Compilers Course Project

Report

- Jerin Philip <jerin.philip@research.iiit.ac.in>
- 201401071

FlatB

Description

Semantics

The following example illustrates a FlatB program.

```
declblock {
    int x, y; // Integer
    int A[100];  // Integer Array
}

codeblock {

    // Assignment
    x = 0;
    y = 2;
    A[x] = 1;
    A[y] = 2;

    // Arithmetic Expressions

    x = y + 2;
    y = x * 3 + y;

    // IO
    print "Value of x is:", x;
    read y;
    read A[y];


    // Conditionals
    if ( x < y ) {
        print " x < y : ", x, "<", y;
```

```
    } else {

        print " x > y : ", x, ">", y;
    }


    // Loops
    for i=0, 100-1 {
        A[i] = i + 1;
    }

    for i=0, 100, 2 {
        print i;

    }

    x = 100;
    while ( x > 0 ){
        print x;
        x = x - 1;
    }

L1: x = 100;
    x = x - 1;
    goto L1 if (x > 0); // Conditional goto


L2: x = 100;
    if (x / 2 < 25) {
        goto EXIT;
    }
    x =  x - 3;
    goto L2; // Unconditional goto


EXIT:
    print "Program exits now!";
    print "Goodbye world";
}
```

AST

The following UML Inheritance diagram describes AST:

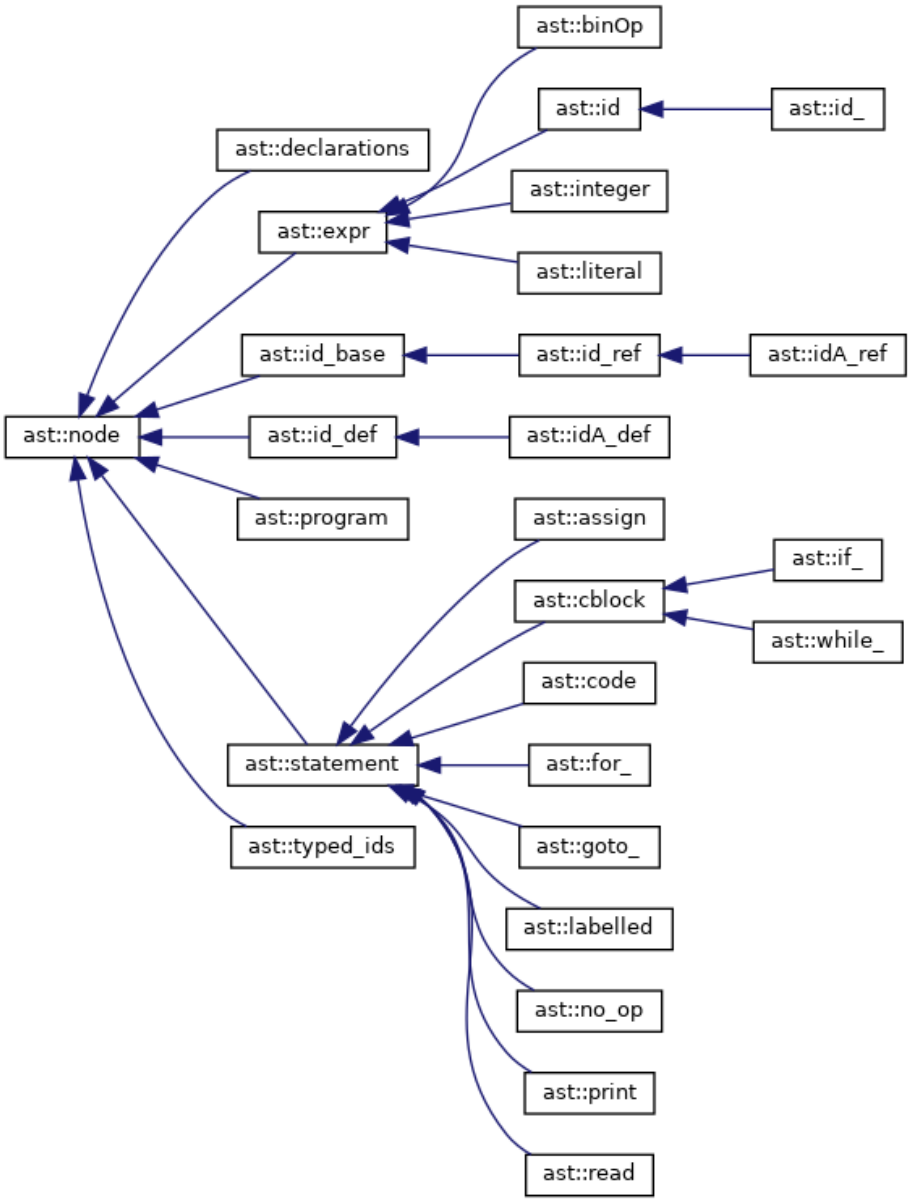

ast::binOp
ast::id
ast::id_
ast::declarations
ast::integer
ast::expr
ast::literal
ast::id_base
ast::id_ref
ast::idA_ref
ast::node
ast::id_def
ast::idA_def
ast::program
ast::assign
ast::if_
ast::cblock
ast::while_
ast::code
ast::statement
ast::for_
ast::typed_ids
ast::goto_
ast::labelled
ast::no_op
ast::print
ast::read

Figure 1: AST

It should be understandable which class maps to which statement/construct of the programming language.

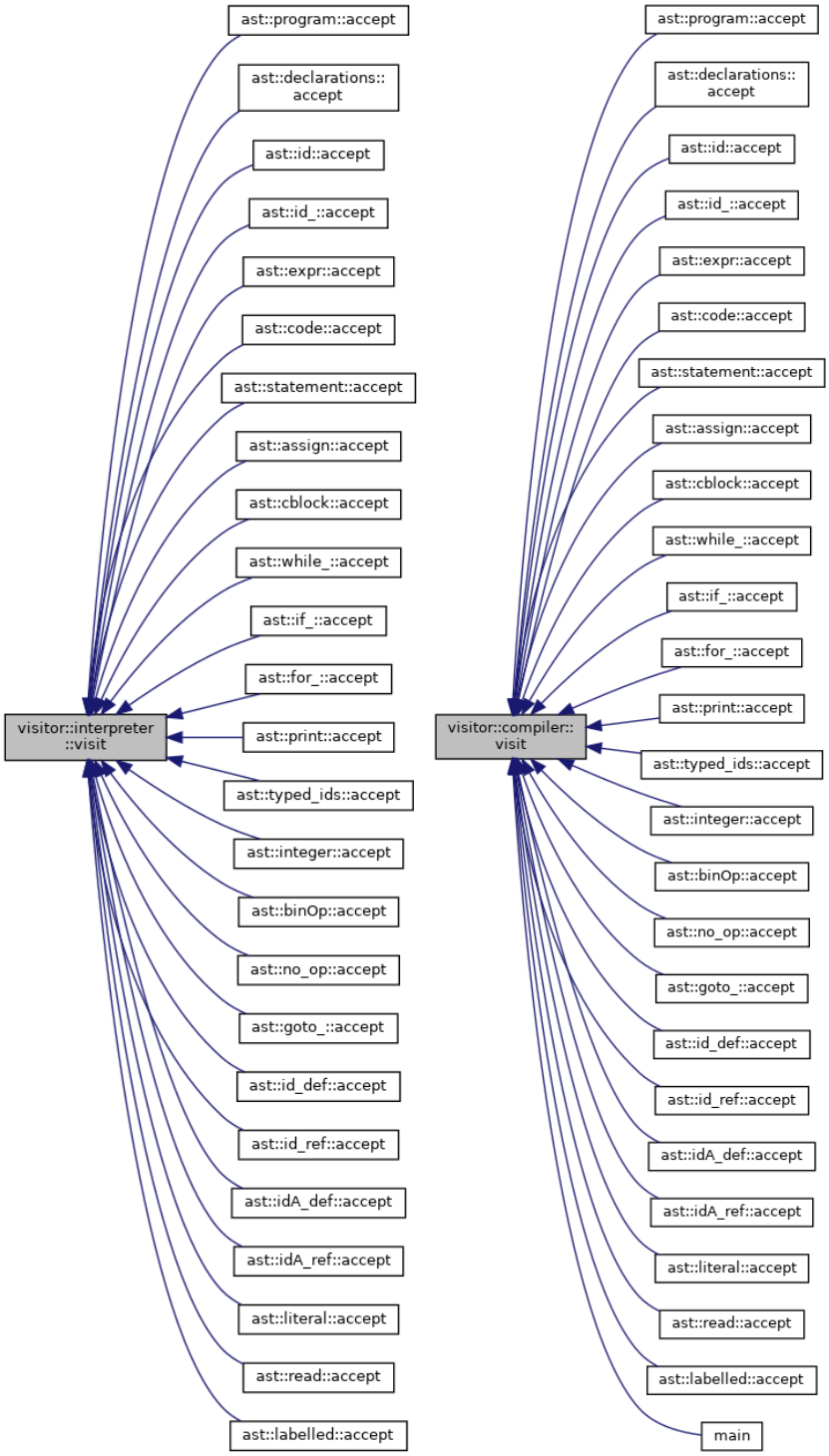
Details which are not straightforward:

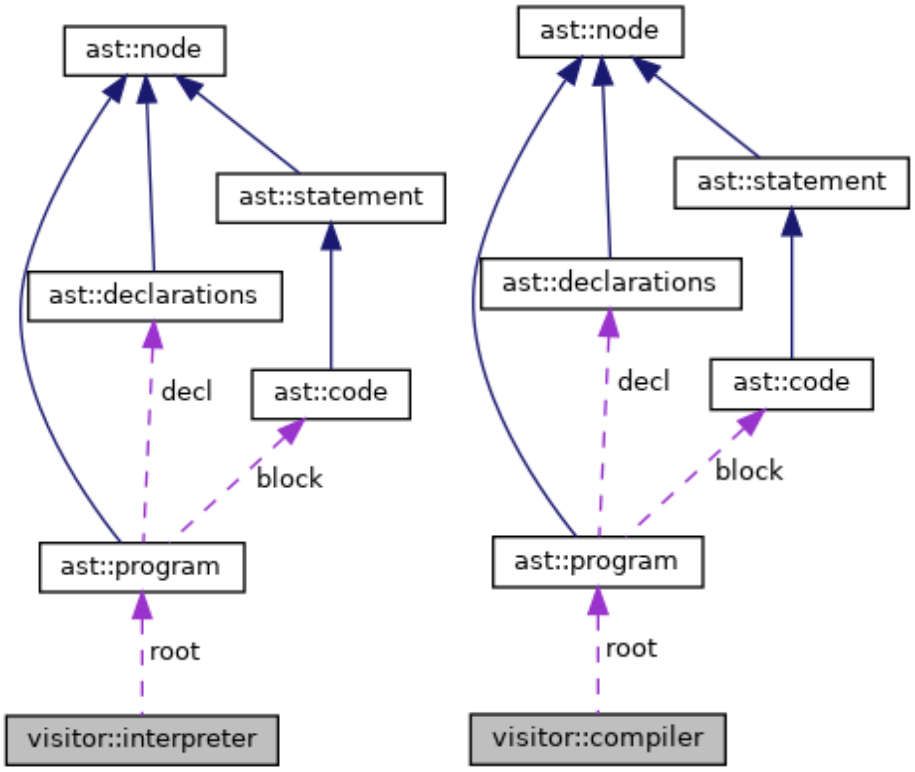
- There are 3 types of id objects, 1 each for an array type and integer type.
  - `ast::id_def`, `ast::idA_def`: Used in declarations, `visit` creates the variables.
  - `ast::id`, `ast::id_`: Used in access, `visit` returns value or returns load memory instruction, depending on interpreter or compiler.
  - `ast::id_ref`, `ast::idA_ref`: Used in access, `visit` returns value or returns load memory instruction, depending on interpreter or compiler.
- For is implemented C-style internally.
  - `for(init; check; step) { block };`
  - `init` is an assignment statement, created using the parsed entities - initializes the looping variable.
  - `check` is a boolean operation created using the end of range and variable.
  - `step` is the update of the loop.
  - `block` is the loop body.
- Interpreter does an inorder tree walk and `goto` shows undefined behaviour in certain badly-formed cases.

Visitor pattern

The visitor patter enables separation of the algorithm - here the interpretation or compilation by run-time detection of which member of the class hierarchy an object resolves to.

The following diagrams shows how compiler's and interpreter's `visit` functions are called with the types of the respective ast node object by using `accept` inside the ast node object.


visitor::interpreter ::visit
ast::program::accept
ast::declarations:: accept
ast::id::accept
ast::id_::accept
ast::expr::accept
ast::code::accept
ast::statement::accept
ast::assign::accept
ast::cblock::accept
ast::while_::accept
ast::if_::accept
ast::for_::accept
ast::print::accept
ast::typed_ids::accept
ast::integer::accept
ast::binOp::accept
ast::no_op::accept
ast::goto_::accept
ast::id_def::accept
ast::id_ref::accept
ast::idA_def::accept
ast::idA_ref::accept
ast::literal::accept
ast::read::accept
ast::labelled::accept
visitor::compiler:: visit
ast::program::accept
ast::declarations:: accept
ast::id::accept
ast::id_::accept
ast::expr::accept
ast::code::accept
ast::statement::accept
ast::assign::accept
ast::cblock::accept
ast::while_::accept
ast::if_::accept
ast::for_::accept
ast::print::accept
ast::typed_ids::accept
ast::integer::accept
ast::binOp::accept
ast::no_op::accept
ast::goto_::accept
ast::id_def::accept
ast::id_ref::accept
ast::idA_def::accept
ast::idA_ref::accept
ast::literal::accept
ast::read::accept
ast::labelled::accept
main



From the call-graph, we can deduce that the calls for the base node is being processed, and internally, using v-table rerouted to the respective object's accept. There is no explicit cast to the actual object in the program.

Interpreter

The interpreter is written in C++. The AST Structure and the corresponding logic is programmed as a visitor, shown in the earlier diagrams.

Compiler

Compiler emits LLVM 5 IR equivalent to the logic of the program, loading it from the AST.

Performance Comparison

The following table indicates the `real` time taken on running programs in `test-units/non-trivial` on my system. On the smaller programs, there is not much indication, but matrix multiplication of higher dimensions show that my own interpreter is the worst, `lli` performs better but worst than compiled bytecode.

Time Elapsed

program	interpreter	lli	llc
99-bottles	0m0.019s	0m0.020s	0m0.002s
fibonnacci	0m0.010s	0m0.013s	0m0.000s
reversal	0m0.016s	0m0.017s	0m0.002s
bubblesort	0m0.081s	0m0.011s	0m0.002s
matmul_5	0m0.019s	0m0.028s	0m0.002s
matmul_125	0m27.972s	0m0.037s	0m0.014s
matmul_250	3m43.217s	0m0.101s	0m0.081s

Instructions

program	interpreter	lli	llc
99-bottles.b	4,20,31,101	4,91,23,189	44,16,209
fibonnacci.b	4,11,16,141	4,83,83,573	42,94,451
reversal.b	4,80,11,228	5,54,04,987	45,04,330
bubblesort.b	33,84,15,192	7,06,64,031	1,87,77,417
matmul_125.b	1,13,80,51,77,753	15,02,50,333	8,00,21,360
matmul_5.b	5,96,79,582	7,34,06,000	43,87,608
matmul_250.b	9,00,64,72,80,584	52,91,62,979	45,53,01,005