PERSPEX MACHINE X: SOFTWARE DEVELOPMENT

Perspex Machine X: Software Development

Sam Noble, Benjamin A. Thomas, James A.D.W. Anderson[*]
Computer Science, The University of Reading, England

Abstract

The Perspex Machine arose from the unification of computation with geometry. We now report significant redevelopment of both a partial C compiler that generates perspex programs and of a Graphical User Interface (GUI). The compiler is constructed with standard compiler-generator tools and produces both an explicit parse tree for C and an Abstract Syntax Tree (AST) that is better suited to code generation. The GUI uses a hash table and a simpler software architecture to achieve an order of magnitude speed up in processing and, consequently, an order of magnitude increase in the number of perspexes that can be manipulated in real time (now 6,000). Two perspex-machine simulators are provided, one using trans-floating-point arithmetic and the other using transrational arithmetic. All of the software described here is available on the world wide web.

The compiler generates code in the neural model of the perspex. At each branch point it uses a jumper to return control to the main fibre. This has the effect of pruning out an exponentially increasing number of branching fibres, thereby greatly increasing the efficiency of perspex programs as measured by the number of neurons required to implement an algorithm. The jumpers are placed at unit distance from the main fibre and form a geometrical structure analogous to a myelin sheath in a biological neuron. Both the perspex jumper-sheath and the biological myelin-sheath share the computational function of preventing cross-over of signals to neurons that lie close to an axon. This is an example of convergence driven by similar geometrical and computational constraints in perspex and biological neurons.

Keywords: perspex compiler, perspex GUI, perspex simulator, trans-floating-point arithmetic.

1. Introduction

The perspex machine, introduced in,[2] unifies geometry and computation so that all Turing computations can be performed geometrically, and all geometrical shapes and operations describe some, possibly super-Turing, computation. In particular, this means that Turing programs can be given in linguistic and pictorial form. Hence, compilers and Graphical User Interfaces (GUIs) have an equal role to play in developing Turing programs for the perspex machine. Whilst the super-Turing properties of a perspex machine are not accessible to a digital computer, it may be that the continuity of perspex operations[4] is important in practical programs that interact with an apparently analogue world. In this case the different psychological limits in a user's language processing and hand-eye coordination might mean that some operations can only be performed linguistically whilst others can only be performed graphically. For example, a user might easily describe a first-order predicate calculus program linguistically, but find it impossible to paint one with a virtual paint brush. Conversely, a user might easily weld two perspex programs together with a virtual arc-welder, but find it impossible to describe the weld linguistically. Hence, compilers and GUIs have a complementary role to play in enabling a user to express computations in the perspex machine and we should, therefore, expect that perspex compilers and perspex GUIs will co-evolve.

There is, to date, very little experience of developing perspex programs so it is not clear how to balance the roles of linguistic and graphical description of computations. All previous perspex compilers,[5] GUIs,[3,6] and simulators[3,6] were one-off research developments aimed at supporting the C programming language. This language was chosen as a target because it is a relatively simple language that exercises much of the imperative programming paradigm, and because there are good compiler-generator tools for C. In future, we expect to switch effort to developing a specialist language for the perspex machine, but, for the time being, the discipline of implementing a standard language is beneficial.

[*] Corresponding author. author@bookofparagon.com, http://www.bookofparagon.com
Computer Science, The University of Reading, Reading, Berkshire, England, RG6 6AY.

We now report the development of a compiler, GUI, and two simulators using robust software tools. One of the simulators uses trans-floating-point arithmetic and the other uses transrational arithmetic. Whilst the simulators and GUI can be used to manipulate useful perspex programs, the compiler is at too early a stage of development to be widely useful. Nonetheless, it provides a basis for developing future C to perspex compilers.

The reader may wish to note that all of the software described here is available on the world wide web.[11]

2. Compiler

In earlier work we reported a hand-built, top-down compiler, implemented in the AI language Pop11, that compiles a subset of the C programming language into perspexes.[5] This compiler generates code in terms of the projective perspex-instruction that is used to prove the Turing completeness of the perspex machine;[2] but this instruction is far more difficult to use than the general-linear instruction introduced with the universal perspex machine.[7] The new compiler uses the general-linear form of the instruction. This leads to simpler code generation and, in particular, a simpler method for performing multiplication and division. The new compiler compiles a subset of BS ISO 9899:1900 C. The compiler is generated by standard compiler-generator tools – Flex[12] and Bison.[13] It relies on the GCC compiler[14] to perform pre-processing of source code, such as file inclusion and macro expansion. The compiler reads a single source file and, optionally, provides four kinds of output. Firstly, it produces the printable form of a full syntax tree for C which is intended to support debugging of the compiler. Secondly, it produces a verbose description of the compilation, again as an aid to de-bugging the compiler. Thirdly, it produces the printable form of an Abstract Syntax Tree (AST) which is used to generate perspex code. And, finally, it generates perspex code in an interchange language. The interchange language is then loaded into a perspex-machine simulator where it is executed. The transrational simulator has a Graphical User Interface (GUI) that can be used to step through the execution and obtain diagnostic feedback.

The compiler preserves some aspects of the original compiler.[6] It continues to use the $t = 0$ hyperplane for builtin functions; it continues to use the $t = 1$ hyperplane for user-defined constants, variables, and functions; and it reserves the $t > 1$ hyperplanes for recursive function calls. The allocation of program elements to perspex space is handled by a perspex-space-manager within the compiler. The compiler is intended to be used on correct C code that strictly conforms to the BS ISO 9899:1900 C standard. It does report syntax errors, but error handling is not so sophisticated as in a typical C compiler. The compiler is intended as a research vehicle to explore perspex implementations of standard programming languages. Later, it would be sensible to consider programming languages specifically designed for the perspex machine, but this must await a fuller understanding of the practical properties of the machine. The compiler, GUI, and simulators are available on the world wide web.[11]

2.1 Scope of the Compiler

The compiler can read only a single source file, because the perspex-space-manager does not have the facility to relocate code. However, the source file may be arbitrarily long and may be pre-processed to give effect to *#include,* macro expansion, and inlining; but the machine-code interface is not supported. The perspex to C compiler can handle only C source, not machine code. The compiler can perform full syntax checking, though the checking is not so sophisticated as in a standard compiler. It can generate code only for a very limited subset of C, but it does give warnings of any unsupported language features that are used in the source code.

Code is generated to support the following language features. Declarations of numerical constants and variables, with or without initialisers, are supported, but all such objects are cast to trans-floating-point numbers. The arithmetical operations +, -, *, /, are supported, as is assignment. The postfix incremental operators ++ and -- are supported, but the corresponding prefix operators are not supported. Function declaration, with or without parameter lists is supported, as is ellipsis in function prototypes. The control statement *if–else* is supported, as are *while* loops. Compound statements of all supported declarations and statements are supported. This is a slight reduction in language support as compared to the hand-written compiler, but the present compiler has the advantage of full syntax checking and much easier development, because it uses standard compiler-generator tools (Flex[12] and Bison,[13] variants of Lex and Yacc[9]). The compiler is bundled with a C^{++} library that supports trans-floating-point arithmetic and a trans-floating-point simulation of the perspex machine.[11]

2.2 Perspex-Space Manager

The perspex-space-manager is responsible for associating user code and data with locations in perspex space. It obeys the convention of using the $t = 0$ hyperplane for builtin functions; the $t = 1$ hyperplane for user-defined constants, variables, and functions; and it preserves the $t > 1$ hyperplanes recursive structures, but recursion is not supported by the compiler.

The builtin functions are arranged spatially with respect to the origin of the $t = 0$ hyperplane. Transput variables which perform both input and output, or just internal transfers of information, are laid along the x-axis. Input variables are laid along the y-axis. And output variables are laid out along the z-axis.

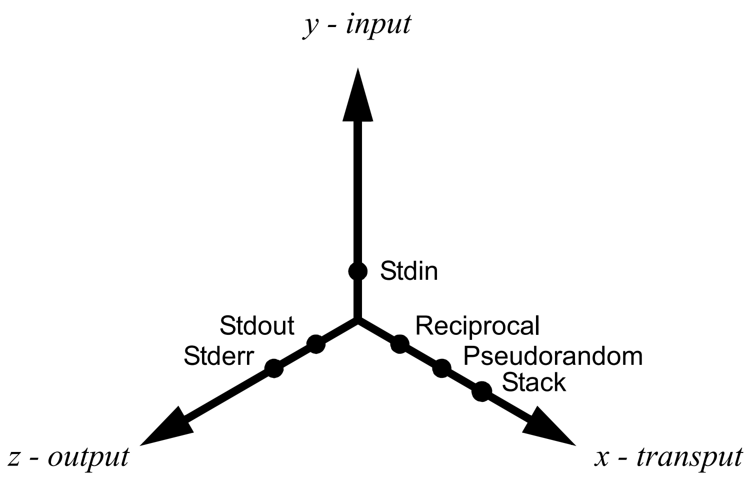

Figure 1: Spatial arrangement of input, output, and transput.

The builtin functions are implemented in terms of active perspexes that perform some action on reading and/or writing. These include: active perspexes that read and/or write one character at a time to/from the C streams *stdin, stdout, stderr;* an active perspex that accepts a perspex on write, but returns a perspex with the reciprocal of each element on read; a perspex that accepts any perspex on write, but returns a perspex with pseudorandom elements on read; and, finally, a perspex that operates as a stack manager. If an attempt is made to read from *stdout* or *stderr* it returns nullity, by analogy with the C value *Null.* If an attempt is made to write to *stdin* it over-writes the value there. This is a design fault, it ought to push the value back onto the input stream by analogy with the C function *ungetc.*

Some of the builtin functions are intended for research unrelated to the development of the perspex compiler, GUI, and simulator. In future, it might be better to introduce builtins via libraries that instantiate them in the $t = 0$ hyperplane. Builtins must be supplied to provide physical side effects, such as input, output, and power management. There may also be practical advantage in supplying them to perform operations that are costly to support directly in the perspex machine's operations, such as finding the reciprocal. But supplying abstract data types, such as a stack, as a builtin is purely a matter of convenience that could have been done otherwise. For example, a stack manager could be defined in perspexes and could be stored in the $t = 0$ hyperplane to be copied into any location where it is needed. In other words, the $t = 0$ hyperplane could be a repository of "genetic" machines that are transported into the body of perspex space to do their work at the location and time they are needed. Once they have fulfilled their role these "phenotypic" machines could be deleted.

2.3 Perspex Code Templates

The compiler generates code in the neural model of the perspex. At each branch point it uses a jumper[5,7] to return control to the main fibre. This has the effect of pruning out an exponentially increasing number of branching fibres, thereby greatly increasing the efficiency of perspex programs as measured by the number of neurons required to implement an algorithm. The jumpers are placed at unit distance from the main fibre and form a geometrical structure analogous to a myelin sheath in a biological neuron. Myelin sheaths increase the speed of the transmission of electrical spikes along a biological axon and electrically insulate axons, thereby reducing the cross-over of information from one neuron to another. The jumpers in perspex neurons have no effect on the speed of transmission of a signal, but they do prevent cross-over of signals by pruning out fibres. Thus, the jumper sheath in perspex neurons and the myelin sheath in biological neurons have the common function of preventing cross-over of signals. That these two structures have some commonality in function is to be expected, because both perspex neurons and biological neurons are geometrical structures that effect computation.

2.4 Abstract Syntax Tree

The Abstract Syntax Tree (AST) is an n-branch tree where each node is of type *ASTNode.* The AST only stores information that is necessary to traverse the tree itself and to reconstruct the original meaning of the source code for the purposes of implementing it in the neural model of the perspex machine. The AST does not preserve all of the distinctions made in the source code, indeed, its value is that it removes unnecessary distinctions, thereby simplifying code generation in the perspex neural model. Each node stores: a count of any sub-branches; the target location for the node in perspex space; a descriptive textual label for the node, used for debugging and building textual version of the AST; a name if the AST represents an identifier or function; the node's type, stored as an *int;* a *union* containing a pointer to a *struct* holding node-specific data; and an array of pointers to sub branches, if any. The tree holds data for C's: declarations; function declarations; literals; identifiers; expressions; jump statements; iteration statements; selection statements; and parameter-type lists.

The AST is built bottom up as the source code is parsed. A terminal statement in the grammar either creates an AST node with some data and passes it back to its parent statement, or else creates a 'zombie' node, used purely for passing information back to its parent. Depending on the parent, for each AST node passed to it, it either references the nodes in its branches-list directly, or uses information from the nodes to construct one or more AST nodes which it passes back.

A YACC grammar for the AST is included in the Appendix. The full source code for the compiler is available on the world wide web.[11]

2.5 Perspex Interchange-Language

The perspex interchange-language is very simple. The language is written in textual form as a, possibly empty, sequence of data, written between matching brackets. A description of type *"X"* is bracketed by *"StartX"* and *"EndX"*. Brackets are recursive. An unrecognised *"X"* is ignored so recursively nested comments may be introduced by any unrecognised *"X"*. However, the brackets *"StartComment"* and *"EndComment"* are provided to allow the explicit introduction of comments as uninterpreted text. So far, very few brackets have been defined. Data sequences are written in the printable form used by C^{++}.

- Numbers are written as C^{++} integer or floating-point numbers, or as transrational ratios of an integer numerator and denominator separated by the symbol *"/"*.

- *StartPerspex / EndPerspex* brackets sixteen numbers in row order that make up the elements of a perspex represented as a 4×4 matrix.

- *StartPerspexPosition / EndPerspexPosition* brackets four numbers in row order that make up a position vector in perspex space. Pairs of *Perspex* and *PerspexPosition* brackets define a perspex and its location. This is a design fault. It would be better to bracket the *Perspex* and *PerspexPosition* brackets together in some superordinate bracket.

- *StartEntryFrom / EndEntryFrom* brackets four numbers in row order that specify a position vector in perspex space that is the entry point of the perspex machine. That is, the machine is started at this point. All text after the *EndEntryFrom* bracket is ignored. This is a design fault.

- *StartCommnet / EndComment* brackets uninterpreted text.

The following example shows the specification of a single perspex at location $(0, 0, 0, 0)$ with the perspex machine starting execution at this point.

```
StartPerspex
    1.0 2 3/1 4/5
    1/0 -1/0 0/0 0
    1 2 3 4
    1 2 3 4
EndPerspex
StartPerspexPosition 0 0 0 0 EndPerspexPosition
StartEntryFrom 0 0 0 0 EndEntryFrom
```

This language is overly primitive and is under specified, relying on the textual form of numbers in C^{++}. In future it might be helpful to define brackets that pass compiler pragmas and GUI commands both ways between the GUI and the compiler and to adopt a more formal definition of the language. The current definition is, however, sufficient to support research on compiler design and GUIs for the perspex machine. It is expected that the interchange language will co-evolve with the compiler and GUI.

3. Graphical User Interface

The original graphical user interface was implemented in C^{++} and made heavy use of OpenGL and the GLUT library to handle visual display and user interaction.[6] The GUI interfaced to a simulation of the perspex machine implemented in C^{++} that, in turn, made heavy use of the LEDA library to perform rational arithmetic.[6] The simulation accessed 4D perspex space in linear time. This was sufficient to operate on 600 perspexes in real time as demonstrated in an implementation of Dijkstra's solution to the travelling salesman problem. This simulation supported the saving of images to file, but did not support the input or output of the perspex interchange language to file. Therefore, all data had to be hand coded in the source file.

The simulator continues to use LEDA to support rational arithmetic. It uses a hash table to access perspex space in near constant time. It supports all of the builtin procedures described in section 2.2. In particular, it supports the input and output of perspexes in the perspex interchange language, section 2.5. The new GUI runs an order of magnitude faster than the old one and can now operate on 6,000 perspexes in real time. The source code for the GUI is available at.[11]

The GUI is bundled with a library that overloads many of the arithmetical operations in C^{++} so that they operate on transrational numbers. It was found to be useful to extend the *sign* operation so that it returns the infinities directly. In this way, it is possible to use the *sign* operation in a switch statement to operate individually on each of the strictly transrational numbers and to operate of the sign of the rational numbers. This, programmatic, definition of the sign of a number is given next.

$$\operatorname{sgn}(a) = \begin{cases} \infty : a = \infty \\ 1 : 0 < a < \infty \\ 0 : a = 0 \\ \Phi : a = \Phi \\ -1 : -\infty < a < 0 \\ -\infty : a = -\infty \end{cases} \qquad \text{(Eqn 1)}$$

A library that provides the transrational trigonometric functions of the half-tangent, defined in,[1] is also provided. This library uses nullity and positive infinity, not negative infinity, following an earlier specification of the transrational numbers. This library does *not* provide the transreal trigonometric functions defined in.[8]

The GUI provides functionality similar to the previous version,[6] but also adds *mouse look*. This is an alteration of the virtual camera's yaw and pitch with, respectively, horizontal and vertical motions of the mouse. This look and feel is provided by some computer games and is copied here as an intuitive mode of interaction.

4. Conclusion

We report the development of a compiler, GUI, and two simulators using robust software tools. One of the simulators uses trans-floating-point arithmetic and the other uses transrational arithmetic. Whilst the simulators and GUI can be used to manipulate useful perspex programs, the compiler is at too early a stage of development to be widely useful. Nonetheless, it provides a basis for developing future C to perspex compilers. The compiler uses the general-linear form of the instruction. This leads to simpler code generation and, in particular, a simpler method for performing multiplication and division. Code generation exploits an Abstract Syntax Tree (AST) which compresses the semantics of the C source so that it retains just enough information to translate the source to the neural model of the perspex machine. For example, the AST discards a great deal of type information because, for example, all C numerical types are represented by the single numerical type supplied by a perspex neuron. The AST is used in conjunction with a perspex space manager that is responsible for laying out code and built in resources so that no clashes occur. The space manager lays out all I/O in the $t = 0$ hyperplane with transput variables laid out along the x-axis, input variables laid out along the y-axis, and output variable along the z-axis. Code is laid out in the $t > 0$ hyperplanes. It is laid out in neural model with jumpers that force control back onto a single fibre. The jumpers operate like myelin sheaths around a biological neuron, they keep signals within the neuron and prevent cross-over to nearby neurons. This simplifies the task of the space manager. It need only lay out a single filament of neurons corresponding to a single thread of processing in a serial processor.

A combined GUI and perspex-simulator is also reported. It allows the user to interact with a perspex machine simulator operating on the neural model. Various software enhancements deliver a ten-fold increase in speed over previous GUI/simulators. The GUI can now display programs with 6,000 neurons operating in real time. The implementation of *mouse look* has improved the usability of the GUI.

5. Appendix

The following YACC grammar defines the Abstract Syntax Tree used in the compiler. The full source is available on the world wide web.[11]

5.1 Expressions

```
identifier:
    IDENTIFIER (Create new identifier)

primary_expression:
    identifier (Pass back Identifier node)
    INTEGER (Identifier, type = INTEGER)
    CHARACTER (Identifier, type = CHARACTER)
    FLOATING (Identifier, type = FLOATING)
    STRING (Identifier, type = STRING)
    `(` expression `)’ (Pass back expression node)

postfix_expression:
    primary_expression (Pass back single node)
    postfix_expression '['expression']' (not implemented)
    postfix_expression '('argument_expression_list')'
        (not implemented)
    postfix_expression PLUSPLUS (Create new node, exp as branch)
    postfix_expression MINUSMINUS (Create new node, exp as branch)

argument_expression_list:
    argument_expression_list ',' assignment_expression
        (Pass back both expression nodes)
```

```
unary_expression:
    postfix_expression

cast_expression:
    unary_expression

multiplicative_expression:
    cast_expression (Pass back node)
    multiplicative_expression '*' cast_expression
        (Create new expression node, nodes as branches)
    multiplicative_expression '/' cast_expression
        (Create new expression node, nodes as branches)
    multiplicative_expression '%' cast_expression
        (Create new expression node, nodes as branches)

additive_expression:
    multiplicative_expression (Pass back single node)
    additive_expression '+' multiplicative_expression
        (Create new exp node, exp nodes as branches)
    additive_expression '-' multiplicative_expression
        (Create new exp node, exp nodes as branches)

shift_expression:
    additive_expression
relational_expression:
    shift_expression (Pass back single node)
    relational_expression '<' shift_expression
    relational_expression '>' shift_expression
    relational_expression LTEQ shift_expression
    relational_expression GTEQ shift_expression

equality_expression:
    relational_expression
    equality_expression EQ relational_expression
    equality_expression NOTEQ relational_expression

and_expression:
    equality_expression

exclusive_or_expression:
    and_expression

inclusive_or_expression:
    exclusive_or_expression

logical_and_expression:
    inclusive_or_expression

logical_or_expression:
    logical_and_expression
```

```
conditional_expression:
    logical_or_expression

assignment_expression:
    conditional_expression

assignment_operator:
    '=' (Create new terminal node, type = assignment)

expression:
    assignment_expression
    expression ',' assignment_expression

constant_expression:
    conditional_expression

declaration:
    declaration_specifiers init_declarator_list ';'
        (Return node = $1, with initializer and
        initializeTo set to those in $2)
    declaration_specifiers ';'

declaration_specifiers:
    storage_class_specifier declaration_specifiers
    storage_class_specifier
    type_specifier declaration_specifiers
    type_specifier
    type_qualifier declaration_specifiers
        (Return node = $2, with type qualifier set to $1)
    type_qualifier

init_declarator_list:
    init_declarator

init_declarator:
    declarator (Return node = $1 with initializeTo = AST_ZOM)
    declarator '=' initializer
        (Return node $1 with initializeTo and initializer
        set tp those in $3)

type_specifier:
    YYVOID
    YYCHAR
    YYSHORT
    YYINT
    YYLONG
    YYFLOAT
    YYDOUBLE
    YYSIGNED
    YYUNSIGNED
        (For YYVOID to YYUNSIGNED, a new terminal, declaration
        node is created, with its type equal to the approptiate token)
    struct_or_union_specifier
    enum_specifier
    TYPEDEF_NAME

type_qualifier:
    YYCONST
    YYVOLATILE
        (For YYCONST and YYVOLATILE, a new terminal, declaration
        node is created, with its type equal to the approptiate
        token)

declarator:
    direct_declarator

direct_declarator:
    identifier (Create new terminal declaration node, where the
    name is the that of $1)
        '(' declarator ')'
    direct_declarator '(' parameter_type_list ')'
        (Create new function declaration node. The
        function's type is given by $1, the presence of
        ellipsis, parameter count and parameters is
        supplied by $3)
    direct_declarator '(' ')'
    type_qualifier_list:
    type_qualifier

parameter_type_list:
    parameter_list
    parameter_list ',' ELLIPSIS

parameter_list:
    parameter_declaration
        (Create new parameter list node, with $1 added to
        the list)
    parameter_list ',' parameter_declaration
        (Add $3 to list in $1)

parameter_declaration:
    declaration_specifiers declarator
        (Return $1 with name taken from $2)

identifier_list:
    identifier

abstract_declarator:
     pointer

initializer:
     assignment_expression
```

```
initializer_list:
    initializer
```

5.2 Statements

```
statement:
    labeled_statement
    compound_statement
    expression_statement
    selection_statement
    iteration_statement
    jump_statement

compound_statement:
    '{' '}' (Create new (empty) function body node with)
    '{' statement_list '}'
        (Create a new function body with statements only)
    '{' declaration_list '}'
        (Create a new function body with local declarations only)
    '{' declaration_list statement_list '}'
        (Create a new function body with declarations and
        statements)

declaration_list:
    declaration
    declaration_list declaration

statement_list:
    statement
    statement_list statement

expression_statement:
    ';' (Return NULL)
    expression ';'

selection_statement:
    YYIF '(' expression ')' statement
        (Create new IF selection node with branches pointing to
    $3 and $5)
    YYIF '(' expression ')' statement YYELSE statement
        (Create new IF selection node with branches pointing to
    $3, $5 and $7)

iteration_statement:
    YYWHILE '(' expression ')' statement
        (Create new ITERATION statement with branches pointing to
        $3 and $5)
    YYDO statement YYWHILE '(' expression ')' ';'
        (Create new ITERATION statement with branches pointing to
        $2 and $5)
    YYFOR '(' ';' ';' ')' statement
        (Create empty ITERATION statement of type YYFOR)
    YYFOR '(' expression ';' ';' ')' statement
```

```
        (Create ITERATION statement of type YYFOR with branches
        pointing to $3 and $7)
    YYFOR '(' ';' expression ';' ')' statement
        (Create ITERATION statement of type YYFOR with branches
        pointing to $4 and $7)
    YYFOR '(' expression ';' expression ';' ')' statement
        (Create ITERATION statement of type YYFOR with branches
        pointing to $3, $4 and $8)
    YYFOR '(' ';' ';' expression ')' statement
        (Create ITERATION statement of type YYFOR with branches
        pointing to $5 and $7)
    YYFOR '(' expression ';' ';' expression ')' statement
        (Create ITERATION statement of type YYFOR with branches
        pointing to $3, $6 and $8)
    YYFOR '(' ';' expression ';' expression ')' statement
        (Create ITERATION statement of type YYFOR with branches
        pointing to $4, $6 and $7)
    YYFOR '(' expression ';' expression ';' expression ')'
statement
    (Create ITERATION statement of type YYFOR with branches
    pointing to $3, $5 and $7)

jump_statement:
    YYGOTO identifier ';'
YYCONTINUE ';'
YYBREAK ';'
YYRETURN ';'
YYRETURN expression ';'
    (Create new JUMP statement with $2 as a branch)
```

5.3 External Definitions

```
start:
    translation_unit
        (Set the AST's root node to $1)

translation_unit:
    external_declaration
    translation_unit external_declaration
        (Create new TRANSLATION UNIT node with $1 and $2 as
        branches)

external_declaration:
    function_definition
    declaration

function_definition:
    declaration_specifiers declarator compound_statement
    declarator declaration_list compound_statement
    declarator compound_statement
```

6. References

1 J. A. D. W. Anderson "Exact Numerical Computation of the Rational General Linear Transformations" in Vision Geometry XI, Longin Jan Lateki, David M. Mount, Angela Y. Wu, Editors, Proceedings of SPIE Vol. 4794, 22-28 (2002).

2 J.A.D.W. Anderson, "Perspex Machine" in *Vision Geometry XI* Longin Jan Latecki, David M. Mount, Angela Y. Wu, Editors, Proceedings of the SPIE Vol. 4794, 10-21 (2002).

3 J.A.D.W. Anderson, "Perspex Machine II: Visualisation" in *Vision Geometry XIII* Longin Jan Latecki, David M. Mount, Angela Y. Wu, Editors, Proceedings of the SPIE Vol. 5675, 100-111 (2005).

4 J. A. D. W. Anderson "Perspex Machine III: Continuity Over the Turing Operations" in Vision Geometry XIII, Longin Jan Lateki, David M. Mount, Angela Y. Wu, Editors, Proceedings of SPIE Vol. 5675, 112-123 (2005).

5 M. P. Spanner & J. A. D. W. Anderson "Perspex Machine V: Compilation of C Programs" in Vision Geometry XIV, Longin Jan Lateki, David M. Mount, Angela Y. Wu, Editors, Proceedings of SPIE Vol. 6066 (2006).

6 C. J. A. Kershaw & J. A. D. W. Anderson "Perspex Machine VI: A Graphical User Interface to the Perspex Machine" in Vision Geometry XIV, Longin Jan Lateki, David M. Mount, Angela Y. Wu, Editors, Proceedings of SPIE Vol. 6066 (2006).

7 J.A.D.W. Anderson, "Perspex Machine VII: The Universal Perspex Machine" in *Vision Geometry XIV* Longin Jan Latecki, David M. Mount, Angela Y. Wu, Editors, Proceedings of the SPIE Vol. 6066 (2006).

8 J.A.D.W. Anderson, "Perspex Machine IX: Transreal Analysis" *Vision Geometry XV* Longin Jan Latecki, David M. Mount, Angela Y. Wu, Editors, Proceedings of the SPIE, this volume, (2007).

9 J.R. Levine, T. Mason, D. Brown *Lex and Yacc,* 2nd edition, O'Reilly (1992).

10 *ISO/IEC 9899 Programming Languages - C,* British Standards Institute, London, England (1990).

11 At the time of the conference source code will be available at: http://www.bookofparagon.com/Pages/Downloads.htm

12 Flex: http://gnuwin32.sourceforge.net/flex.htm (Last accessed 27 April 2006).

13 Bison: http://www.gnu.org/software/bison/bison.html (Last accessed 1 June 2006).

14 GCC: http://gcc.gnu.org/ (Last accessed 2 June 2006).