CM4106 LANGUAGE AND COMPILERS

COMPILERS

WHY LEARN ABOUT COMPILERS?

- It is unlikely that you will ever need to write a compiler for a big language like Java, C# or C

A GOOD CRAFTSMAN?

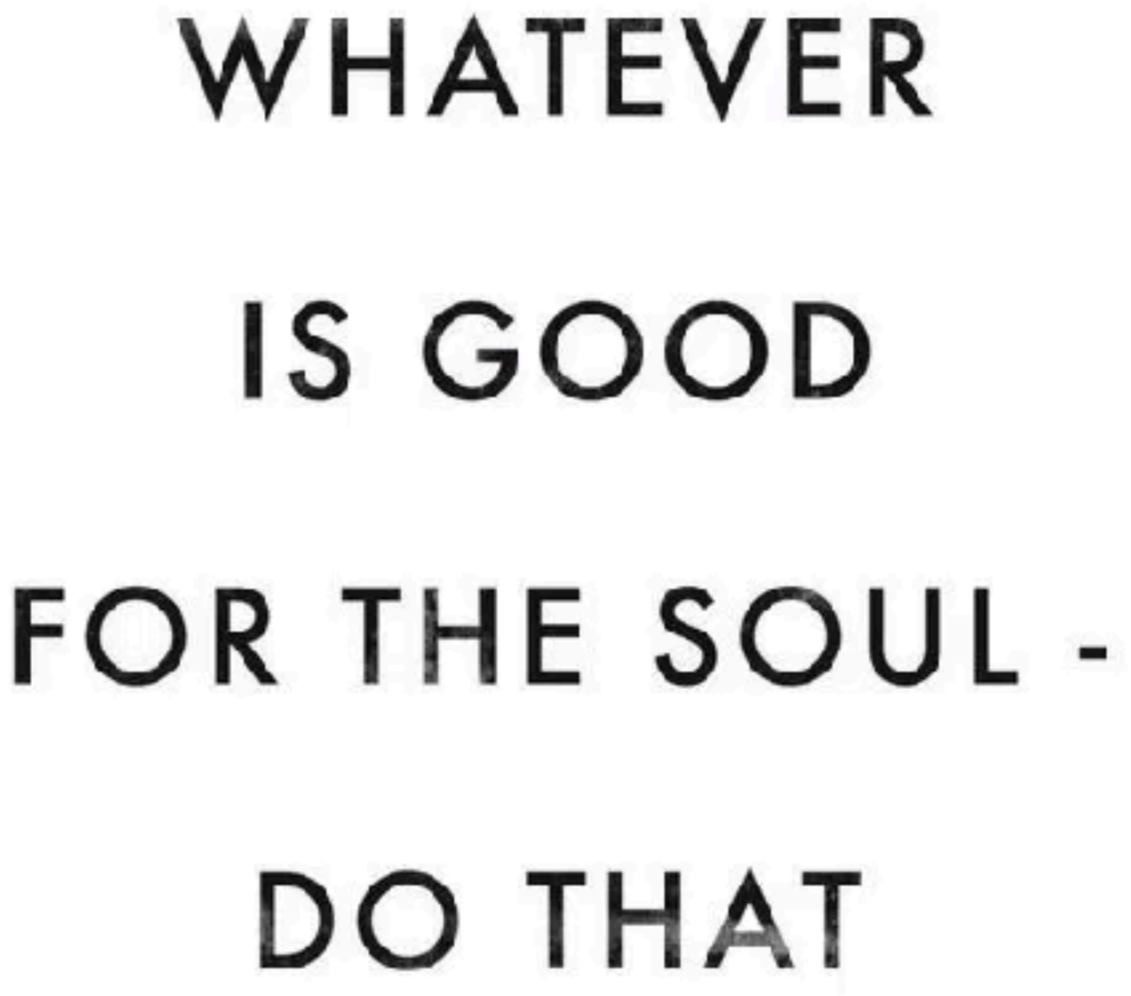
- Most of you are / will be coders, or work with other people who are coders.
- If you know how a compiler works you can guess how you code will compile and make it more efficient
- At the very least it will help you understand error messages!
- A good craftsman knows his tools.

OTHER SKILLS

- Compilers use a number of techniques to compile code.
  - Scanning
  - Parsing
  - Recursive functions
  - Finite State Machines

BUT WHY?

- DSL - Domain Specific Languages
- A language written for a specific problem -
  - data access
  - setting up simulations
  - npc character control
- Target would be another language, ie the language the game or simulation is written in.


WHATEVER

IS GOOD

FOR THE SOUL -

DO THAT

It’s hard, so it makes you think

Way better than yoga.

WHAT IS A COMPILER

- A translator from a program written in a high level language
- To a program written in a low - level or machine language
- Used to spot errors and report mistakes in code

WHY DO WE NEED THEM.

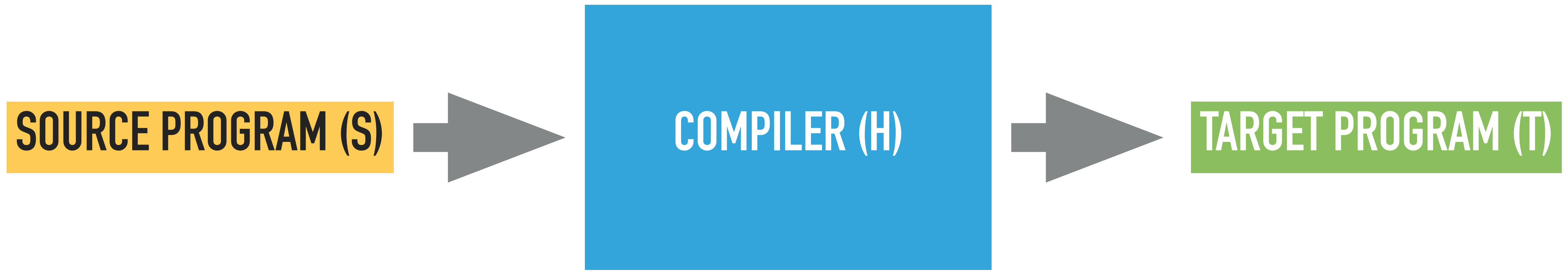
```
b8   21 0a 00 00
a3   0c 10 00 06
b8   6f 72 6c 64
a3   08 10 00 06
b8   6f 2c 20 57
a3   04 10 00 06
b8   48 65 6c 6c
a3   00 10 00 06
b9   00 10 00 06
ba   10 00 00 00
bb   01 00 00 00
b8   04 00 00 00
cd   80
b8   01 00 00 00
cd   80
```

Any body guess what this does?

WE NEED THEM

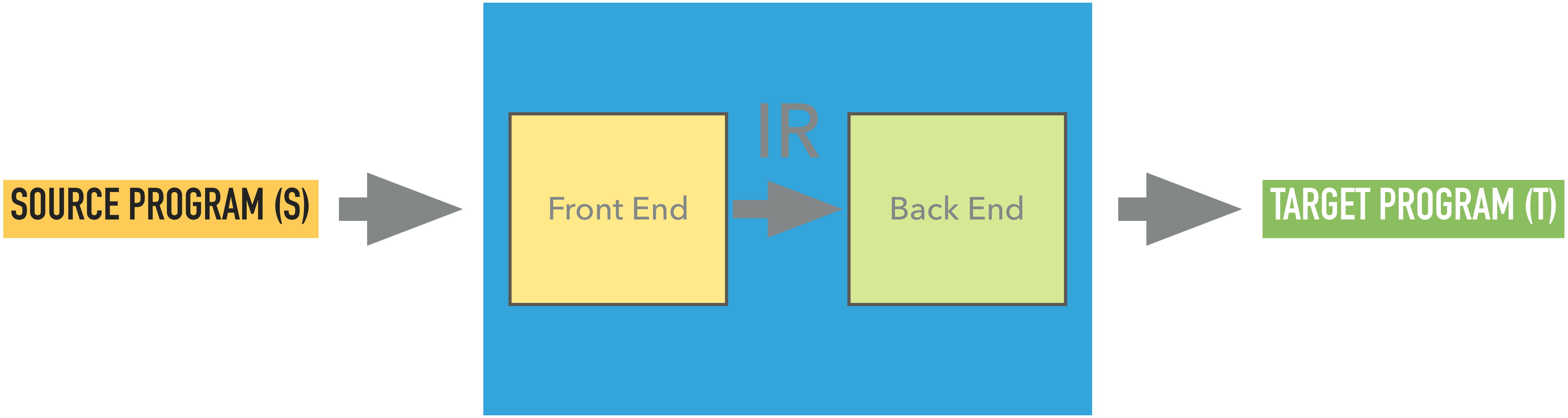
- What we could do with code would be drastically limited by the time taken to write it if we needed to develop it in pure machine code.

WHAT ARE COMPILERS IN A BASIC SENSE



- A compiler is:
  - A recogniser of language S
  - A translator from S to T
  - A program in language H

LITTLE MORE COMPLICATED



- Front End : recognise source code S; map S - IR
- IR : intermediate representation
- Back End = map IR to T

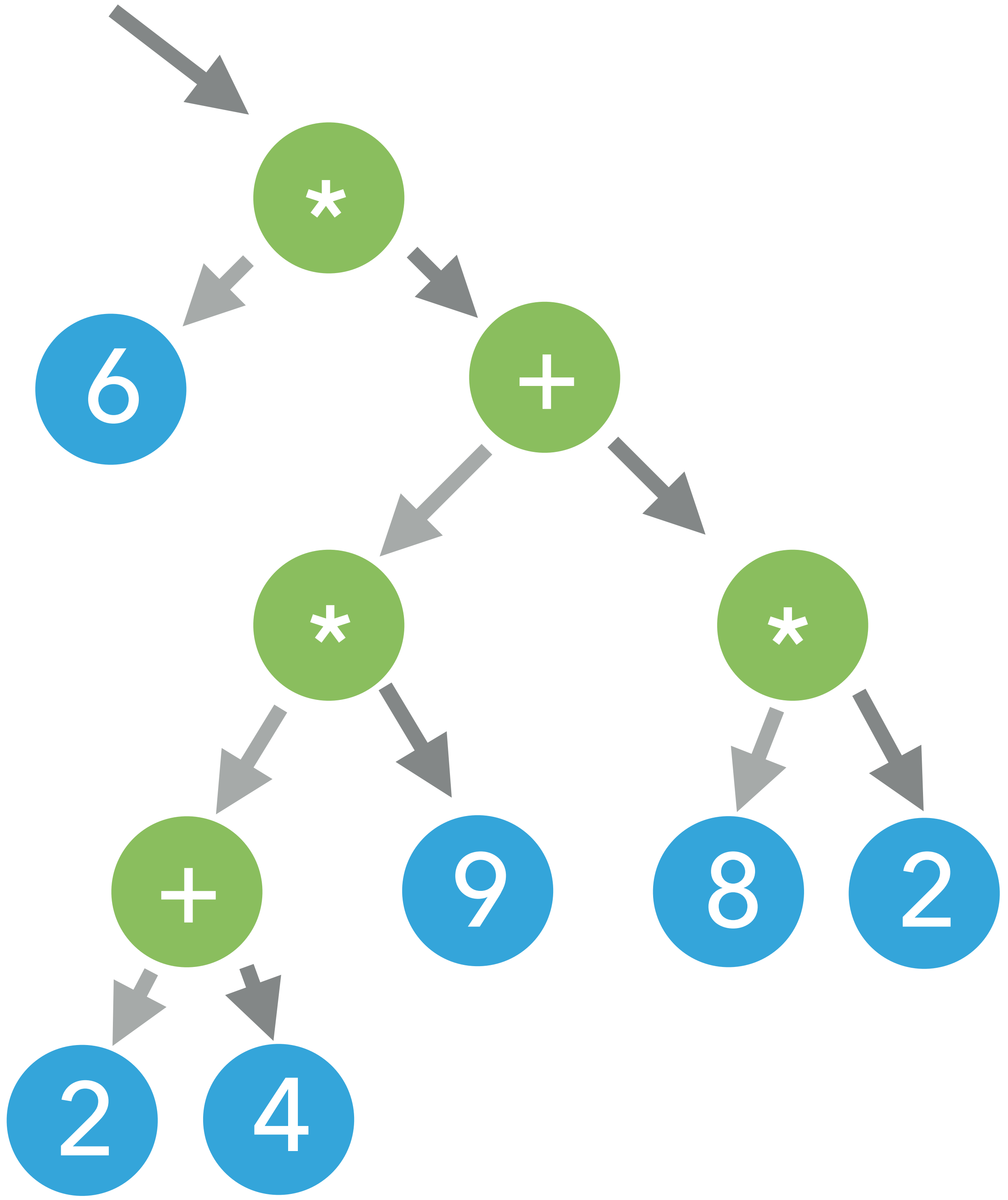
INTERPRETATION VS COMPILATION

- Interpreter -
  - Translates program one statement at a time
  - No intermediate object is generated
  - Continues until an error is met
- Compiler -
  - Scans entire program and translates it into code or executable
  - Intermediate code
  - Generates error message after whole process
  - Scans the entire program and translates it as a whole into machine cod

IMAGINE THE FOLLOWING

- **(6 * (((2+4) * 7) + (8*2)))**
- How do we evaluate it?

(6 * (((2+4) * 7) + (8*2)))



SO BASICALLY OUR SEQUENCE IS

- Start at the top
- Look at the operator
    - Evaluate all down the left until a value is reached
    - Evaluate all down the right until a value is reached
    - When you have 2 values, perform the operator and pass the result back up?

INTERPRETATION

```
int EvalTree(Tree T) {
  switch(T.oper) {
    case ‘*’:
      l = EvalTree(T.leftChild);
      r = EvalTree(T.rightChild);
      return l * r; break;
    case ‘+’:
      l = EvalTree(T.leftChild);
      r = EvalTree(T.rightChild);
      return l + r; break;
    default: // Leaf: T holds an int
    return T.value; break;
  }
}
```

```
EvalTree(6 * (((2+4) * 7) + (8 * 2)))
|  EvalTree(6)
|  EvalTree(((2+4) * 7) + (8 * 2))))
|  |  EvalTree((2+4) * 7)
|  |  |  EvalTree(2+4)
|  |  |  |  EvalTree(2)
|  |  |  |  EvalTree(4)
|  |  |  2 + 4 = 6
|  |  |  EvalTree(7)
|  |  6 * 7 = 42
|  |  EvalTree(8 * 2)
|  |  |  EvalTree(8)
|  |  |  EvalTree(2)
|  |  8 * 2 = 16
|  42 + 16 = 58
T 6 * 58 = 348
```

COMPILATION

```
instSeq GenInstSeq(Tree T) {
  switch(T.op) {
    case ‘*’:
      lSeq = GenInstSeq(T.leftChild);
      rSeq = GenInstSeq(T.rightChild);
      return lSeq || rSeq || Multiply;
    case ‘+’:
      lSeq = GenInstSeq(T.leftChild);
      rSeq = GenInstSeq(T.rightChild);
      return lSeq || rSeq || Add;
    default: // Leaf: T holds an int
      return InstSeq(“Push”, T.value);
  }
}
```

```
GenInstSeq(6 * (((2+4) * 7) + (8 * 2)))
|  GenInstSeq(6) [Push(6)]
|  GenInstSeq(((2+4) * 7) + (8 * 2))))
|  |  GenInstSeq((2+4) * 7)
|  |  |  GenInstSeq(2+4)
|  |  |  |  GenInstSeq(2) [Push(2)]
|  |  |  |  GenInstSeq(4) [Push(4)]
|  |  |  [Push(2); Push(4); Add]
|  |  |  GenInstSeq(7) [Push(7)]
|  |  [Push(2); Push(4); Add; Push(7); Multiply]
|  |  GenInstSeq(8 * 2)
|  |  |  GenInstSeq(7) [Push(8)]
|  |  |  GenInstSeq(2) [Push(2)]
|  |  [Push(8); Push(2); Multiply]
|  [Push(2); Push(4); Add; Push(7); Multiply;
   Push(8); Push(8); Multiply; Add]
```

= [Push(6); Push(2); Push(4); Add; Push(7); Multiply; Push(8); Push(2); Multiply; Add; Multiply]

INSTRUCTION SEQUENCE

```
[Push(6);
Push(2);
Push(4);
Add;
Push(7);
Multiply;
Push(8);
Push(2);
Multiply;
Add;
Multiply]
```

EXECUTION?

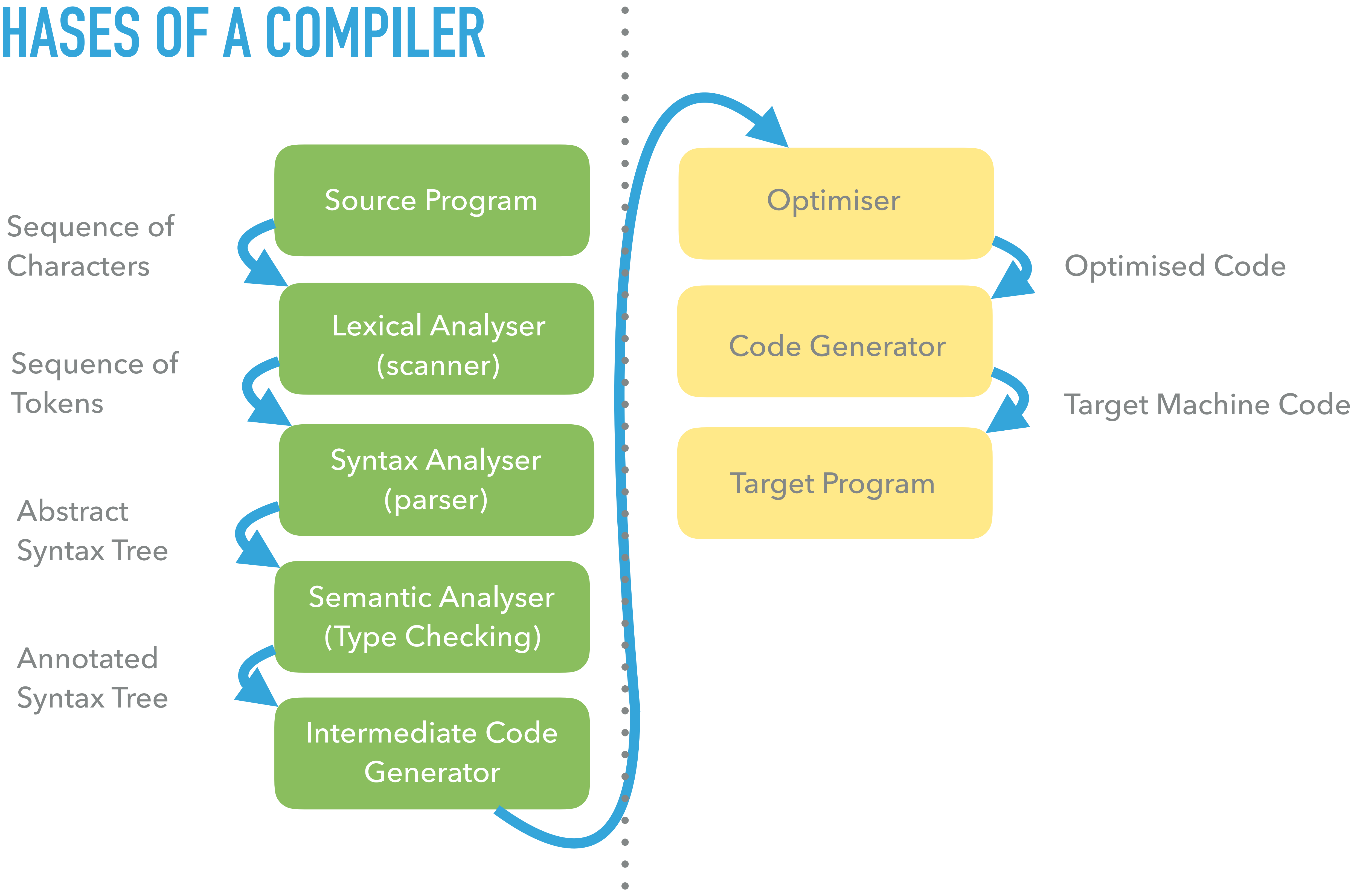
```
Stack S = EmptyStack;
int EvalInstSeq(InstSeq IS) {
   while(!null(IS)) {
      inst = IS.head;
      switch(inst.opCode) {
         case Multiply:
            r = S.pop();
            l = S.pop();
            S.push(l * r);
            break;
         case '+':
            r = S.pop();
            l = S.pop();
            S.push(l + r);
            break;
         default: // Push(…)
            S.push(inst.argument);
            break;
      }
      IS = IS.tail;
   }
return S.pop();
}
```

```
EVALINSTSEQ([P(6); P(2); P(4); A; P(7); M; P(8); P(2); M; A; M])

S = [ ]
IS = [P(6); P(2); P(4); A; P(7); M; P(8); P(2); M; A; M]
S = [6]
IS = [P(2); P(4); A; P(7); M; P(8); P(2); M; A; M]
S = [6, 2]
IS = [P(4); A; P(7); M; P(8); P(2); M; A; M]
S = [6, 2, 4]
IS = [A; P(7); M; P(8); P(2); M; A; M]
S = [6, 6]
IS = [P(7); M; P(8); P(2); M; A; M]
S = [6, 6, 7]
IS = [M; P(8); P(2); M; A; M]
S = [6, 42]
IS = [P(8); P(2); M; A; M]
S = [6, 42, 8]
IS = [P(2); M; A; M]
S = [6, 42, 8, 2]
IS = [M; A; M]
S = [6, 42, 16]
IS = [A; M]
S = [6, 58]
IS = [M]
S = [348]
IS = [ ]
```

Operation acts on previous two values in the stack

PHASES OF A COMPILER



SCANNER

- INPUT: takes characters from source code
- OUTPUT : a sequence of tokens
- What it does:
  - groups characters into recognised tokens
  - Identify and ignore whitespace
- Performs Error Checking :
  - eg bad characters
  - Unterminated strings “BigBaddaBoom

EXAMPLE

x = 4 * y + rnd(5);

identifier	assignment	int literal	multiply	identifier	plus	identifier	lbracket	int literal	rbracket	terminal
x	=	4	*	y	+	rnd	(	5	)	;

```
x =        4 *
   y + rnd(
      5
     ) ;
```

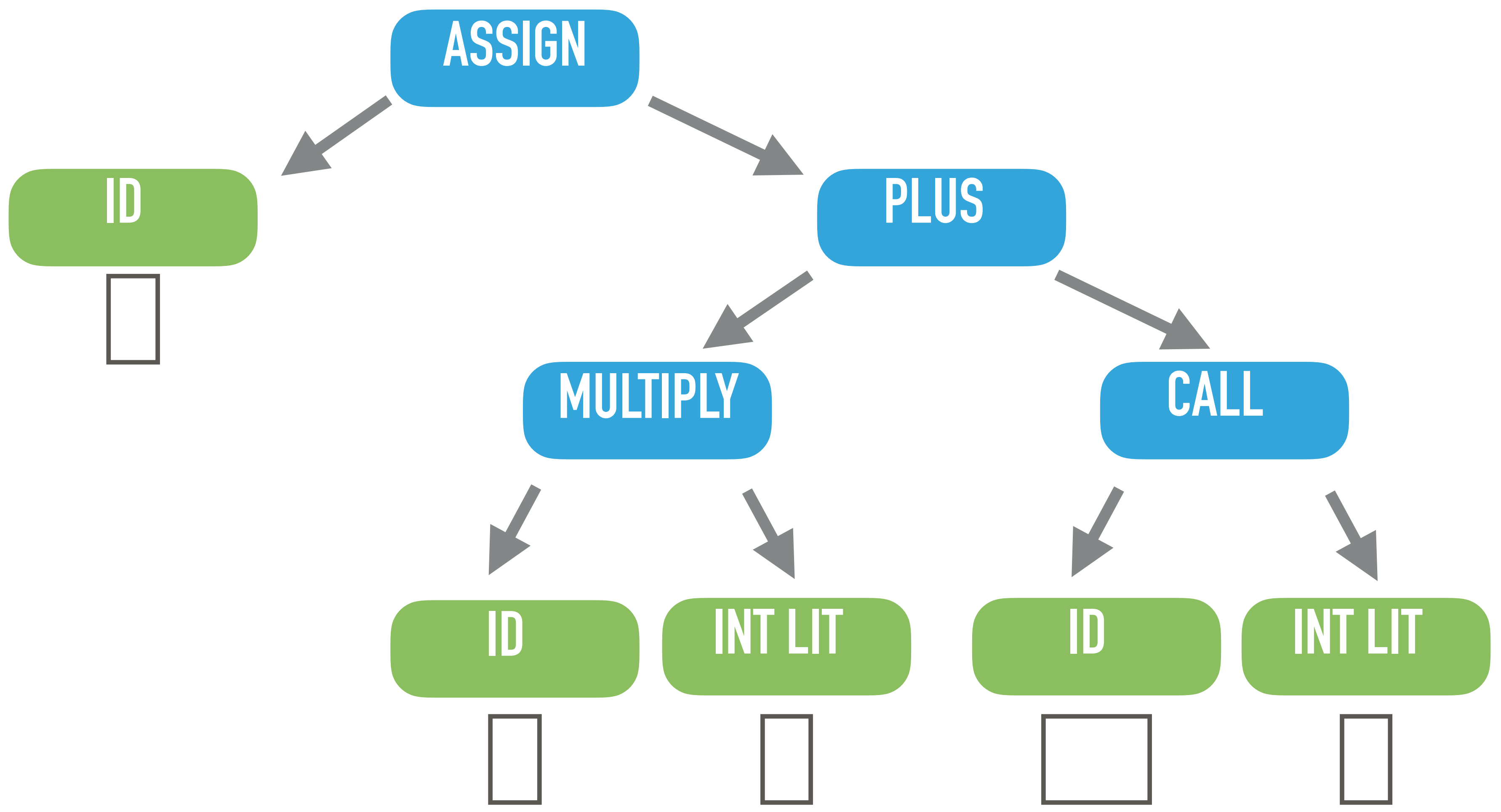
identifier	assignment	int literal	multiply	identifier	plus	identifier	lbracket	int literal	rbracket	terminal
x	=	4	*	y	+	rnd	(	5	)	;

PARSER

- INPUT: sequence of tokens from the scanner
- OUTPUT : AST (Abstract Syntax Tree)
- What it does:
  - groups tokens in to sentences (instructions)
- Performs Error Checking :
  - Syntax Errors, e.g x= y *= 5
  - Can check some semantics, eg x is not declared

EXAMPLE

x = 4 * y + rnd(a);



SEMANTIC ANALYSER

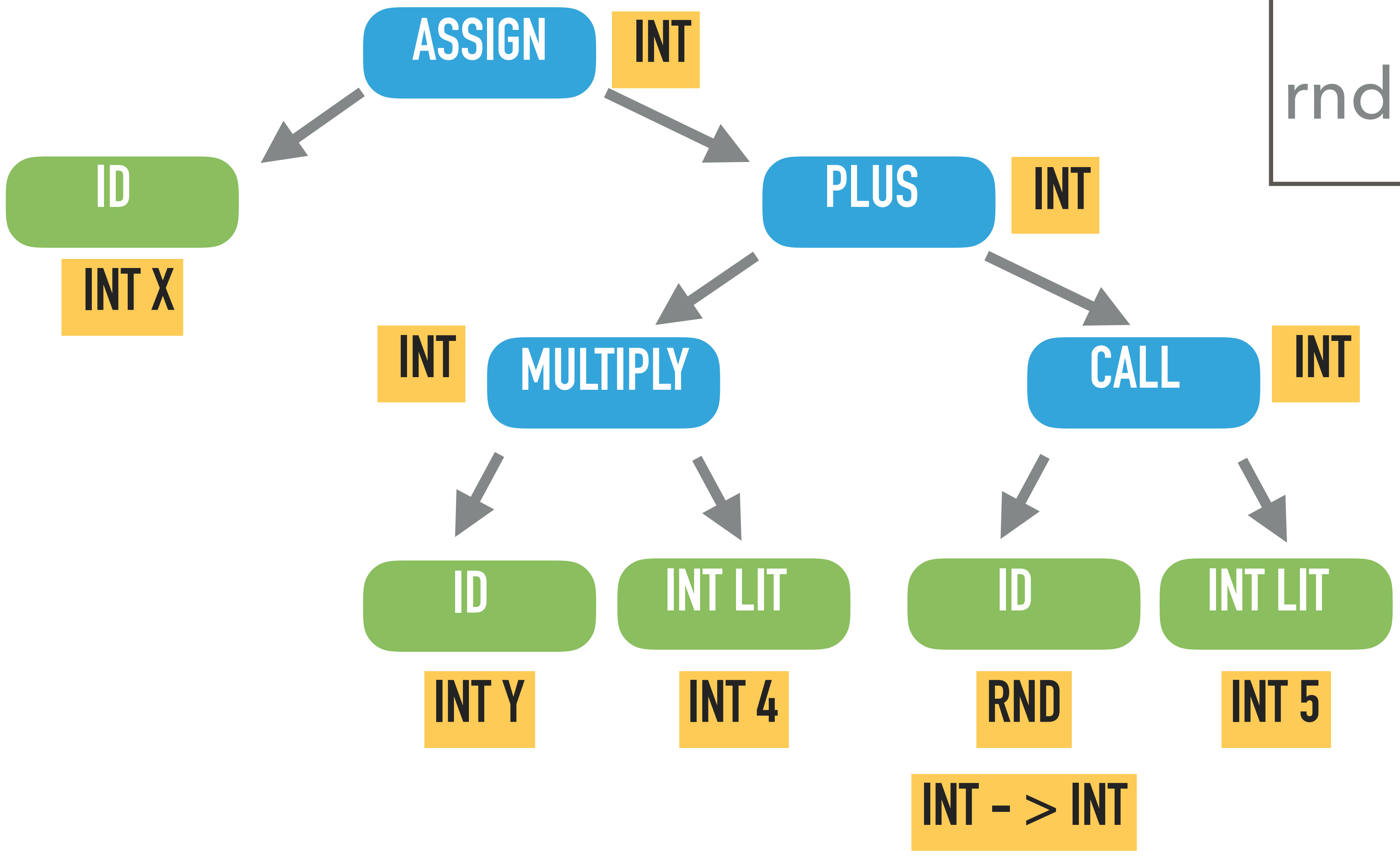
- INPUT: AST
- OUTPUT : Annotated AST
- What it does:
    - Semantic checks
    - checks declarations and use of variables
    - enforces scope
    - checks types

SYMBOL TABLE

- Compiler keeps track of names for a number of process during compilation
  - semantic analyser - names & type checking
  - code generation - names in stack

EXAMPLE

x = 4 * y + rnd(a);



SYMBOL TABLE
x variable int
y variable int
rnd function int -> int

EXAMPLE

Semantics - Is written correctly but is meaningless (something that looks right but isn't)

Scope Example

```
{
    int i = 42;
    i++;
}
 i = 50;
```

IN SCOPE

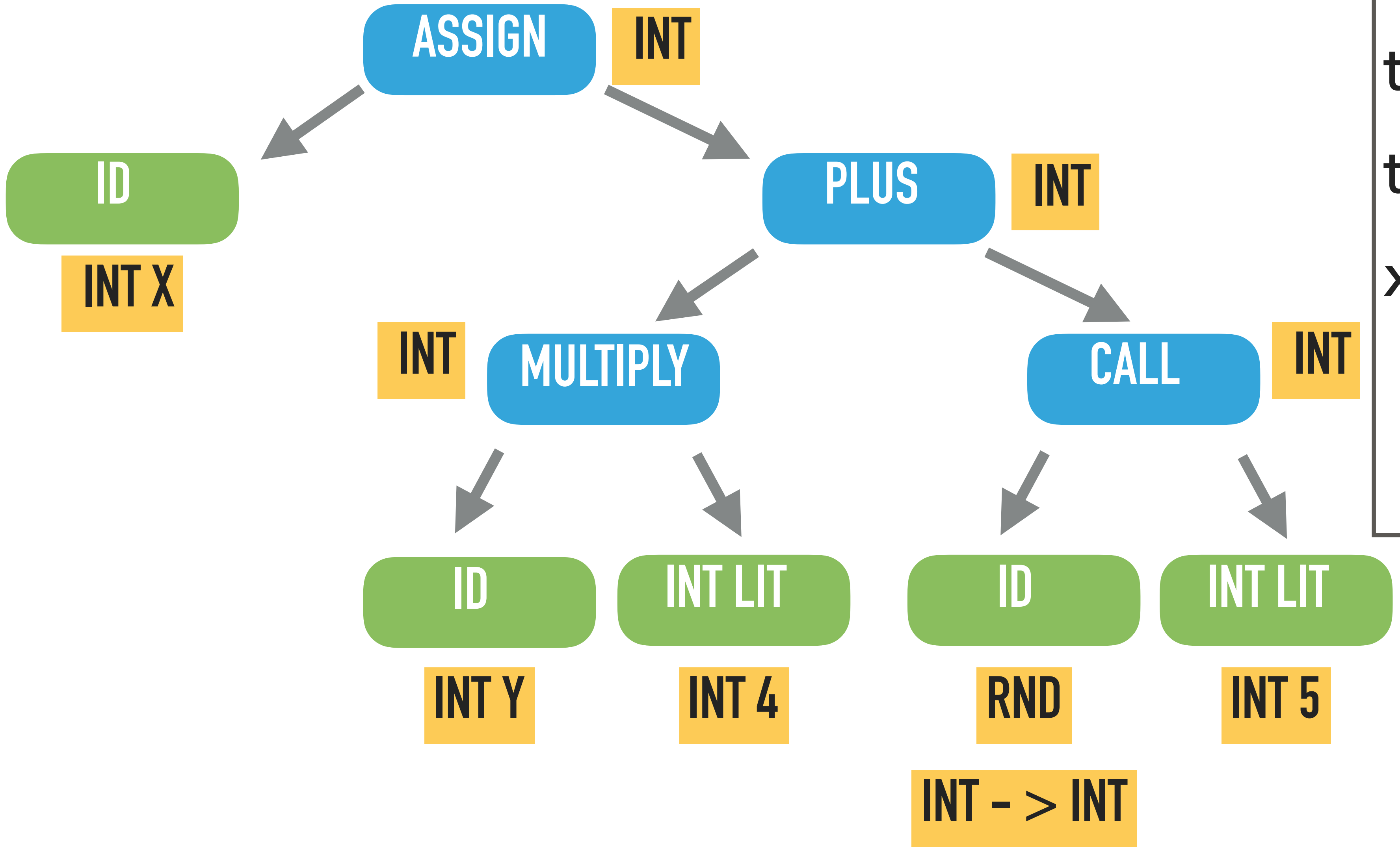
OUT OF SCOPE

INTERMEDIATE CODE GENERATOR

- INPUT: annotated AST (no errors from sec analyser)
- OUTPUT : Intermediate Representation (IR)
- What it does:
  - e.g instructions have 2 values and an operand
  - generated from AST
  - 1 instruction per node (line)

EXAMPLE

x = 4 * y + rnd(a);



IR CODE

```
param1 = 5
call rnd
move return1 temp1
temp2 = 4 * y
temp3 = temp2 + temp1
x = temp3
```

OPTIMISER

- INPUT: IR
- OUTPUT : Optimised IR
- What it does:
  - Improve code
  - faster, smaller, better
  - local and global optimisation

FINAL CODE GENERATOR

- INPUT: IR from Optimiser
- OUTPUT : Target Code
- What it does:
  - Similar to before but code generated based on new optimised code.

SUMMARY

- The compiler is actually a set of different components, each of which performs it's own specific job
- If one part does not function correctly there will be a knock on effect across the system
- Most sections perform some form of error checking and reporting.