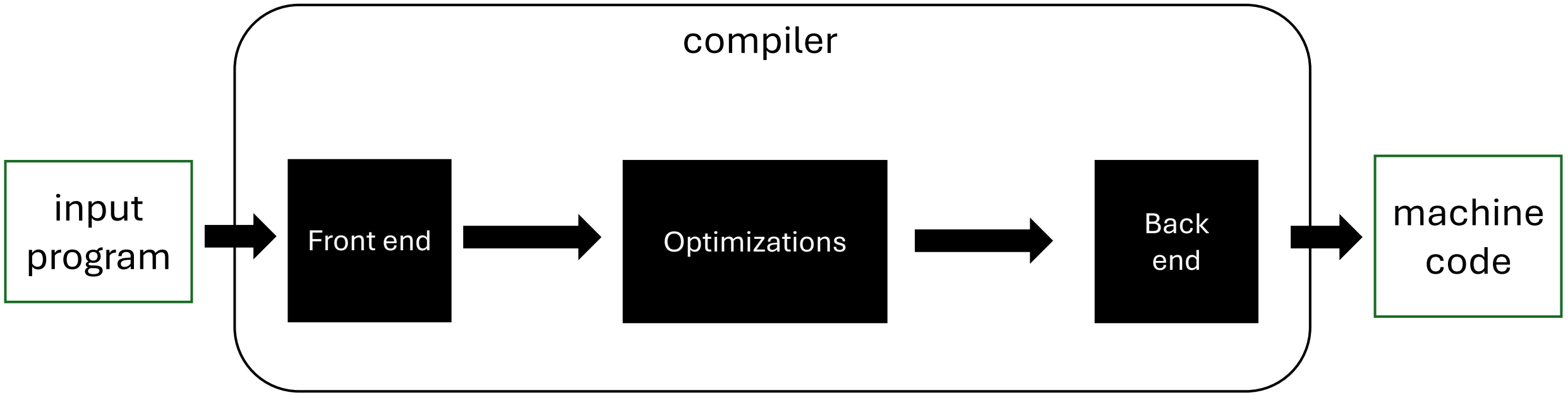
Journey into a Compiler

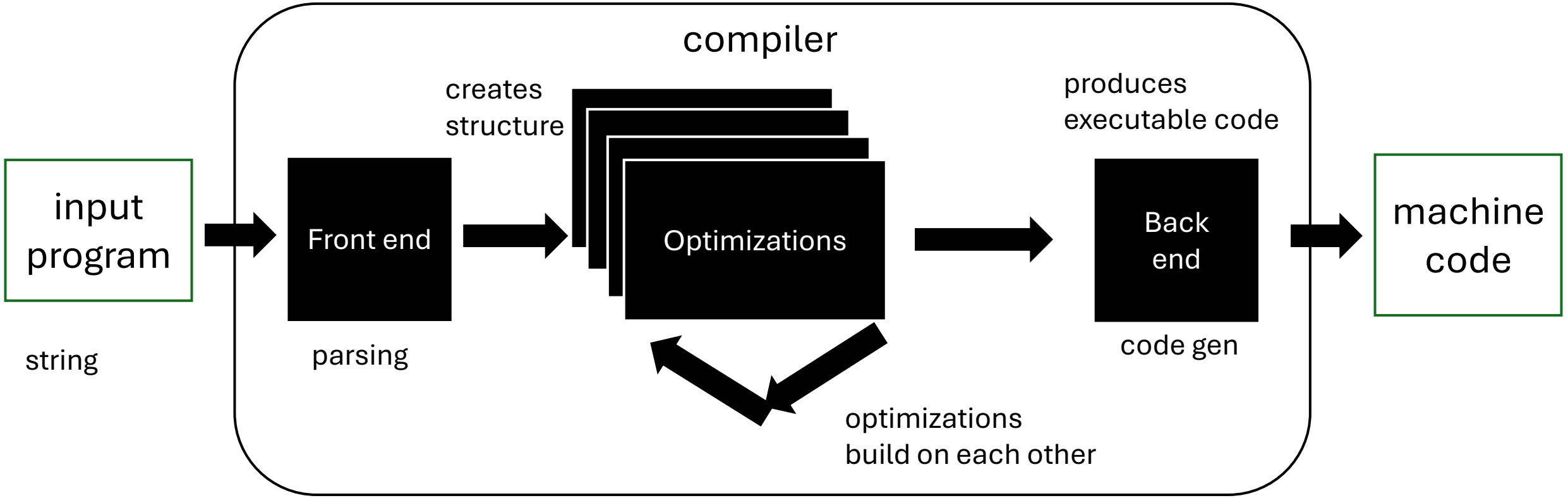
ref; 2015 movie *In the Heart of the Sea*

Compiler Architecture



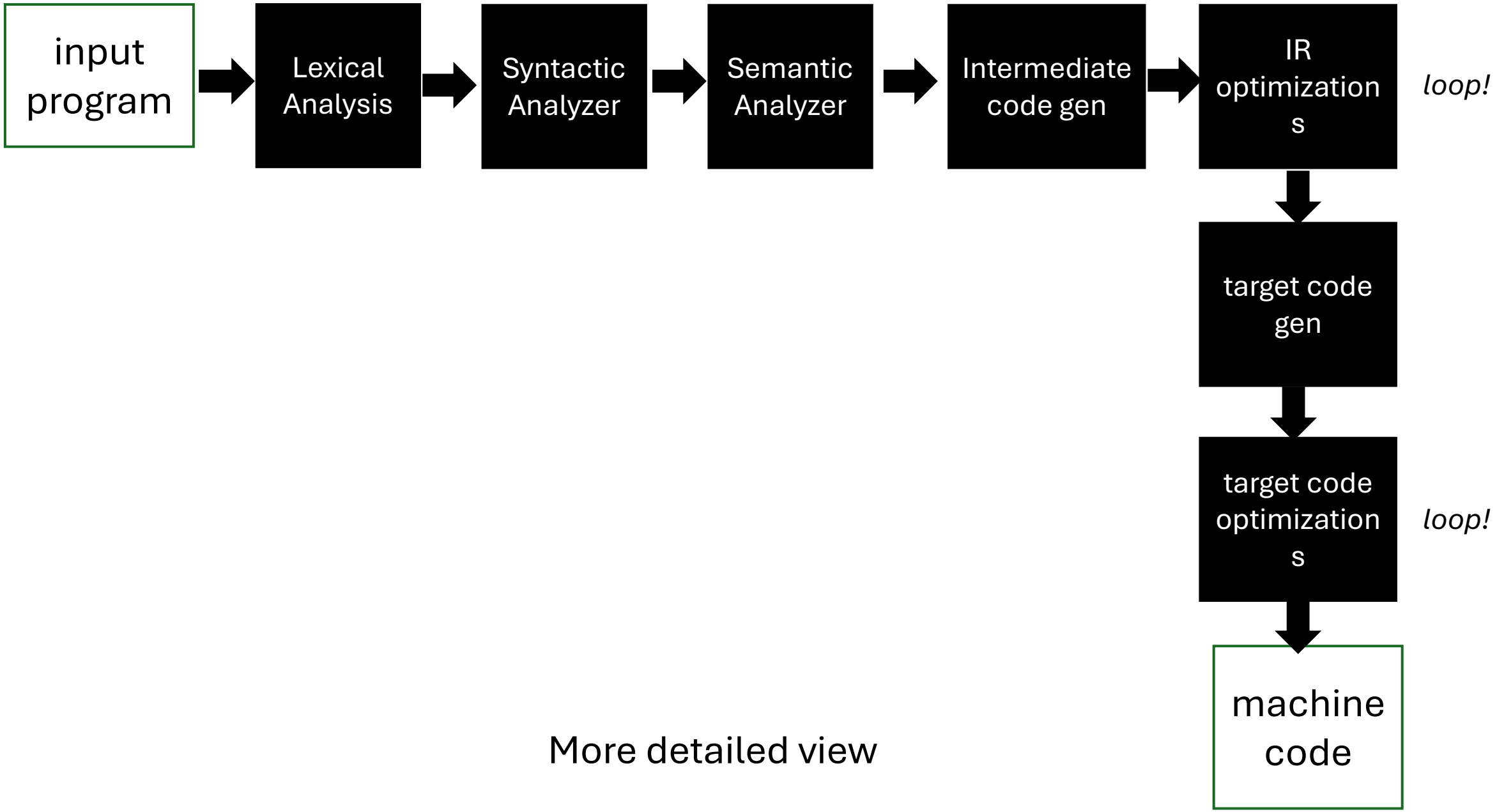
Medium detailed view

Benefits of Modular Compiler Design

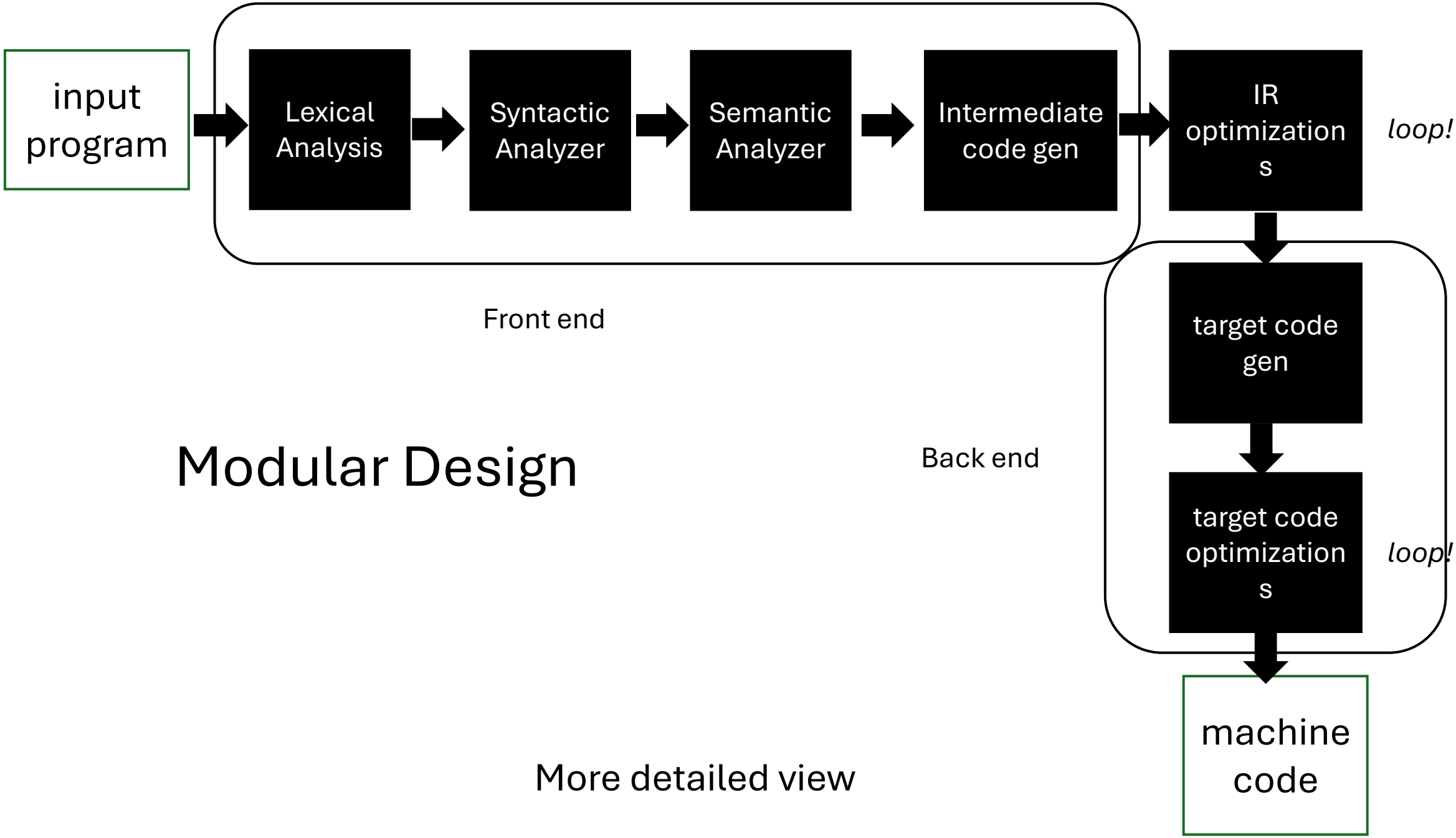


Medium level view of homogeneous optimizing compiler

more about optimizations: https://stackoverflow.com/questions/15548023/clang-optimization-levels



More detailed view

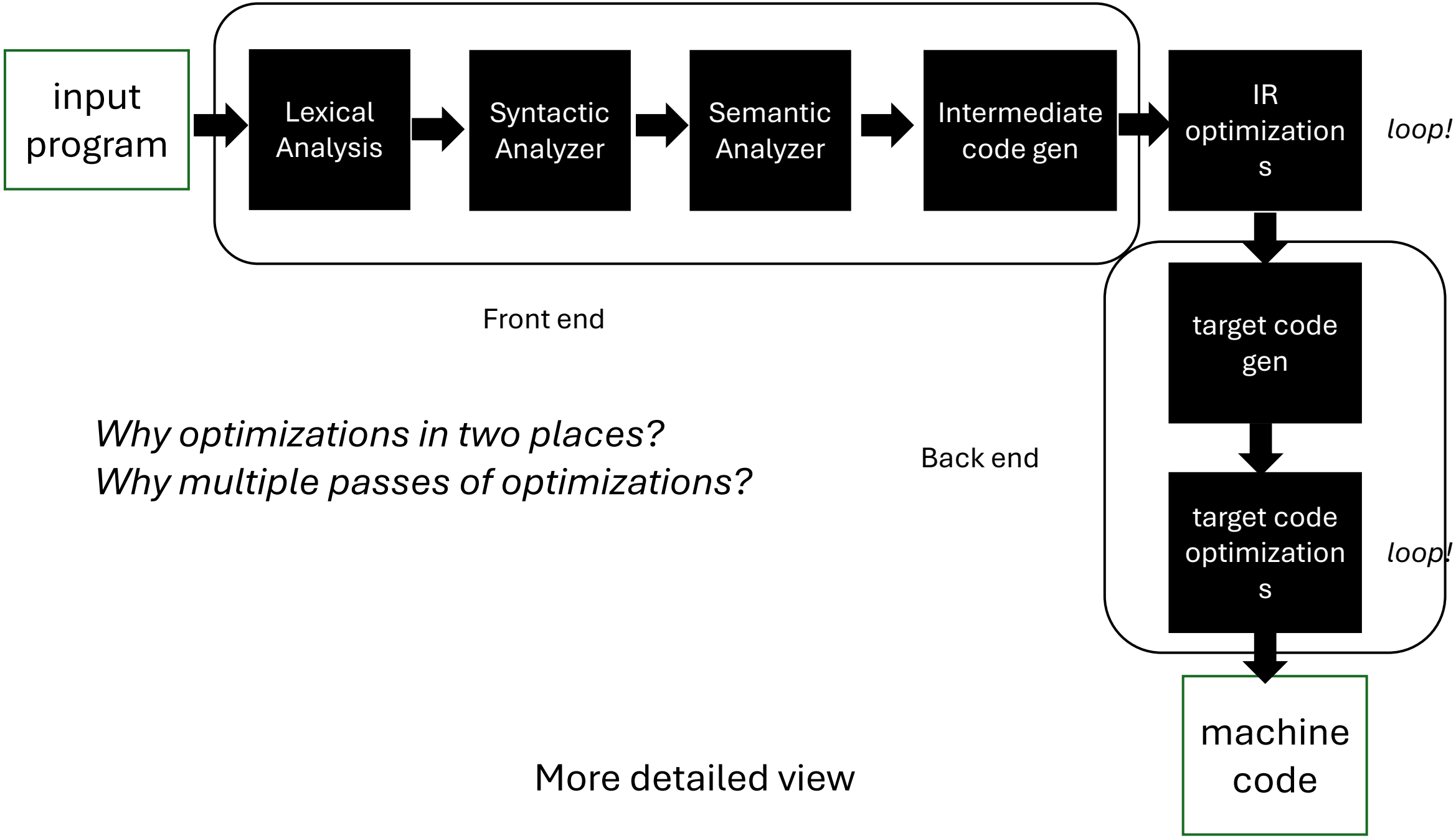


Modular Design

More detailed view

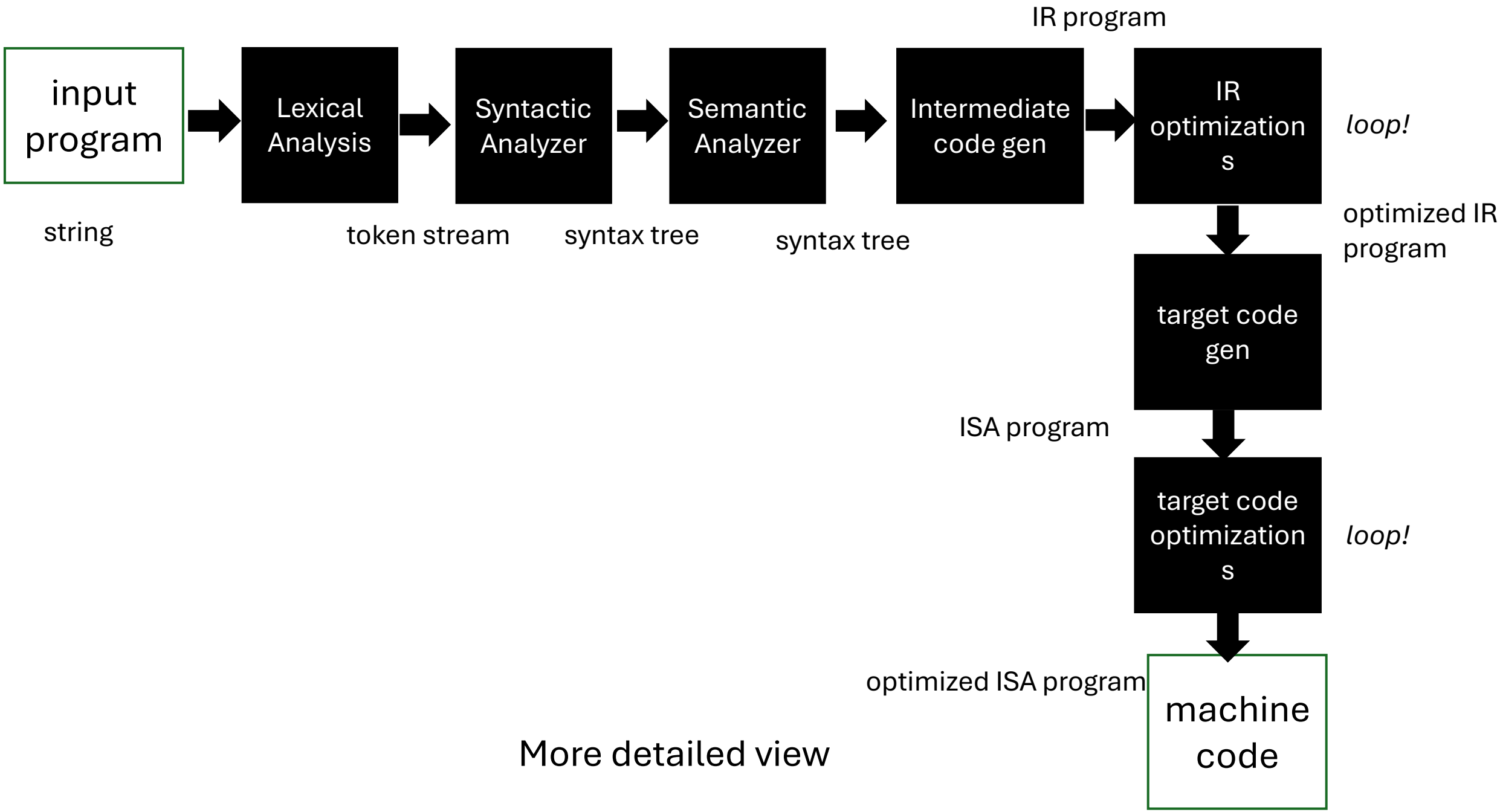
LLVM Modular Modern Compiler Infrastructure

- Front ends:
  - clang -> c
  - clang++ -> c++
  - Many others (rust, Swift, Julia, etc.)
- LLVM Intermediate Representation (IR):
  - easy to analyze
  - Optimize
  - Targe Independent
  - 3 forms:
    - Textual (.ll),
    - Binary bitcode (.bc),
    - In-memory representation
- Backends
  - X86, ARM, MIPS, RISC-V, many more

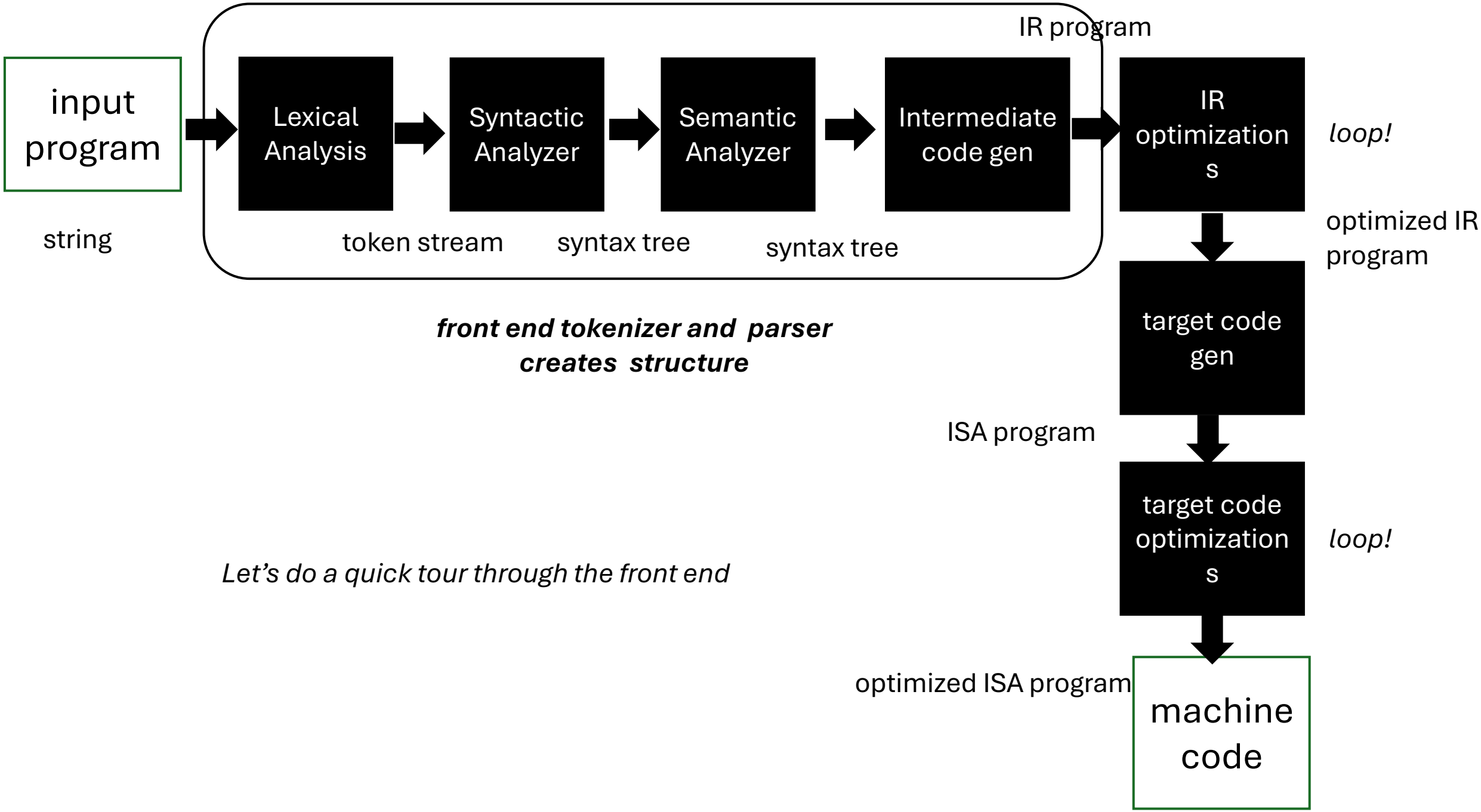


Why optimizations in two places?
Why multiple passes of optimizations?

More detailed view

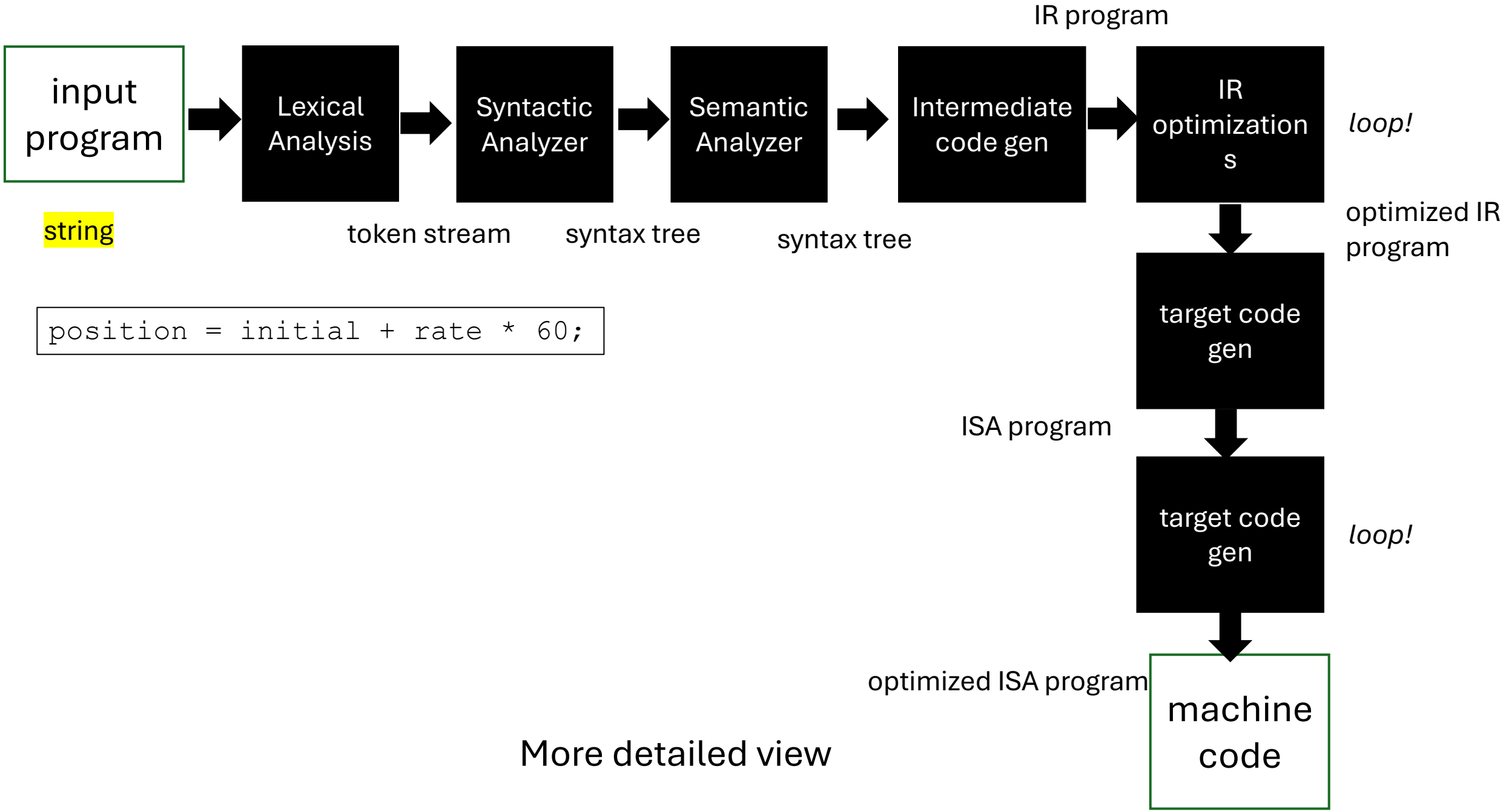


More detailed view



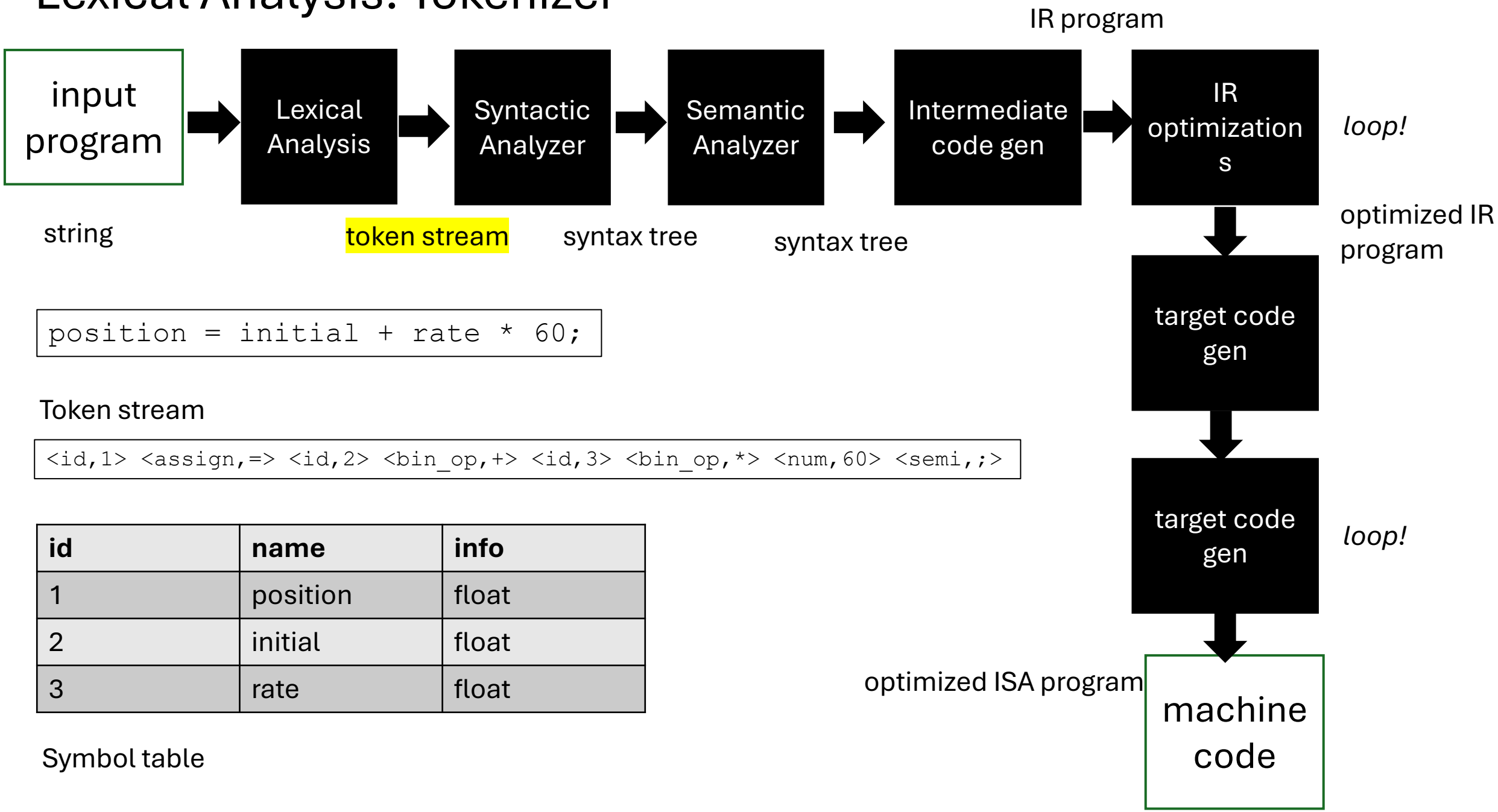
front end tokenizer and parser creates structure

Let's do a quick tour through the front end



More detailed view

Lexical Analysis: Tokenizer




```
position = initial + rate * 60;
```

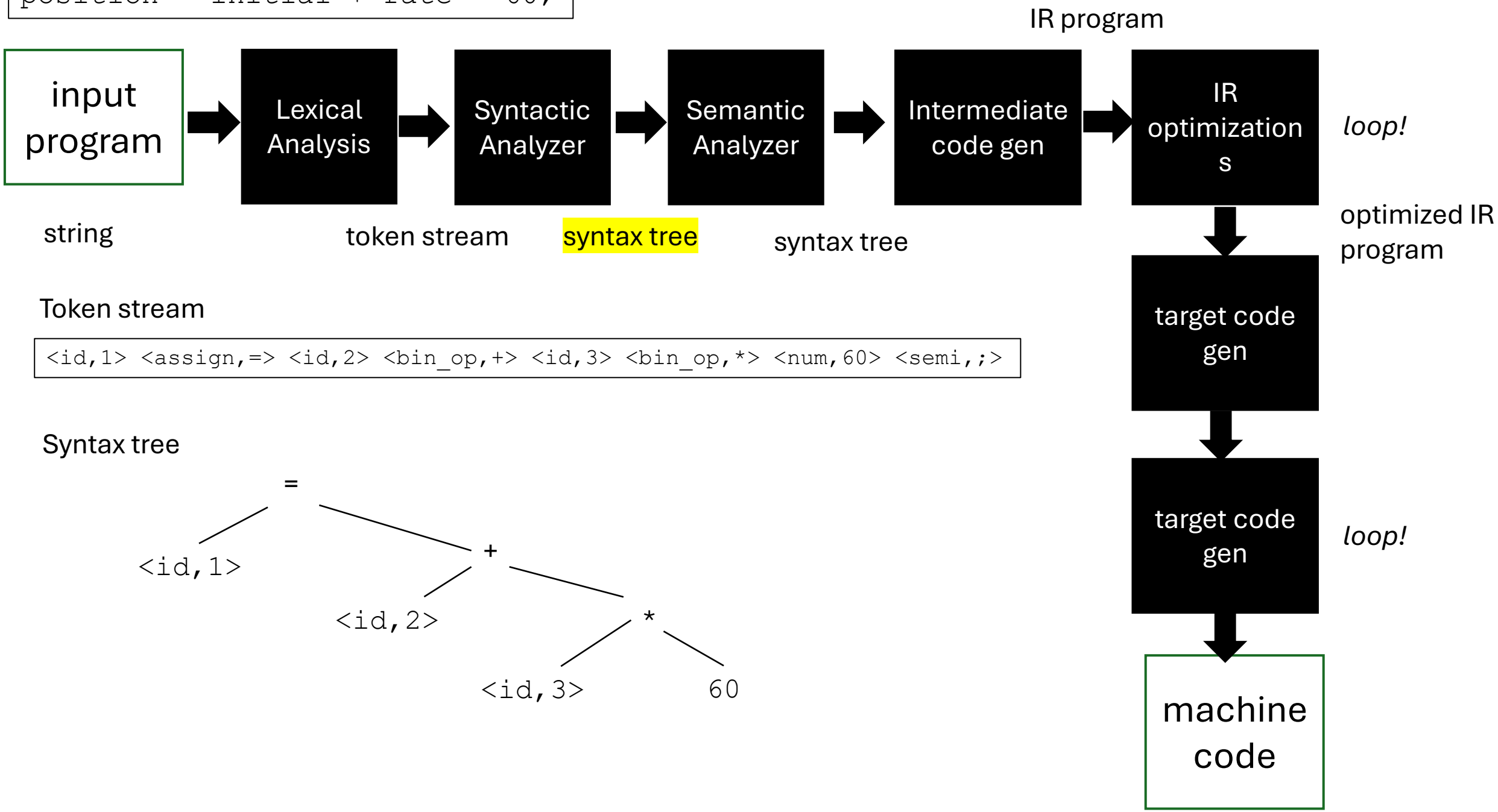
Token stream

```
<id,1> <assign,=> <id,2> <bin_op,+> <id,3> <bin_op,*> <num,60> <semi,;>
```

id	name	info
1	position	float
2	initial	float
3	rate	float

Symbol table

```
position = initial + rate * 60;
```

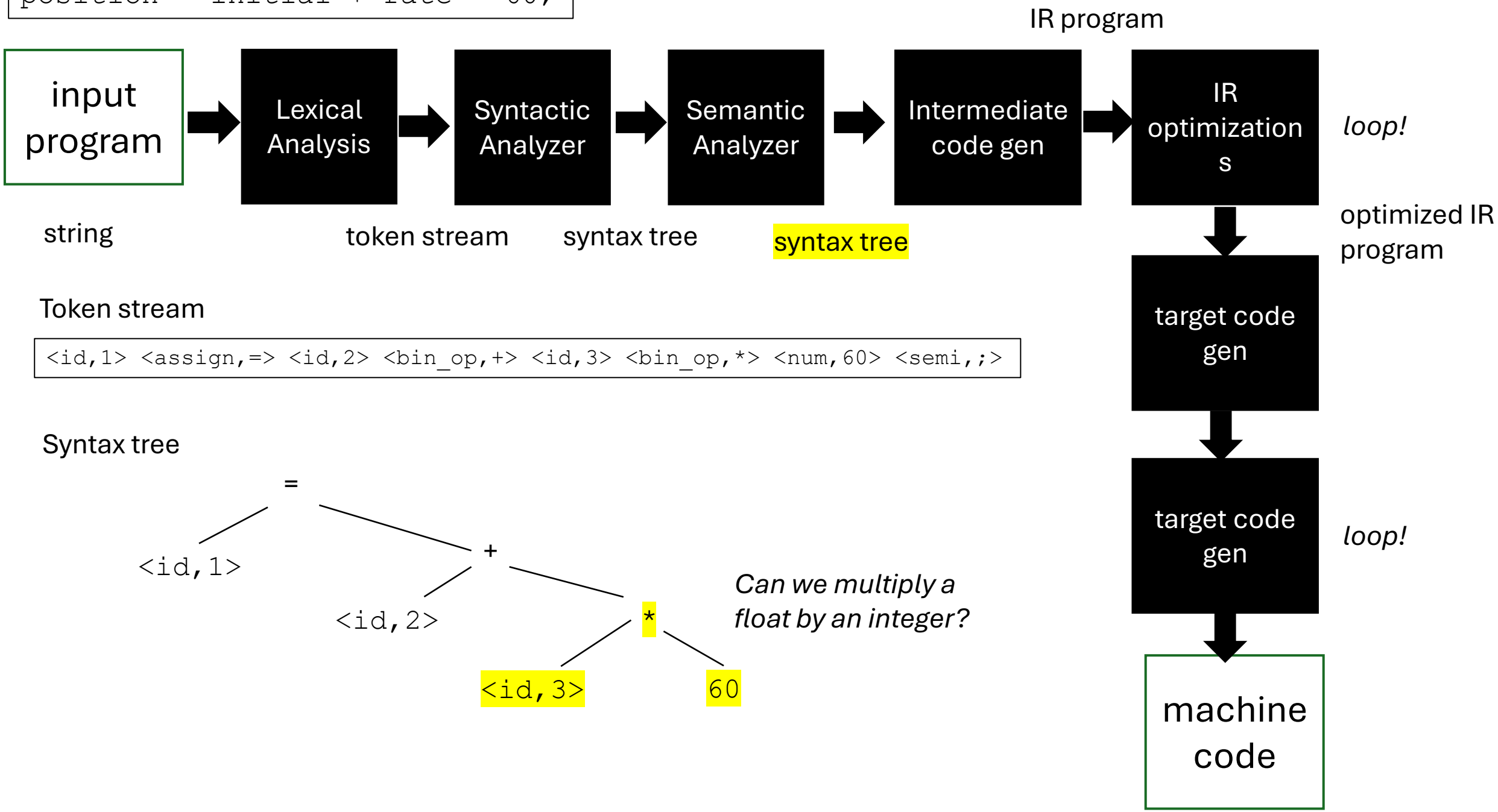


Token stream

```
<id,1> <assign,=> <id,2> <bin_op,+> <id,3> <bin_op,*> <num,60> <semi,;>
```

Syntax tree

```
position = initial + rate * 60;
```



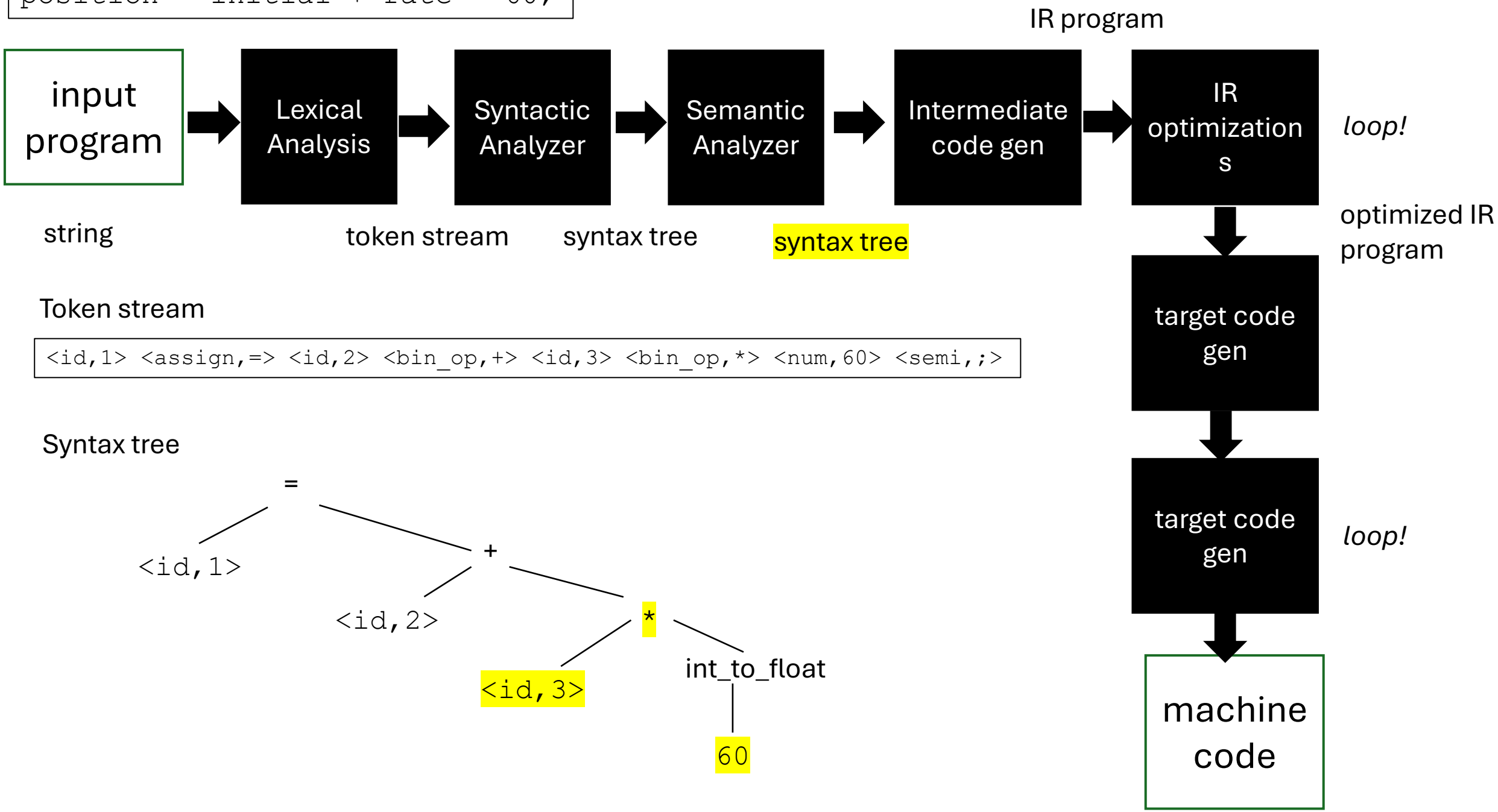
Token stream

```
<id,1> <assign,=> <id,2> <bin_op,+> <id,3> <bin_op,*> <num,60> <semi,;>
```

Syntax tree

Can we multiply a float by an integer?

```
position = initial + rate * 60;
```

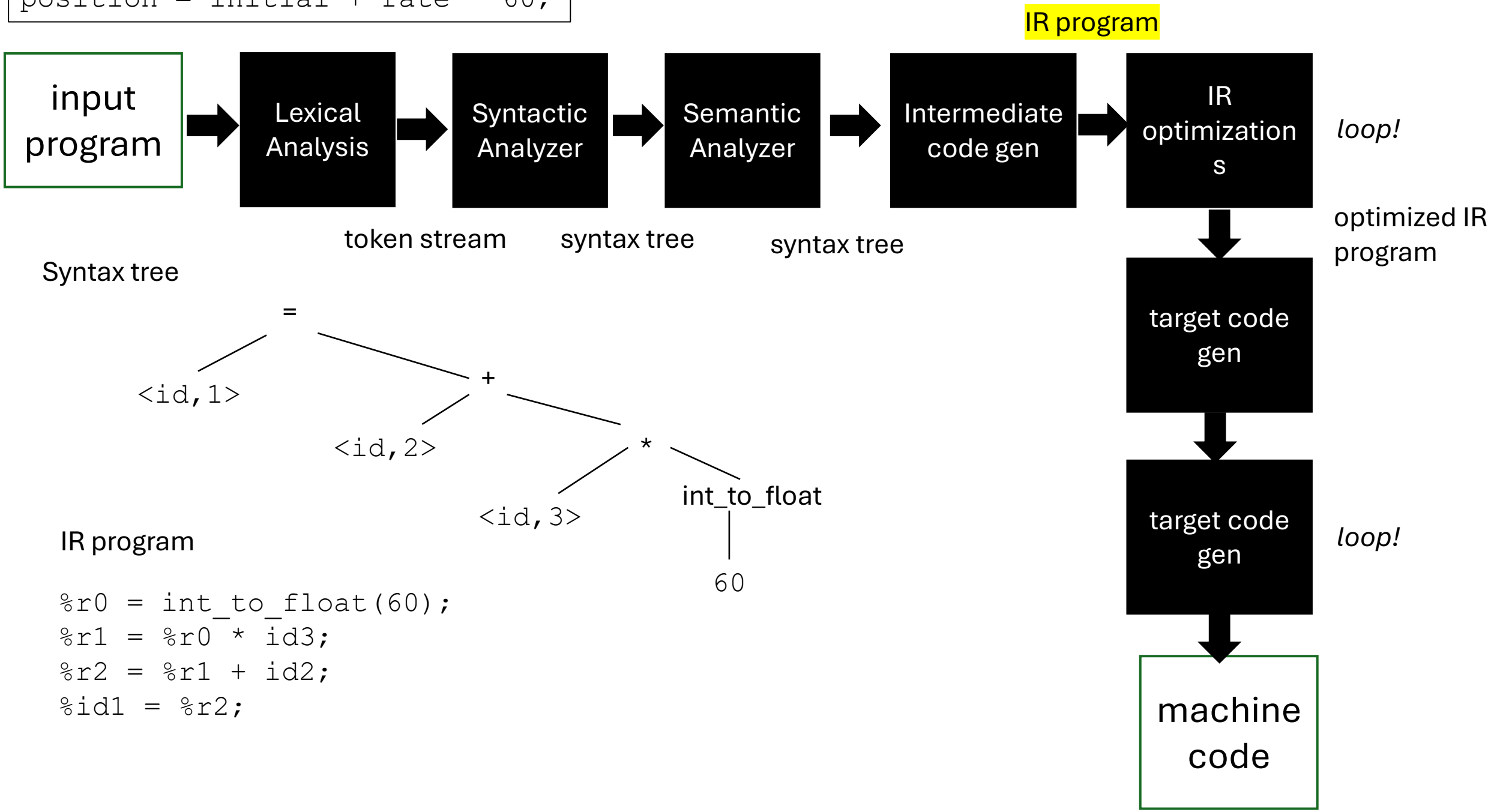


Token stream

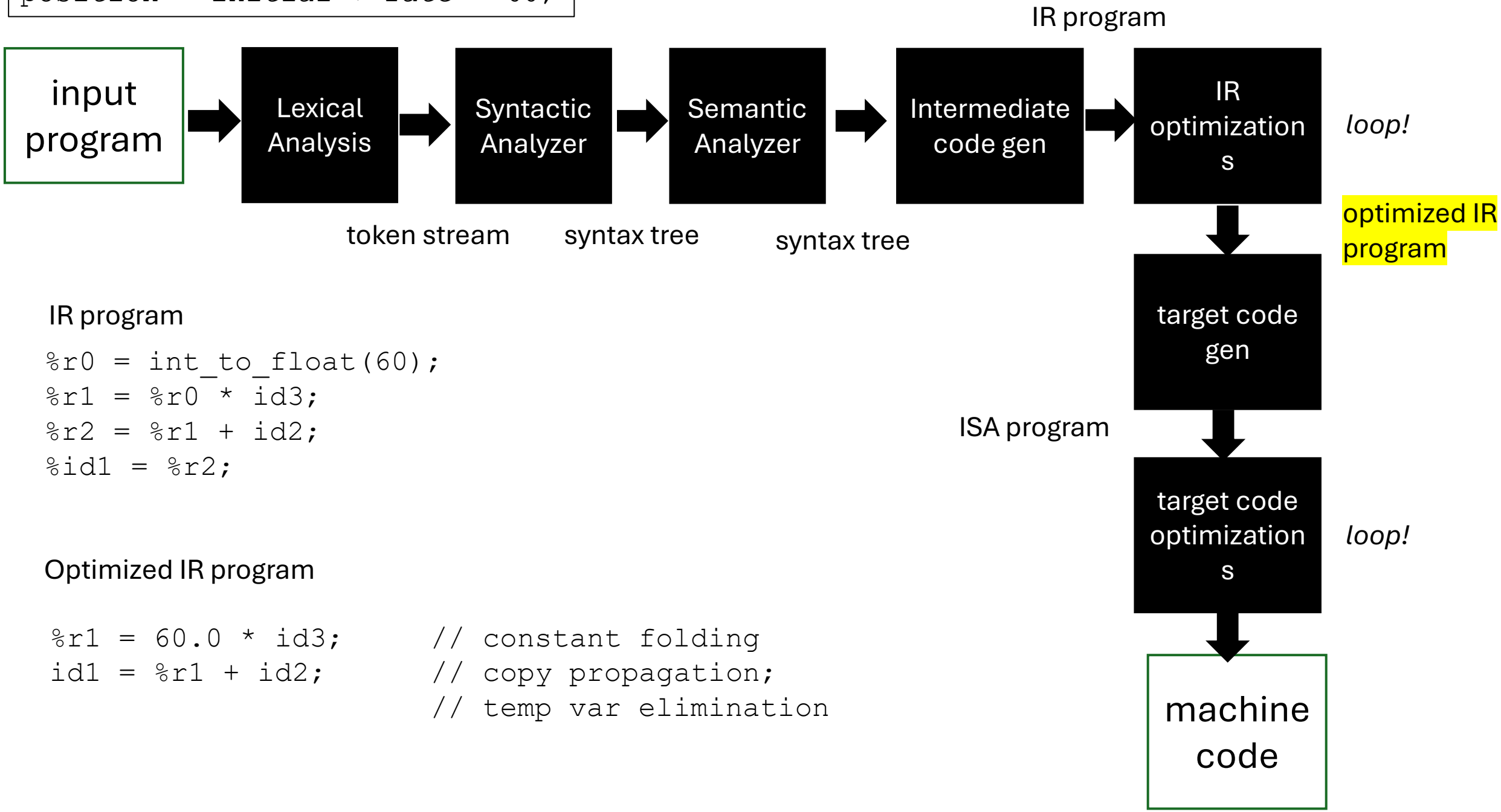
```
<id,1> <assign,=> <id,2> <bin_op,+> <id,3> <bin_op,*> <num,60> <semi,;>
```

Syntax tree

```
position = initial + rate * 60;
```



```
position = initial + rate * 60;
```



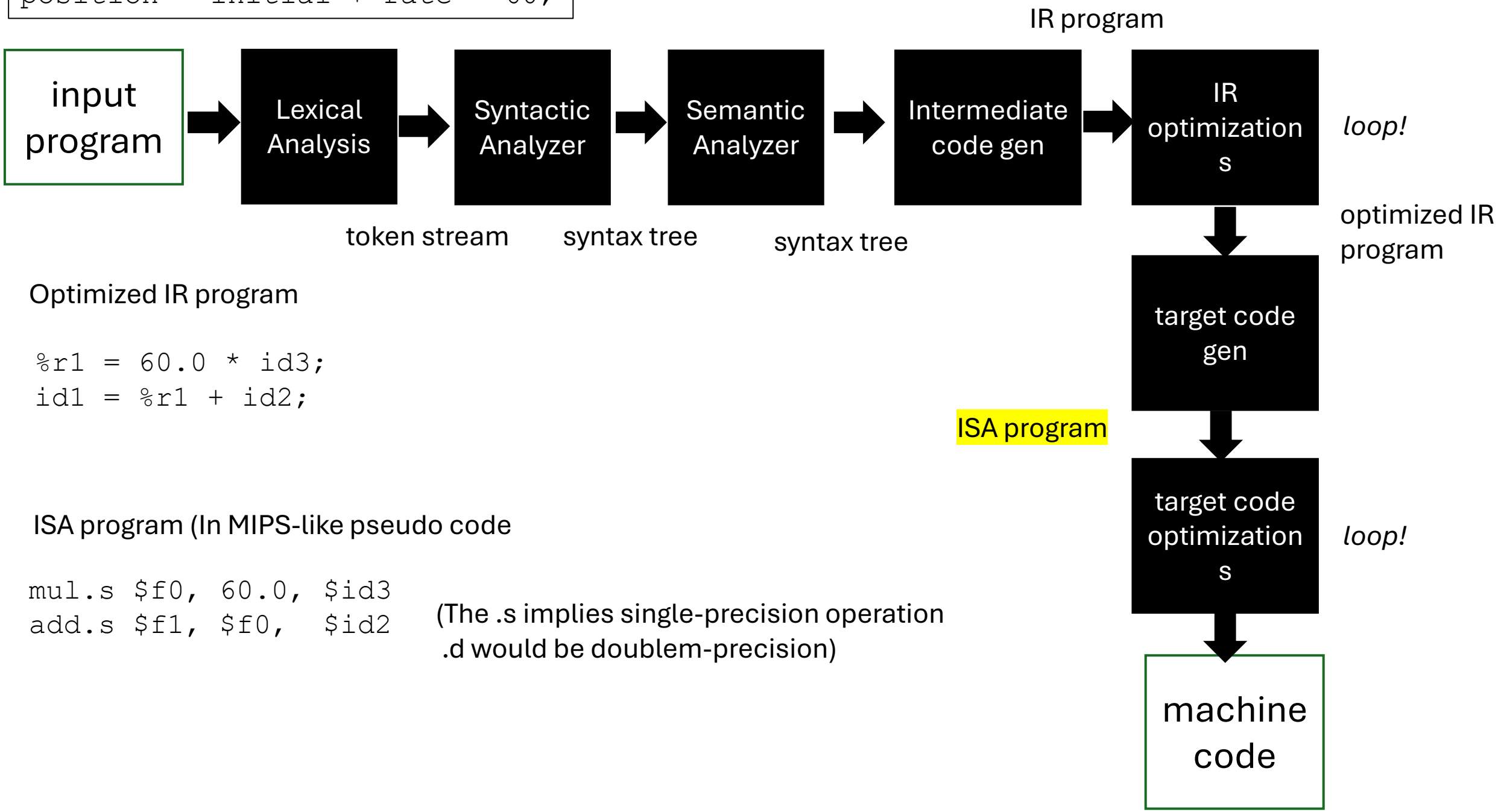
IR program

```
%r0 = int_to_float(60);
%r1 = %r0 * id3;
%r2 = %r1 + id2;
%id1 = %r2;
```

Optimized IR program

```
%r1 = 60.0 * id3;       // constant folding
id1 = %r1 + id2;        // copy propagation;
                        // temp var elimination
```

```
position = initial + rate * 60;
```



Optimized IR program

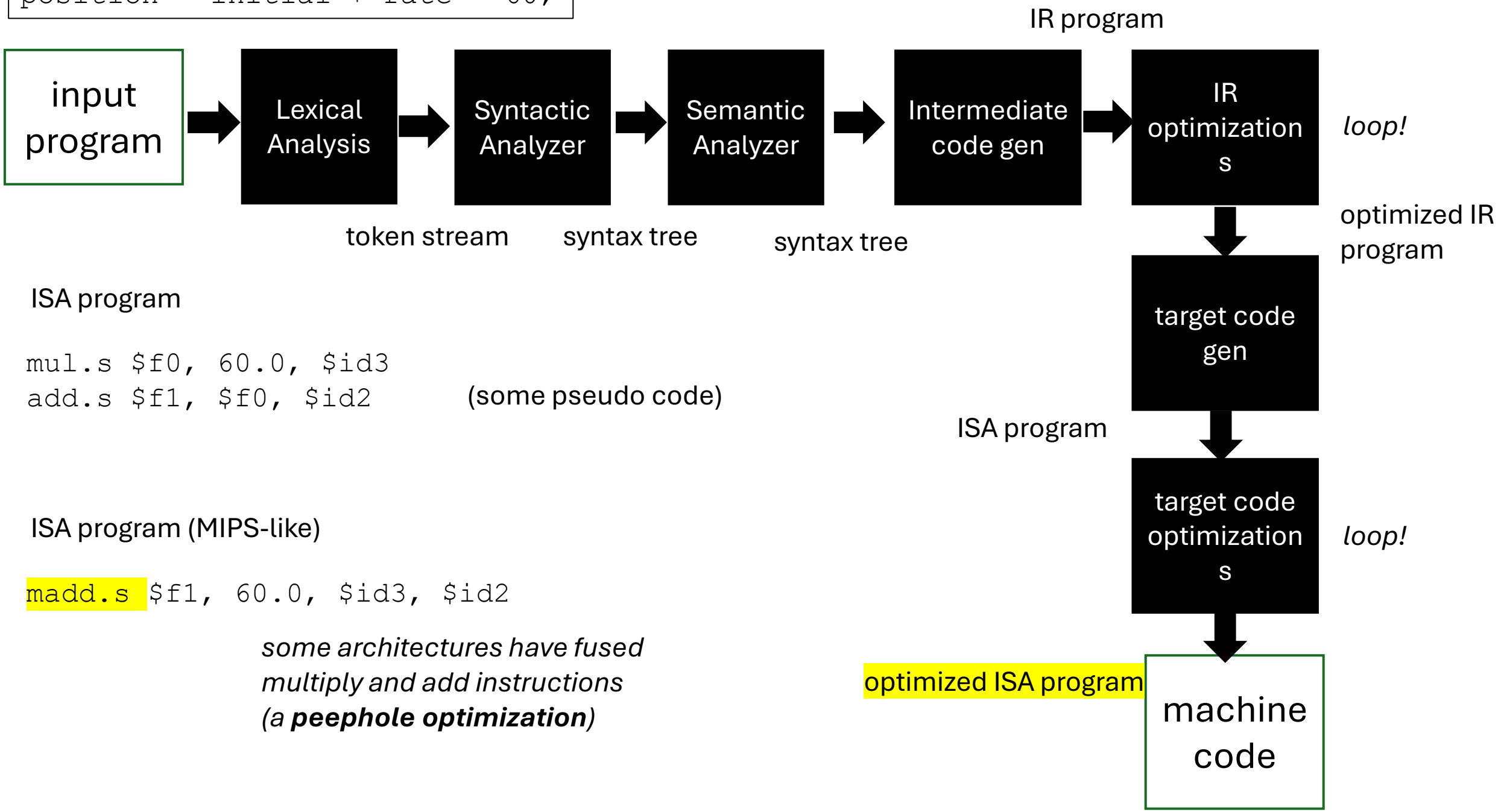
```
%r1 = 60.0 * id3;
id1 = %r1 + id2;
```

ISA program (In MIPS-like pseudo code

```
mul.s $f0, 60.0, $id3
add.s $f1, $f0,  $id2
```

(The .s implies single-precision operation
.d would be doublem-precision)

```
position = initial + rate * 60;
```



ISA program

```
mul.s $f0, 60.0, $id3
add.s $f1, $f0, $id2
```

(some pseudo code)

ISA program (MIPS-like)

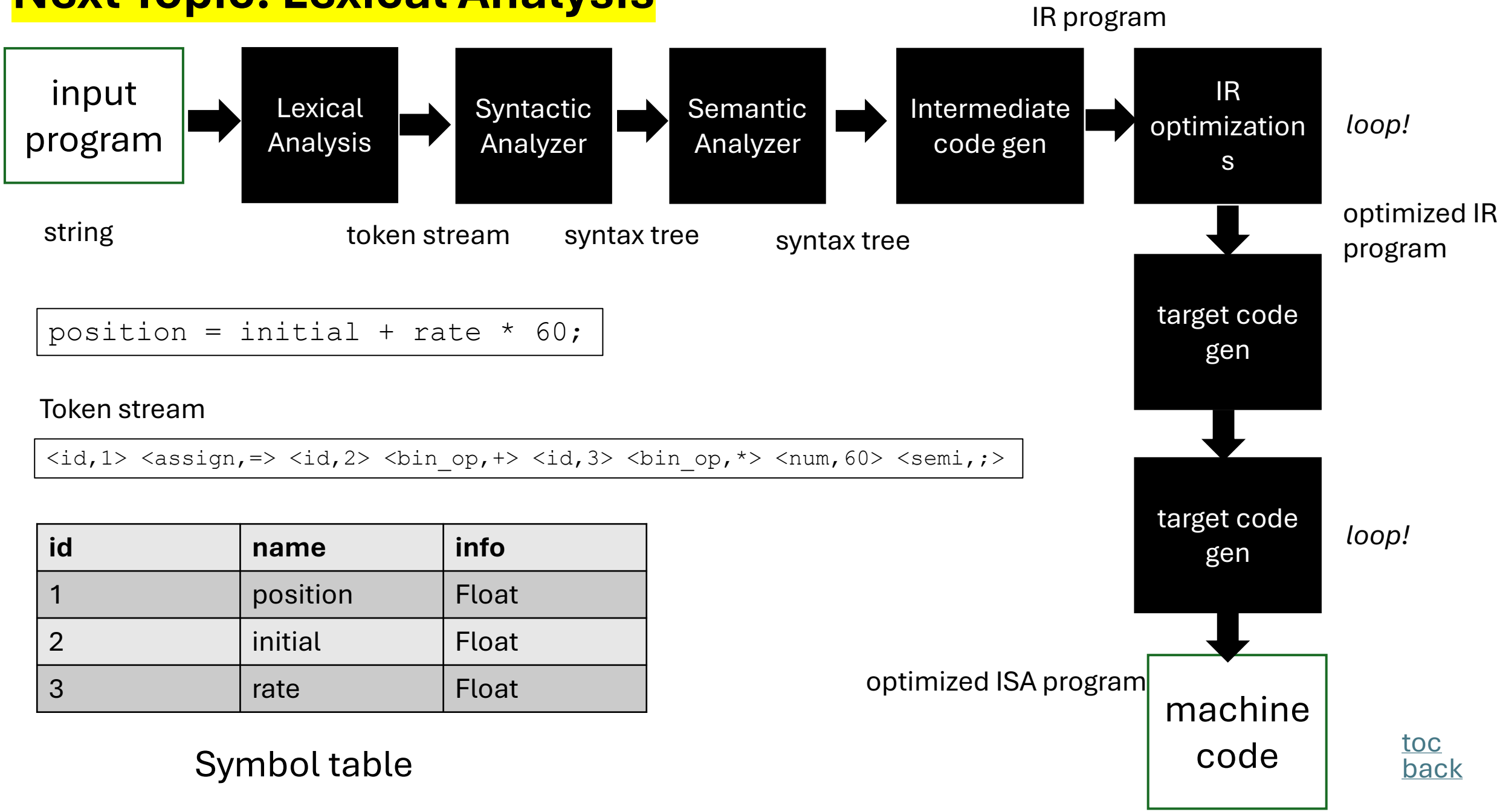
```
madd.s $f1, 60.0, $id3, $id2
```

*some architectures have fused multiply and add instructions (a **peephole optimization**)*

Compiler Architecture

You have just taken your first journey into the heart of a compiler!

Next Topic: Lexical Analysis




```
position = initial + rate * 60;
```

Token stream

```
<id,1> <assign,=> <id,2> <bin_op,+> <id,3> <bin_op,*> <num,60> <semi,;>
```

id	name	info
1	position	Float
2	initial	Float
3	rate	Float

Symbol table

toc
back