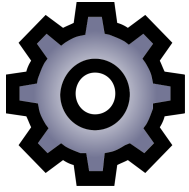
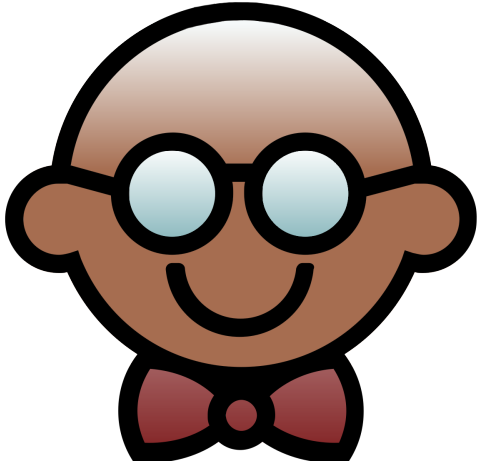
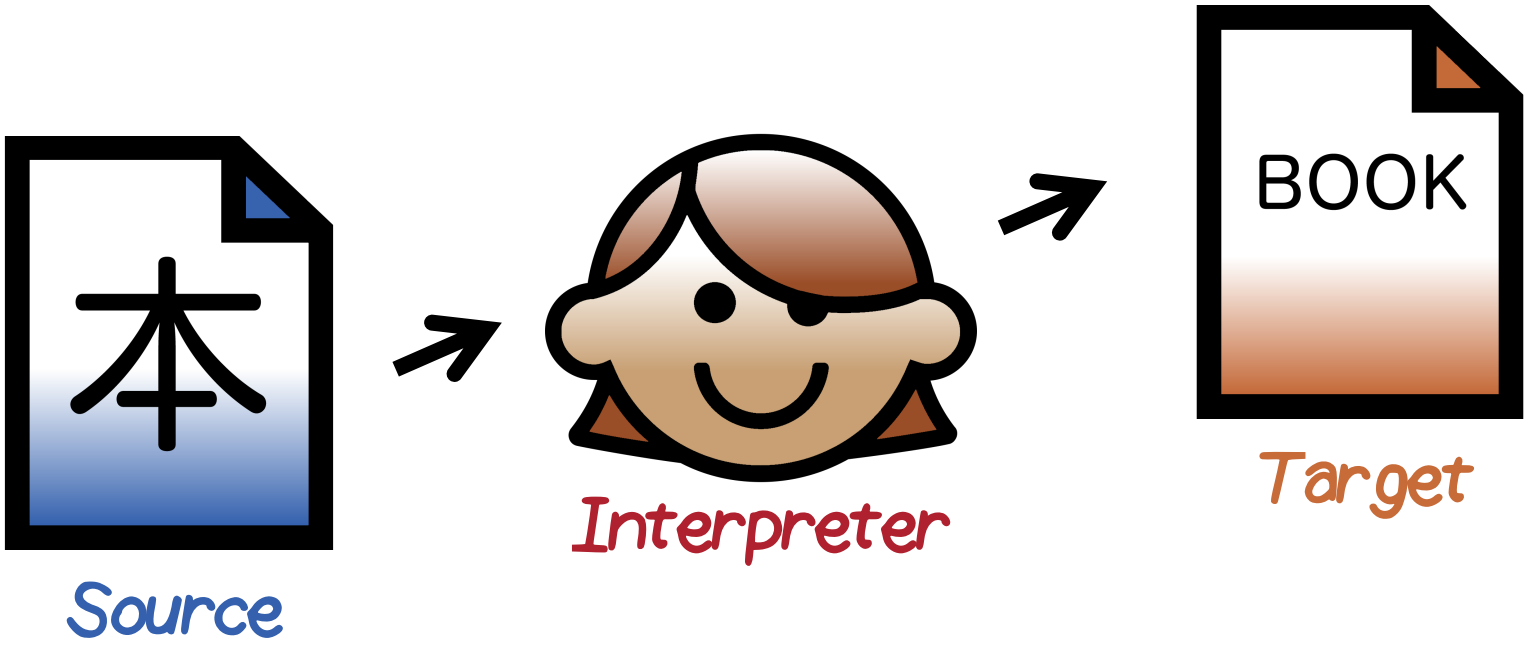


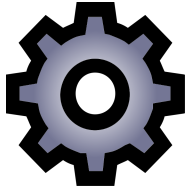
# What is a Compiler?



A program that **translates** a program from a **source language** into a **target language**!

# What is a Compiler?





# What is a Compiler?

High-level Language

```
temp    =v[k];
v[k]    =v[k+1];
v[k+1]  =temp;
```

```
TEMP    =V(K)
V(K)    =V(K+1)
V(K+1)  =TEMP
```

C/Java Compiler

Fortran Compiler

Assembly Language

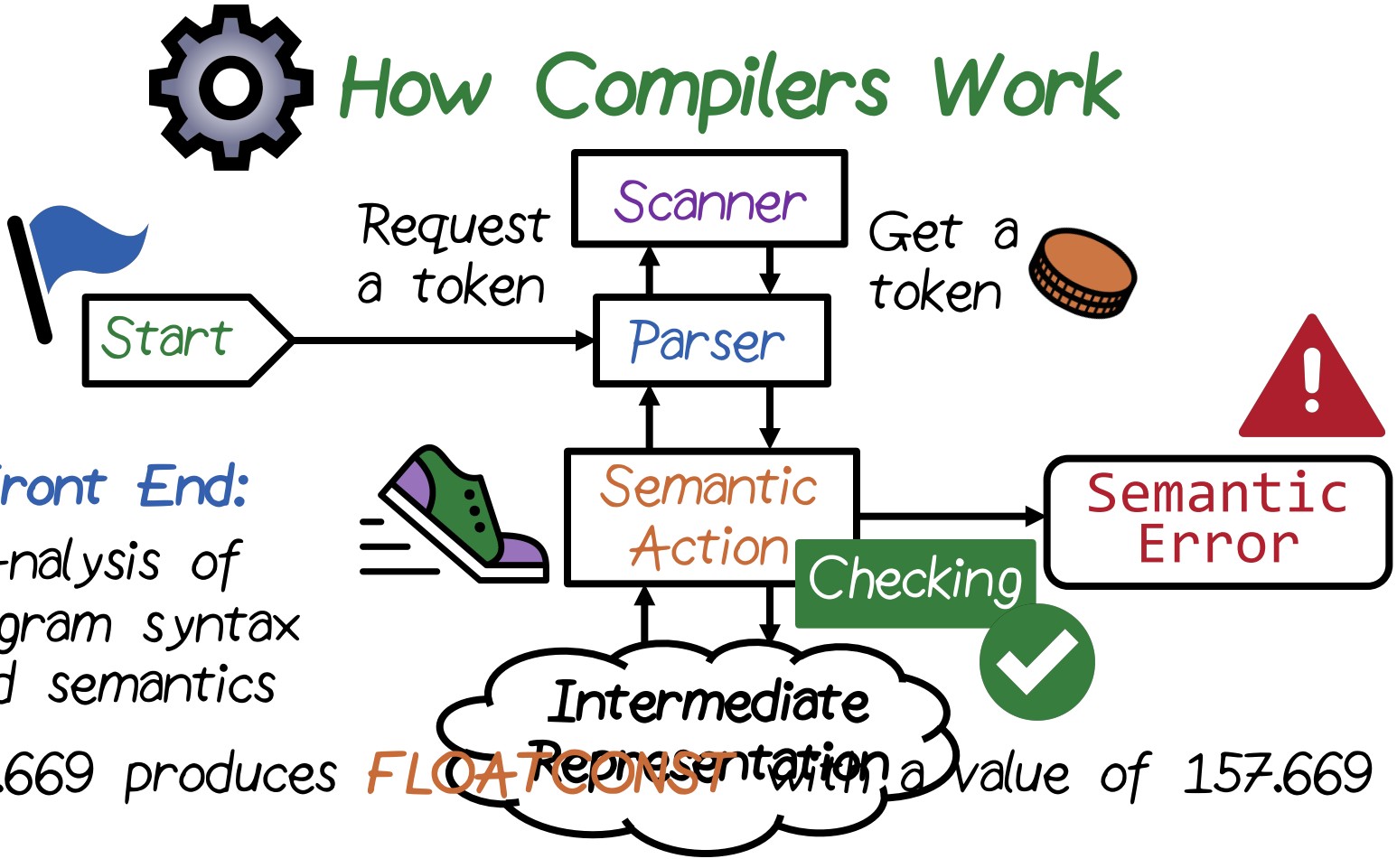
```
lw $to, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $to, 4($2)
```

MIPS Assembler

Machine Language

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

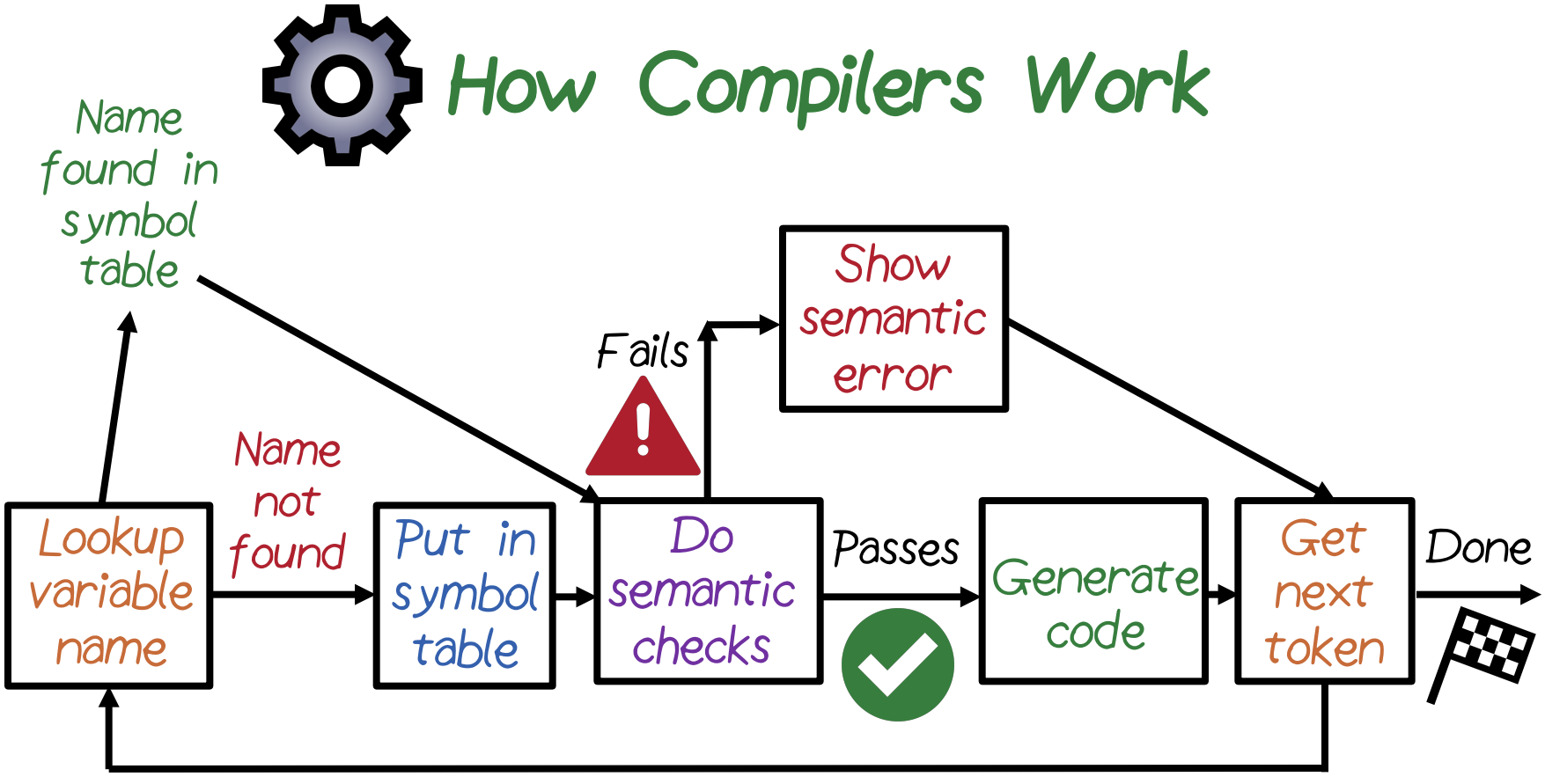
# How Compilers Work

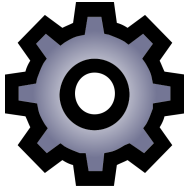



**Front End:**

Analysis of program syntax and semantics

157.669 produces FLOATCONST with a value of 157.669

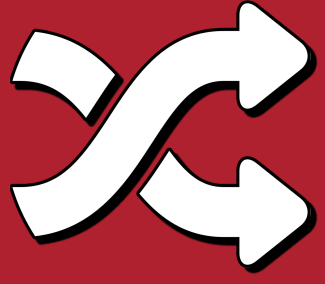

How Compilers Work
Name found in symbol table
Show semantic error
Fails
Name not found
Lookup variable name
Put in symbol table
Do semantic checks
Passes
Generate code
Get next token
Done



# Compiler Parts

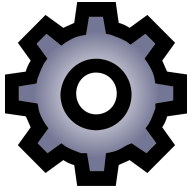
## Front End: Analysis

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis



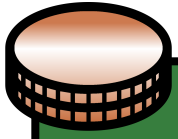
## Back End: Synthesis

4. Code Generation
5. Optimization



# The Big Picture


Parsing: Translating code to rules of grammar. Building representation of code.



Scanning: Converting input text into stream of known objects called tokens. Simplifies parsing process.

- Grammar dictates syntactic rules of language
  - i.e., how legal sentence could be formed
- Lexical rules of language dictate how legal word is formed by concatenating alphabet.

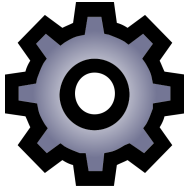
# Tokenization Quiz

If we use a space to denote the end of a section, how many tokens are in the following sentence?

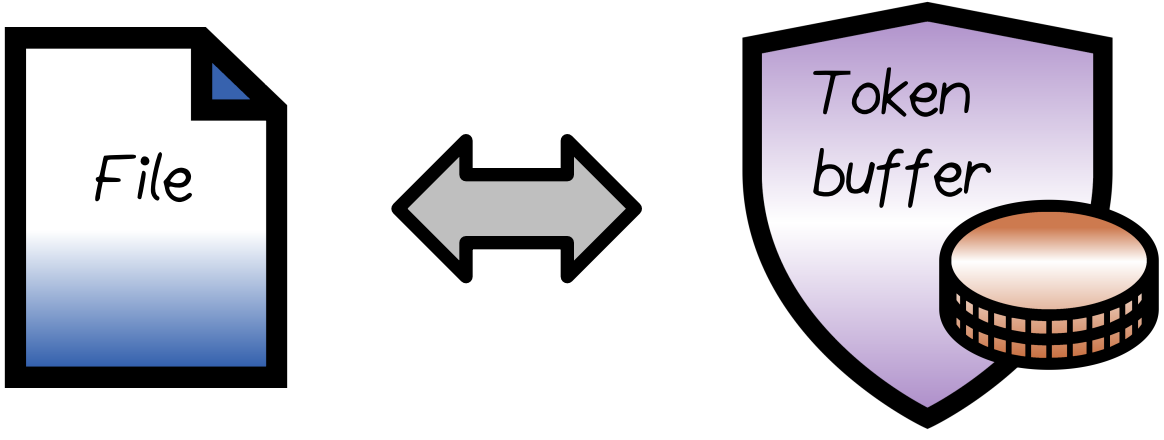
**Fill in the blank:** 21

*"The question of whether a computer can think is no more interesting than the question of whether a submarine can swim."*

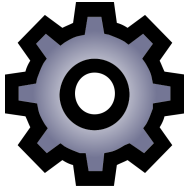
- Edsger W. Dijkstra



# Scanning and Tokenization



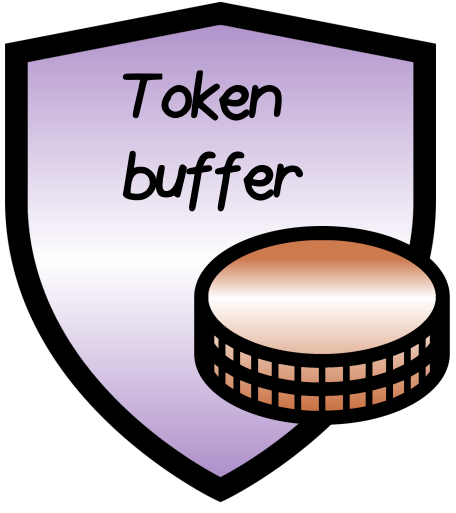

**Token buffer** contains: **token** being identified



# Scanning and Tokenization

```
main()
{
    printf(“Hello World”);
}
```

| Token buffer |
| --- |
| niam |




Keyword: main

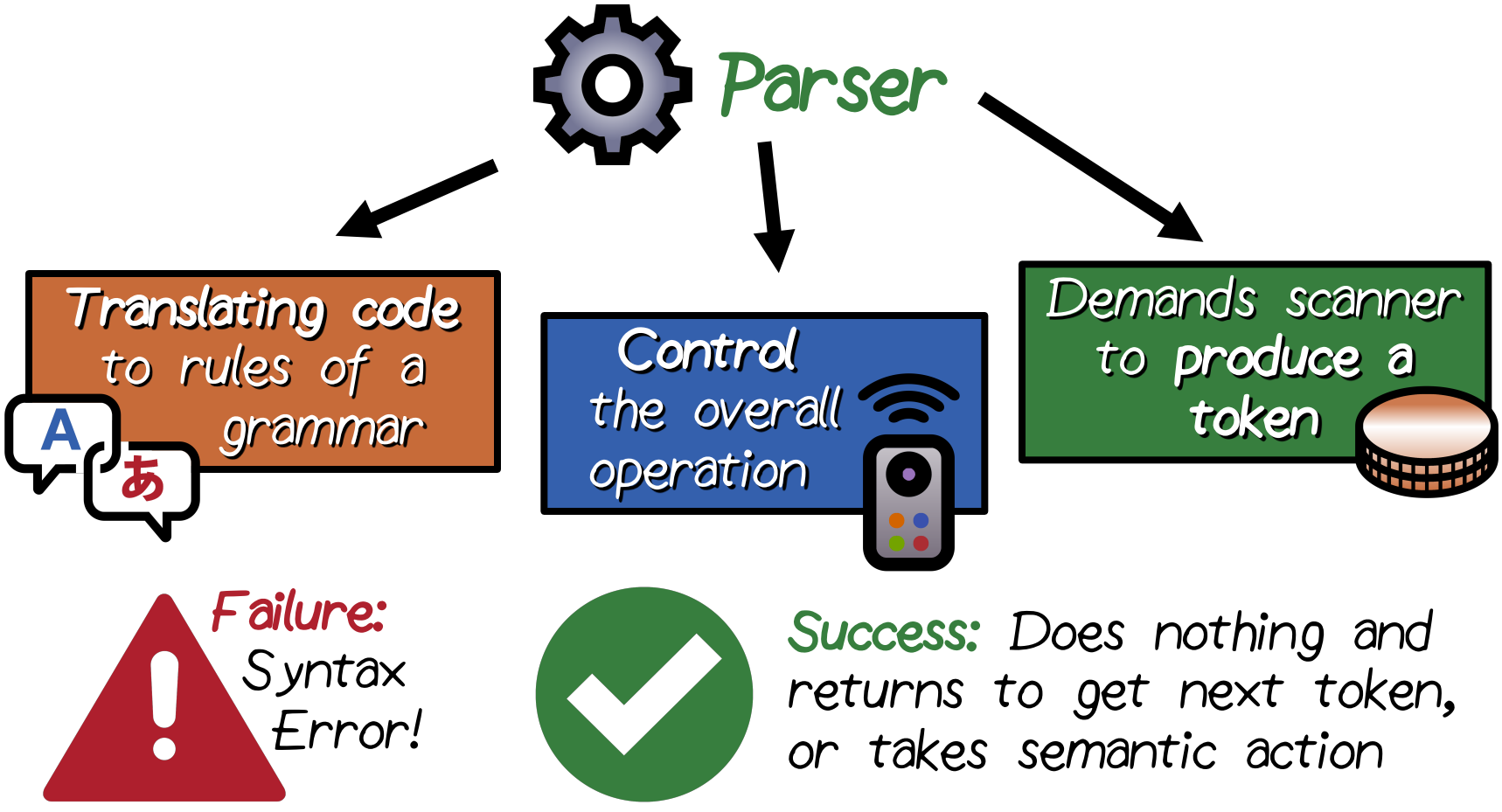
# Parser Quiz

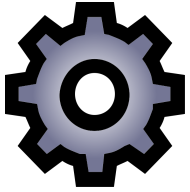
## Grammar Rules:

- E -> E + E
- E -> E * E
- E -> -E
- E -> (E)
- E -> id

Check all the statements that are **grammatically correct** given the grammar rules.

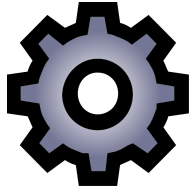
- [x] a + b
- [x] a + b * c
- [ ] a b + c
- [ ] a + b + a c


Parser
Translating code to rules of a grammar
A
あ
Control the overall operation
Demands scanner to produce a token
Failure:
Syntax Error!
Success: Does nothing and returns to get next token, or takes semantic action



# Parser: Grammar Rules

<C-PROG> → MAIN OPENPAR <PARAMS> CLOSEPAR <MAIN-BODY>
<PARAMS> → NULL
<PARAMS> → VAR <VARLIST>
<VARLIST> → , VAR <VARLIST>
<VARLIST> → NULL
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<DECL-STMT> → <TYPE> VAR <VARLIST>;
<ASSIGN-STMT> → VAR = <EXPR>;
<EXPR> → VAR
<EXPR> → VAR <OP> <EXPR>
<OP> → +
<OP> → -
<TYPE> → INT
<TYPE> → FLOAT



# Parser

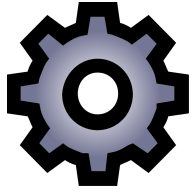
Example:

```
main() {
   int a,b;
   a = b;
}
```

Scanner

Token Buffer

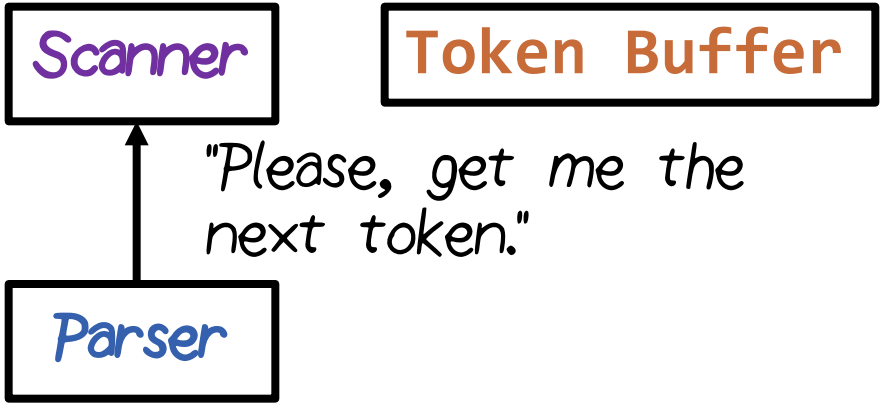
Parser

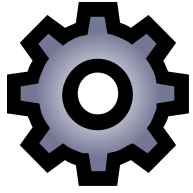


# Parser

Example:

```
main() {
   int a,b;
   a = b;
}
```





# Parser

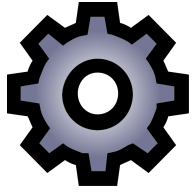
Example:

```
main() {
   int a,b;
   a = b;
}
```

Scanner

Parser

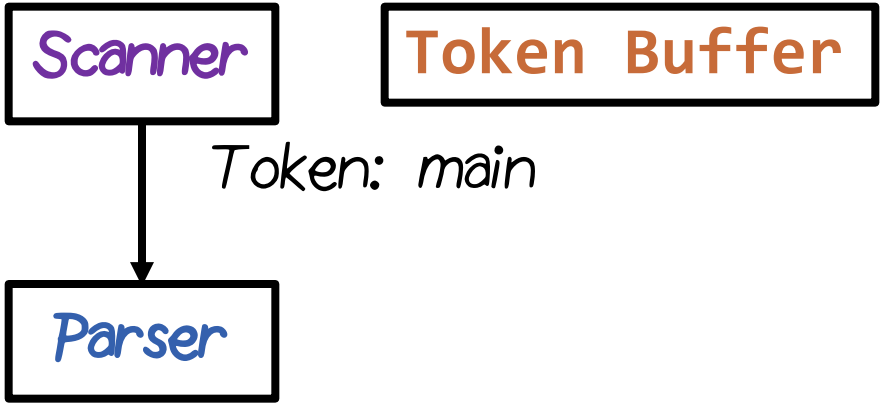
| Token Buffer |
| --- |
| (niam |

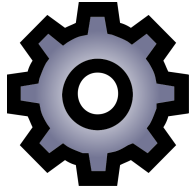


# Parser

Example:

```
main() {
   int a,b;
   a = b;
}
```





# Parser

Example:

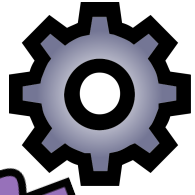
```
main() {
   int a,b;
   a = b;
}
```

Scanner

Token Buffer

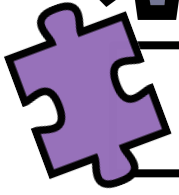
Parser

I recognize this as per grammar rules!
Every C program should start with main.
This program starts with main.

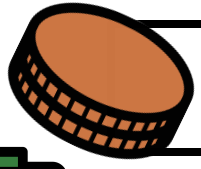


# Parser

**Matching:**



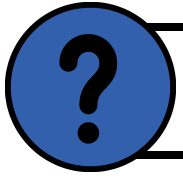
Start matching using a rule



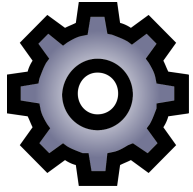
When match takes place at certain position, move further (get next token & repeat)

If expansion needs to be done, choose appropriate rule (How to decide which rule to choose?)

If no rule found, declare error



If several rules found, the grammar (set of rules) is ambiguous - something wrong with it!



# Parser

Ambiguity can lead to understanding a given sentence in two different ways which allows assigning two meanings to the same sentence, some thing highly undesirable, e.g. 1 * 2 + 3 can be interpreted as understanding 1 to be multiplied by 2 and the result to be added to 3...

OR...

2 and 3 to be added together first and then multiply 1 to the result.

# Ambiguous Parsing Quiz

Given:

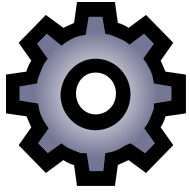
```
if x==1 then | if y==2 print 1 | else print 2
```

*Which grammar rule should be applied?*

- ☐ `stmt -> if expr then stmt`
- ☐ `stmt -> if expr then stmt else stmt`

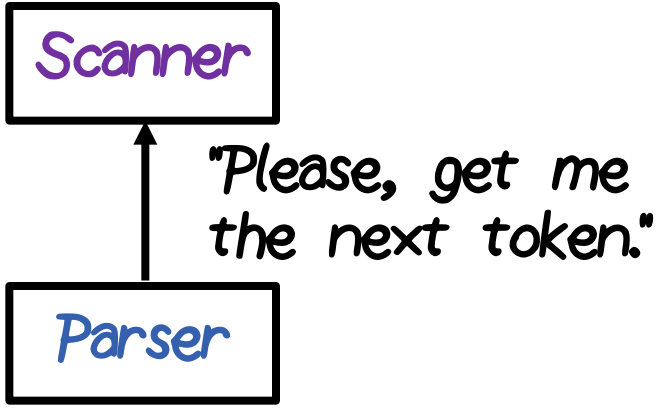
# Ambiguous Parsing Quiz

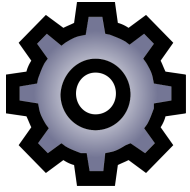
```
stmt           →  matched_stmt | unmatched_stmt / other_stmt
matched_stmt   → if expr then matched_stmt else matched_stmt
                 | other_stmt
unmatched_stmt → if expr then stmt
                 | if expr then matched_stmt else unmatched_stmt
```



# Scanning and Parsing

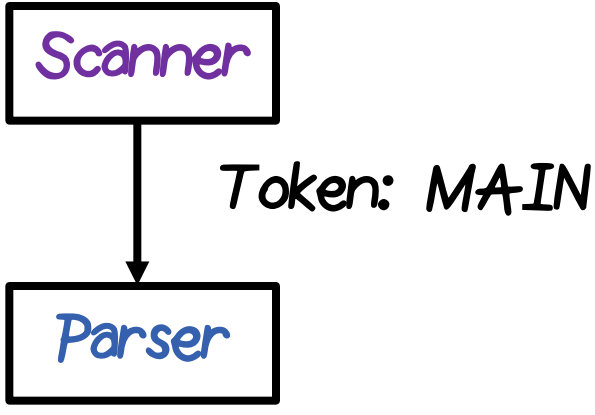
```
main() {
   int a,b;
   a = b;
}
```



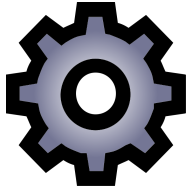


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

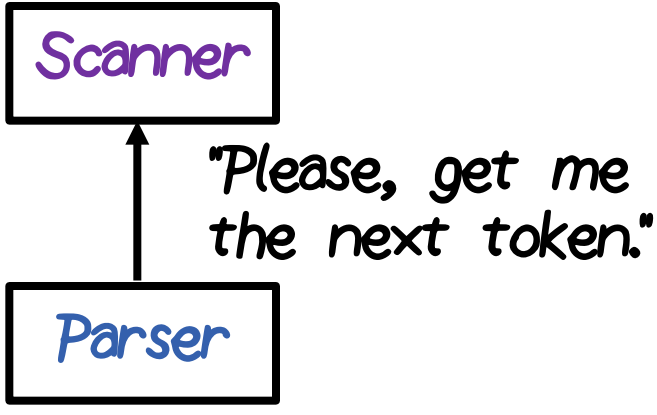



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>

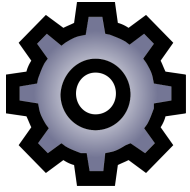


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

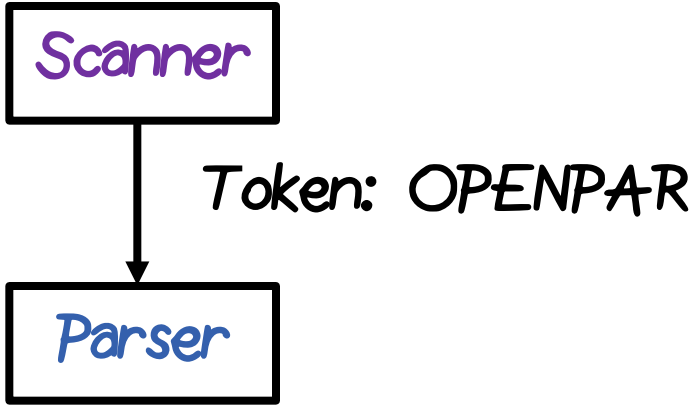



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>

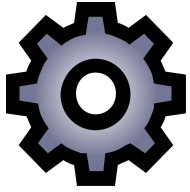


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

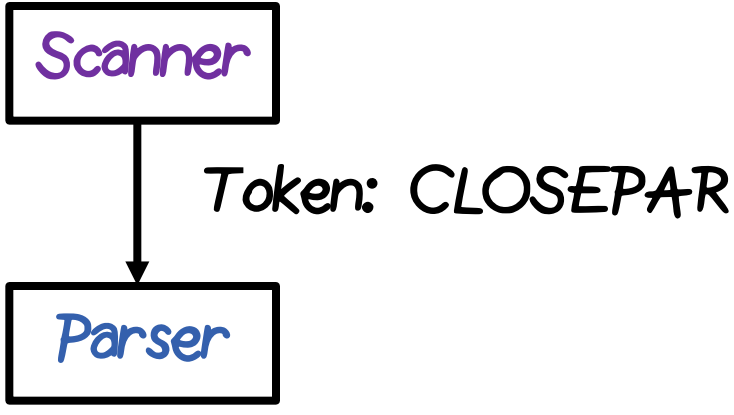



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>

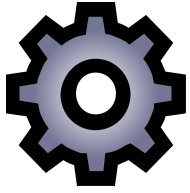


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

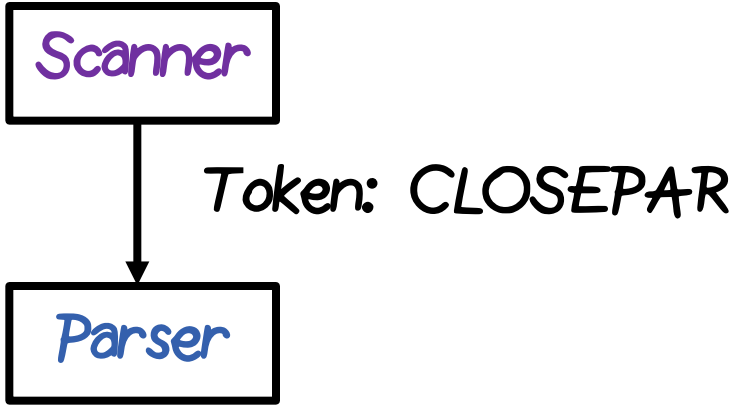



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<PARAMETERS> → NULL

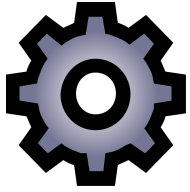


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

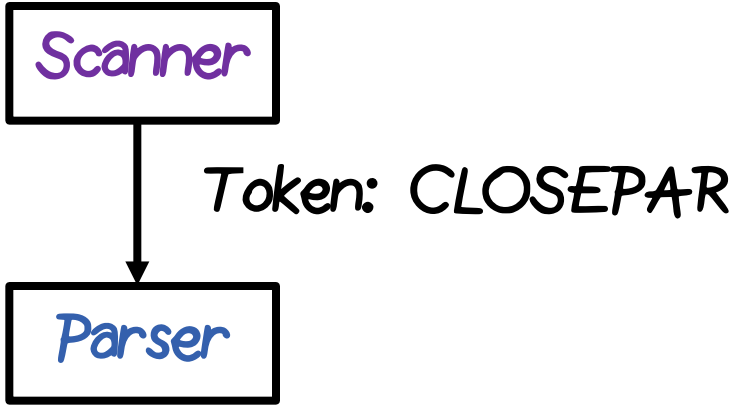



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<PARAMETERS> → NULL

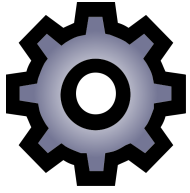


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

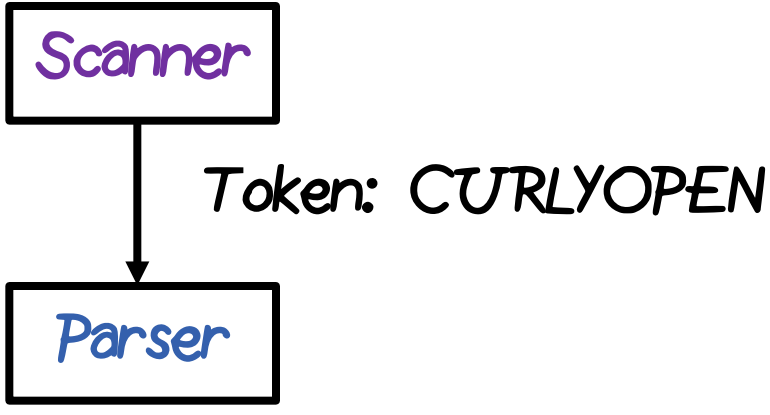



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>

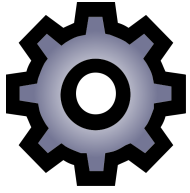


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

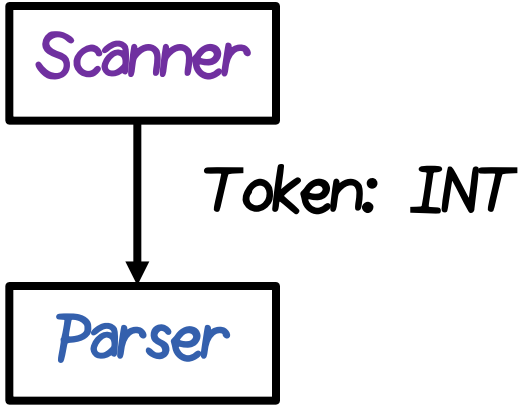



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

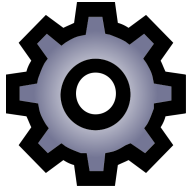


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

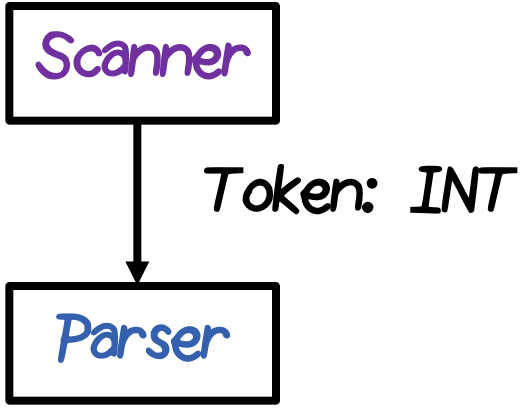



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<DECL-STMT> → <TYPE>VAR<VAR-LIST>;
<TYPE> → INT

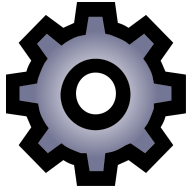


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

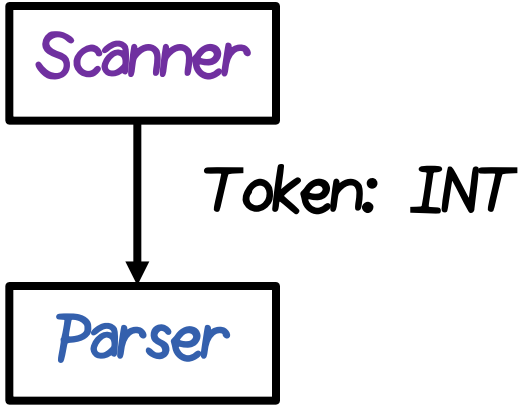



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<DECL-STMT> → <TYPE>VAR<VAR-LIST>;
<TYPE> → INT

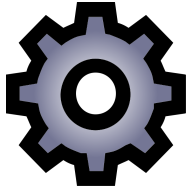


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

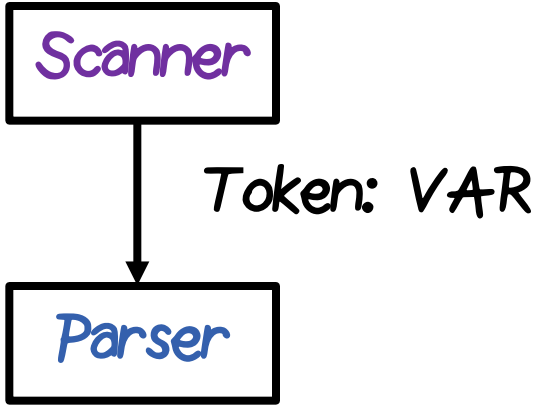



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<DECL-STMT> → <TYPE>VAR<VAR-LIST>;
<TYPE> → INT

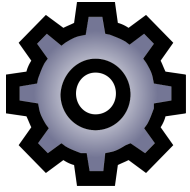


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

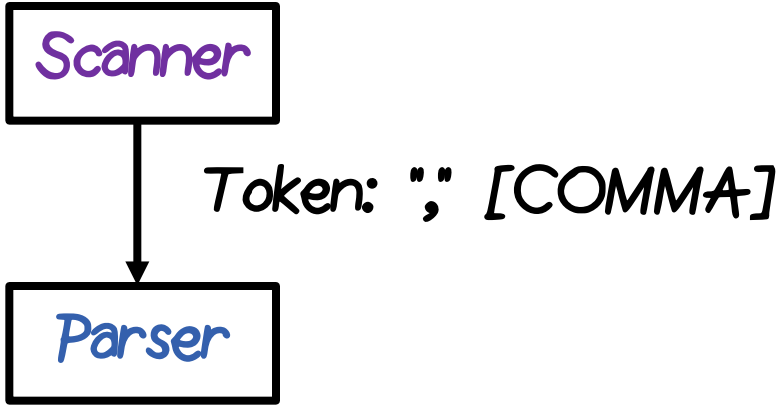



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<DECL-STMT> → <TYPE>VAR<VAR-LIST>;
<VARLIST> → , VAR  <VARLIST>
<VARLIST> → NULL

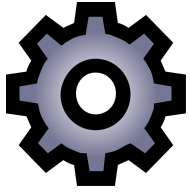


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

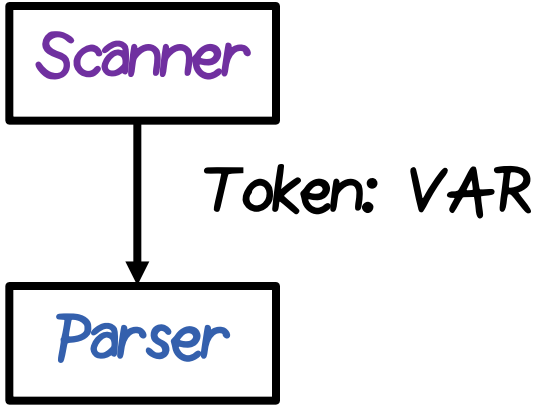



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<DECL-STMT> → <TYPE>VAR<VAR-LIST>;
<VARLIST> → , VAR  <VARLIST>
<VARLIST> → NULL

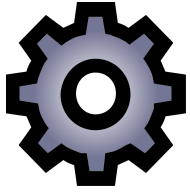


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

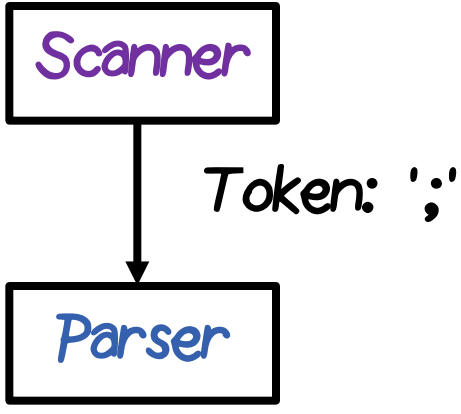



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<DECL-STMT> → <TYPE>VAR<VAR-LIST>;
<VARLIST> → , VAR  <VARLIST>
<VARLIST> → NULL

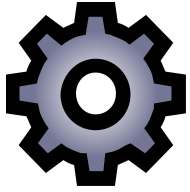


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

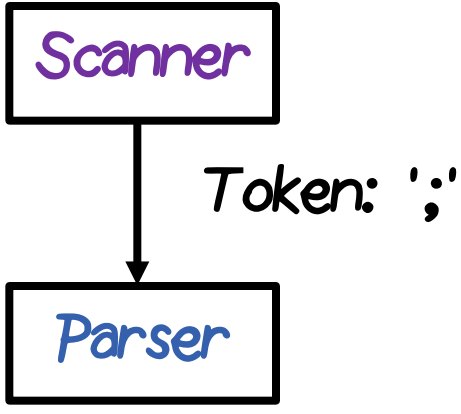



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<DECL-STMT> → <TYPE>VAR<VAR-LIST>;
<VARLIST> → , VAR  <VARLIST>
<VARLIST> → NULL

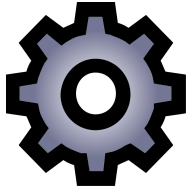


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

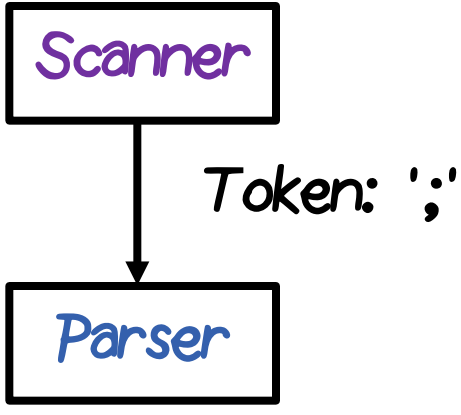



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<DECL-STMT> → <TYPE>VAR<VAR-LIST>;
<VARLIST> → , VAR  <VARLIST>
<VARLIST> → NULL

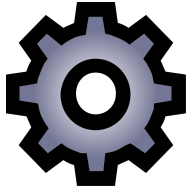


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

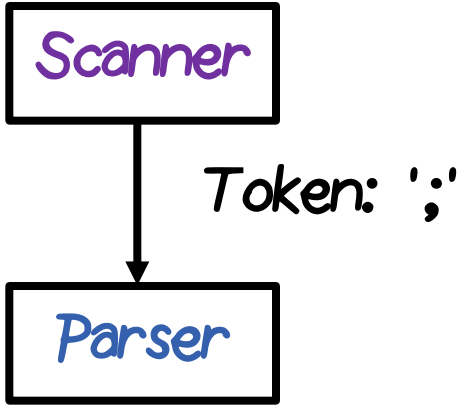



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<DECL-STMT> → <TYPE>VAR<VAR-LIST>;
<VARLIST> → , VAR  <VARLIST>
<VARLIST> → NULL

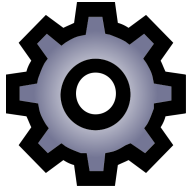


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

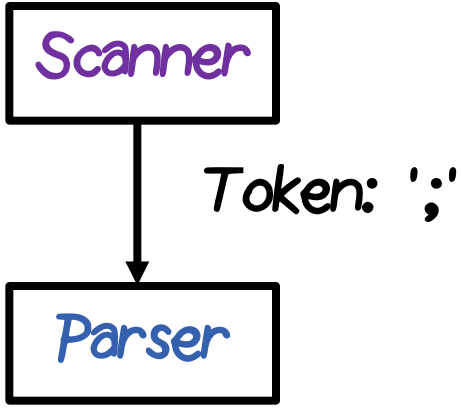



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<DECL-STMT> → <TYPE>VAR<VAR-LIST>;
<VARLIST> → , VAR  <VARLIST>
<VARLIST> → NULL

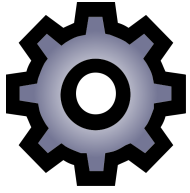


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

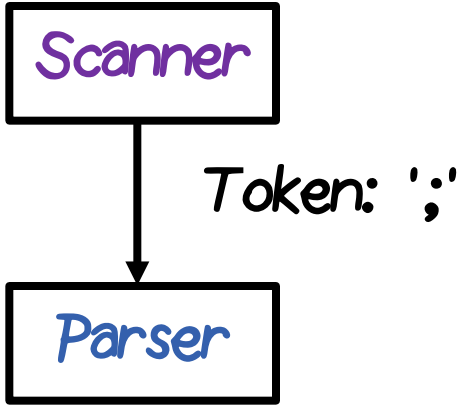



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<DECL-STMT> → <TYPE>VAR<VAR-LIST>;

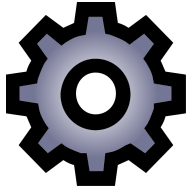


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

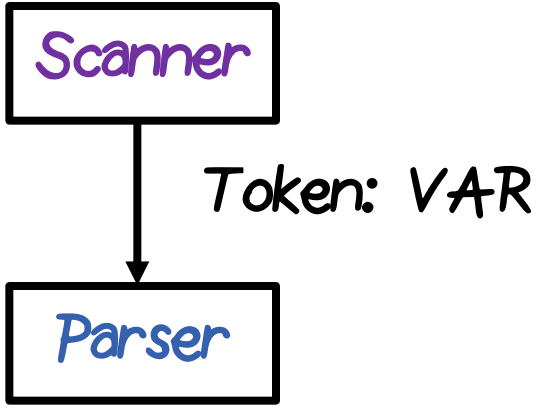



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<DECL-STMT> → <TYPE>VAR<VAR-LIST>;

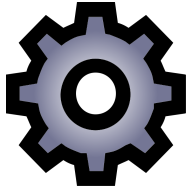


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

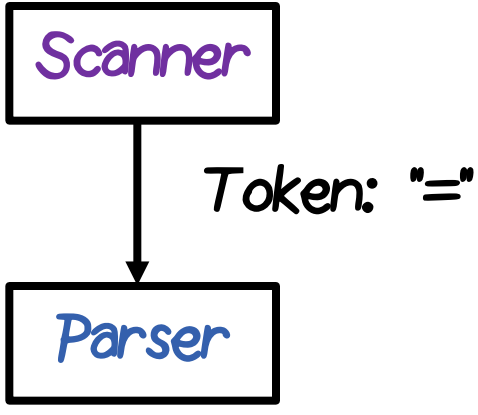



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<ASSIGN-STMT> → VAR = <EXPR>;
<EXPR> → VAR

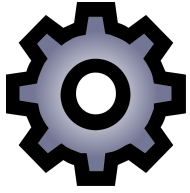


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

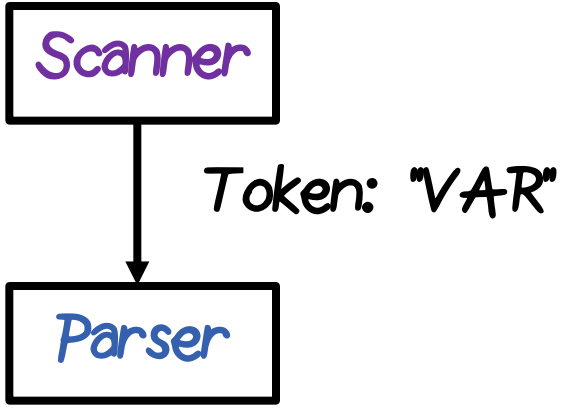



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<ASSIGN-STMT> → VAR = <EXPR>;
<EXPR> → VAR

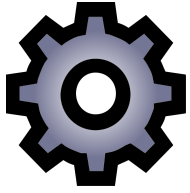


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

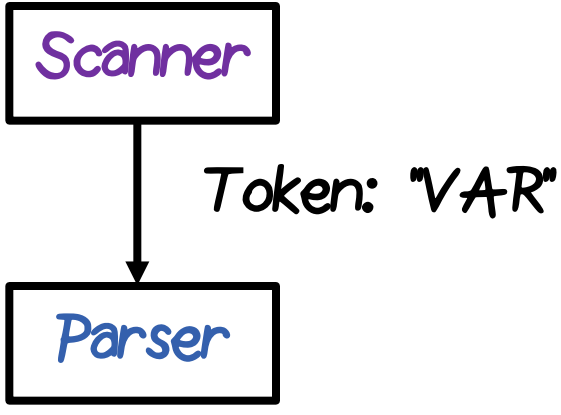



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<ASSIGN-STMT> → VAR = <EXPR>;
<EXPR> → VAR

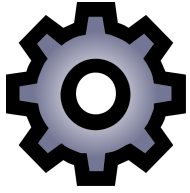


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

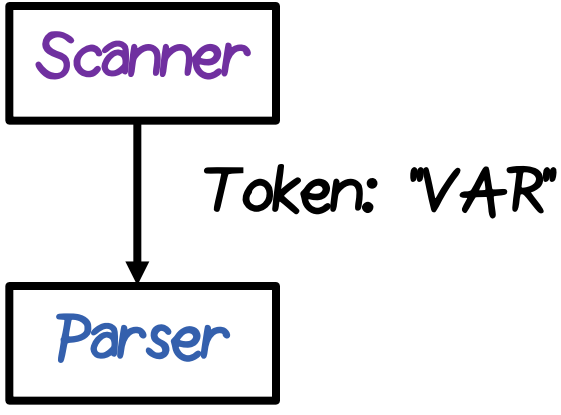



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<ASSIGN-STMT> → VAR = <EXPR>;
<EXPR> → VAR

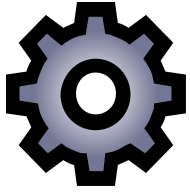


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

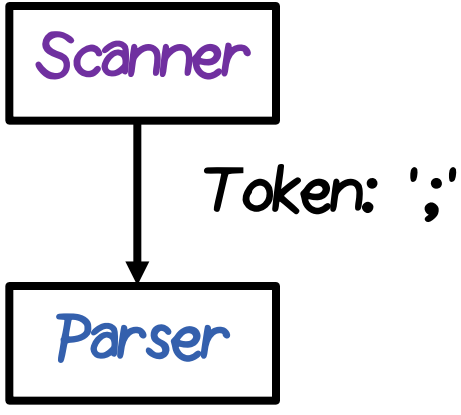



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<ASSIGN-STMT> → VAR = <EXPR>;
<EXPR> → VAR

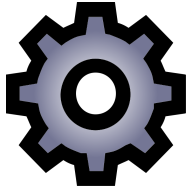


# Scanning and Parsing

```
main() {
    int a,b;
    a = b;
}
```

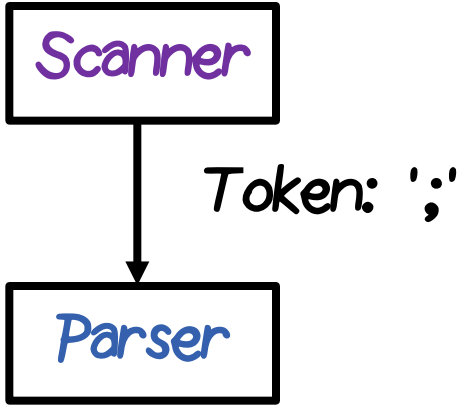



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<ASSIGN-STMT> → VAR = <EXPR>;

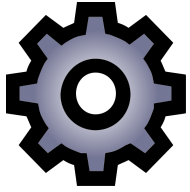


# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```

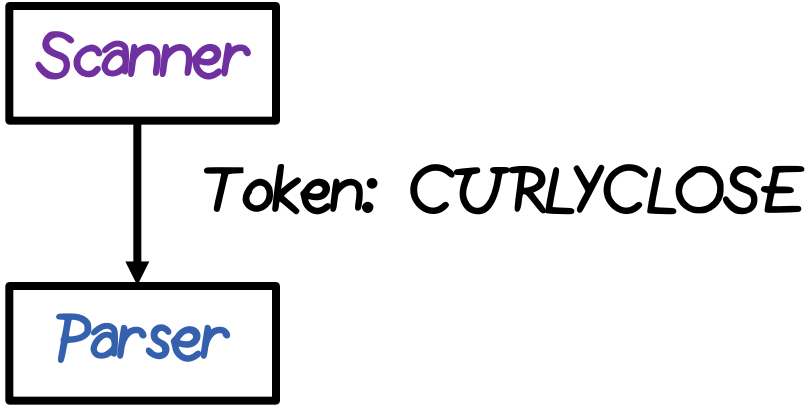



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<ASSIGN-STMT> → VAR = <EXPR>;



# Scanning and Parsing

```
main() {
   int a,b;
   a = b;
}
```




<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

# Syntax vs Semantics Quiz

Assuming we are using 'C' programming language, determine which statements are syntactically incorrect (A), which are semantically incorrect (B), and which are both syntactically and semantically incorrect (C).

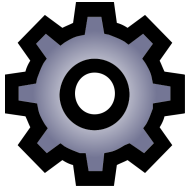
```
int main() {
            int a,b,c;
```

Statement 3 is both syntactically and semantically incorrect, missing operand after 2nd + (syntax error), and undeclared d causing semantic error

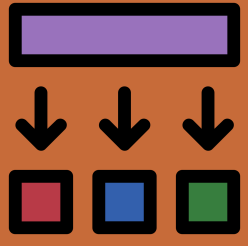
Statement 4 is semantically incorrect due to undeclared variable d.

```
    d = a + b;      /* 4 */
}
```

B

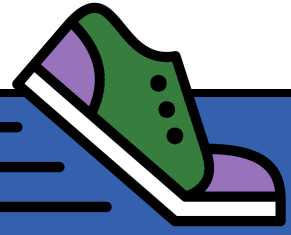


# Syntax vs Semantics



During parsing?
(Syntax checking)

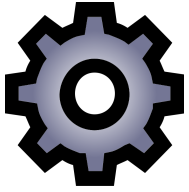
- Tokens are matched and consumed by parser
- Symbol tables
- Variables have attributes
- Declaration attached attributes to variables



Semantic actions

- What are semantic actions?

Semantic checks



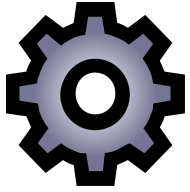
# Symbol Table

`int a,b;` →

Declares a and b, Within current scope, Type integer

Use of a and b now legal

| Basic Symbol Table | | |
|---|---|---|
| Name | Type | Scope |
| a | int | ”main” |
| b | int | ”main” |



# Semantic Actions

Enter variable declaration into symbol table

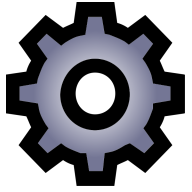
Look up variables in symbol table

Do binding of looked-up variables (scoping rules, etc.)

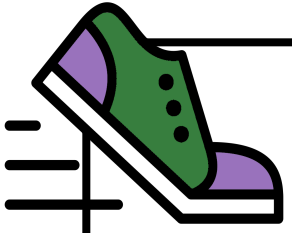
Do type checking for compatibility

Keep the semantic context of processing

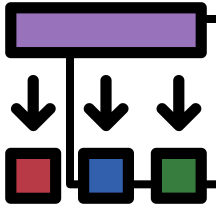
```
a + b + c -> t1 = a + b
             t2 = t1 + c
```



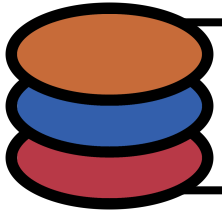
# Semantic Actions



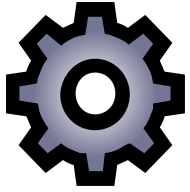
**Action symbols embedded in the grammar.**
Each action symbol represents a semantic procedure.
These procedures do things and/or return values.



**Semantic procedures are called by parser** at appropriate places during parsing

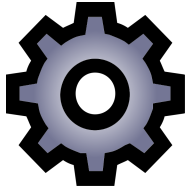


**Semantic stack** implements & stores semantic records

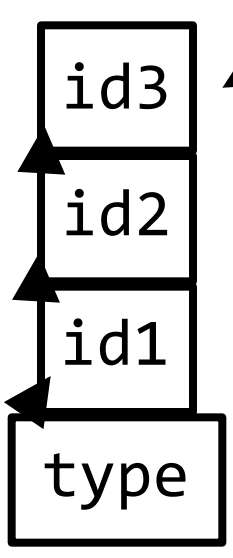


# Semantic Actions

```
<decl-stmt>   → <type>#put-type<varlist>#do-decl
<type>        → int | float
<varlist>     → <var>#add-decl <varlist>
<varlist>     → <var>#add-decl
<var>         → ID#proc-decl
#put-type     puts given type on semantic stack
#proc-decl    builds decl record for var on stack
#add-decl     builds decl-chain
#do-decl      traverses chain on semantic stack using
              backwards pointers entering each var into
              symbol table
```

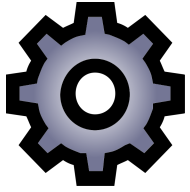


# Semantic Actions




#do-decl
⇒

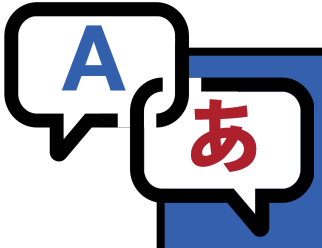
| Name | Type | Scope |
|---|---|---|
| id1 | 1 | 3 |
| id2 | 1 | 3 |
| id3 | 1 | 3 |



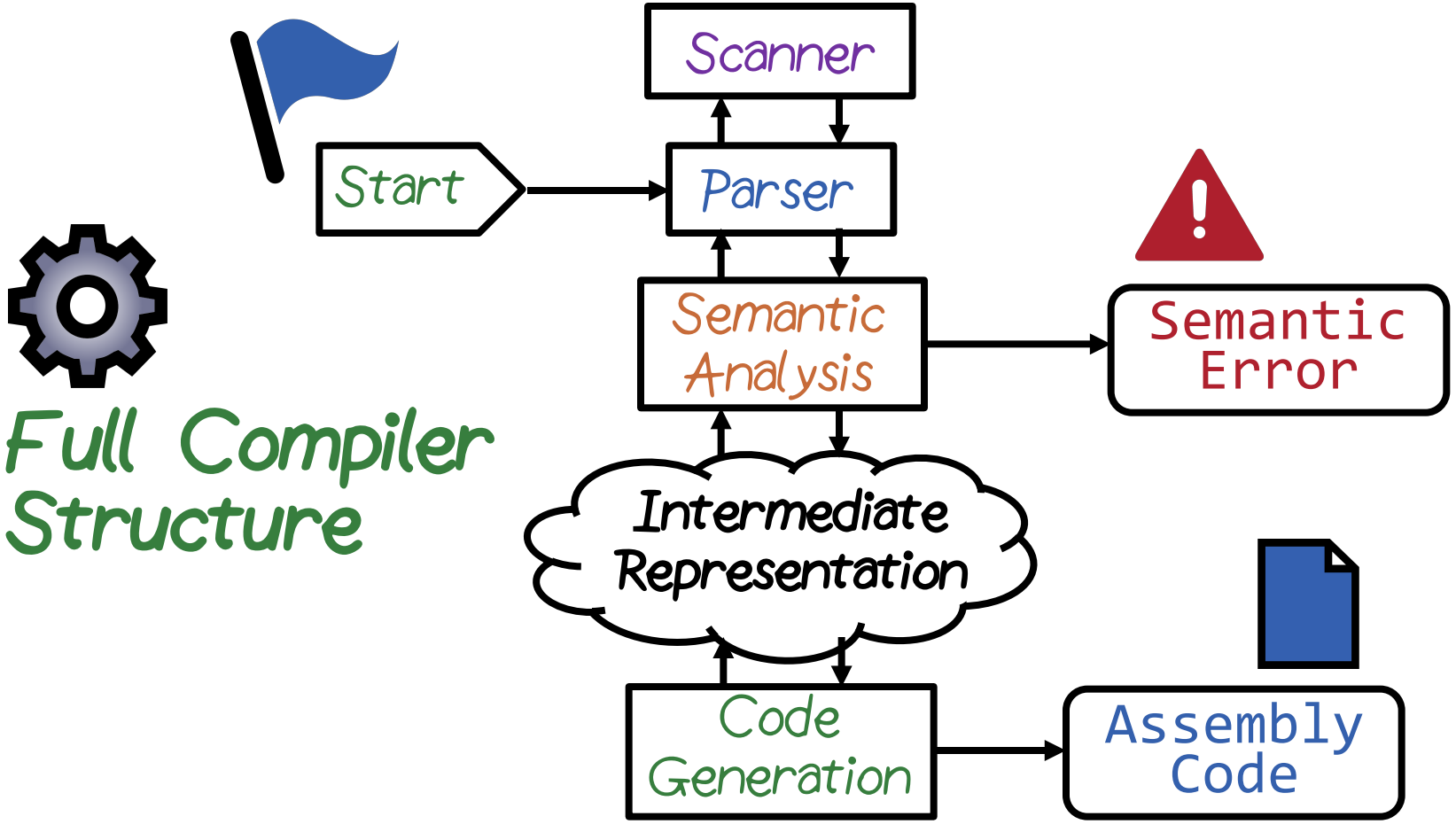
# Semantic Actions

Two types of semantic actions:

Checking: binding, type compatibility, scoping, etc




Translation: generate temporary values, propagate them to keep semantic context


Full Compiler Structure
Scanner
Start
Parser
Semantic Analysis
Semantic Error
Intermediate Representation
Code Generation
Assembly Code