CIP: The Programming Language

A general purpose programming language with SIMD capabilities

D406F12

Aalborg University

June, 2012

Author	E-mail
Christoffer Moesgaard	cmoesg10@student.aau.dk
Daniel Hillerström	dhille10@student.aau.dk
Daniel Rune Jensen	drje10@student.aau.dk
Eric Vignola Ruder	eruder10@student.aau.dk
Mathias Ruggaard Pedersen	mrpe10@student.aau.dk
Kimmo Andersen	kander10@student.aau.dk
Søren Kejser Jensen	skje10@student.aau.dk

Table of Contents

About the project

- CIP Computing in Parallel
- An imperative general purpose programming language
- SIMD capabilities
- Inspired by ZPL

Problem Statement

- SIMD has been proven faster under the right circumstances
- Programmer should focus on problem at hand
- Ideally, SIMD has to be faster than SISD

How may we design and implement a programming language that utilises the concepts of SIMD?

Problem Statement

- How can such a language be formalised?
- Which data types should be established?
- How can we design constructs that are easy for the compiler to generate SIMD instructions from?
- How can we encourage the programmer to use these constructs?
- How does the performance of our compiler compare with already established compilers?

Table of Contents

Design philosophy

- Data parallelism without encumbering the programmer
- Focus on being productive
- C-like syntax with enhanced readability
- Arrays as first-class citizens
- Procedures can not return arrays

Contextual rules

- int8, int16, int32, int64, float32, float64
- Static type checking and type binding
- No implicit or explicit casting between the two primitives
- Implicit casting from smaller primitive to larger primitive
- Static scope rules

Control flow

While loop

```
while (a < 10)
  ...
end
```

If-else statement

```
if (b < 10)
  ...
else
  ...
end
```

Procedures

Declaration

```
procedure void proc(float a, int[] B)
  ...
end
```

Procedure call

```
foo(2.0, C);
```

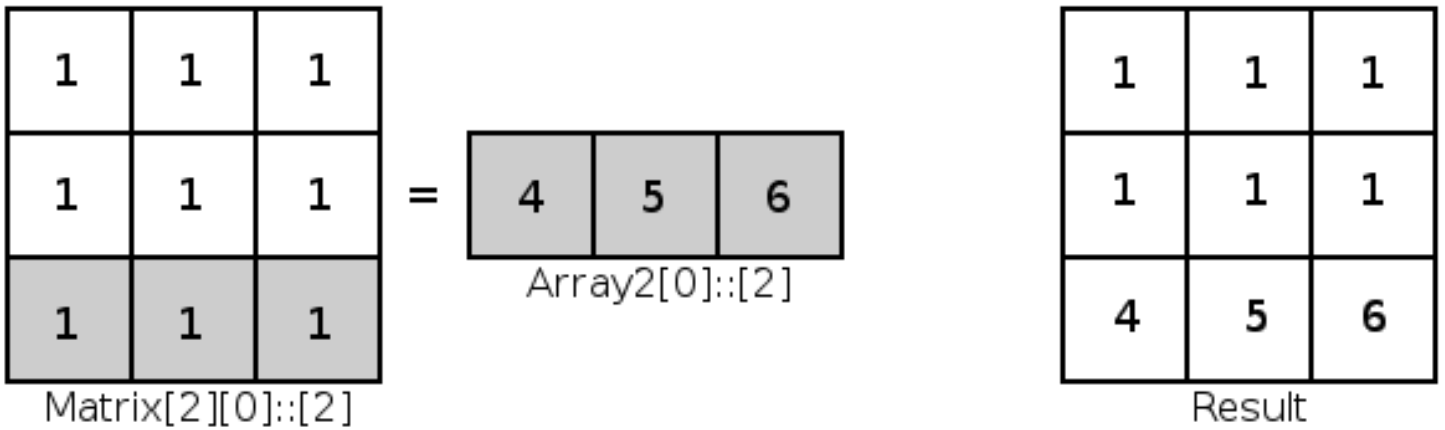
Arrays

Declaration

```
int8 Array2[3] = {4,5,6};
int8 Vector[3][1] = {{7},{7},{7}};
int8 Matrix[3][3] = {{1,1,1},{1,1,1},{1,1,1}};
```

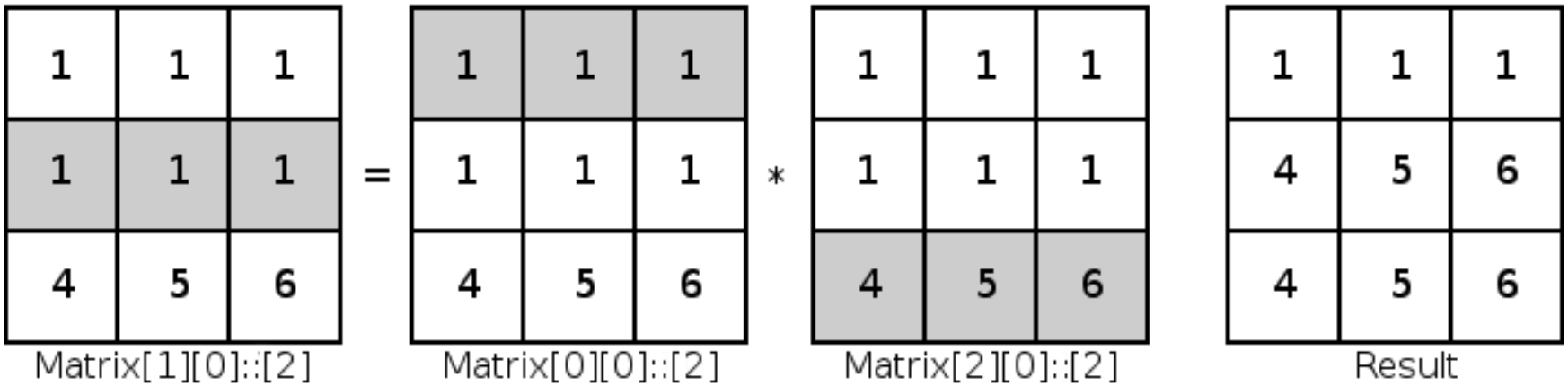
Array range operator

```
Matrix[2][0]::[2] = Array2[0]::[2];
```



Array range multiplication

```
Matrix[1][0]::[2] = Matrix[0][0]::[2] * Matrix[2][0]::[2];
```



Array operators

Binary operators

```
int32 Array1[4] = {1,2,3,4}, Array2[4] = {5,6,7,8};

Array1[0]::[3] = Array1[0]::[3] * Array2[0]::[3];
Array1[0]::[3] = Array1[0]::[3] > Array2[0]::[3];
```

Unary operators

```
int8 Array1[3] = {1,2,3};

Array1[0]::[2] = !Array1[0]::[2];
Array1[0]::[2] = -Array1[0]::[2];
```

Table of Contents

Our visitors

- Traversing the AST.
- Visitor design pattern.
- Reflective visitor.
- Twelve visitors:
  - Pretty print.
  - Error and warning checks.
  - Decorating the AST.
  - Code generation.

Declaration visitor

- Two visitors for analysing variable declaration and use.
- Declaration visitor:
  - Goes through all declaration nodes.
  - If the variable is not in the symbol table, add to symbol table.
  - If it is, issue an "already declared" error.
- CheckIdentifier visitor:
  - Goes through all variable nodes.
  - If variable is in symbol table, continue as normal.
  - If not, issue a "variable not declared" error.

Type check visitor

- Three visitors take care of type checking.
- TypeCheck:
  - Checks type correctness.
  - Throws TypeError if types are incompatible.
    - Types do not match.
    - Modulo used with float.
    - Left-hand side is procedure.
- GeneralizeType:
  - Generalises types after types have been verified as correct.
  - int8 + int32 $\rightarrow$ int32 + int32.
- TypeAdder:
  - Adds type information to relevant nodes.

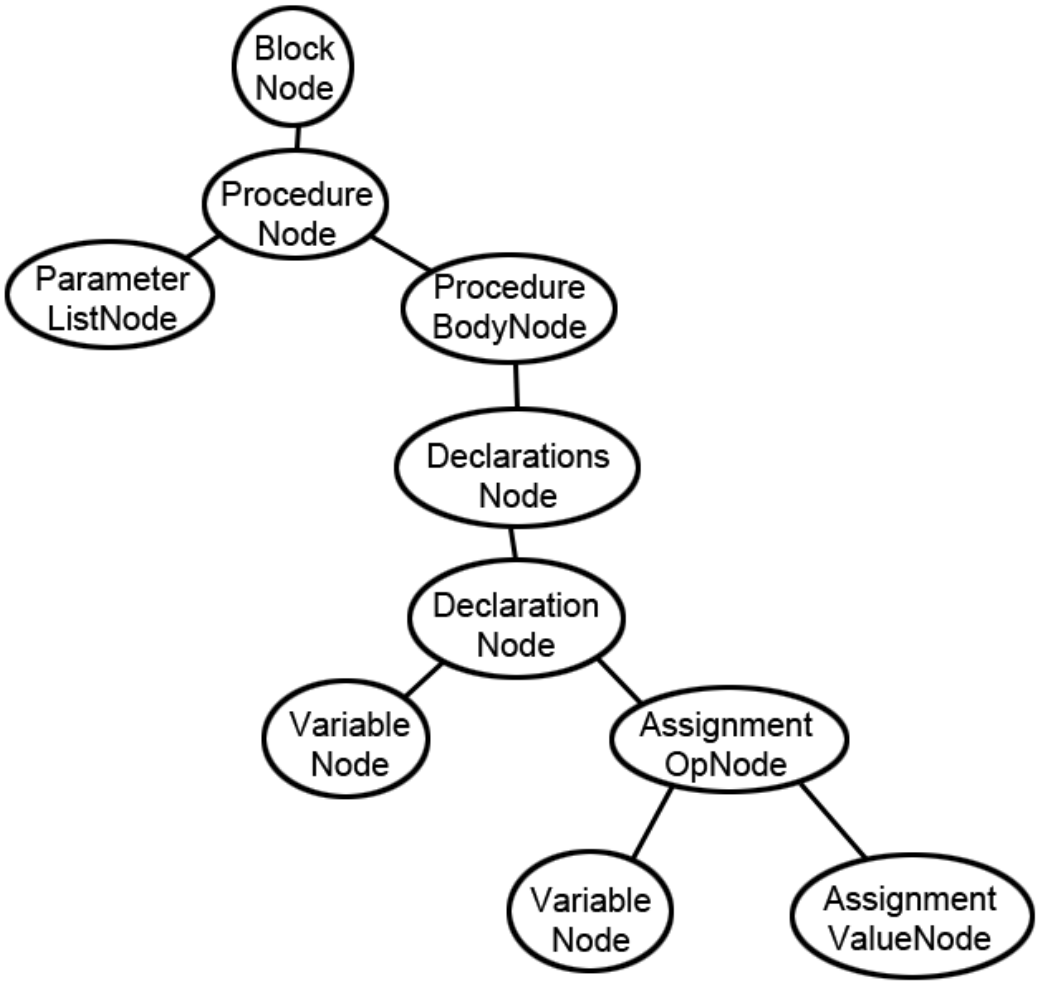
An example: Declaration visitor

Declaration example

```
procedure void proc()
  int a = 5;
end
```

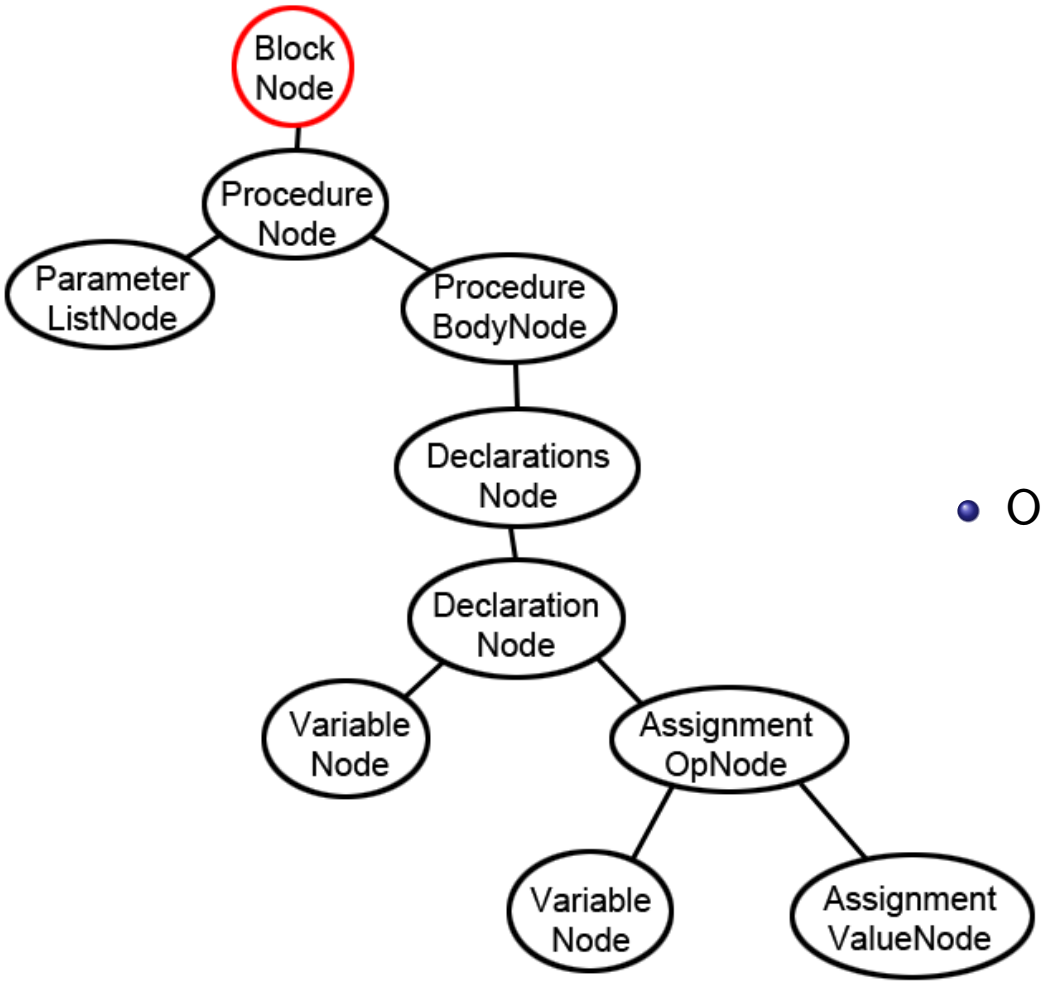
An example: Declaration visitor

Abstract Syntax Tree



An example: Declaration visitor

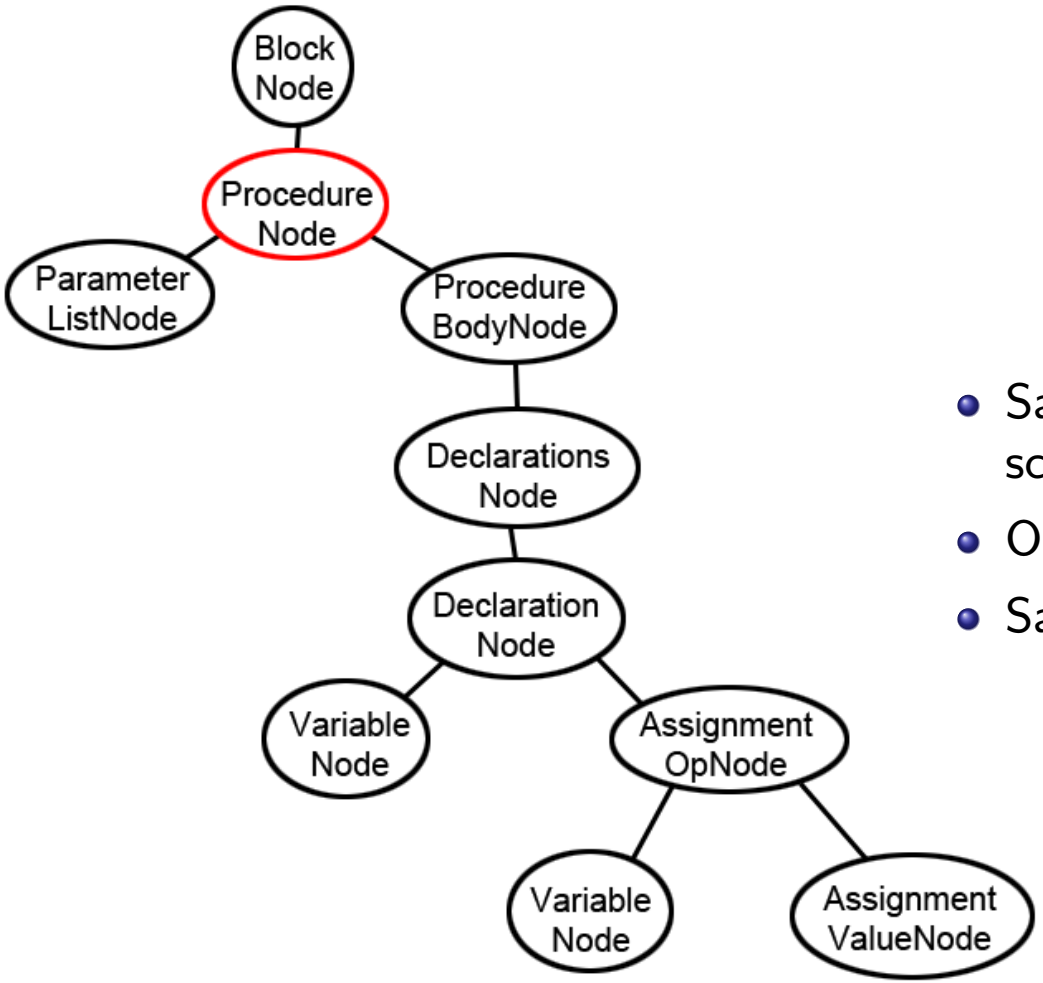
Abstract Syntax Tree



- Open new scope (global).

An example: Declaration visitor

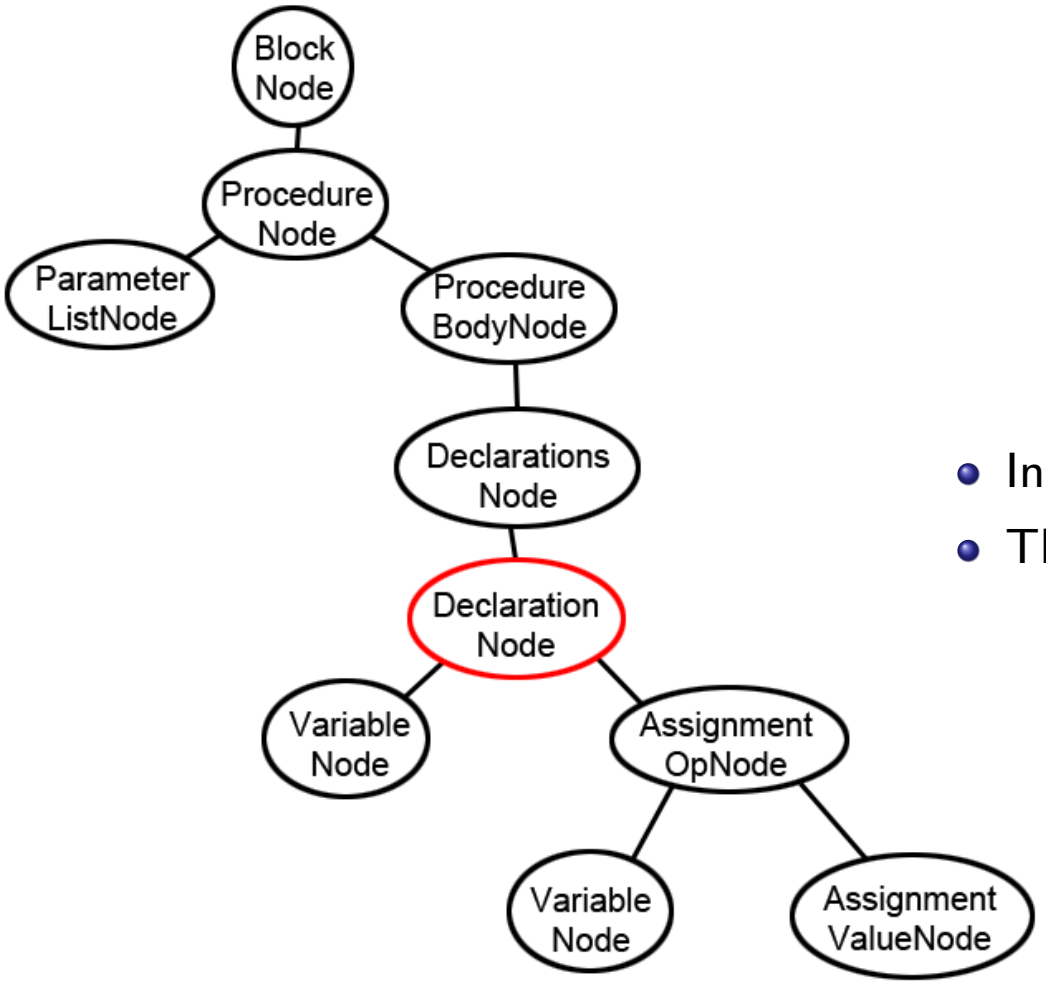
Abstract Syntax Tree



- Save procedure in current scope.
- Open new scope (proc).
- Save parameters in new scope.

An example: Declaration visitor

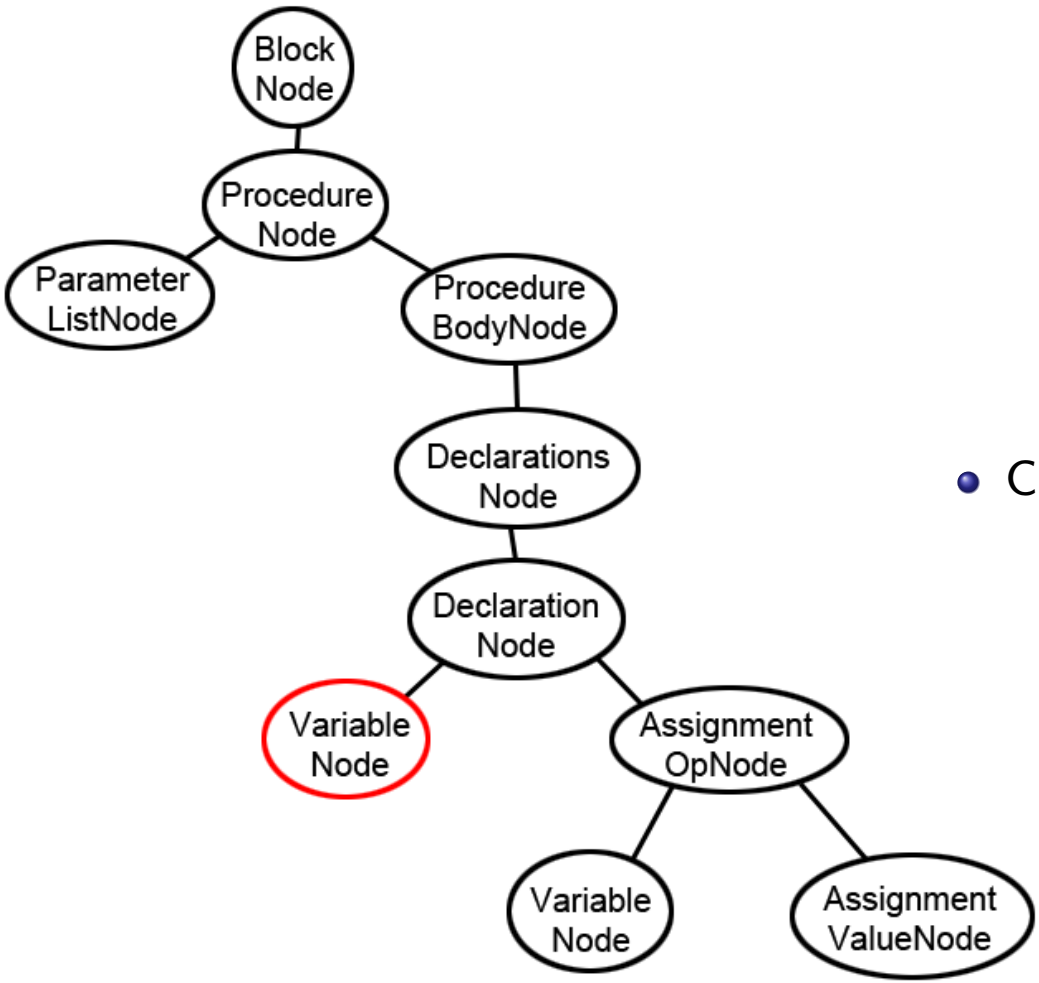
Abstract Syntax Tree



- Insert into current scope.
- Throw error if already declared.

An example: Declaration visitor

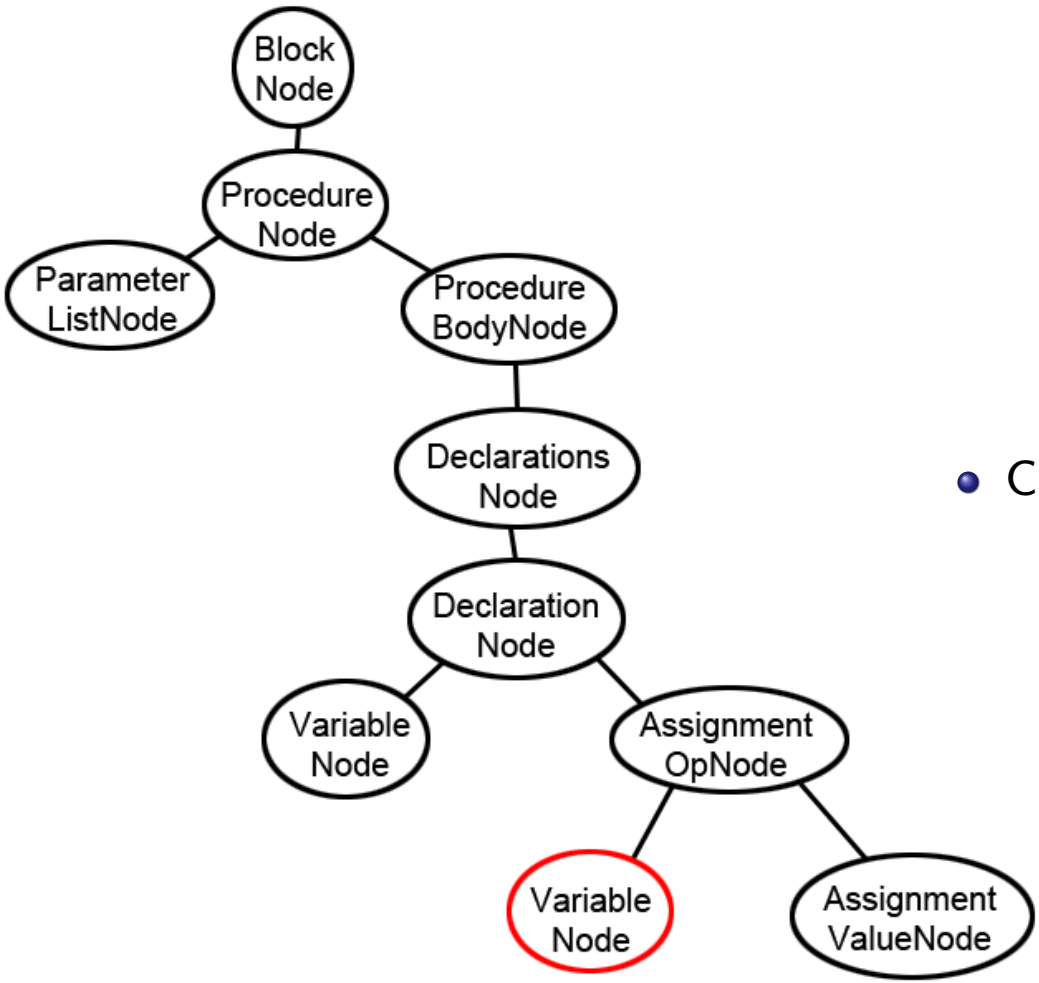
Abstract Syntax Tree



- Check if already declared.

An example: Declaration visitor

Abstract Syntax Tree



- Check if already declared.

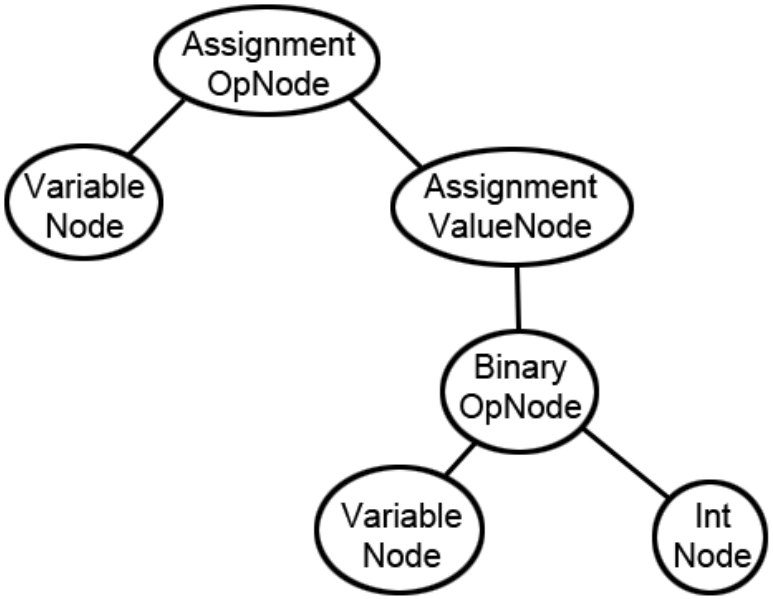
An example: TypeCheck visitor

Type checking example

```
int a = 1;
float b = 1.0;

a = b + 1;
```

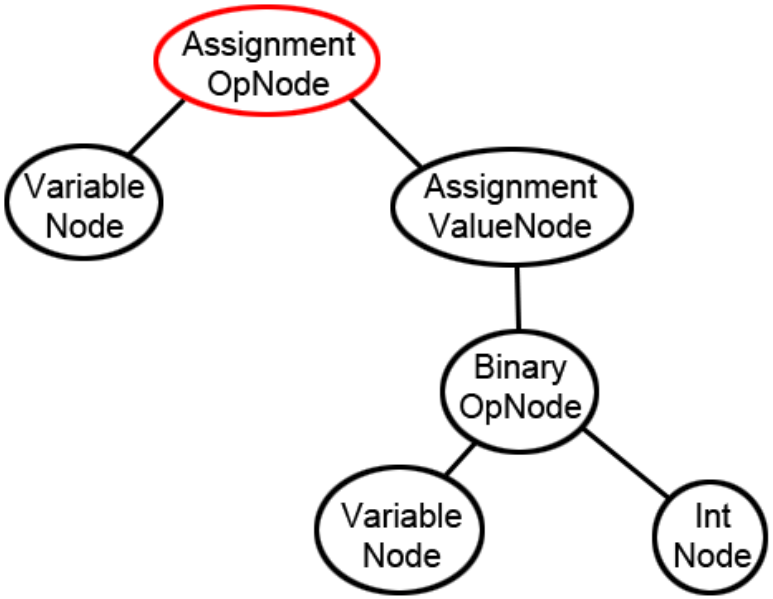
An example: TypeCheck visitor

Abstract Syntax Tree



An example: TypeCheck visitor

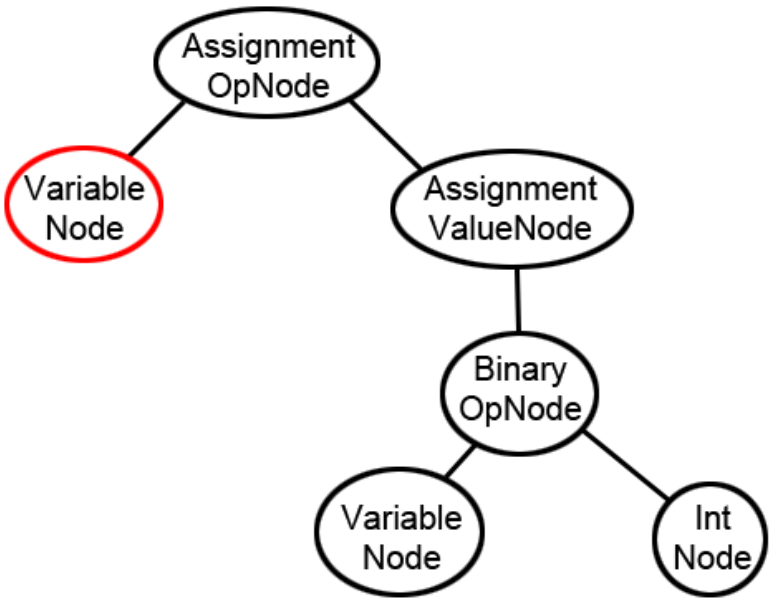
Abstract Syntax Tree



- Visit left and right subtree.
- Throw error if types are incompatible.
- TypeError At Line: 4, Column: 7.

An example: TypeCheck visitor

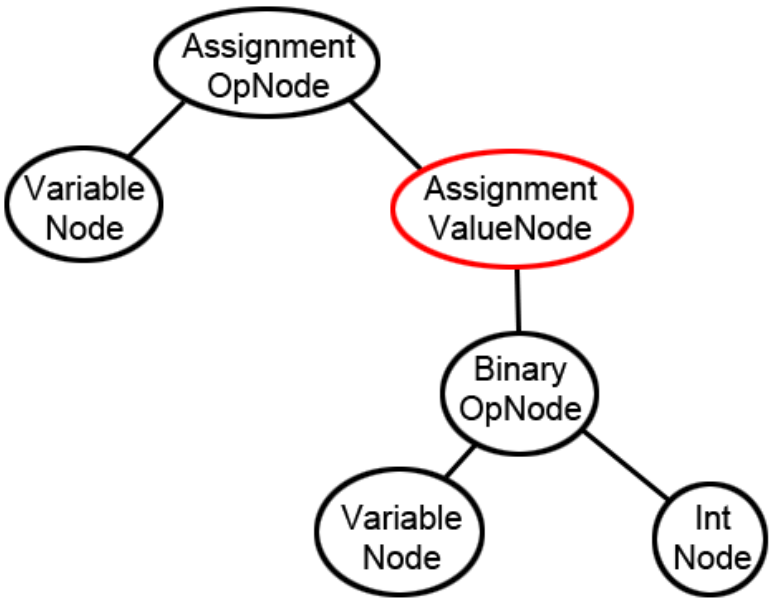
Abstract Syntax Tree



- Return data type.

An example: TypeCheck visitor

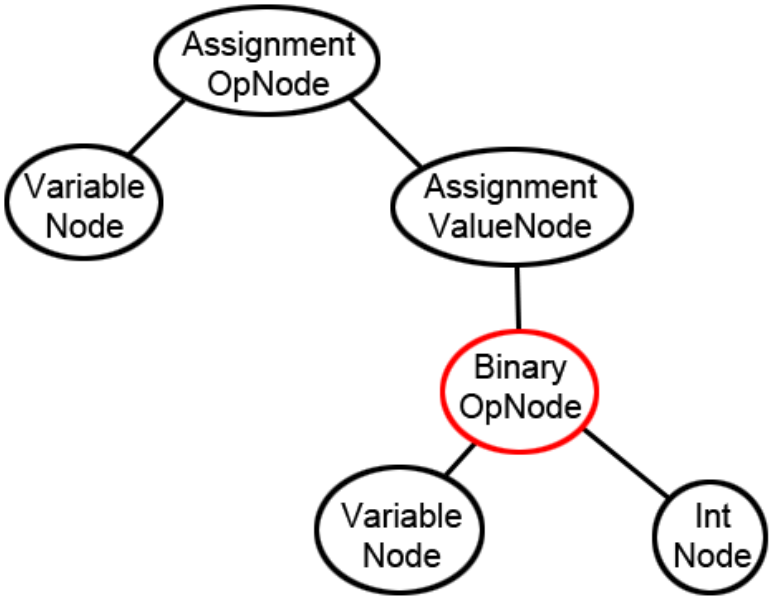
Abstract Syntax Tree



- Propagate type up the tree.

An example: TypeCheck visitor

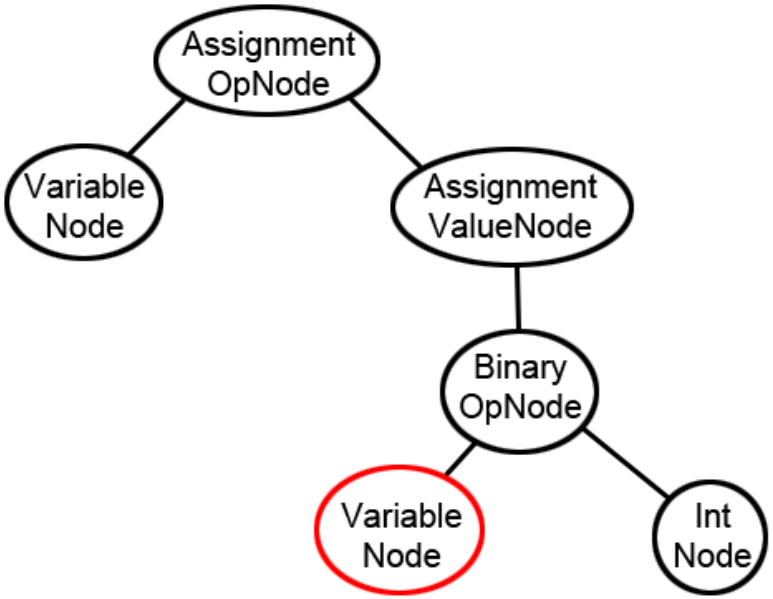
Abstract Syntax Tree



- Visit left and right subtree.
- Throw error if types are incompatible.
- TypeError At Line: 4, Column: 3.

An example: TypeCheck visitor

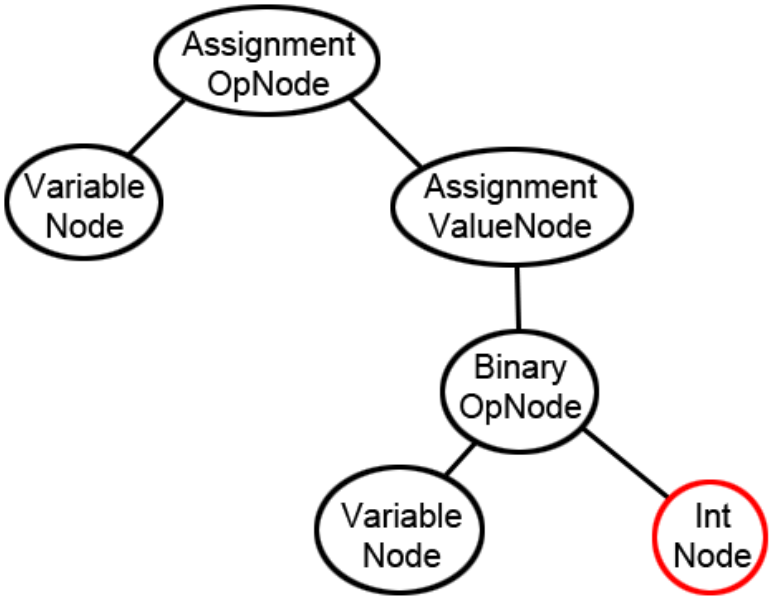
Abstract Syntax Tree



- Return data type.

An example: TypeCheck visitor

Abstract Syntax Tree



- Return data type.

Table of Contents

Code generation for CIP

Key notes about the code generation:

- The code generator is yet another visitor.
- The generated code is a mix of GCC extended C and inline assembly code.

Generally about the code generation:

- The AST is being traversed differently.
- Attempts to exploit the C-like syntax of CIP.
- Specific code is generated during every node visit.

Code emission

When is code being emitted?

- Procedure declarations: The procedure signature and body is emitted independently.
- Variable declarations: Whenever a variable declaration node is visited.
- Scalar operations: Only when an assignment operation node is visited.
- Array operations: At every binary operation node visit.
- Loops and ifs: Like procedures.

An example program

Example program

```
// Declarations
// Scalars
int32 a = 1, b = 2, c = 3;
// Arrays
int32 A[8] = {1, 2, 3, 4, 5, 6, 7, 8},
      B[4] = {2, 4, 6, 8};
int32 C[2][4];

// Scalar binary operation
a = a + b + c;

// Array binary operation
C[1][0]::[3] = A[4]::[7] - (A[0]::[3] + B[0]::[3]);

// Array unary operation (ignore the assignment)
A[0]::[3] = -A[0]::[3];
```

Scalar unary & binary operations

Important notes:

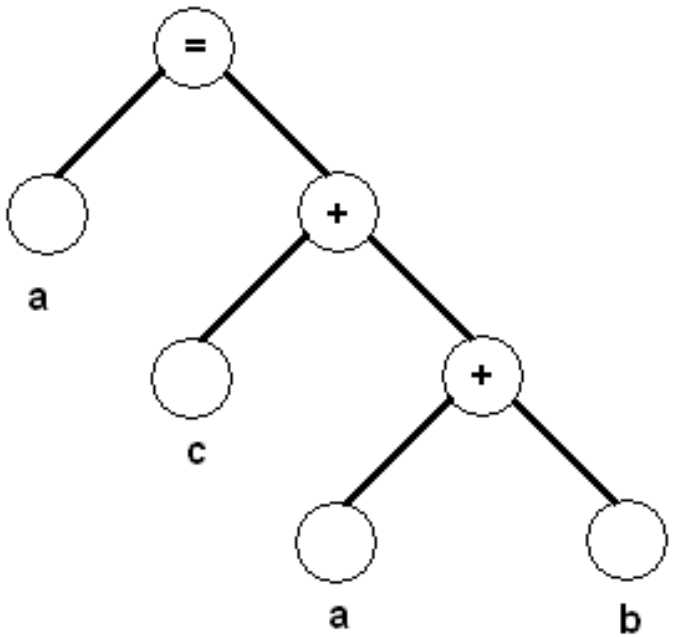
- Exploits the C-like syntax of CIP.
- Conducts a postorder traversal of the AST.
- Only emits code at assignment operations.

An example: Scalar binary operation

Abstract Syntax Tree

Assume:

- a, b and c is already declared.



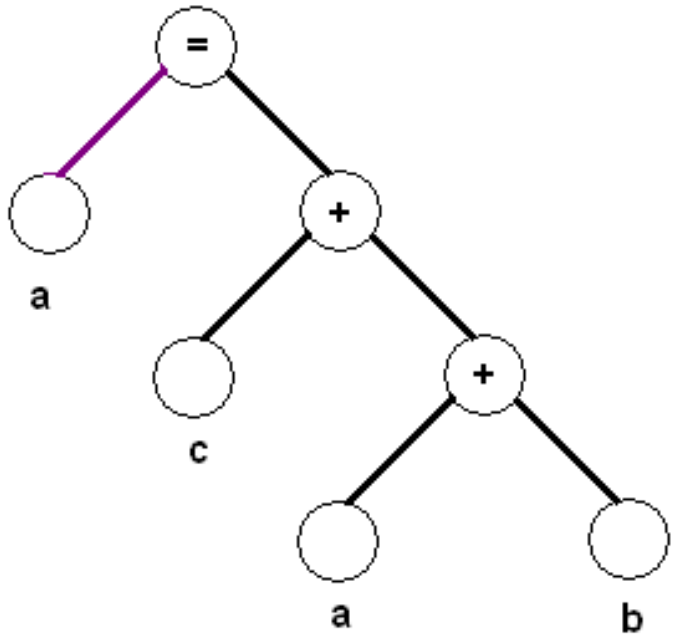
Emitted code

An example: Scalar binary operation

Abstract Syntax Tree

Assume:

- a, b and c is already declared.



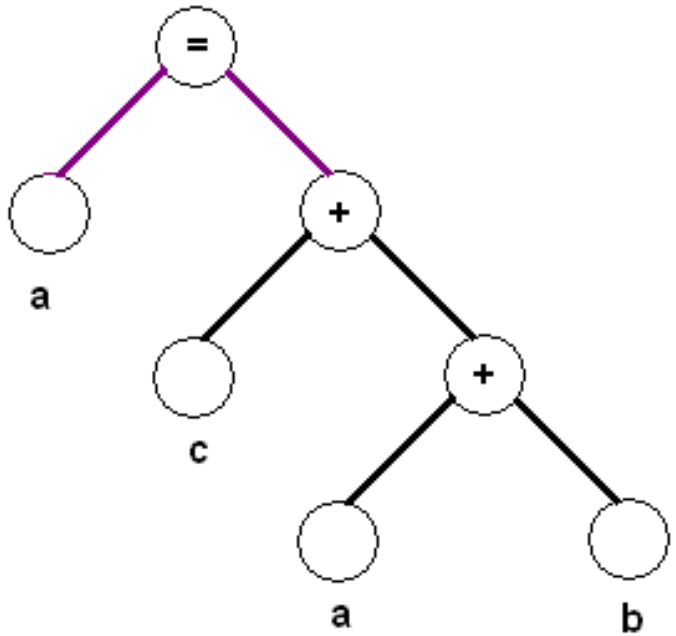
Emitted code

An example: Scalar binary operation

Abstract Syntax Tree

Assume:

- a, b and c is already declared.



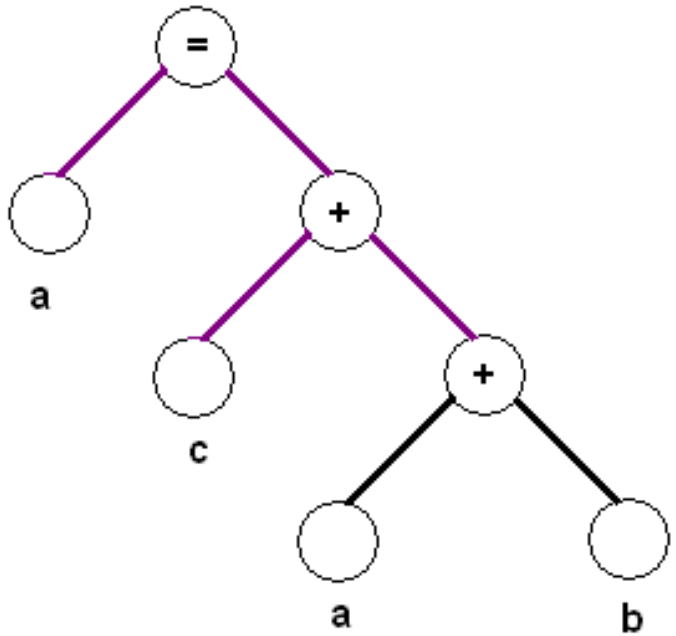
Emitted code

An example: Scalar binary operation

Abstract Syntax Tree

Assume:

- a, b and c is already declared.



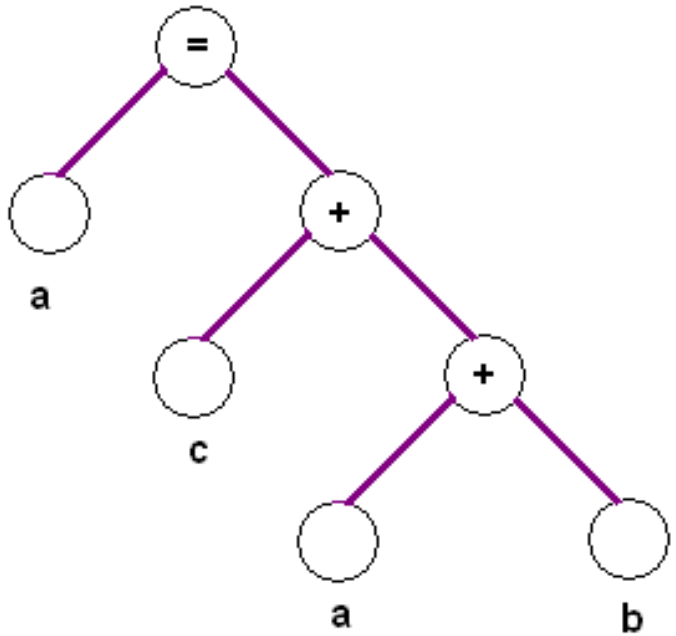
Emitted code

An example: Scalar binary operation

Abstract Syntax Tree

Assume:

- a, b and c is already declared.



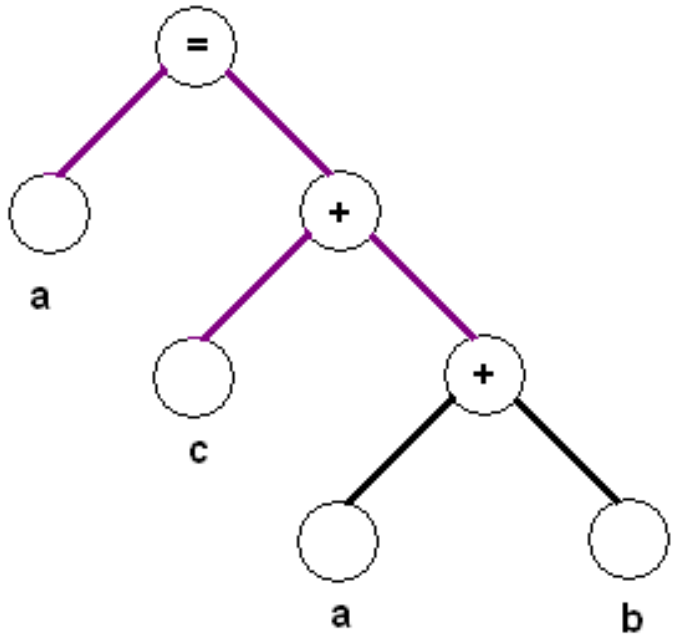
Emitted code

An example: Scalar binary operation

Abstract Syntax Tree

Assume:

- a, b and c is already declared.



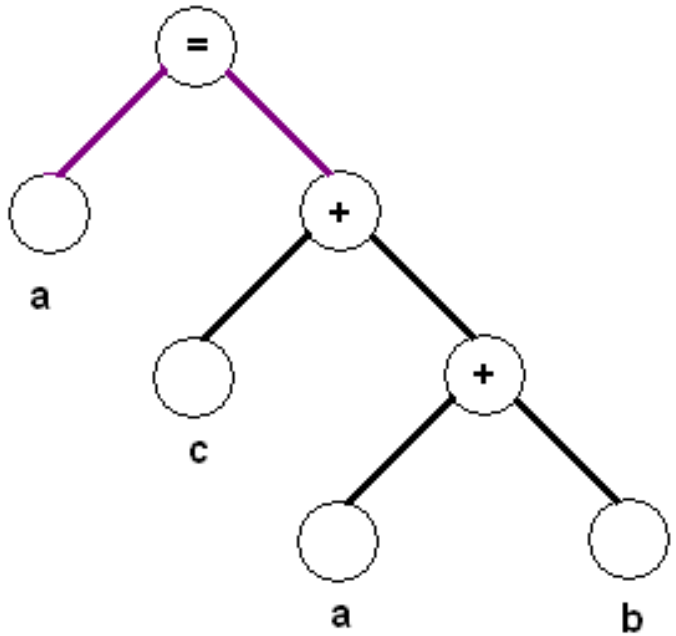
Emitted code

An example: Scalar binary operation

Abstract Syntax Tree

Assume:

- a, b and c is already declared.



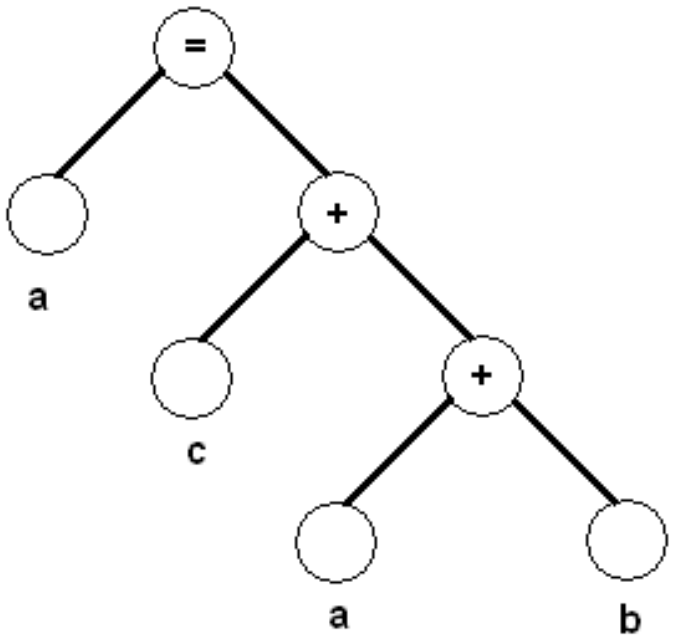
Emitted code

An example: Scalar binary operation

Abstract Syntax Tree

Assume:

- a, b and c is already declared.



Emitted code

```
(a = ((a + b) + c));
```

Array binary operations

A little bit trickier.

- Assembly code emission.
- Cannot exploit the syntax similarities as well as scalar operations.
- Have to maintain an array for the result of the computations.
- Has to make sure all desired elements are computed.

Array binary operations

A little bit trickier.

- Assembly code emission.
- Cannot exploit the syntax similarities as well as scalar operations.
- Have to maintain an array for the result of the computations.
- Has to make sure all desired elements are computed.

An interesting point

- So all operations on arrays are performed in parallel?

Array binary operations

A little bit trickier.

- Assembly code emission.
- Cannot exploit the syntax similarities as well as scalar operations.
- Have to maintain an array for the result of the computations.
- Has to make sure all desired elements are computed.

An interesting point

- So all operations on arrays are performed in parallel? *No, but mostly yes*.

Array binary operations

A little bit trickier.

- Assembly code emission.
- Cannot exploit the syntax similarities as well as scalar operations.
- Have to maintain an array for the result of the computations.
- Has to make sure all desired elements are computed.

An interesting point

- So all operations on arrays are performed in parallel? *No, but mostly yes.*
  - Not all operations have SIMD/SSE2 instructions

Array binary operations

A little bit trickier.

- Assembly code emission.
- Cannot exploit the syntax similarities as well as scalar operations.
- Have to maintain an array for the result of the computations.
- Has to make sure all desired elements are computed.

An interesting point

- So all operations on arrays are performed in parallel? *No, but mostly yes.*
  - Not all operations have SIMD/SSE2 instructions
- OK, OK, so all operations that HAVE SIMD instructions are performed in parallel?

Array binary operations

A little bit trickier.

- Assembly code emission.
- Cannot exploit the syntax similarities as well as scalar operations.
- Have to maintain an array for the result of the computations.
- Has to make sure all desired elements are computed.

An interesting point

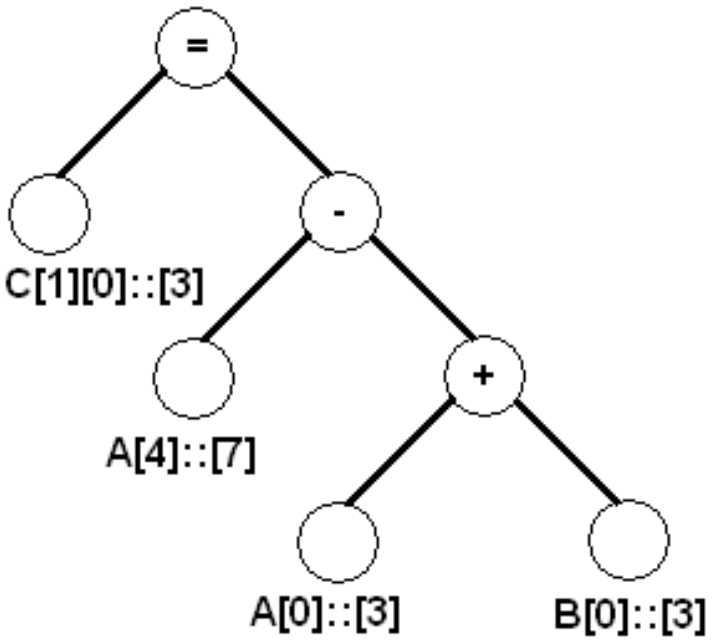
- So all operations on arrays are performed in parallel? *No, but mostly yes.*
  - Not all operations have SIMD/SSE2 instructions
- OK, OK, so all operations that HAVE SIMD instructions are performed in parallel? *No, but mostly yes.*

An example: Array binary operation

Abstract Syntax Tree

Assume:

- A, B is an 1D-array.
- C is a 2D-array.



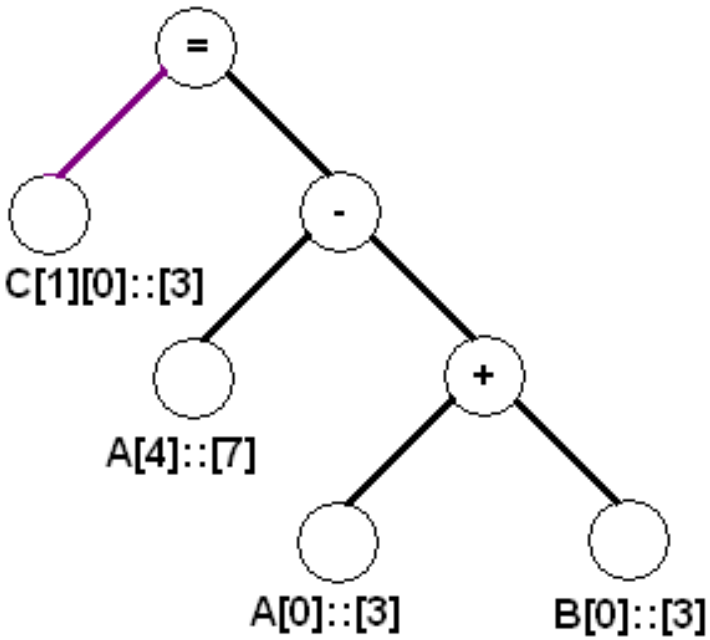
Emitted code

An example: Array binary operation

Abstract Syntax Tree

Assume:

- A, B is an 1D-array.
- C is a 2D-array.



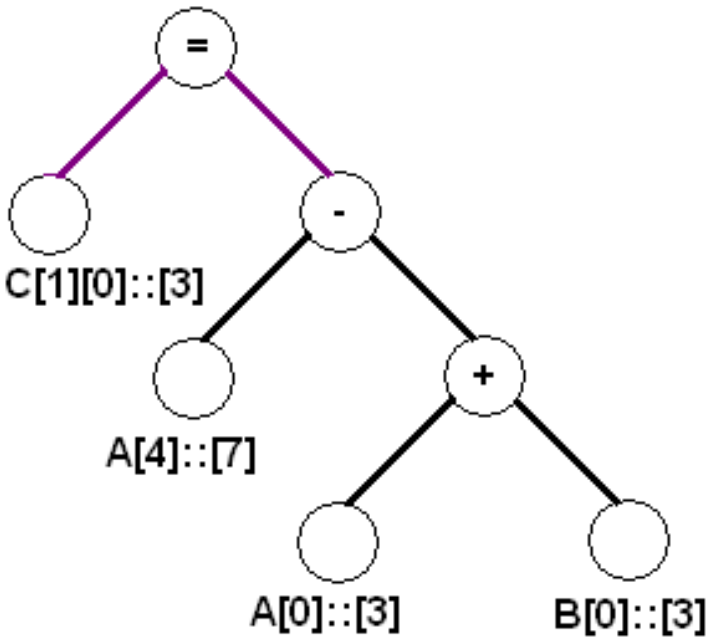
Emitted code

An example: Array binary operation

Abstract Syntax Tree

Assume:

- A, B is an 1D-array.
- C is a 2D-array.



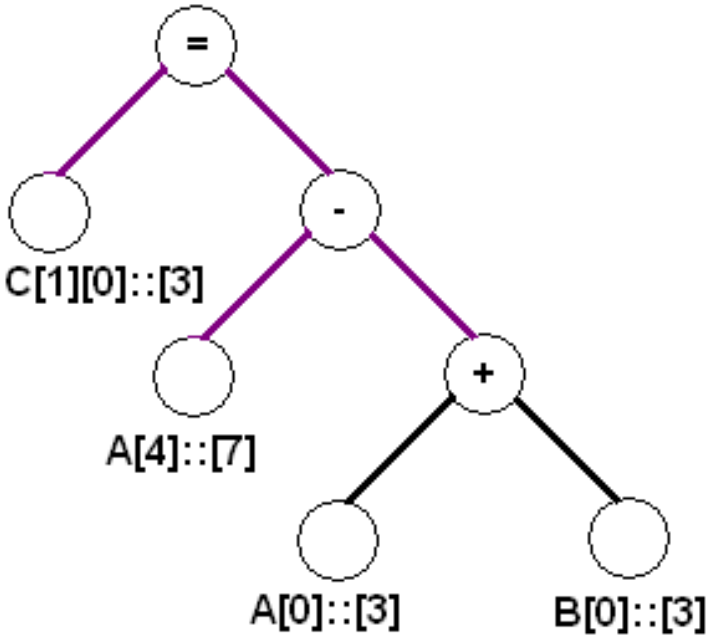
Emitted code

An example: Array binary operation

Abstract Syntax Tree

Assume:

- A, B is an 1D-array.
- C is a 2D-array.



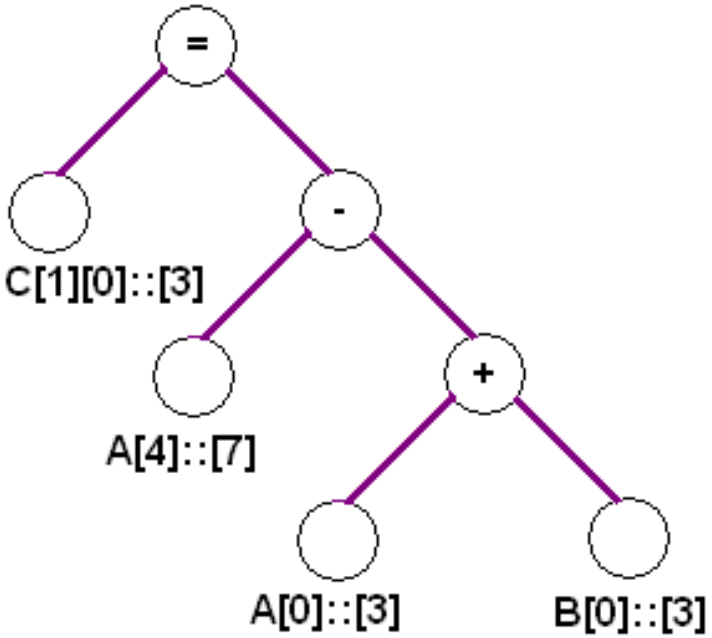
Emitted code

An example: Array binary operation

Abstract Syntax Tree

Assume:

- A, B is an 1D-array.
- C is a 2D-array.



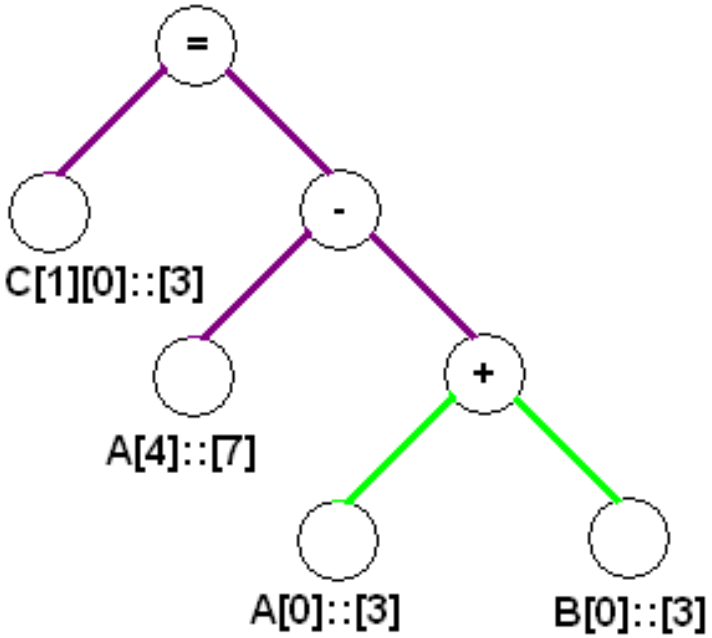
Emitted code

An example: Array binary operation

Abstract Syntax Tree

Assume:

- A, B is an 1D-array.
- C is a 2D-array.



Emitted code

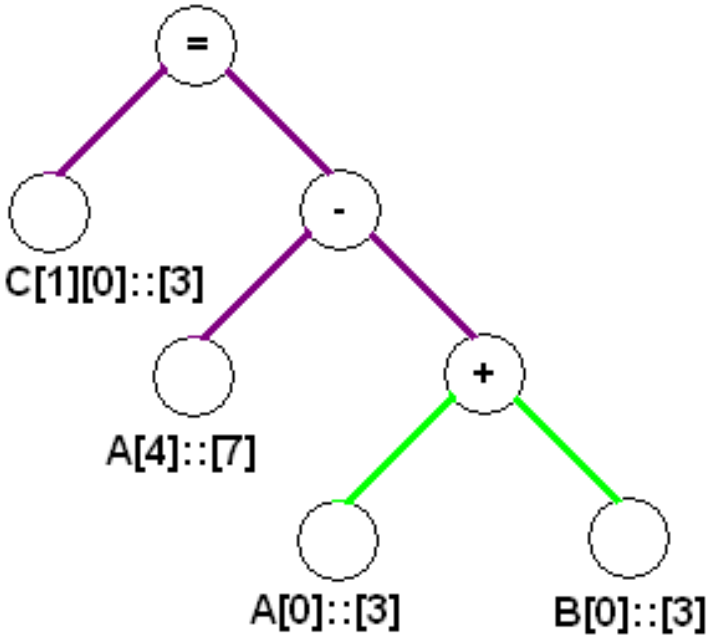
```
long var1  = (3 + 1);
long var2  = (var1 - (var1 % sizeof(signed int)));
signed int* ptr1 = cip_resultarray(var1, sizeof(signed int));
signed int* ptr2 = &A[0];
signed int* ptr3 = &B[0];
asm(``/*...*/ paddd %%xmm1, %%xmm0 /*...*/''
   /* dest: ptr1, src1: ptr2, src2: ptr3 */);
for (var2; var2 < var1; var2++) ptr1[var2] = ptr2[var2] + ptr3[var2];
```

An example: Array binary operation

Abstract Syntax Tree

Assume:

- A, B is an 1D-array.
- C is a 2D-array.



Emitted code

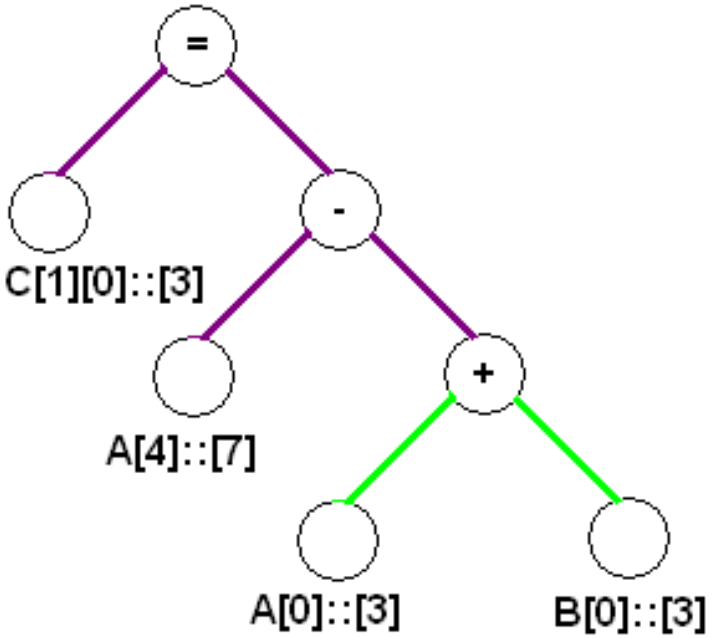
```
long var1  = (3 + 1);
long var2  = (var1 - (var1 % sizeof(signed int)));
signed int* ptr1 = cip_resultarray(var1, sizeof(signed int)));
signed int* ptr2 = &A[0];
signed int* ptr3 = &B[0];
asm(``/*...*/ paddd %%xmm1, %%xmm0 /*...*/''
   /* dest: ptr1, src1: ptr2, src2: ptr3 */);
for (var2; var2 < var1; var2++) ptr1[var2] = ptr2[var2] + ptr3[var2];
```

An example: Array binary operation

Abstract Syntax Tree

Assume:

- A, B is an 1D-array.
- C is a 2D-array.



Emitted code

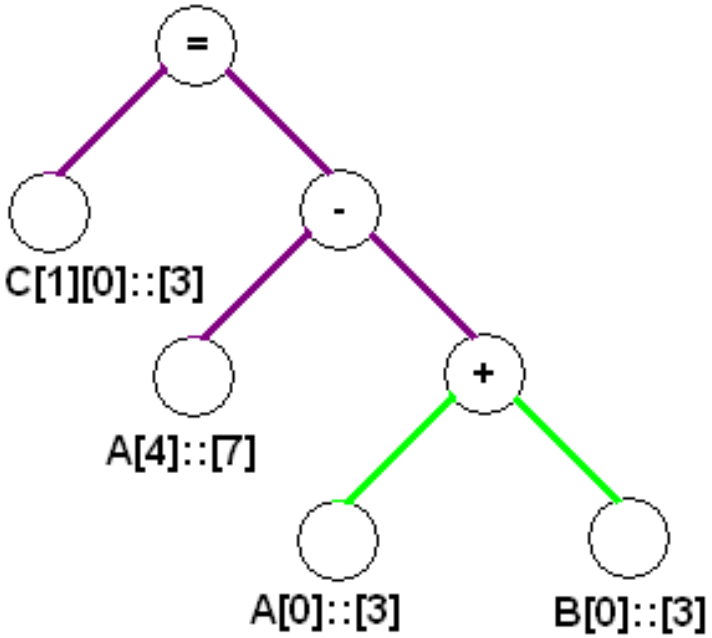
```
long var1  = (3 + 1);
long var2  = (var1 - (var1 % sizeof(signed int)));
signed int* ptr1 = cip_resultarray(var1, sizeof(signed int)));
signed int* ptr2 = &A[0];
signed int* ptr3 = &B[0];
asm(``/*...*/ paddd %%xmm1, %%xmm0 /*...*/''
   /* dest: ptr1, src1: ptr2, src2: ptr3 */);
for (var2; var2 < var1; var2++) ptr1[var2] = ptr2[var2] + ptr3[var2];
```

An example: Array binary operation

Abstract Syntax Tree

Assume:

- A, B is an 1D-array.
- C is a 2D-array.



Emitted code

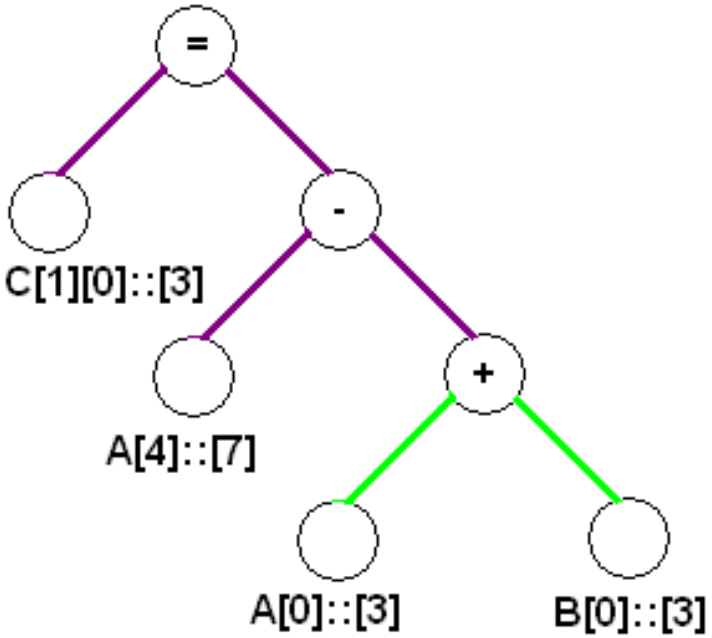
```
long var1  = (3 + 1);
long var2  = (var1 - (var1 % sizeof(signed int)));
signed int* ptr1 = cip_resultarray(var1, sizeof(signed int));
signed int* ptr2 = &A[0];
signed int* ptr3 = &B[0];
asm(``/*...*/ paddd %%xmm1, %%xmm0 /*...*/''
   /* dest: ptr1, src1: ptr2, src2: ptr3 */);
for (var2; var2 < var1; var2++) ptr1[var2] = ptr2[var2] + ptr3[var2];
```

An example: Array binary operation

Abstract Syntax Tree

Assume:

- A, B is an 1D-array.
- C is a 2D-array.



Emitted code

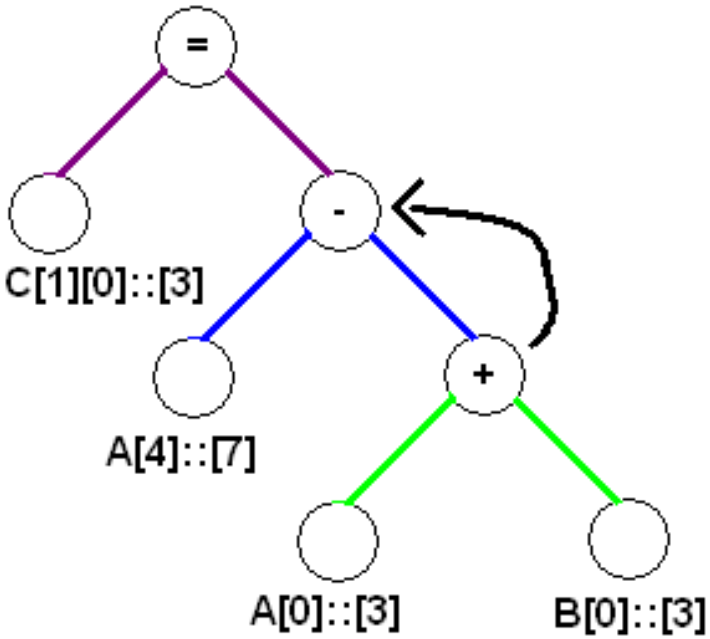
```
long var1  = (3 + 1);
long var2  = (var1 - (var1 % sizeof(signed int)));
signed int* ptr1 = cip_resultarray(var1, sizeof(signed int)));
signed int* ptr2 = &A[0];
signed int* ptr3 = &B[0];
asm(``/*...*/ paddd %%xmm1, %%xmm0 /*...*/''
   /* dest: ptr1, src1: ptr2, src2: ptr3 */);
for (var2; var2 < var1; var2++) ptr1[var2] = ptr2[var2] + ptr3[var2];
```

An example: Array binary operation

Abstract Syntax Tree

Assume:

- A, B is an 1D-array.
- C is a 2D-array.



Emitted code

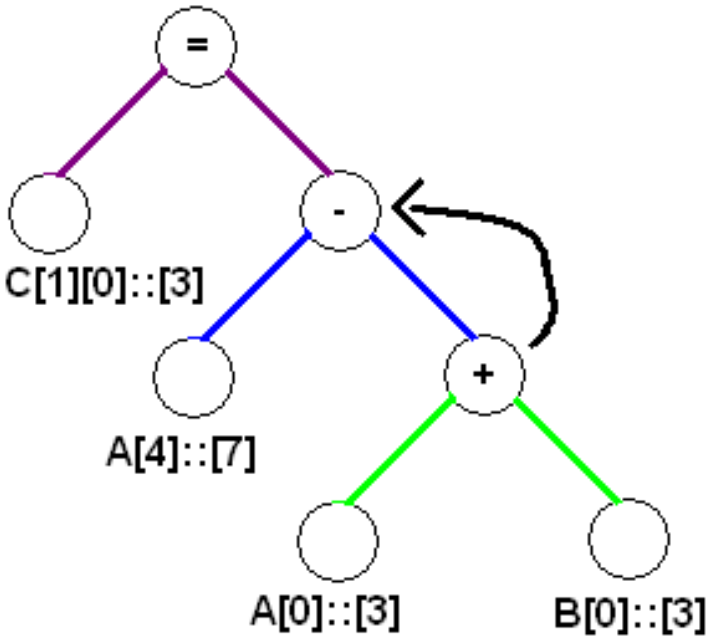
```
long var1  = (3 + 1);
long var2  = (var1 - (var1 % sizeof(signed int)));
signed int* ptr1 = cip_resultarray(var1, sizeof(signed int));
signed int* ptr2 = &A[0];
signed int* ptr3 = &B[0];
asm(``/*...*/ paddd %%xmm1, %%xmm0 /*...*/''
   /* dest: ptr1, src1: ptr2, src2: ptr3 */);
for (var2; var2 < var1; var2++) ptr1[var2] = ptr2[var2] + ptr3[var2];

long var3  = (3 + 1);
long var4  = (var3 - (var3 % sizeof(signed int)));
signed int* ptr4 = &A[3];
signed int* ptr5 = ptr1;
asm(``/*...*/ psubd %%xmm1, %%xmm0 /*...*/''
   /* dest: ptr1, src1: ptr4, src2: ptr5 */);
for (var4; var4 < var3; var4++) ptr1[var4] = ptr4[var4] + ptr5[var4];
```

An example: Array binary operation

Abstract Syntax Tree

Assume:

- A, B is an 1D-array.
- C is a 2D-array.



Emitted code

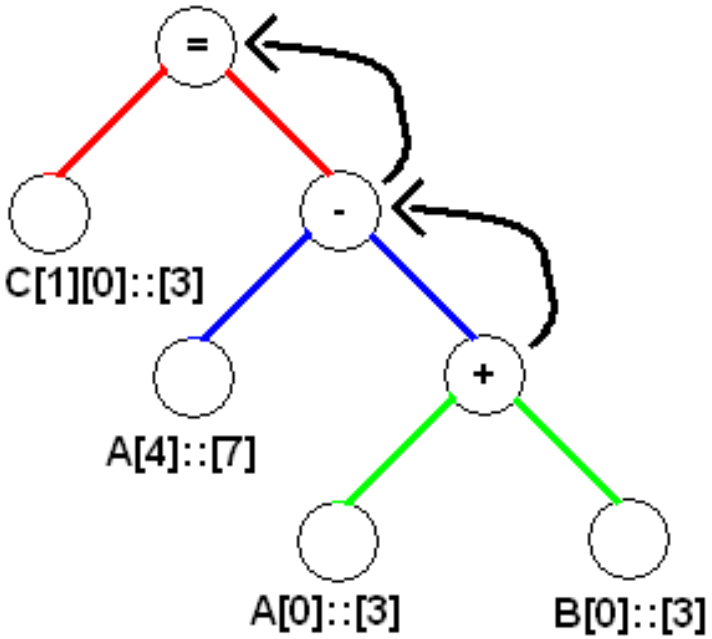
```
long var1  = (3 + 1);
long var2  = (var1 - (var1 % sizeof(signed int)));
signed int* ptr1 = cip_resultarray(var1, sizeof(signed int)));
signed int* ptr2 = &A[0];
signed int* ptr3 = &B[0];
asm(``/*...*/ paddd %%xmm1, %%xmm0 /*...*/''
   /* dest: ptr1, src1: ptr2, src2: ptr3 */);
for (var2; var2 < var1; var2++) ptr1[var2] = ptr2[var2] + ptr3[var2];

long var3  = (3 + 1);
long var4  = (var3 - (var3 % sizeof(signed int)));
signed int* ptr4 = &A[3];
signed int* ptr5 = ptr1;
asm(``/*...*/ psubd %%xmm1, %%xmm0 /*...*/''
   /* dest: ptr1, src1: ptr4, src2: ptr5 */);
for (var4; var4 < var3; var4++) ptr1[var4] = ptr4[var4] + ptr5[var4];
```

An example: Array binary operation

Abstract Syntax Tree

Assume:

- A, B is an 1D-array.
- C is a 2D-array.



Emitted code

```
long var1  = (3 + 1);
long var2  = (var1 - (var1 % sizeof(signed int)));
signed int* ptr1 = cip_resultarray(var1, sizeof(signed int)));
signed int* ptr2 = &A[0];
signed int* ptr3 = &B[0];
asm(``/*...*/ paddd %%xmm1, %%xmm0 /*...*/''
   /* dest: ptr1, src1: ptr2, src2: ptr3 */);
for (var2; var2 < var1; var2++) ptr1[var2] = ptr2[var2] + ptr3[var2];

long var3  = (3 + 1);
long var4  = (var3 - (var3 % sizeof(signed int)));
signed int* ptr4 = &A[3];
signed int* ptr5 = ptr1;
asm(``/*...*/ psubd %%xmm1, %%xmm0 /*...*/''
   /* dest: ptr1, src1: ptr4, src2: ptr5 */);
for (var4; var4 < var3; var4++) ptr1[var4] = ptr4[var4] + ptr5[var4];

long var5  = (3 + 1);
long var6  = (var3 - (var3 % sizeof(signed int)));
signed int* ptr6 = &C[1][0];
signed int* ptr7 = ptr5;
asm(``/*...*/ movaps %xmm0, %[dest] /*...*/''
   /* dest: ptr6, src1: ptr7 */);
for (var6; var6 < var5; var5++) ptr6[var6] = ptr7[var6];
```

Array unary operations

Two different unary operators:

- Logical not (!): Generates assembly code and leftovers handle.
- Negation (-): Simulates a binary operation: $-1 * A$.

Array allocation

We use a pseudo-memory management system to maintain heap allocated arrays.
Code generation considerations about array allocations:

- Allocations in row-major-fashion.
- Side effects during allocations.

Allocation side effect example

Example code

```
int i = 1;
procedure int p() i = i + 1; return i; end
int A[4][p()];
```

Naïve attempt at code generation

Allocation side effect example

Example code

```
int i = 1;
procedure int p() i = i + 1; return i; end
int A[4][p()];
```

Naïve attempt at code generation

```
// Declarations
signed int i = 1;
int p() { i = i + 1; return i; }
signed int **A;
```

Allocation side effect example

Example code

```
int i = 1;
procedure int p() i = i + 1; return i; end
int A[4][p()];
```

Naïve attempt at code generation

```
// Declarations
signed int i = 1;
int p() { i = i + 1; return i; }
signed int **A;
// Allocation of first dimension
signed int var1 = 0;
for (var1; var1 < 1; var1++) *A = cip_alloc(4, sizeof(signed int));
```

Allocation side effect example

Example code

```
int i = 1;
procedure int p() i = i + 1; return i; end
int A[4][p()];
```

Naïve attempt at code generation

```
// Declarations
signed int i = 1;
int p() { i = i + 1; return i; }
signed int **A;
// Allocation of first dimension
signed int var1 = 0;
for (var1; var1 < 1; var1++) *A = cip_alloc(4, sizeof(signed int));
// Allocation of second dimension
signed int var2 = 0;
for (var2; var2 < 4; var2++) **A = cip_alloc(p(), sizeof(signed int));
```

Allocation side effect example

Example code

```
int i = 1;
procedure int p() i = i + 1; return i; end
int A[4][p()];
```

Better attempt at code generation

```
// Declarations
signed int i = 1;
int p() { i = i + 1; return i; }
signed int **A;
```

Allocation side effect example

Example code

```
int i = 1;
procedure int p() i = i + 1; return i; end
int A[4][p()];
```

Better attempt at code generation

```
// Declarations
signed int i = 1;
int p() { i = i + 1; return i; }
signed int **A;
// Allocation of first dimension
signed int var1 = 0, var2 = 1, var3 = 4;
for (var1; var1 < var2; var1++) *A = cip_alloc(var3, sizeof(signed int));
```

Allocation side effect example

Example code

```
int i = 1;
procedure int p() i = i + 1; return i; end
int A[4][p()];
```

Better attempt at code generation

```
// Declarations
signed int i = 1;
int p() { i = i + 1; return i; }
signed int **A;
// Allocation of first dimension
signed int var1 = 0, var2 = 1, var3 = 4;
for (var1; var1 < var2; var1++) *A = cip_alloc(var3, sizeof(signed int));
// Allocation of second dimension
signed int var4 = 0, var5 = p();
for (var4; var4 < var3; var4++) **A = cip_alloc(var5, sizeof(signed int));
```

Table of Contents

Syntax errors

- Errors produced by the parser were originally handled by the parser itself.

```
-- line 3 col 10: ";" expected

Compile errors:
        [ UsedBeforeDeclarationError At Line: 3, Column: 1 ]
                Identifier: function
```

Syntax errors

- Errors produced by the parser were originally handled by the parser itself.

```
-- line 3 col 10: ";" expected

Compile errors:
        [ UsedBeforeDeclarationError At Line: 3, Column: 1 ]
                Identifier: function
```

- The parser now adds them to the compiler's exception handler.

```
        Compile errors:
                [ SyntaxError At Line: 3, Column: 10 ]
                        ";" expected
```

Result array

- Memory for the result array was always reallocated, despite its size.

Result array

- Memory for the result array was always reallocated, despite its size.
- This approach was unnecessary as the array might be reusable.

Result array

- Memory for the result array was always reallocated, despite its size.
- This approach was unnecessary as the array might be reusable.
- Memory allocation is now only done if a larger result array is needed.

Result array

- Memory for the result array was always reallocated, despite its size.
- This approach was unnecessary as the array might be reusable.
- Memory allocation is now only done if a larger result array is needed.
- Has its own drawbacks when large and small arrays are used together.

Result array

- Memory for the result array was always reallocated, despite its size.
- This approach was unnecessary as the array might be reusable.
- Memory allocation is now only done if a larger result array is needed.
- Has its own drawbacks when large and small arrays are used together.
- More benchmarks would be needed to show which approach is best suited.

Corrections

- Inline assembler is now also generated for all comparison operations.

Corrections

- Inline assembler is now also generated for all comparison operations.
- A single element from an array can now be returned by procedures.

Corrections

- Inline assembler is now also generated for all comparison operations.
- A single element from an array can now be returned by procedures.
- The correct type is now used for temporary variables and pointers.

Corrections

- Inline assembler is now also generated for all comparison operations.
- A single element from an array can now be returned by procedures.
- The correct type is now used for temporary variables and pointers.
- Code for logical negation is now produced correctly for arrays.

Corrections

- Inline assembler is now also generated for all comparison operations.
- A single element from an array can now be returned by procedures.
- The correct type is now used for temporary variables and pointers.
- Code for logical negation is now produced correctly for arrays.
- The check for floats as array indexes is now only done on array indexes.

Corrections

- Inline assembler is now also generated for all comparison operations.
- A single element from an array can now be returned by procedures.
- The correct type is now used for temporary variables and pointers.
- Code for logical negation is now produced correctly for arrays.
- The check for floats as array indexes is now only done on array indexes.
- Use of the range operator on an array passed to a procedure is now verified correctly.

Corrections

- Operations involving scalars and arrays now use the array as input.

Corrections

- Operations involving scalars and arrays now use the array as input.

```
int a = 10;
int A[5] = {1, 2, 3, 4, 5};

A[0]::[4] = A[0]::[4] * 10;
```

Corrections

- Operations involving scalars and arrays now use the array as input.

```
int a = 10;
int A[5] = {1, 2, 3, 4, 5};

A[0]::[4] = A[0]::[4] * 10;
```

- Assignment between arrays and scalars is no longer possible at all.

Corrections

- Operations involving scalars and arrays now use the array as input.

```
int a = 10;
int A[5] = {1, 2, 3, 4, 5};

A[0]::[4] = A[0]::[4] * 10;
```

- Assignment between arrays and scalars is no longer possible at all.

```
scalar = Array[0]::[4];
Array[0]::[4] = scalar;
```

Corrections

- The dimensions of an actual parameter are now checked correctly.

Corrections

- The dimensions of an actual parameter are now checked correctly.

```
procedure void proc(int[][] A)
    print "I am a procedure";
end

int Array[5] = {1, 2, 3, 4, 5};
proc(Array);
```

Corrections

- The dimensions of an actual parameter are now checked correctly.

```
procedure void proc(int[][] A)
    print "I am a procedure";
end

int Array[5] = {1, 2, 3, 4, 5};
proc(Array);
```

- A range operator used on an actual parameter now generates correct code.

Corrections

- The dimensions of an actual parameter are now checked correctly.

```
procedure void proc(int[][] A)
    print "I am a procedure";
end

int Array[5] = {1, 2, 3, 4, 5};
proc(Array);
```

- A range operator used on an actual parameter now generates correct code.

```
proc(Array[3]::[4]);

proc(Array[3]);
```

Known problems

- The result of array comparisons is not consistent.

Known problems

- The result of array comparisons is not consistent.
- Not a problem as our language does not contain bitwise operators.

Known problems

- The result of array comparisons is not consistent.
- Not a problem as our language does not contain bitwise operators.

- The result array is shared between scopes which could give side effects.

Known problems

- The result of array comparisons is not consistent.
- Not a problem as our language does not contain bitwise operators.

- The result array is shared between scopes which could give side effects.
- Only a problem if operations in the new scopes use the result array.

Known problems

- The result of array comparisons is not consistent.
- Not a problem as our language does not contain bitwise operators.

- The result array is shared between scopes which could give side effects.
- Only a problem if operations in the new scopes use the result array.

- Assembly code generated from scalar and array computations of type int8 contains some errors.

Known problems

- The result of array comparisons is not consistent.
- Not a problem as our language does not contain bitwise operators.

- The result array is shared between scopes which could give side effects.
- Only a problem if operations in the new scopes use the result array.

- Assembly code generated from scalar and array computations of type int8 contains some errors.
- This can sometimes result in memory corruption or incorrect results.

Table of Contents

- Compiler Flags
- Fibonacci
- Array Arithmetic

Table of Contents

How may we design and implement a programming language that utilises the concepts of SIMD

- Programing language
- SIMD capabilities
- Improvements and corrections
- Still some problems

Language goals

- Data parallelism without encumbering the programmer
- Focus on being productive
- Arrays as first-class

CIP compared to C

- Time utility
- Two equivalent program for each data type
- Test of int32

CIP test code for int32

```
procedure void FillArray(int32[] array, int32 n)
int32 i = 0;
while (i < n)
   array[i] = 128;
   i = i + 1;
end
end
int32 i = 134217728;

int32 A[i], B[i];

FillArray(A, i);
FillArray(B, i);
A[0]::[i-1] = A[0]::[i-1] + B[0]::[i-1];
```

C test code for int32

```
#include <stdio.h>
#include <stdlib.h>
#include "stklib.h"
#include "cipalloclib.h"

void fill_array(int array[], int n) {
    int i = 0;

    while(i < n) {
        array[i] = 128;
        i = i + 1;
    }
}

int main() {

    int i,j;

    j = 134217728;
```

C test code for int32 continued

```
    int *A = cip_alloc(j, sizeof(int));
    int *B = cip_alloc(j, sizeof(int));
    int *C = cip_alloc(j, sizeof(int));

    fill_array(A, j);
    fill_array(B, j);

    for(i=0; i < j; i++){
        A[i] = A[i] + B[i];
    }

    for(i=0; < j; i++){
        C[i] = A[i];
    }

    cip_free();

    return 0;
}
```

Int32 test results in seconds

	1.5 GB allocated	
data type	**C**	**CIP**
32-bit Integer	2.397	1.493

- LL(k) or LALR instead of LL(1)
- Memory management
- Expand SIMD to contructs
- Implementation of bitwise operations
- Return array from procedures