamic dispatch, to name a few. The semantics are similar to C++, thus VIPR can be used for low or high level programming.

An expression oriented component, VEX, which is used for lambda calculus, also was added to VIPR. Lambda calculus is a notation to describe computable functions.

Prograph [9]: developed by Cox and Pietryzkowsky around 1990. Several versions have been released, and the latest one is Prograph/CPX. It is an object-oriented visual language that combines visual dataflow specifications with notions such as classes and objects. Prograph is an imperative language, which is a programming paradigm that describes computation in terms of statements that change the state of a program and the statements are executed in a sequential manner. In Prograph, *cases* and *multiplexes* are control structures used to replace explicit iteration by a sophisticated flow control. Prograph also provides persistent objects that are stored in a database. Methods are built up as accumulations of cases; each case in a method is a dataflow diagram that describes how the case should be executed. The diagrams are comprised of inputs, outputs and a set of operations; these entities are all connected. In Prograph, the order of execution is data-driven: the edges in a flow diagram indicate the data flow from one operation to another.

Visual TPL [10]: Visual TPL was proposed by Tu, Chen and Cheng, as the result of a research they conducted. This language is a domain-specific VPL, as its only use is for transforming data for generating reports. The inputs for a Visual TPL program are tables that come from a database. This language has four native components: table, helper, aggregation, and data source. A table component transforms one table into another. The resulting table typically is the data used in reports. The helper component is a collection of functions used for transforming data, and the functions are grouped as arithmetic, logical and relational operations. The aggregation component is another tool for data transformation, and it permits the programmer to perform aggregates such as averages, counts etc. The last native component Tu, et al. present in their report is the data source component, which is basically the component that will connect to a database to provide requested tables. The programmer also can combine preexisting components to make a composite one. The authors mention that the construction of composite components can be viewed as performing abstraction since the subcomponents used in a composite component are hidden.

A Visual TPL program is developed using an environment called Visual TPS, and this environment was designed specifically for Visual TPL. Tu. et al. describe the environment as having five areas. One area has icons for the native components, and another area is designated for the composite components; the components from these two areas can be dragged and dropped into a third area, which is a canvas where a program is built. All the components dropped on the canvas are linked by connectors that will drive the flow of the program. The fourth area in the Visual TPS environment is a display for immediate feedback, allowing the programmer to preview the result. The last area makes the components most used by the programmer easily accessible. The authors claim that the Visual TPS environment, which generates reports by designing a graphical data flow program, is easy to use and intuitive.

2.6 Scaling up Visual Programming Languages

The scalability issue is an important one for the viability of VPLs. Even though using visual languages can be a very interesting approach for editing a program, their usefulness has been affected by the inability of these languages to uphold large projects. Burnett and Baker describe this issue as "how to expand applicability without sacrificing the goals of better logic expression and understanding" [11].

In their paper, they discuss some issues pertaining to scaling up VPLs and some possible solutions, described below.

Static representation of a program, which is the complete representation of a program at rest, is de facto in traditional TPLs; however for VPLs – and more particularly interactive VPLs — it can be difficult to represent the entire program statically. Consequently, the review of a VPL program can be a difficult task. Some ideas that have been proposed would resolve this issue, but at the price of a VPL partially losing its visual nature. For instance, Burnett and Baker mention the translation of the program to a textual program for static representation; however this solution defies the purpose of VPLs, since the result of that transformation is a textual program. The usefulness of a VPL program representation is measured by evaluating the editability vs. the ability of a VPL to achieve some level of abstraction to hide excessive visual details.

Management of Screen real estate is another important problem, because of the nature of visual languages. It is challenging to edit and display a large visual program if the ratio of screen size to visual object size is too small. This issue involves how to display a large enough part of a VPL program to represent a logical block within the program. Burnett and Baker state that one solution to this problem is the use of scroll bars, but this solution would need to be coupled with others to be effective.

Burnett and Baker raise another issue concerning the incorporation of *internal documentation* in a VPL; this issue is solved in TPLs by the use of in-line comments ignored by the compiler. Documentation participates in scalability, because any type of documentation needs space—whether the documentation is always apparent, or whether the documentation is a dynamic text, where the text only appears at certain events such as a 'mouse over'. Some VPLs can be, by their nature, self-documenting, which alleviates the need for extra, explicit documentation; however, for VPLs that do not have implicit documentation, other solutions have been used. The VPL Forms/3 uses a form of documentation that is neither text, nor does it use space; rather, visual markers such as, coloring or boxing and lining perform the work of documentation. Another type of documentation, named ad-hoc documentation, also has been used; since the purpose of documenting is to help the reader of a program understand it better, ad-hoc documentation is a technique that tries to achieve this goal by providing an ad-hoc animation that displays the computation and the intermediate values for a portion of the program.

For a modern programmer the use of *procedural abstraction* is taken for granted, but in the early days of programming, it was considered as an important step forward. Similarly for VPLs the ability to reduce a logical portion of a program to an icon is considered an advanced way to apply procedural abstraction, and is considered a big contribution to the ability to scale up VPLs.

Jamal and Wenzel, in research on the scalability of LabView, point out that the criticism that has affected VPLs mostly is the lack of visual abstraction methods [12]. They explored the scalability of LabView and the abstraction mechanisms present in this language that help in managing large scale programs. Such mechanisms include icons on a diagram to describe its functionality. Another mechanism is the reuse of a diagram that was previously iconified.

Data abstraction—which is the use of user-defined data types—is as important as procedural abstraction. Burnett, et al. state that this object-oriented feature can contribute to the problem of VPL scalability [11]; even though data abstraction contributes in achieving a high level language, it might prevent interactivity. Proper access of a user-defined object is allowed only through operations defined in the data type of the object; if those operations are not visual, but rather textual, there is a possibility of losing interactivity or visibility [11].

In order to address this issue, a VPL that supports data abstraction needs to meet —according to Burnett and Baker—the following requirement: a VPL that supports data abstraction should provide a visual process to define a new data type, which also results in a visual program.

Finally Burnett, et al. discuss the relationship between programming language efficiency and scaling up a VPL. As most VPLs strive to supply immediate feedback, the need to provide responsiveness can affect the efficiency of a program, since the program will need to be translated and executed more often than a program in a language that does not provide immediate visual feedback.

2.7 Iteration constructs in VPLs

Another important issue in designing VPLs is the design of program control constructs, such as iteration. The nature of VPLs might make the representations more challenging. The biggest challenge in VPLs regarding the mechanisms of iteration is how to provide a compact viprocon with enough information to represent them properly. In the particular case of data flow VPLs, the issue is how to provide a mechanism for iteration without violating the very nature of a data flow paradigm. Mosconi and Porta, two researchers from the University of Pavia in Italy wrote a paper that presents the minimum set of characteristics to implement iterations in a data flow VPL, and they also show some types of iterations that could be implemented using the characteristics they defined [13].

Mosconi and Porta survey different iteration mechanisms adopted by several data flow VPLs such as LabView and Prograph, and they argue that some of these mechanisms do not respect the data-flow paradigm, even though they do contribute to a simplified user interaction. Mosconi and Porta state that one rule that should be followed in data flow languages is to avoid cycles; however, they notice that all the VPLs they studied use cycles to implement the constructs for their iteration. This is why the authors came to the conclusion that some data flow VPLs do not respect the data flow paradigm. They agreed that using cycles to represent data flow in iterations works, but they also studied others aspects of the data flow model to help implement better iterations.

Their studies allowed them to come up with four definitions, three theorems and six corollaries that describe pure data flow VPLs. Some relevant ones are given below.

Definition 1: A pure, data-driven, data-flow VPL is one that is made up only of nodes (visual elements representing functions, variables, constants) and links (visual elements connecting the nodes).

Definition 3: A pure, data flow VPL sub-graph is said to be iterative if there exists a function A in the sub-graph such that at least one of its inputs derives from an output of another function B for which, in turn, at least one input derives from an output of A (vice versa).

Theorem 1: In a pure, data-driven, data flow VPL it is not possible to implement an iterative behavior unless at least one function in the looped sub-graph receives more than one link for the same input.

Corollary 1: If a pure, data-driven …, data flow VPL does not allow functions to receive more than one link for the same input, iterative behaviors can be obtained by introducing into the language a special element that has two or more inputs and that behaves in the following way: it fires whenever one of its inputs is available; simply emitting that input as an output introducing the special element means that the data flow VPL is no longer pure.

With respect to these characteristics and some others that are not quoted here, Mosconi and Porta described in the remaining part of their paper the implementations of some iteration constructs that use enabling signals to avoid synchronization issues possible with inhibitor signals.

2.8 Arrays representation in VPLs

Allen Ambler published two papers pertaining to the representation and manipulation of data structures such as arrays in VPLs. He states that manipulating arrays in textual languages always has been a difficult task, especially for the non-trained programmer, since all manipulations have to be done through indexing. A certain level of abstraction in a visual language definition can allow certain kinds of operations on arrays without the need to index in any way. In his papers he proposes a different representation of arrays and also describes their manipulation [14] [15].

In his representation, arrays are represented by cells, and the user can choose to display scroll bars, since the array could be of any dimension.

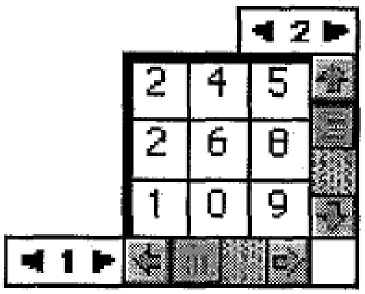


Figure 2.8.1 Array of 2 dimensions with scroll bars [14]

Arrays can be split into multiple parts called regions. Formulas or expressions can be applied to a whole region rather than just a cell, and therefore the user never has to deal with indexing.

Allen gives a few examples of manipulating arrays using his technique in the VPL Formulate. For instance, appending two arrays is performed by just providing to the function the two arrays to append. He also shows how arrays can be partitioned to form new regions by selecting and dragging borders. He demonstrates how summing a vector or a list could be done by creating a second vector or list that will carry along the partial sum of the elements, and thus the last element will contain the sum of the entire array, as shown in figure 2.8.2.

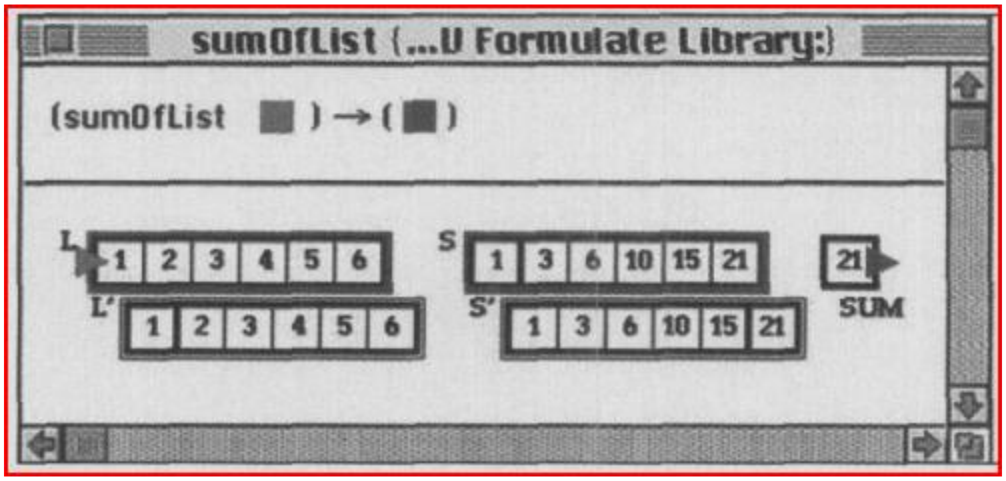


Figure 2.8.2: Summing a list in Formulate [14]

However by attempting to solve the Eight Queen problem, Allen concludes that not all the problems involving arrays can be solved without explicitly indexing the array.

One can come to the conclusion that representing arrays and manipulating them can be facilitated to some extent by providing the users some functions for the most common tasks, giving them the ability to build their own functions and providing them the ability to index the arrays. If the goal is to provide an easier way to manipulate arrays to inexperienced programmers, the goal can be achieved with built-in functions such as 'append array', 'sum a list' etc. The experienced programmers who already understand how to manipulate arrays can use either the built in functions or make up their own, as they most likely will be the users that will need more than just the built-in functions. Finally, displaying arrays as cells implies that the programmer probably prefers entering values or formulas into the cells rather than using indexing. The programmer many times does not know these values, and inserting formulas into the cells is not an elegant solution; thus, it might be preferable to abstract the structure of arrays in VPLs in order to better manage the edit area.

2.9 Principles of programming languages

In the book Principles of Programming Languages, McLennan aspires to provide descriptive tools, which he suggests are important for designing programming languages [16]. He insists that these principles are not laws that absolutely have to be followed; also, they are neither axioms nor a set of formal constraints. Further, some of these principles of programming languages cannot be applied at the same time because they contradict each other. Also, some principles may complement each other. It then becomes difficult to know which principles to adopt. Furthermore, unlike principles such as scientific laws, the principles of programming languages do not have quantitative measurements yet; therefore, McLennan suggests making tradeoffs based on qualitative judgments. The principles defined by McLennan that are used in this research are the following:

- The *responsible design principle:* find out what users need, not what they want.
- The *automation principle:* automate mechanical or error-prone activities.
- The *syntactic consistency principle:* similar things should look similar and different things different.
- The *defense in depth principle:* if an error is not caught by one defense, it probably will be caught by another.
- The *information hiding principle*: the user has all information needed to use a module and nothing more; all information needed to implement a module is provided and nothing more.
- The *security principle*: if a program violates its language definition or intended structure, the violation should be detected.
- The *abstraction principle*: avoid anything to be stated more than once.
- The *elegance principle:* designs look good because they are good.
- The *simplicity principle*: use a minimum number of concepts, with simple rules for their combination.
- The *impossible error principle*: making errors impossible to commit.
- The *orthogonality principle*: independent functions should be controlled by independent mechanisms.
- The *preservation of information principle*: representation of information that user might know and compiler might need.

- The *structure principle*: the visual form of a program leads the user to visualize its behavior.

- The *0 – 1– ∞ principle*: zero, one and infinity are the only reasonable numbers.[16].

Chapter III

METHODOLOGY

3.1 VPLs Selection Process

As a starting point, some visual programming languages (VPLs) are selected for a short survey. The starting list was composed of 43 currently available VPLs; each of these languages was considered for inclusion in the survey based on the following characteristics: the language purpose, the availability, the type of support available, the platforms supported and whether or not it is a teaching tool.

The language purpose is an important criterion, because some languages that are too specific, such as languages to edit music.

- Because this research in not funded, the availability criterion is used to eliminate the languages that are not freely available and the languages that do not provide free support.
- The study is conducted entirely on a Microsoft Windows machine, and so only languages available on Windows platforms are considered.
- VPLs used as teaching tools might not be good examples for designing a language for scalable programs; however, they probably have features that can be considered for helping non-programmers.

Applying these criteria, the list of VPLs was reduced to 10, as shown in Table 3.1-1 below.

VPL	Purpose	availability	Platform	Support	Teaching tool
AgentSheets	game	Trial	Mac / Win	yes	No
Alice	game	yes	Lin / Mac / Win	yes	Yes
Analytica	spreadsheet	Trial	Win	yes	No
Labview	testing, control device	Trial	Win	yes	No
Lily	Web dev.	yes	Lin / Mac / Win	yes	No
Microsoft VPL	robotics	yes	Win	yes	No
PointDragon	Web dev	yes	browser	yes	No
Simulink /Matlab	math	Trial	Win	yes	No
Tersus	Web dev	yes	Win / Lin	yes	No
VisSim	hardware testing	Trial	Win	yes	No

Table 3.1-1: VPLs to select from

From these remaining VPLs, one from each purpose category was selected arbitrarily, and the list of VPLs chosen for use in this study was reduced to Alice 2.2, Lumina Analytica 4.2, Microsoft VPL 2.1 and Tersus 1.3.

Alice is a VPL designed for high school and college students. It uses 3D graphics to teach introductory computing to an audience already familiar with videogames. Programs are built on a drag and drop interface. The 3D objects that are provided by Alice are used to create virtual worlds, and the program animates those objects.

Analytica is used to create and manipulate decision models. It is not a teaching tool. The user creates models by dragging to the work area viprocons (visual programming icons) that represent decisions, variables, chances, objectives, modules, indices, constants, functions, and text. The viprocons are connected with arrows that represent the flow of data. Each node has a definition that can be written with a procedural language very similar to Pascal. Analytica has 11 system libraries, and the user also can build more libraries.

Microsoft Visual Programming Language (MS VPL) is part of Robotics Developer Studio. It is a dataflow VPL and supports concurrency. MS VPL is designed for novice programmers, but also can be used by professionals. It is designed mainly for robotics programming, but also can be used for general purpose programming. The user manipulates blocks that are connected with arrows. Blocks such as the "If", "Calculate", or "Case", have expressions similar to C#. Libraries are wrapped around decentralized software services. Users can create their own services in C#, and can edit the not escape preexisting ones.

Tersus is designed for web application development. It is not a teaching tool. Tersus is a data flow programming language, and so the blocks in the diagrams need to be connected with arrows. A Tersus program has a top-down design, and is composed of web services, and built-in or user-defined components. The Tersus work area is called an "infinite drawing board," because the top model represents the system, and the user drills down to specify the components of the system and the details of those components, and the user can continue to drill deeper and deeper.

3.2 Analysis of principles for the selected VPLs

A design analysis needs to begin by laying down the principles that should be followed. For this purpose, a compilation of strategies from Burnett [1], who has focused her research on visual programming and especially on achieving scalability with VPLs, and principles from McLennan [16] are compiled in

Table 3.2-1. These strategies and principles are described in sections 2.4 and 2.5 of this thesis, and are used throughout this section to analyze the grokens and viprocons of the selected VPLs and later UVPL. The following sections describe the analysis of the categories of the programming constructs.

🟥 DIRECTNESS
🟥 EXPLICITNESS
🟥 LABELING
🟥 PORTABILITY
🟥 REGULARITY
🟥 RESPONSIBLE DESIGN
🟧 IMMEDIATE VIS. FEED.
🟧 AUTOMATION
🟧 SYNTACTIC CONSISTENCY
🟧 DEFENSE IN DEPTH
🟧 INFORMATION HIDING
🟧 SECURITY
🟨 ABSTRACTION
🟨 ELEGANCE
🟨 SIMPLICITY
🟨 IMPOSSIBLE ERROR
🟩 ORTHOGONALITY
🟩 PRESERVATION OF INFO
🟩 STRUCTURE
🟩 0 – 1– ∞

🟥 Most important	🟧	🟨	🟩 Least important

Table 3.2-1: List of Strategies and principles

3.3 Variables and literals

The analysis of variables in the VPLs selected for this research helps to determine which languages follow, or do not follow, the identified principles. Some examples related to variables and literals are given, and the analysis is summarized in Table 3.2-2.

The *directness strategy*: e.g. naming a variable should be done directly on the groken rather than by entering the name in a property window, since the name of the variable is part of the groken.

The *explicitness strategy*: e.g. the flow of data should be visually or textually explicit. Arrows can be used to direct the data flow explicitly. Keywords such as set get or the equal sign can be used as well to show whether a value is being assigned to a variable or a value is being retrieved from a variable.

The *labeling* principle*:* e.g. the memory location of a variable is not used to manipulate it, instead its name is used.

The *portability* principle*: e.g.* the data type of a variable should not be specific to a subset of machines architecture.

The *regularity* principle*:* e.g. in a language all variables are initialized automatically, or none of them are initialized automatically.

The *responsible design* principle*:* e.g. it would be irresponsible to design a language that provides to the user only integers of precision 128 so that a novice programmer will not have to worry about which precision to use. A responsible approach provides to the user integers of different precisions.

The *immediate visual feedback* principle: e.g. the value of a variable shall be displayed as the program is being edited, provided no run time value is needed.

The *automation* principle*: e.g.* the declaration of a variable is one of the activities where errors commonly occur. A common error made by novice programmers is to use a variable in the code without declaring it. This declaration could be performed automatically.

The *syntactic consistency* principle: e.g. the grokens for variables shall all look similar and they shall look different from other programming constructs.

The *defense in depth* principle: e.g. if the user can assign a string value to an integer variable while editing a program, the system should catch that error when an expression uses that variable.

The *information hiding* principle: e.g. for a string variable, the user will be provided all the string operations the language provides, but the system will hide the operations for integers.

The *abstraction* principle: e.g. two pieces of information are not needed to identify a variable as being an integer.

The *elegance* principle: e.g. this principle is violated if the groken for a variable is a really complicated geometric figure,. Add more about having choice of a simpler design.

The *simplicity* principle: e.g. the concepts should be simple.

The *impossible error* principle: e.g. mechanisms such as not allowing a string literal to be assigned to an integer variable can be implemented to avoid those errors.

The *preservation of information* principle: e.g. the user declares a variable to be of a certain type, and the system keeps track of that type.

The *structure* principle: e.g. use of a unidirectional arrow to represent assignment of the content of a variable to a different variable.

The *0 – 1–* ∞ principle: e.g. the maximum dimensions of arrays should not be limited to arbitrary numbers such as 4 or 7; the language should either not allow arrays (0), or allow only arrays of one dimension (1) or allow arrays of any dimension (∞).

	ALICE	ANALYTICA	MS VPL	TERSUS
DIRECTNESS				✓
EXPLICITNESS	✓	✓		✓
LABELING	✓	✓		
PORTABILITY	✓	✓		
REGULARITY	✓		✓	
RESPONSIBLE DESIGN				
IMMEDIATE VIS. FEED.				
AUTOMATION			✓	✓
SYNTACTIC CONSISTENCY		✓		✓
DEFENSE IN DEPTH			✓	
INFORMATION HIDING	✓	✓	✓	✓
SECURITY	n/a	n/a	n/a	n/a
ABSTRACTION	✓	✓		
ELEGANCE				
SIMPLICITY			✓	✓
IMPOSSIBLE ERROR				
PRESERVATION OF INFO		✓	✓	✓
STRUCTURE			✓	
0 – 1– ∞		✓	✓	

Table 3.2-2: Principles related to variables and literals

Strength and Weaknesses of Variables in Alice

Strengths:

Even though immediate visual feedback – as defined by Burnett – is not provided in Alice, the user can watch the values of the variables being updated when the program is running. The data type and the value assigned to the variable are presented visually and explicitly as shown in Figure 3.2.1.

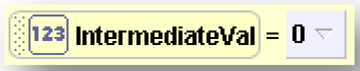


Figure 3.2.1: Example of a variable in Alice

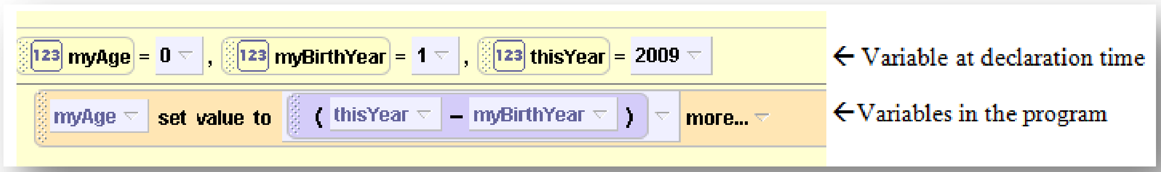

123 myAge = 0 , 123 myBirthYear = 1 , 123 thisYear = 2009
myAge set value to (thisYear – myBirthYear) more...
← Variable at declaration time
←Variables in the program

Figure 3.2.2: Syntactic consistency violation

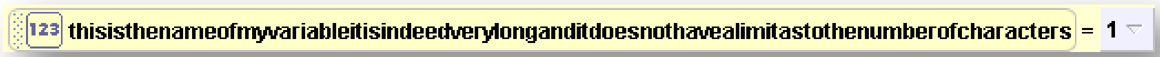

123 thisisthenameofmyvariableitisindeedverylongandit doesnothavealimitastothenumberofcharacters = 1

Figure 3.2.3: Example of a variable with a long name

Weaknesses:

Alice lacks direct manipulation of the variable grokens, leading assignments to be very cumbersome. One goes through several selection menus to assign to a variable a number, the value of a different variable, or the value of an expression. In Alice, a variable looks different when used in an expression than when declared, as shown in Figure 3.2.2. Figure 3.2.3 shows how variable grokens do not have a fixed size, whereby the icon grows as the name gets longer; this can lead to issues for screen real estate. All numbers in Alice are double precision floating point.

Strength and Weaknesses of Variables in Analytica

Strengths:

Analytica does not provide typed variables, however the type is deduced when operations are performed against the variables.

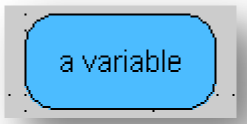


Figure 3.2.4: example of a variable in Analytica

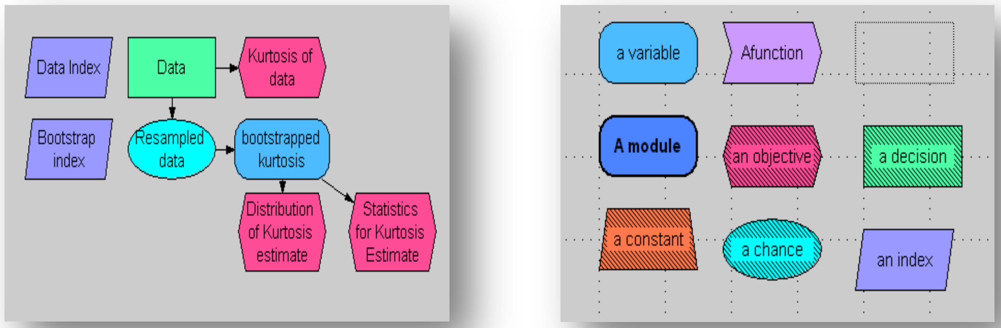


Figure 3.2.5: some grokens and viprocons in Analytica

Weaknesses:

The variable grokens in Analytica as depicted in Figure 3.2.4 are not manipulated directly; all interactions are effectuated in secondary screens, using a procedural, textual, language. Some grokens and viprocons, such as variables and modules are very similar to each other as shown in Figure 3.2.5, and furthermore the user has the option to make them look identical by setting them to the same color. The behavior of the program is not visualized easily, as values assigned to variables are not shown explicitly.

Strength and Weaknesses of Variables in Microsoft VPL

Strengths:

In MS VPL, regardless of the data type, all variables look the same and are differentiated from other grokens and viprocons by the color and the object label.

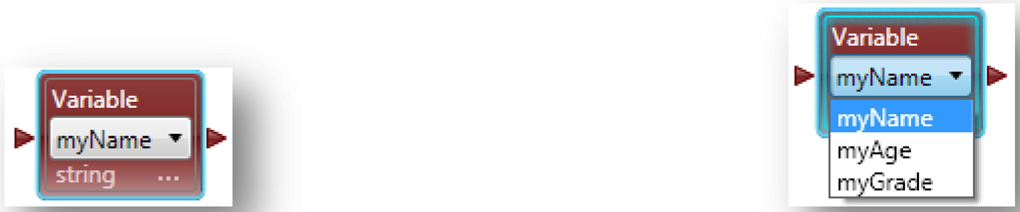


Figure 3.2.6: Example of a variable in MS VPL

Weaknesses:

The variable a groken represents is interchangeable at any point during editing by simply choosing a different variable from the dropdown, as seen in Figure 3.2.6. On one hand this feature adds convenience to programming, since on most VPLs changing a variable requires the groken to be deleted and replaced. But on the other hand this feature can be error prone.

Strength and Weaknesses of Variables in Tersus

Strengths:

Unlike in MS VPL, variable grokens in Tersus consistently receive data from their left side and output data through their right side, consequently leading to a simple design. The declaration of a variable is automated, whereby the user only needs to drag and drop the groken and starts using it.

Figure 3.2.7 shows how the data type is unnecessarily stated twice on the groken; nevertheless, the data type tags on the variable groken are persistent, which can help during editing of a program.

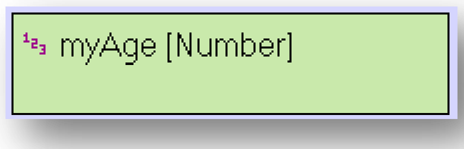


Figure 3.2.7: example of a variable in Tersus

3.4 Arithmetic, Boolean, and Comparison Operations

An operation groken accepts operands, and produces a result after some computation(s) are performed on the operands. The way in which these actions are performed in Alice, Analytica, MS VPL and Tersus are analyzed in this section, using the same strategies and principles described earlier in this chapter, and the findings are summarized in Table.

	ALICE	ANALYTICA	MS VPL	TERSUS
DIRECTNESS	✓	✓	✓	✓
EXPLICITNESS	✓	✓	✓	✓
LABELING	n/a	n/a	n/a	n/a
PORTABILITY	✓	✓	✓	✓
REGULARITY		✓	✓	✓
RESPONSIBLE DESIGN	✓		✓	✓
IMMEDIATE VIS. FEED.				
AUTOMATION	n/a	n/a	n/a	n/a
SYNTACTIC CONSISTENCY	✓	✓	✓	✓
DEFENSE IN DEPTH			✓	
INFORMATION HIDING	✓	✓	✓	✓
SECURITY	n/a	n/a	n/a	n/a
ABSTRACTION	✓		✓	✓
ELEGANCE	✓			✓
SIMPLICITY	✓			✓
IMPOSSIBLE ERROR	✓			✓
ORTHOGONALITY		✓		✓
PRESERVATION OF INFO	n/a	n/a	n/a	n/a
STRUCTURE	✓		✓	✓

Table 3.2-3: Principles related to operations

To put into context Burnett's strategies and McLennan's principles, some examples are provided to relate them to the operations analyzed in this section.

The *directness strategy*: e.g. arguments should be directly assigned to an operation by the use of arrows or other directive components.

The *explicitness strategy*: e.g. the purpose of the operation should be visually explicit; if it is an addition the operation groken should have the name or the symbol of the operation in it.

The *regularity principle*: e.g. all operations should accept arguments on a particular side (such as the left side) and output results from a different side (such as the right side).

The *responsible design principle*: e.g. the language should not permit a programmer to rename a built-in operation.

The *immediate visual feedback principle*: e.g. the result of an operation is displayed as the program is being edited.

The *syntactic consistency principle*: e.g. all grokens for categories of operations should have the same look and feel.

The *defense in depth principle*: e.g. if the VPL development environment fails to catch that not enough arguments are given to an operation, this error should be caught later in the editing process of the program, as the output from the operation is being used in another operation.

The *information hiding principle*: e.g. when the user is manipulating string variables, arithmetic operations should be disabled or hidden from the user.

The *abstraction principle: e.g.* two sorts of information are not needed to define an operation – like having the word "addition" and the symbol "+" used in the same groken.

The *impossible error principle*: e.g. the example provided for the information hiding principle , reduces the likelihood of programmer error.

The *orthogonality principle*: e.g. using the addition operation to perform additions and subtractions would be a lack of orthogonality.

The *structure principle*: e.g. the use of a unidirectional arrow to represent the result of an operation being sent to an output argument.

Strength and Weaknesses of Operations in Alice

<u>Strengths:</u>

Alice has an approach that follows information hiding, whereby the contextual menus do not display string functions when the variables being manipulated are numbers.

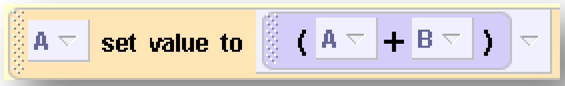


Figure 3.2.8: addition in Alice

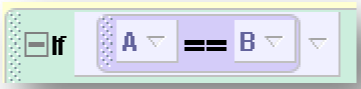


Figure 3.2.9: comparison in Alice

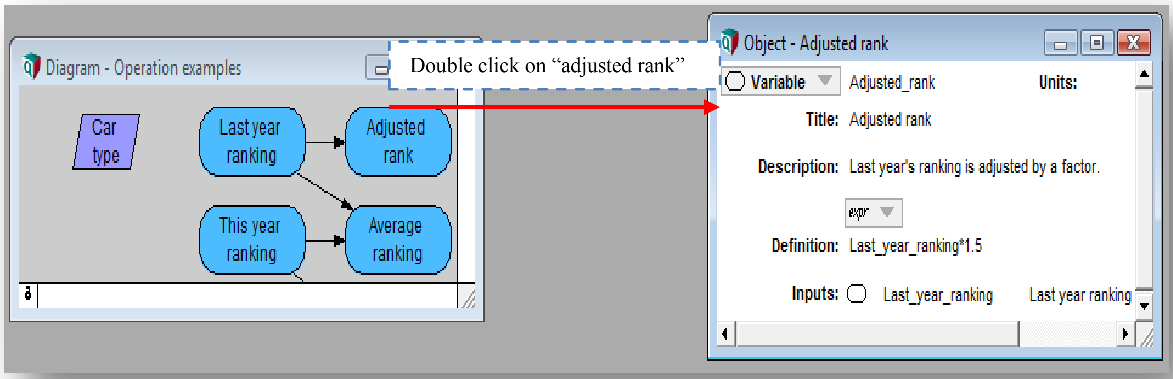
Weaknesses:

In Alice there is no notion of grokens to represent operations, and thus the operations are closer to being textual as illustrated in Figure 3.2.8 and Figure 3.2.9. Manipulating operations is not simple, because the user builds expressions entirely through selection menus. Floating point division is provided, but integer division is not; this is a direct effect of the lack of orthogonality in the design of variables, since in Alice all numbers are double precision floating point numbers.

Strength and Weaknesses of Operations in Analytica

Strengths:

It is not readily apparent that Analytica strongly complies with Burnett's strategies and McLennan's principles.



Adjusted_rank
Units:
Title: Adjusted rank
Description: Last year's ranking is adjusted by a factor.
expr
Definition: Last_year_ranking*1.5
Inputs: Last_year_ranking Last year ranking

Figure 3.2.10: Example of Operation in Analytica

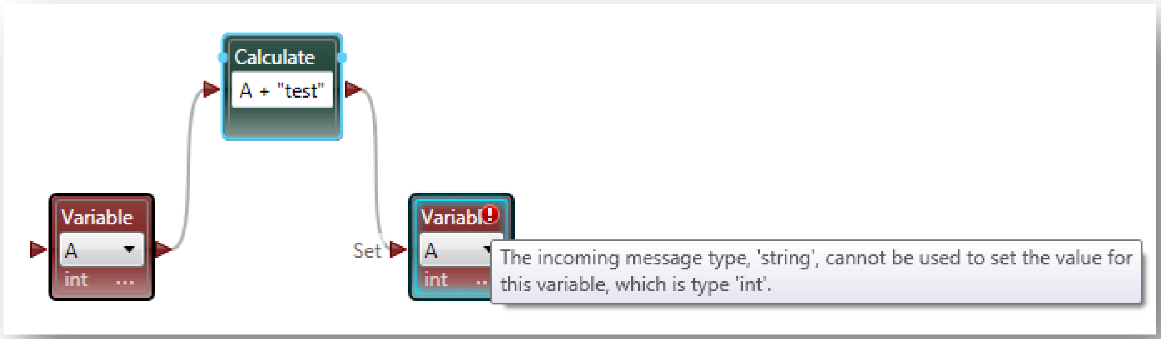
Weaknesses

Similar to VPLs such as MS VPL the user types expressions in a textual, procedural, language, and thus all operations are textual, as shown in the property form in Figure 3.2.10.

Strength and Weaknesses of Operations in MS VPL

Strengths:

If an operation is adding a string to an integer, an error occurs if the result is being set as shown in Figure 3.2.11.



The incoming message type, 'string', cannot be used to set the value for this variable, which is type 'int'.

Figure 3.2.11: invalid operation in MS VPL

Weaknesses:

In MS VPL, the operations are not iconic – they are textual, and are used like TPL operations.

Strength and Weaknesses of Operations in Tersus

Strengths:

Tersus operations have dedicated grokens, and in general follow the defense in depth principle, such as detecting when an integer is being added to a string as shows Figure 3.2.12.

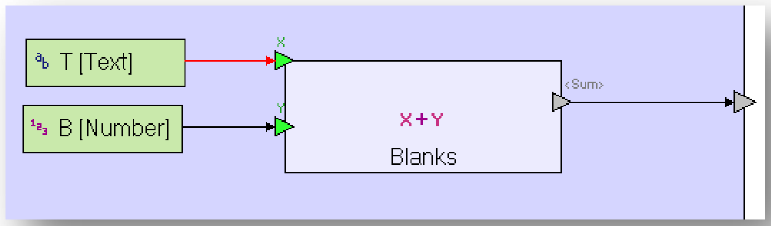


Figure 3.2.12: Invalid operation in Tersus

Weaknesses:

The user has the ability to rename an operation – for example, addition – to meaningless or misleading names such as 'division', '&' etc.; this feature gives the user the freedom to name an operation anything, but on the other hand it can lead to maintainability issues, if the programmer does not use it responsibly.

3.5 Control Flow

The result of the analysis of the control flow from the selected VPLs is presented in Table 3.2-4.

	ALICE	ANALYTICA	MS VPL	TERSUS
DIRECTNESS	✓		✓	✓
EXPLICITNESS	✓			✓
LABELING	n/a	n/a	n/a	n/a
PORTABILITY	✓	✓		✓
REGULARITY	n/a	n/a	n/a	n/a
RESPONSIBLE DESIGN	✓		✓	
IMMEDIATE VIS. FEED.				
AUTOMATION	✓	✓		
SYNTACTIC CONSISTENCY			✓	
DEFENSE IN DEPTH				
INFORMATION HIDING	✓	✓	✓	✓
SECURITY	✓			
ABSTRACTION	✓		✓	✓
ELEGANCE				
SIMPLICITY				✓
IMPOSSIBLE ERROR	✓			
ORTHOGONALITY	n/a	n/a	n/a	n/a
PRESERVATION OF INFO	✓	✓	✓	✓
STRUCTURE	✓		✓	

Table 3.2-4: Principles related to control flow

What follows are examples of applications of the strategies and principles related to control flow:

The *directness strategy: e.g.* "for loop" counters or "while loop" conditions could be assigned directly to a control flow viprocon by the use of arrows.

The *explicitness strategy: e.g.* the purpose of a control flow should be visually explicit; the viprocon should have the name or the symbol of the type of the control flow construct. The programmer should not have to infer the type of control flow.

The *responsible design principle: e.g.* the language should not permit an instruction within a loop to jump to any other part of the program except to the statements of the loop or to the statement right after the loop.

The *immediate visual feedback principle: e.g.* the language allows the display of the value of a counter as a "for loop" is unfolding.

The *automation principle: e.g.* the language should provide the option to increment counters in iterations automatically.

The *syntactic consistency principle: e.g.* the viprocons for all control flow should have a similar look and feel.

The *defense in depth principle: e.g.* the programming language could generate a warning if an infinite loop is detected; this could be useful to novice programmers.

The *information hiding principle: e.g.* a control flow viprocon should not request more information from the user than is needed to start or stop iterations.

The *security principle: e.g.* if a loop runs infinitely, all resources could be consumed, which in turn could lead to security issues.

The *abstraction principle: e.g.* a conditional loop should be implemented in such a way that the condition itself is stated only once, either at the beginning of the loop or at the end of the loop, instead of both at the beginning and at the end, or at the beginning of each case value.

The *impossible error principle: e.g.* mechanisms that could detect possible infinite loops should be encouraged.

The *orthogonality principle:* this principle is not applicable because, *e.g.,* writing a for loop as a while loop is not a bad design

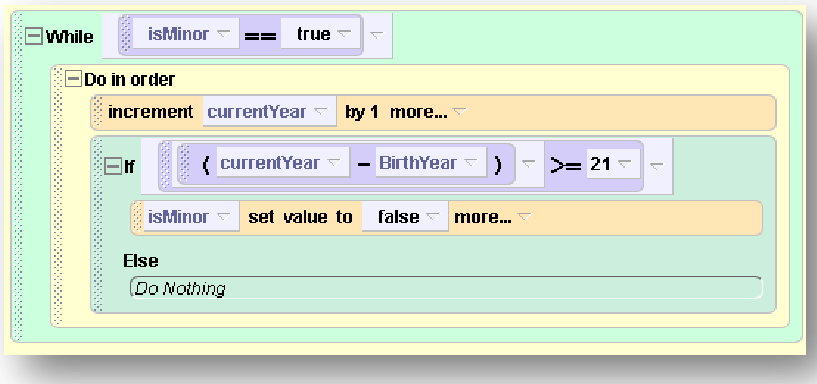
The *preservation of information principle: e.g.* representing a stopping condition in a "for loop" is an example of preserving information that the user knows and the compiler needs.

The *structure principle: e.g.* symbols to represent the beginning and the end of a loop could be used to add structure in a control flow viprocon.

Control flow in Alice

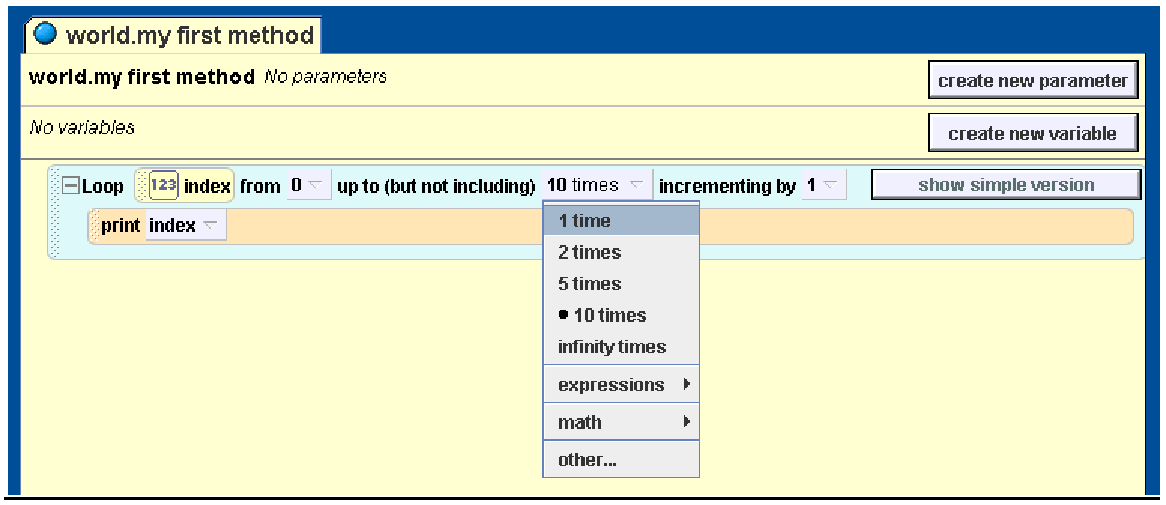
Strengths:

Alice provides a lot of automation, whereby counter variables are created automatically if the user does not specify any; there is also an option to set automatically a loop to run infinitely. Figure 3.2.14Figure 3.2.13 illustrates how in Alice the beginning and the end of a loop can be distinguished visually; indeed, a control flow block is represented by a distinctly-colored rectangle.



```
While  isMinor == true
    Do in order
        increment currentYear by 1 more...
        If  ( currentYear - BirthYear ) >= 21
            isMinor set value to false more...
        Else
            Do Nothing
```

Figure 3.2.13: *While loop* in Alice



```
world.my first method  No parameters
No variables
Loop  index from 0 up to (but not including) 10 times incrementing by 1
    print index
```

Figure 3.2.14: For loop in Alice

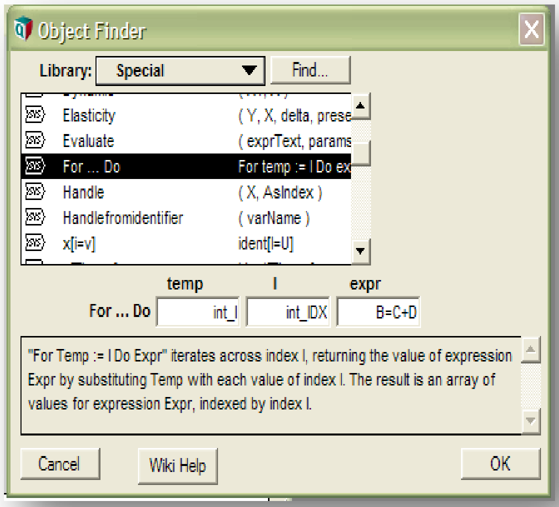
Weaknesses:

Figure 3.2.14 shows how in Alice *for loops* do not decrement, and when incrementing by (-1) – to perform a decrement – the program halts without throwing an error or returning any results. Furthermore, in the *for loop an existing variable* cannot be used as the loop index; Alice creates that index automatically.

Control flow in Analytica

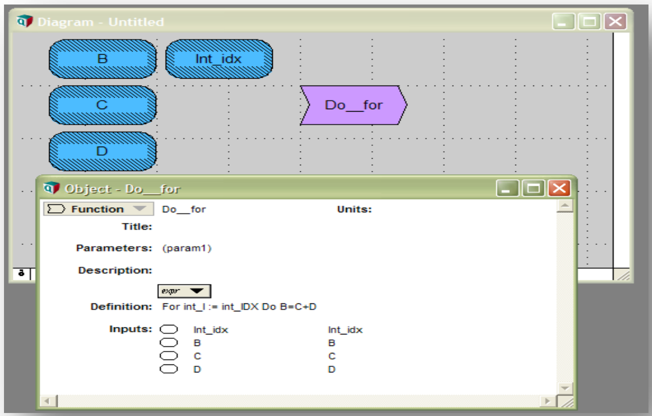
Strengths:

If the programmer defines the statements to execute in a control flow using undeclared variables, Analytica creates these variables automatically.


Object Finder
Library: Special
Find...
Elasticity (Y, X, delta, prese
Evaluate (exprText, params
For ... Do For temp := I Do ex
Handle (X, AsIndex)
Handlefromidentifier (varName)
x[i=v] ident[I=U]
temp I expr
For ... Do int_I int_IDX B=C+D
Cancel
Wiki Help
OK

"For Temp := I Do Expr" iterates across index I, returning the value of expression Expr by substituting Temp with each value of index I. The result is an array of values for expression Expr, indexed by index I.

Figure 3.2.15: Special library in Analytica



Do__for Units:
Title:
Parameters: (param1)
Description:
expr
Definition: For int_I := int_IDX Do B=C+D
Inputs: Int_idx Int_idx
B B
C C
D D

Figure 3.2.16: Indirectness in Analytica.

Replace this image with added function.

Weaknesses:

In Analytica, the control flow viprocons are the same as the function viprocons, and even though the control flow constructs are considered to be special functions, they cannot be distinguished from regular functions as shown in Figure 3.2.156. The user never manipulates directly the control flow viprocons; instead there are additional windows where the indices and conditions are specified as shown in Figure 3.2.16. Neither control flow nor iteration viprocons are visually explicit, so the programmer needs to give the viprocons a proper name.

Control Flow in Microsoft Visual Programming Language

Strengths:

The design of control flow in MS VPL has syntactic consistency; those viprocons are grey in contrast to red and green for variables and data as illustrated in Figure 3.2.18. The structure of the viprocons helps visualize their behavior for different outcomes.

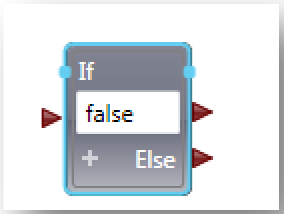


Figure 3.2.17: If viprocon in MS VPL

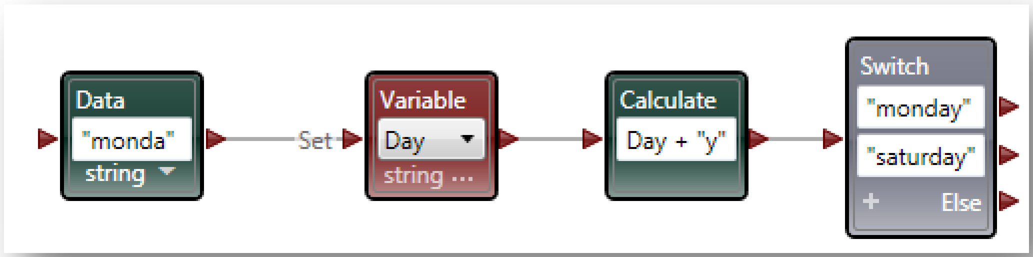


Figure 3.2.18: Switch in MS VPL

Weaknesses:

Control flow concepts in MS VPL are arguably explicit: for system-provided viprocons such as the *if statement* shown in Figure 3.2.17, the purpose of the control flow is explicit, but for viprocons such as a *for loop* the type of the control flow has to be inferred.

Control Flow in Tersus

Strengths:

The control flow viprocons are explicit; they are tagged by their names and have a representation of how the triggers (inputs) could affect the exits (outputs). Loops can be implemented through a repetitive functionality in Tersus. This feature simplifies for the user the set up process of loops, however it might be a concept hard to grasp for novice programmers.

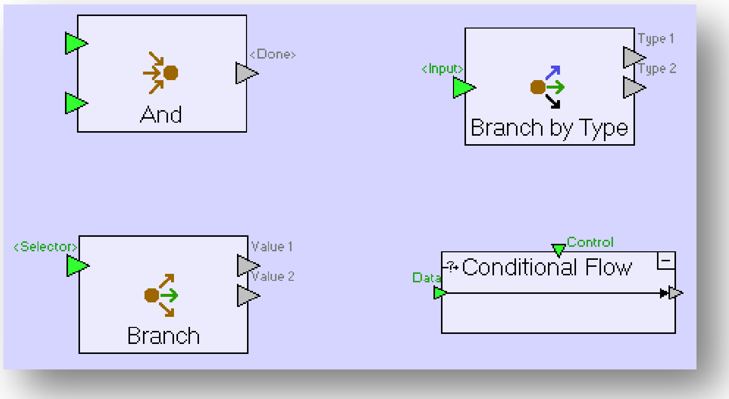


Figure 3.2.19: Control flow in Tersus

Weaknesses:

Unlike many other VPLs, Tersus does not provide viprocons for control flow constructs such as “if” or “for…loop”. Instead, Tersus provides "if, then, else" control flow in the form of comparisons, and therefore each comparison is a viprocon by itself with "then" and "else" branches. Loops in general are implemented through recursion. Further, Tersus provides an “and” viprocon – depicted in Figure 3.2.19 – which exits only if all mandatory triggers have values. Another iteration in Tersus is the “branch” viprocon, which evaluates its inputs and, based on the values, takes the corresponding exit; this viprocon is similar to a switch. The other iterations are the “branch by type” viprocon, which evaluates the type of its inputs, and based on the data type takes the corresponding exit. The input of the “conditional flow” viprocon as depicted in Figure 3.2.19 is transferred to the exit if all required triggers receive data.

3.6 Input / Output

A programming language is not of much value if it does not have functionalities to process inputs and to produce outputs. I/O functions in VPLs are of as much importance as operations or control flow, and the result of whether or not they were implemented with Burnett's strategies and McLennan's principles are summarized in table 5.

	ALICE	ANALYTICA	MS VPL	TERSUS
DIRECTNESS			✓	✓
EXPLICITNESS	✓		✓	✓
LABELING	n/a	n/a	n/a	n/a
PORTABILITY	n/a			
REGULARITY	n/a	n/a	n/a	n/a
RESPONSIBLE DESIGN	✓		✓	✓
IMMEDIATE VIS. FEED.				
AUTOMATION	n/a	n/a	n/a	n/a
SYNTACTIC CONSISTENCY				
DEFENSE IN DEPTH			✓	✓
INFORMATION HIDING	✓		✓	✓
SECURITY	✓	✓	✓	✓
ABSTRACTION	✓	✓	✓	✓
ELEGANCE			✓	✓
SIMPLICITY	✓		✓	✓
IMPOSSIBLE ERROR	✓	✓		
ORTHOGONALITY	n/a	n/a	n/a	n/a
PRESERVATION OF INFO	✓	✓	✓	✓
STRUCTURE	✓		✓	✓
0 – 1– ∞	n/a	n/a	n/a	n/a

Table 3.2-5: Principles related to I/O

As in previous sections, examples of applications of the strategies and principles to the design of I/O are stated below.

The *directness strategy: e.g.* the user could connect directly an input variable to an I/O viprocon.

The *explicitness strategy: e.g.* reading an input into a variable should be explicit, with the use of flow arrows or similar mechanisms.

The *responsible design principle: e.g.* the programmer should be limited on the number of files s/he is allowed to have open at the same time.

The *immediate visual feedback principle: e.g.* this strategy could be achieved by visually acknowledging changes to the state of a file, as a programmer is writing code affecting the file.

The *syntactic consistency: e.g.* I/O viprocons should look similar.

The *defense in depth principle: e.g.* if the user neglects to close explicitly a file in the program, the file should be closed upon exit of the running program by the system.

The *information hiding principle: e.g.* only the path of a file and the open mode of a file should be needed to perform an open file operation.

The *security principle: e.g.* some I/O operations should be subject to file permissions settings.

The *abstraction principle: e.g.* the information about the path of a file could be optional if the file is located in the same folder as the executables of the programs accessing it.

The *preservation of information principle: e.g.* the user should provide information of what needs to be read and where to store it.

The *structure principle: e.g.* reading from an input should be visualized as information leaving the input; writing to an output should be visualized as information entering the output.

I/O in Alice

Strengths:

Alice adopts the impossible error principle whereby the user is constrained to build an I/O operation by picking items from contextual menus as Figure 3.2.20 shows, and those menus only have items that can be used without causing errors.

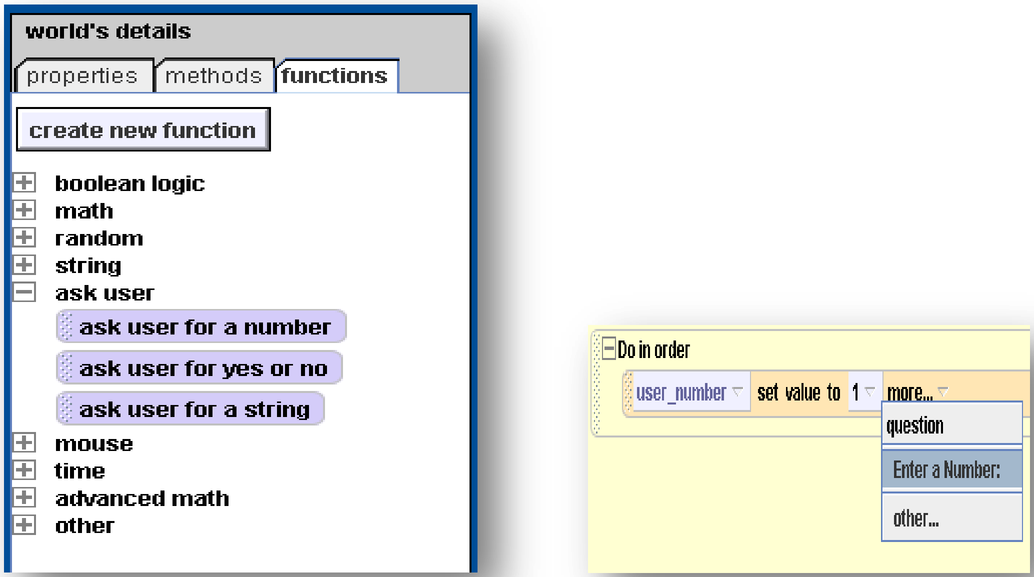


Figure 3.2.20: Setting up input in Alice

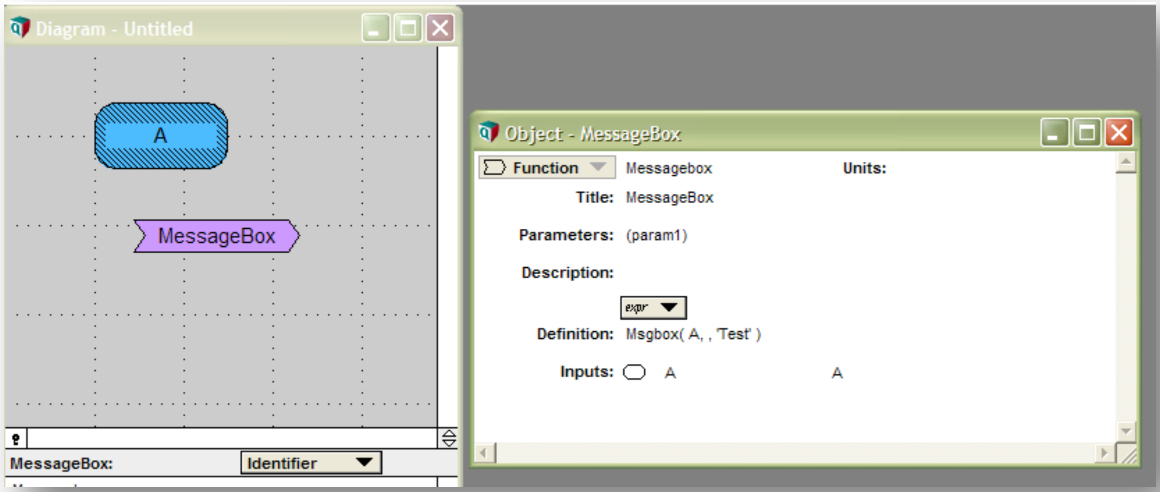
Weaknesses:

Because an Alice program looks like one in a TPL, there is no notion of manipulating directly an I/O viprocon. Alice 2.2 does not support files I/O, however a user can import music files; all other I/O is executed through standard input and output.

I/O in Analytica

Strengths:

Analytica uses modal dialog boxes to read information from the user or to display information to the user as depicted in Figure 3.2.22. Unlike in Alice, files can be handled in Analytica.



Messagebox Units:

Title: MessageBox

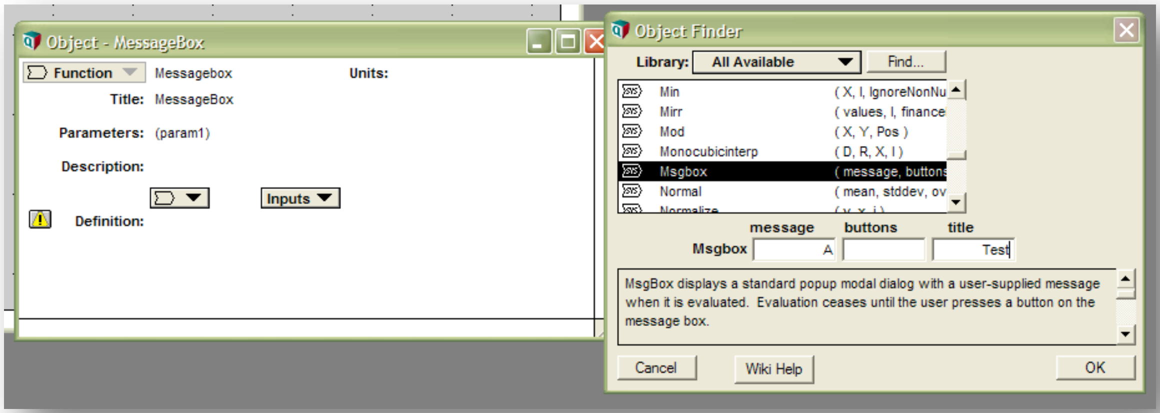
Parameters: (param1)

Description:

Definition: Msgbox(A, , 'Test')

Inputs: A A

Figure 3.2.21: Setting up input in Analytica



Messagebox Units:

Title: MessageBox

Parameters: (param1)

Description:

Definition:

Library: All Available

Min	(X, I, IgnoreNonNu
Mirr	(values, I, finance
Mod	(X, Y, Pos)
Monocubicinterp	(D, R, X, I)
Msgbox	(message, buttons
Normal	(mean, stddev, ov

	message	buttons	title
Msgbox	A		Test

MsgBox displays a standard popup modal dialog with a user-supplied message when it is evaluated. Evaluation ceases until the user presses a button on the message box.

Figure 3.2.22: Input in Analytica

Weaknesses:

Although Analytica supports files handling, there are not any dedicated viprocons to perform file I/O; instead the user writes code to perform these actions. Analytica does not follow the syntactic consistency principle, since I/O operations are set up as functions, and therefore can be difficult to differentiate from other functions. The information hiding principle is not observed, as Figure 3.2.22 shows; parameters such as units are requested by the system but are not needed to perform an output.

I/O in Microsoft Visual Programming Language

Strengths:

MS VPL has I/O for different data types, including text or numbers as shown in figures Figure 3.2.23 and Figure 3.2.24. MS VPL supports input from video sources or direct input from game controllers such as joysticks, as shown in Figure 3.2.25. I/O viprocons have directness and explicitness, as those are manipulated directly by the programmer, and the text tag or image explicitly define the nature of the I/O.

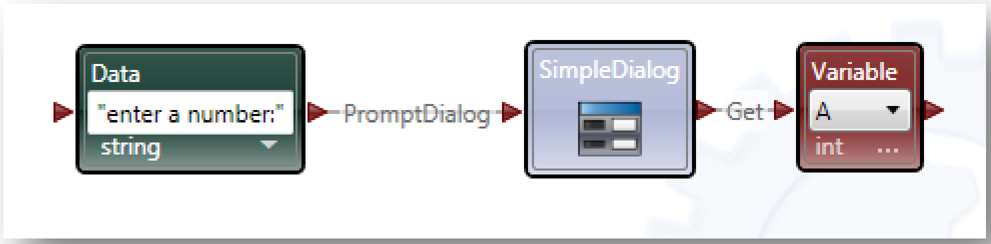


Figure 3.2.23: Output and Input example in MS VPL

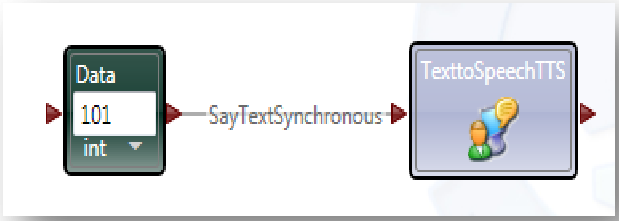


Figure 3.2.24: Text to speech Output in Ms VPL

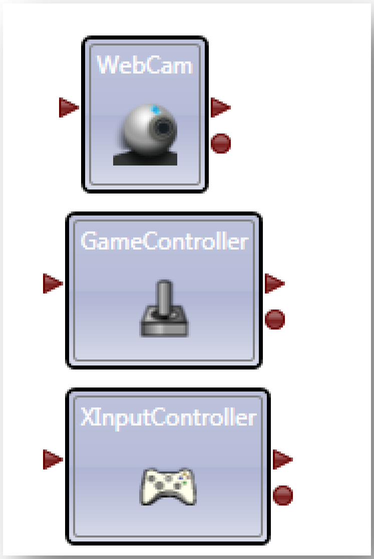


Figure 3.2.25: Miscellaneous I/O example in MS VPL

Weaknesses:

MS VPL does not handle natively text file I/O; instead the user needs to implement a decentralized software service in C# for reading and writing text files.

I/O in Tersus

Strengths:

Tersus VPL has a plethora of I/O viprocons, whereby the program can accept all native data type data for I/O operations. There are also some specialized I/O operations such as outputting an image as depicted in Figure 3.2.26, or reading an MS Excel document as shown in Figure 3.2.27. Tersus does not adopt the impossible error principle. Instead a defense in depth protocol is implemented; Figure 3.2.27 shows that the program cannot be validated if, for instance, a boolean variable is provided as the argument for the *read file* viprocon.

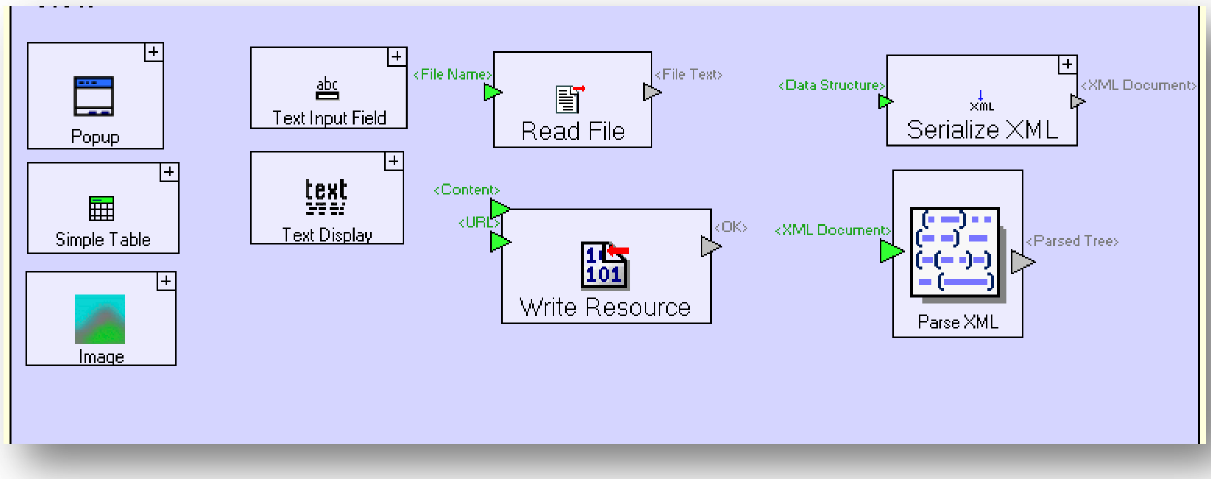


Figure 3.2.26: I/Oexamples in Tersus

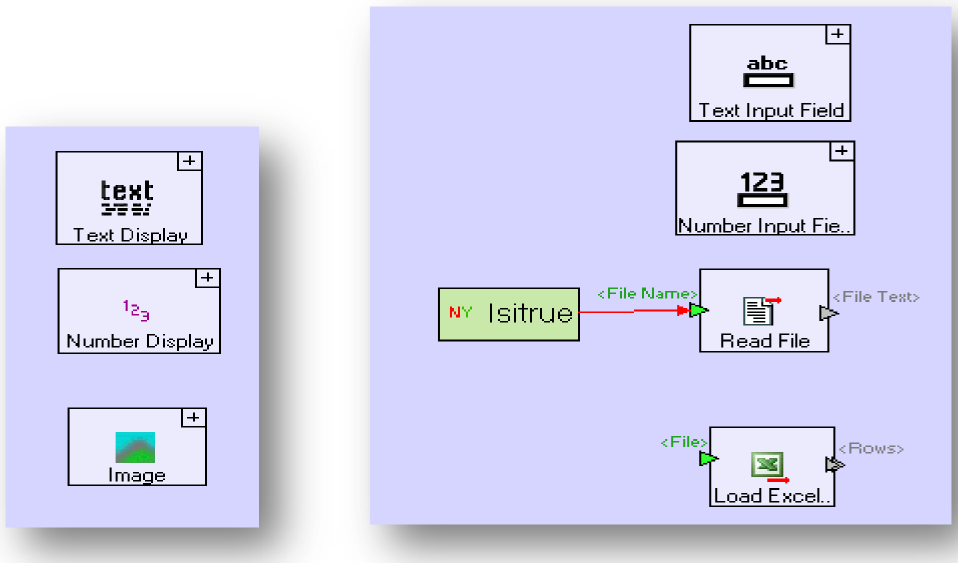


Figure 3.2.27: More I/Oexamples in Tersus

Weaknesses:

In Tersus, the responsible design principle is not followed; the user is allowed to change the name of a viprocon to an improper name.

3.7 Unified Visual Programming Language – UVPL

The analysis of the four selected VPLs is used as a basis to design the unified visual programming language (UVPL). The design of UVPL is inspired by the programming constructs in Alice, Analytica, MS VPL and Tersus. As a result, some elements in UVPL are similar to the ones in those languages. Nevertheless, different features are added to facilitate the programming task, ensuring that enterprise-sized programs can be developed with UVPL, all the while keeping in mind factors that could affect scalability. UVPL is intended to be a genera-purpose, object–oriented, visual programming language.

It has been noticed that the design of a VPL goes hand-in-hand with its development environment. For this reason, the design of UVPL is comprised of elements that are related to the development environment – programming features – and elements that define the language – programming constructs.

3.7.1 UVPL Programming Features

UVPL Development Environment Layout

The programming environment has a panel layout design to use more efficiently screen space but also to facilitate the viewing of a program. Initially only one panel is available to the user; as that panel fills up a scroll bar appears to enable viewing of items that do not fit on the screen. Subsequently, the user can opt to use more than one panel. By choosing to do so, the part of the program that cannot be viewed without scrolling is pushed automatically into the additional panel (s). Only the right-most and left-most panels have a vertical scrolling bar at that point: the left-most panel can only scroll up, and the right-most panel can only scroll down. Scrolling affects all the panels as the program moves as a whole. A program in UVPL is read in top-down, left-to-right order. Figure 3.3.1 and Figure 3.3.2 illustrate an example of the partial view of a program in a 3-panel layout.

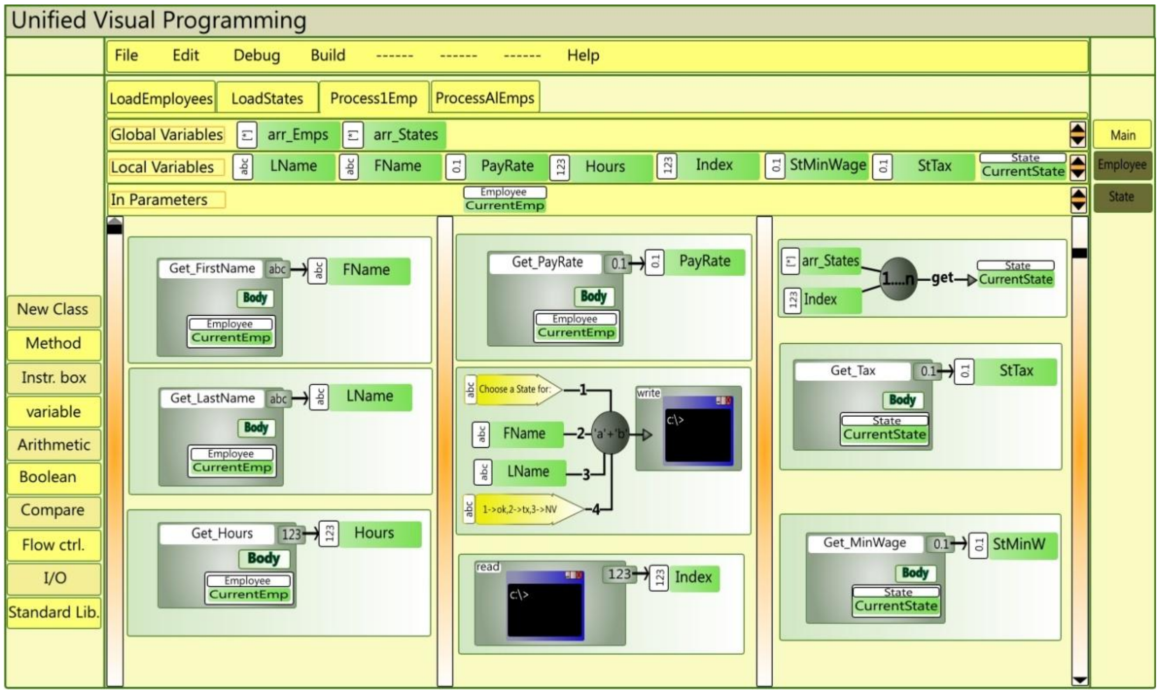


Figure 3.3.1: Partial view of a UVPL program -1

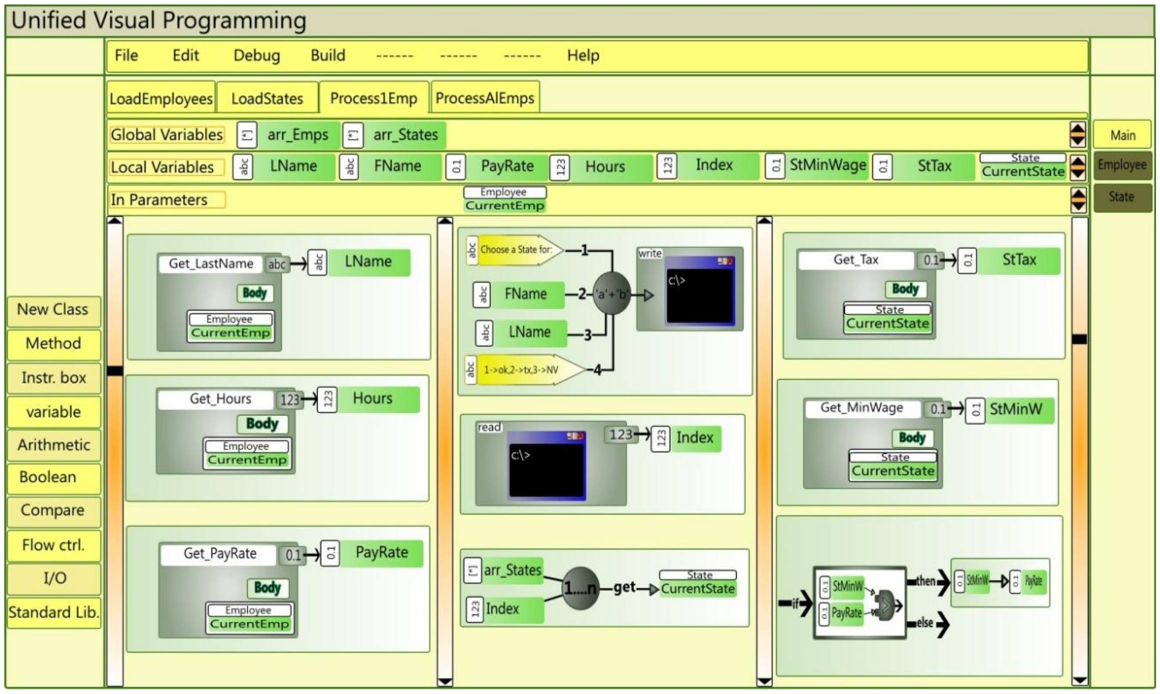


Figure 3.3.2: Partial view of a UVPL program -2

Sequentiality

In flow-graph-based VPLs such as MS VPL or Tersus, programming constructs need to be connected to propagate values or to represent explicitly the execution sequences of a program. The static representation of large programs in those languages is similar to a gigantic graph, and they can be difficult to view and understand. To alleviate this issue, the programmer can choose a modular programming approach, keeping each module a reasonable size. However one needs to be careful in adopting an 'extreme' modular approach, because if the modules are very small, as the program grows larger it will, at some point, become as difficult to understand as an un-modularized program that performs the same tasks. To this effect, UVPL has a different approach and combines the boxing effect of Alice, the top-down approach of Tersus, the notion of instruction found in TPLs, and a minimal use of connecting elements such as arrows in flow graphs. The result is what we call an "instruction box". Figure 3.3.3 shows an example of a box with two nested instruction boxes, and an instruction without any nesting. An instruction box contains a single instruction or a sequence of instruction boxes each containing a single instruction. Within an outer instruction box, the nested boxes are always in a single columnar arrangement. Apart from allowing a visual separation for the instructions, these boxes can be used for other purposes described later in this section.

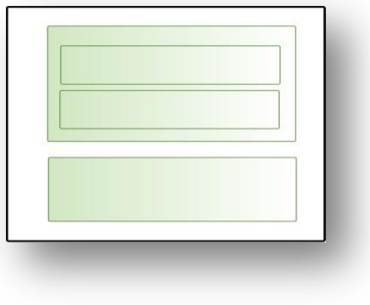


Figure 3.3.3: Instruction boxes

Comments

A well-written or well-built program – in the case of VPLs – clearly informs the reviewer what the program is doing. Adding comments to a well-built program provides more information; for instance a comment for a mathematical expression can explain why that particular formula was chosen over others. This additional information provided by comments is very useful when programs are being reviewed, and therefore can influence its scalability. However, when it comes to the design of VPLs, one need to pay particular attention to the implementation of comments. Indeed, while in a TPL, a comment can take as little space as an instruction, following that same approach for VPLs like Alice causes comments to occupy much needed screen space. MS VPL solves this issue by allowing comments to be minimized to an icon; unfortunately those comments in MS VPL are not attached to any part of the program diagram. In UVPL a different approach is taken; comments are interactive and are displayed only if the user chooses so. A comment can be added to an instruction box or to a sequence of instruction boxes if they first are nested into another instruction box. The border of the outer-most box then becomes a red line as shown in Figure 3.3.4. On a mouse-over of the red line, the comment appears in a call-out box as Figure 3.3.5 depicts. In this way, comments never use space permanently, and even when hidden the red line informs the reader about the presence of comments for a particular instruction box.

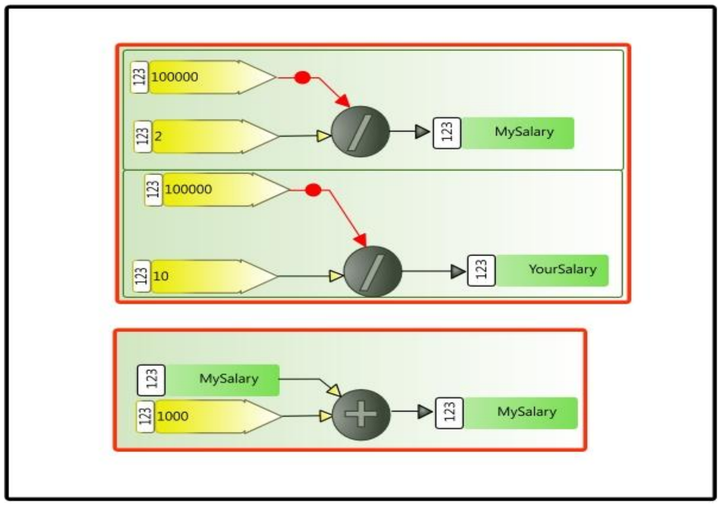


Figure 3.3.4: Hidden comments

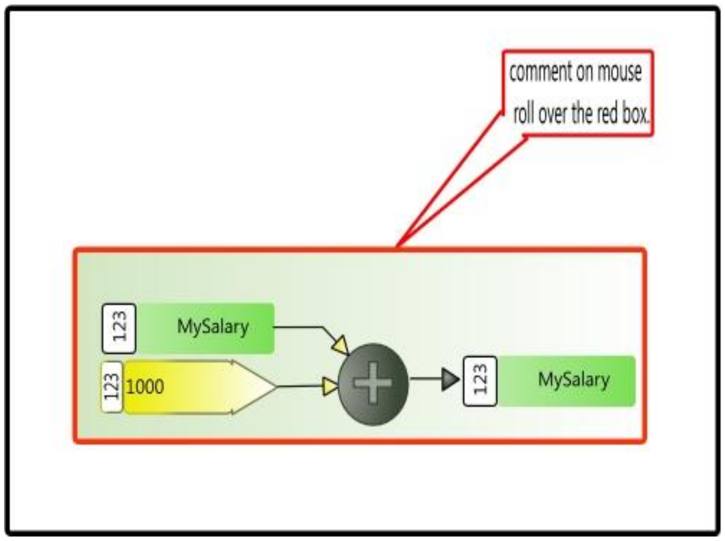


Figure 3.3.5: shown comment

Concealing / Revealing Expressions

For better management of screen real estate, UVPL adds the concealment of expressions to save screen space. Expressions are built as trees, and at each operation level, the user can choose to conceal the incoming branches to that operation, whether the incoming branches are just variables or expressions. In Figure 3.3.6, concealment occurs at a point where there are incoming branches, and the result of the addition is itself an input to the multiplication. In this case, everything before the addition is concealed, and the expression is reduced to what is depicted in Figure 3.3.7. In Figure 3.3.8, the user chooses to conceal at the division; in this case, everything except the division is concealed, as Figure 3.3.9 depicts. The user also can conceal several levels in the same expression with a single click. The concealing and revealing expressions allow the user to choose how much they want to see.

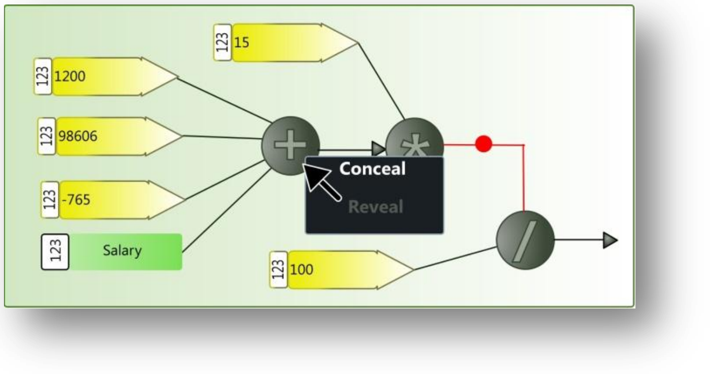


Figure 3.3.6: Concealing incoming variables

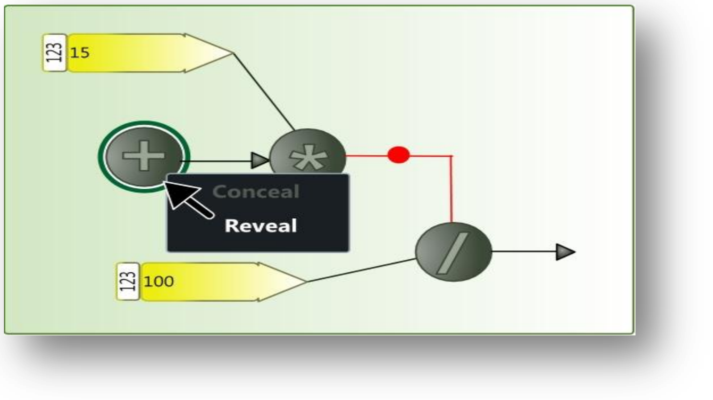


Figure 3.3.7: Revealing concealed variables

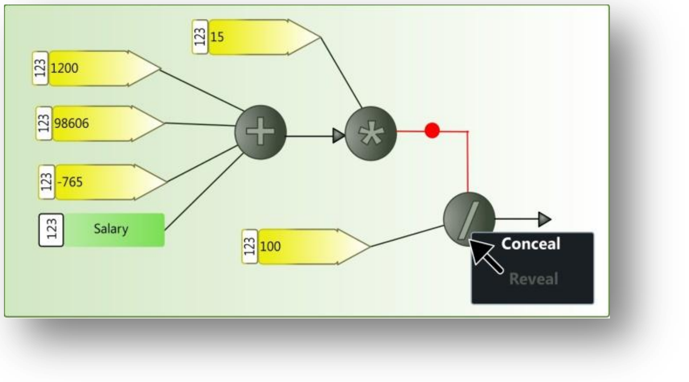


Figure 3.3.8: concealing incoming expression and variable

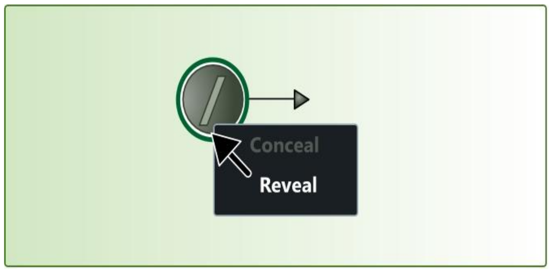


Figure 3.3.9: revealing incoming expression and variable

Docking

Docking is another concept that is added to UVPL for better management of screen real estate. Docking a program is an option that can be turned on or off in the *Edit* menu. When a program is in docking mode, blocks in the program are minimized to icon size as Figure 3.3.10 shows. A mouse over a minimized block magnifies it; Figure 3.3.11 pictures an example. Docking allows the programmer to have a better overall view of the program, and a block that is of interest can be magnified for a close up view.

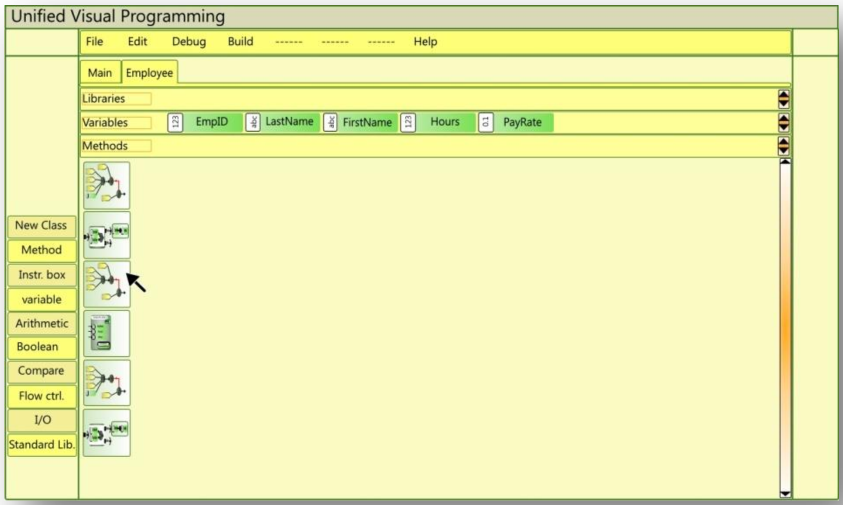


Figure 3.3.10: docked program

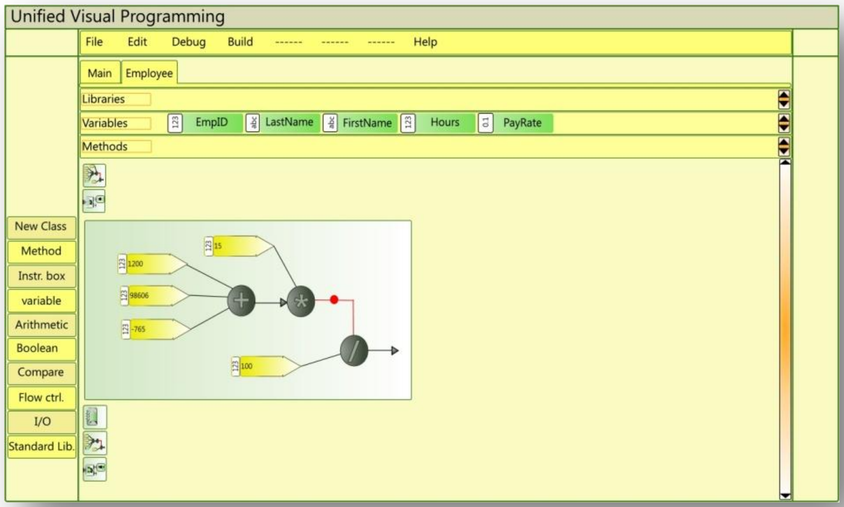


Figure 3.3.11: mouse over to magnify minimized block

Exception Handling

Unlike most visual languages, UVPL incorporates exception handling mechanisms.

After a method is built, the user can add exceptions. First, the user right-clicks on the viprocon of the method in which exceptions need to be handled, and then chooses "add Exception" from the menu. This action adds a button with the symbol **E!** to the method's viprocon, as shown in Figure 3.3.13: Exception handling in UVPL2. Additionally, a tab labeled using the name of the corresponding method with the symbol **E!** appended to it is created. Initially, this tab is not visible; to open that newly created tab the user clicks on the **E!** button in the method's viprocon, as figure Figure 3.3.12: Adding Exception Handling Stub3 shows.

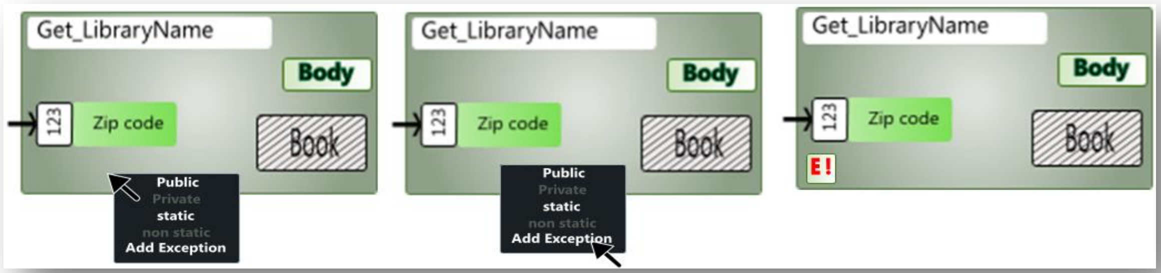


Figure 3.3.12: Adding Exception Handling Stub

The newly created tab contains in dock mode a read-only representation of the corresponding method's code. Into that tab, the programmer adds exception handling code under any block of instruction boxes, as Figure 3.3.13: Exception handling in UVPL depicts.

Having method code in one tab, and exception handling for that method in a different tab allows a clear separation of the algorithmic code from the exception handling code.

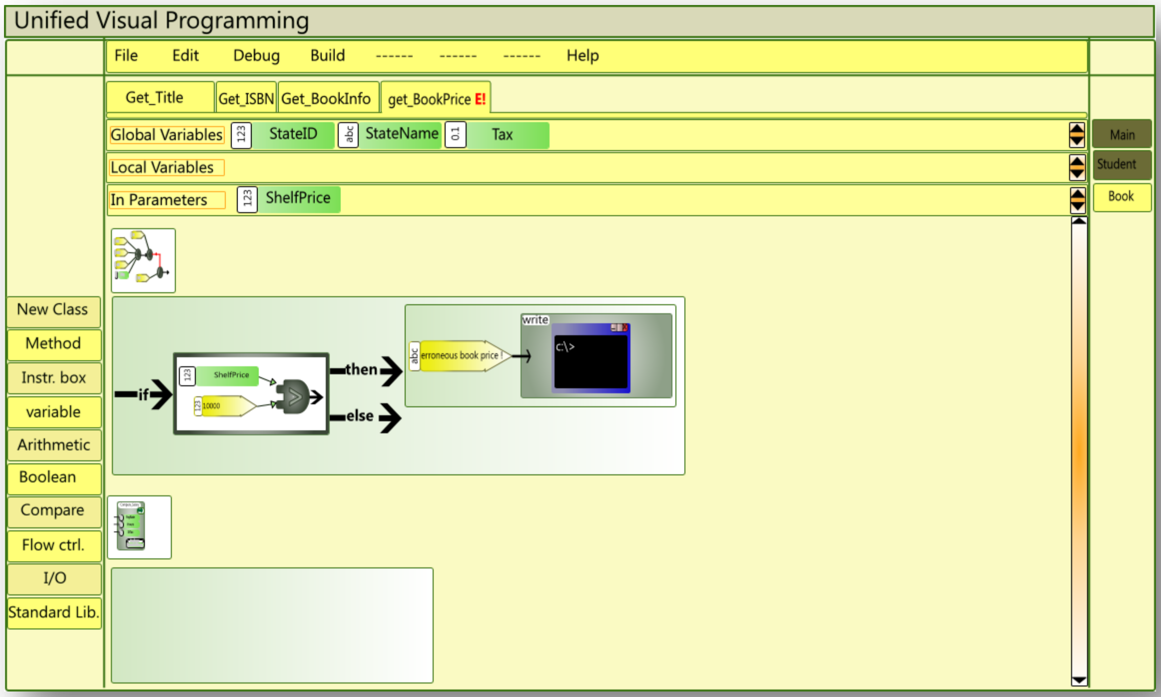


Figure 3.3.13: Exception handling in UVPL

3.7.2 UVPL Programming Constructs

Variables and Literals

Variables in UVPL are similar to variables in Alice and Tersus, where the type of the variable is attached to the groken. At any point in a UVPL program, the type of a variable is always known, as Figure 3.3.14 shows. In contrast to Alice and Tersus, UVPL has more native data types:

- Integers: they are by default int and can be set via a right click to tinyint (1 byte), smallint (2 bytes), int (4 bytes), or long (8 bytes).
- Floating point numbers: they are by default single precision, but can be set via a right click to single or double precision.
- String.
- Boolean.
- Object.

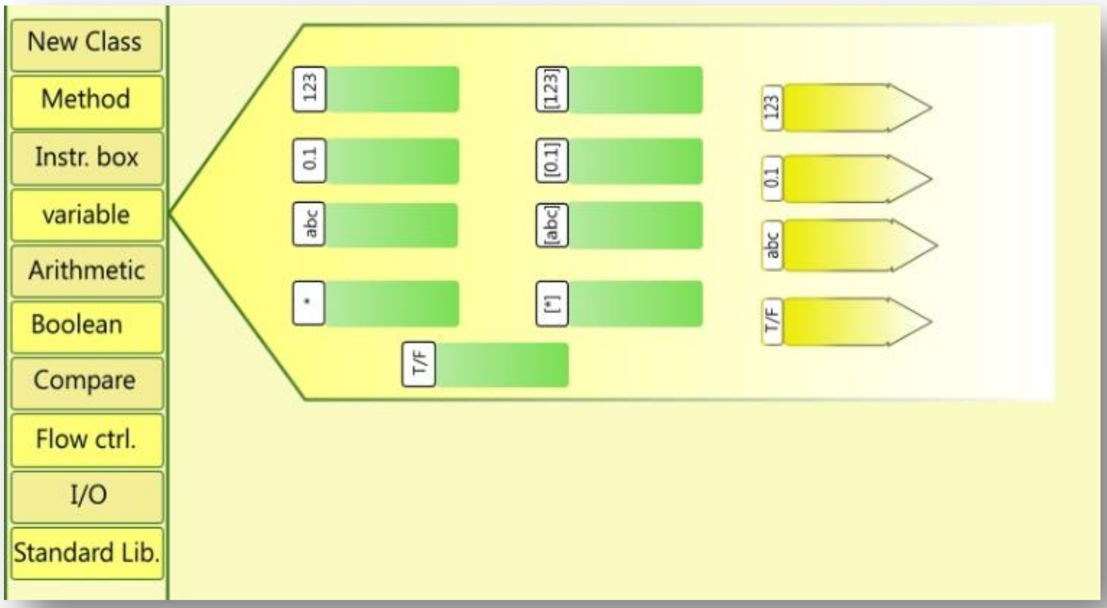


Figure 3.3.14: Data types in UVPL

Arithmetic Operators

The grokens for arithmetic operators shown in Figure 3.3.15 can take more than two operands for inputs, allowing expressions to be more compact.

For additions and multiplications, the orders in which operands are added or multiplied do not affect the result, and therefore the user can add as many operands as necessary to the same groken.

For subtractions, divisions and modulus, it is important to know the minuend and the dividend, and thus these two terms are connected to the groken through a red, single-dotted, connecting line, as represented in Figure 3.3.16. For a division operation with more than two arguments, the dividend is divided first by any of the divisors and the quotient of that operation is in turn divided by any remaining divisors until no more divisors are left. The key in this operation is that as long as a dividend is identified, the divisors are applied one by one in any order to the quotients. This same rule applies for a subtraction operation. However, for a modulo operation the order in which the divisors are applied to the remainder is important; therefore they are used from top to bottom.

If the user decides to change the minuend or the dividend to a different argument, s/he needs to drag the red dot to the desired argument. At that point the selected minuend or dividend has a red, single-dotted, connecting line and the previous selection is turned to a black line without the red dot. The selected minuend and dividend is put always automatically at the top.

Assignment to a variable is represented simply by an arrow.

Setting the value of a cell array, or getting the value from a cell array is performed by using the *get* and *set* grokens represented in Figure 3.3.16.

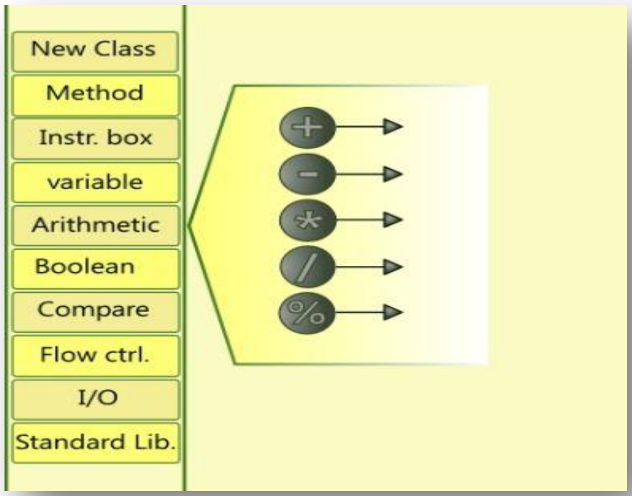


Figure 3.3.15: Arithmetic operators

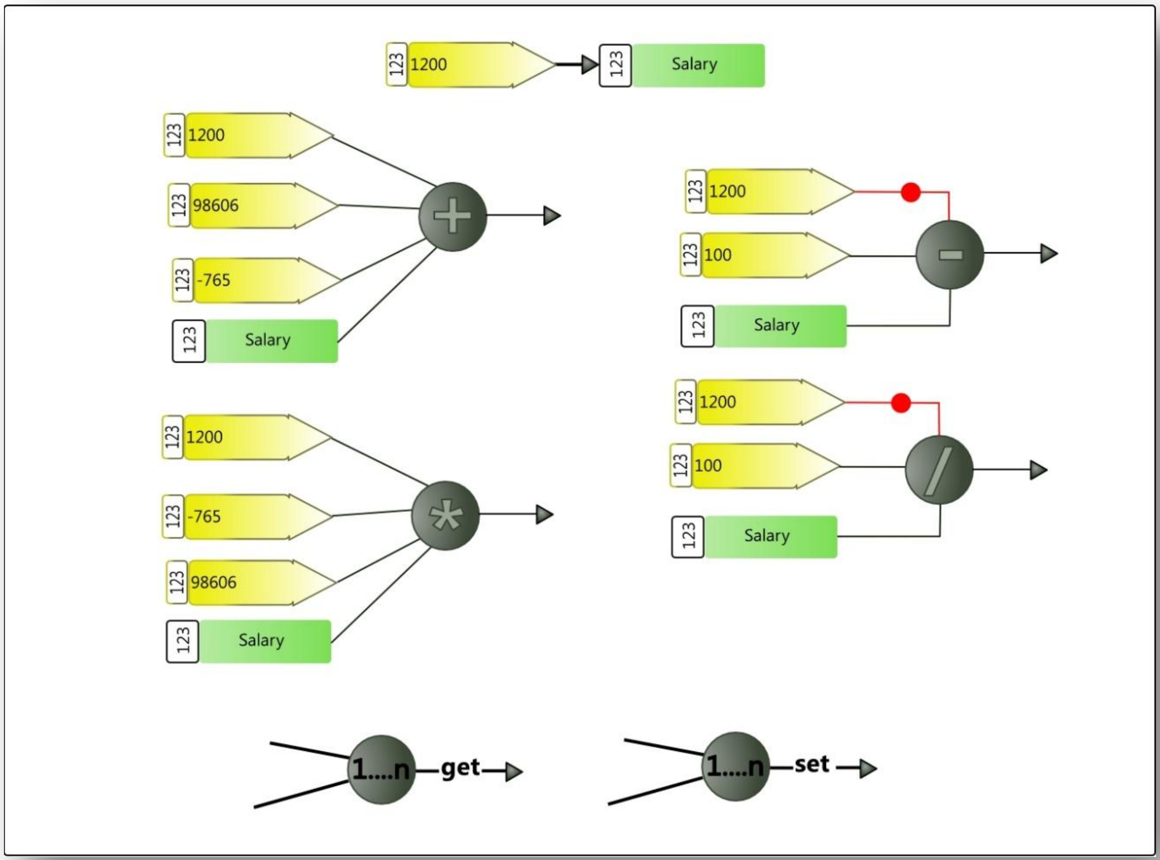


Figure 3.3.16: Arithmetic operations in UVPL

Boolean and comparison operators

The Boolean operators *AND*, *OR* and *NOT* are represented by logic gates symbols. They accept Boolean values and return a Boolean value. To fulfill the syntactic consistency principle, all operators have the same look and feel, and are manipulated in the same way.

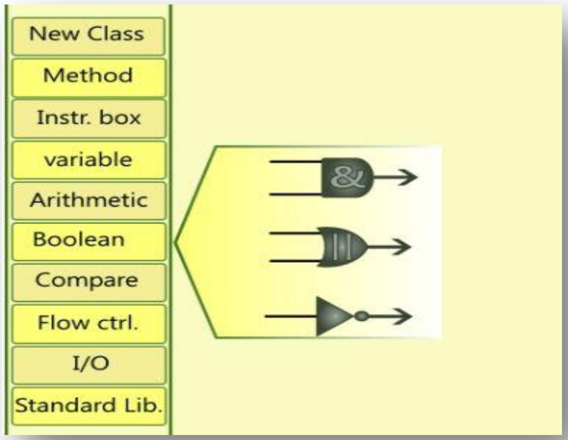


Figure 3.3.17: Boolean operators

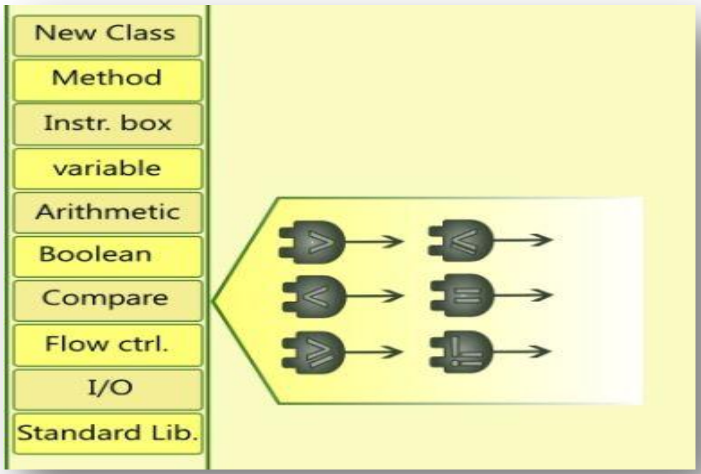


Figure 3.3.18: Comparison operators

Control Flow

Aside from the *switch viprocon*, which is designed based on the one in MS VPL, the iteration viprocon designs are not based on the selected VPLs for the following reasons: Alice does not have dedicated viprocons that can be manipulated directly by the user for iterations; Analytica iterations are in fact textual; and Tersus does not have traditional iteration constructs. In Tersus for instance, the *if statement* is combined with the result of a comparison. Instead of having a single *if…then…else* viprocon, Tersus has for each type of comparison a different viprocon representing an *if…then…else*. The other control flow viprocons in Tersus are branching by data type of a variable, and branching by value, the latter being basically a *switch*.

UVPL has four different iteration viprocons; Figure 3.3.19 has a representation for each. The viprocons have a box where the user builds the condition that is evaluated to decide how to branch or how to loop. There are three variants of the *for* loop viprocon:

- The first one is a ***for***<variable>***from***<starting value>,***to***<last value>,***by***<increment>.
- The second variant is a ***for each***<arrIndex>***in***<array>.
- The last variant is a ***for each***<arrIndex>,***in***<array>, ***key***<condition>.

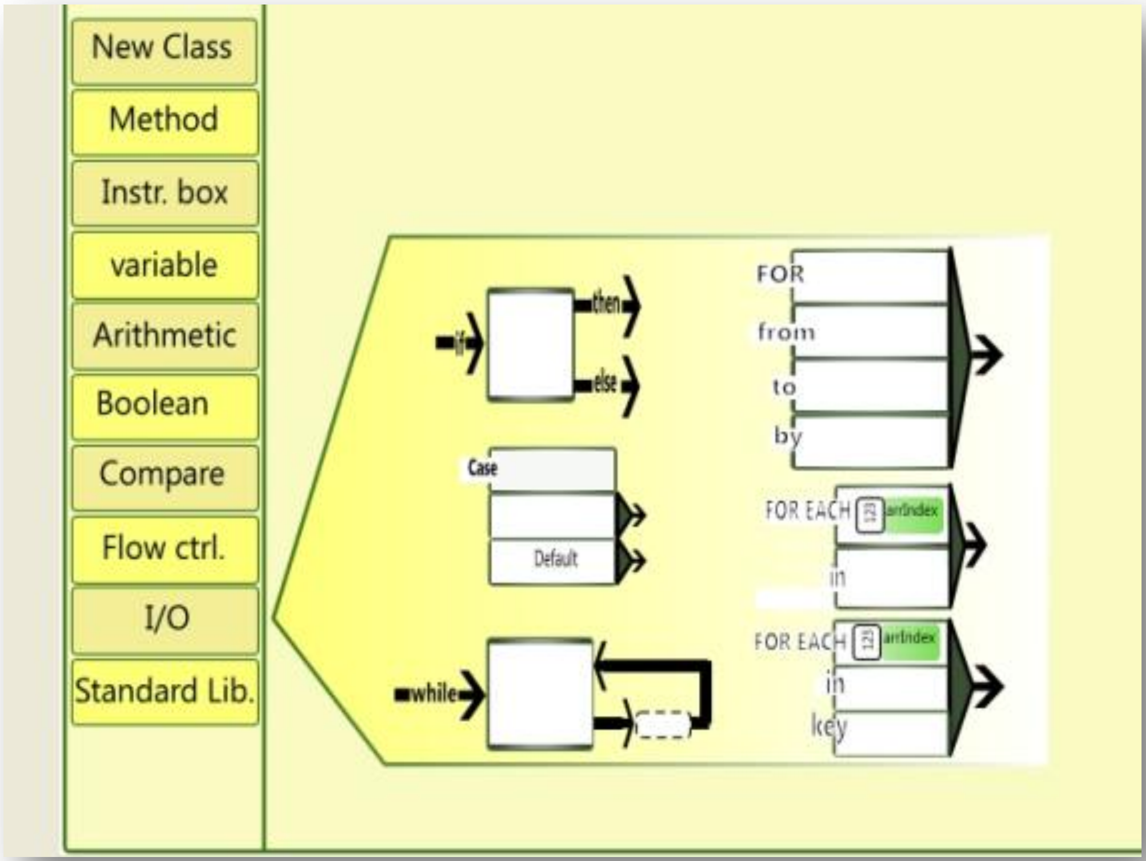


Figure 3.3.19: Iterations in UVPL

Input / Output

UVPL has standard I/O and file I/O as Figure 3.3.20 shows. The design of the I/O is based on the viprocons of Tersus file I/O. Among the selected VPLs, Tersus alone handles different types of files; furthermore, the dedicated I/O viprocons of Tersus explicitly represent their purpose and are manipulated directly by the user.

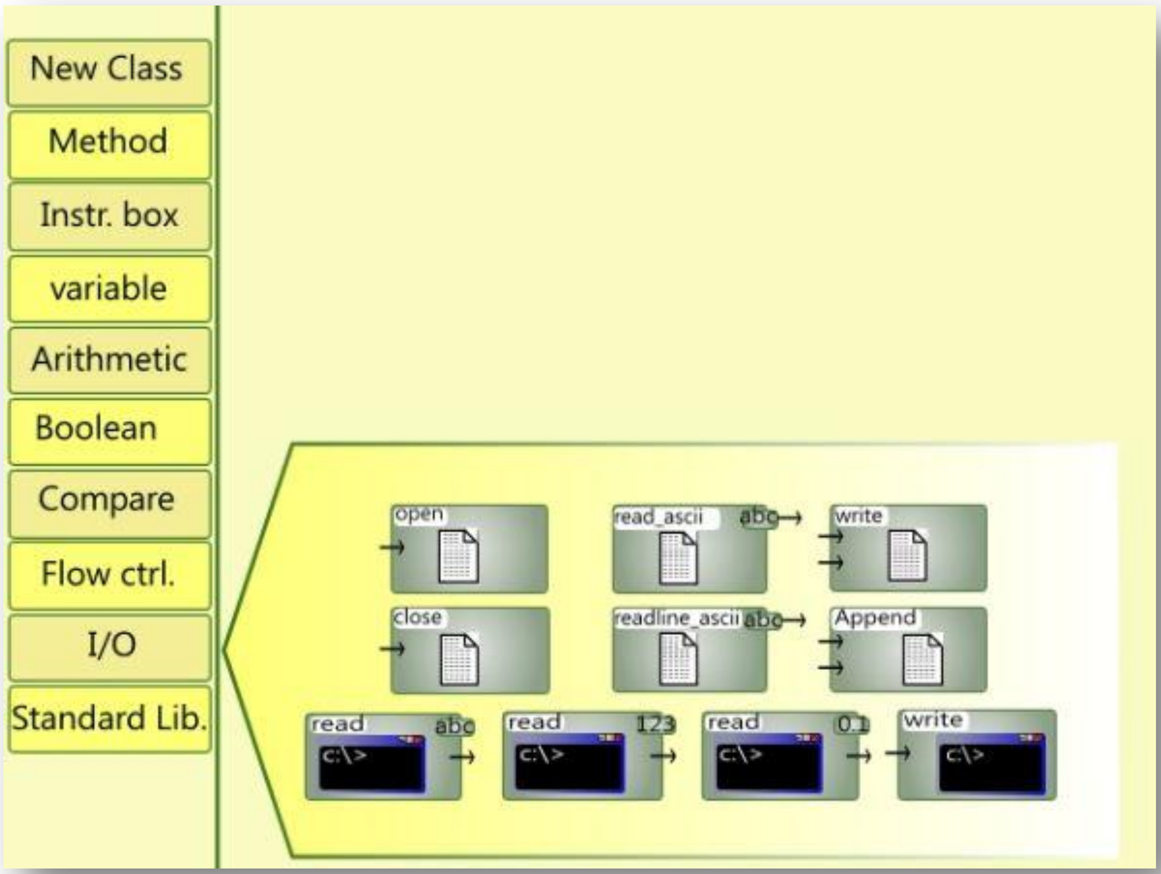


Figure 3.3.20: File and standard I/O

3.7.3 Object-Oriented UVPL

UVPL is not a fully object-oriented language, because it does not have OO features such as inheritance or packages. However a UVPL program is constructed with classes defining objects. An object is represented like a variable with the type attached to its groken, as shown in Figure 3.3.21.

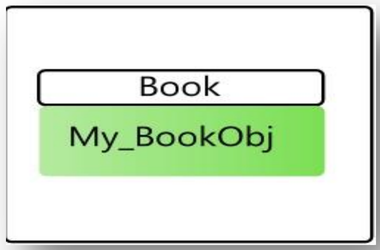


Figure 3.3.21: an object in UVPL

Methods, shown in Figure 3.3.22, are represented as viprocons to allow direct manipulation and reuse when the method needs to be invoked more than once. A unidirectional arrow is used to pass variables into the method by value; a bidirectional arrow is used to pass variables by reference. The visual design of methods is based on that of *activities* in MS VPL. An activity is a viprocon that symbolizes a method. Figure 3.3.23 shows an example of an activity, *ProcessEmployee,* in MS VPL. But because UVPL is an object-oriented language, its methods have more capabilities than those in MS VPL. Indeed, UVPL methods can be public or private, static or not and can accept arguments by value or by reference. Furthermore, all arguments passed are visually represented, allowing a reviewer to have more information about the method, all the while abstracting the details of what the method is doing.

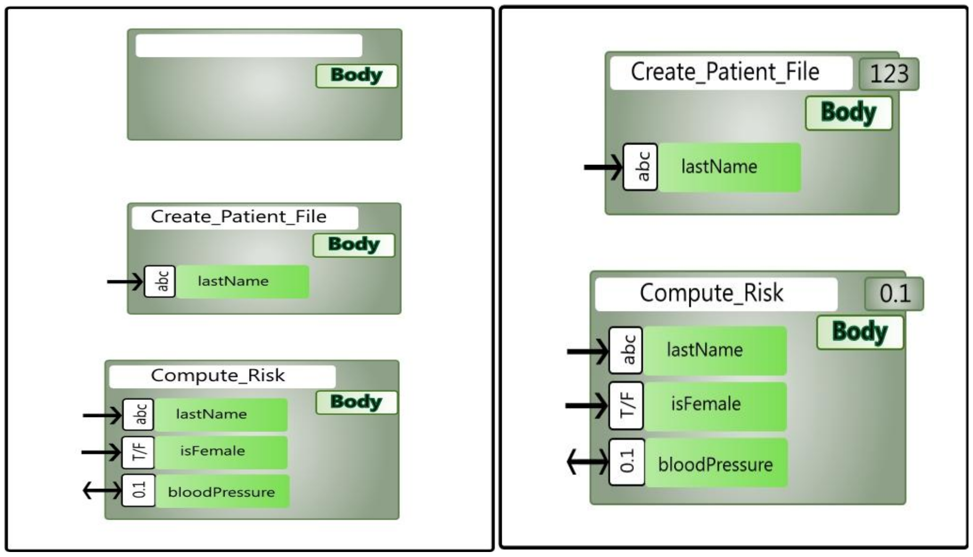


Figure 3.3.22: Procedures and functions in UVPL

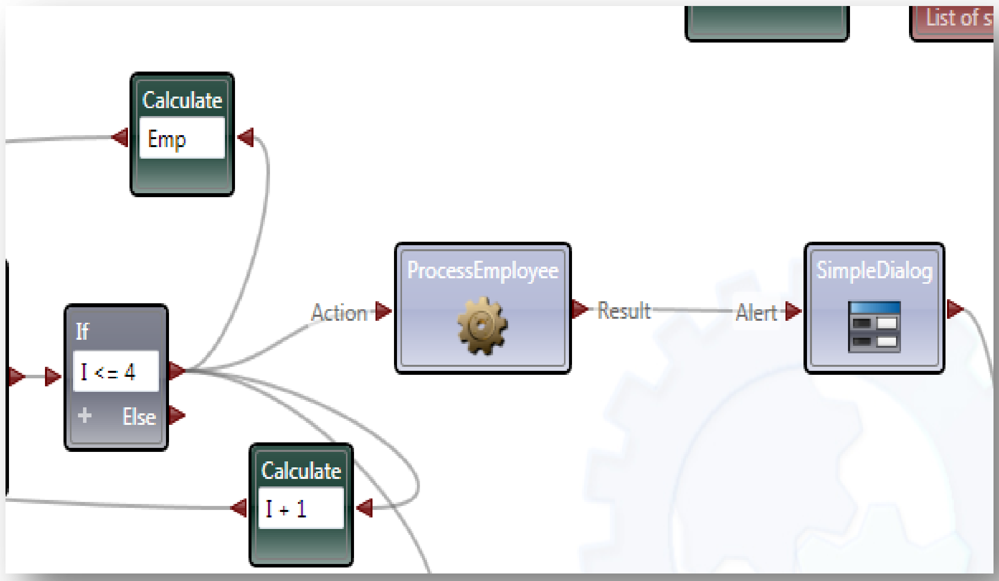


Figure 3.3.23: Example of an activity (ProcessEmployee) in MS VPL

A public method, as depicted in Figure 3.3.24, has a slot where the groken for the object that is referenced is dropped. In this example, *My_BookObj* is an instance of the class *Book*; the user invokes the public method Search_word, to search a given word within the object *My_BookObj*.

In a public static method, as shown in Figure 3.3.25, the slot contains no groken, but instead is hatched to symbolize inaccessibly. However, when invoked outside the host class, the hatched slot will have the host class name.

Figure 3.3.26 represents a private static method; the background of the viprocon is hatched as well as the object slot.

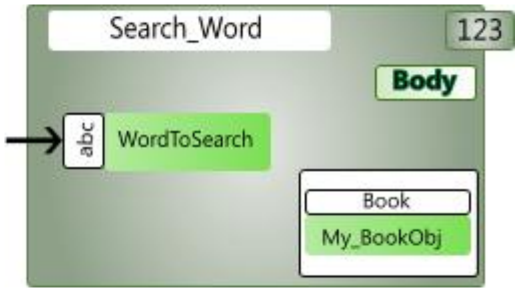


Figure 3.3.24: Public non-static method signature

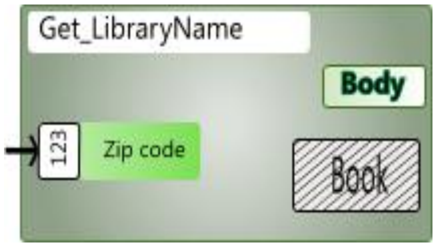


Figure 3.3.25:Public static method signature

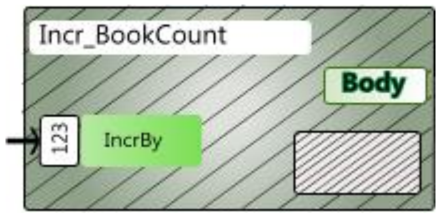


Figure 3.3.26: Private static method signature

3.7.4 Principles analysis of UVPL

Variables and literals

The table 3.3-1 summarizes the strategies and principles for variables and literals present in UVPL in comparison to those present in the selected VPLs.

	ALICE	ANALYTICA	MS VPL	TERSUS	UVPL
DIRECTNESS				✓	✓
EXPLICITNESS	✓	✓		✓	✓
LABELING	✓	✓			✓
PORTABILITY	✓	✓			✓
REGULARITY	✓		✓		✓
RESPONSIBLE DESIGN					✓
IMMEDIATE VIS. FEED.					
AUTOMATION			✓	✓	✓
SYNTACTIC CONSISTENCY		✓		✓	✓
DEFENSE IN DEPTH					
INFORMATION HIDING	✓	✓	✓	✓	✓
SECURITY	n/a	n/a	n/a	n/a	n/a
ABSTRACTION	✓	✓			✓
ELEGANCE					✓
SIMPLICITY			✓	✓	✓
IMPOSSIBLE ERROR			!		✓
ORTHOGONALITY	n/a	n/a	n/a	n/a	n/a
PRESERVATION OF INFO		✓	✓	✓	✓
STRUCTURE			✓		✓
0 – 1– ∞		✓	✓		✓

Table 3.3-1: Variables and literals principles

Strengths:

UVPL respects the automation principle for declaration and data type assignment, through which the user just drags, drops and names the groken to start using it. Unlike most VPLs, regarding variables, UVPL adopts a responsible design: the language provides different precisions for integers and floats. In UVPL, the variable groken will be resized automatically as the name grows, but to avoid having really long grokens — as in Alice – the names of the variables are constrained to be no longer than 25 characters.

The variable design in UVPL addresses the lack of direct manipulation found in Alice, Analytica and MS VPL, as well as the violation of responsible design, simplicity and elegance principles.

Weaknesses:

UVPL does not provide immediate visual feedback. This VPL strategy, proposed by Burnett, has not been considered for UVPL as a tradeoff for less disruption during the edit of a visual program, but also for better efficiency. As Burnett, et al. mentioned, the need to provide responsiveness can affect the efficiency of a program.

Table 3.2-2 represents principles and strategies of UVPL and the selected VPLs used in this research.

	ALICE	ANALYTICA	MS VPL	TERSUS	UVPL
DIRECTNESS	✓	✓	✓	✓	✓
EXPLICITNESS	✓	✓	✓	✓	✓
LABELING	n/a	n/a	n/a	n/a	n/a
PORTABILITY	✓	✓	✓	✓	✓
REGULARITY		✓	✓	✓	✓
RESPONSIBLE DESIGN	✓		✓	✓	✓
IMMEDIATE VIS. FEED.					
AUTOMATION	n/a	n/a	n/a	n/a	n/a
SYNTACTIC CONSISTENCY	✓	✓	✓	✓	✓
DEFENSE IN DEPTH			✓		
INFORMATION HIDING	✓	✓	✓	✓	✓
SECURITY	n/a	n/a	n/a	n/a	n/a
ABSTRACTION	✓		✓	✓	✓
ELEGANCE	?			✓	✓
SIMPLICITY	✓			✓	✓
IMPOSSIBLE ERROR	✓			✓	✓
ORTHOGONALITY		✓		✓	✓
PRESERVATION OF INFO	n/a	n/a	n/a	n/a	n/a
STRUCTURE	✓		✓	✓	✓
0 – 1– ∞	n/a	n/a	n/a	n/a	n/a

Table 3.3-2: Principles for operations

Strengths:

The function of an operation always is represented explicitly on the groken. The operations in UVPL are designed with syntactic consistency at different levels. UVPL is not designed with defense in depth mechanisms; instead, the impossible error principle is implemented.

Weaknesses:

By choosing a design in which the user does not type expressions as in a TPL, the expressions in UVPL tend to occupy more space than in the other selected VPLs. Nevertheless this choice was necessary to allow direct manipulation of operations, less abstraction and better reviewing of visual programs. To overcome the screen space issue, UVPL introduces the concealing and revealing of expressions described in section 0.

Iteration

Table 3.3-3 is a summary of the principles present in UVPL in comparison to the selected VPLs.

	ALICE	ANALYTICA	MS VPL	TERSUS	UVPL
DIRECTNESS	✓		✓	✓	✓
EXPLICITNESS	✓			✓	✓
LABELING	n/a	n/a	n/a	n/a	n/a
PORTABILITY	✓	✓		✓	✓
REGULARITY	n/a	n/a	n/a	n/a	n/a
RESPONSIBLE DESIGN	✓		✓		✓
IMMEDIATE VIS. FEED.					
AUTOMATION	✓	✓			
SYNTACTIC CONSISTENCY			✓		✓
DEFENSE IN DEPTH					✓
INFORMATION HIDING	✓	✓	✓	✓	✓
SECURITY	✓				✓
ABSTRACTION	✓		✓	✓	✓
ELEGANCE					✓
SIMPLICITY				✓	✓
IMPOSSIBLE ERROR	✓				✓
ORTHOGONALITY	n/a	n/a	n/a	n/a	n/a
PRESERVATION OF INFO	✓	✓	✓	✓	✓
STRUCTURE	✓		✓		✓
0 – 1– ∞	n/a	n/a	n/a	n/a	n/a

Table 3.3-3: Principles and Strategies for Control Flows

Strengths:

Elements such as counters and conditional statements are part of the iteration viprocons; this visual information clearly indicates the type of the control flow and the expected behavior. The responsible design principle has been respected since the programmer does not have the ability to change the keyword of the viprocons (e.g. If, For, While).

UVPL implements a soft version of the impossible error and defense in depth principles:

1. Impossible error: for a *while* loop the compiler checks and issues a warning if the value(s) of the variable(s) that define whether or not the loop continues are not being modified in some fashion.

2. Defense in depth: if a program segment is looping for a fairly large number of times, the system will issue a warning to caution the user about the possibility of an infinite loop. A default number is used to set off the warning; however the user has the option to set different numbers for different loops.

None of these measures can prevent completely issues such as infinite loops, but they can contribute to avoiding them. Because infinite loops cannot be avoided completely, it can be concluded that UVPL has only some level of security.

<u>Weaknesses:</u>

For control flow UVPL does not provide immediate visual feedback or automation such as automatically creating undeclared counter variables used in the control flow viprocons.

Input / Output

Table 3.3-4 presents the principles followed in UVPL in contrast with the ones followed in the selected VPLs, regarding I/O.

	ALICE	ANALYTICA	MS VPL	TERSUS	UVPL
DIRECTNESS			✓	✓	✓
EXPLICITNESS	✓		✓	✓	✓
LABELING	n/a	n/a	n/a	n/a	n/a
PORTABILITY	n/a				✓
REGULARITY	n/a	n/a	n/a	n/a	n/a
RESPONSIBLE DESIGN	✓		✓	✓	✓
IMMEDIATE VIS. FEED.					
AUTOMATION	n/a	n/a	n/a	n/a	n/a
SYNTACTIC CONSISTENCY					✓
DEFENSE IN DEPTH			✓	✓	
INFORMATION HIDING	✓		✓	✓	✓
SECURITY	✓	✓	✓	✓	✓
ABSTRACTION	✓	✓	✓	✓	✓
ELEGANCE			✓	✓	✓
SIMPLICITY	✓		✓	✓	✓
IMPOSSIBLE ERROR	✓	✓			✓
ORTHOGONALITY	n/a	n/a	n/a	n/a	n/a
PRESERVATION OF INFO	✓	✓	✓	✓	✓
STRUCTURE	✓		✓	✓	✓
0 – 1– ∞	n/a	n/a	n/a	n/a	n/a

Table 3.3-4: Principles for I/O

Strengths:

Unlike Tersus, the user is not allowed to change the name of the I/O viprocon. The visual structure of the viprocons helps in visualizing their behaviors; for instance, in UVPL a write viprocon has in and out parameters symbolized by arrows to show the flow of data.

Weaknesses:

UVPL does not have as many file I/O operations as Tersus; for instance, UVPL does not have XML parsing or PDF file generation.

Chapter IV

TESTING

4.1 Program Tests

The analysis of the selected VPLs using principles defined by McLennan, and strategies defined by Burnett, helped identify the strengths and weaknesses of the selected VPLs. The result of this analysis was the basis on which UVPL was designed, by avoiding – where possible –the weaknesses and by incorporating the strengths identified by the analysis of the VPLs.

The next phase of this research involves implementing a test program in each language, i.e. in the selected VPLs and UVPL. A quantitative analysis is performed, whereas various metrics are computed for each implementation. These metrics are used to determine how UVPL measures against the selected VPLs in achieving scalability.

A specific test program is designed, because most standard test programs used in research focus on the performance of the languages rather than on the scalability of the language. No standard test program for comparing programming languages for scalability is yet available.

It is important to note that benchmarking programming languages is a difficult task. Indeed, the ideal way to achieve this task is to implement the test program using the exact same algorithm in each language. However, every programmer has her/his own programming style, which means that there are multiple ways to implement an algorithm. Furthermore, programming languages are designed differently, and this implies that it might be more appropriate to use a particular construct in one language, but in another language a different construct is more suitable to serve the same purpose.

Taking into account these facts about the difficulty of conducting benchmarks on programming languages, a few rules are drawn to conduct this step of the research to obtain meaningful results:

- All programs are implemented by the same person; this insures that the same programming style is kept across the different implementations.
- Programs are implemented using programming constructs or data types that best fit the language. In other words, the programmer is not required to build an abstract data type if the language does not provide it, just so that s/he can use the same data types used in other implementations.
- If a language does not provide a necessary feature – for instance, the capability to read a file – the programmer uses workarounds rather than eliminating the language.

The idea behind the designed test program is to perform simple yet common tasks. The test program ensures that, where possible, the following actions are performed:

- Use of objects such as primary data types and data structures
- Value assignments
- Execution of arithmetic, comparison and Boolean operations
- Use of iterations and conditional jumps
- Use of libraries such as math or string libraries

- Create and invoke methods
- Create and instantiate classes
- Perform I/O operations
- Handle exceptions
- Comment code

The program reads and stores the records from a file of employee data in an array or a list. The file has five records and each record has five fields concerning an employee:

- ID
- last name
- first name
- the number of hours worked in a given month
- pay rate

The program also should read and store the records from a file containing data for five states, where each state's record contains:

- ID
- name
- tax rate as a percentage
- minimum wage.

For each employee, the first and last names should be displayed, and then the user is asked the name of the state used to compute the wages of that employee. If the pay rate of the employee is less than or equal to zero, an exception should be thrown. If the pay rate of the employee is less than the state's minimum wage, then the state's minimum wage should be used in place of the pay rate to compute the wage. The wage is computed using the following formula:

hours worked * pay rate – hours worked * pay rate * state’s tax rate / 100

The result is rounded to the nearest integer value and displayed to the user of the program.

The subsequent sections of this chapter aim at describing and discussing the implementation of the test program in the different selected languages and in UVPL.

4.1.1 Program Test in Alice 2.2

Classes in Alice are represented by animals, people and other 3D objects that move, spin or react to the mouse, and thus are not necessarily suitable to create, for example, an employee object. Each Alice program has a class “world”; some of the properties of this class are “atmosphereColor” and “fogStyle”. The “world” class basically defines the environment in which the 3D objects interact. The programmer can define more properties, methods or functions.

The test program is implemented in Alice with workarounds, because Alice 2.2 does not support file I/O. Instead of reading a file, the program’s specifications are modified to read the fields of the records one by one through a dialog box. Alice 2.2 does not have error handling mechanisms either; in this test program, the execution of the program is stopped after an error message is displayed to the user. The test program is implemented with 4 methods and 2 functions added to the “world” class.

- Method *MainEntry* is the equivalent of a Java or C main static method that specifies where the execution of a program should start. Alice does not have this concept, but to specify where the program should start executing, the programmer needs to create an event, as shown in Figure 4.1.1: Events in Alice.

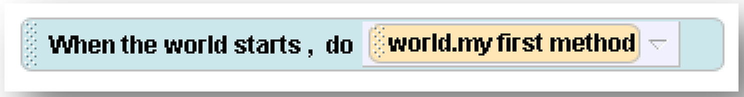


Figure 4.1.1: Events in Alice

- Method *ReadEmps* prompts the user to enter the different fields for the employee records.

- Method *ReadStates* prompts the user to enter the different fields for the state records.

- Method *DisplayResults* takes as input 3 arrays of the same size, used to simulate a 3-dimensional array, and displays the content of the rows of the arrays. This method is called in the MainEntry method to display the first name, last name and computed wage of employees.

- Function *GetStateTax* returns the tax percentage of a given state.

- Function *GetStateMinWage* returns the state minimum wage of a given state.

4.1.2 Program Test in Lumina Analytica 4.2

Similar to Alice, the implementation of the program in Analytica is performed with workarounds. File I/O in Analytica is provided only for the paid professional edition. For the freely-available version of Analytica used in this research, the records of employees and states are provided as initialized values to the program, instead of a file. These values are used to simulate the content of a file, and thus records are read one by one using the “spliText " function provided by Analytica. In turn, each record is split again to capture the different fields of an employee or state record. Analytica is not an object-oriented VPL, and thus no classes are created, and the program is procedural. Variables in Analytica are defined such that the definition of the variable itself is either a piece of textual code or values; the use of these variables triggers the execution of the code if it is present in the definition. The result of the computation is used instead of the variables. In other words, variables are manipulated like functions when they have code as their definition.

The program is implemented with 7 Analytica variable grokens, and 1 function viprocon

- The variable *Employees* contains all the employee records as in a text file
- The variable *States* contains all the state records as in a text file
- The variable *EmpRecs* holds the employee records; it is a result of the "spliText" function on the Employees.
- The variable *StateRecs* holds the state records.
- The variable *EmpFields* holds the fields of a given employee; it is the result of the "spliText" function on a row in *EmpRecs*. This variable takes as input an index.
- The variable *StateFields* holds the fields of a given state; it is the result of the "spliText" function on a row in *StateRecs*.
- The variable *Display* calls in a loop the procedure *ComputendDisplay.*
- The procedure *ComputeandDisplay* is called with an employee variable, with *StateRecs* and with *StateFields*. This procedure performs all the processing and rounds the value of the employee's wages.

4.1.3 Program Test in Microsoft VPL 2.1

The implementation of the test program in MS VPL encompasses most of the programming features listed earlier. Because MS VPL is not an object-oriented language, the program is built solely with procedure-like objects, called activities. The *Diagram* holds the entire data-flow that represents the program. Only activities have input and output pins to receive input data and send result data. MS VPL does not provide text file I/O; however the platform allows a programmer to create easily decentralized software services (DSS) to perform tasks that are not part of the language as provided. In fact most of the library items in MS VPL are DSS. DSS items are lightweight, state-oriented, service models, and they can be modified effortlessly by the programmer. Therefore, in this test, a DSS item is created to read text files. There are no exception handling mechanisms in MS VPL; the programmer needs to validate the data and branch to the end of the data-flow in order to stop the execution of the program, should a catchable error be detected.

The test program is implemented with 5 activities:

- The *GetListOfRecords* activity returns a list of records from a file; the input is the pathname of the file.
- The *GetStateInfo* activity parses and returns the tax and the minimum wage of a state, given a list of states and the index of a given state in that list.
- The *GetEmpInfo* activity parses and returns the hours, pay rate, last name, and first name of an employee, given an employee record.
- The *CompSalary* activity computes and returns a salary given the hours, a pay rate, a state's minimum wage and a tax rate.
- The *ProcessEmployee* activity accepts as input a list of employees, a list of states, and the index of the employee to process. It extracts from its input only the data needed to compute the wage of the employee. In return the activity provides a formatted string of the processed employee data and the calculated wage.
- In the main Diagram the files of employee and state data are read, and the records are stored in a list – MS VPL does not provide arrays. Those lists are used to process all the employees. For each employee, the result is displayed.

4.1.4 Program Test in Tersus 1.3

The Tersus VPL is not object-oriented; however, the user can use *systems* to group logically method-like entities called *actions.* Tersus does not provide exceptions handling mechanisms, and thus the programmer needs to handle properly any possible exceptions that can occur in the program. There are not any mechanisms for adding comments in a Tersus program. A solution could be to use a text literal groken to add comments; however those comments will not be tied logically to any portion of the diagram.

The test program in Tersus is implemented with 7 actions:

- The *GetAState* action takes as inputs the path of the file of state data and a state ID; the action returns the minimum wage and the tax rate percentage of that state.

- The *GetAnEmployee* action takes as inputs the path for the file of employee data and the ID of the employee to process. This action returns the last name, the first name, the hours worked, and the pay rate of the employee.

- *MakeACaption* is an action that is used to format the output question used in the UI for a user of the program, given a last name and a first name.

- The *AskState* action generates an interactive webpage to capture the answer from the user, when the user is asked the state to be used to process an employee.

- The *ComputeEmpSalary* action returns a computed salary given a number of hours worked, a pay rate, and a tax rate percentage.

- The *OneRound* action calls *GetAnEmployee,* asks the user which state to use for processing, decides which pay rate to use, calls *ComputeEmpSalary* and displays to the user the computed salary for a given employee.

- *ProcessAllEmployee* calls in a repetitive mode – which is how Tersus VPL performs loops – the action *OneRound*.

4.1.5 Program Test in UVPL

The test program in UVPL is implemented, but is neither compiled nor executed because only the language specifications have been defined in this research. The definition of this language allows the manipulating primitive and non-primitive data structures the use of various types of operations and libraries, and the performance of I/O operations as well as conditional jumps and iterations. UVPL also provides the means to add comments within the program.

UVPL has some features of object-oriented programming: data abstraction, encapsulation and modularity. The test program in UVPL is represented with 3 classes, and 21 methods.

- The class *Employee* has properties EmpID, LastName, FirstName, Hours, Payrate and Salary. Each property has an accessor and a mutator. The class *Employee* also has a method *Compute_Salary* to calculate the salary of an employee object.

- The class *State* has properties StateID, StateName, Tax and Minwage; and each property has an accessor and a mutator.

- The class *Main* is a static class, and has 4 methods:

    - *LoadEmployees* is a method that takes as input the pathname for a file of employee data and loads into an objects array representing employees.

    - *LoadStates* takes as input the pathname for a file of state data and loads into an array objects representing states.

    - *Process1Emp* takes as input an employee object and processes it by gathering information to compute salary and by calling *Compute_Salary* for that employee. This method returns the modified employee object.

    - *ProcessAllEmps* calls in a loop *Process1Emp* for each employee object in the array.

4.1.6 Analysis of the Program Tests

This step of the research allows hands-on interaction with the selected VPLs, and that permits further identification of features not provided in those languages.

None of the visual languages selected provide exceptions handling mechanisms. This feature is important to the ability to scale up a program. When a language does not provide a way to handle run-time errors, the programmer needs to perform more validations for possible run-time errors such as division by zero, and also needs to provide appropriate responses. However this practice leads to adding to the program code that is not part of the algorithm. As a result, there is no separation of the algorithmic code from the error handling code, which tends to add avoidable complexity to the program.

Commenting is another programming element that is provided neither in Analytica nor Tersus; on the other hand, Alice and MS VPL provide a means for adding comments. Nonetheless, comments in MS VPL lack structure, because a particular comment does not belong to any part in an MS VPL diagram.

File I/O is not implemented in most of those VPLs probably because the use of those languages generally excludes the need of reading from files or writing to files. The approach taken by Tersus is to provide different viprocons for reading and writing files of different formats; so Tersus has viprocons to read and write text files, to load a CSV text or an Excel sheet into a Tersus table, to parse an XML document or serialize a data structure as an XML document, to create or parse a JSON, to concatenate PDFs, etc.

VPLs usually allow modular programming, but they seldom provide OOP features. Among the selected VPLs, only Alice provides a simplistic version of OO programming, by focusing more on the concepts of objects in a story-telling context. Nowadays, OOP plays an important role in scalable computing. OOP allows reuse of objects and a better way to modify programs, since the visibility of methods in classes can be limited and modifying one object does not necessarily affect another object. OOP contributes as well in maintaining programs, because again classes can be maintained separately.

4.1.7 VPL Metrics for the Test Programs

The test programs in the four selected VPLs and in UVPL are evaluated and compared using the following metrics:

- The program volume
- The program visual density
- The ratio of vocabulary to total visual components

- The average number of connectors per container

- The average deepest browsing level.

These metrics are computed using operands, operators, connectors and containers further defined as followed:

Any labels – textual or graphical – in a groken or a viprocon that can be edited by the programmer are counted as operators because they either convey a piece of information about the type or can be considered to carry the same weight as a comment. However if a label in a groken or a viprocon cannot be edited by the programmer, it is not counted as part of the language. For instance, programs in Analytica have additional property windows to define further the attributes of an object; in those windows there are labels such as *Unit* or *Definition that* cannot be edited by the user. These labels are part of the UI, not the language.

An instance of a class is counted as an operand. The methods of a class are counted as operators.

An arrow is an operator, and arrows serving different purposes or arrows with different labels are counted individually.

A groken can be an operator and an operand at the same time; this happens in cases where the groken is an operand but information such as the type of the operand is embedded in the same groken.

Any declared variables that are not used in the program are not counted as part of the program.

A container is counted as a pair of parenthesis, thus as an operator.

In compound statements, each atomic entity is counted; however a user-entered string is counted as one operand – from opening to closing string markers – and as one operator for comments – from opening to closing comment markers.

Uniqueness is at the module level; i.e. a global variable is counted once throughout the program, and a local variable is counted once within its scope. By doing so, variables of the same name in different methods are counted once in each method.

Pieces of code that can be disabled, such as found in Alice, are not counted.

Variables that are part of a function’s signature are counted as operands. In the particular case of Analytica, variables that have literals as their value are counted as operands, and variables that have executable code as their value are counted as operators.

The VPL metrics listed earlier are described as followed:

- The *program volume* corresponds to the number of screens necessary to visualize the entire program, under the default settings of the system. The visual elements are neither maximized nor minimized; and those elements also are neither magnified nor reduced. This metric is used primarily in this research to compute other metrics. The *program volume* by itself is not an accurate measurement for comparing the size of the implementation of the same algorithm in different VPLs.
- The *visual density* is the average number of visual components per screen*.* It is the total number of components in a VPL divided by the *program volume*. Compared to the *program volume,* it gives a more accurate indication of the density of a program. A high value could be an indication that the program produced is very dense; such programs are difficult to review because they have a high concentration of visual elements, and the user may find it difficult to navigate through the program. A low value could be an indication that the program produced is very sparse; such programs also can be difficult to review because the user needs to flip between many screens.
- The *vocabulary* is the count of distinct operators and operands. The vocabulary size by itself is not a useful metric, because its meaning or importance is relative to the size of the program.
- The *ratio of vocabulary to total visual components* indicates the level of a VPL. The lower the ratio, the more frequently operators and operands are repeated in the program. Low-level languages have, in general, a small vocabulary and programs

written in those languages are, in general, harder to understand. A high ratio is a sign that the language has too much visual abstraction.

- The *average number of connectors per container* is used to get an insight into the visual complexity of a visual program. The higher this value is, the more connectors a container has. The total number of connectors is not used because this metric by itself cannot reflect the visual complexity.

- The *average deepest browsing level* is the depths to which the user must go on average to visualize parts of a visual program. If a program is symbolized as multiple sets of Russian nesting dolls, each doll and its contents being a subset of the program, this metric would correspond to the average number of Russian dolls to open to get to any given doll. This metric is important because it reflects how much of a program is visually abstracted to the viewer. The lowest average is one – meaning there is no need to browse any deeper – and there is no upper bound.

4.1.8 Test Programs Counts

The counts of operators, operands, connectors and containers are gathered from the implementations of the test algorithm in Alice, MS VPL, Tersus and Analytica, as well as from the representation of the test algorithm in UVPL. Table 4.1.8-1 shows the corresponding counts for each language.

	Alice	**Ms VPL**	**Tersus**	**Analytica**	***UVPL***
Total number of operators N1	*391*	369	389	276	*480*
Total number of operands N2	*198*	176	30	152	*245*
number of distinct operators n1	125	113	*171*	127	*185*
number of distinct operands n2	87	*90*	29	85	*94*
Vocabulary n1 + n2	212	203	200	212	*279*
Total Program components N1+ N2	589	545	419	428	*725*
Total # of containers	23	19	24	25	*136*
Total # of connectors	0	156	85	6	*86*
Program volume	6	7	7	14	*28*

Table 4.1.8-1: Test Programs counts

4.1.9 VPL Metrics Values for the Test Programs

Each metric is used to evaluate how UVPL performs compared to the selected VPLs. This analysis is based on a single algorithm. An analysis based on multiple algorithms would give a more complete picture, but is beyond the scope of the current research. Nevertheless, this short analysis gives an insight of how UVPL could perform on small programs, and the result of the analysis could be used further to extrapolate how UVPL may perform on enterprise-size programs.

The results presented below are ordered from less desirable to more desirable, using the scheme shown in Table 4.1.9-1.

More desirable
Less Desirable

Table 4.1.9-1: Desirability order

Visual Density

In reference to Table 4.1.9-2, Tersus has a better performance. This result is important because it indicates that a program in Tersus may be easier to review. The high value for Alice points out that the program in Alice is dense. The low value for UVPL is explained by the fact that UVPL is an object-oriented language, and thus has more structures since the programmer defines classes and methods. Because the test program is relatively small, most of the features in UVPL are not used to its advantage. However, as programs become larger and more complex, one can predict that the UVPL program volume value will improve relative to the program volume values of the selected VPLs.

	Alice	UVPL	Analytica	MS VPL	Tersus
Visual density	98.17	25.89	30.57	77.86	59.86

Table 4.1.9-2: Visual Density

The Vocabulary to total visual components ratio (VTVC)

This ratio should be neither too high nor too low. A reasonable level of abstraction is important in achieving scalability, especially considering that the reviewer of a VPL might not be a seasoned programmer who can understand in a timely manner programs with very high abstraction levels.

As illustrated in Table 4.1.9-3, the values for all the languages are very close, Tersus and Analytica being respectively at the lowest and highest extremities, and UVPL lying in the middle. Given that the ratios lie at neither extreme, it can be concluded that all the VPLs including UVPL have adequate vocabulary to components ratios.

	Tersus	Ms VPL	UVPL	Alice	Analytica
vocabulary to visual components	0.36	0.37	0.38	0.48	0.49

Table 4.1.9-3: VTVC ratio

Average number of connectors per container

A low average is desirable, because a program with too many connectors is, in general, difficult to view.

Alice has a value of zero, as shown in Table 4.1.9-5, because this VPL does not use connectors to direct the execution flow of a program. The MS VPL test program has a lot of connectors and could be the hardest to review and this is reflected here by its value.

	Ms VPL	Tersus	UVPL	Analytica	Alice
average # of connectors per container	8.21	3.54	0.63	0.24	**0**

Table 4.1.9-5: Average Connectors per Container

Average deepest browsing level

On one hand, average deepest browsing levels that are close or equal to one are not desirable because that VPL may not support iconization for abstraction purposes. On the other hand, averages that are too high are not desirable either, because the program becomes then difficult to review.

Table 4.1.9-6 shows that, as expected, Tersus has the deepest browsing level since the user-interface is designed in such a way that the user needs to drill down to view details of any objects.

	Alice	Tersus	Ms VPL	UVPL	Analytica
avg deepest browsing level	1	2.43	2.14	1.43	2

Table 4.1.9-6: Average Deepest Browsing Level

As a summary, UVPL has:

- One of the worst program visual densities, because it is too sparse.
- An acceptable value for the vocabulary to total visual components ratio.
- An acceptable average for the connectors per container value.
- One of the best average deepest browsing levels.

These values are in accordance with the experience of the tester.

Chapter V

CONCLUSION

5.1 Findings

The objectives of this research have been to propose a visual language – UVPL— that could fulfill the need for a general-purpose, object-oriented, scalable, visual, programming language. The larger family of programming languages is the general-purpose one. This group of programming languages is dominated largely by TPLs. General-purpose programming languages are more popular because they can solve a wider range of problems. Unfortunately general-purpose VPLs have not had their breakthrough yet, thus the need for more research in this area. In that same line of thought, visual languages need to be designed with more object-oriented features to achieve scalable programs.

For this purpose, an analysis of the grokens and viprocons of Alice, Analytica, MS VPL and Tersus has been conducted. The results of that analysis were used as a basis to design UVPL, which is built upon the strengths of those languages, all the while avoiding their weaknesses. The focus of this research has been on the visual aspects of UVPL and its development environment that affect scalability of visual programs in general. New elements – non-existent in the selected VPLs – were introduced to ease the review and maintenance of UVPL programs and to address scalability issues.

To validate the result of the proposed programming language, UVPL, it has been compared to the selected VPLs using:

- A qualitative analysis: VPL strategies from Burnett and programming language principles from McLennan.
- A quantitative analysis: metrics relevant to scalability issues.

The result of the qualitative analysis shows the strengths and weaknesses of UVPL.

Strengths:

UVPL has automation for the declaration of variable grokens and adopts a responsible design approach for handling integers and floats: the language provides different precisions. The variable grokens can be resized to better manage screen real estate. UVPL allows direct manipulation of operations, less abstraction and better reviewing of visual programs. To overcome the screen space issue, UVPL introduces the ability to conceal and reveal expressions; this permits the user to confine an expression to a smaller space (concealing) and view part or all the expression as needed (revealing). Elements such as counters and conditional statements are embedded within the iteration viprocons to better indicate the type of the control flow and its expected behavior. The flow of data into and out of an iteration viprocon or a method is symbolized by arrows. The flow of control is symbolized by consecutive instruction boxes, top to bottom.

The strengths of UVPL work together for better scalability of the programs from the perspective of a novice programmer.

Weaknesses:

For variables and flow controls, UVPL does not provide immediate visual feedback during the editing of a program. This feature can provide responsiveness but it might have an effect on the efficiency of the editing process. In UVPL, the user does not type expressions; as a result, expressions in UVPL tend to occupy more space than in the other selected VPLs, but this weakness is offset by the ability to conceal or reveal expressions. UVPL does not have as many file I/O operations as Tersus; for instance, UVPL does not have XML parsing or PDF file generation.

The qualitative analysis of UVPL is rather subjective. Because this analysis cannot be used alone to determine whether or not UVPL has attained its objectives, a quantitative analysis was used in parallel. This analysis produced metrics used to rank UVPL and the selected VPLs. The following paragraphs present the results of that analysis.

The *program visual density* value of UVPL compared poorly to those of the selected VPLs; the test program in UVPL is too sparse compared to the implementations in the other languages. Paradoxically, this result is a good one for UVPL to some extent: UVPL is an object-oriented language, and thus harbors mechanisms to construct a well-modularized program. When building relatively small visual programs such as toy programs, the UVPL program will have more structures – and might be spread across more screens – than the same program in its counterparts. These structures are accessors, mutators and other methods for each class. They add volume to the program, but are necessary to follow an object-oriented approach. As the program is scaled up, the visual density metric is expected to improve for UVPL, which indicates that UVPL might be more suitable for large programs.

UVPL has an *acceptable value for the vocabulary to total visual components ratio.* This metric is used to determine the level of abstraction of a VPL. , Too much abstraction can be a drawback for a novice programmer, as the program might be harder to understand. The ratio value for UVPL implies that it has an adequate level of abstraction.

UVPL has an *acceptable average for the connectors per container value*. Compared to Tersus or MS VPL, a program implemented in UVPL is expected to be easier to decipher because it has fewer arrows. This is comparable to complex flow charts, which are difficult to understand because the reader needs to follow many connecting arrows between objects to understand the flow of the program.

UVPL has one of *the best average deepest browsing levels*. This metric signifies that the test program in UVPL is viewed more easily than a program in Alice, Tersus and MS VPL. Indeed, the user needs on average fewer clicks to reach any point of the program.

5.2 Goals achieved

The goals of this research were to propose a programming language that is visual, general-purpose, object-oriented and scalable. This section evaluates each of these goals.

UVPL is a *visual language*, but not a purely visual language. As mentioned in previous chapters, a purely visual language is not practical. Such languages are represented entirely with visual elements or symbols other than textual symbols, and thus virtually do not need a keyboard for implementing programs. Purely textual languages are languages entirely represented with textual symbols. On the scale between visual and textual languages, UVPL is closer to a purely visual language, because most of a program is constructed with grokens (graphical tokens) and viprocons (visual programming constructs). Textual symbols are needed only when naming a structure or assigning a literal value.

UVPL is a *general-purpose language*. UVPL is considered to be more general-purpose than Tersus or MS VPL, for instance, that are for web development and robotics, respectively. These languages are specifically designed for a single domain; web development or robotics problems are solved more easily with those domain-specific VPLs. Tersus and MS VPL are not the appropriate choices to solve decision-support problems; those would be better solved by VPLs such as Analytica or UVPL. Native libraries of the domain-specific languages usually have specialized functions to fit the nature of the language. If the programmer is allowed to build user-defined libraries, they generally are built on top of the specialized native libraries. However, in a general-purpose language, the native libraries are a support to solve a wide range of problems, and the user has more flexibility when building user-defined libraries.

UVPL is an *object-oriented language*. Even though UVPL is not a fully object-oriented language, the programs are built with classes, objects and their members. A UVPL object can be instantiated, and UVPL has abstraction and encapsulation. However, UVPL was not developed with inheritance and polymorphism. These features could be added to the language later. Novice programmers might not have a strong need for inheritance and polymorphism.

Because UVPL is a visual, general-purpose, object-oriented language, it should have the capabilities to produce scalable programs.

5.3 Halstead measurements

In a first attempt to gather quantitative measurements, Halstead complexity measures were used [17]. Even though this method was developed in 1977, it still is used by institutions such as the Metric Data Program of NASA and by Verifysoft Technology, a German company specializing in software testing [18-19]. In a nutshell, Halstead metrics are based on the fact that algorithms are made up of operands and operators only, and that it is possible to identify those operands and operators in the implementation of an algorithm in any language. Halstead states that the operators and operands are defined as symbols or combinations of symbols. Some of the measurable properties defined by Halstead are:

$\eta 1 \rightarrow$ The number of distinct operators

$\eta 2 \rightarrow$ the number of distinct operands

$N1 \rightarrow$ the total number of operators

$N2 \rightarrow$ the total number of operands

These measurable properties are used to compute metrics such as the program length ($V = (N1+N2) \operatorname{Log}_2(\eta 1+ \eta 2)$) and the estimated number of delivered bugs ($B = V / 3000$).

However Halstead complexity measures were developed at a time when computer programs were purely textual. Using these metrics on visual programs gives results that do not reflect or take into account the visual aspects of VPLs, and thus are not suitable to evaluate visual programs quantitatively. More research needs to be conducted to develop a set of standard metrics more appropriate for VPLs.

5.4 Future works

UVPL has not been specified formally in a grammar, because the research area of VPL grammars is still in its infancy. Research conducted by Marriot on constraints multiset grammars (CMG) give a sense of the difficulty in formally specifying a VPL using a grammar. This field of study needs to be more developed for VPLs to be defined properly in this way.

As mentioned earlier, this research focuses on the design of the visual aspects of UVPL. For this language to be fully functional, its design – as well as a compiler or an interpreter –needs to be implemented as well.

Since UVPL is a general-purpose VPL, it will need to be delivered with enough libraries such as math, text processing, regular expressions, database, security, etc. Concepts such as the sharing of libraries between different users could be introduced to allow the libraries of UVPL to grow faster.

The results of this research were validated using a single test program. To obtain a more accurate result, it will be necessary to implement in UVPL and the selected VPLs several test algorithms solving a large variety of problems. This will insure a more statistically accurate assessment of UVPL.

As stated in previous chapters, VPLs are tightly coupled to their development environment; thus testing their usability should be performed with human subjects. Such tests can be conducted using methods such as the *cognitive walkthrough*; this human/computer interaction (HCI) technique was proposed by T.R.G. Green and is used to help designers of VPLs detect the level of usability they have achieved [6] and correct usability problems on a user interface.

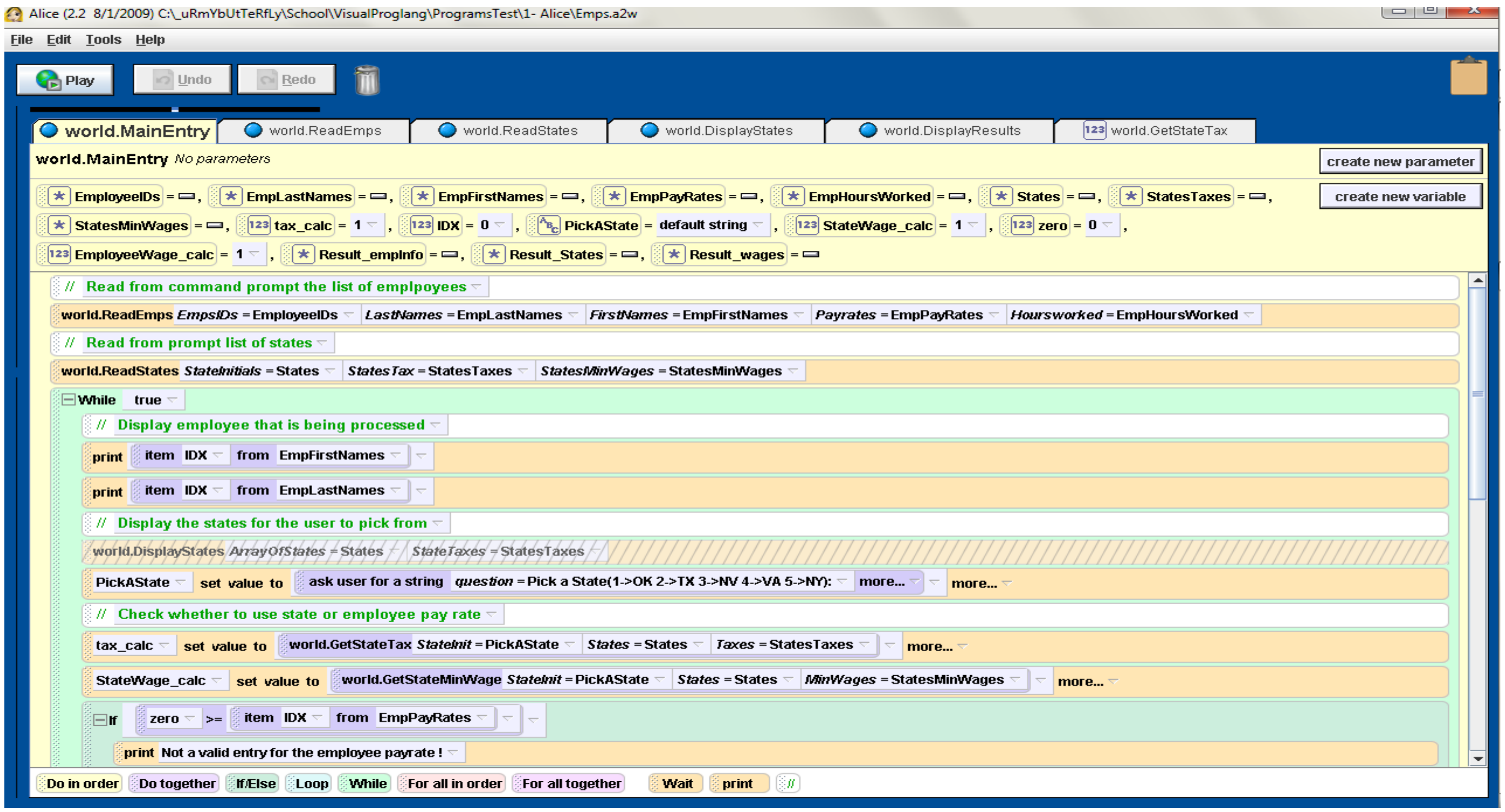
Is unifying currently-popular VPLs the best approach to design a general-purpose, object-oriented, scalable, visual, programming language? UVPL certainly achieved its goals for being a visual programming language that is general-purpose and object-oriented. However a more definite conclusion shall be made once UVPL is fully implemented and functional, once UVPL is tested using a statistical approach and once the UVPL user-interface is tested as well.

REFERENCES

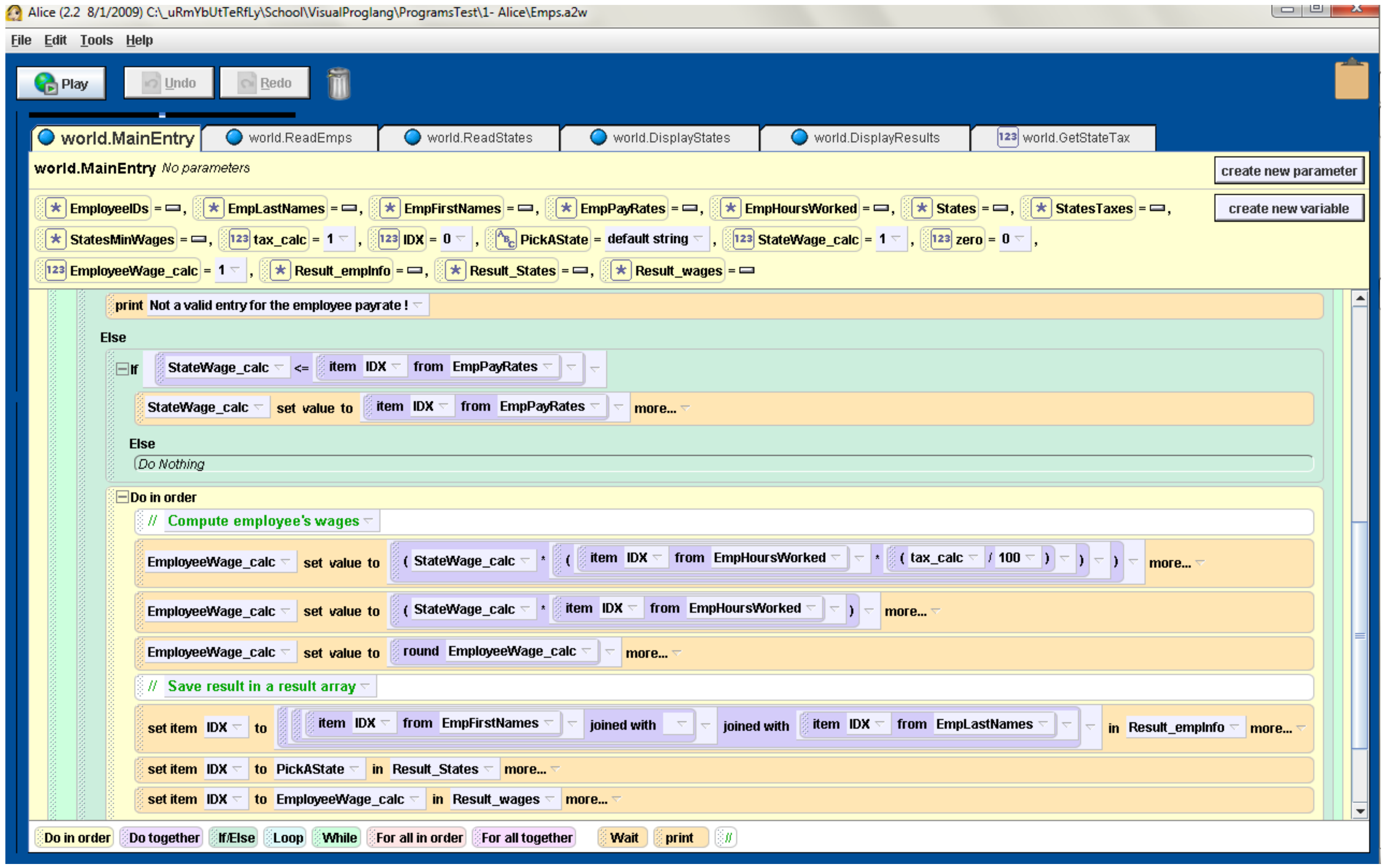
[1] M. Burnett, *Visual Programming*. New York, 1999.
[2] M. M. Burnett and M. J. Baker, "A Classification System for Visual Programming Languages," Oregon State University1993.
[3] ACM. (1998, 04/04/2010). *The 1998 ACM Computing Classification System — Association for Computing Machinery*. Available: http://www.acm.org/about/class/ccs98-html
[4] K. Marriott, "Constraint multiset grammars," in *Visual Languages, 1994. Proceedings., IEEE Symposium on*, 1994, pp. 118-125.
[5] B. C. Pierce, "Bounded quantification is undecidable," presented at the Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Albuquerque, New Mexico, United States, 1992.
[6] T. R. G. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A [`]Cognitive Dimensions' Framework," *Journal of Visual Languages & Computing,* vol. 7, pp. 131-174, 1996.
[7] R. B. Smith, "Experiences with the alternate reality kit: an example of the tension between literalism and magic," presented at the Proceedings of the SIGCHI/GI conference on Human factors in computing systems and graphics interface, Toronto, Ontario, Canada, 1987.
[8] M. D. Wayne Citrin, Benjamin Zorn, "The Design of a Completely Visual Object-Oriented Programming Language," *Visual Object-Oriented Programming,* 1994.
[9] P. T. Cox*, et al.*, "Prograph: a step towards liberating programming from textual conditioning," in *Visual Languages, 1989., IEEE Workshop on*, 1989, pp. 150-156.
[10] T. Pin-Ying*, et al.*, "A Visual Programming Language for Data Transformation," in *Computer Science and its Applications, 2008. CSA '08. International Symposium on*, 2008, pp. 96-101.
[11] M. J. B. Margaret M. Burnett, Carisa Bohus, Paul Carlson, Sherry Yang, Pieter van Zee, "Scaling Up Visual Programming Languages," *Computer 0018-9162,* vol. 28, pp. 45-54, 1995.
[12] R. Jamal and L. Wenzel, "The applicability of the visual programming language LabVIEW to large real-world applications," in *Visual Languages, Proceedings., 11th IEEE International Symposium on*, 1995, pp. 99-106.
[13] M. M. M. a. M. Porta, "Iteration constructs in data-flow visual programming languages " *Computer Languages,* vol. 26, pp. 67-104, 2001.
[14] A. Ambler, "The Formulate Visual Programming Language : Representing Structured Data," *Dr. Dobb's journal,* vol. 24, pp. 21-29, 1999.
[15] J. L. Leopold and A. L. Ambler, "A User Interface for the Visualization and Manipulation of Arrays," presented at the Proceedings of the 1996 IEEE Symposium on Visual Languages, 1996.
[16] B. J. MacLennan, "Principles of Programming Languages: Design, Evaluation, and Implementation," ed New York: Oxford University Press, 1999, p. 509.
[17] M. H. Halstead, "Elements of Software Science," ed Amsterdam: Elsevier Science Inc. New York, NY, USA, 1977, p. 128.
[18] J. Long. (2008, *NASA IV&V Facility Metrics Data Program - HALSTEAD METRICS:* . Available: http://mdp.ivv.nasa.gov/halstead_metrics.html
[19] (2007, March 30th). *Verifysoft → Halstead Metrics*. Available: http://www.verifysoft.com/en_halstead_metrics.html

APPPENDICES

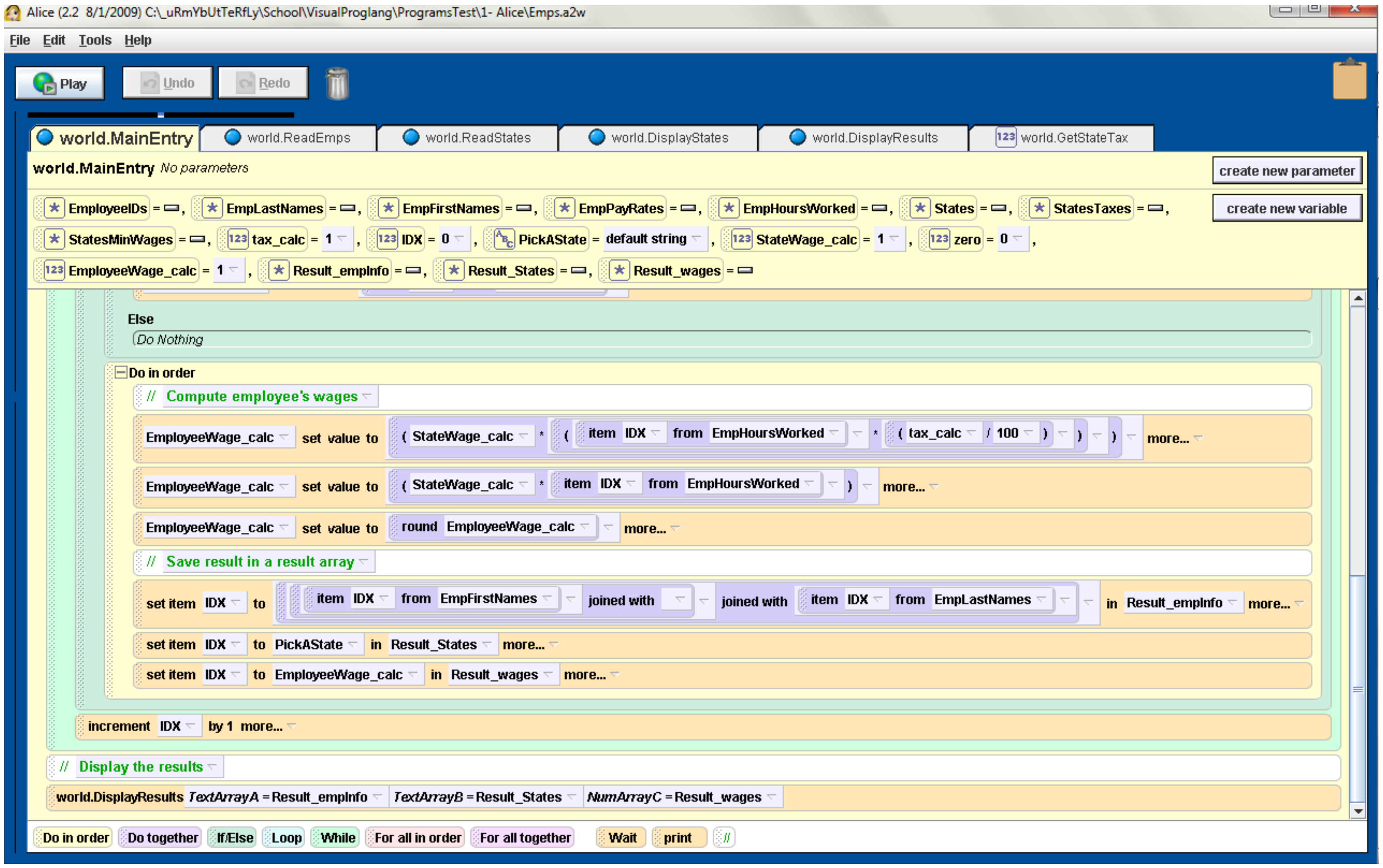
Appendix A: Program test in Alice


Alice (2.2 8/1/2009) C:_uRmYbUtTeRfLy\School\VisualProglang\ProgramsTest\1- Alice\Emps.a2w
File Edit Tools Help
Play Undo Redo
world.MainEntry
world.ReadEmps
world.ReadStates
world.DisplayStates
world.DisplayResults
world.GetStateTax
create new parameter
create new variable
Do in order Do together If/Else Loop While For all in order For all together Wait print //

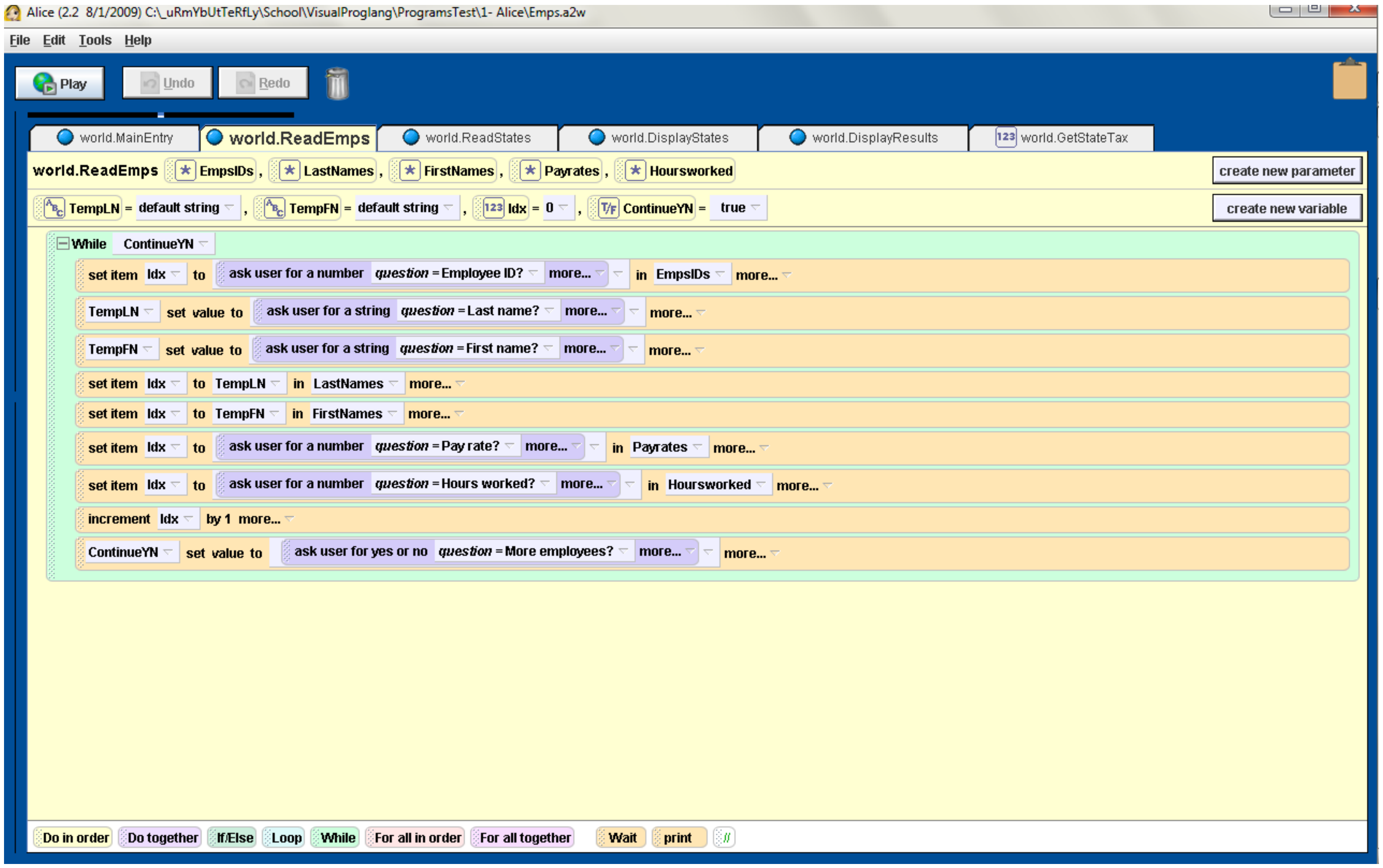
```
world.MainEntry No parameters
EmployeeIDs = , EmpLastNames = , EmpFirstNames = , EmpPayRates = , EmpHoursWorked = , States = , StatesTaxes = ,
StatesMinWages = , tax_calc = 1 , IDX = 0 , PickAState = default string , StateWage_calc = 1 , zero = 0 ,
EmployeeWage_calc = 1 , Result_empInfo = , Result_States = , Result_wages =

// Read from command prompt the list of emplpoyees
world.ReadEmps EmpsIDs = EmployeeIDs LastNames = EmpLastNames FirstNames = EmpFirstNames Payrates = EmpPayRates Hoursworked = EmpHoursWorked
// Read from prompt list of states
world.ReadStates StateInitials = States StatesTax = StatesTaxes StatesMinWages = StatesMinWages
While true
    // Display employee that is being processed
    print item IDX from EmpFirstNames
    print item IDX from EmpLastNames
    // Display the states for the user to pick from
    world.DisplayStates ArrayOfStates = States StateTaxes = StatesTaxes
    PickAState set value to ask user for a string question = Pick a State(1->OK 2->TX 3->NV 4->VA 5->NY): more... more...
    // Check whether to use state or employee pay rate
    tax_calc set value to world.GetStateTax StateInit = PickAState States = States Taxes = StatesTaxes more...
    StateWage_calc set value to world.GetStateMinWage StateInit = PickAState States = States MinWages = StatesMinWages more...
    If zero >= item IDX from EmpPayRates
        print Not a valid entry for the employee payrate !
```



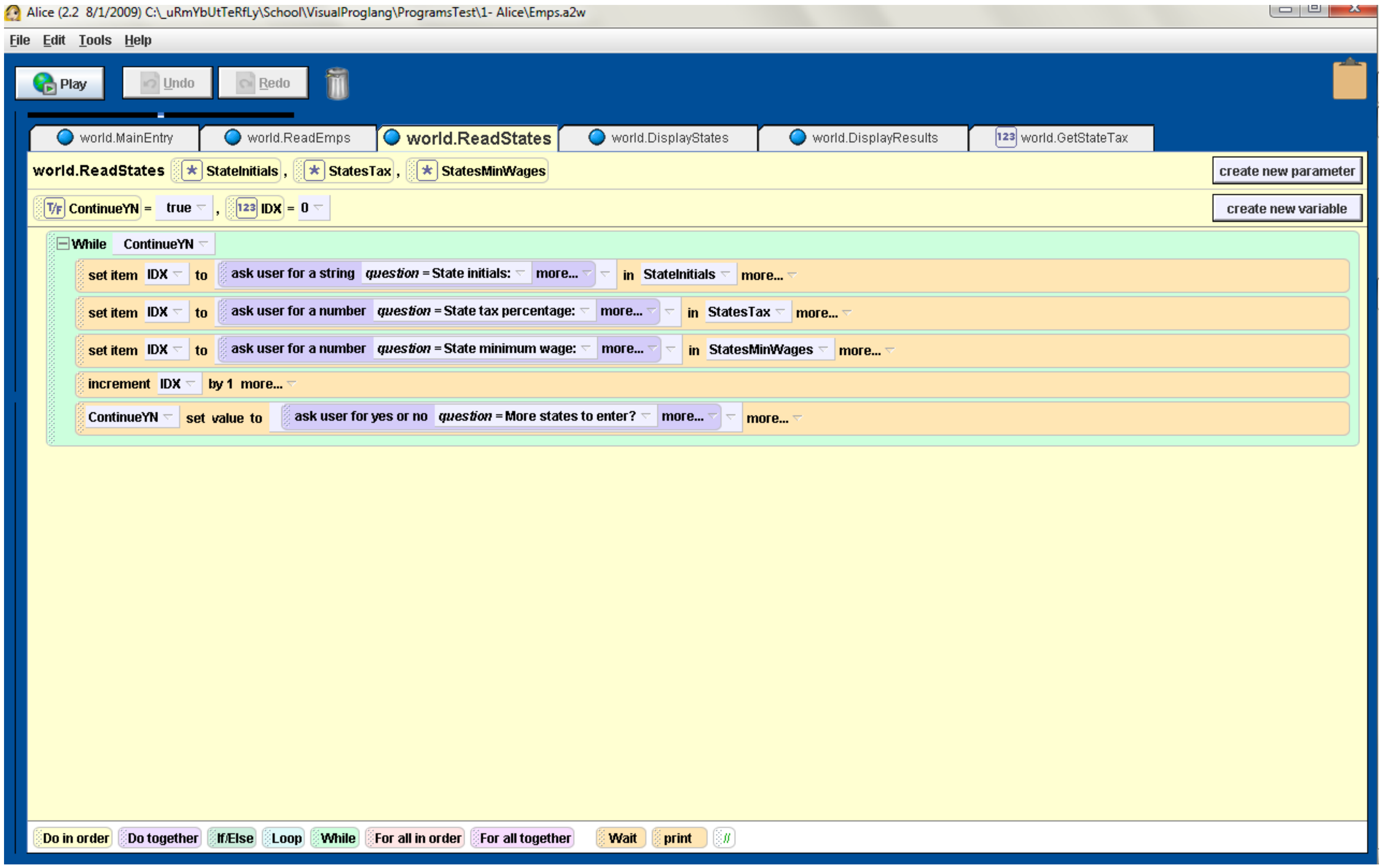
```
world.MainEntry   No parameters

EmployeeIDs = , EmpLastNames = , EmpFirstNames = , EmpPayRates = , EmpHoursWorked = , States = , StatesTaxes = ,
StatesMinWages = , tax_calc = 1 , IDX = 0 , PickAState = default string , StateWage_calc = 1 , zero = 0 ,
EmployeeWage_calc = 1 , Result_empInfo = , Result_States = , Result_wages =

      print Not a valid entry for the employee payrate !
    Else
      If StateWage_calc <= item IDX from EmpPayRates
        StateWage_calc set value to item IDX from EmpPayRates more...
      Else
        Do Nothing
    Do in order
      // Compute employee's wages
      EmployeeWage_calc set value to ( StateWage_calc * ( item IDX from EmpHoursWorked * ( tax_calc / 100 ) ) ) more...
      EmployeeWage_calc set value to ( StateWage_calc * item IDX from EmpHoursWorked ) more...
      EmployeeWage_calc set value to round EmployeeWage_calc more...
      // Save result in a result array
      set item IDX to item IDX from EmpFirstNames joined with   joined with item IDX from EmpLastNames in Result_empInfo more...
      set item IDX to PickAState in Result_States more...
      set item IDX to EmployeeWage_calc in Result_wages more...
```



```
Alice (2.2 8/1/2009) C:\_uRmYbUtTeRfLy\School\VisualProglang\ProgramsTest\1- Alice\Emps.a2w
File  Edit  Tools  Help
Play  Undo  Redo

world.MainEntry | world.ReadEmps | world.ReadStates | world.DisplayStates | world.DisplayResults | world.GetStateTax

world.MainEntry No parameters                                   create new parameter
EmployeeIDs = , EmpLastNames = , EmpFirstNames = , EmpPayRates = , EmpHoursWorked = , States = , StatesTaxes = ,   create new variable
StatesMinWages = , tax_calc = 1 , IDX = 0 , PickAState = default string , StateWage_calc = 1 , zero = 0 ,
EmployeeWage_calc = 1 , Result_empInfo = , Result_States = , Result_wages =

      Else
        Do Nothing
      Do in order
        // Compute employee's wages
        EmployeeWage_calc set value to ( StateWage_calc * ( item IDX from EmpHoursWorked * ( tax_calc / 100 ) ) ) more...
        EmployeeWage_calc set value to ( StateWage_calc * item IDX from EmpHoursWorked ) more...
        EmployeeWage_calc set value to round EmployeeWage_calc more...
        // Save result in a result array
        set item IDX to item IDX from EmpFirstNames joined with  joined with item IDX from EmpLastNames in Result_empInfo more...
        set item IDX to PickAState in Result_States more...
        set item IDX to EmployeeWage_calc in Result_wages more...
    increment IDX by 1 more...
  // Display the results
  world.DisplayResults TextArrayA = Result_empInfo TextArrayB = Result_States NumArrayC = Result_wages

Do in order | Do together | If/Else | Loop | While | For all in order | For all together | Wait | print | //
```

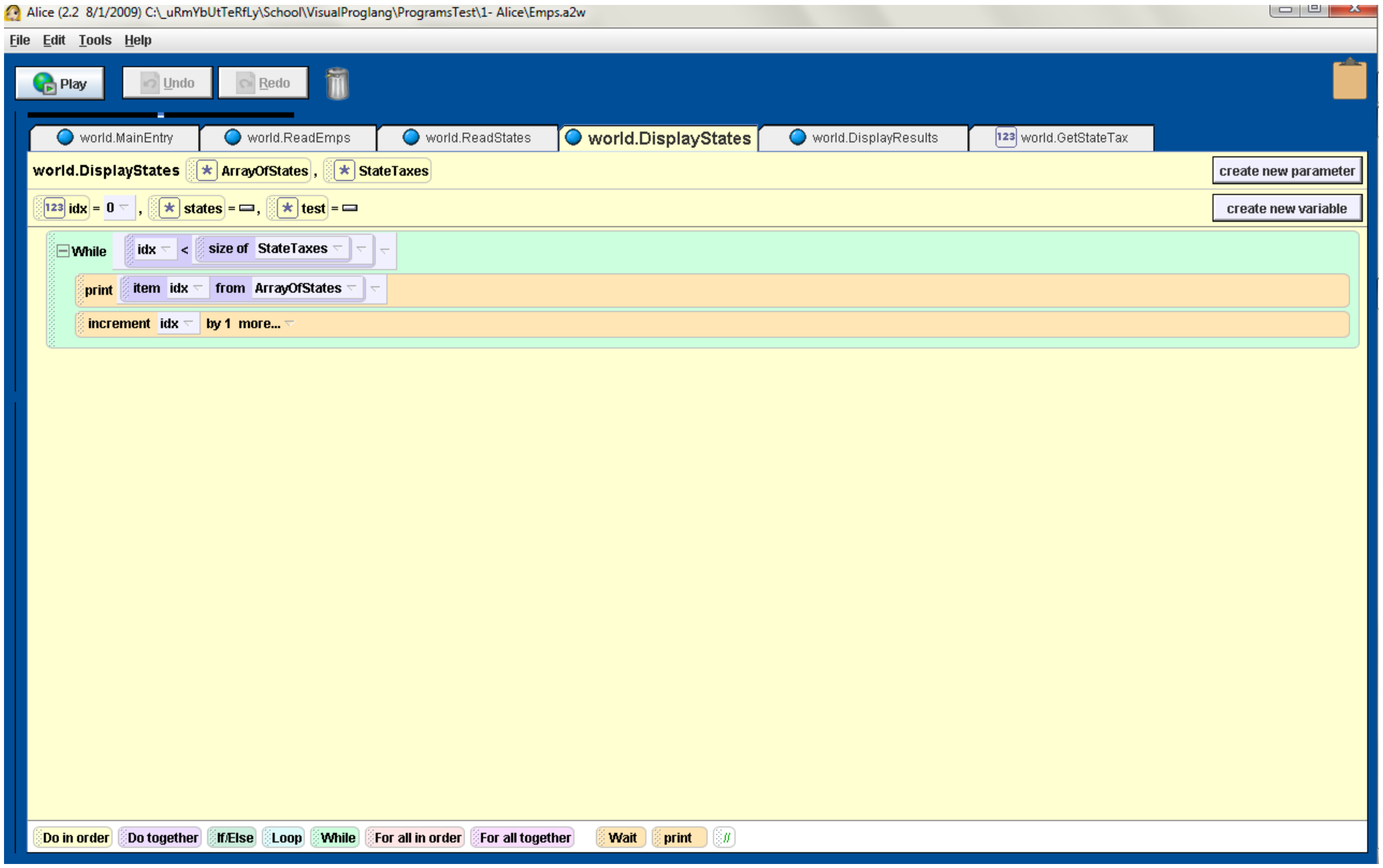

Alice (2.2 8/1/2009) C:_uRmYbUtTeRfLy\School\VisualProglang\ProgramsTest\1- Alice\Emps.a2w

```
world.ReadEmps EmpsIDs , LastNames , FirstNames , Payrates , Hoursworked
TempLN = default string , TempFN = default string , Idx = 0 , ContinueYN = true
While ContinueYN
    set item Idx to ask user for a number question = Employee ID? more... in EmpsIDs more...
    TempLN set value to ask user for a string question = Last name? more... more...
    TempFN set value to ask user for a string question = First name? more... more...
    set item Idx to TempLN in LastNames more...
    set item Idx to TempFN in FirstNames more...
    set item Idx to ask user for a number question = Pay rate? more... in Payrates more...
    set item Idx to ask user for a number question = Hours worked? more... in Hoursworked more...
    increment Idx by 1 more...
    ContinueYN set value to ask user for yes or no question = More employees? more... more...
```



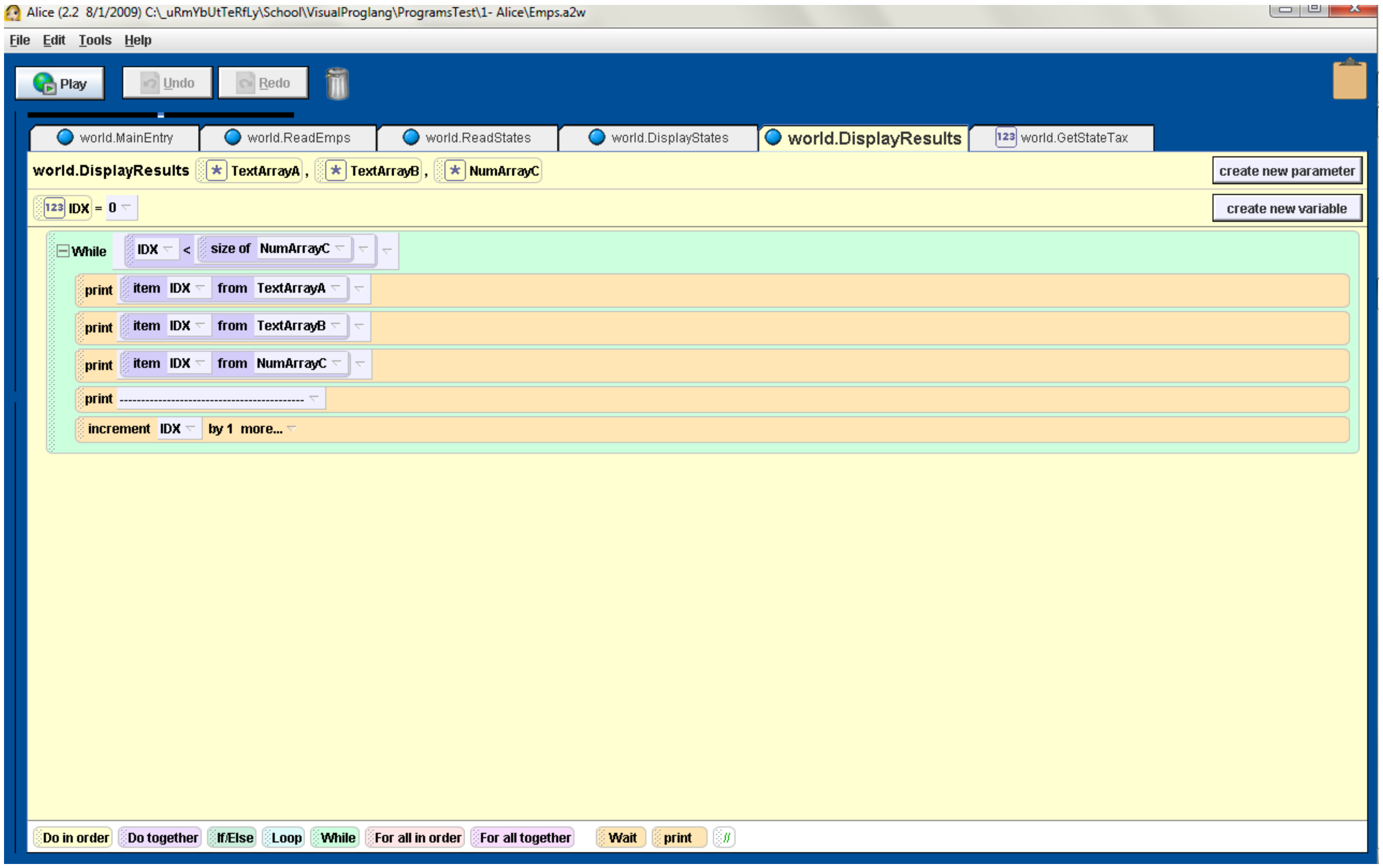
Alice (2.2 8/1/2009) C:_uRmYbUtTeRfLy\School\VisualProglang\ProgramsTest\1- Alice\Emps.a2w

```
world.ReadStates  StateInitials , StatesTax , StatesMinWages
ContinueYN = true , IDX = 0

While ContinueYN
    set item IDX to ask user for a string question = State initials: more... in StateInitials more...
    set item IDX to ask user for a number question = State tax percentage: more... in StatesTax more...
    set item IDX to ask user for a number question = State minimum wage: more... in StatesMinWages more...
    increment IDX by 1 more...
    ContinueYN set value to ask user for yes or no question = More states to enter? more... more...
```

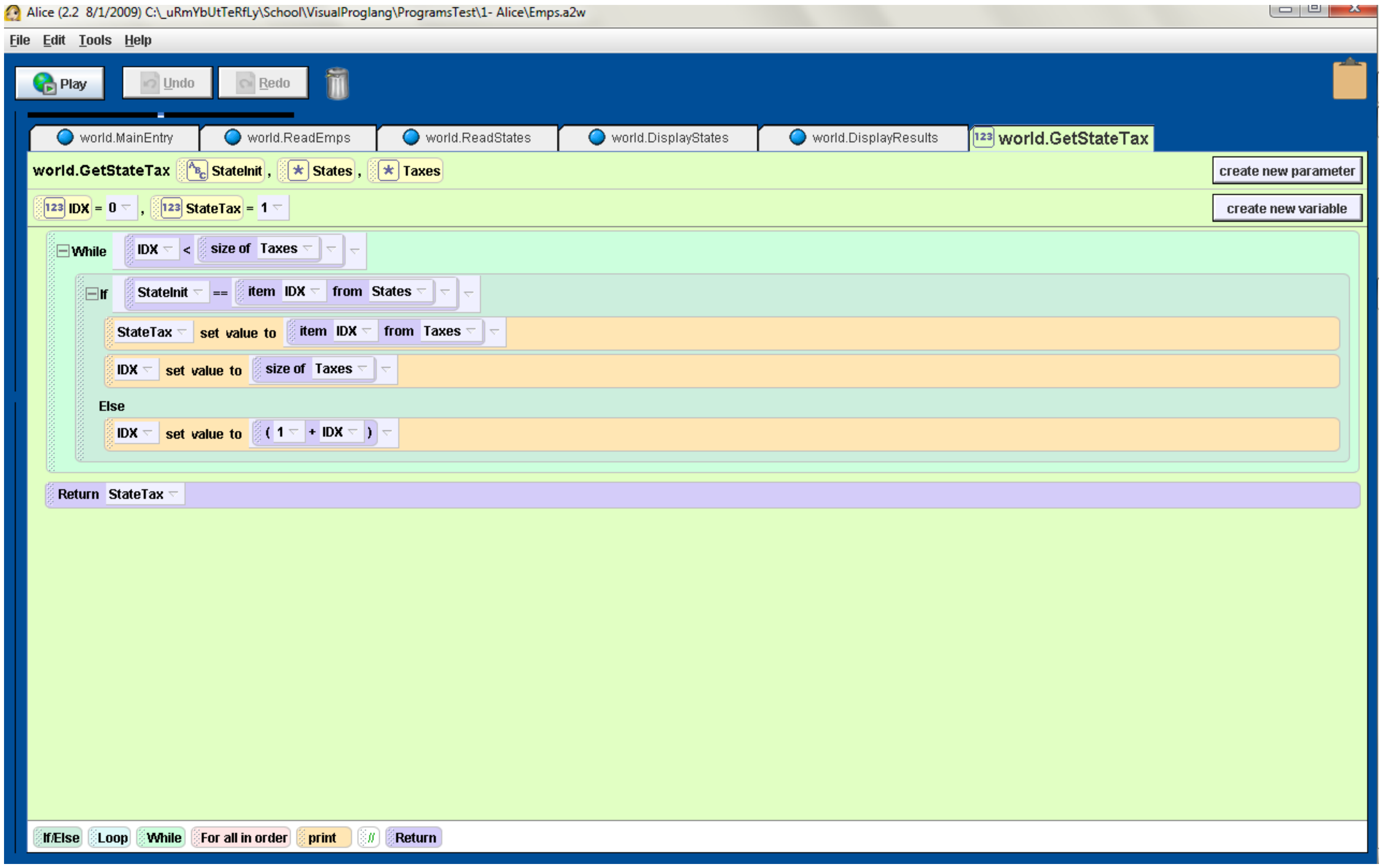


Alice (2.2 8/1/2009) C:_uRmYbUtTeRfLy\School\VisualProglang\ProgramsTest\1- Alice\Emps.a2w

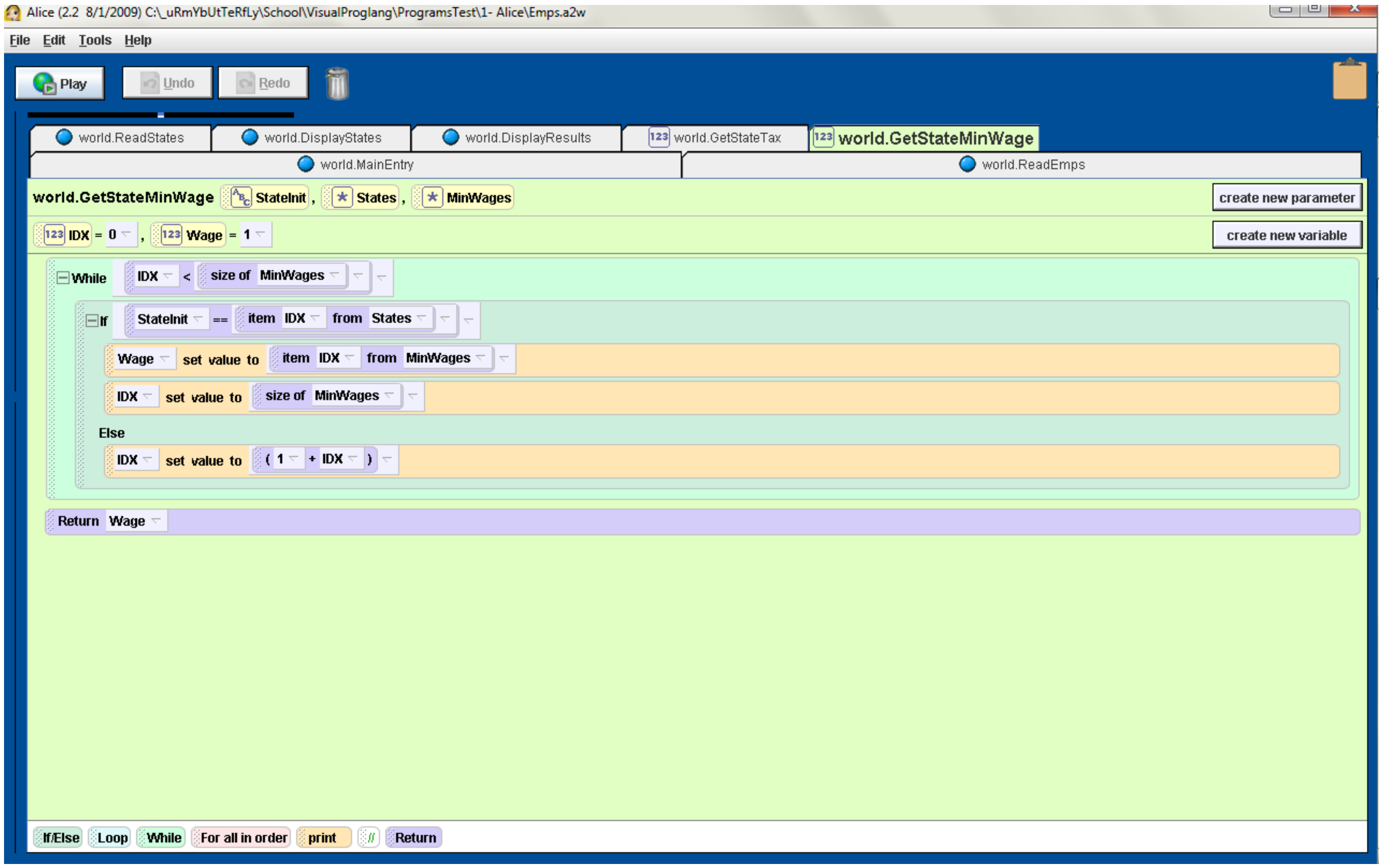
```
world.DisplayStates  ArrayOfStates , StateTaxes
idx = 0 , states = , test =
While  idx < size of StateTaxes
    print  item idx from ArrayOfStates
    increment idx by 1 more...
```


Alice (2.2 8/1/2009) C:_uRmYbUtTeRfLy\School\VisualProglang\ProgramsTest\1- Alice\Emps.a2w

```
world.DisplayResults  TextArrayA , TextArrayB , NumArrayC
IDX = 0
While  IDX < size of NumArrayC
    print  item IDX from TextArrayA
    print  item IDX from TextArrayB
    print  item IDX from NumArrayC
    print  ----------------------------------------
    increment  IDX  by 1  more...
```

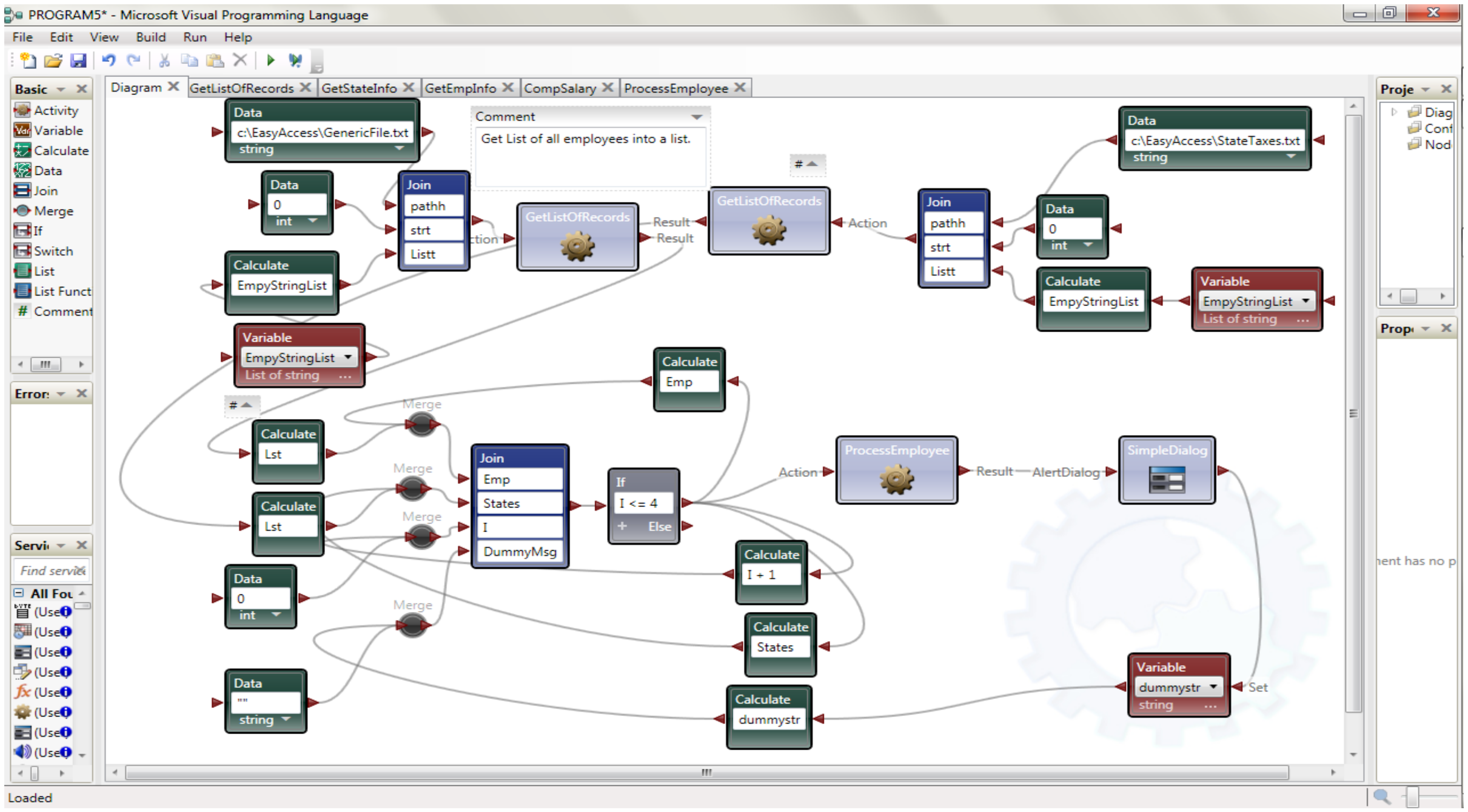


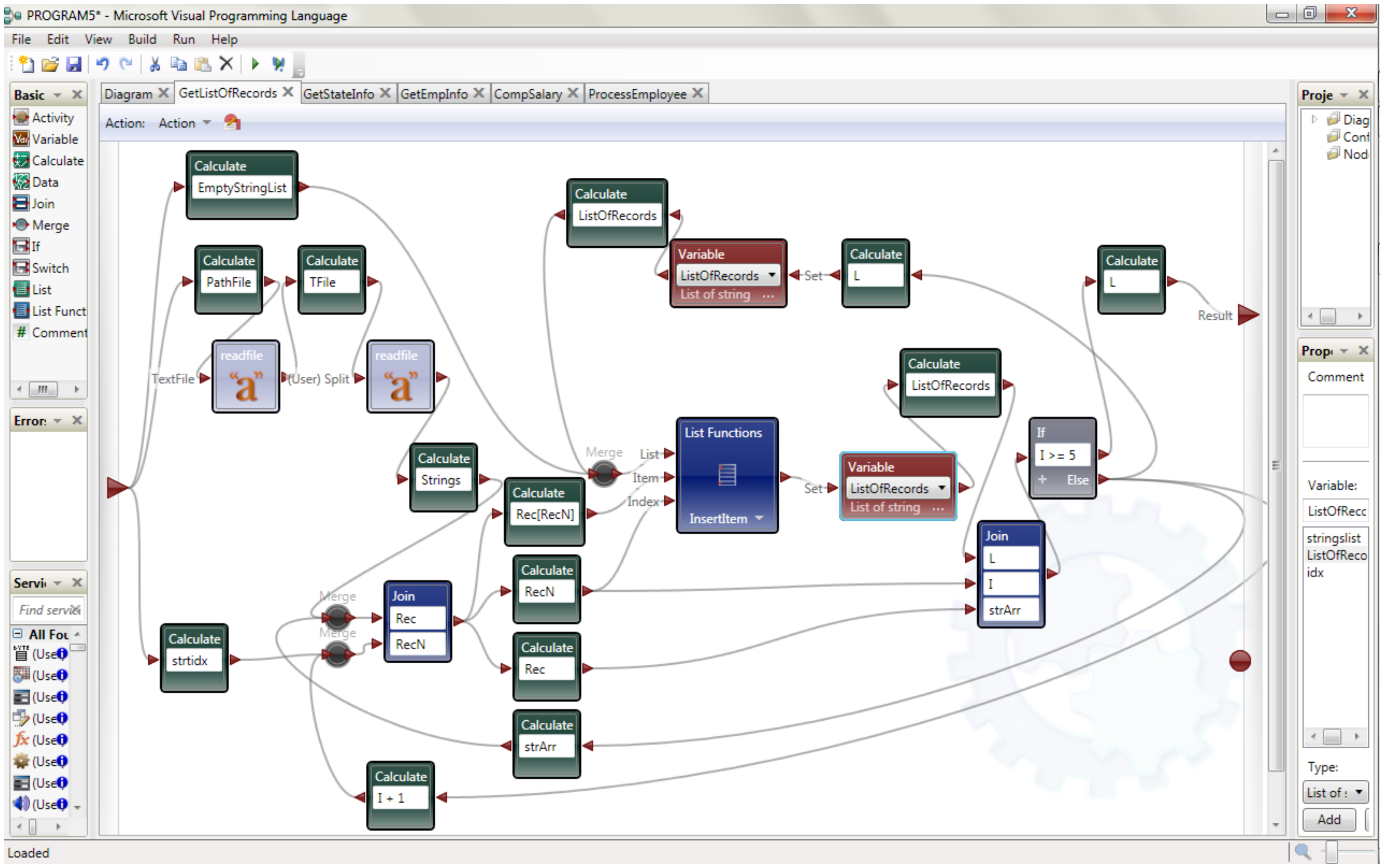
```
world.GetStateTax  StateInit , States , Taxes
IDX = 0 , StateTax = 1
While  IDX < size of Taxes
  If  StateInit == item IDX from States
    StateTax set value to item IDX from Taxes
    IDX set value to size of Taxes
  Else
    IDX set value to ( 1 + IDX )
Return StateTax
```

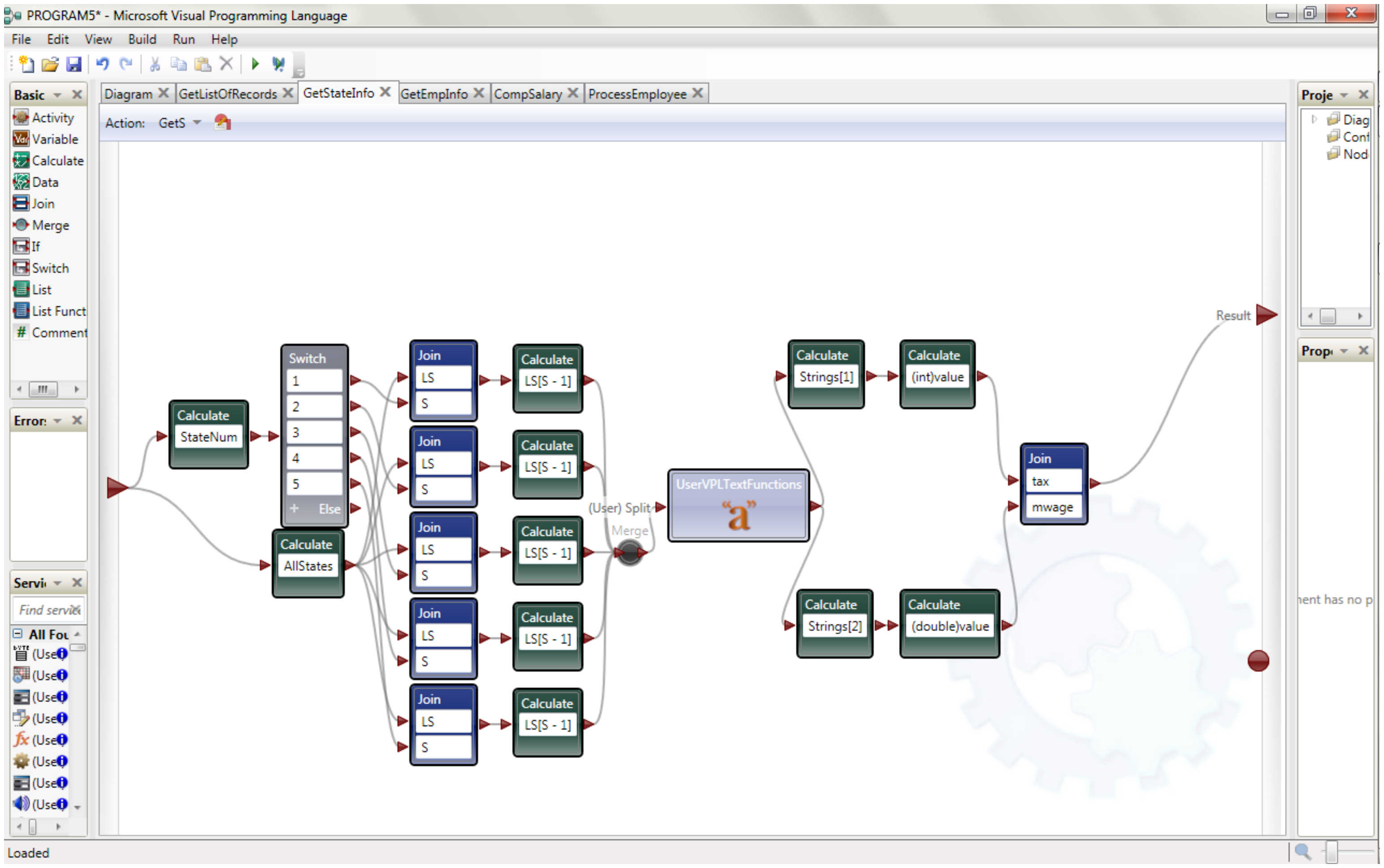


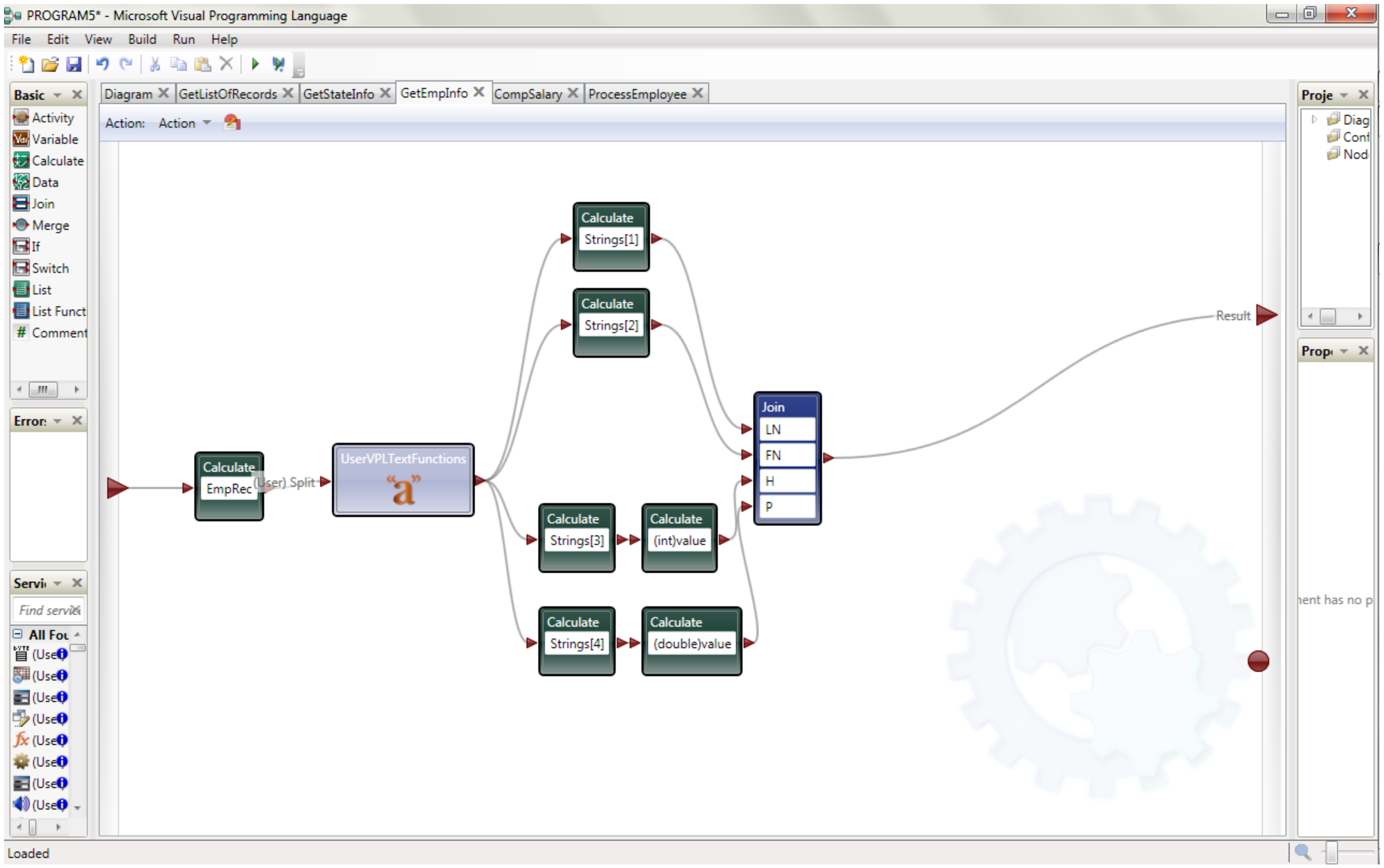
```
world.GetStateMinWage  StateInit , States , MinWages
IDX = 0 , Wage = 1
While  IDX < size of MinWages
    If  StateInit == item IDX from States
        Wage set value to item IDX from MinWages
        IDX set value to size of MinWages
    Else
        IDX set value to ( 1 + IDX )
Return Wage
```

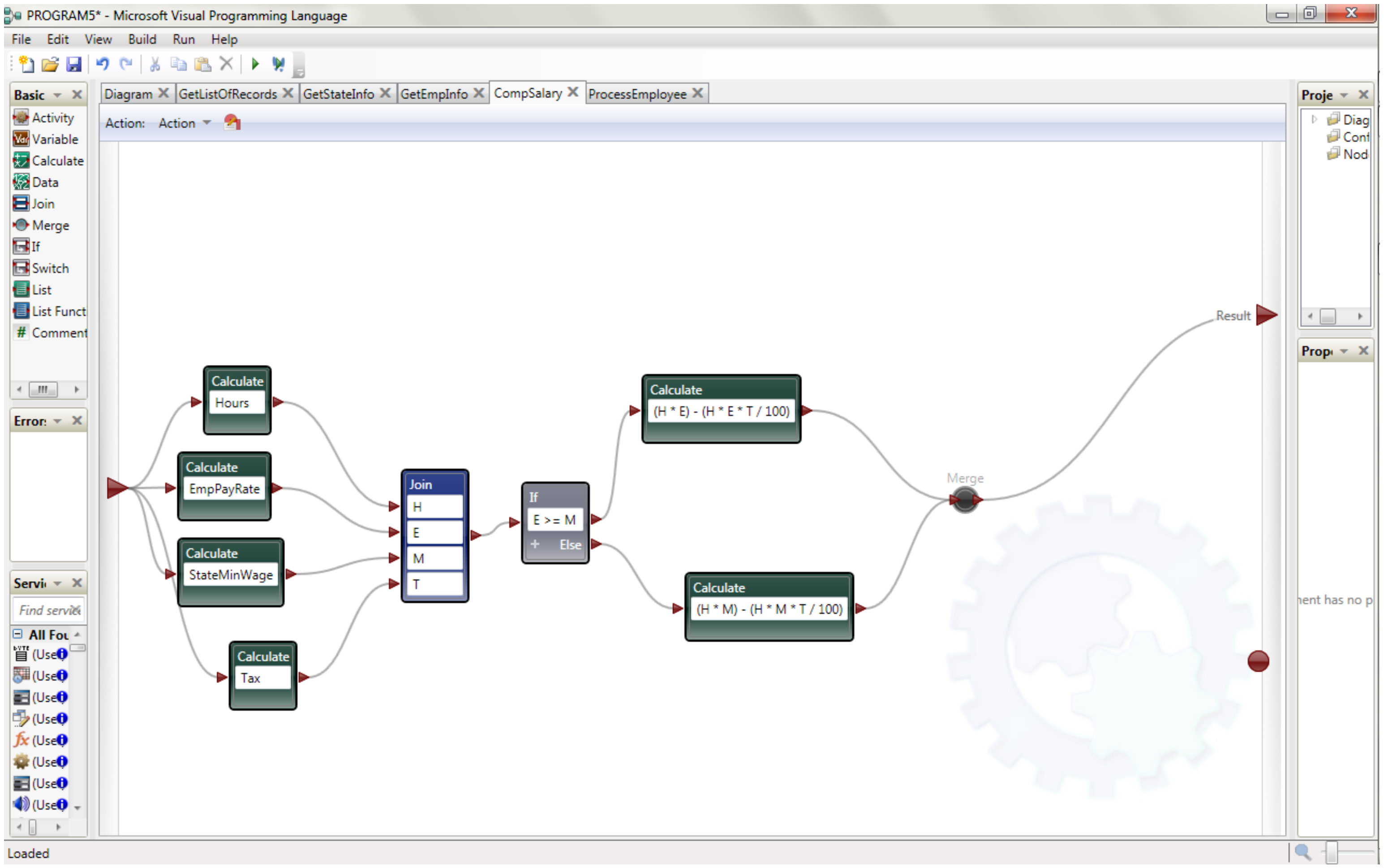
Appendix B: Program test in MS VPL

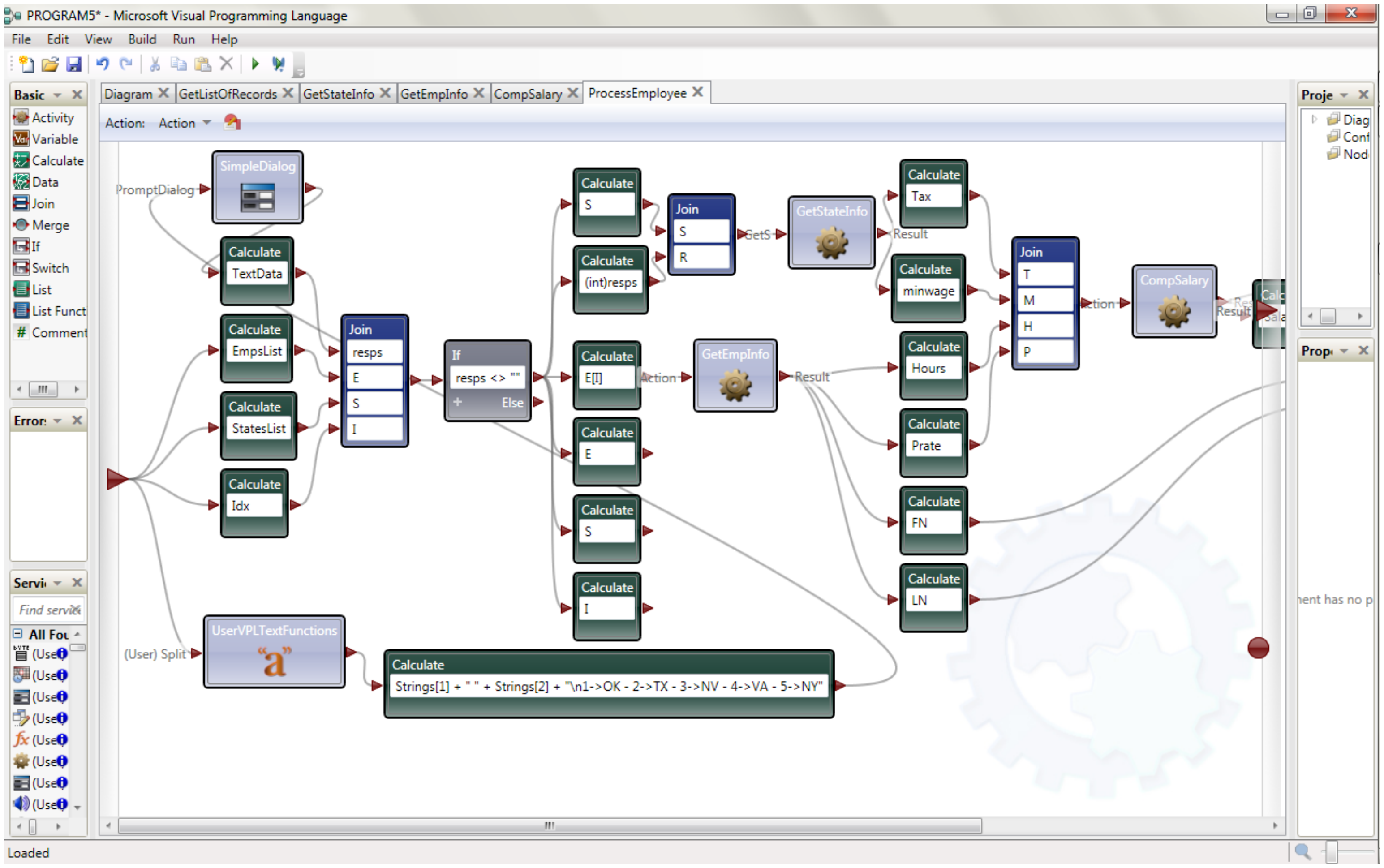


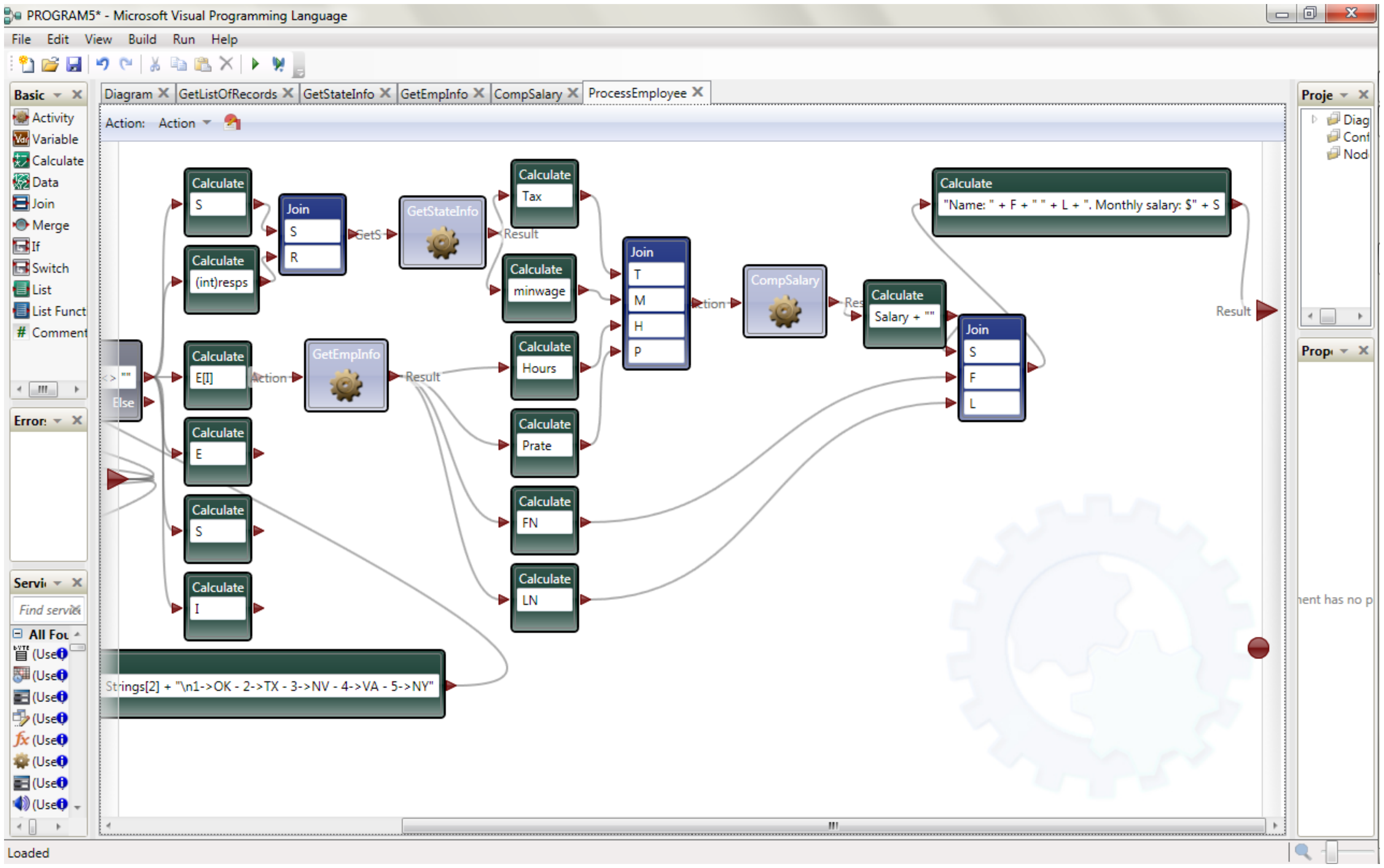

PROGRAM5* - Microsoft Visual Programming Language
File Edit View Build Run Help
Basic
Activity
Variable
Calculate
Data
Join
Merge
If
Switch
List
List Funct
Comment
Diagram
GetListOfRecords
GetStateInfo
GetEmpInfo
CompSalary
ProcessEmployee
Action: Action
Calculate EmptyStringList
Calculate PathFile
Calculate TFile
readfile
TextFile
(User) Split
Calculate Strings
Calculate ListOfRecords
Variable ListOfRecords List of string
Set
Calculate L
Calculate L
Result
Calculate ListOfRecords
Merge
List
Item
Index
List Functions InsertItem
Variable ListOfRecords List of string
If I >= 5 Else
Calculate Rec[RecN]
Join L I strArr
Calculate strtidx
Join Rec RecN
Calculate RecN
Calculate Rec
Calculate strArr
Calculate I + 1
Proje
Diag
Conf
Nod
Prop
Comment
Variable:
ListOfRecc
stringslist
ListOfReco
idx
Type:
List of
Add
Error
Servi
Find servic
All Fou
Loaded


PROGRAM5* - Microsoft Visual Programming Language
File Edit View Build Run Help
Basic
Activity
Variable
Calculate
Data
Join
Merge
If
Switch
List
List Funct
Comment
Errors
Servi
Find service
All Fou
Diagram
GetListOfRecords
GetStateInfo
GetEmpInfo
CompSalary
ProcessEmployee
Action: GetS
Proje
Diag
Conf
Nod
Prop
Calculate
StateNum
Switch
1
2
3
4
5
Else
Calculate
AllStates
Join
LS
S
Calculate
LS[S - 1]
(User) Split
Merge
UserVPLTextFunctions
Calculate
Strings[1]
Calculate
(int)value
Calculate
Strings[2]
Calculate
(double)value
Join
tax
mwage
Result
Loaded

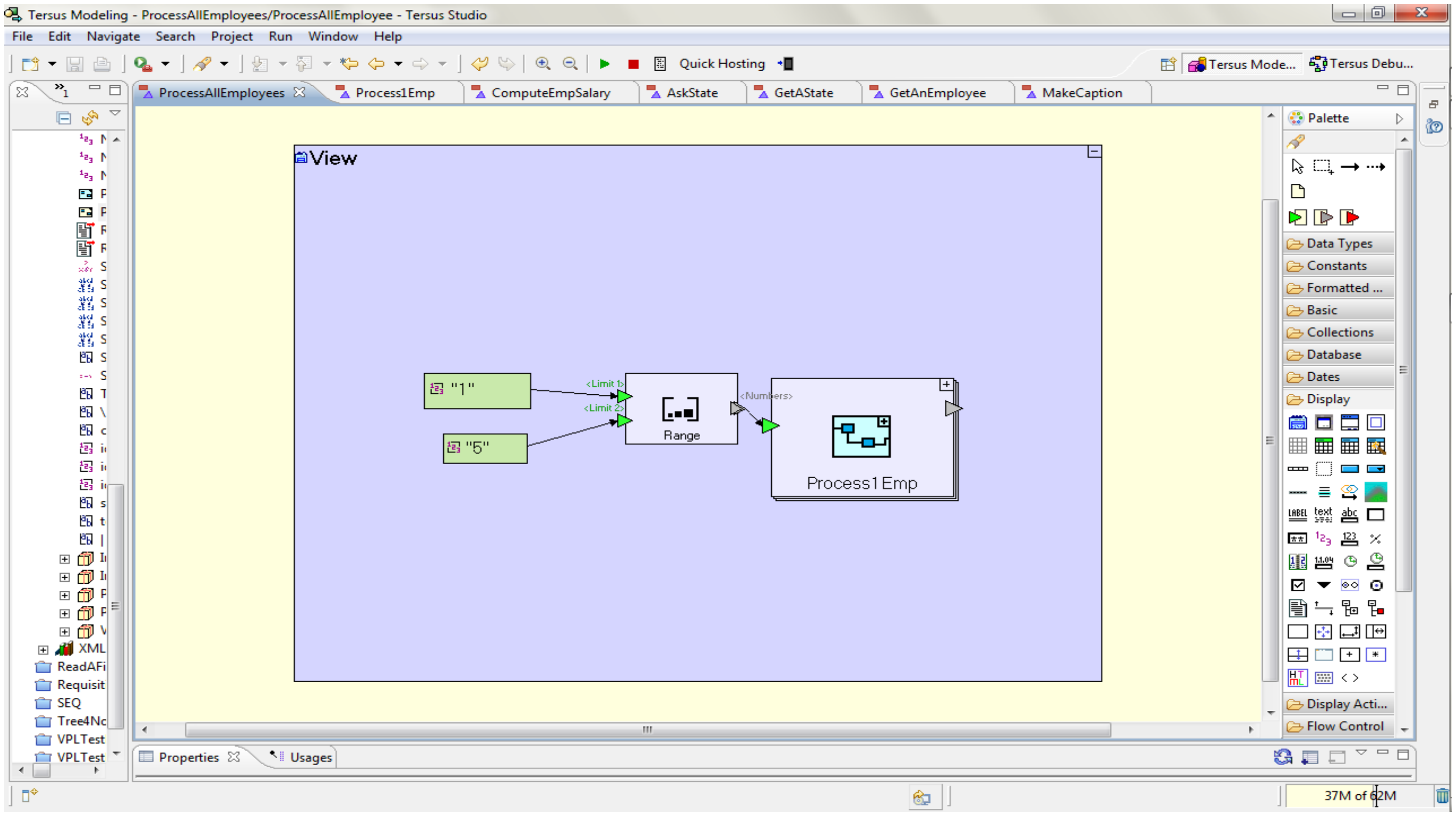

PROGRAM5* - Microsoft Visual Programming Language
File Edit View Build Run Help
Basic
Activity
Variable
Calculate
Data
Join
Merge
If
Switch
List
List Funct
Comment
Error
Servi
Find servic
All Fou
Diagram
GetListOfRecords
GetStateInfo
GetEmpInfo
CompSalary
ProcessEmployee
Action: Action
Calculate
EmpRec
(User) Split
UserVPLTextFunctions
Calculate
Strings[1]
Calculate
Strings[2]
Calculate
Strings[3]
Calculate
(int)value
Calculate
Strings[4]
Calculate
(double)value
Join
LN
FN
H
P
Result
Proje
Diag
Conf
Nod
Prop
Loaded

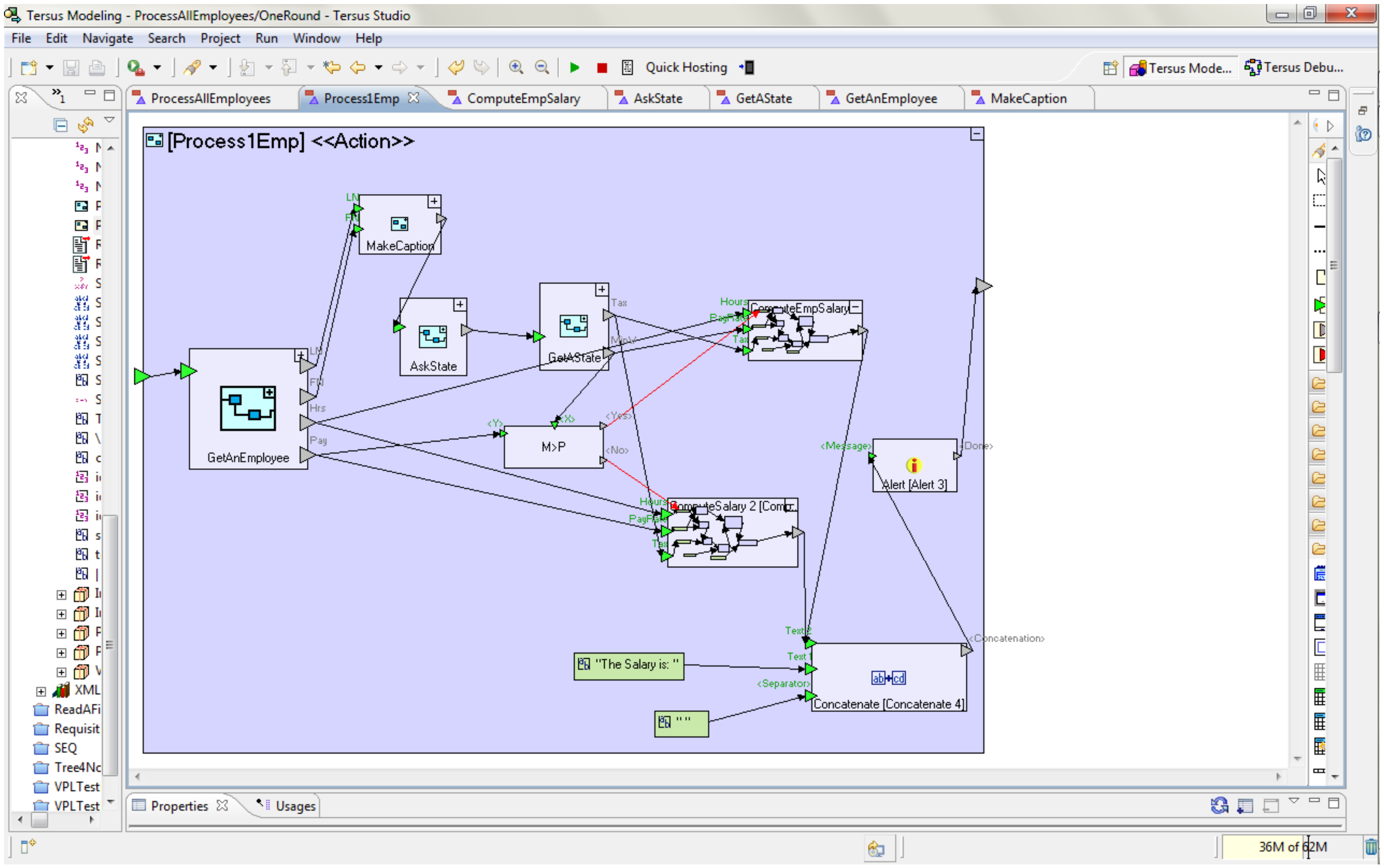

PROGRAM5* - Microsoft Visual Programming Language
File Edit View Build Run Help
Basic
Activity
Variable
Calculate
Data
Join
Merge
If
Switch
List
List Funct
Comment
Error:
Servi
Find servic
All Fou
Diagram
GetListOfRecords
GetStateInfo
GetEmpInfo
CompSalary
ProcessEmployee
Action: Action
Calculate
Hours
Calculate
EmpPayRate
Calculate
StateMinWage
Calculate
Tax
Join
H
E
M
T
If
E >= M
Else
Calculate
(H * E) - (H * E * T / 100)
Calculate
(H * M) - (H * M * T / 100)
Merge
Result
Proje
Diag
Conf
Nod
Prop
Loaded

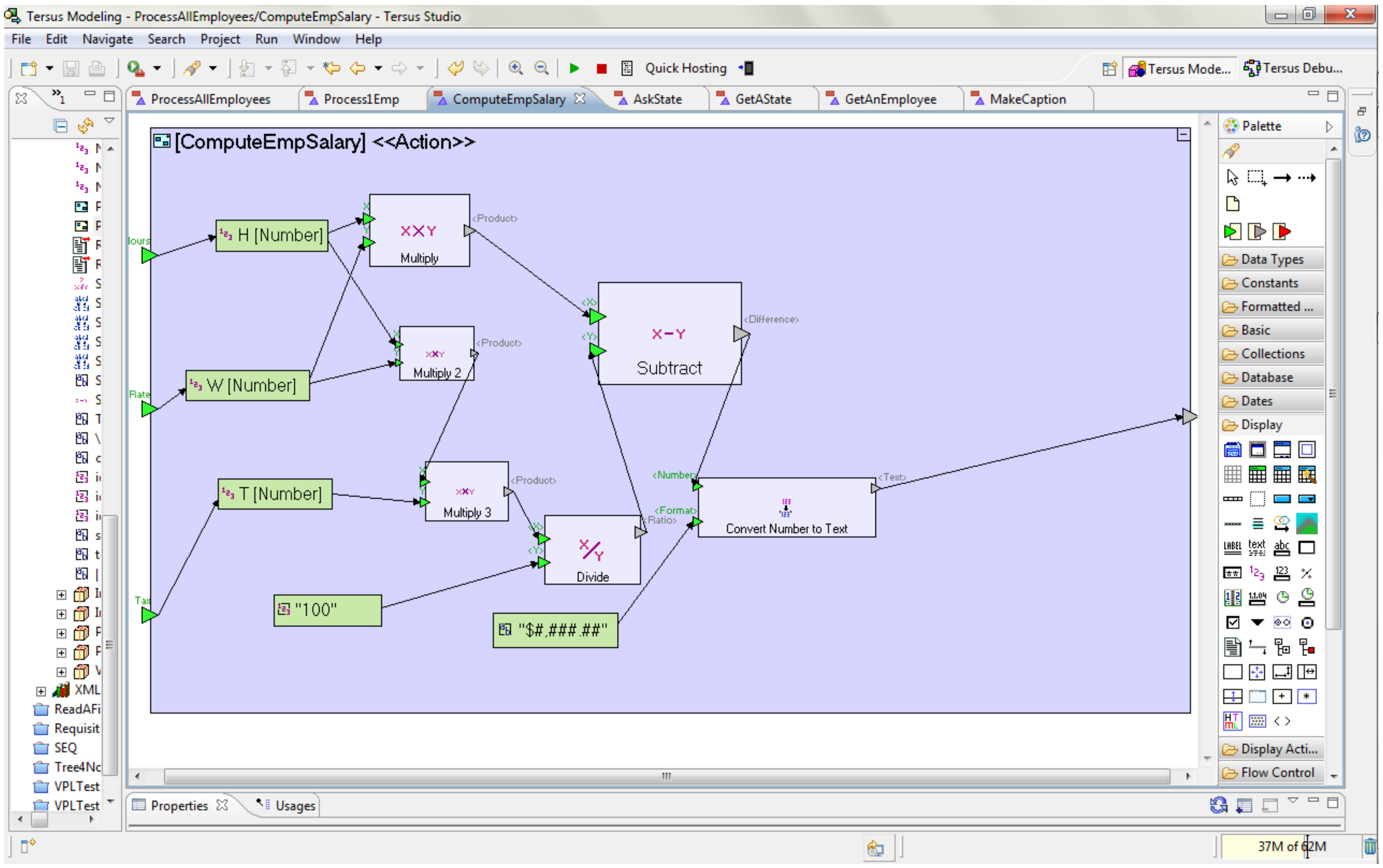

PROGRAM5* - Microsoft Visual Programming Language
File Edit View Build Run Help
Basic
Activity
Variable
Calculate
Data
Join
Merge
If
Switch
List
List Funct
Comment
Diagram
GetListOfRecords
GetStateInfo
GetEmpInfo
CompSalary
ProcessEmployee
Action: Action
SimpleDialog
PromptDialog
Calculate
TextData
EmpsList
StatesList
Idx
Join
resps
E
S
I
If
resps <> ""
Else
(int)resps
E[I]
GetStateInfo
GetS
Result
GetEmpInfo
Action
Tax
minwage
Hours
Prate
FN
LN
T
M
H
P
CompSalary
UserVPLTextFunctions
(User) Split
Strings[1] + " " + Strings[2] + "\n1->OK - 2->TX - 3->NV - 4->VA - 5->NY"
Loaded

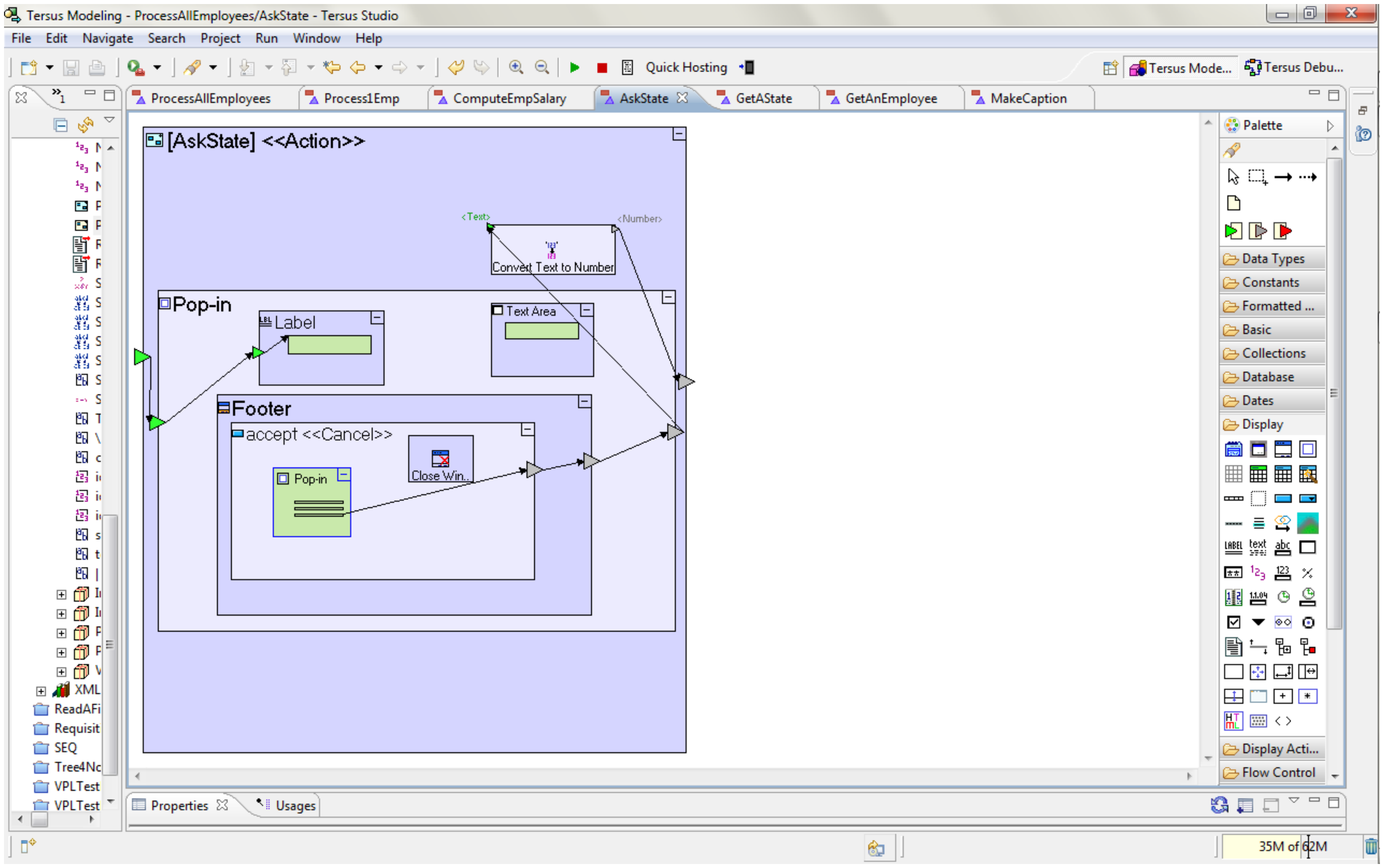

PROGRAM5* - Microsoft Visual Programming Language
File Edit View Build Run Help
Basic
Activity
Variable
Calculate
Data
Join
Merge
If
Switch
List
List Funct
Comment
Diagram
GetListOfRecords
GetStateInfo
GetEmpInfo
CompSalary
ProcessEmployee
Action: Action
Calculate S
Calculate (int)resps
Join S R
GetS
GetStateInfo
Result
Calculate Tax
Calculate minwage
Join T M H P
Action
CompSalary
Calculate Salary + ""
Calculate "Name: " + F + " " + L + ". Monthly salary: $" + S
Join S F L
Result
Else
Calculate E[I]
Action
GetEmpInfo
Result
Calculate Hours
Calculate Prate
Calculate FN
Calculate LN
Calculate E
Calculate S
Calculate I
Strings[2] + "\n1->OK - 2->TX - 3->NV - 4->VA - 5->NY"
Proje
Diag
Conf
Nod
Prop
Error
Servi
Find service
All Fou
Loaded

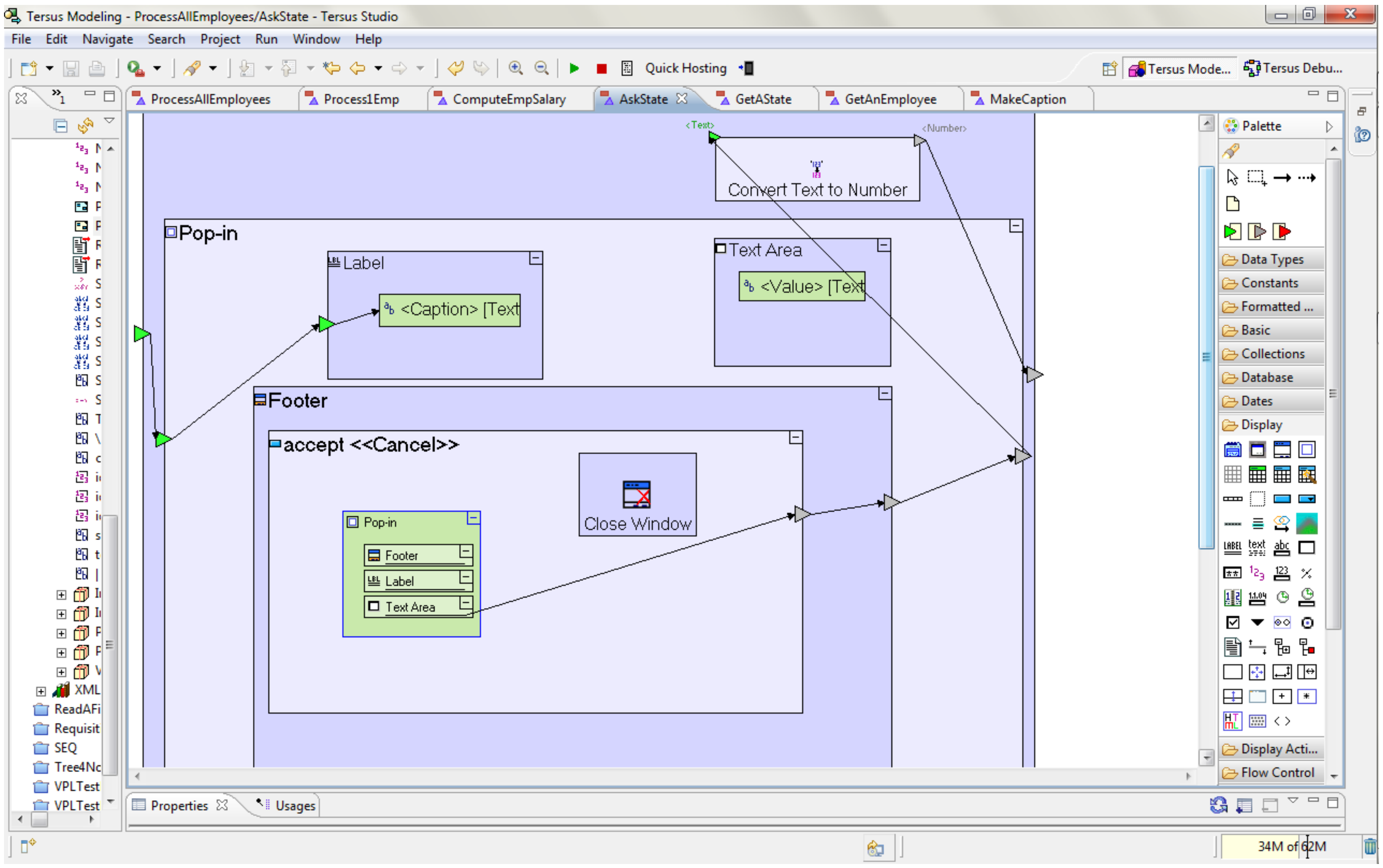
Appendix C: Program test in Tersus

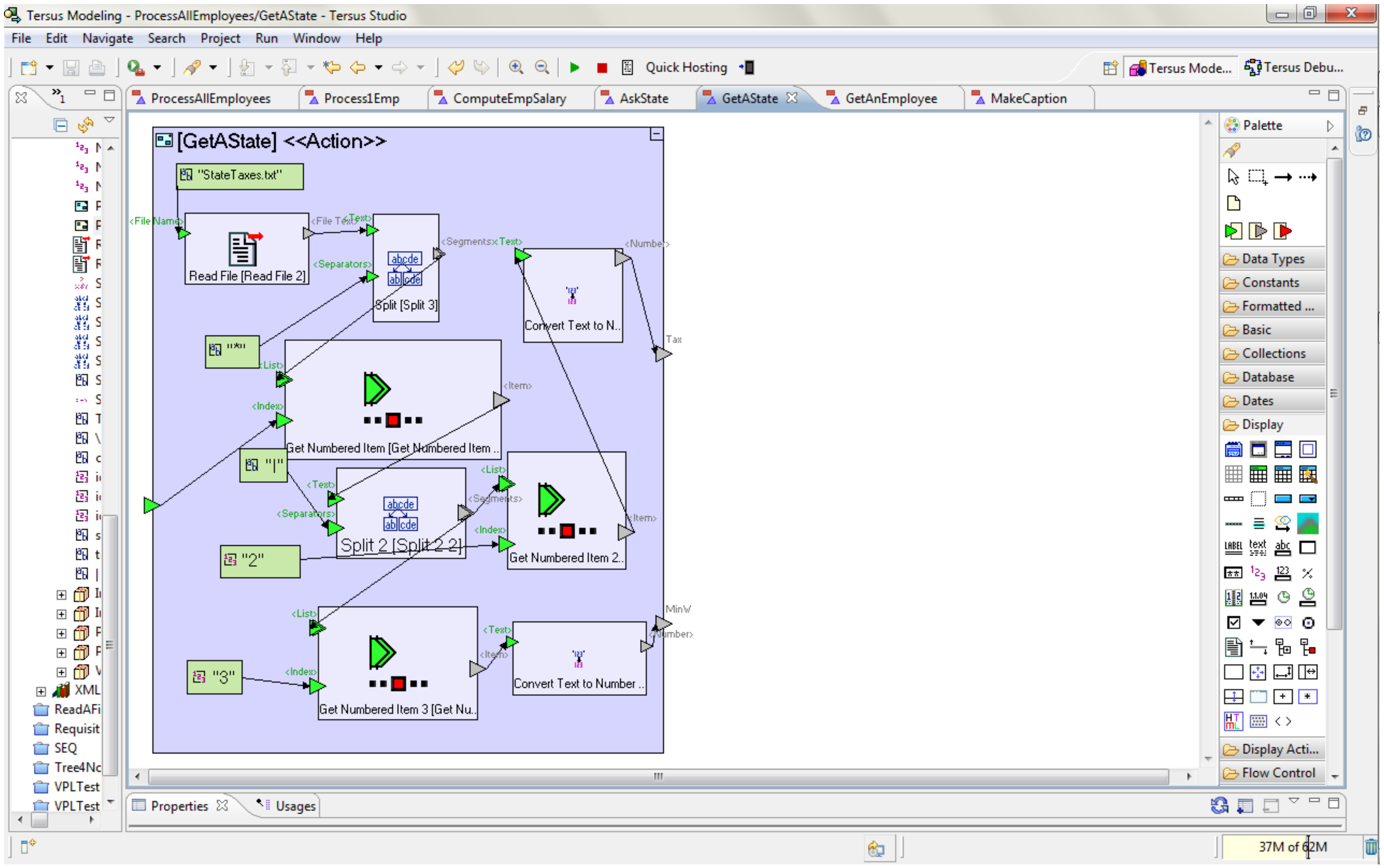


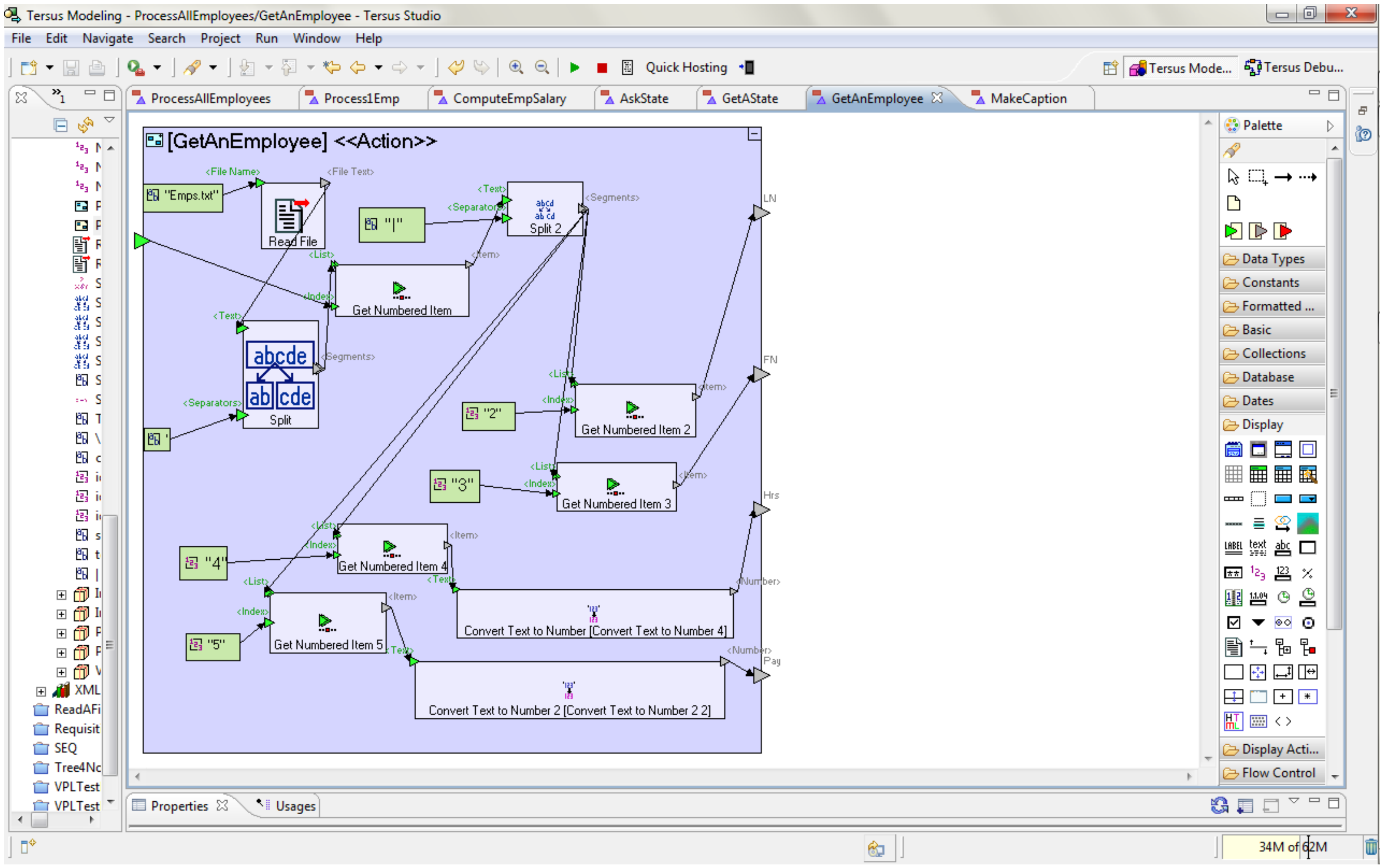

Tersus Modeling - ProcessAllEmployees/OneRound - Tersus Studio
File Edit Navigate Search Project Run Window Help
Quick Hosting
Tersus Mode...
Tersus Debu...
ProcessAllEmployees
Process1Emp
ComputeEmpSalary
AskState
GetAState
GetAnEmployee
MakeCaption
[Process1Emp] <<Action>>
MakeCaption
AskState
GetAState
ComputeEmpSalary
GetAnEmployee
M>P
<Yes>
<No>
ComputeSalary 2 [Comp...
Alert [Alert 3]
<Message>
<Done>
"The Salary is: "
" "
Concatenate [Concatenate 4]
<Separator>
<Concatenation>
Properties
Usages
36M of 62M

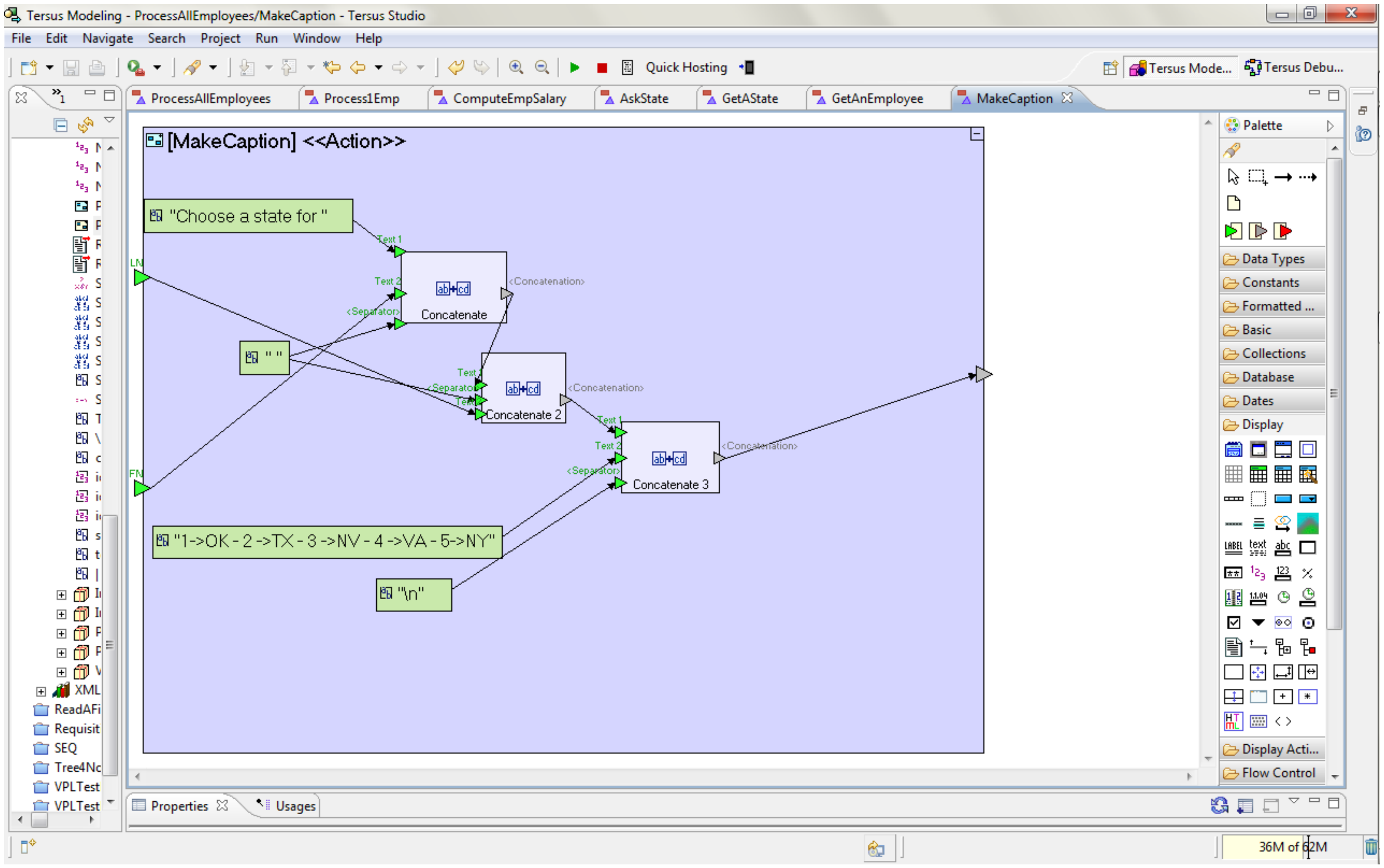

Tersus Modeling - ProcessAllEmployees/ComputeEmpSalary - Tersus Studio
File Edit Navigate Search Project Run Window Help
Quick Hosting
Tersus Mode...
Tersus Debu...
ProcessAllEmployees
Process1Emp
ComputeEmpSalary
AskState
GetAState
GetAnEmployee
MakeCaption
[ComputeEmpSalary] <<Action>>
Hours
Rate
Tax
H [Number]
W [Number]
T [Number]
"100"
"$#,###.##"
X×Y
Multiply
Multiply 2
Multiply 3
<Product>
X−Y
Subtract
<Difference>
X/Y
Divide
<Ratio>
<Number>
<Format>
Convert Number to Text
<Text>
Palette
Data Types
Constants
Formatted ...
Basic
Collections
Database
Dates
Display
Display Acti...
Flow Control
XML
ReadAFi
Requisit
SEQ
Tree4Nc
VPLTest
VPLTest
Properties
Usages
37M of 62M


Tersus Modeling - ProcessAllEmployees/AskState - Tersus Studio
File Edit Navigate Search Project Run Window Help
Quick Hosting
Tersus Mode...
Tersus Debu...
ProcessAllEmployees
Process1Emp
ComputeEmpSalary
AskState
GetAState
GetAnEmployee
MakeCaption
[AskState] <<Action>>
<Text>
<Number>
Convert Text to Number
Pop-in
Label
Text Area
Footer
accept <<Cancel>>
Pop-in
Close Win...
Palette
Data Types
Constants
Formatted ...
Basic
Collections
Database
Dates
Display
Display Acti...
Flow Control
XML
ReadAFi
Requisit
SEQ
Tree4Nc
VPLTest
VPLTest
Properties
Usages
35M of 62M

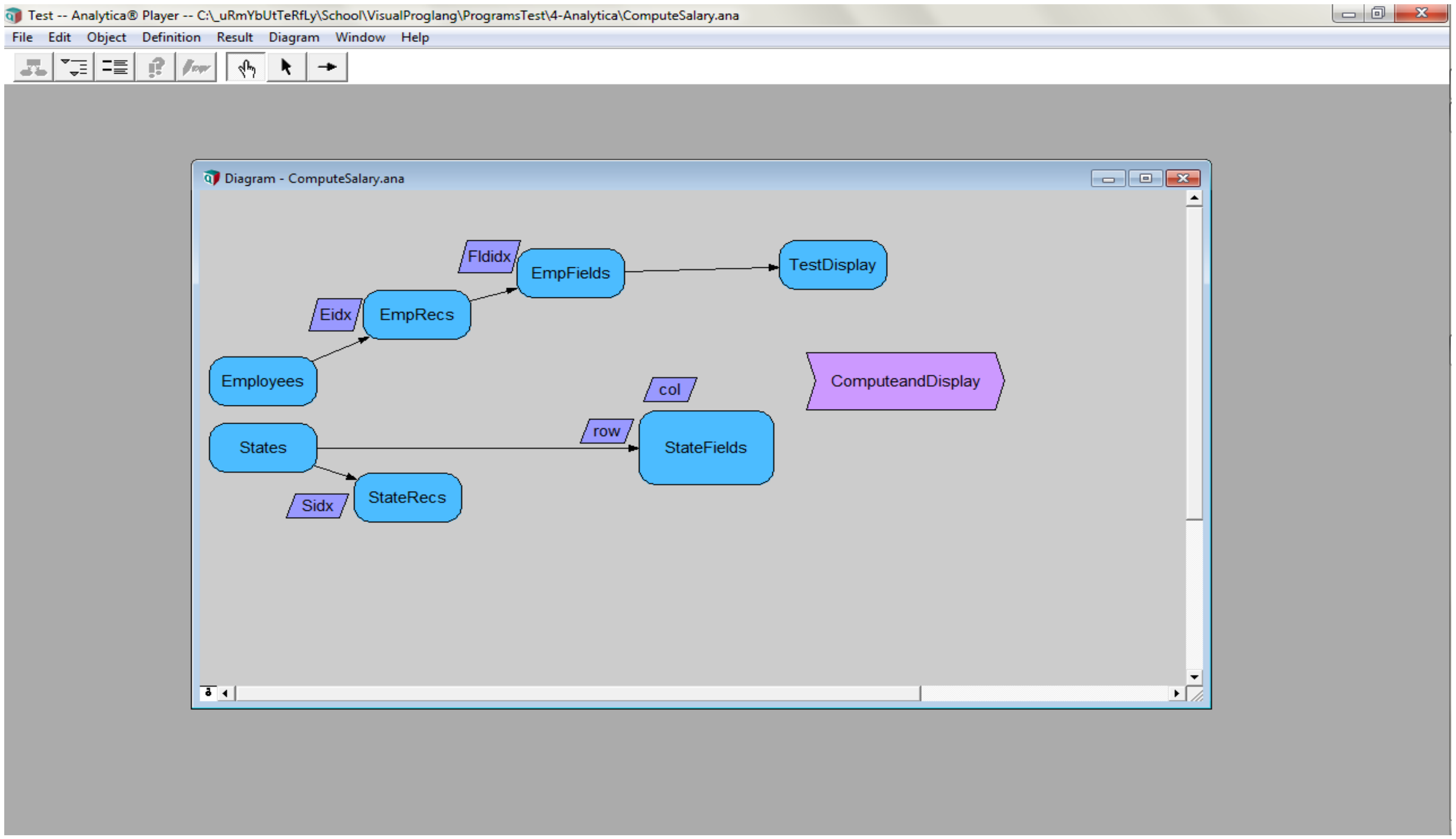

Tersus Modeling - ProcessAllEmployees/AskState - Tersus Studio
File Edit Navigate Search Project Run Window Help
Quick Hosting
Tersus Mode...
Tersus Debu...
ProcessAllEmployees
Process1Emp
ComputeEmpSalary
AskState
GetAState
GetAnEmployee
MakeCaption
<Text>
<Number>
Convert Text to Number
Pop-in
Label
<Caption> [Text
Text Area
<Value> [Text
Footer
accept <<Cancel>>
Close Window
Pop-in
Footer
Label
Text Area
Palette
Data Types
Constants
Formatted ...
Basic
Collections
Database
Dates
Display
Display Acti...
Flow Control
XML
ReadAFi
Requisit
SEQ
Tree4Nc
VPLTest
VPLTest
Properties
Usages
34M of 62M

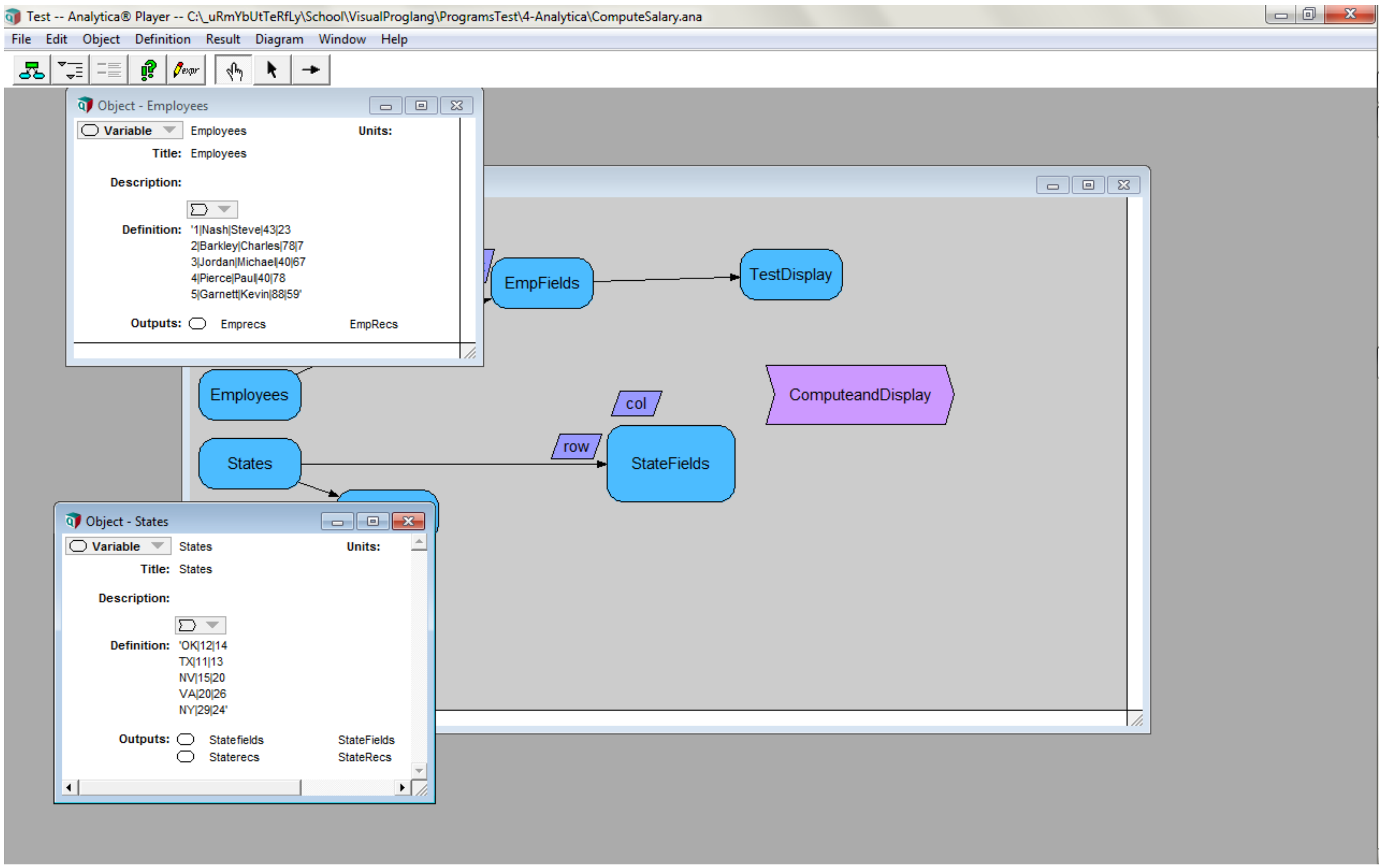

Tersus Modeling - ProcessAllEmployees/GetAState - Tersus Studio
File Edit Navigate Search Project Run Window Help
Quick Hosting
Tersus Mode... Tersus Debu...
ProcessAllEmployees Process1Emp ComputeEmpSalary AskState GetAState GetAnEmployee MakeCaption
[GetAState] <<Action>>
"StateTaxes.txt"
Read File [Read File 2]
Split [Split 3]
Convert Text to N...
"*"
Get Numbered Item [Get Numbered Item ...
"|"
Split 2 [Split 2 2]
Get Numbered Item 2...
"2"
"3"
Get Numbered Item 3 [Get Nu...
Convert Text to Number ...
Tax
MinV
Palette
Data Types
Constants
Formatted ...
Basic
Collections
Database
Dates
Display
Display Acti...
Flow Control
Properties
Usages
37M of 62M


Tersus Modeling - ProcessAllEmployees/GetAnEmployee - Tersus Studio
File Edit Navigate Search Project Run Window Help
Quick Hosting
Tersus Mode...
Tersus Debu...
ProcessAllEmployees
Process1Emp
ComputeEmpSalary
AskState
GetAState
GetAnEmployee
MakeCaption
[GetAnEmployee] <<Action>>
"Emps.txt"
Read File
"|"
Split 2
Get Numbered Item
Split
"2"
Get Numbered Item 2
"3"
Get Numbered Item 3
"4"
Get Numbered Item 4
"5"
Get Numbered Item 5
Convert Text to Number [Convert Text to Number 4]
Convert Text to Number 2 [Convert Text to Number 2 2]
LN
FN
Hrs
Pay
Palette
Data Types
Constants
Formatted ...
Basic
Collections
Database
Dates
Display
Display Acti...
Flow Control
XML
ReadAFi
Requisit
SEQ
Tree4Nc
VPLTest
VPLTest
Properties
Usages
34M of 62M


Tersus Modeling - ProcessAllEmployees/MakeCaption - Tersus Studio
File Edit Navigate Search Project Run Window Help
Quick Hosting
Tersus Mode...
Tersus Debu...
ProcessAllEmployees
Process1Emp
ComputeEmpSalary
AskState
GetAState
GetAnEmployee
MakeCaption
[MakeCaption] <<Action>>
"Choose a state for "
" "
"1->OK - 2->TX - 3->NV - 4->VA - 5->NY"
"\n"
LN
FN
Text 1
Text 2
<Separator>
Concatenate
<Concatenation>
Concatenate 2
Concatenate 3
Palette
Data Types
Constants
Formatted ...
Basic
Collections
Database
Dates
Display
Display Acti...
Flow Control
XML
ReadAFi
Requisit
SEQ
Tree4Nc
VPLTest
Properties
Usages
36M of 62M

Appendix D: Program test in Analytica




Test -- Analytica® Player -- C:_uRmYbUtTeRfLy\School\VisualProglang\ProgramsTest\4-Analytica\ComputeSalary.ana
File Edit Object Definition Result Diagram Window Help
EmpFields
TestDisplay
Employees
States
col
row
StateFields
ComputeandDisplay

Object - Employees

Variable Employees Units:

Title: Employees

Description:

Definition: '1|Nash|Steve|43|23
2|Barkley|Charles|78|7
3|Jordan|Michael|40|67
4|Pierce|Paul|40|78
5|Garnett|Kevin|88|59'

Outputs: Emprecs EmpRecs

Object - States

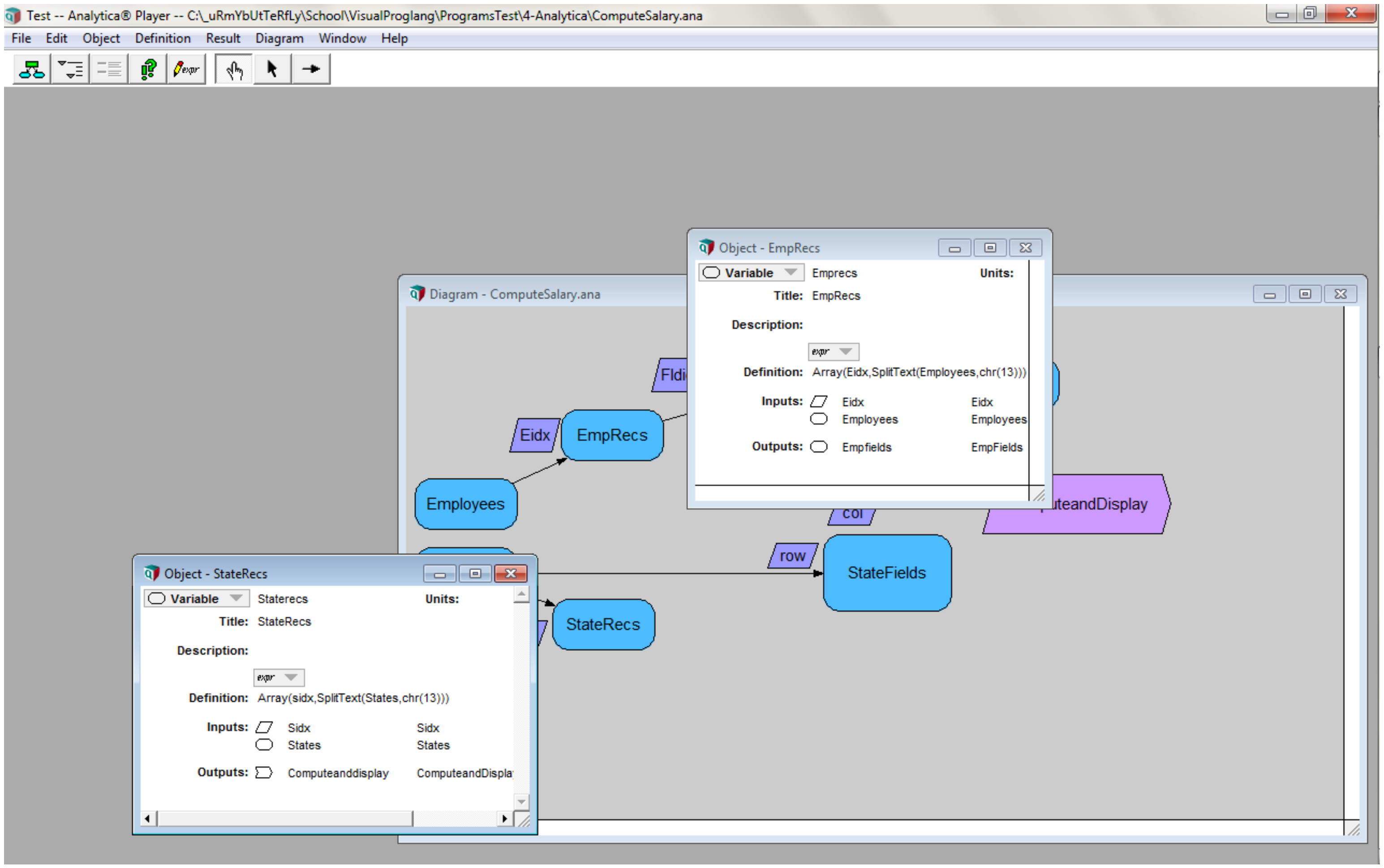
Variable States Units:

Title: States

Description:

Definition: 'OK|12|14
TX|11|13
NV|15|20
VA|20|26
NY|29|24'

Outputs: Statefields StateFields
Staterecs StateRecs

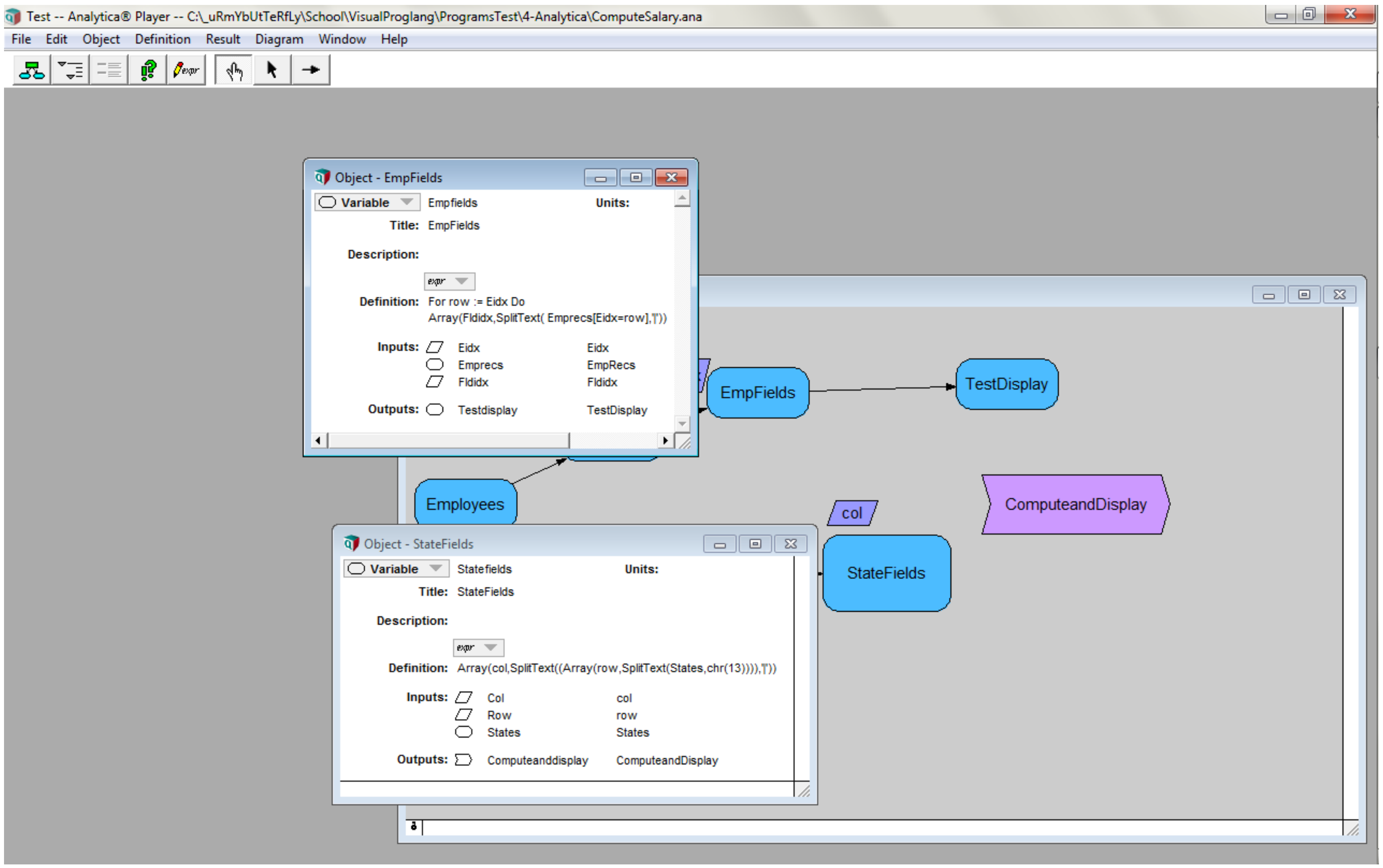

Test -- Analytica® Player -- C:_uRmYbUtTeRfLy\School\VisualProglang\ProgramsTest\4-Analytica\ComputeSalary.ana
File Edit Object Definition Result Diagram Window Help
Diagram - ComputeSalary.ana
Fldi
Eidx
EmpRecs
Employees
col
row
StateFields
StateRecs
teandDisplay
Object - EmpRecs
Object - StateRecs

Object - EmpRecs

Variable Emprecs Units:
Title: EmpRecs
Description:
Definition: Array(Eidx,SplitText(Employees,chr(13)))
Inputs: Eidx Eidx
Employees Employees
Outputs: Empfields EmpFields

Object - StateRecs

Variable Staterecs Units:
Title: StateRecs
Description:
Definition: Array(sidx,SplitText(States,chr(13)))
Inputs: Sidx Sidx
States States
Outputs: Computeanddisplay ComputeandDispla



Test -- Analytica® Player -- C:_uRmYbUtTeRfLy\School\VisualProglang\ProgramsTest\4-Analytica\ComputeSalary.ana
File Edit Object Definition Result Diagram Window Help
EmpFields
TestDisplay
Employees
col
ComputeandDisplay
StateFields

Object - EmpFields

Variable Empfields Units:

Title: EmpFields

Description:

Definition:
```
For row := Eidx Do
Array(Fldidx,SplitText( Emprecs[Eidx=row],'|'))
```

Inputs:	Eidx	Eidx
	Emprecs	EmpRecs
	Fldidx	Fldidx
Outputs:	Testdisplay	TestDisplay

Object - StateFields

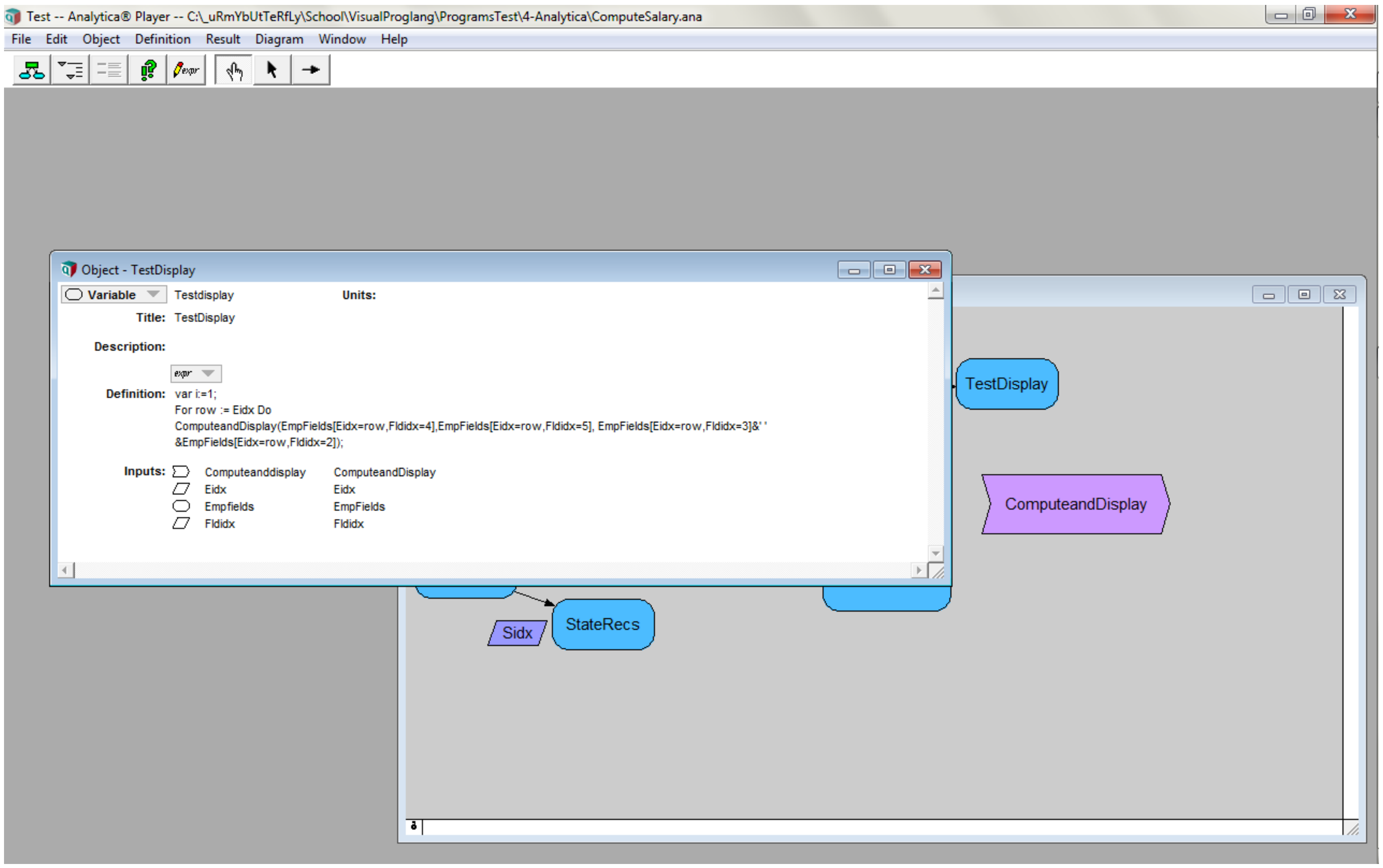
Variable Statefields Units:

Title: StateFields

Description:

Definition:
```
Array(col,SplitText((Array(row,SplitText(States,chr(13)))),'|'))
```

Inputs:	Col	col
	Row	row
	States	States
Outputs:	Computeanddisplay	ComputeandDisplay


Test -- Analytica® Player -- C:_uRmYbUtTeRfLy\School\VisualProglang\ProgramsTest\4-Analytica\ComputeSalary.ana
File Edit Object Definition Result Diagram Window Help
Object - TestDisplay
TestDisplay
ComputeandDisplay
Sidx
StateRecs

Variable Testdisplay **Units:**

Title: TestDisplay

Description:

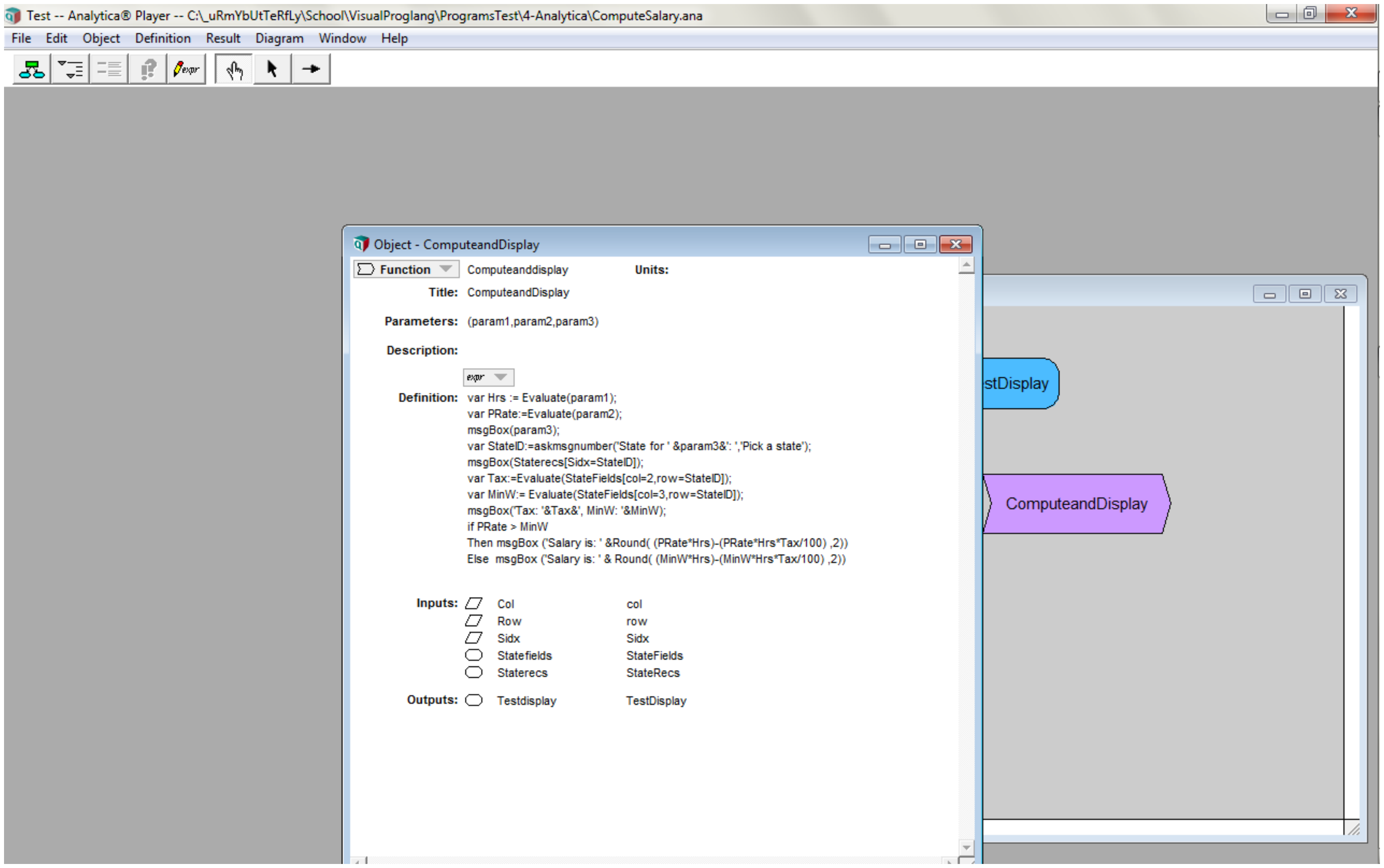
expr

Definition:

```
var i:=1;
For row := Eidx Do
ComputeandDisplay(EmpFields[Eidx=row,Fldidx=4],EmpFields[Eidx=row,Fldidx=5], EmpFields[Eidx=row,Fldidx=3]&' '
&EmpFields[Eidx=row,Fldidx=2]);
```

Inputs:

⊐	Computeanddisplay	ComputeandDisplay
▱	Eidx	Eidx
⬭	Empfields	EmpFields
▱	Fldidx	Fldidx


Test -- Analytica® Player -- C:_uRmYbUtTeRfLy\School\VisualProglang\ProgramsTest\4-Analytica\ComputeSalary.ana
File Edit Object Definition Result Diagram Window Help
Object - ComputeandDisplay
stDisplay
ComputeandDisplay

Function Computeanddisplay Units:

Title: ComputeandDisplay

Parameters: (param1,param2,param3)

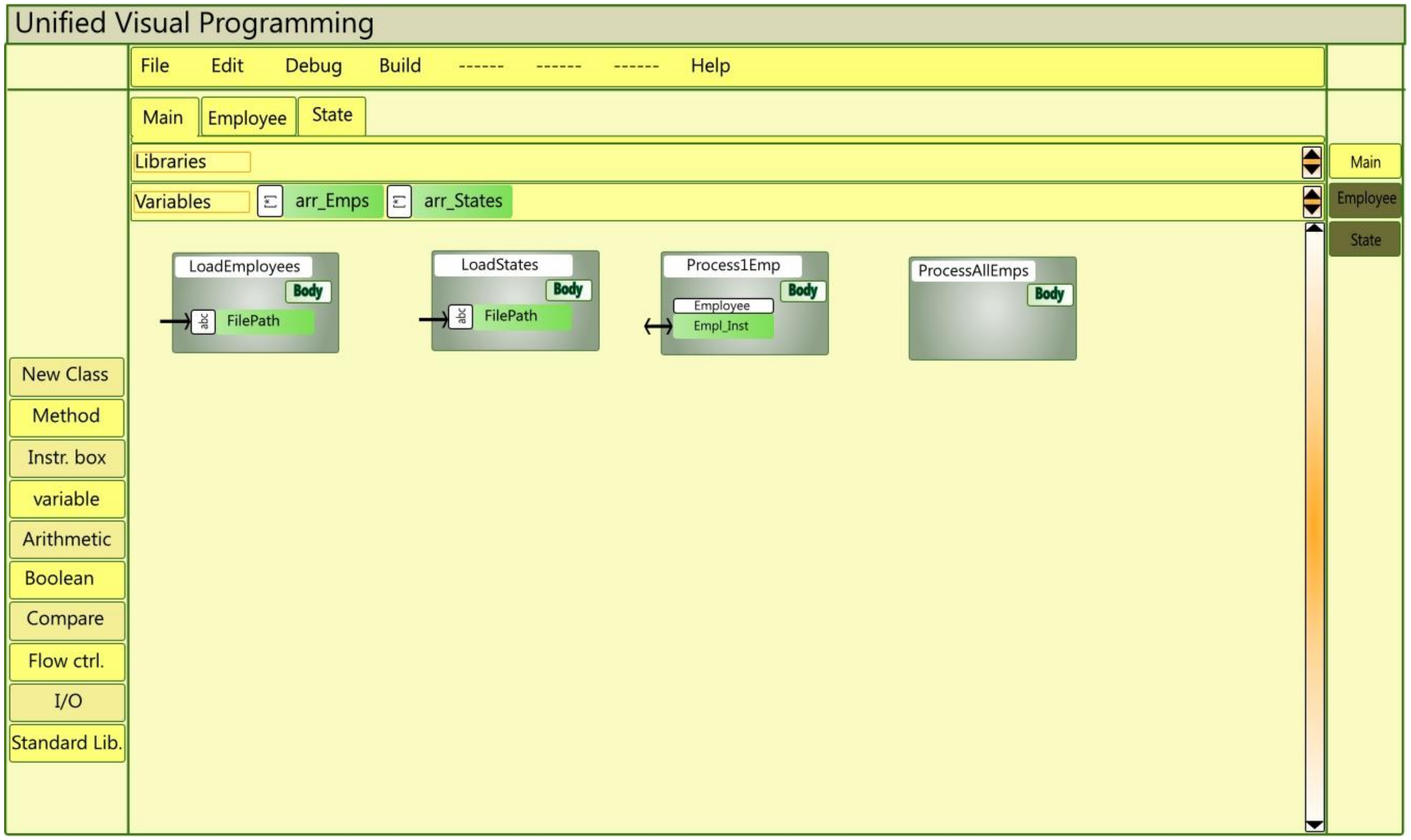
Description:

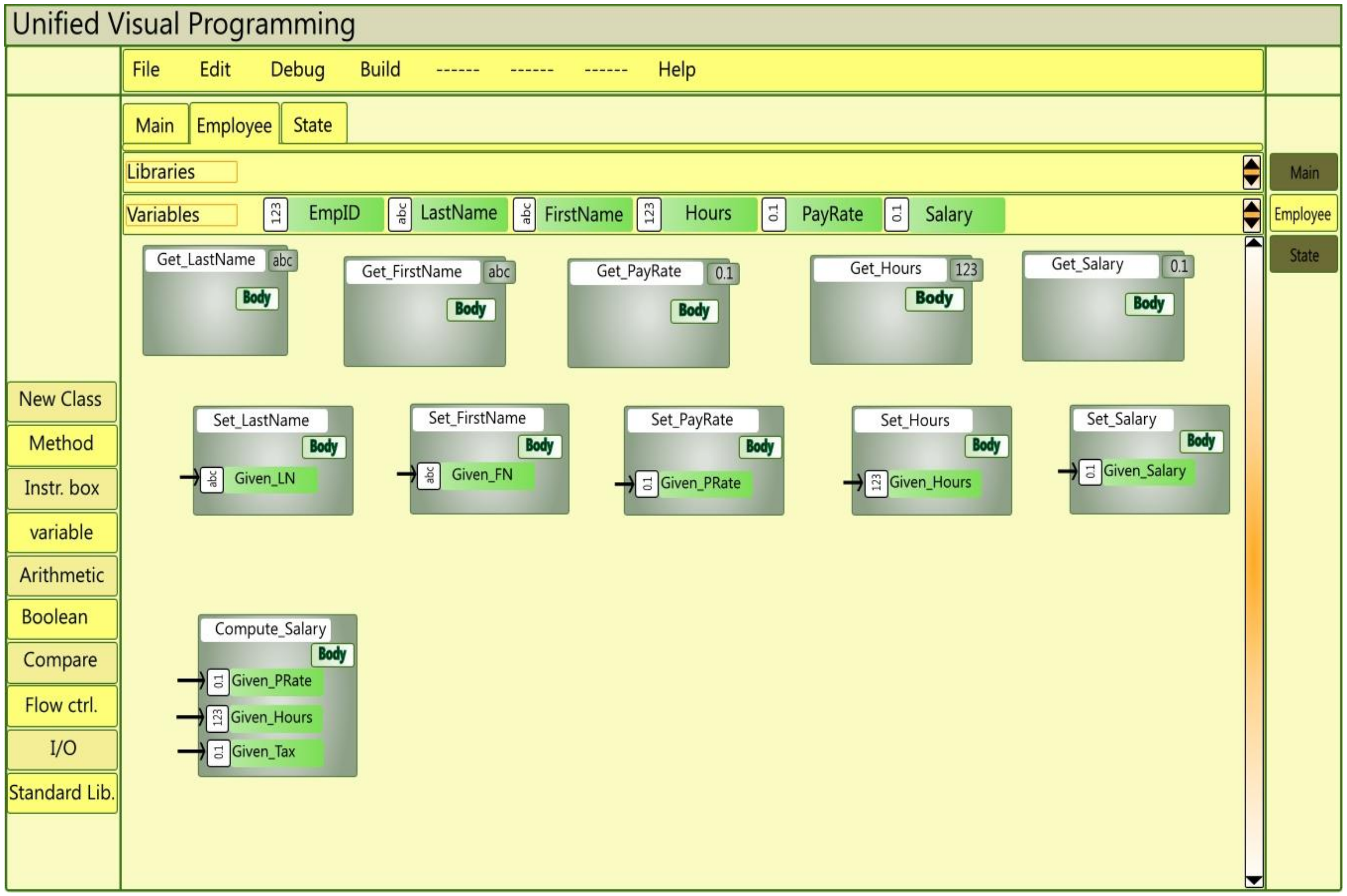
Definition:

```
var Hrs := Evaluate(param1);
var PRate:=Evaluate(param2);
msgBox(param3);
var StateID:=askmsgnumber('State for ' &param3&': ','Pick a state');
msgBox(Staterecs[Sidx=StateID]);
var Tax:=Evaluate(StateFields[col=2,row=StateID]);
var MinW:= Evaluate(StateFields[col=3,row=StateID]);
msgBox('Tax: '&Tax&', MinW: '&MinW);
if PRate > MinW
Then msgBox ('Salary is: ' &Round( (PRate*Hrs)-(PRate*Hrs*Tax/100) ,2))
Else  msgBox ('Salary is: ' & Round( (MinW*Hrs)-(MinW*Hrs*Tax/100) ,2))
```

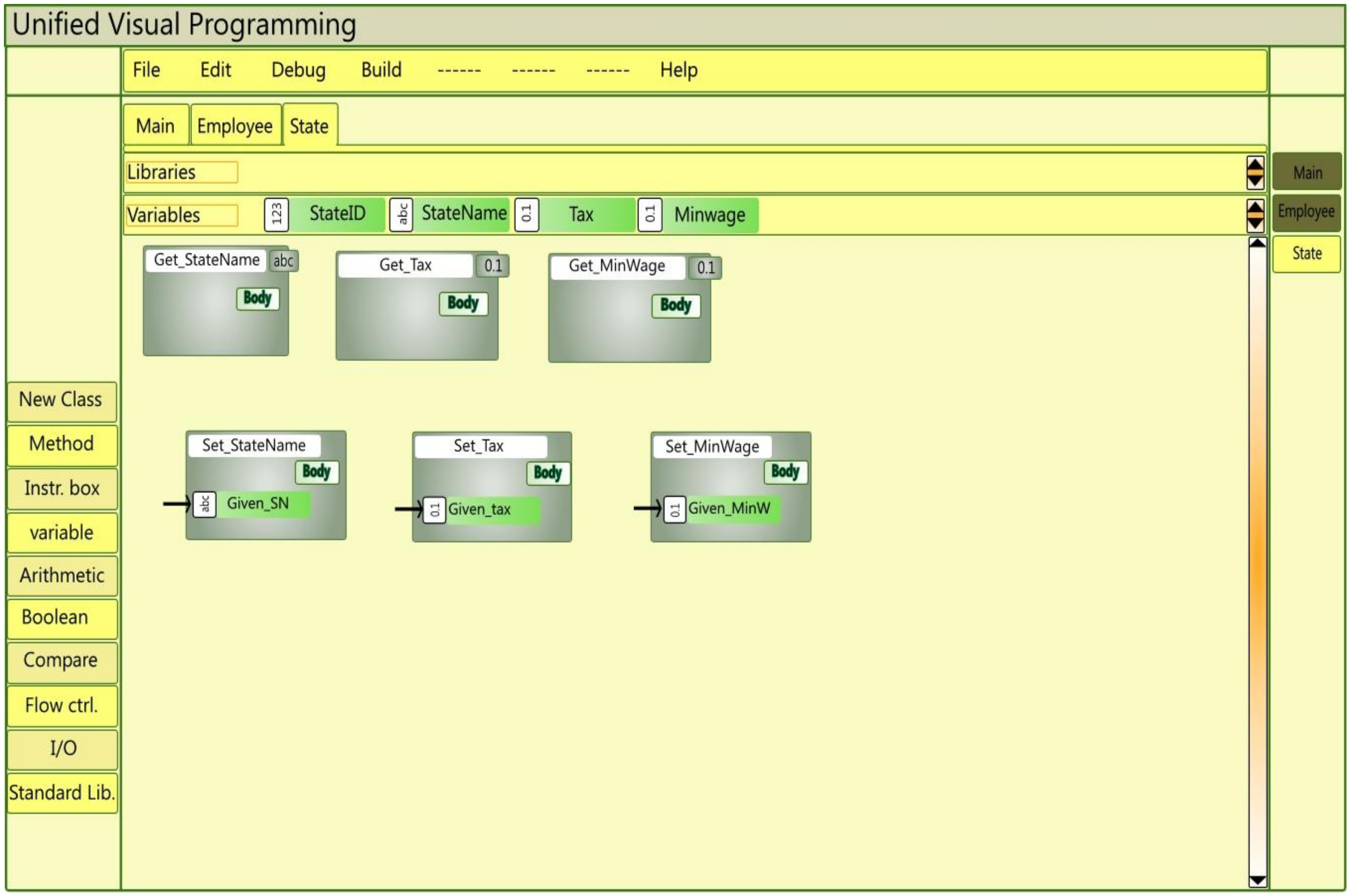
Inputs:	Col	col
	Row	row
	Sidx	Sidx
	Statefields	StateFields
	Staterecs	StateRecs
Outputs:	Testdisplay	TestDisplay

Appendix E: Program test in UVPL

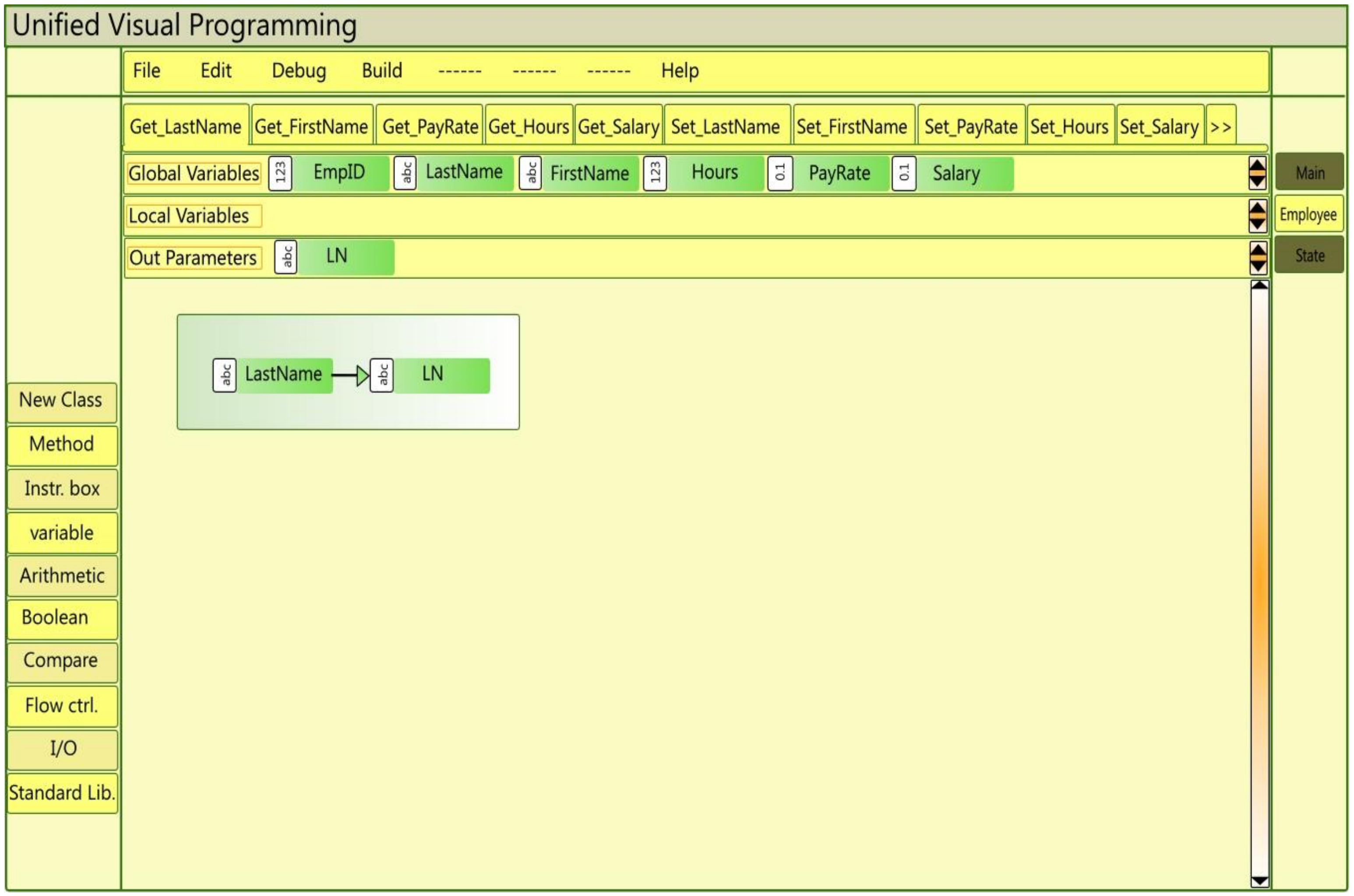



Unified Visual Programming
File
Edit
Debug
Build

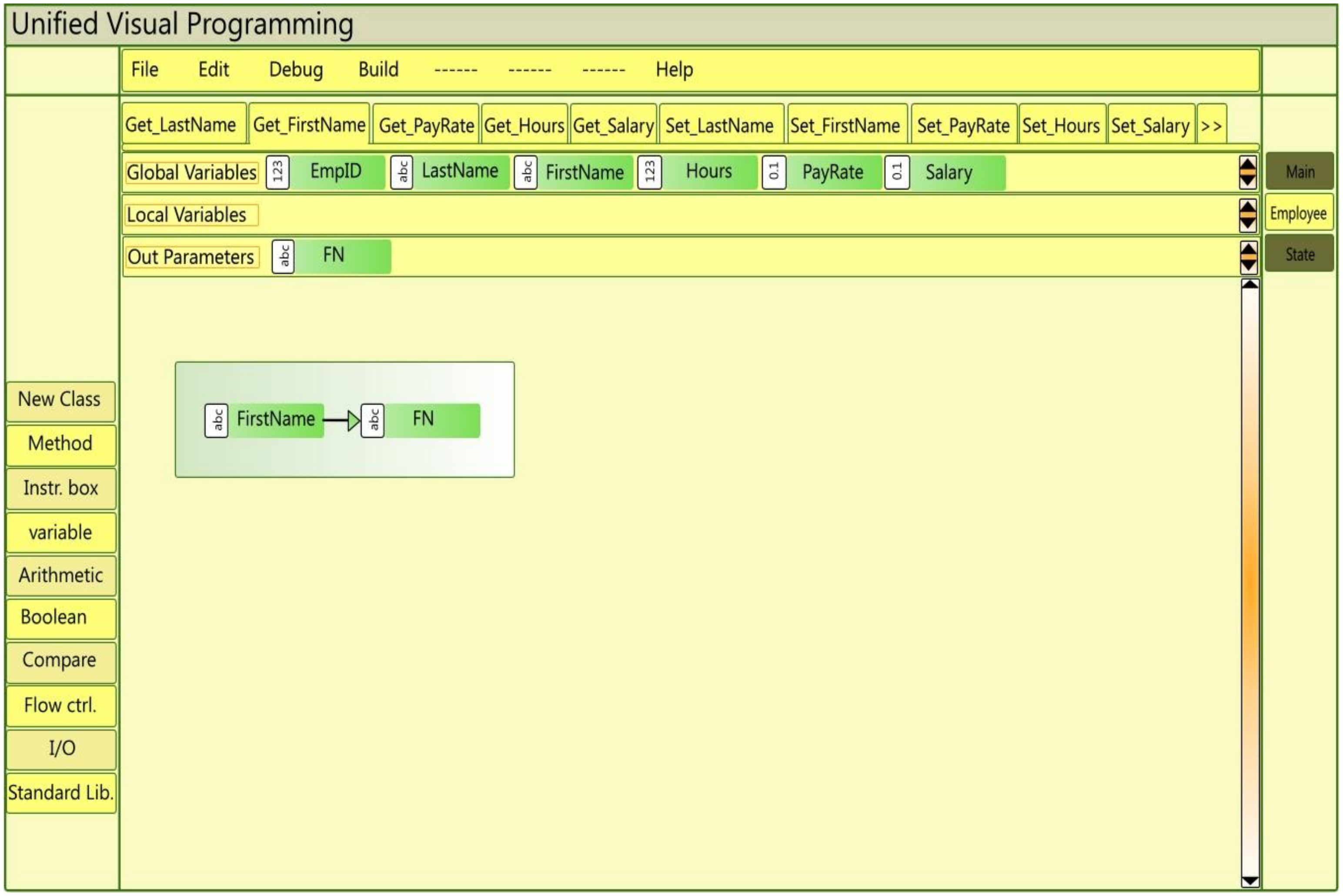
Help
Main
Employee
State
Libraries
Variables
123 EmpID
abc LastName
abc FirstName
123 Hours
0.1 PayRate
0.1 Salary
Get_LastName abc Body
Get_FirstName abc Body
Get_PayRate 0.1 Body
Get_Hours 123 Body
Get_Salary 0.1 Body
Set_LastName Body abc Given_LN
Set_FirstName Body abc Given_FN
Set_PayRate Body 0.1 Given_PRate
Set_Hours Body 123 Given_Hours
Set_Salary Body 0.1 Given_Salary
Compute_Salary Body 0.1 Given_PRate 123 Given_Hours 0.1 Given_Tax
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.
Main
Employee
State


Unified Visual Programming
File
Edit
Debug
Build

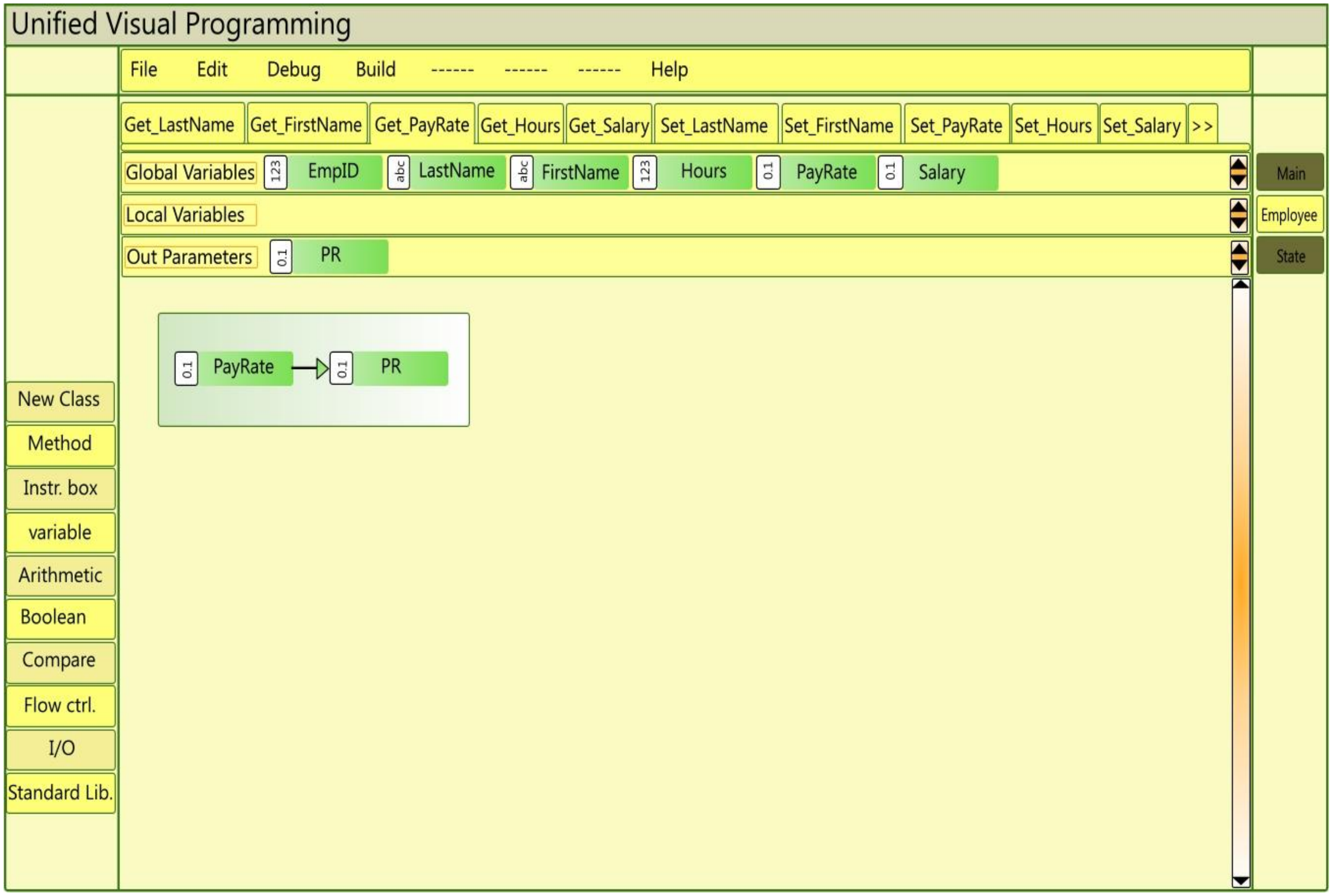
Help
Main
Employee
State
Libraries
Variables
123 StateID
abc StateName
0.1 Tax
0.1 Minwage
Get_StateName abc
Body
Get_Tax 0.1
Body
Get_MinWage 0.1
Body
Set_StateName
Body
abc Given_SN
Set_Tax
Body
0.1 Given_tax
Set_MinWage
Body
0.1 Given_MinW
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.
Main
Employee
State


Unified Visual Programming
File
Edit
Debug
Build

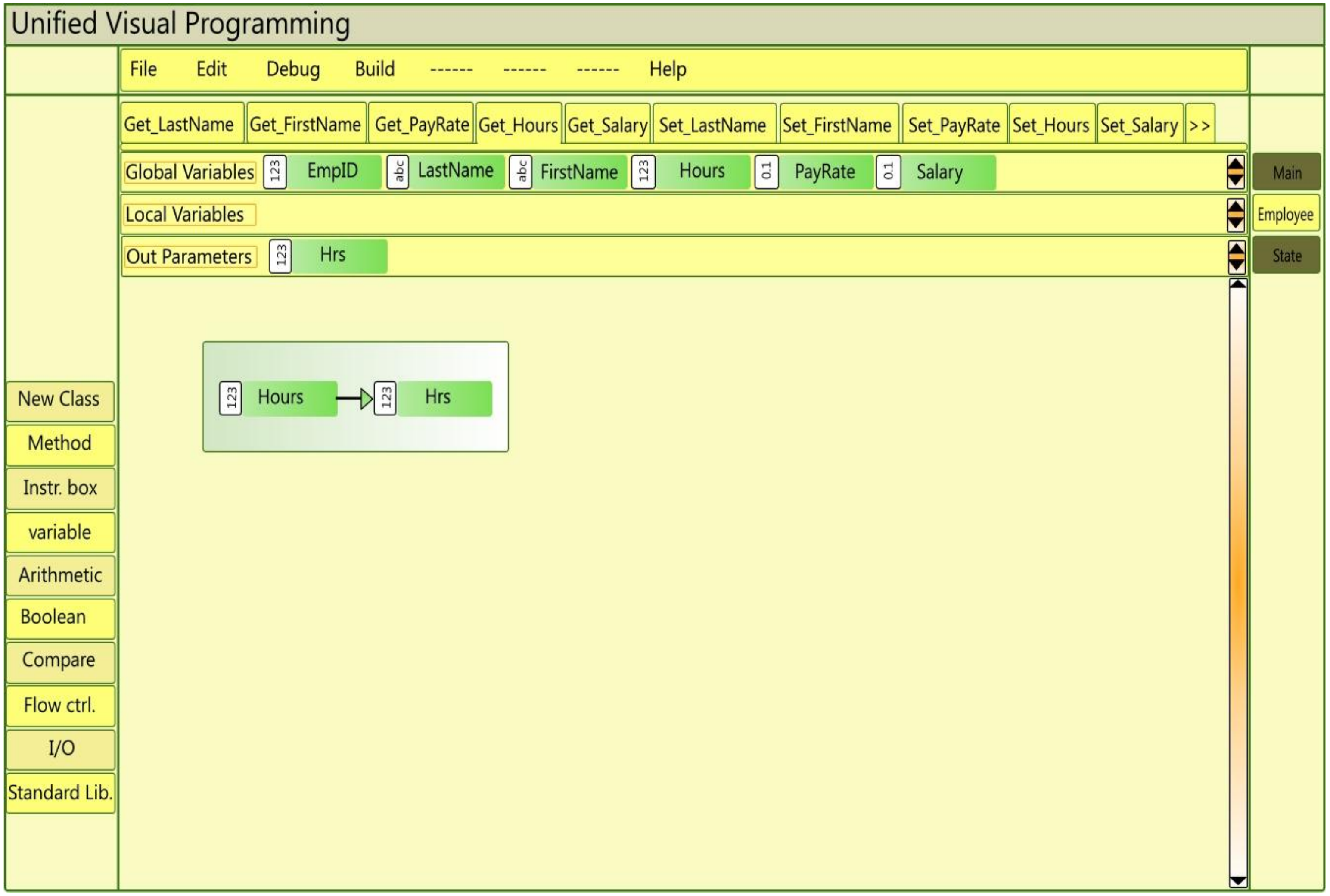
Help
Get_LastName
Get_FirstName
Get_PayRate
Get_Hours
Get_Salary
Set_LastName
Set_FirstName
Set_PayRate
Set_Hours
Set_Salary
>>
Global Variables
123 EmpID
abc LastName
abc FirstName
123 Hours
0.1 PayRate
0.1 Salary
Local Variables
Out Parameters
abc LN
abc LastName
abc LN
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.


Unified Visual Programming
File
Edit
Debug
Build

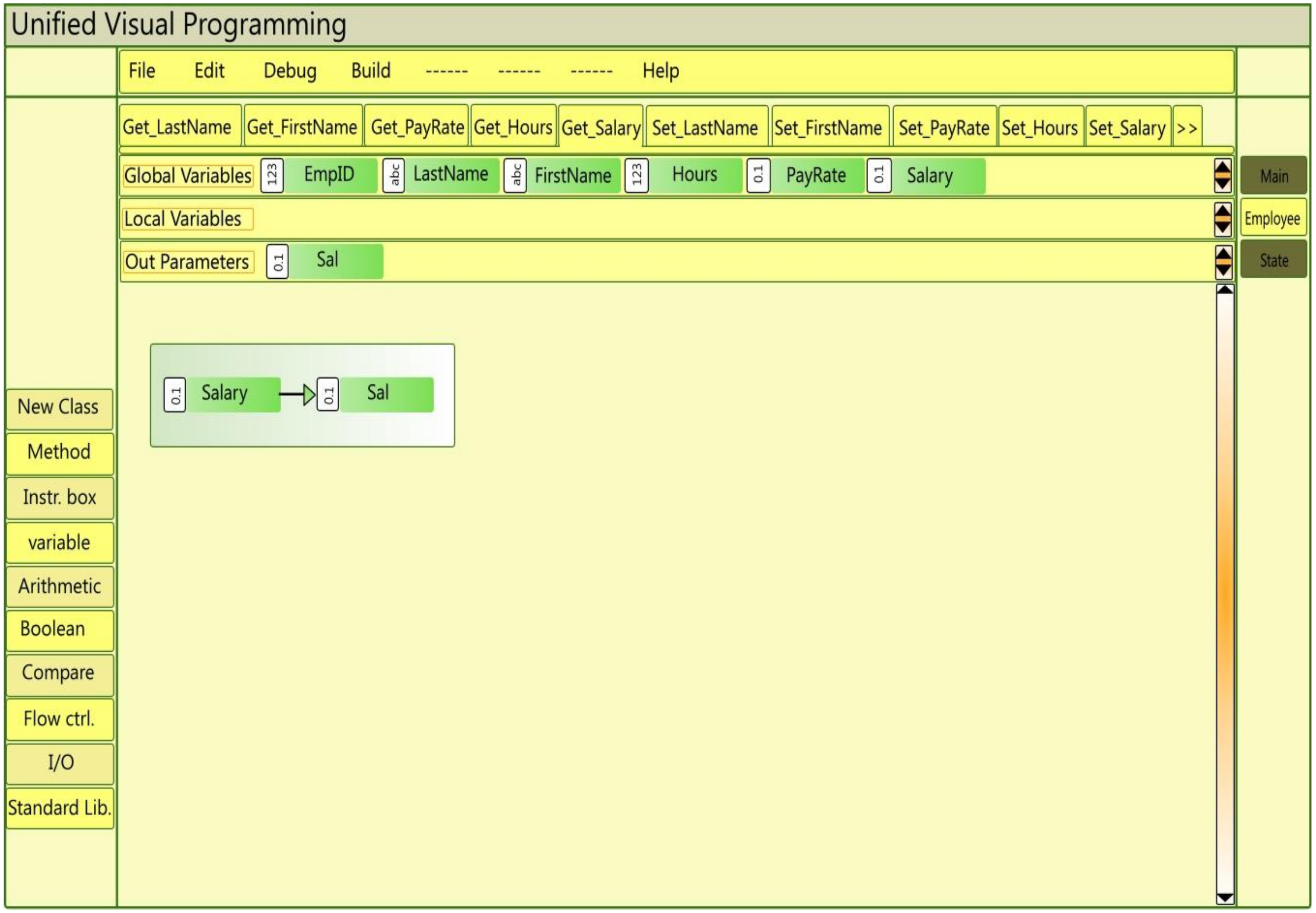
Help
Get_LastName
Get_FirstName
Get_PayRate
Get_Hours
Get_Salary
Set_LastName
Set_FirstName
Set_PayRate
Set_Hours
Set_Salary
>>
Global Variables
123 EmpID
abc LastName
abc FirstName
123 Hours
0.1 PayRate
0.1 Salary
Local Variables
Out Parameters
abc FN
Main
Employee
State
abc FirstName
abc FN
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.


Unified Visual Programming
File
Edit
Debug
Build

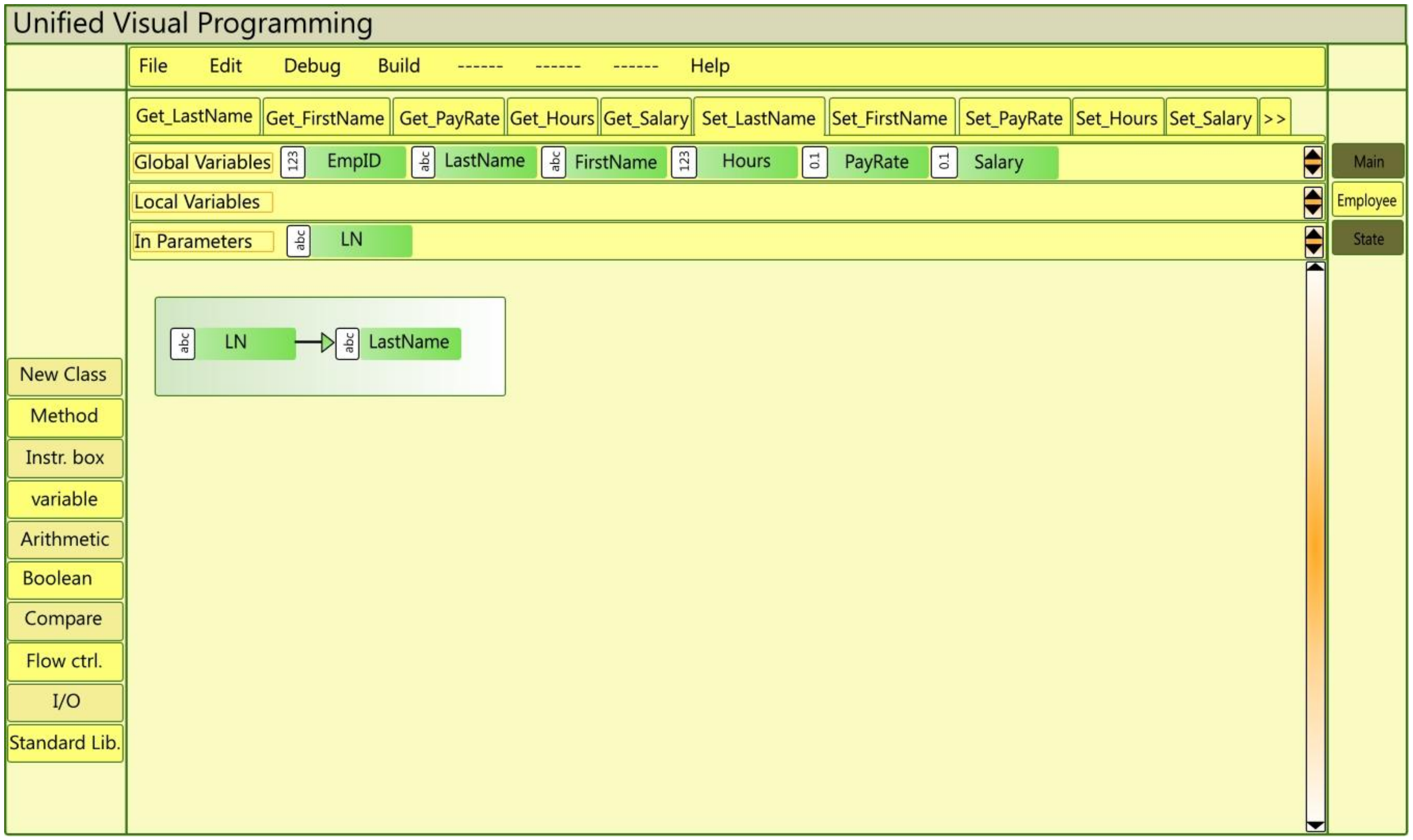
Help
Get_LastName
Get_FirstName
Get_PayRate
Get_Hours
Get_Salary
Set_LastName
Set_FirstName
Set_PayRate
Set_Hours
Set_Salary
>>
Global Variables
123
EmpID
abc
LastName
abc
FirstName
123
Hours
0.1
PayRate
0.1
Salary
Local Variables
Out Parameters
0.1
PR
0.1
PayRate
0.1
PR
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.


Unified Visual Programming
File
Edit
Debug
Build

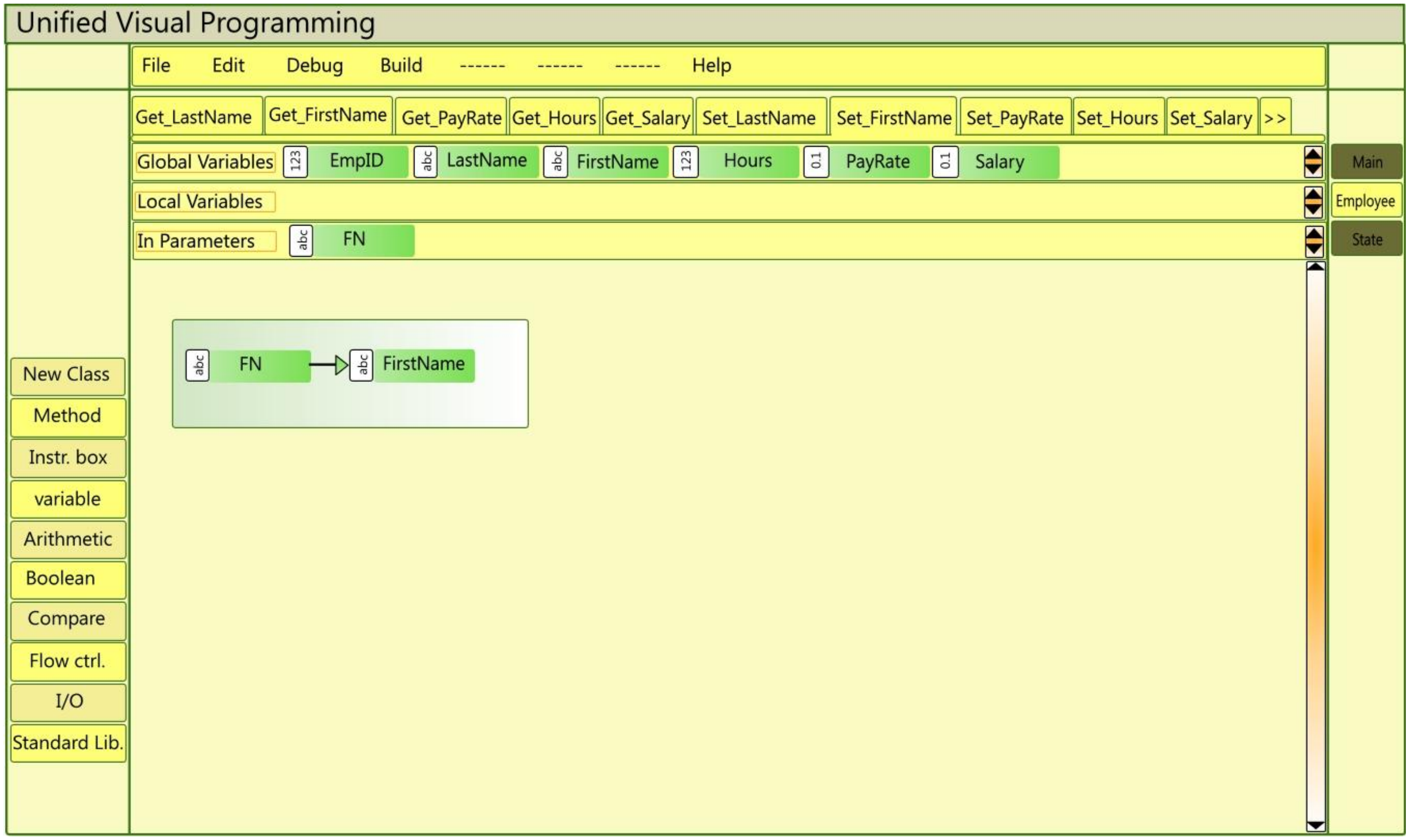
Help
Get_LastName
Get_FirstName
Get_PayRate
Get_Hours
Get_Salary
Set_LastName
Set_FirstName
Set_PayRate
Set_Hours
Set_Salary
>>
Global Variables
123 EmpID
abc LastName
abc FirstName
123 Hours
0.1 PayRate
0.1 Salary
Local Variables
Out Parameters
123 Hrs
123 Hours
123 Hrs
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.


Unified Visual Programming
File
Edit
Debug
Build

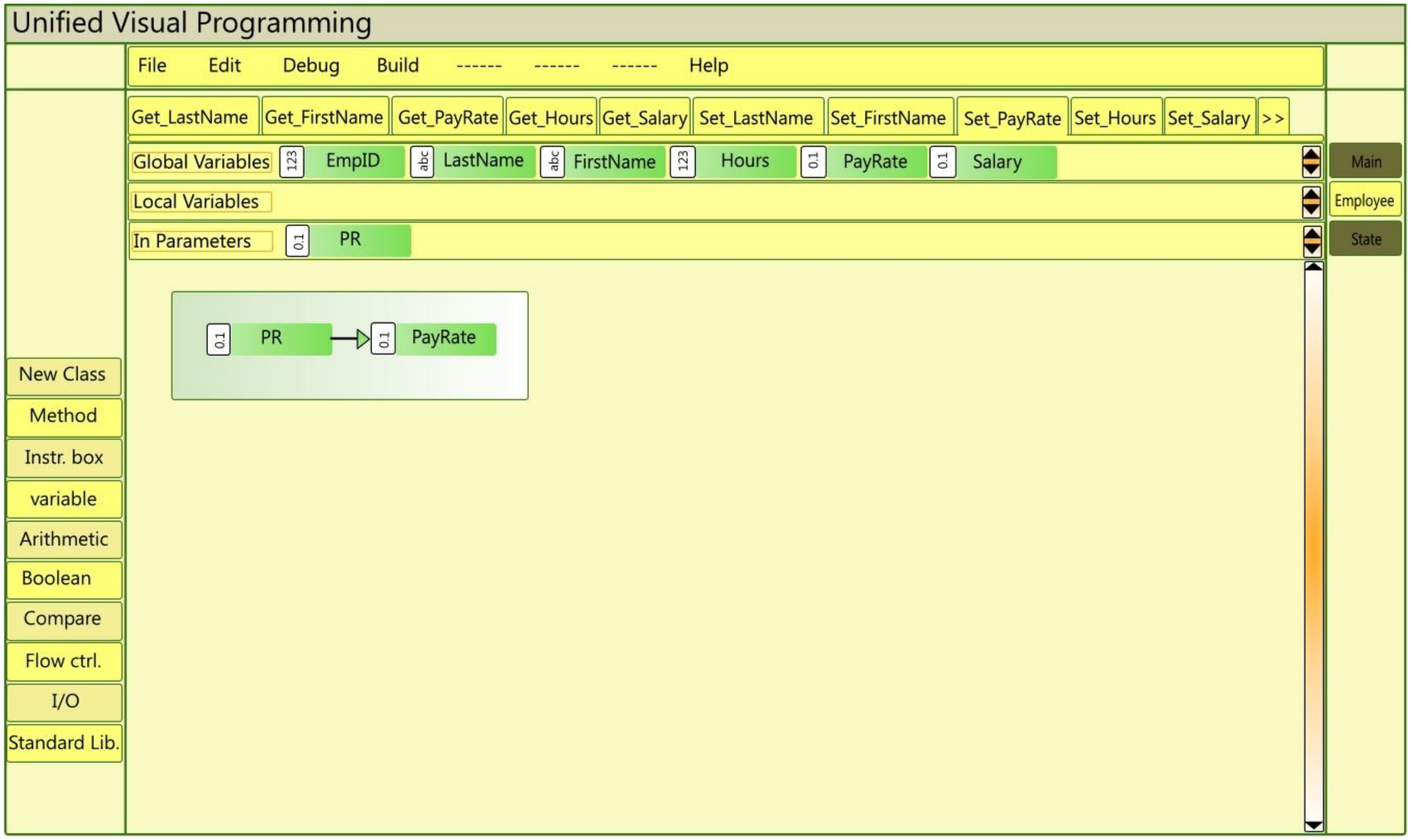
Help
Get_LastName
Get_FirstName
Get_PayRate
Get_Hours
Get_Salary
Set_LastName
Set_FirstName
Set_PayRate
Set_Hours
Set_Salary
>>
Global Variables
123 EmpID
abc LastName
abc FirstName
123 Hours
0.1 PayRate
0.1 Salary
Local Variables
Out Parameters
0.1 Sal
0.1 Salary
0.1 Sal
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.


Unified Visual Programming
File
Edit
Debug
Build

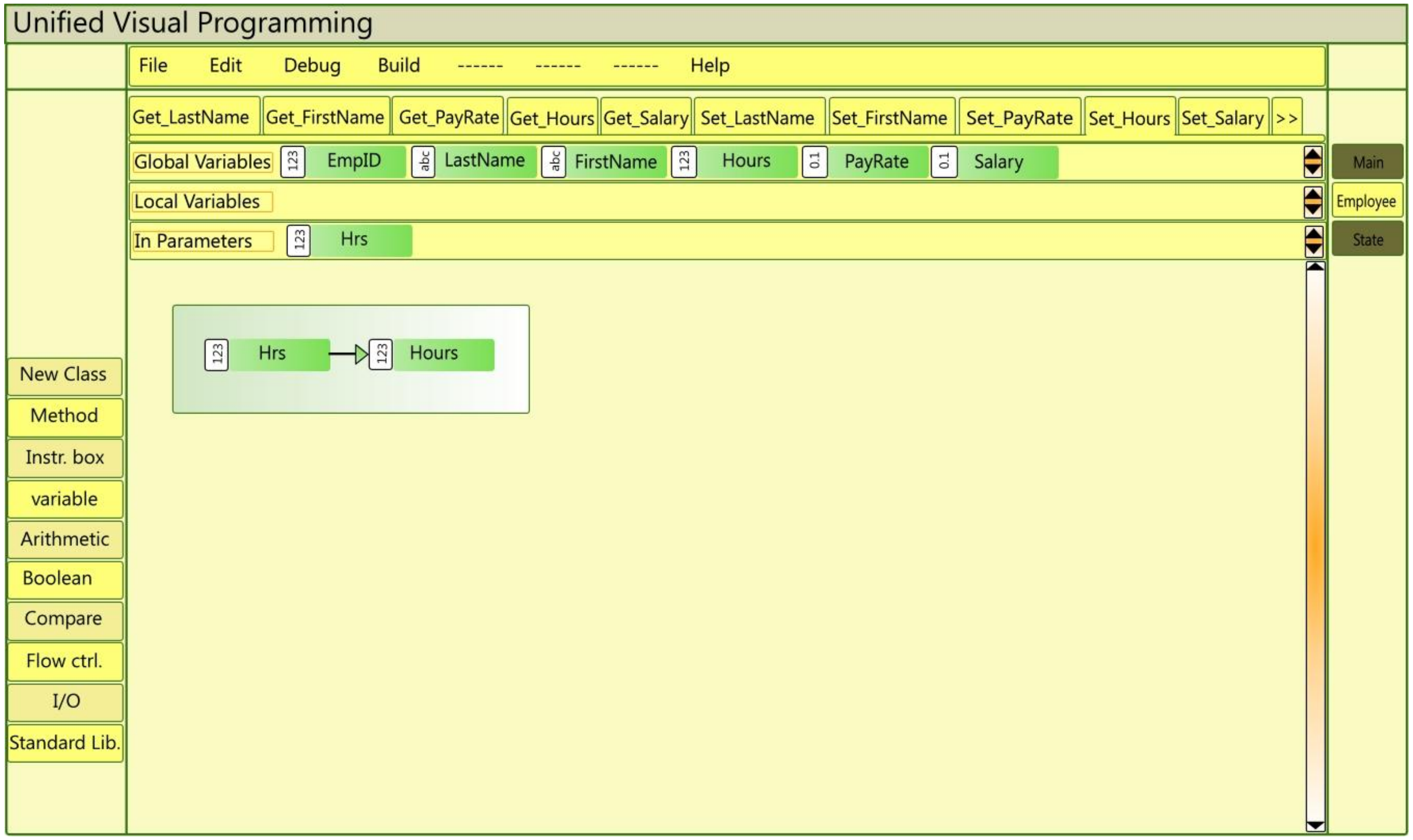
Help
Get_LastName
Get_FirstName
Get_PayRate
Get_Hours
Get_Salary
Set_LastName
Set_FirstName
Set_PayRate
Set_Hours
Set_Salary
> >
Global Variables
123 EmpID
abc LastName
abc FirstName
123 Hours
0.1 PayRate
0.1 Salary
Local Variables
In Parameters
abc LN
abc LN
abc LastName
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.


Unified Visual Programming
File
Edit
Debug
Build

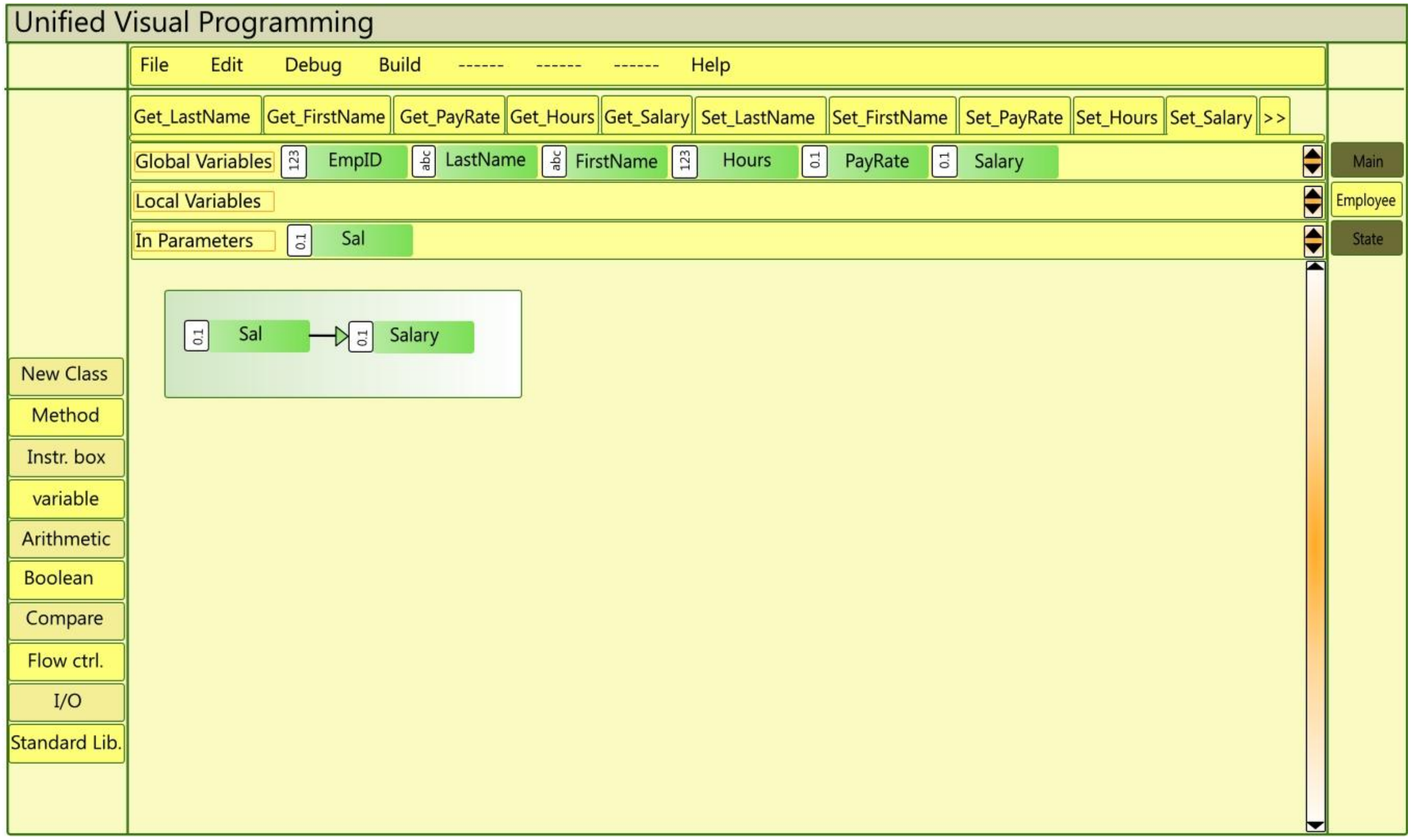
Help
Get_LastName
Get_FirstName
Get_PayRate
Get_Hours
Get_Salary
Set_LastName
Set_FirstName
Set_PayRate
Set_Hours
Set_Salary
>>
Global Variables
123 EmpID
abc LastName
abc FirstName
123 Hours
0.1 PayRate
0.1 Salary
Local Variables
In Parameters
abc FN
abc FN
abc FirstName
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.


Unified Visual Programming
File
Edit
Debug
Build

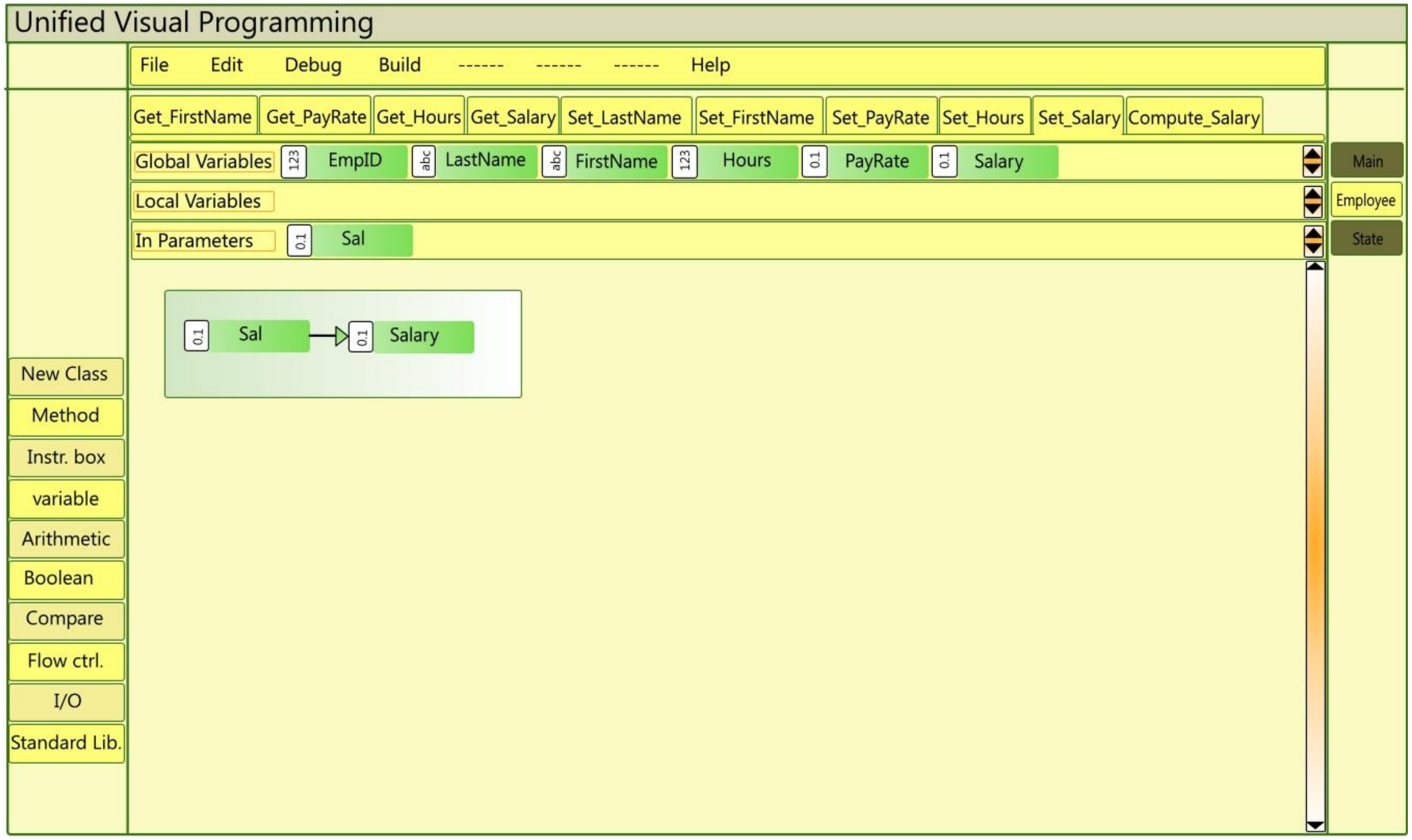
Help
Get_LastName
Get_FirstName
Get_PayRate
Get_Hours
Get_Salary
Set_LastName
Set_FirstName
Set_PayRate
Set_Hours
Set_Salary
>>
Global Variables
123 EmpID
abc LastName
abc FirstName
123 Hours
0.1 PayRate
0.1 Salary
Local Variables
In Parameters
0.1 PR
0.1 PR
0.1 PayRate
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.


Unified Visual Programming
File
Edit
Debug
Build

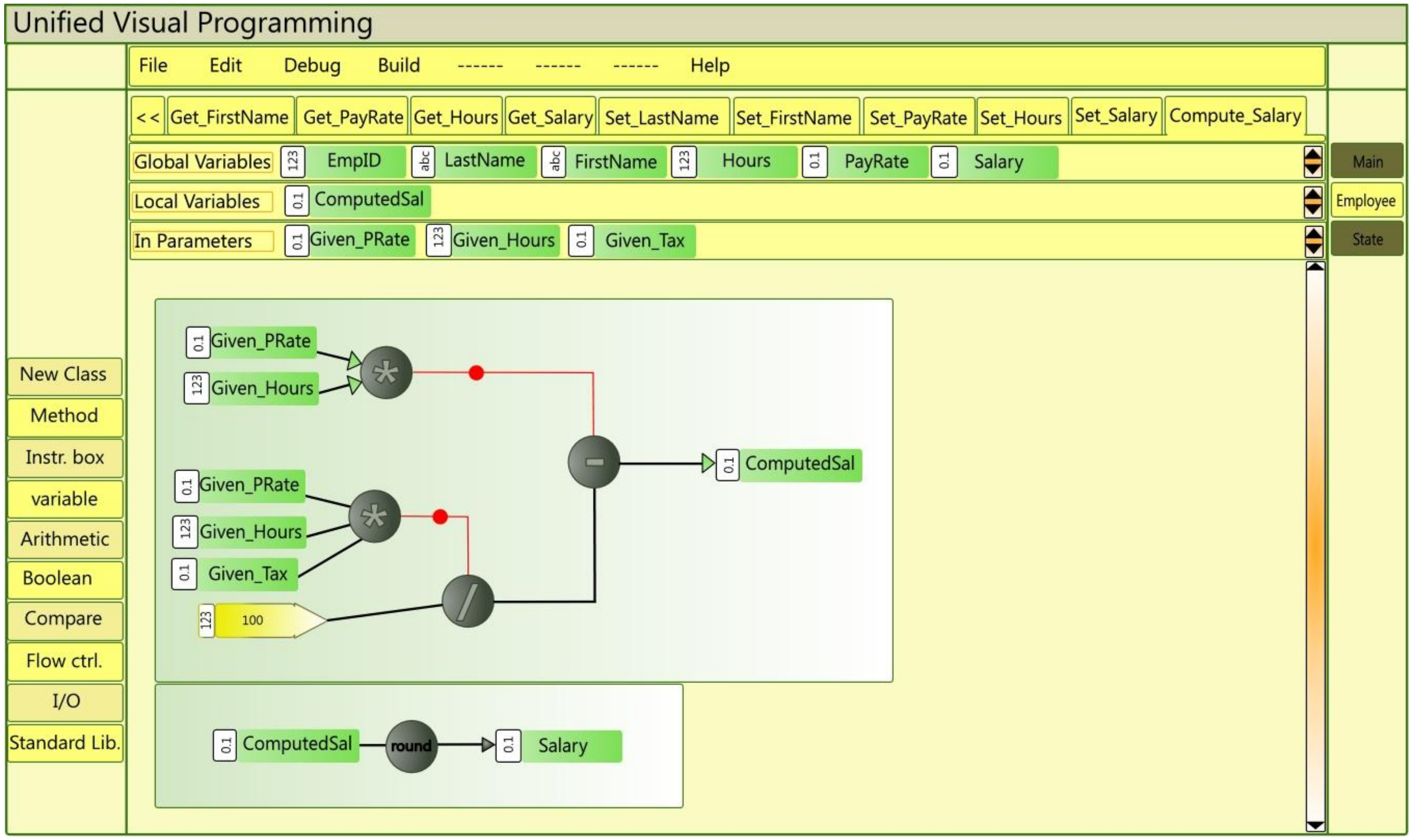
Help
Get_LastName
Get_FirstName
Get_PayRate
Get_Hours
Get_Salary
Set_LastName
Set_FirstName
Set_PayRate
Set_Hours
Set_Salary
>>
Global Variables
123 EmpID
abc LastName
abc FirstName
123 Hours
0.1 PayRate
0.1 Salary
Local Variables
In Parameters
123 Hrs
123 Hrs
123 Hours
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.


Unified Visual Programming
File
Edit
Debug
Build

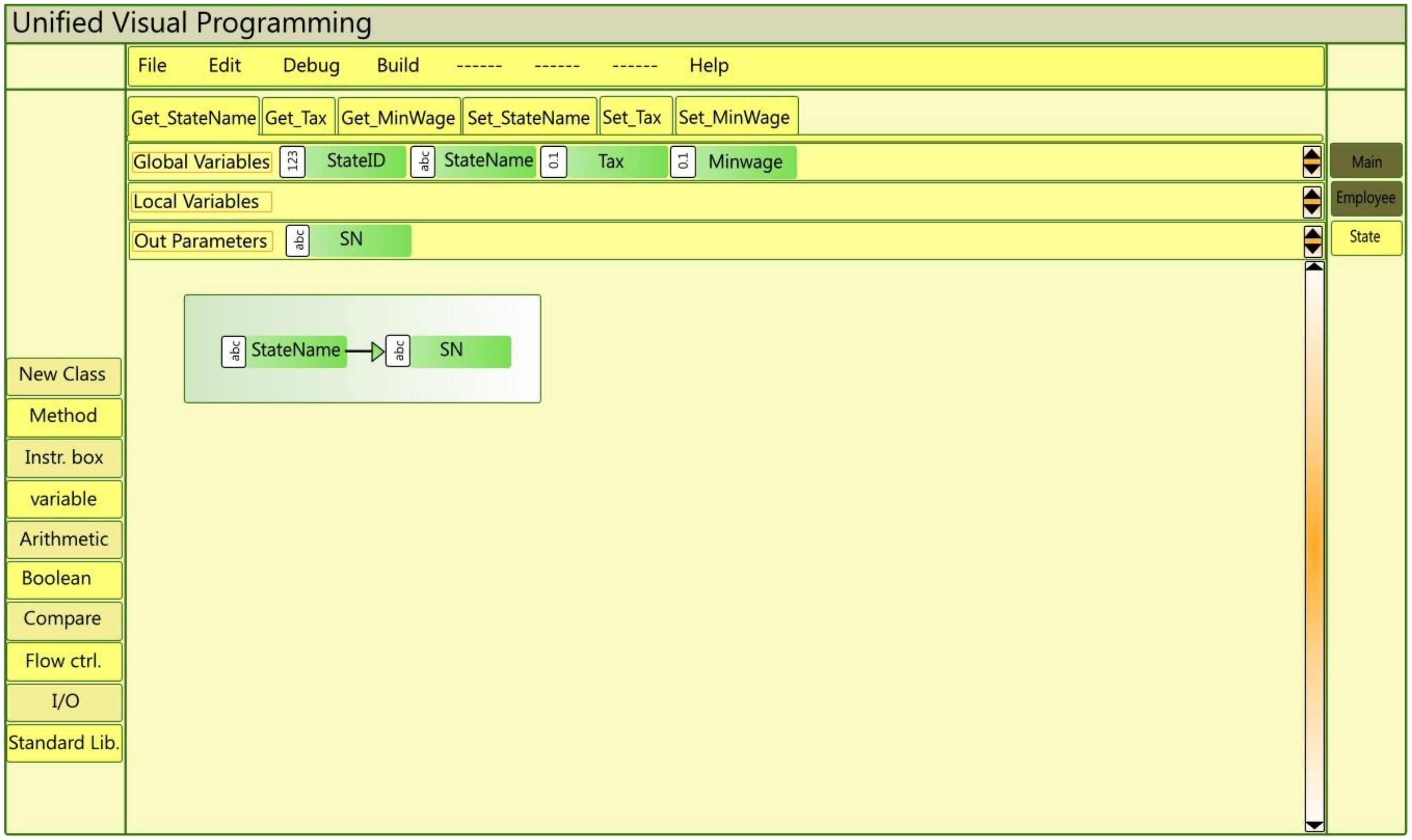
Help
Get_LastName
Get_FirstName
Get_PayRate
Get_Hours
Get_Salary
Set_LastName
Set_FirstName
Set_PayRate
Set_Hours
Set_Salary
>>
Global Variables
123 EmpID
abc LastName
abc FirstName
123 Hours
0.1 PayRate
0.1 Salary
Local Variables
In Parameters
0.1 Sal
0.1 Sal
0.1 Salary
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.


Unified Visual Programming
File
Edit
Debug
Build

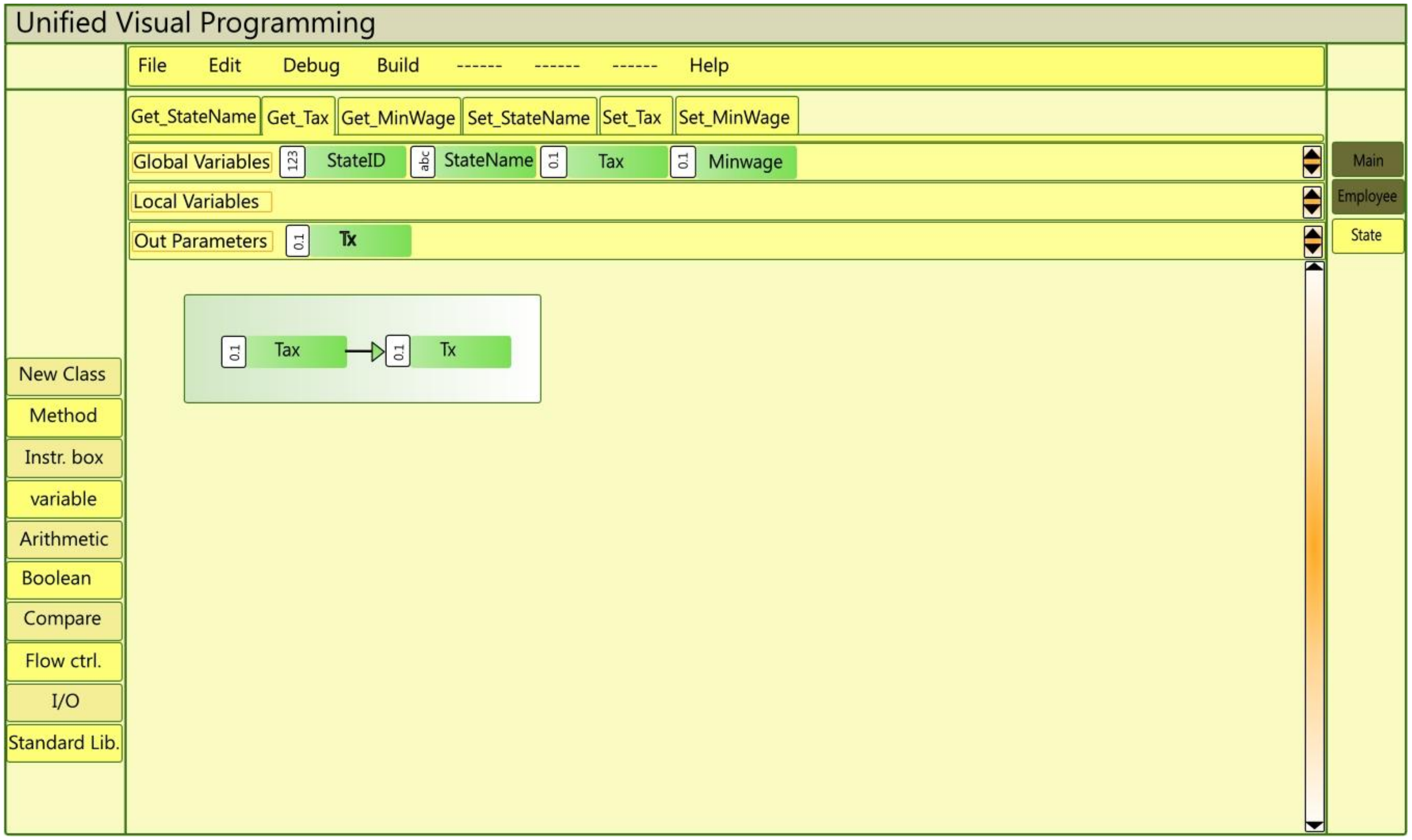
Help
Get_FirstName
Get_PayRate
Get_Hours
Get_Salary
Set_LastName
Set_FirstName
Set_PayRate
Set_Hours
Set_Salary
Compute_Salary
Global Variables
123 EmpID
abc LastName
abc FirstName
123 Hours
0.1 PayRate
0.1 Salary
Local Variables
In Parameters
0.1 Sal
0.1 Sal
0.1 Salary
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.


Unified Visual Programming
File
Edit
Debug
Build

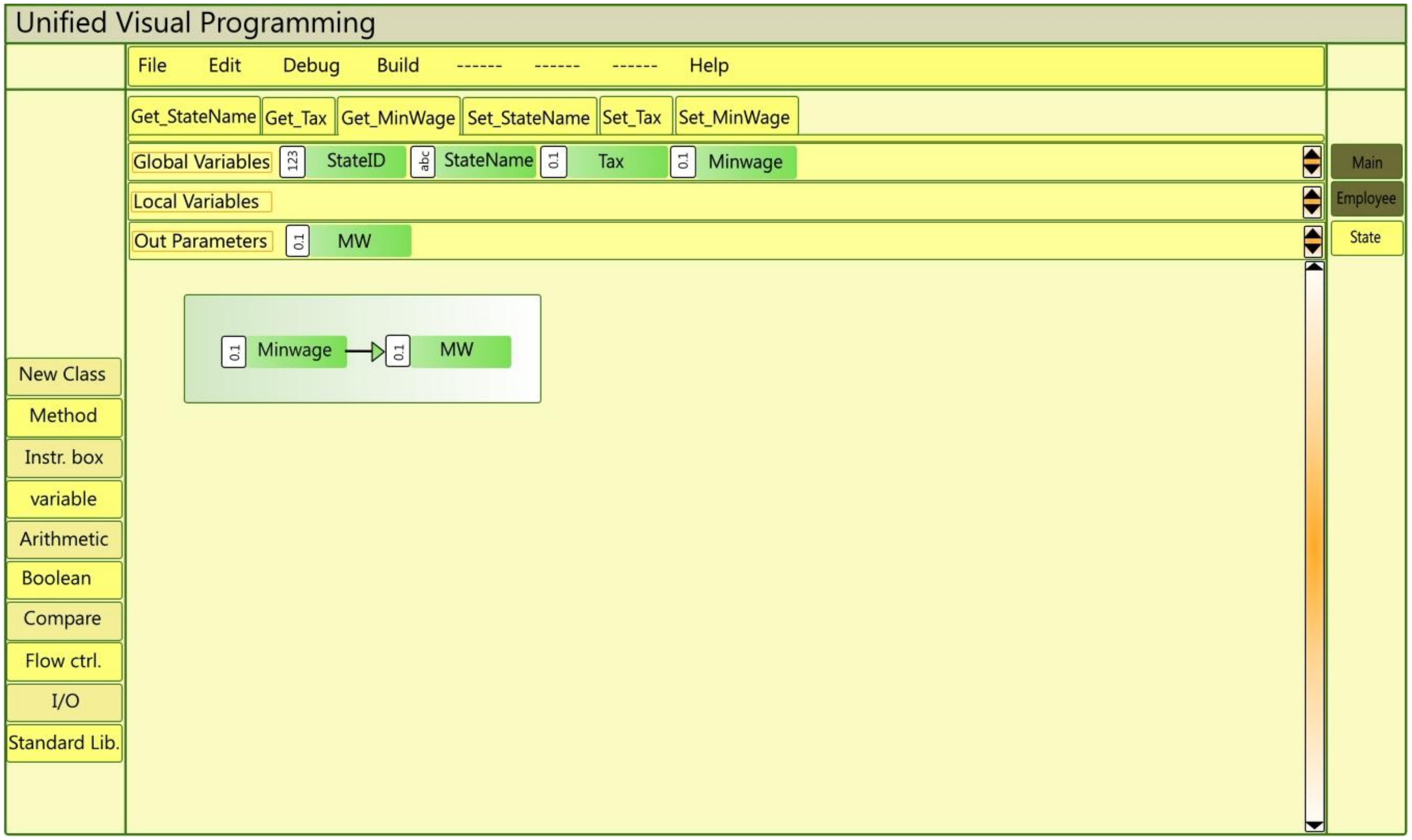
Help
< <
Get_FirstName
Get_PayRate
Get_Hours
Get_Salary
Set_LastName
Set_FirstName
Set_PayRate
Set_Hours
Set_Salary
Compute_Salary
Global Variables
123 EmpID
abc LastName
abc FirstName
123 Hours
0.1 PayRate
0.1 Salary
Local Variables
0.1 ComputedSal
In Parameters
0.1 Given_PRate
123 Given_Hours
0.1 Given_Tax
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.
0.1 Given_PRate
123 Given_Hours
*
-
0.1 ComputedSal
0.1 Given_PRate
123 Given_Hours
0.1 Given_Tax
*
123 100
/
0.1 ComputedSal
round
0.1 Salary


Unified Visual Programming
File
Edit
Debug
Build

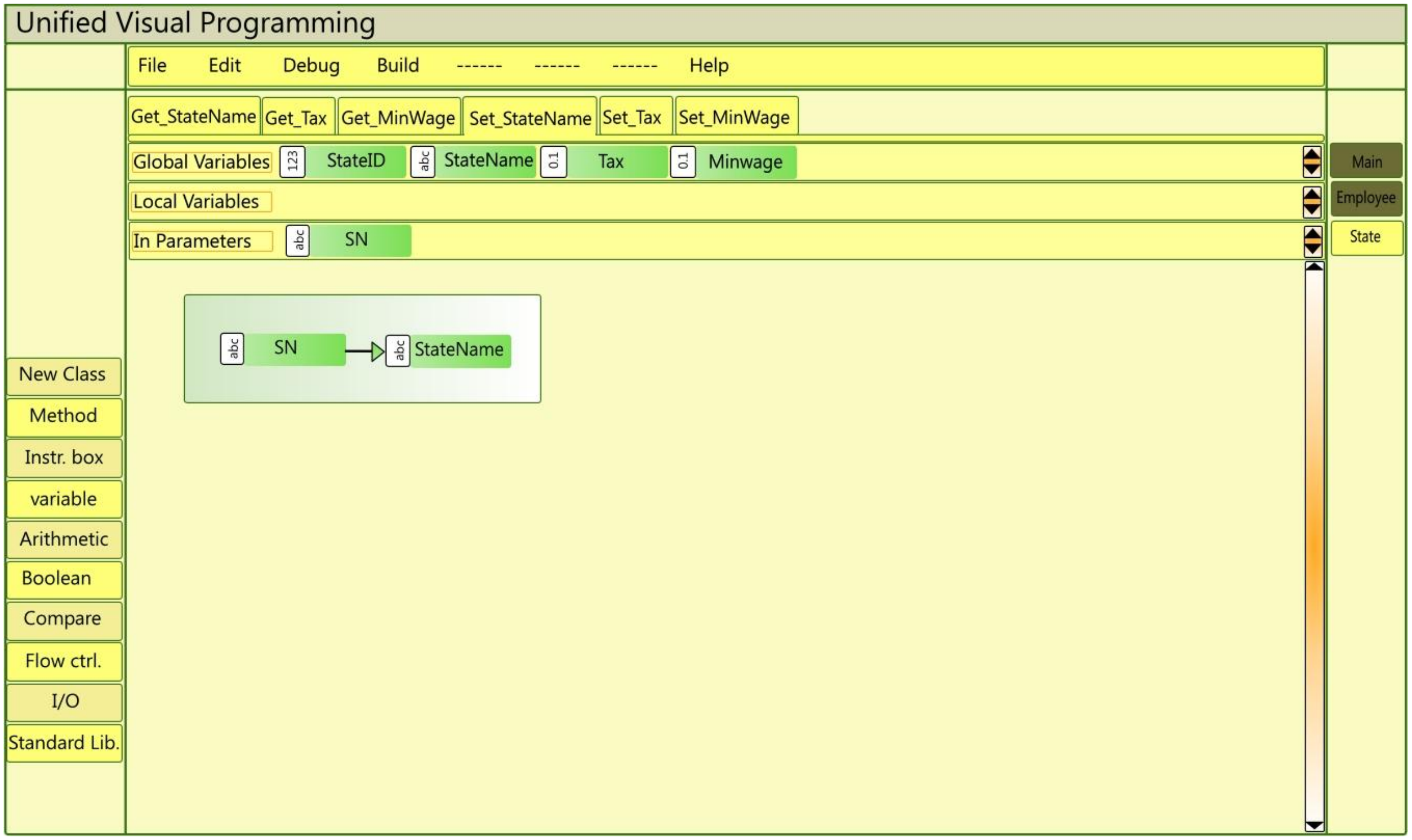
Help
Get_StateName
Get_Tax
Get_MinWage
Set_StateName
Set_Tax
Set_MinWage
Global Variables
123
StateID
abc
StateName
0.1
Tax
0.1
Minwage
Local Variables
Out Parameters
abc
SN
abc
StateName
abc
SN
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.


Unified Visual Programming
File
Edit
Debug
Build

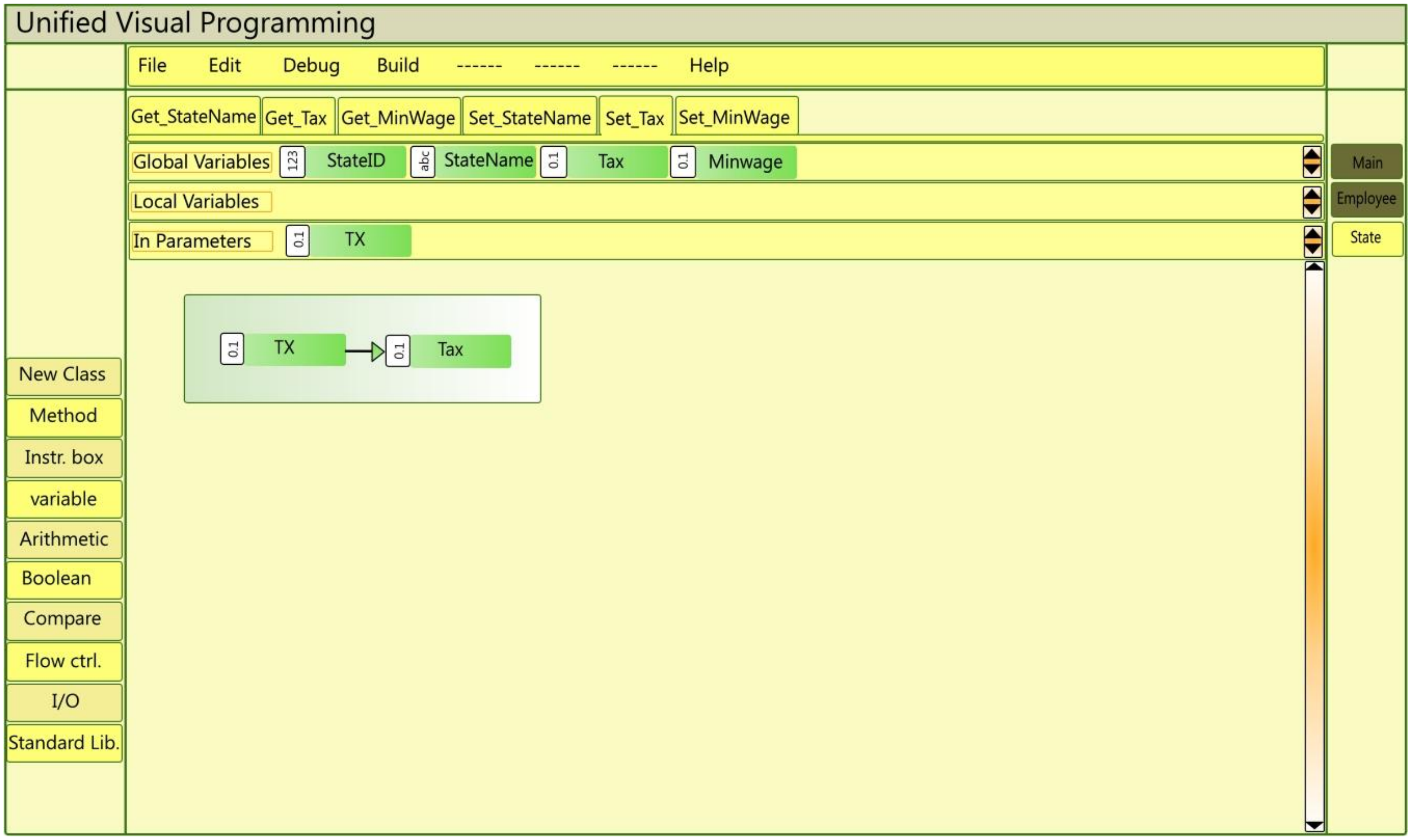
Help
Get_StateName
Get_Tax
Get_MinWage
Set_StateName
Set_Tax
Set_MinWage
Global Variables
123
StateID
abc
StateName
0.1
Tax
0.1
Minwage
Local Variables
Out Parameters
0.1
Tx
0.1
Tax
0.1
Tx
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.


Unified Visual Programming
File
Edit
Debug
Build

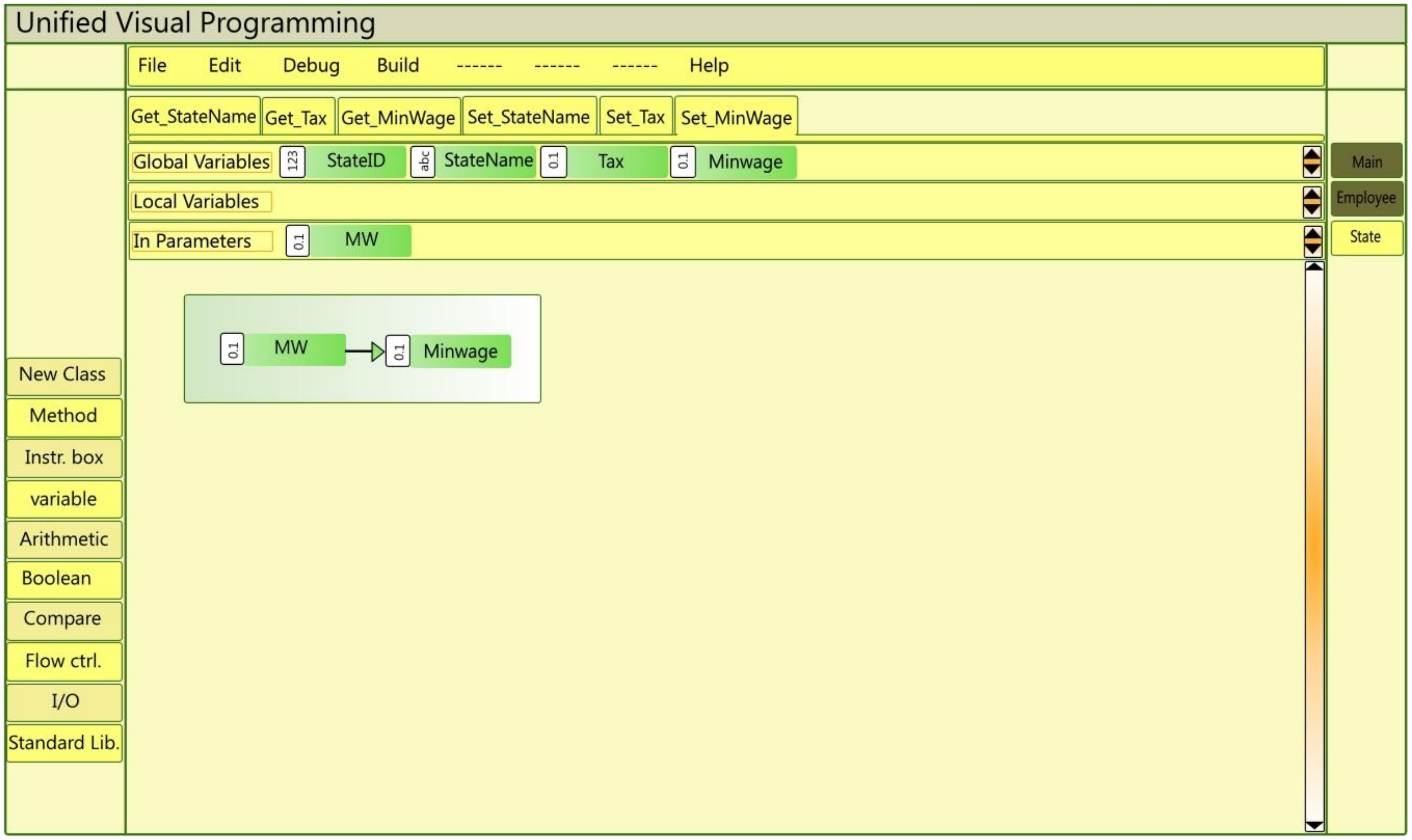
Help
Get_StateName
Get_Tax
Get_MinWage
Set_StateName
Set_Tax
Set_MinWage
Global Variables
123
StateID
abc
StateName
0.1
Tax
0.1
Minwage
Local Variables
Out Parameters
0.1
MW
0.1
Minwage
0.1
MW
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.


Unified Visual Programming
File
Edit
Debug
Build

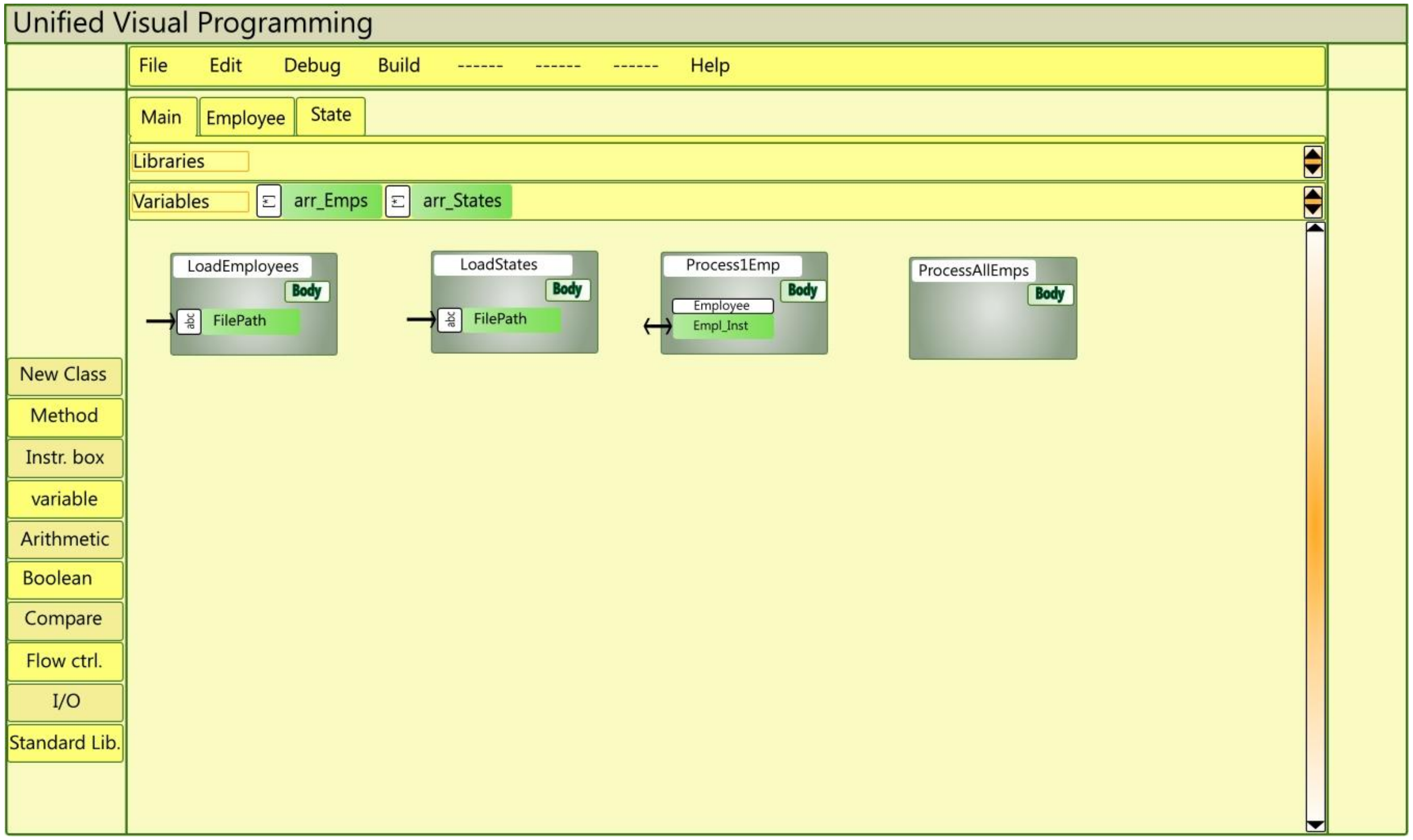
Help
Get_StateName
Get_Tax
Get_MinWage
Set_StateName
Set_Tax
Set_MinWage
Global Variables
123
StateID
abc
StateName
0.1
Tax
0.1
Minwage
Local Variables
In Parameters
abc
SN
abc
SN
abc
StateName
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.


Unified Visual Programming
File
Edit
Debug
Build

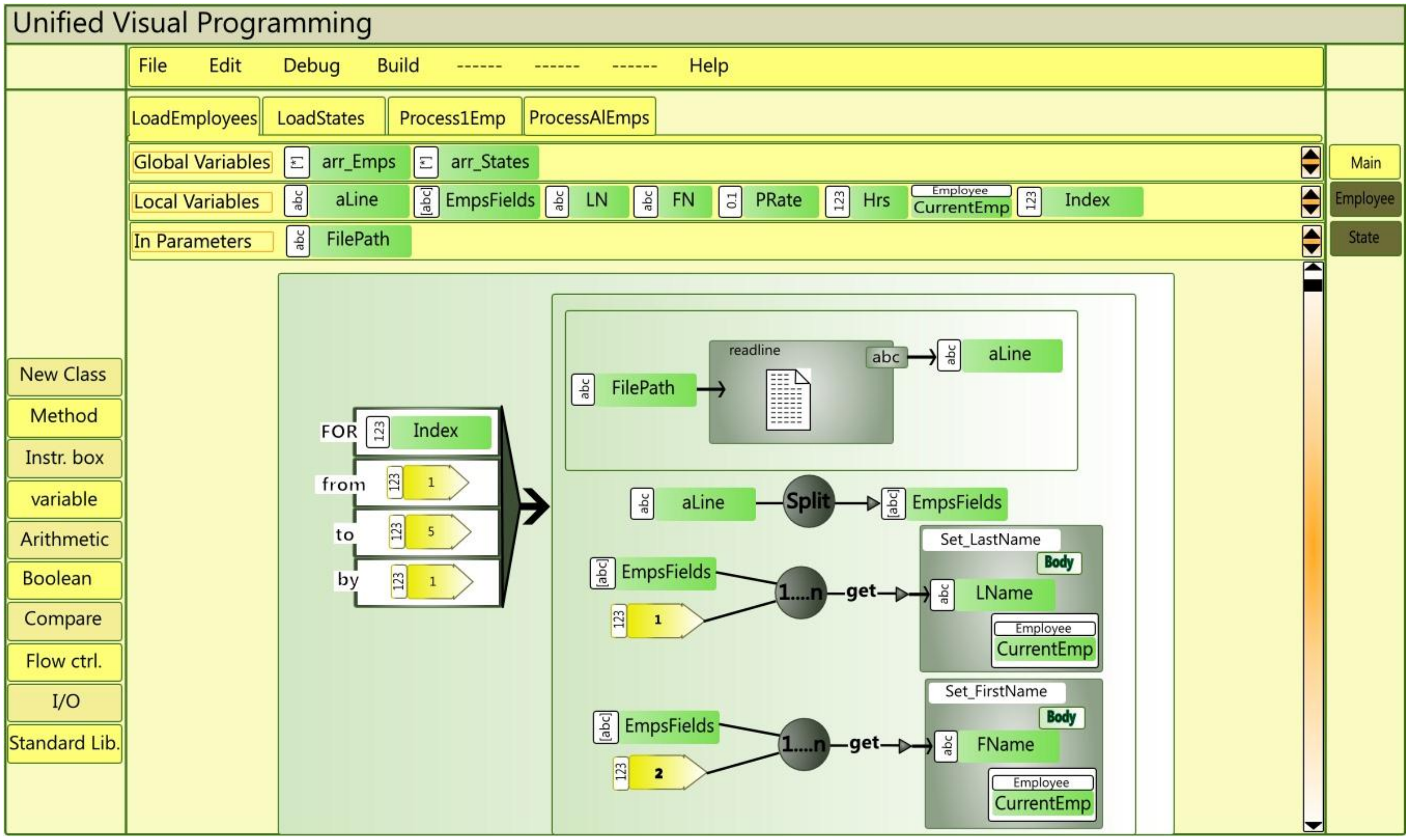
Help
Get_StateName
Get_Tax
Get_MinWage
Set_StateName
Set_Tax
Set_MinWage
Global Variables
123
StateID
abc
StateName
0.1
Tax
0.1
Minwage
Local Variables
In Parameters
0.1
TX
0.1
TX
0.1
Tax
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.


Unified Visual Programming
File
Edit
Debug
Build

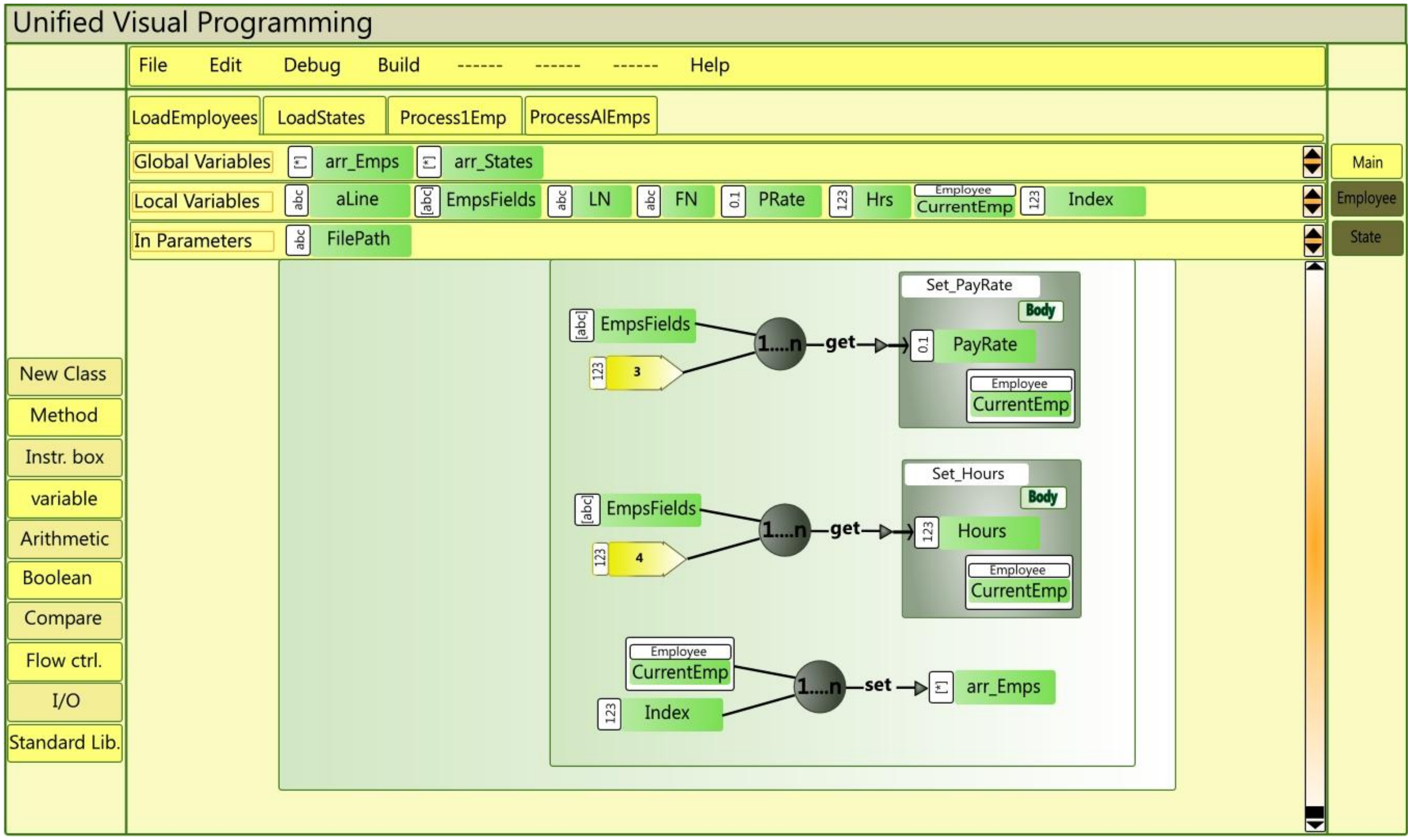
Help
Get_StateName
Get_Tax
Get_MinWage
Set_StateName
Set_Tax
Set_MinWage
Global Variables
123
StateID
abc
StateName
0.1
Tax
0.1
Minwage
Local Variables
In Parameters
0.1
MW
0.1
MW
0.1
Minwage
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.


Unified Visual Programming
File
Edit
Debug
Build

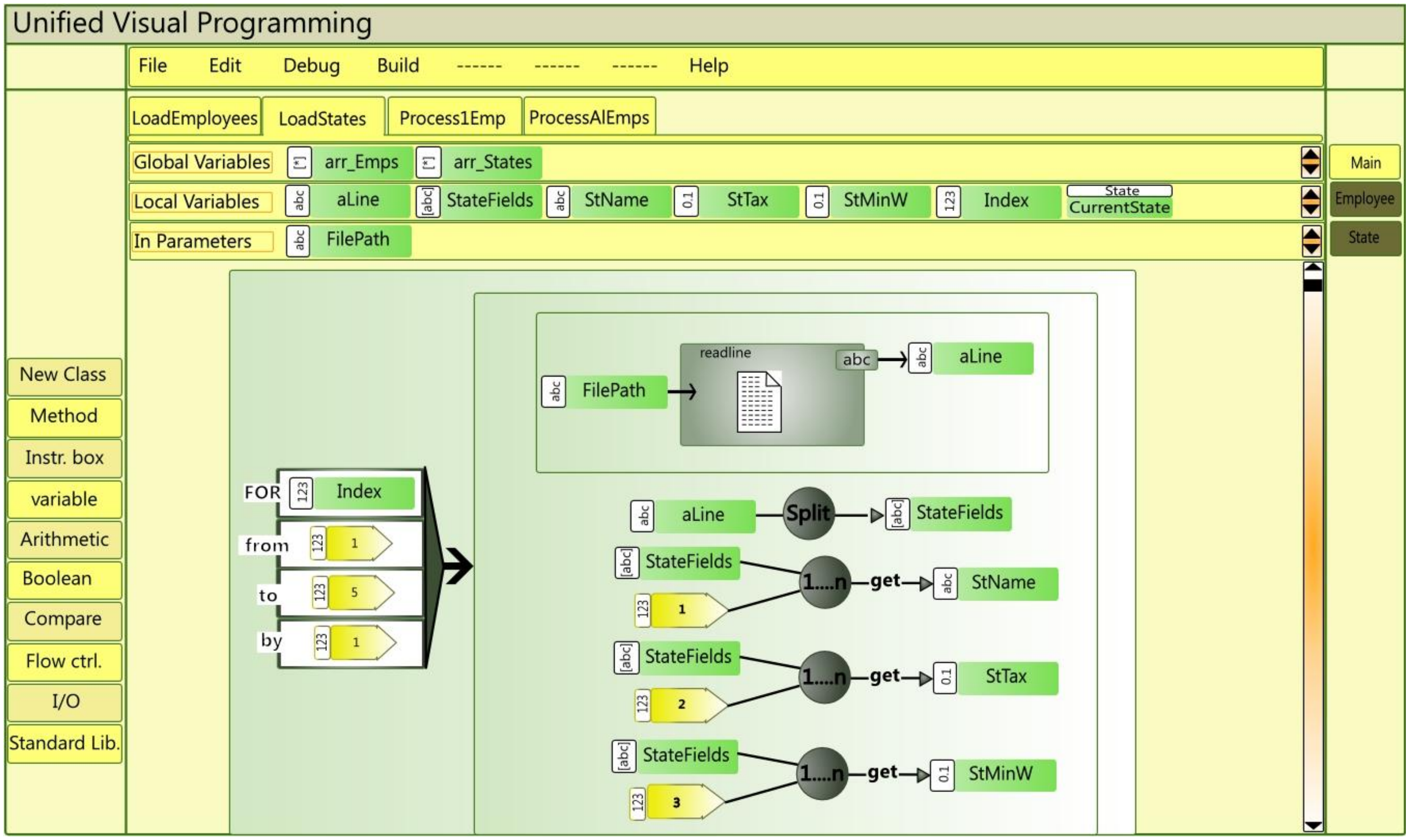
Help
Main
Employee
State
Libraries
Variables
arr_Emps
arr_States
LoadEmployees
Body
abc
FilePath
LoadStates
Body
abc
FilePath
Process1Emp
Body
Employee
Empl_Inst
ProcessAllEmps
Body
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.


Unified Visual Programming
File
Edit
Debug
Build

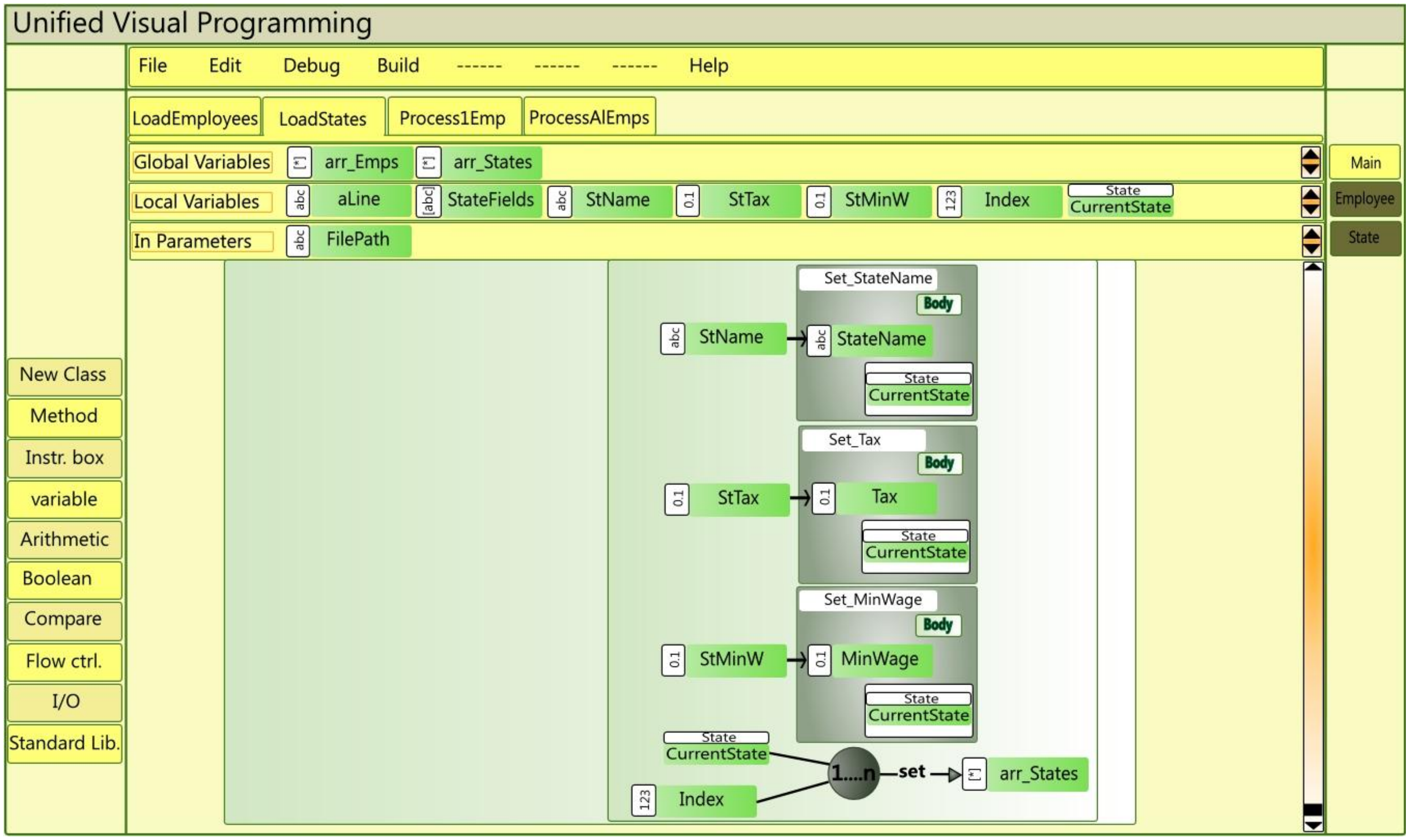
Help
LoadEmployees
LoadStates
Process1Emp
ProcessAlEmps
Global Variables
arr_Emps
arr_States
Local Variables
aLine
EmpsFields
LN
FN
PRate
Hrs
Employee
CurrentEmp
Index
In Parameters
FilePath
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.
FOR
Index
from
1
to
5
by
1
readline
FilePath
abc
aLine
aLine
Split
EmpsFields
EmpsFields
1
1....n
get
Set_LastName
Body
LName
Employee
CurrentEmp
EmpsFields
2
1....n
get
Set_FirstName
Body
FName
Employee
CurrentEmp


Unified Visual Programming
File
Edit
Debug
Build

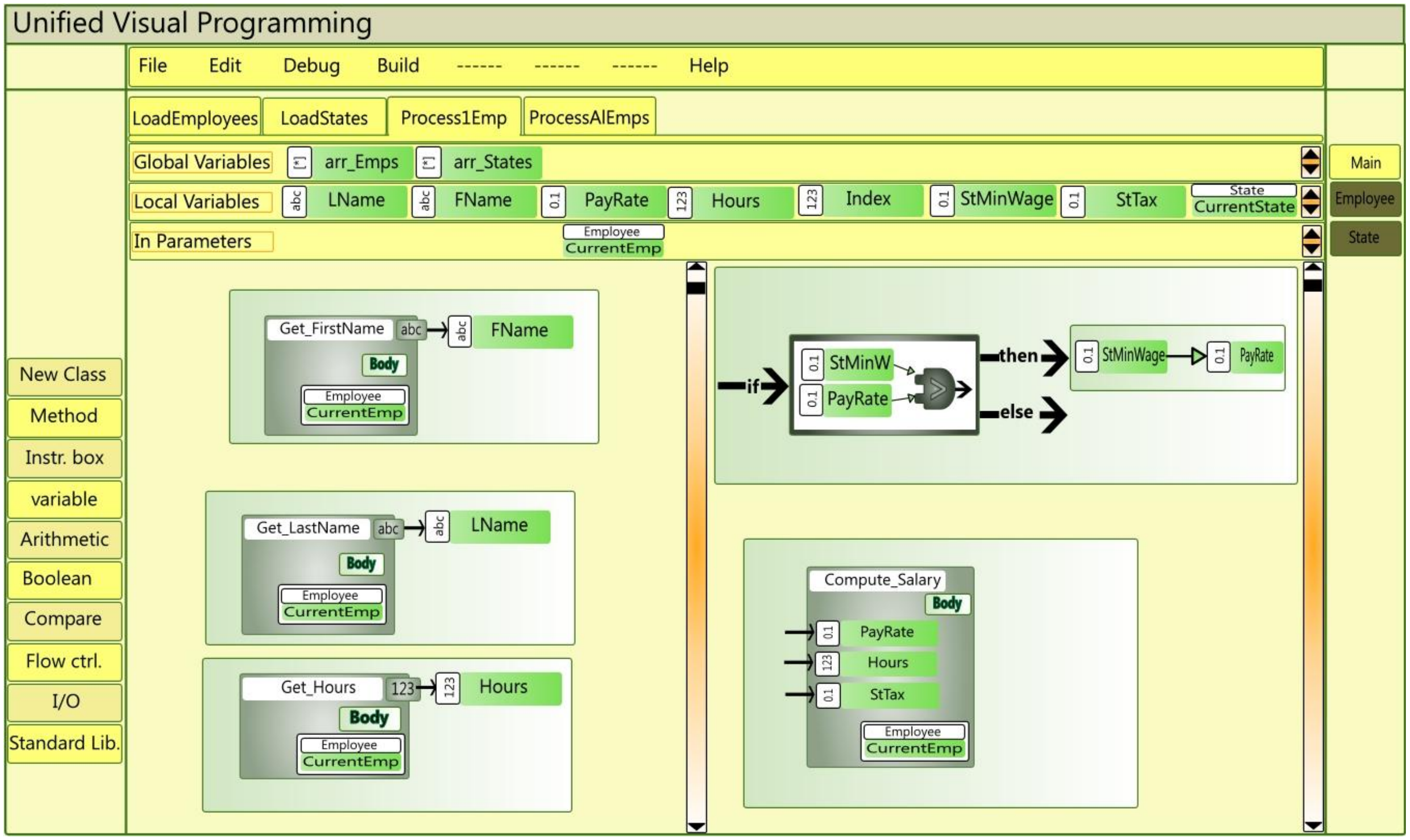
Help
LoadEmployees
LoadStates
Process1Emp
ProcessAlEmps
Global Variables
arr_Emps
arr_States
Local Variables
abc aLine
[abc] EmpsFields
abc LN
abc FN
0.1 PRate
123 Hrs
Employee CurrentEmp
123 Index
In Parameters
abc FilePath
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.
Set_PayRate
Body
[abc] EmpsFields
123 3
1....n
get
0.1 PayRate
Employee CurrentEmp
Set_Hours
Body
[abc] EmpsFields
123 4
1....n
get
123 Hours
Employee CurrentEmp
Employee CurrentEmp
123 Index
1....n
set
arr_Emps


Unified Visual Programming
File
Edit
Debug
Build

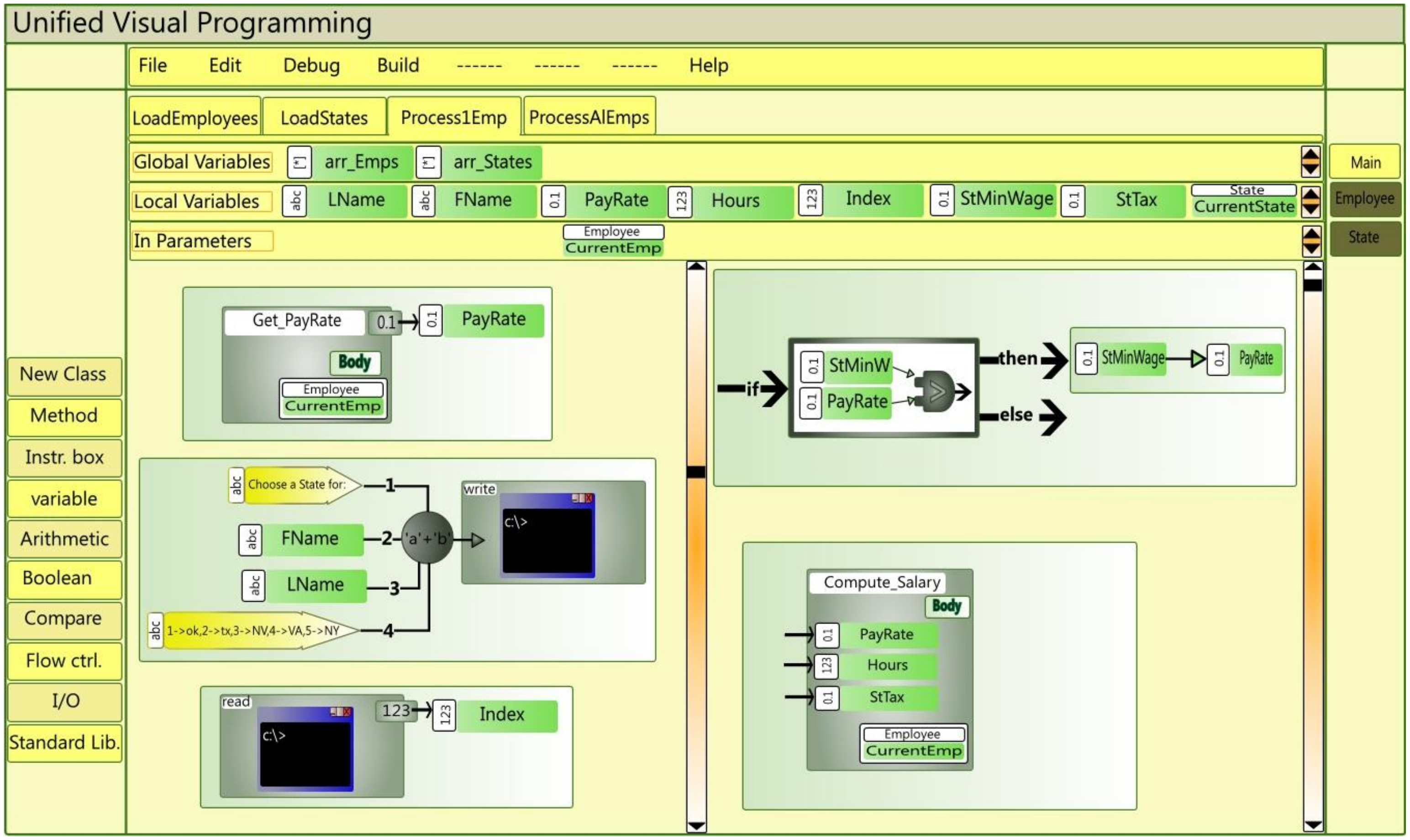
Help
LoadEmployees
LoadStates
Process1Emp
ProcessAlEmps
Global Variables
arr_Emps
arr_States
Local Variables
aLine
StateFields
StName
StTax
StMinW
Index
State
CurrentState
In Parameters
FilePath
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.
FOR
Index
from
1
to
5
by
1
readline
abc
FilePath
aLine
aLine
Split
StateFields
StateFields
1
1....n
get
StName
StateFields
2
1....n
get
StTax
StateFields
3
1....n
get
StMinW


Unified Visual Programming
File
Edit
Debug
Build

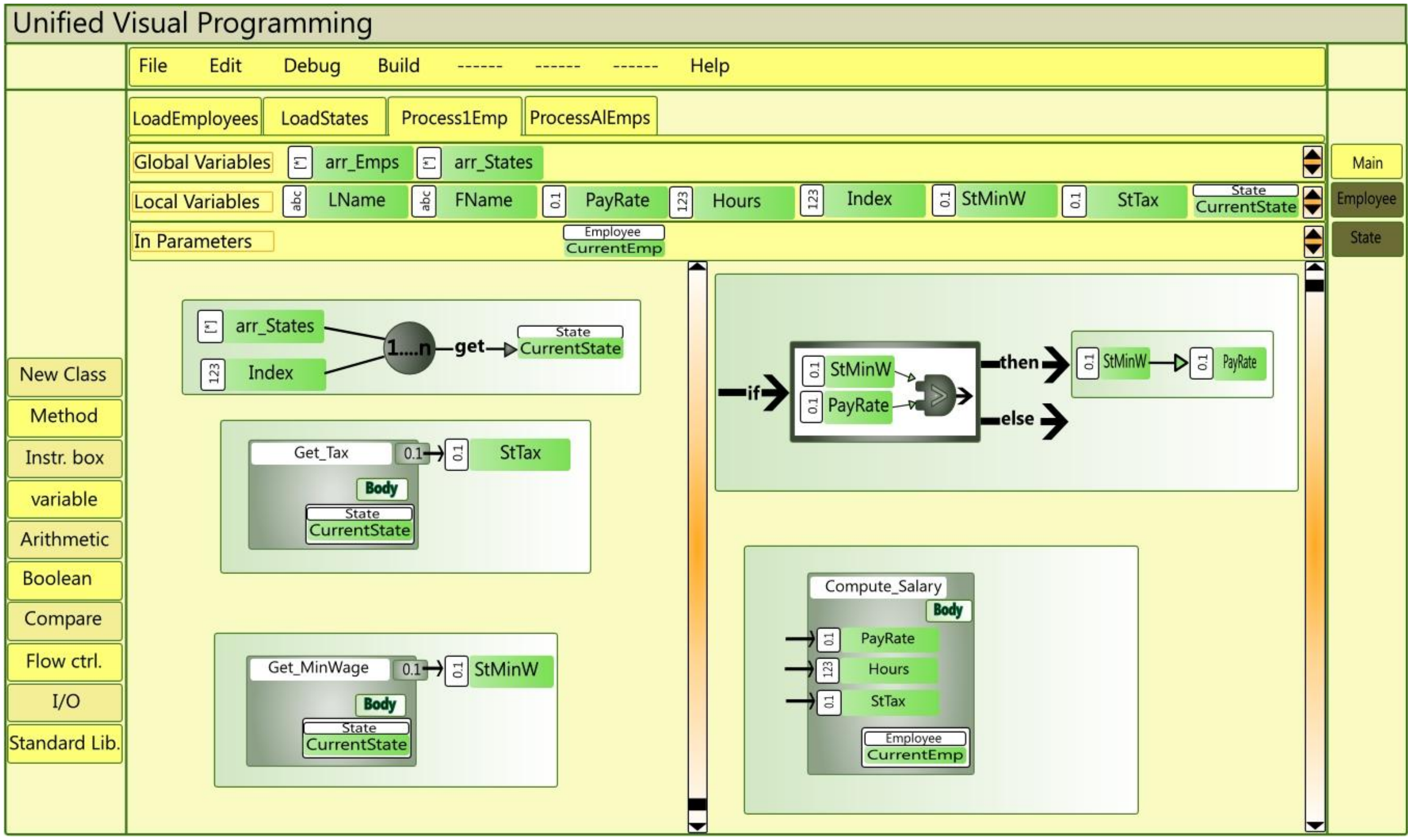
Help
LoadEmployees
LoadStates
Process1Emp
ProcessAlEmps
Global Variables
arr_Emps
arr_States
Local Variables
aLine
StateFields
StName
StTax
StMinW
Index
State
CurrentState
In Parameters
FilePath
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.
Set_StateName
Body
StName
StateName
State
CurrentState
Set_Tax
Body
StTax
Tax
State
CurrentState
Set_MinWage
Body
StMinW
MinWage
State
CurrentState
State
CurrentState
Index
1....n
set
arr_States


Unified Visual Programming
File
Edit
Debug
Build

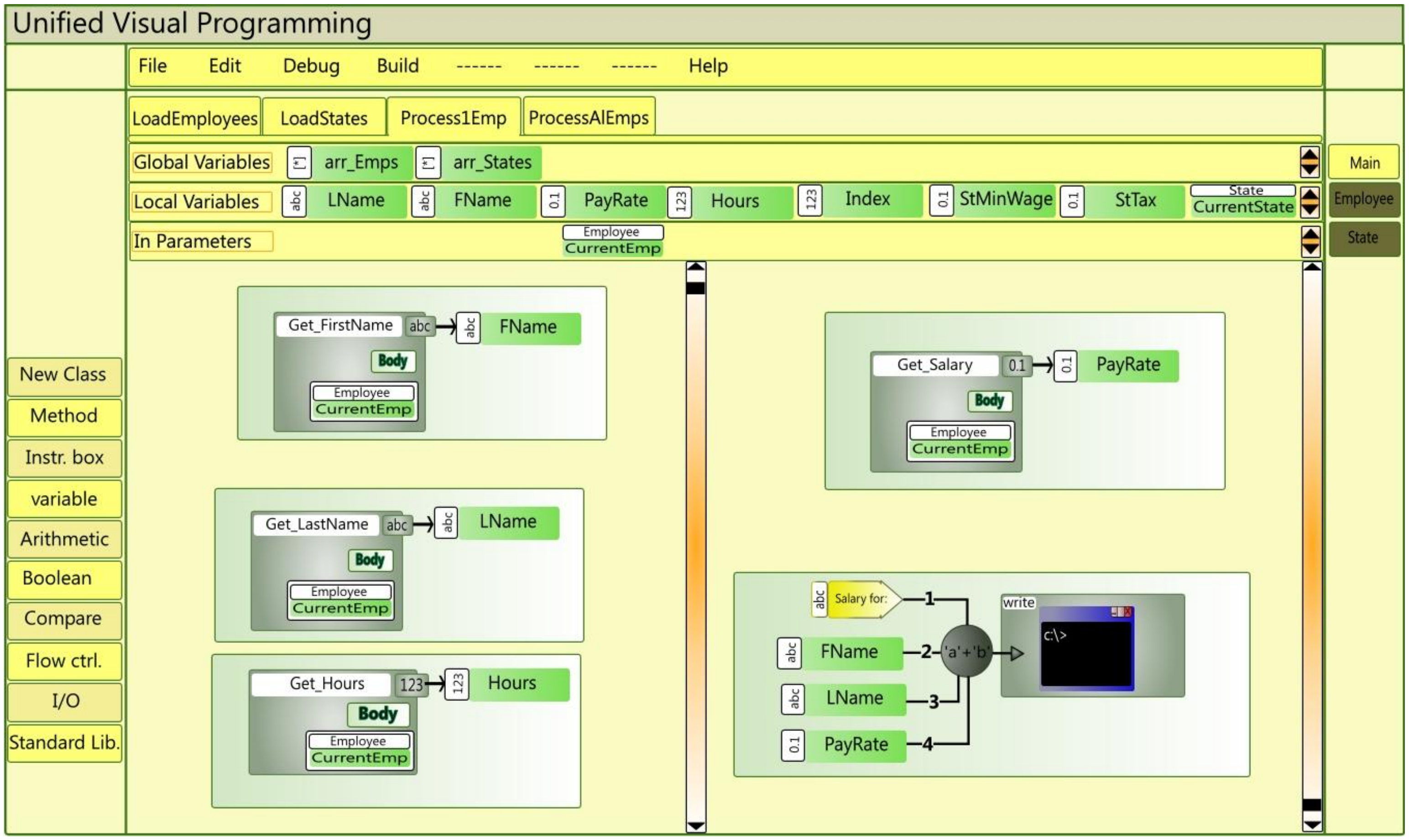
Help
LoadEmployees
LoadStates
Process1Emp
ProcessAlEmps
Global Variables
arr_Emps
arr_States
Local Variables
abc
LName
abc
FName
0.1
PayRate
123
Hours
123
Index
0.1
StMinWage
0.1
StTax
State
CurrentState
In Parameters
Employee
CurrentEmp
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.
Get_FirstName
abc
Body
Employee
CurrentEmp
FName
Get_LastName
LName
Get_Hours
Hours
if
StMinW
PayRate
then
else
StMinWage
PayRate
Compute_Salary
PayRate
Hours
StTax


Unified Visual Programming
File
Edit
Debug
Build

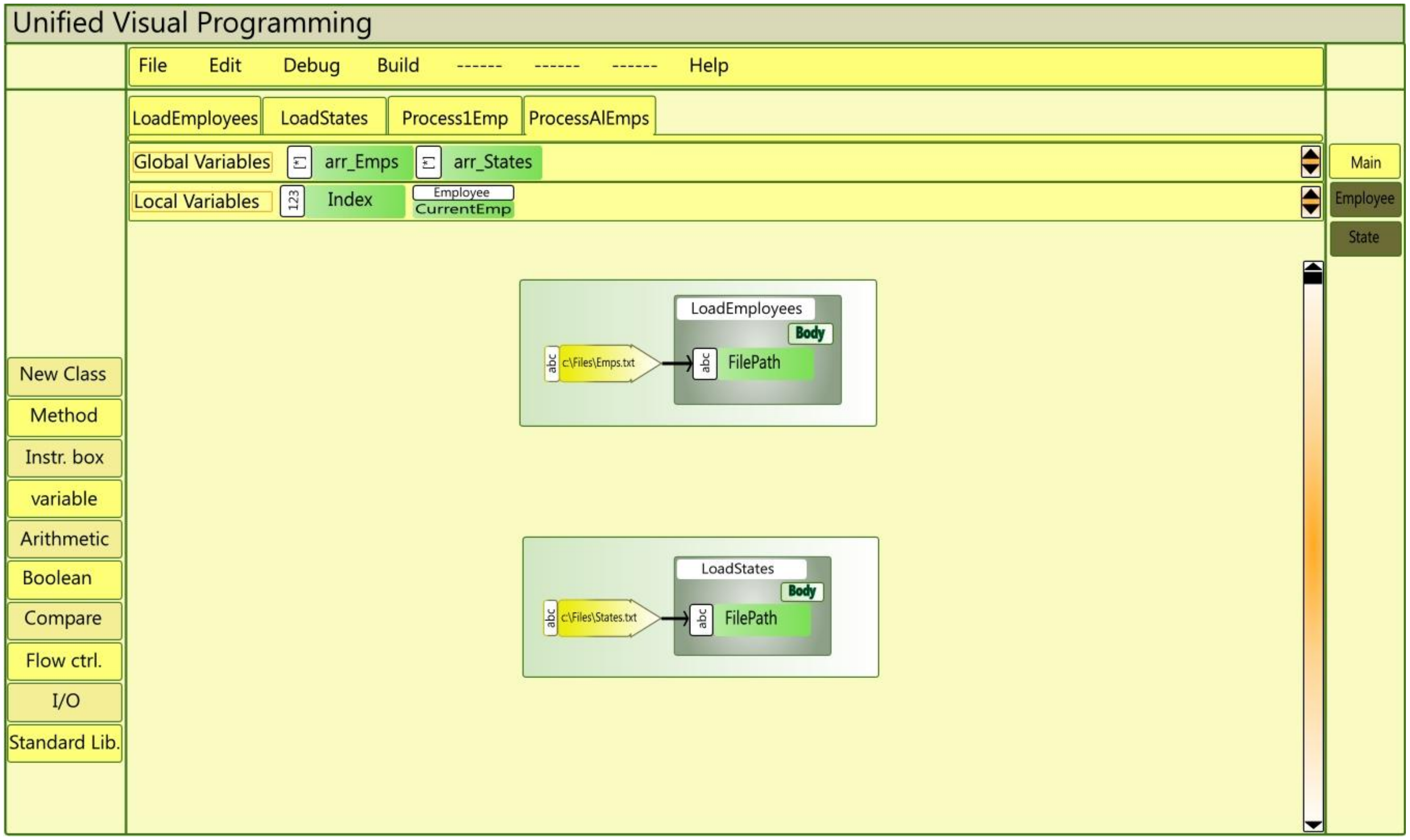
Help
LoadEmployees
LoadStates
Process1Emp
ProcessAlEmps
Global Variables
arr_Emps
arr_States
Local Variables
abc
LName
abc
FName
0.1
PayRate
123
Hours
123
Index
0.1
StMinWage
0.1
StTax
State
CurrentState
In Parameters
Employee
CurrentEmp
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.
Get_PayRate
0.1
PayRate
Body
Employee
CurrentEmp
Choose a State for:
1
FName
2
'a'+'b'
LName
3
1->ok,2->tx,3->NV,4->VA,5->NY
4
write
c:\>
read
c:\>
123
Index
if
StMinW
PayRate
then
else
StMinWage
PayRate
Compute_Salary
Body
PayRate
Hours
StTax
Employee
CurrentEmp


Unified Visual Programming
File
Edit
Debug
Build

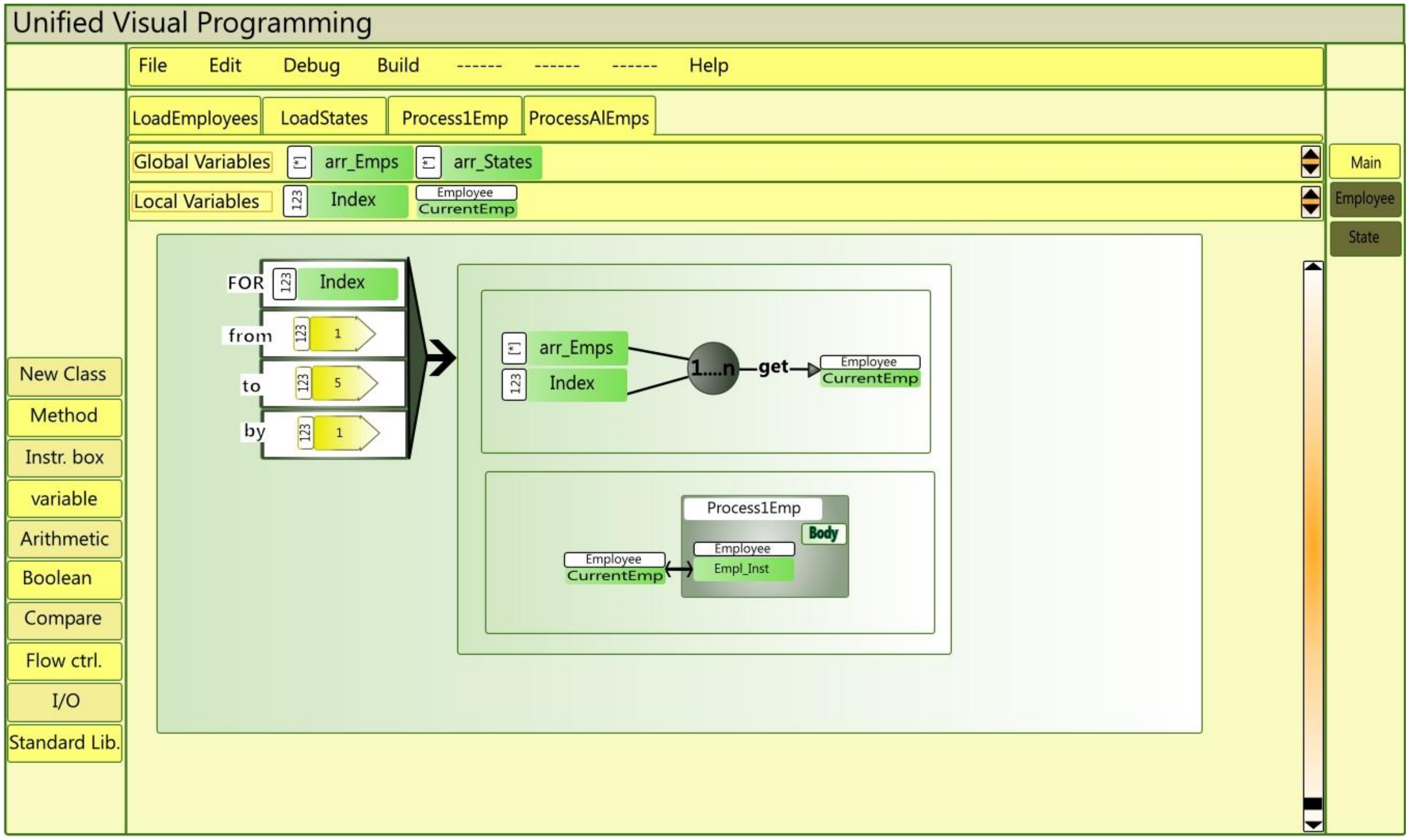
Help
LoadEmployees
LoadStates
Process1Emp
ProcessAlEmps
Global Variables
arr_Emps
arr_States
Local Variables
LName
FName
PayRate
Hours
Index
StMinW
StTax
State CurrentState
In Parameters
Employee CurrentEmp
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.
arr_States
Index
1....n
get
State CurrentState
Get_Tax
0.1
StTax
Body
State CurrentState
Get_MinWage
0.1
StMinW
Body
State CurrentState
if
StMinW
PayRate
then
else
StMinW
PayRate
Compute_Salary
Body
PayRate
Hours
StTax
Employee CurrentEmp


Unified Visual Programming
File
Edit
Debug
Build

Help
LoadEmployees
LoadStates
Process1Emp
ProcessAlEmps
Global Variables
arr_Emps
arr_States
Local Variables
abc LName
abc FName
0.1 PayRate
123 Hours
123 Index
0.1 StMinWage
0.1 StTax
State CurrentState
In Parameters
Employee CurrentEmp
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.
Get_FirstName abc → abc FName
Body
Employee CurrentEmp
Get_LastName abc → abc LName
Body
Employee CurrentEmp
Get_Hours 123 → 123 Hours
Body
Employee CurrentEmp
Get_Salary 0.1 → 0.1 PayRate
Body
Employee CurrentEmp
abc Salary for: 1
abc FName 2
abc LName 3
0.1 PayRate 4
'a'+'b'
write
c:\>


Unified Visual Programming
File
Edit
Debug
Build

Help
LoadEmployees
LoadStates
Process1Emp
ProcessAlEmps
Global Variables
arr_Emps
arr_States
Local Variables
123
Index
Employee
CurrentEmp
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.
LoadEmployees
Body
abc
c:\Files\Emps.txt
abc
FilePath
LoadStates
Body
abc
c:\Files\States.txt
abc
FilePath


Unified Visual Programming
File
Edit
Debug
Build

Help
LoadEmployees
LoadStates
Process1Emp
ProcessAlEmps
Global Variables
arr_Emps
arr_States
Local Variables
123
Index
Employee
CurrentEmp
Main
Employee
State
New Class
Method
Instr. box
variable
Arithmetic
Boolean
Compare
Flow ctrl.
I/O
Standard Lib.
FOR
Index
from
1
to
5
by
1
arr_Emps
Index
1....n
get
Employee
CurrentEmp
Process1Emp
Body
Employee
Empl_Inst
Employee
CurrentEmp

VITA

Seynabou Dieng

Candidate for the Degree of

Master of Science

Thesis: A PROPOSED UNIFIED VISUAL PROGRAMMING LANGUAGE

Major Field: Computer Science

Biographical: Born in Dakar, Sénégal on February 12th, 1978. The daughter of Khadijhatou Seck and the late Papa Amath Dieng.

Education: Graduated from Yalla Suur En high school, Dakar, Sénégal with a Baccalauréat with honors in 1998. Received a Technical Diploma in Informatics from Cheikh Anta Diop University –ESP, Dakar, Sénégal in 2000. Received a Bachelor of Science degree in Computer science from Oklahoma State University, Stillwater, Oklahoma in May 2005. Completed the requirements for the Master of Science in Computer Science at Oklahoma State University, Stillwater, Oklahoma in November, 2010.

Experience: Employed as a programmer analyst at MAERSK Sealand in Dakar, Sénégal from November 2000 to July 2001. Employed as a Database Analyst at Site Specific Technology in Stillwater, Oklahoma from July 2003 to August 2010. Employed as a System Analyst II at Sprint Nextel in Overland Park, Kansas from August 2010.

Professional Memberships: None.

Name: Seynabou Dieng Date of Degree: December, 2010

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: A PROPOSED UNIFIED VISUAL PROGRAMMING LANGUAGE

Pages in Study: 171 Candidate for the Degree of Master of Science
Major Field: Computer Science

Scope and Method of Study: this study aimed at contributing to the area of visual programming languages, by proposing the framework for a visual programming language (VPL) that is the unification of the best features of four existing VPLs. This unified VPL, or UVPL, is designed to better achieve scalability, which is lacking – in general – in current VPLs. The design of UVPL is based on the analysis of four existing VPLs. This study also aimed at providing a short survey of the four selected VPLs.

Findings and Conclusions: the design of UVPL succeeded in being a programming language that is general-purpose and object-oriented. However, UVPL as well as its development environment needs to be implemented for further, more concrete testing and comparison.

Dr. Blayne Mayfield