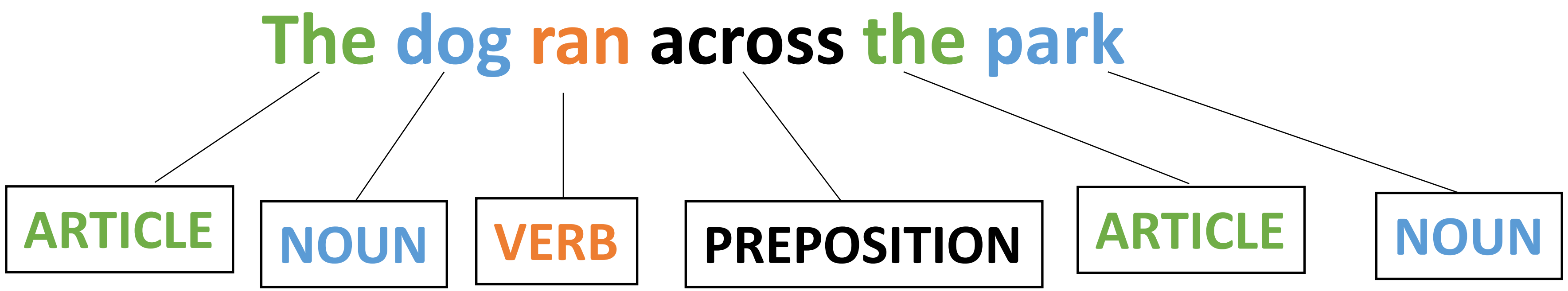
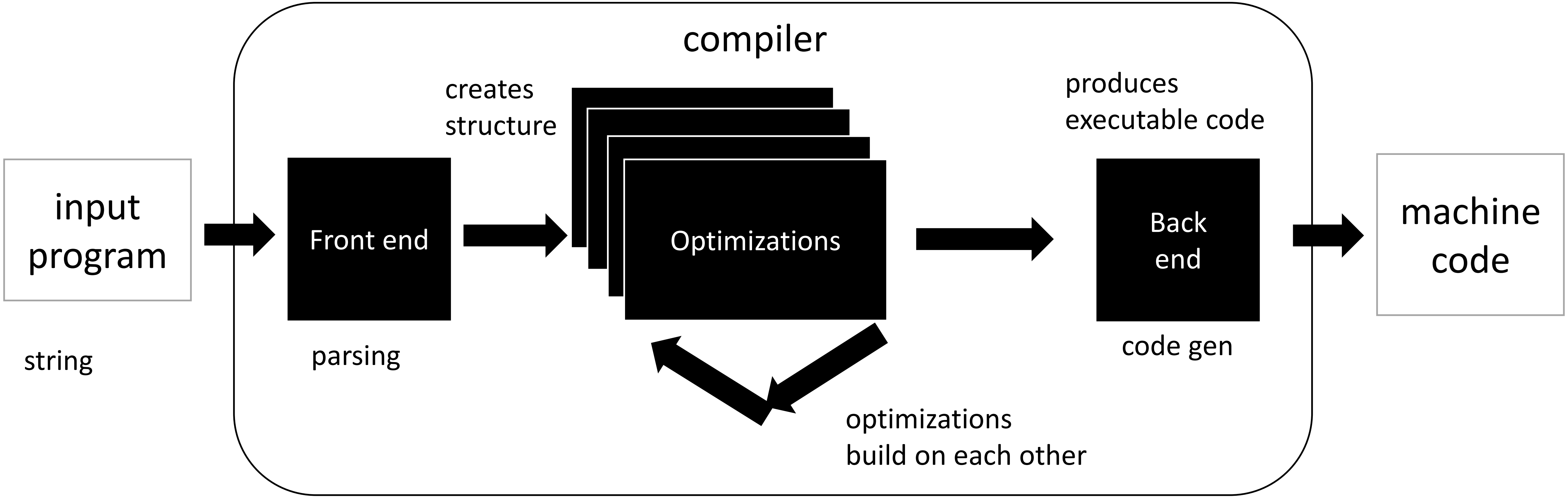
CSE110A: Compilers



- **Topics**:
  - *Lexical Analysis*
    - Introduction
    - Scanners
    - Ad hoc scanner
    - Limitations

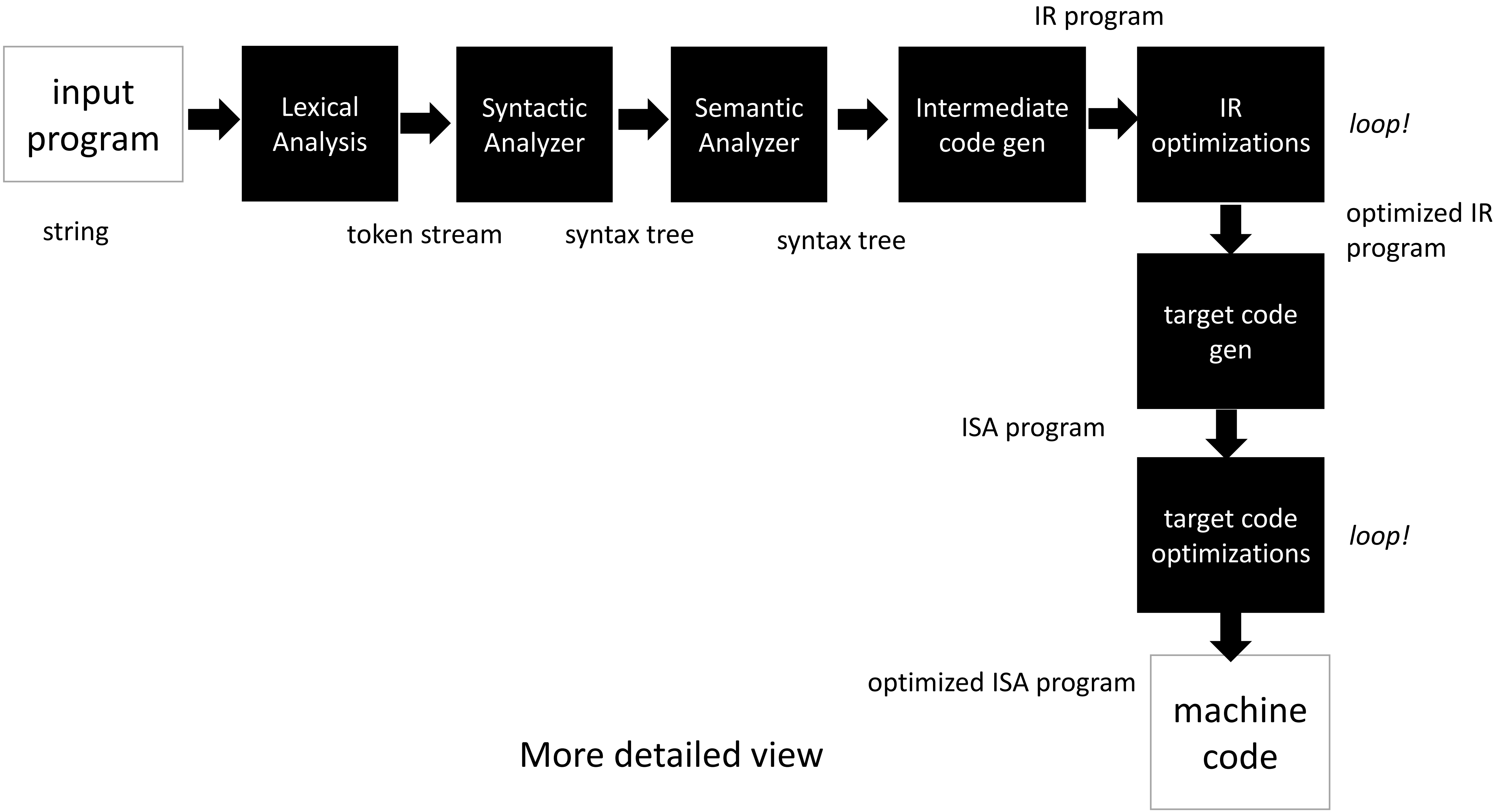
Review: A Modular Compiler

Benefits to modular compiler design

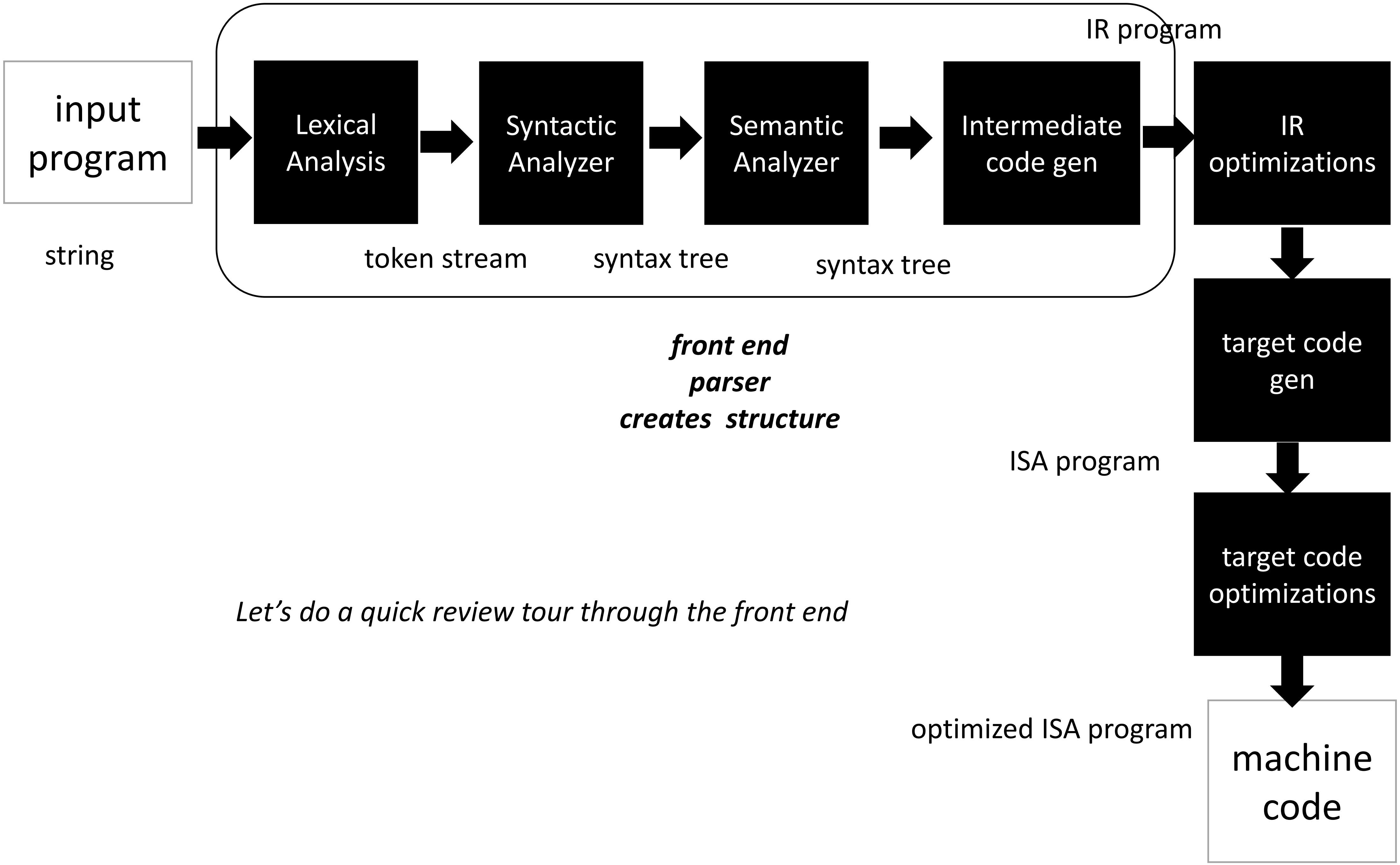


Medium detailed view

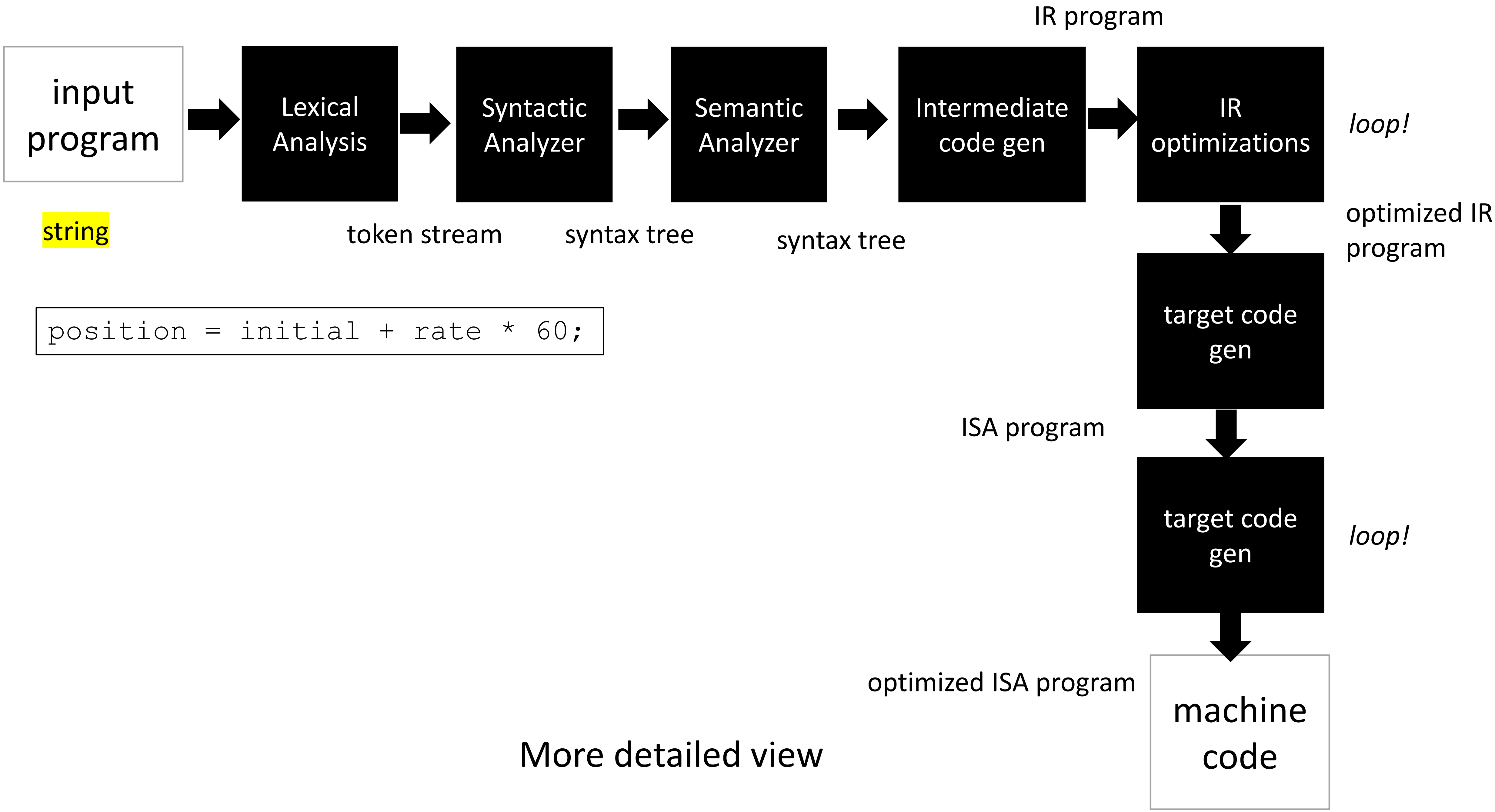
more about optimizations: https://stackoverflow.com/questions/15548023/clang-optimization-levels



More detailed view

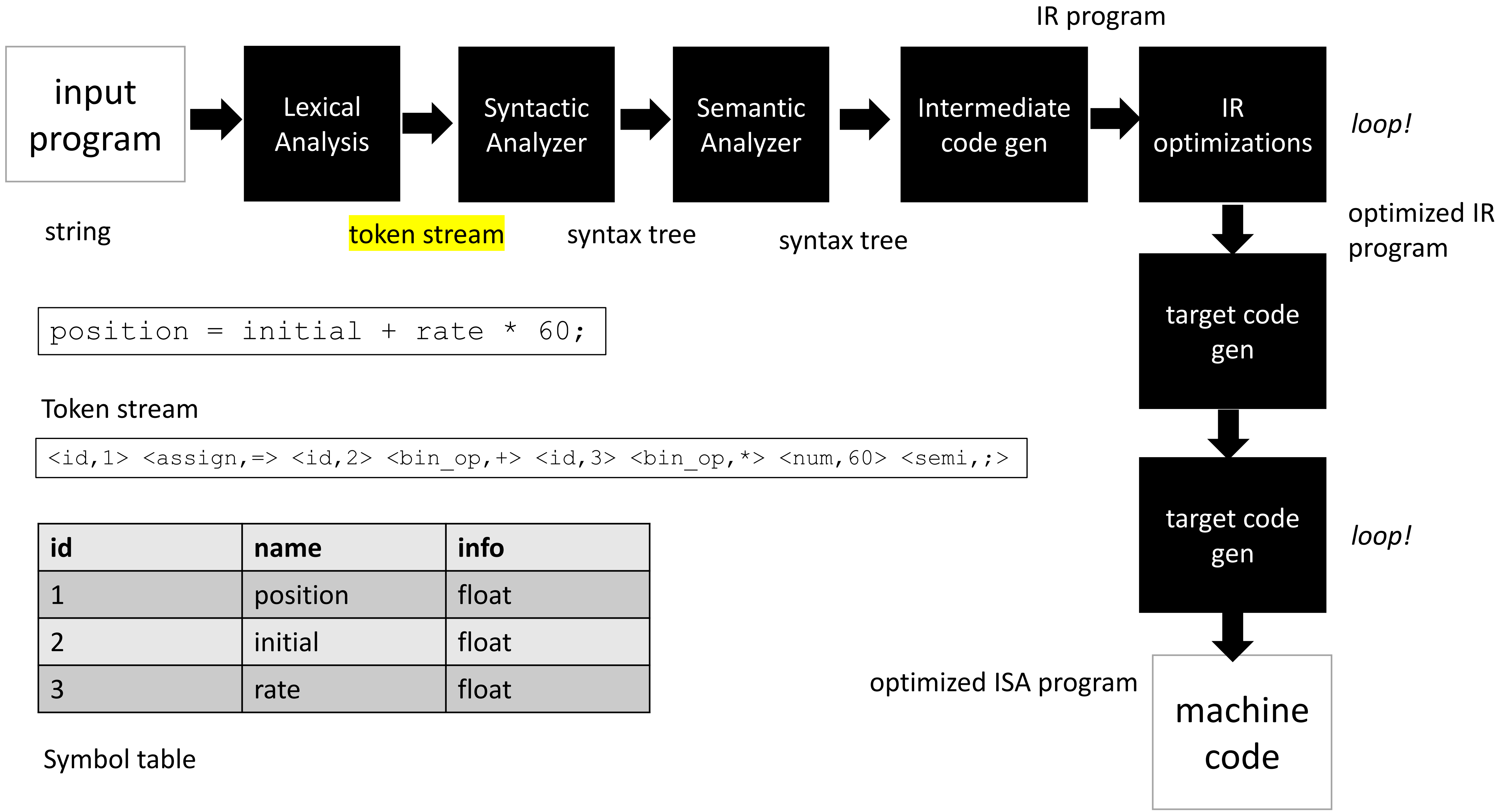


Let's do a quick review tour through the front end




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

More detailed view



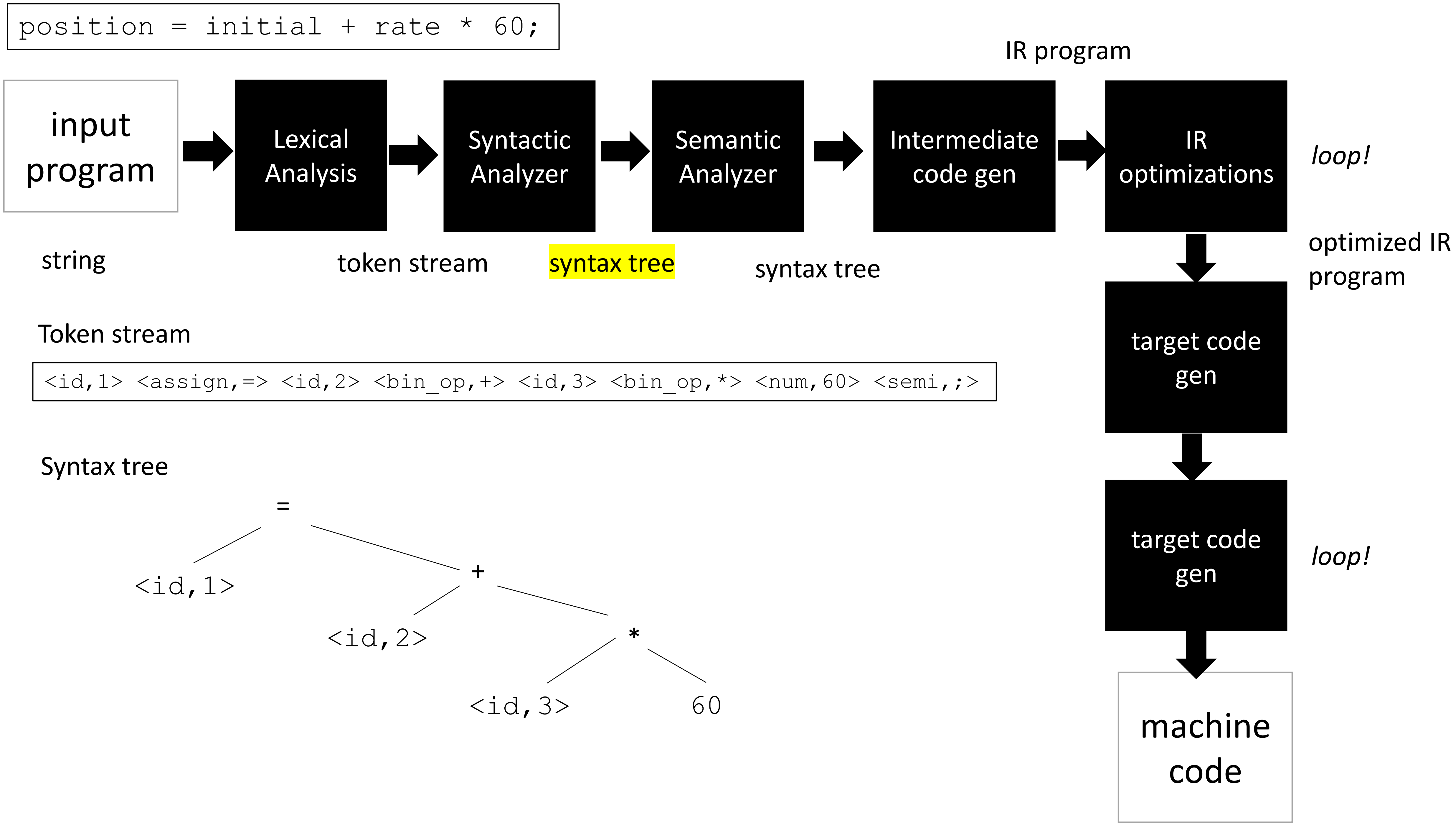

```
position = initial + rate * 60;
```

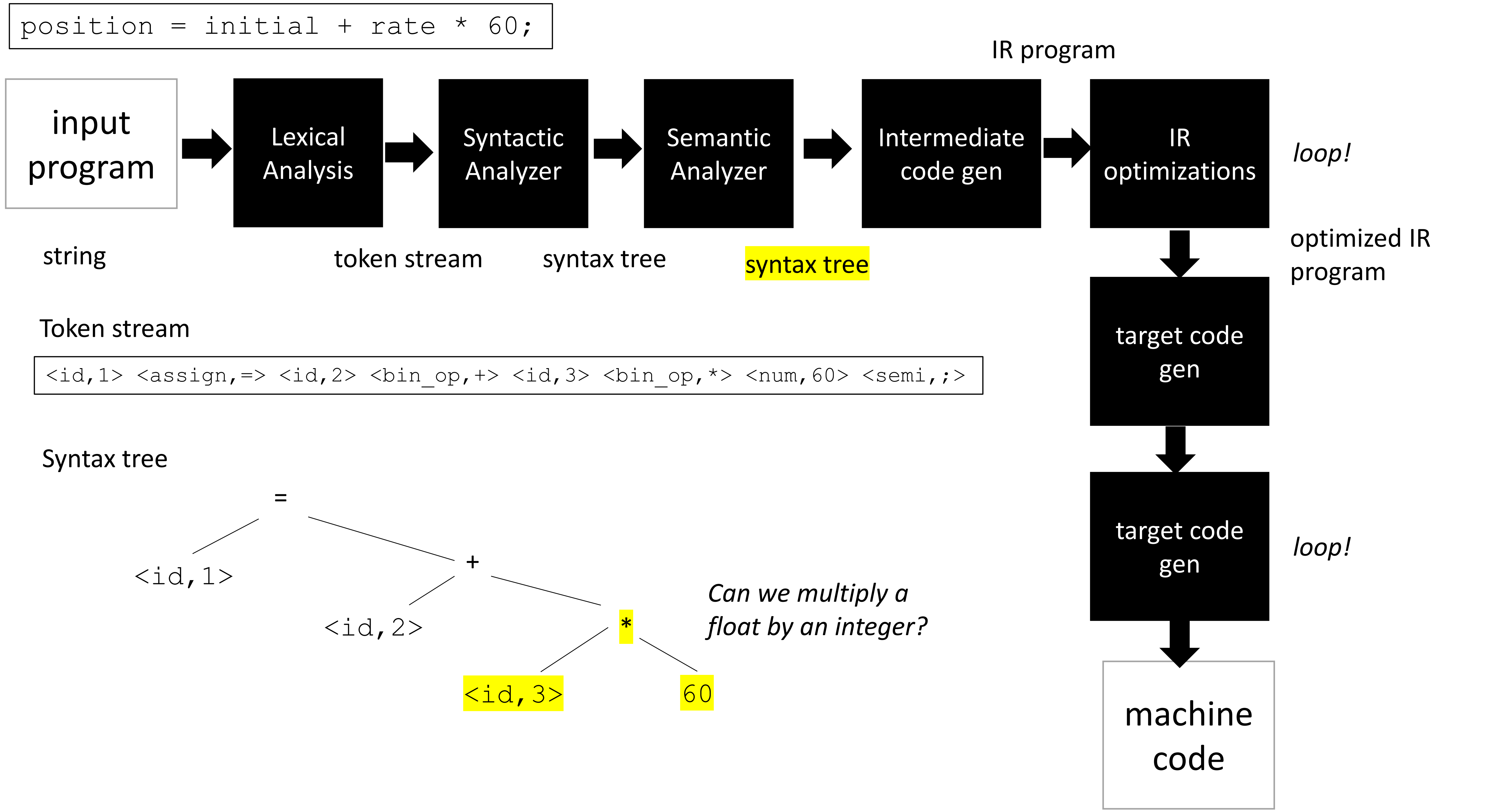
Token stream

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

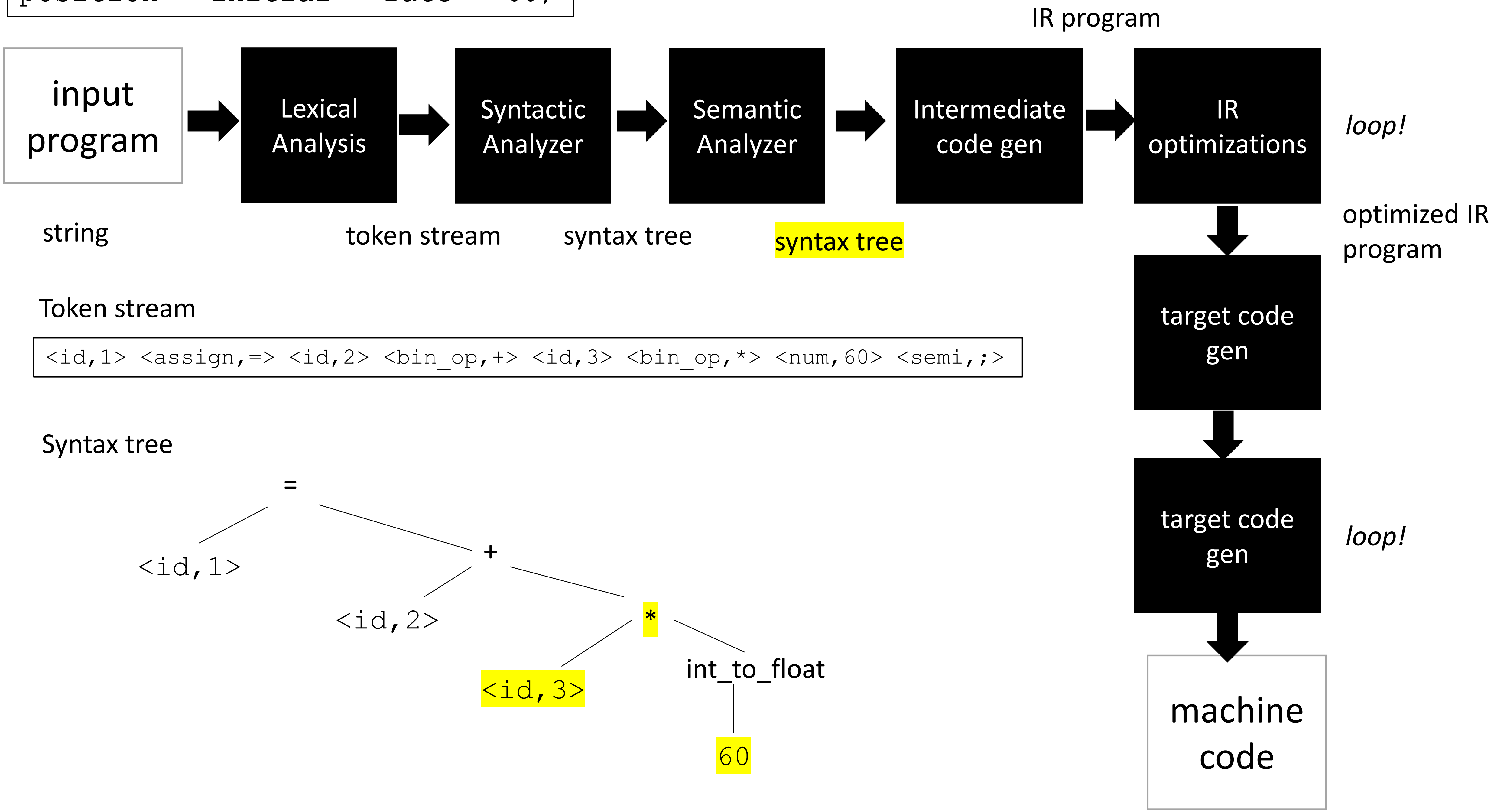
id	name	info
1	position	float
2	initial	float
3	rate	float

Symbol table

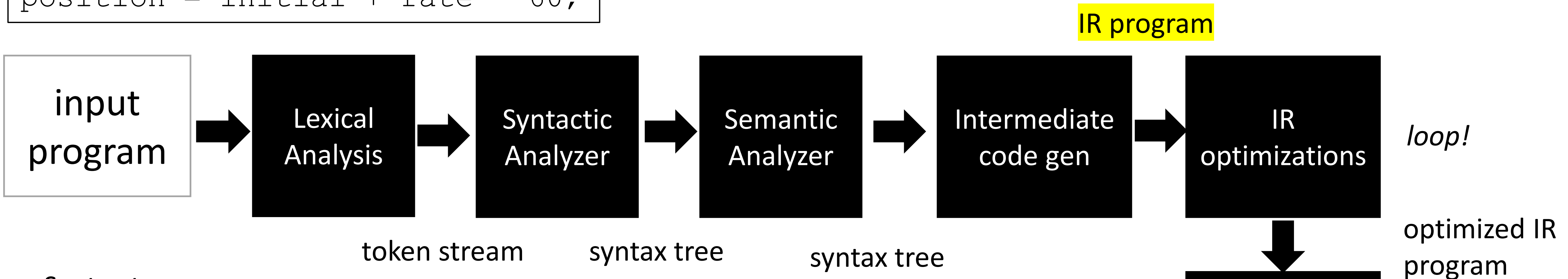

position = initial + rate * 60;
IR program
input program
Lexical Analysis
Syntactic Analyzer
Semantic Analyzer
Intermediate code gen
IR optimizations
loop!
string
token stream
syntax tree
syntax tree
optimized IR program
Token stream
<id,1> <assign,=> <id,2> <bin_op,+> <id,3> <bin_op,*> <num,60> <semi,;>
target code gen
Syntax tree
=
<id,1>
+
<id,2>
*
<id,3>
60
target code gen
loop!
machine code

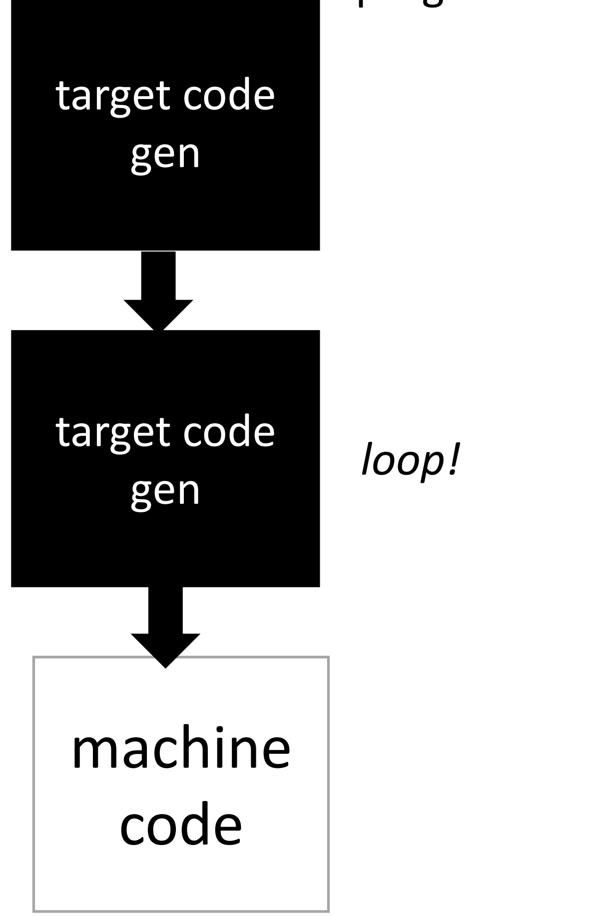

position = initial + rate * 60;
IR program
input program
Lexical Analysis
Syntactic Analyzer
Semantic Analyzer
Intermediate code gen
IR optimizations
loop!
string
token stream
syntax tree
syntax tree
optimized IR program
Token stream
<id,1> <assign,=> <id,2> <bin_op,+> <id,3> <bin_op,*> <num,60> <semi,;>
target code gen
Syntax tree
=
<id,1>
+
<id,2>
*
<id,3>
60
Can we multiply a float by an integer?
target code gen
loop!
machine code

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

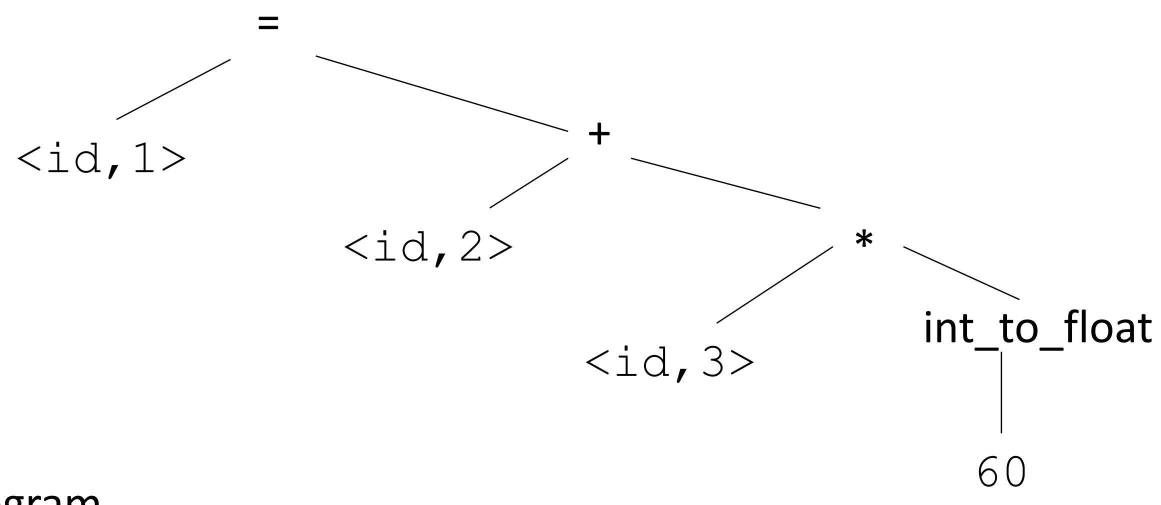


```
position = initial + rate * 60;
```





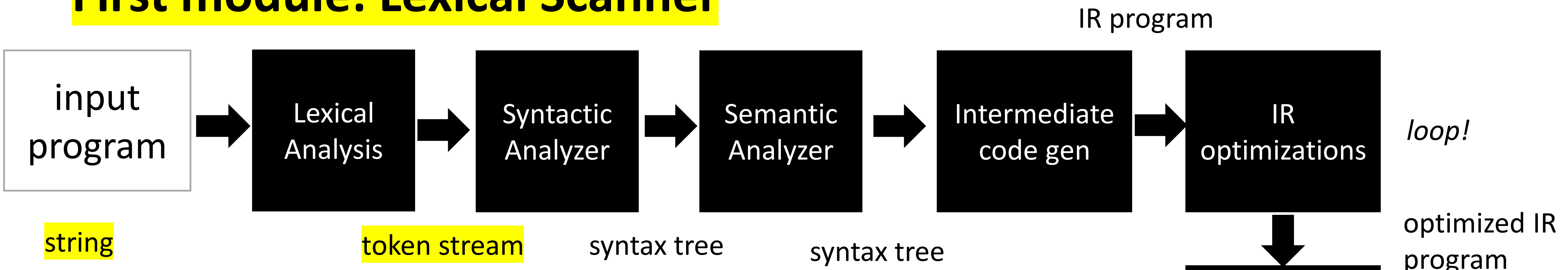
Syntax tree

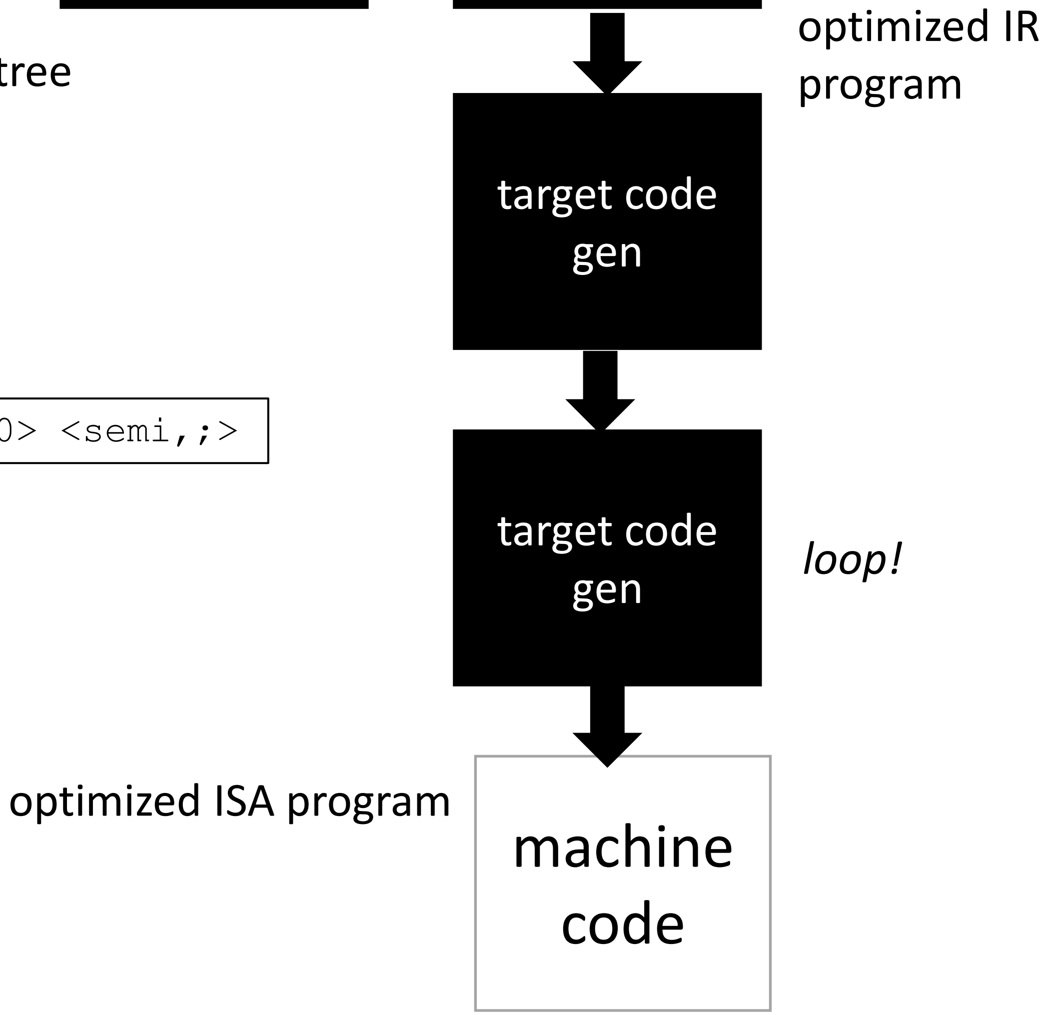


IR program

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

First module: Lexical Scanner






```
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

Topics:

- Introduction Lexical Analysis
- Programs for Lexical Analysis
- Lexical analysis of a simple programming language
- naïve implementation

Topics:

- **Introduction Lexical Analysis**
- Programs for Lexical Analysis
- Lexical analysis of a simple programming language
- Naive implementation

Parsing is the first step in a compiler

- How do we parse a sentence in English?

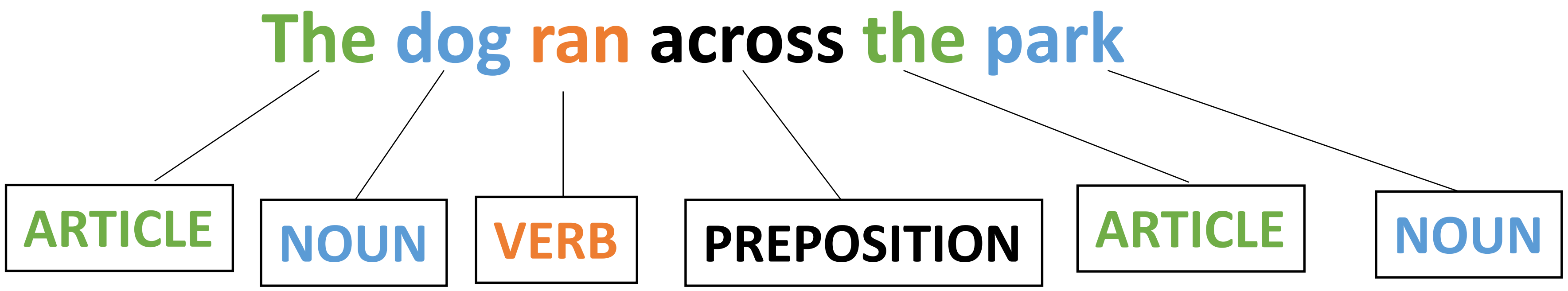
Parsing is the first step in a compiler

- How do we parse a sentence in English?

The dog ran across the park

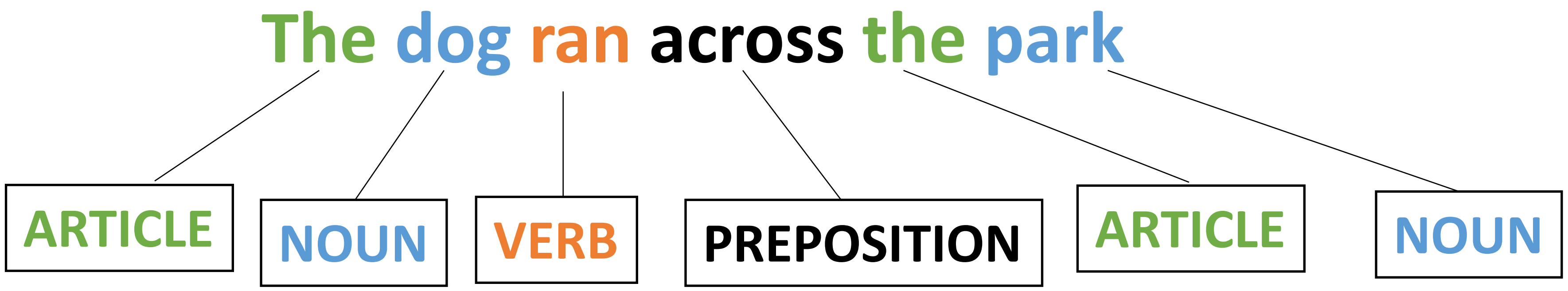
Parsing is one of the first steps in a compiler

- How do we parse a sentence in English?



Parsing is the first step in a compiler

- How do we parse a sentence in English?

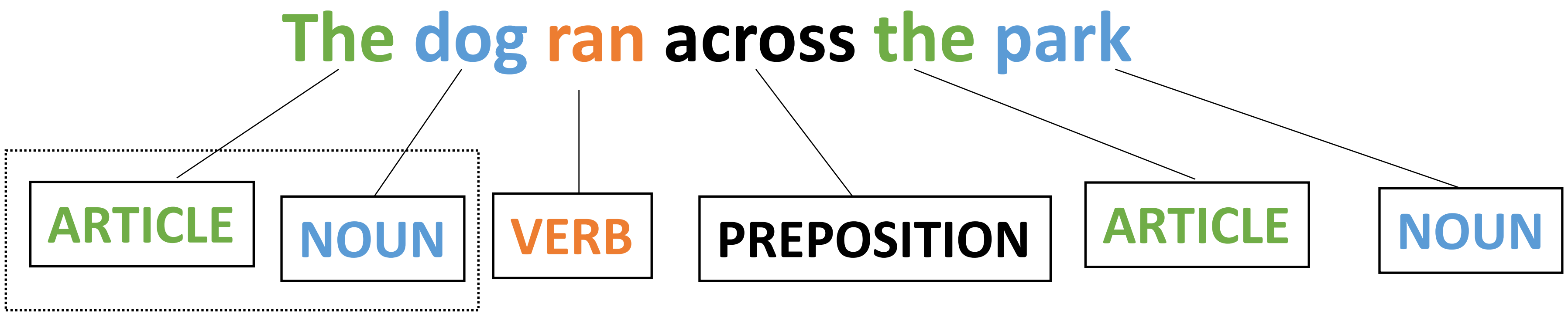


Involves Grammar and Syntax

What about semantics?

Parsing is the first step in a compiler

- How do we parse a sentence in English?



Involves Grammar and Syntax

What about semantics?

New Question

So, can we define a simple language using these building blocks?

- ARTICLE
- NOUN
- VERB
- ADJECTIVE

A Simple Language

- ARTICLE = {The, A, My, Your}
- NOUN = {Dog, Car, Computer}
- VERB = {Ran, Crashed, Accelerated}
- ADJECTIVE = {Purple, Spotted, Old}

A Simple Language

- ARTICLE = {The, A, My, Your}
- NOUN = {Dog, Car, Computer}
- VERB = {Ran, Crashed, Accelerated}
- ADJECTIVE = {Purple, Spotted, Old}

ARTICLE NOUN VERB

A Simple Language

- ARTICLE = {The, A, My, Your}
- NOUN = {Dog, Car, Computer}
- VERB = {Ran, Crashed, Accelerated}
- ADJECTIVE = {Purple, Spotted, Old}

Question mark means optional

ARTICLE ADJECTIVE? NOUN VERB

A Simple Language

- ARTICLE = {The, A, My, Your}
- NOUN = {Dog, Car, Computer}
- VERB = {Ran, Crashed, Accelerated}
- ADJECTIVE = {Purple, Spotted, Old}

ARTICLE	ADJECTIVE?	NOUN	VERB
My	Old	Computer	Crashed

A Simple Language

- ARTICLE = {The, A, My, Your}
- NOUN = {Dog, Car, Computer}
- VERB = {Ran, Crashed, Accelerated}
- ADJECTIVE = {Purple, Spotted, Old}

ARTICLE	ADJECTIVE?	NOUN	VERB
The	Purple	Dog	Crashed

A Simple Language

- ARTICLE = {The, A, My, Your}
- NOUN = {Dog, Car, Computer}
- VERB = {Ran, Crashed, Accelerated}
- ADJECTIVE = {Purple, Spotted, Old}

grammatically correct,
semantically correct?

ARTICLE	ADJECTIVE?	NOUN	VERB
The	Purple	Dog	Crashed

A Simple Language

- ARTICLE = {The, A, My, Your}
- NOUN = {Dog, Car, Computer}
- VERB = {Ran, Crashed, Accelerated}
- ADJECTIVE = {Purple, Spotted, Old}

What other sentences can you construct?

How could we expand the language?

ARTICLE ADJECTIVE? NOUN VERB

A Simple Language

- ARTICLE = {The, A, My, Your}
- NOUN = {Dog, Car, Computer}
- VERB = {Ran, Crashed, Accelerated}
- ADJECTIVE = {Purple, Spotted, Old}

What other languages can you specify?

ARTICLE ADJECTIVE? NOUN VERB

A Simple Language

- ARTICLE = {The, A, My, Your}
- NOUN = {Dog, Car, Computer}
- VERB = {Ran, Crashed, Accelerated}
- ADJECTIVE = {Purple, Spotted, Old}

What other languages can you specify?

ARTICLE ADJECTIVE* NOUN VERB

Stephen Cole Kleene formalized the concept of a regular language.
The so-called Kleene **star** indicates a **repeat of the item (0 or more times)**

Lexical Analysis Labels Parts of Speech

- Parser (module 2) will talk about the organization of the parts of speech

Lexical Analysis

- ARTICLE = {The, A, My, Your}
- NOUN = {Dog, Car, Computer}
- VERB = {Ran, Crashed, Accelerated}
- ADJECTIVE = {Purple, Spotted, Old}

Parser

ARTICLE ADJECTIVE* NOUN VERB

Topics:

- Introduction Lexical Analysis
- **Programs for Lexical Analysis**
- Lexical analysis of a simple programming language
- Naïve implementation

Programs for Lexical Analysis

Scanner (sometimes called lexer)

It gets defined by a list of tokens and definitions:

Tokens		Tokens Definitions
ARTICLE	=	{The, A, My, Your}
NOUN	=	{Dog, Car, Computer}
VERB	=	{Ran, Crashed, Accelerated}
ADJECTIVE	=	{Purple, Spotted, Old}

Programs for Lexical Analysis

Scanner (sometimes called lexer)

Defined by a list of tokens and definitions:

Tokens		Tokens Definitions
• ARTICLE	=	{The, A, My, Your}
• NOUN	=	{Dog, Car, Computer}
• VERB	=	{Ran, Crashed, Accelerated}
• ADJECTIVE	=	{Purple, Spotted, Old}

Original program:
Lex

https://en.wikipedia.org/wiki/Lex_(software)

Popular implementations
Flex

Scanner API

```
// Constructor, generates a Scanner
s = ScannerGenerator(tokens)

// The string we want to do
// lexical analysis on
s.input("My Old Computer Crashed")
```

Scanner API

What do we want?

Scanner API

What do we want?

"My Old Computer Crashed"

↓ Scanner

Scanner API

What do we want?

"My Old Computer Crashed"

Scanner

```
[(ARTICLE), (ADJECTIVE), (NOUN), (VERB)]
```

Useful, but we might need more information

Scanner API

What do we want?

“My Old Computer Crashed”

Scanner

```
[(ARTICLE), (ADJECTIVE), (NOUN), (VERB)]
```

Useful, but we might need more information

Lexeme*: (TOKEN, value)*

Scanner API

What do we want?

"My Old Computer Crashed"

Scanner

```
[(ARTICLE, "My"), (ADJECTIVE, "Old"), (NOUN, "Computer"), (VERB, "Crashed")]
```

Scanner API

What do we want?

"My Old Computer Crashed"

Scanner

```
[(ARTICLE, "My"), (ADJECTIVE, "Old"), (NOUN, "Computer"), (VERB, "Crashed")]
```

***Lexeme**: (TOKEN, value)*

Scanner API

What do we want?

“My Old Computer Crashed”

Scanner

classically, this occurs one lexeme at a time

```
[(ARTICLE, “My”), (ADJECTIVE, “Old”), (NOUN, “Computer”), (VERB, “Crashed”)]
```

Scanner API

```
// Constructor, generates a Scanner
s = ScannerGenerator(tokens)

// The string we want to do
// lexical analysis on
s.input("My Old Computer Crashed")

// Returns the next lexeme
s.token()
```

```
> s = ScannerGenerator(tokens)
> s.input(“My Old Computer Crashed”)
> s.token()
```

```
> s = ScannerGenerator(tokens)
> s.input("My Old Computer Crashed")
> s.token()
(ARTICLE, "My")
> s.token()
```

```
> s = ScannerGenerator(tokens)
> s.input("My Old Computer Crashed")
> s.token()
(ARTICLE, "My")
> s.token()
(ADJECTIVE, "Old")
> s.token()
```

```
> s = ScannerGenerator(tokens)
> s.input("My Old Computer Crashed")
> s.token()
(ARTICLE, "My")
> s.token()
(ADJECTIVE, "Old")
> s.token()
(NOUN, "Computer")
```

```
> s = ScanerGenerator(tokens)
> s.input("My Old Computer Crashed")
> s.token()
(ARTICLE, "My")
> s.token()
(ADJECTIVE, "Old")
> s.token()
(NOUN, "Computer")
> s.token()
```

```
> s = ScanerGenerator(tokens)
> s.input("My Old Computer Crashed")
> s.token()
(ARTICLE, "My")
> s.token()
(ADJECTIVE, "Old")
> s.token()
(NOUN, "Computer")
> s.token()
(VERB, "Crashed")
> s.token()
```

```
> s = ScanerGenerator(tokens)
> s.input(“My Old Computer Crashed”)
> s.token()
(ARTICLE, “My”)
> s.token()
(ADJECTIVE, “Old”)
> s.token()
(NOUN, “Computer”)
> s.token()
(VERB, “Crashed”)
> s.token()
None
```

Schedule

- Introduction Lexical Analysis
- Programs for Lexical Analysis
- **Lexical analysis of a simple programming language**
- naïve implementation

Lexical analysis of a simple programming lang.

Let’s write tokens and definitions for a simple programming language

- integer arithmetic (+,*)
- variables, assignments, non-negative integers

example

x = 5 + 4 * 3;

What tokens should we have? Ideas?

Lexical analysis of a simple programming lang.

Lets write tokens and definitions for a simple programming language

- integer arithmetic (+,*)
- variables, assignments, non-negative integers

example

x = 5 + 4 * 3;

maybe something like this?

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
```

Lexical analysis of a simple programming lang.

Lets write tokens and definitions for a simple programming language

- integer arithmetic (+,*)
- variables, assignments, non-negative integers

example

x = 5 + 4 * 3;

maybe something like this?

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
```

```
[(ID, “x”),  (ASSIGN, “=“), (NUM, ”5”), (PLUS, “+”) ,
 (NUM, “4”), (MULT, “*”),   (NUM, “3”)]
```

Lexical analysis of a simple programming lang.

Lets write tokens and definitions for a simple programming language

- integer arithmetic (+,*)
- variables, assignments, non-negative integers

example

x = 5 + 4 * 3;

Other options for tokens we could define?

maybe something like this?

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
```

Lexical analysis of a simple programming lang.

Lets write tokens and definitions for a simple programming language

- integer arithmetic (+,*)
- variables and assignments

example

x = 5 + 4 * 3;

maybe something like this?

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
```

Other options for tokens we could define?

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = ”=“
OP     = {“+”, “*”}
```

Lexical analysis of a simple programming lang.

Lets write tokens and definitions for a simple programming language

- integer arithmetic (+,*)
- variables and assignments

maybe something like this?

```
ID      = [characters]
NUM     = [numbers]
ASSIGN  = "="
PLUS    = "+"
MULT    = "*"
```

example

x = 5 + 4 * 3;

```
(OP, "+") (OP, "*")
```

We can always distinguish using the value

Other options for tokens we could define?

```
ID      = [characters]
NUM     = [numbers]
ASSIGN  = "="
OP      = {"+", "*"}
```

Lexical analysis of a simple programming lang.

Lets write tokens and definitions for a simple programming language

- integer arithmetic (+,*)
- variables and assignments

example

x = 5 + 4 * 3;

Other options for tokens we could define?

maybe something like this?

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
```

Lexical analysis of a simple programming lang.

Lets write tokens and definitions for a simple programming language

- integer arithmetic (+,*)
- variables and assignments

example

x = 5 + 4 * 3;

maybe something like this?

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
```

what do we think about this?

Other options for tokens we could define?

```
ID      = [characters]
FIVE    = “5”
FOUR    = “4”
...
PLUS    = “+”
MULT    = “*”
```

Lexical analysis of a simple programming lang.

Lets write tokens and definitions for a simple programming language

- integer arithmetic (+,*)
- variables and assignments

example

What are we missing?

x = 5 + 4 * 3;

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
```

Lexical analysis of a simple programming lang.

Lets write tokens and definitions for a simple programming language

- integer arithmetic (+,*)
- variables and assignments

example

x = 5 + 4 * 3;

What are we missing?

whitespace!

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = "="
PLUS   = "+"
MULT   = "*"
```

Lexical analysis of a simple programming lang.

Lets write tokens and definitions for a simple programming language

- integer arithmetic (+,*)
- variables and assignments

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = "="
PLUS   = "+"
MULT   = "*"
IGNORE = " "
```

example

x = 5 + 4 * 3;

What are we missing?

whitespace!

Typically* we ignore whitespace and newlines and tabs

Ignored tokens do not get returned as a lexeme

*unless we are python 😖

Parsing is the first step in a compiler

- How do we parse a sentence in English?

The dog ran across the park

ARTICLE NOUN VERB PREPOSITION ARTICLE NOUN

White space is ignored because it is not meaningful!

Longest possible match

Consider the token:

- `CLASS_TOKEN = {“cse”, “110”, “cse110”}`

What would the lexemes be for: “cse110”

options:

- `(CLASS_TOKEN, ”cse”) (CLASS_TOKEN, ”110”)`
- `(CLASS_TOKEN, ”cse110”)`

Longest possible match

Consider the token:

- `CLASS_TOKEN = {“cse”, “110”, “cse110”}`

What would the lexemes be for: “cse110”

options:

- `(CLASS_TOKEN, ”cse”) (CLASS_TOKEN, ”110”)`
- `(CLASS_TOKEN, ”cse110”)`

This one!

Longest possible match

- Important for operators, e.g. in C
- `++`, `+=`

how would we scan “`x++;`”

```
[(ID, “x”), (ADD, “+”), (ADD, “+”), (SEMI, “;”)]
```

```
[(ID, “x”), (INCREMENT, “++”), (SEMI, “;”)]
```

Longest possible match

Important for variable names and numbers

how would we scan: “`my_var = 10;`” ?

Longest possible match

Important for variable names and numbers

how would we scan: “`my_var = 10;`” ?

```
[(ID, “my_var”), (ASSIGN, “=”), (NUM, “10”), (SEMI, “;”)]
```

Schedule

- Introduction Lexical Analysis
- Programs for Lexical Analysis
- Lexical analysis of a simple programming language
- **naïve implementation**

Naïve implementation

- A scanner that implements

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = [“ “]
```

Naïve implementation

Building block:

This class allows strings to be read as if we were doing I/O from a file.

So you are implementing with an abstraction that works both from a string or from a file.

```
class StringStream:
    def __init__(self, input_string):
        self.string = input_string

    def is_empty(self):
        return len(self.string) == 0

    def peek_char(self):
        if not self.is_empty():
            return self.string[0]
        return None

    def eat_char(self):
        self.string = self.string[1:]
```

Naïve implementation

First step in implementing the scanner

```
class NaiveScanner:
  def __init__(self, input_string):
    self.ss = StringStream(input_string)

  def token(self):
    while self.ss.peek_char() in IGNORE:
      self.ss.eat_char()

    if self.ss.is_empty():
      return None
```

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = "="
PLUS   = "+"
MULT   = "*"
IGNORE = [" "]
```

Naïve implementation

First step in implementing the scanner

```
class NaiveScanner:
    def token(self):
        ...
        if self.ss.peek_char() == "+":
            value = self.ss.peek_char()
            self.ss.eat_char()
            return ("ADD", value)

        if self.ss.peek_char() == "*":
            value = self.ss.peek_char()
            self.ss.eat_char()
            return ("MULT", value)
```

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = "="
PLUS   = "+"
MULT   = "*"
IGNORE = [" "]
```

Naïve implementation

First step in implementing the scanner

```
class NaiveScanner:

  def token(self):
    ...
    if self.ss.peek_char() in NUMS:
      value = ""
      while self.ss.peek_char() in NUMS:
        value += self.ss.peek_char()
        self.ss.eat_char()
      return ("NUM", value)
```

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = "="
PLUS   = "+"
MULT   = "*"
IGNORE = [" "]
```

Code Demo

What are the issues with our Scanner?

- Think about it for next class, where we will discuss:

Regular Expressions!