CS 4240: Compilers

Lecture 20: Lexical Analysis (Scanning)

Instructor: Vivek Sarkar
(vsarkar@gatech.edu)

April 1, 2019

ANNOUNCEMENTS & REMINDERS

- **Project 2 due by 11:59pm on Wednesday, April 3rd**
  - 15% of course grade
  - Next lecture (April 3rd) will be an overview of the ANTLR scanning & parsing framework
- FINAL EXAM: Wednesday, May 1, 2:40 pm - 5:30 pm
  - 30% of course grade

Worksheet #18
Solution

(From Lecture #18 given on 03/25/2019)

Worksheet problem: : Find a spill-free register allocation for symbolic registers s_A, s_B, s_C in the program shown below, assuming that there are $k = 2$ physical registers available.

switch (…) {					
	case 0:		case 1:		case 2:
i_1:	$s_A := \ldots$	i_4:	$s_B := \ldots$	i_7:	$s_A := \ldots$
i_2:	$s_B := \ldots$	i_5:	$s_C := \ldots$	i_8:	$s_C := \ldots$
i_3:	$\ldots := s_A \; op \; s_B$	i_6:	$\ldots := s_B \; op \; s_C$	i_9:	$\ldots := s_A \; op \; s_C$
	break;		break;		break;
}					

Worksheet problem: : Find a spill-free register allocation for symbolic registers s_A, s_B, s_C in the program shown below, assuming that there are $k = 2$ physical registers available.

switch (…) {

case 0:

i_1: $s_A := \ldots$
i_2: $s_B := \ldots$
i_3: $\ldots := s_A \; op \; s_B$

break;

case 1:

i_4: $s_B := \ldots$
i_5: $s_C := \ldots$
i_6: $\ldots := s_B \; op \; s_C$

break;

case 2:

i_7: $s_A := \ldots$
i_8: $s_C := \ldots$
i_9: $\ldots := s_A \; op \; s_C$

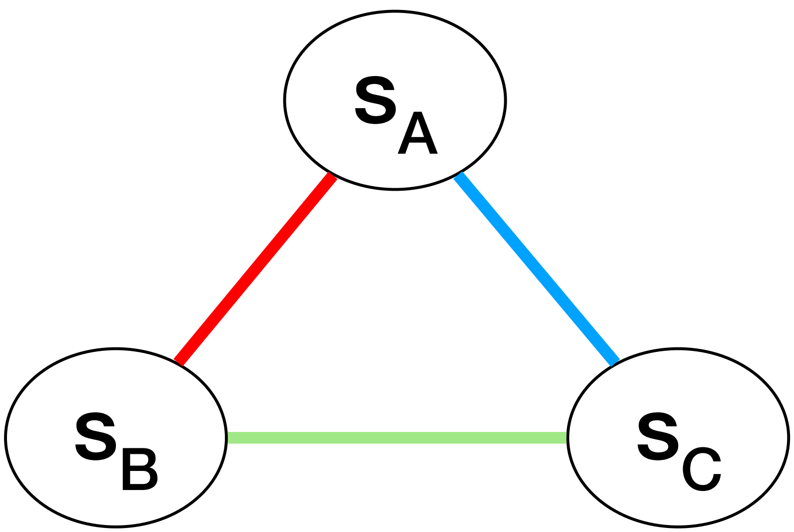
break;

}

s_A and s_B have conflicting live ranges.

s_B and s_C have conflicting live ranges.

s_A and s_C have conflicting live ranges.



Interference graph needs 3 colors

Worksheet problem: Find a spill-free register allocation for symbolic registers s_A, s_B, s_C in the program shown below, assuming that there are $k = 2$ physical registers available.

switch (...) {

case 0:

i_1: s_A := ...
i_2: s_B := ...
i_3: ... := s_A *op* s_B

break;

case 1:

i_4: s_B := ...
i_5: s_C := ...
i_6: ... := s_B *op* s_C

break;

case 2:

i_7: s_A := ...
i_8: s_C := ...
i_9: ... := s_A *op* s_C

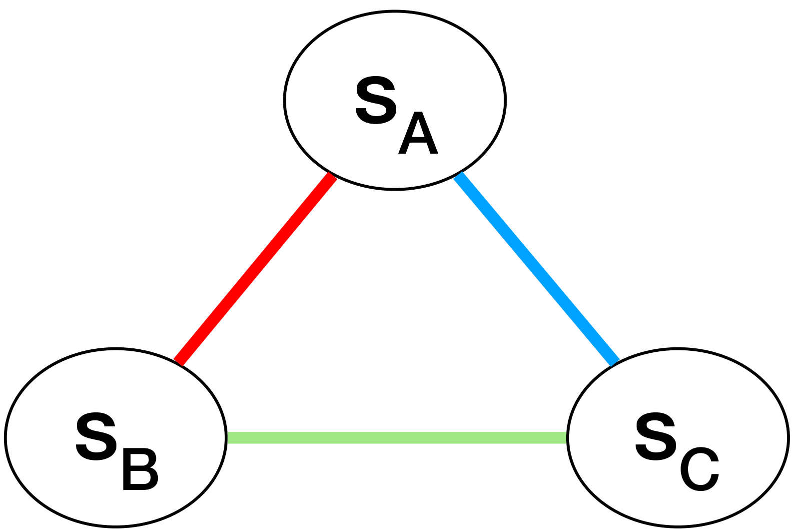
break;

}

s_A : R0
s_B : R1

s_B : R0
s_C : R1

s_A : R0
s_C : R1

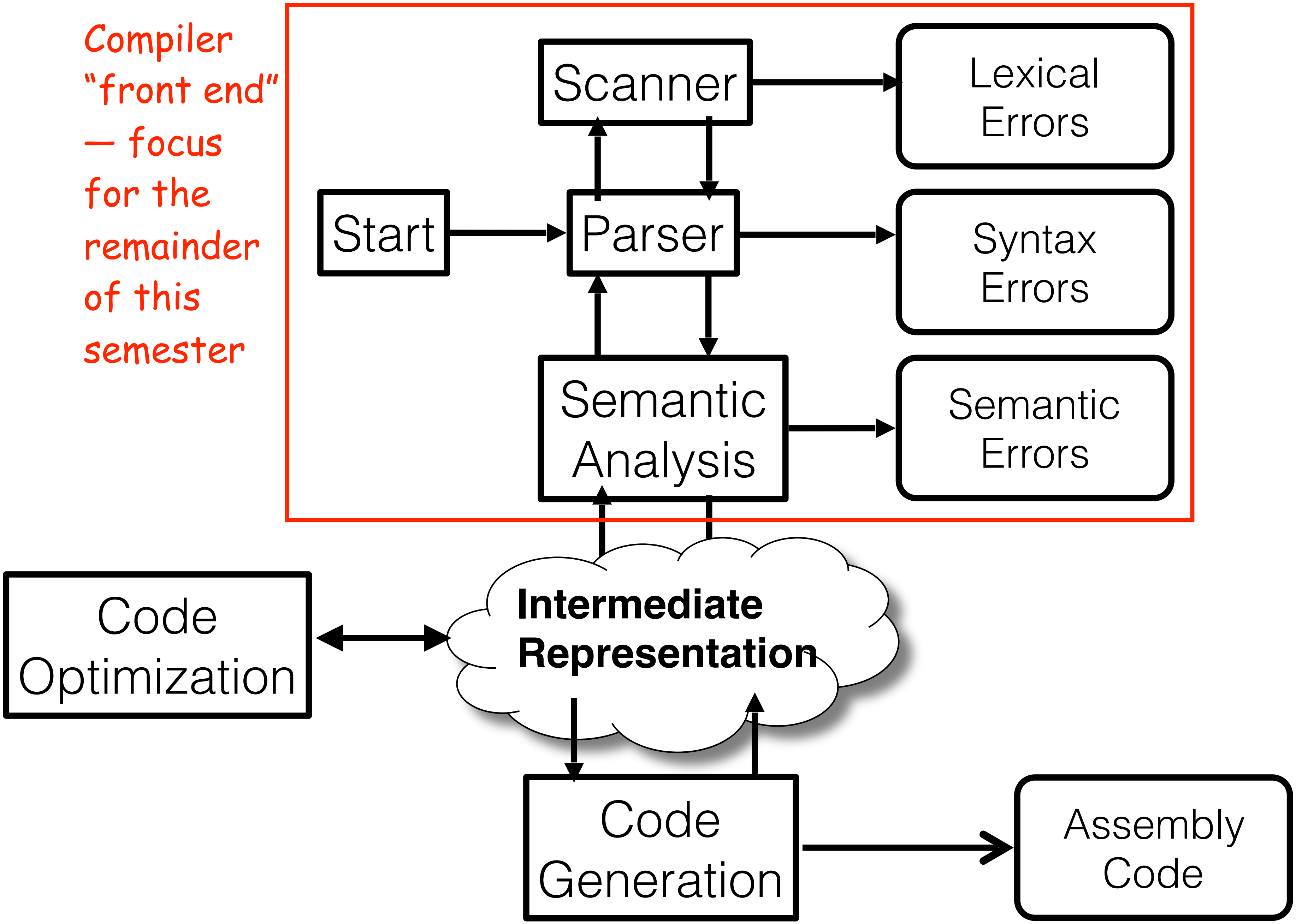


However, it is possible to do a spill-free register allocation with just 2 registers

(no register copy statements needed in this case)

Structure of a Full Compiler (Recap from Lecture 1)

Compiler "front end" — focus for the remainder of this semester



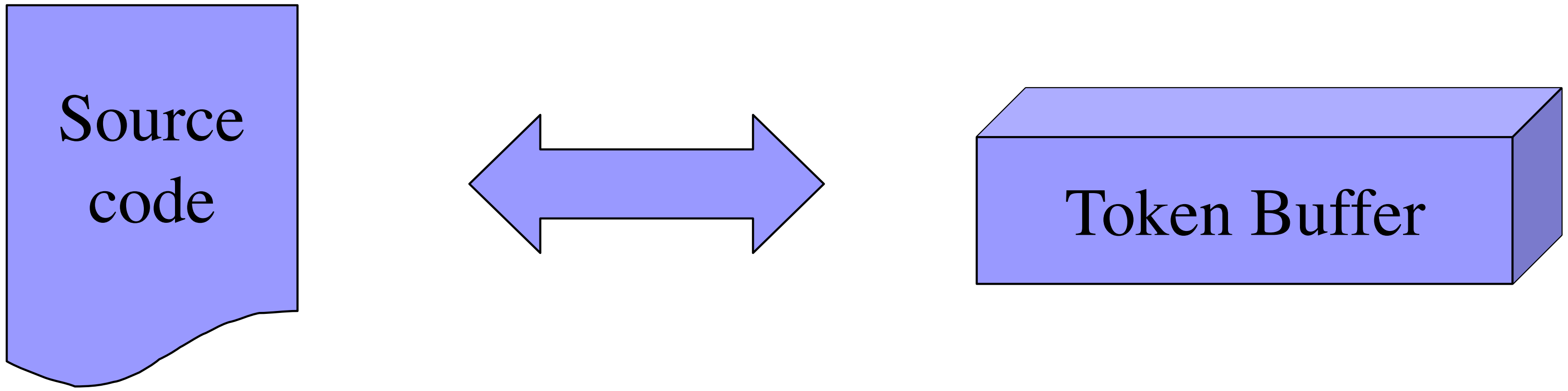
Front-end architecture

- **Scanning**: converting source code into stream of known chunks called **tokens**
  - **Lexical** rules of language dictate how legal **token** is formed as a sequence of **alphabet symbols**
- **Parsing**: building **tree-based** representation of code
  - **Grammar** dictates how legal **tree** is formed as a sequence of **tokens**

Overall Operation

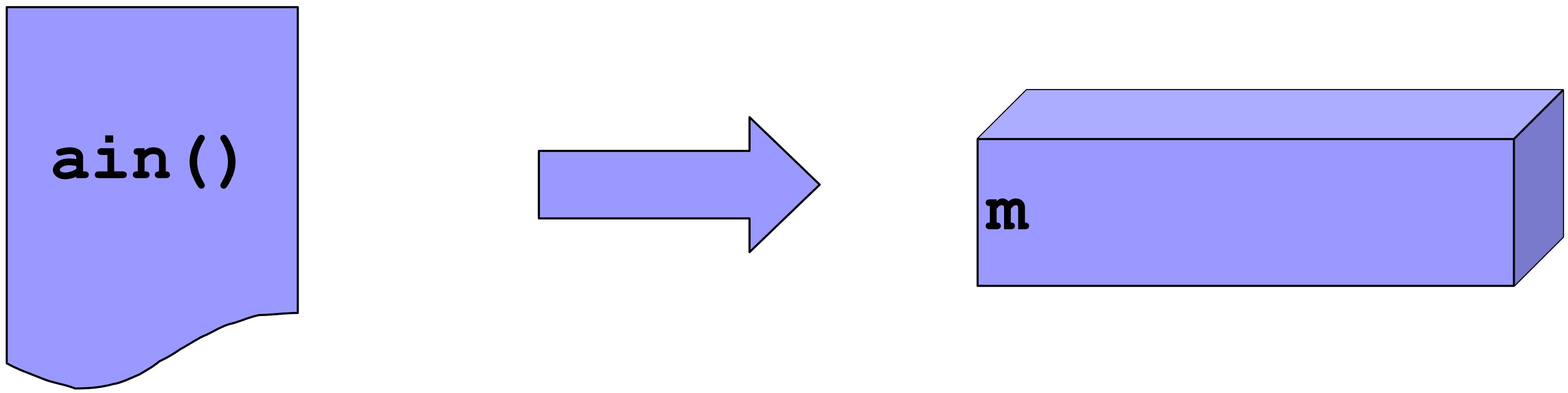
- Parser is in control of the overall operation
  - Demands scanner to produce a token
- Scanner reads input file into token buffer & forms a token (based on specification of tokens as *regular expressions*)
  - Token is returned to parser
- Parser attempts to match the token (based on specification of syntax as a *context-free grammar*)
  - Failure: Syntax Error!
  - Success:
    - Optionally performs a “semantic action”
    - Returns to get next token, or handle end of input

Scanning/Tokenization

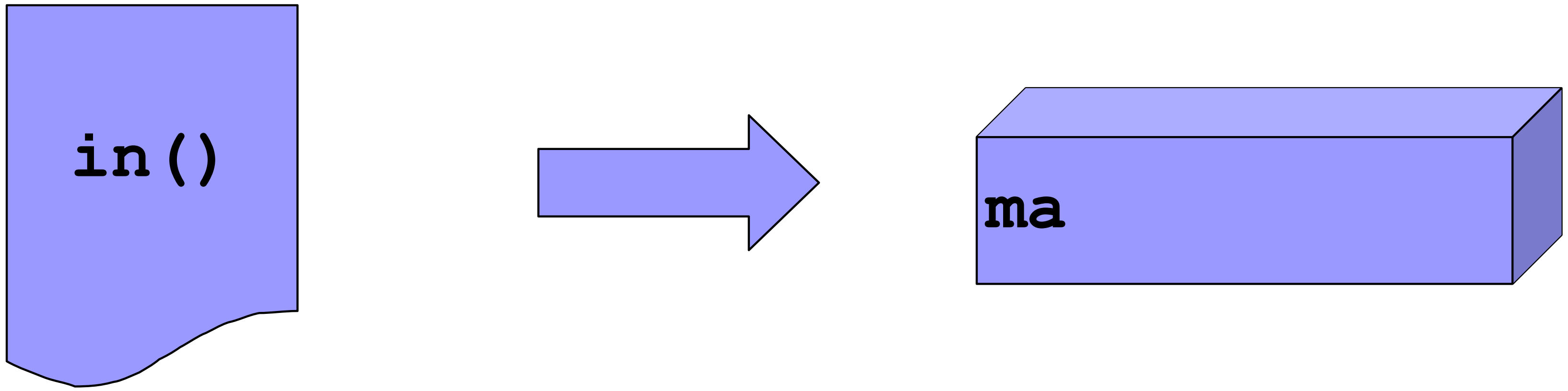


- What does the Token Buffer contain?
  - Token being identified
- Why a two-way ( ) street?
  - Characters can be read
  - and unread
  - Termination of a token

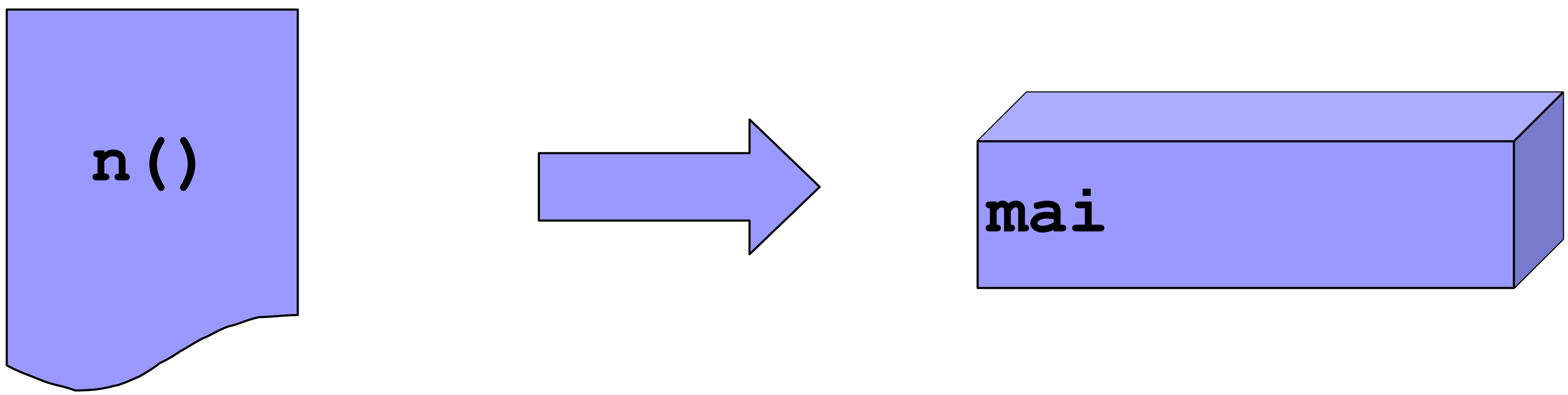
Example



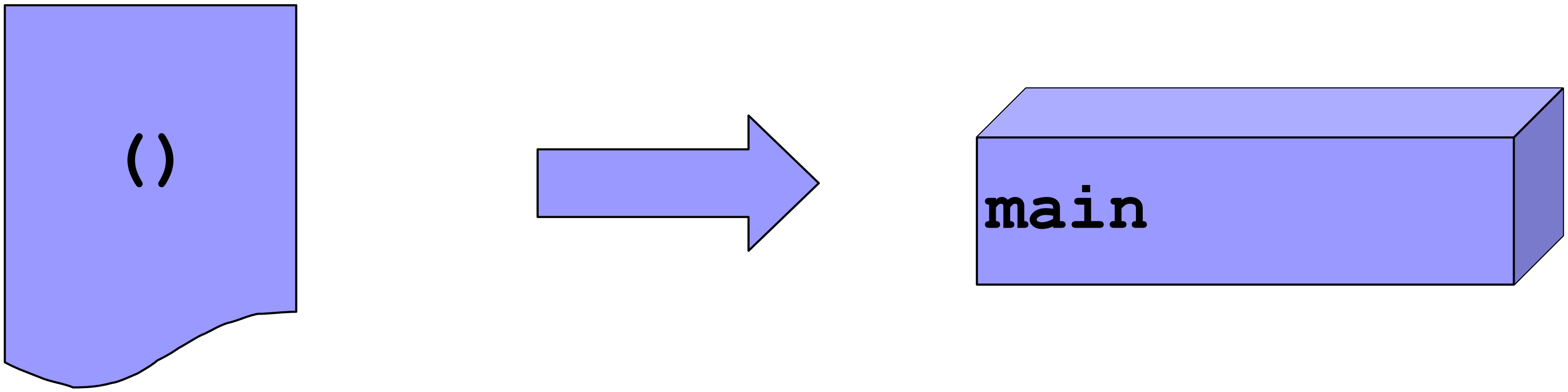
Example



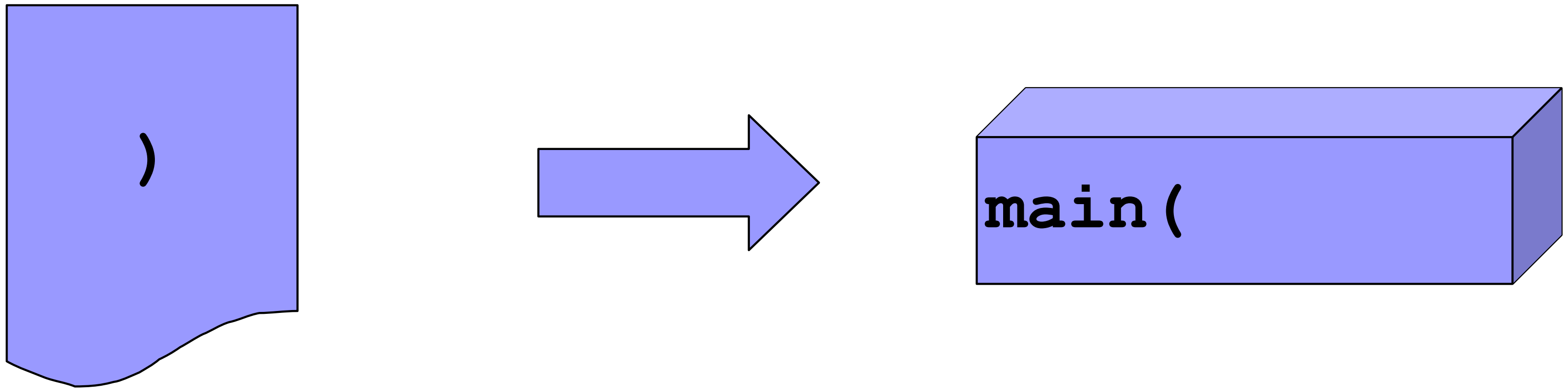
Example



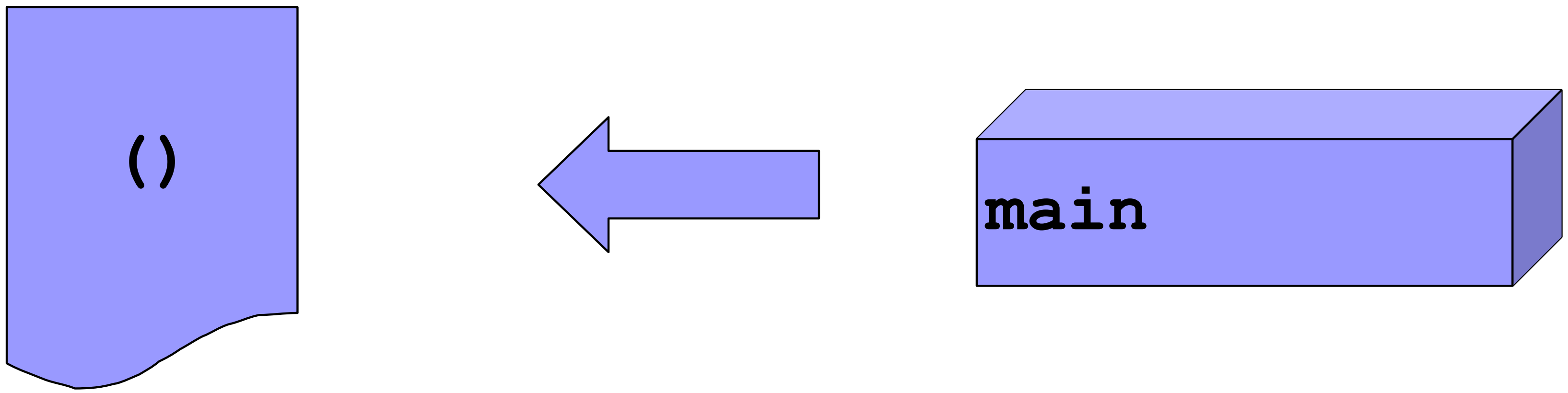
Example



Example



Example



Identifier: **main**

When can scanning be hard?

Poor language design can complicate scanning

- Reserved words are important

  if then then then = else; else else = then (PL/I)

- Insignificant blanks (Fortran & Algol68)

  do 10 i = 1,25

  do 10 i = 1.25

- String constants with special characters (C, C++, Java, …)

  newline, tab, quote, comment delimiters, …

- Finite closures (Fortran 66 & Basic)
  - Limited identifier length
  - Adds states to count length

What can be so hard? (Fortran 66/77)

```
      INTEGERFUNCTIONA
      PARAMETER(A=6,B=2)
      IMPLICIT CHARACTER*(A-B)(A-B)
      INTEGER FORMAT(10), IF(10), DO9E1
100   FORMAT(4H)=(3)
200   FORMAT(4   )=(3)
      DO9E1=1
      DO9E1=1,2
9     IF(X)=1
      IF(X)H=1
      IF(X)300,200
300   CONTINUE
      END
C     THIS IS A “COMMENT CARD”
    $ FILE(1)
      END
```

How does a compiler scan this?

- First pass finds & inserts blanks
- Can add extra words or tags to create a scanable language
- Second pass is normal scanner

Example due to Dr. F.K. Zadeck

Lexical Analysis

- Read input one character at a time
  - Group characters into tokens
  - Remove white space and comments
  - Encode token types
- A lexical language tells us which are legal strings in that language
- Lexical rules specify how alphabet can be combined to form legal strings

Basics of Regular Expression

- **Symbols** and **alphabet**
  - A **symbol** is a valid character in a language
  - An **alphabet** is a *set* of legal symbols (typically denoted as Σ)
- **Strings** and **languages**
  - A **string** is a (possibly empty) sequence of symbols drawn from an alphabet
  - A **language** is a (possibly empty) *set* of strings
- **Metacharacters/metasymbols** that have special meanings
  - Construct regex's (e.g. **ϕ**, **ε**, **|**, **(**, **)**, *, **+**, etc.)
  - Treat as symbol by using escape character (\), e.g., \+

Language, L(r), denoted by regex r

- Atomic regular expressions
  - For each character **a** $\in$**Σ**, **L(a) = { a }**
  - Empty string: **L(ε) = { ε }**
  - Empty set: **L(ϕ) = { }**
- Recursively-defined regular expressions:
  - **Alternation**: for all **a, b** that are regexs, **a** | **b** is a regex that denotes
    - **L(a l b) = L(a) U L(b)**
  - **Concatenation**: for all **a, b** that are regexs, **a b** is a regex that denotes
    - **L(a b) = { s1 s2 l s1** $\in$ **L(a) and s2** $\in$ **L(b)}**
  - **Repetition**: for each **a** that is a regex, **a*** is a regex that denotes
    - **L(a*)** = **L(ε) U L(a) U L(a a) U L(a a a) U …**

Writing regex's

- Precedence of operations
  - Repetition
    tighter than Concatenation
    tighter than Alternation
  - Parentheses change precedence
- Examples of regular expressions
  - **a | b***
  - **(a | b)***
  - **(a | c)* b (a | c)***

Matching: does a string belong to the language denoted by a regex?

- **a b | b a**
  - Matches **a b,** i.e., a b ∈ L(a b | b a)
  - Doesn't match **a a**
- **(a | b) (b | a)**
  - Matches **a a**, **a b**, **b a**, **b b**
  - Doesn't match **a** or **a b a**
- **(a a | b b)***
  - Matches **a a a a**, **a a b b a a**
  - Doesn't match **a b a b**

Regex shorthand notations for convenience

- **a? = (ε | a)**
  - Empty string or **a**
- **a+ = a a***
  - Concatenation of one or more strings drawn from **a**
- **[…]**: character classes
  - **[abcd] = (a | b | c | d)**
  - **[^abcd]**: every symbol except **a, b, c, d**

Regex shorthand over ordered alphabets

- **[…]**: character classes with ranges
  - **[a-d] = [abcd]**
  - **[a-z0-9] = [a-z] | [0-9]**
  - **[^a-z]**: everything but **[a-z]**

- **Example**
  - Match any character that is not a vowel

    [^aeiouAEIOU]

More Examples

- Match any <u>character</u> not a vowel
  `[^aeiouAEIOU]`
- Match any uc or lc consonant
  `[b-df-hj-np-tv-zB-DF-HJ-NP-TV-Z]`

- Regular Expressions are used in grep, sed, awk, perl, vi, shells, lex, yacc
  - Each may use slightly different convention

Compiler-relevant examples

- Matches unsigned integers

  [0-9] [0-9]*
- Matches identifiers:

  [a-zA-Z_] [a-zA-Z_0-9]*
- Matches signed integers:

  (+|-)? [0-9] [0-9]*
- Matches floating point numbers

  **[+-]? [0-9]+ (\. [0-9]*)?
  ([eE] [+-]? [0-9]+)?**

Parser

- Translating code to rules of a grammar
- 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

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
```

Demo

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

Scanner

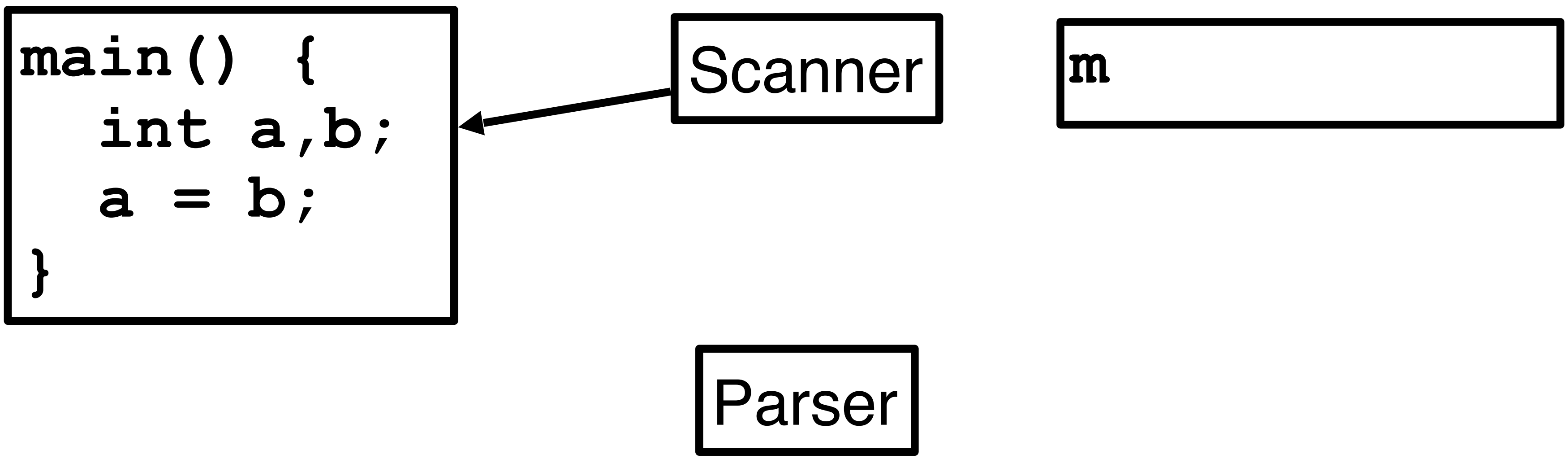
`Token Buffer`

Parser

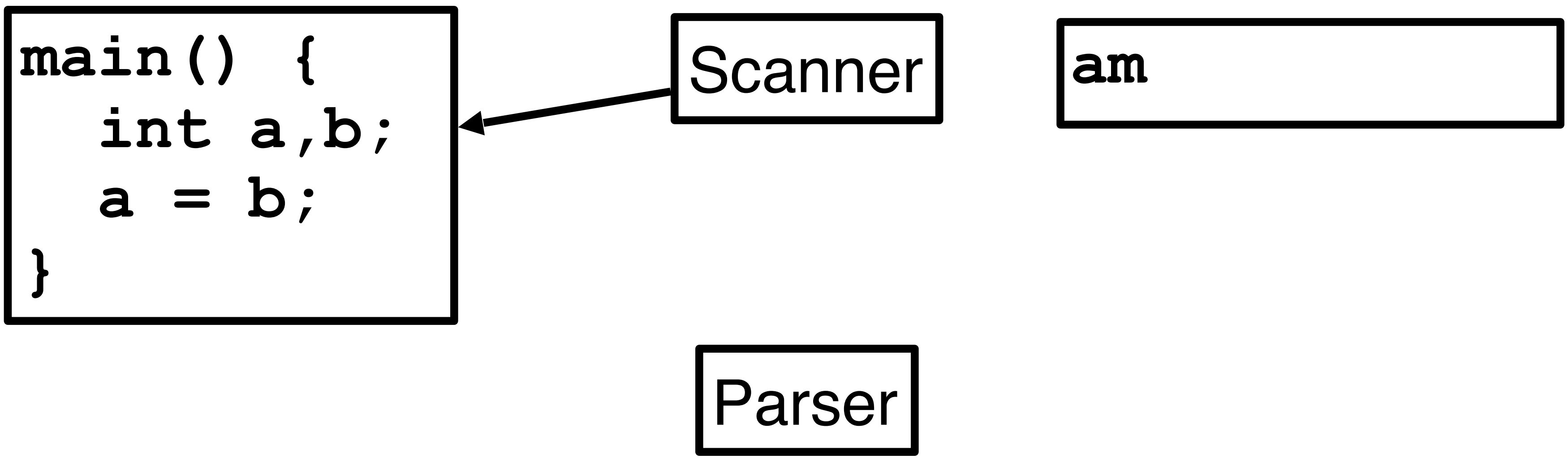
Demo

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

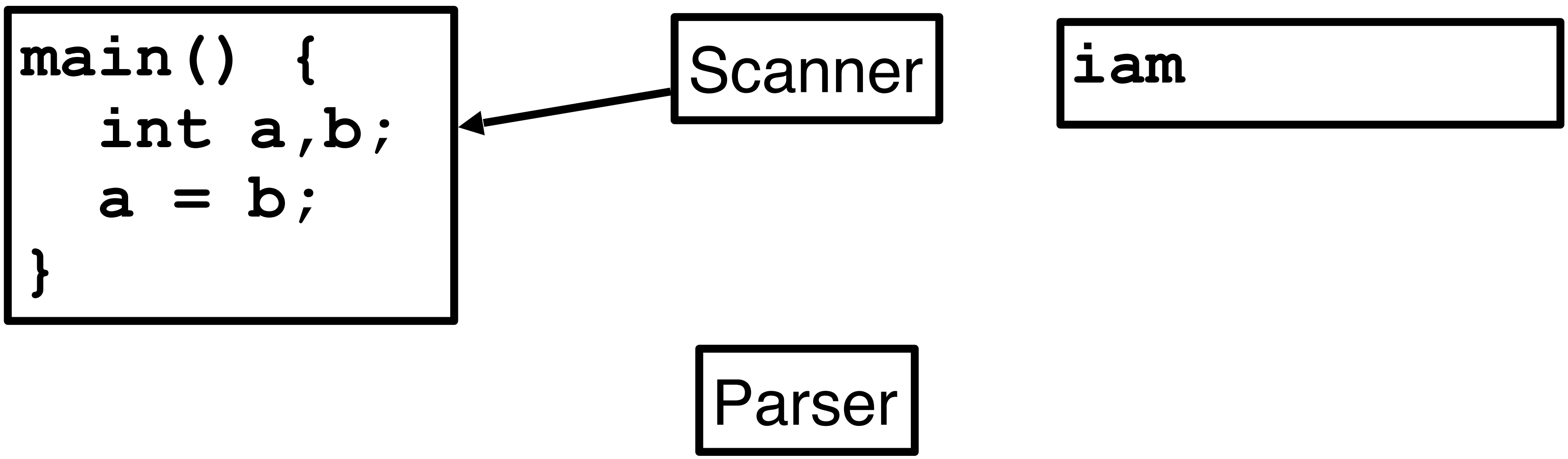
Demo



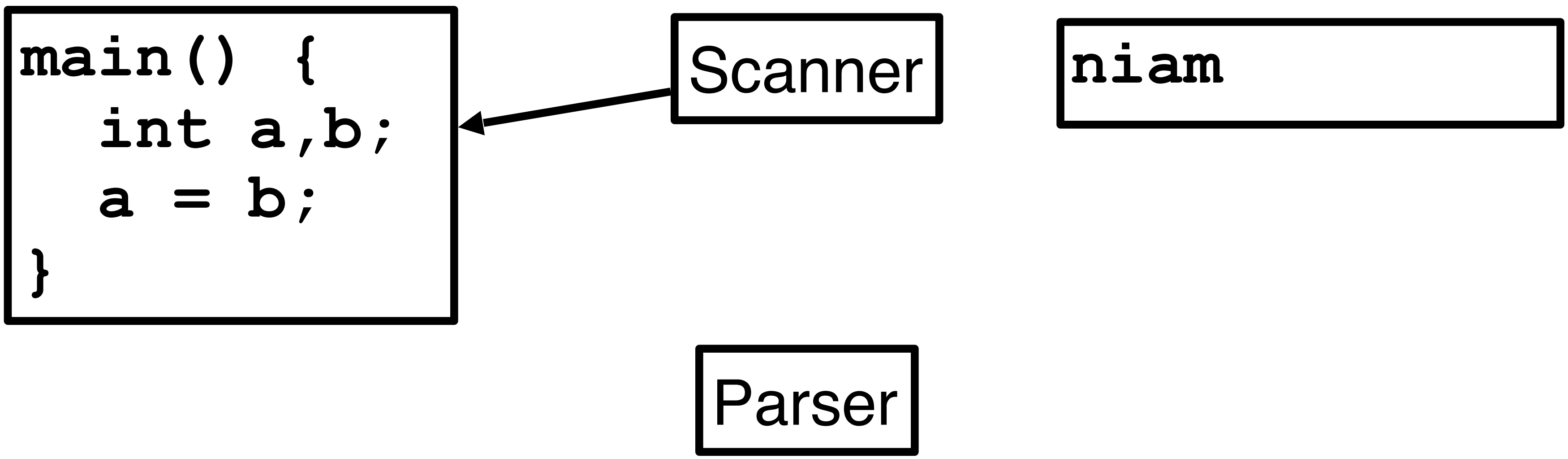
Demo



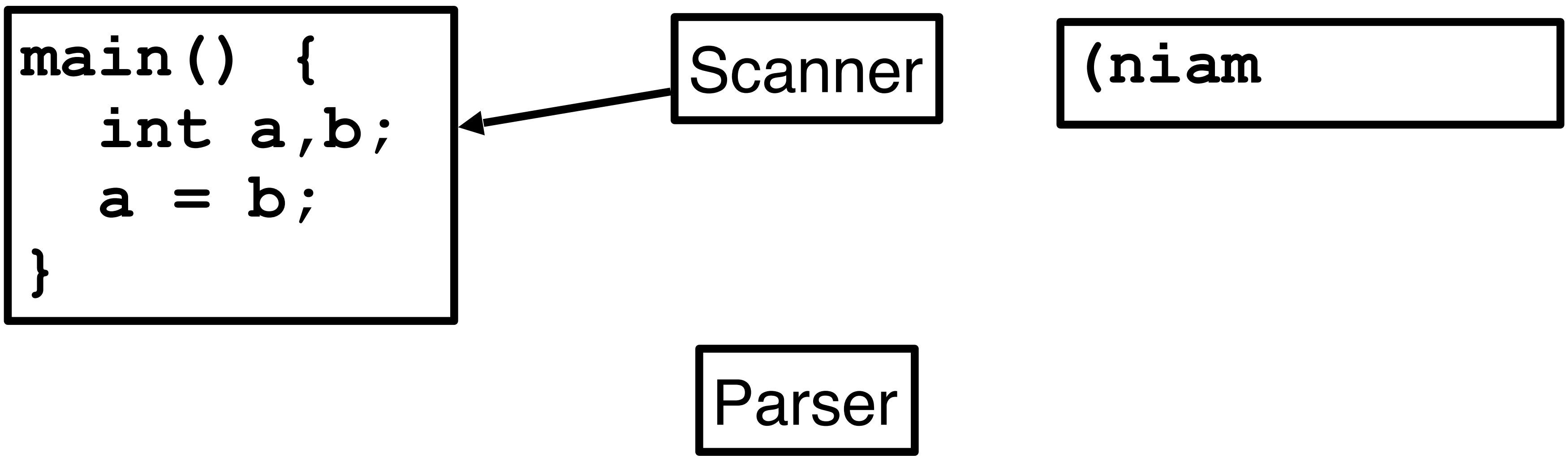
Demo



Demo



Demo



Demo

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

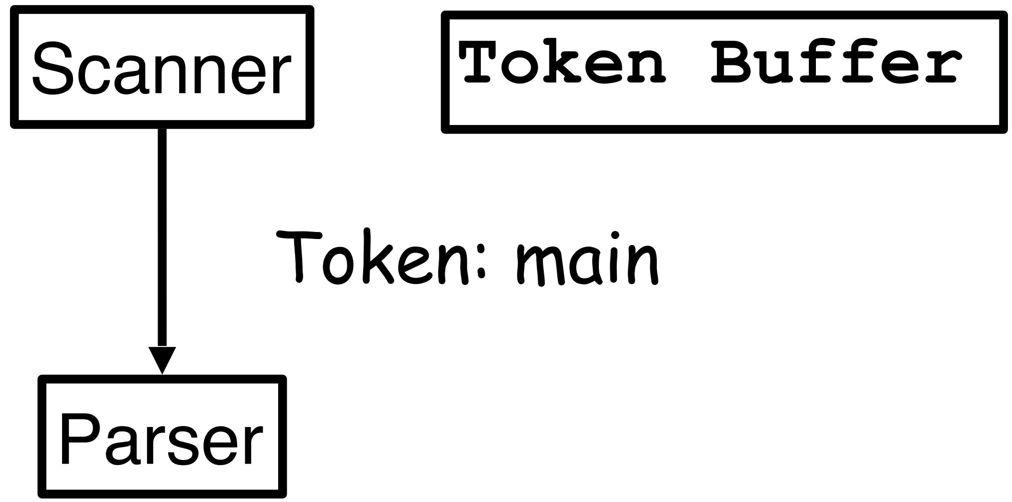
Scanner

niam

Parser

Demo

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



Demo

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

Scanner

`Token Buffer`

Parser

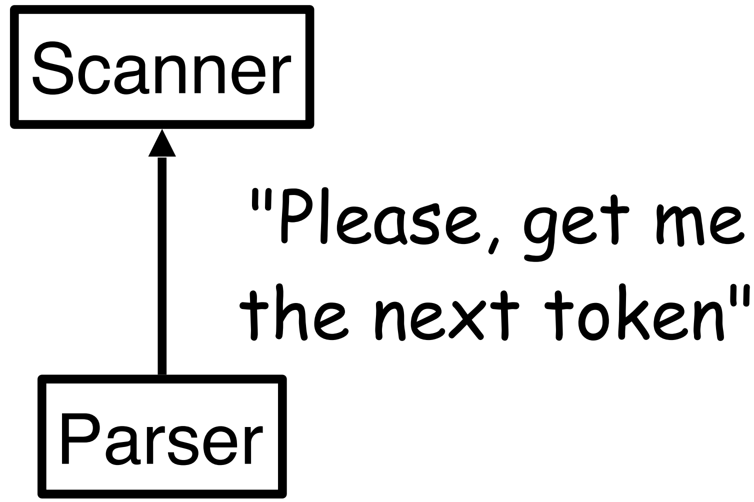
"I recognize this"

Parsing (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

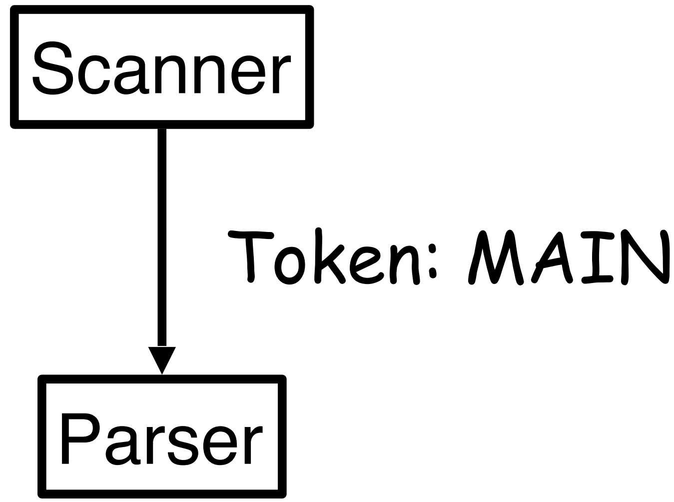
Scanning & Parsing Combined

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



Scanning & Parsing Combined

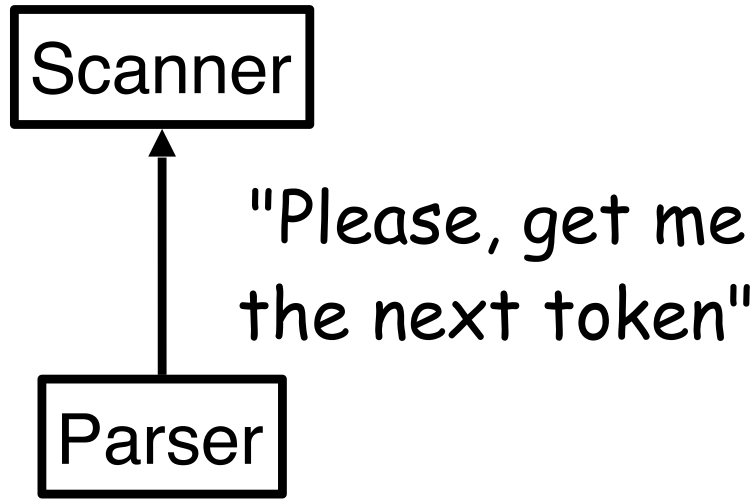
```
main() {
   int a,b;
   a = b;
}
```



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

Scanning & Parsing Combined

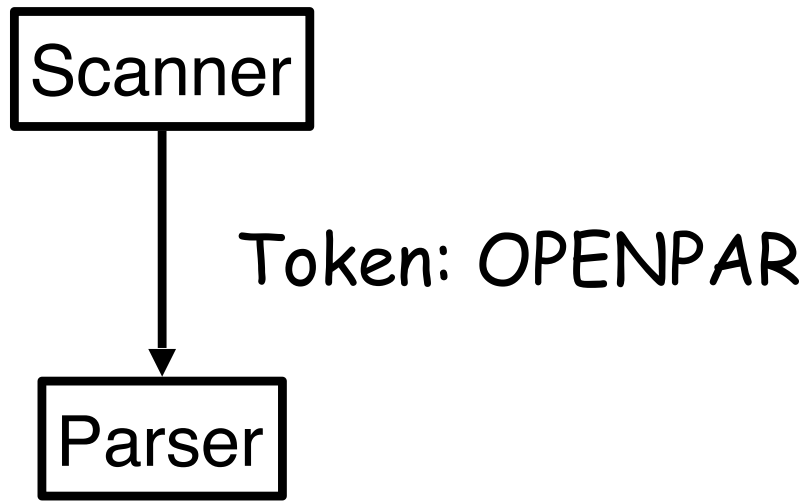
```
main() {
   int a,b;
   a = b;
}
```



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

Scanning & Parsing Combined

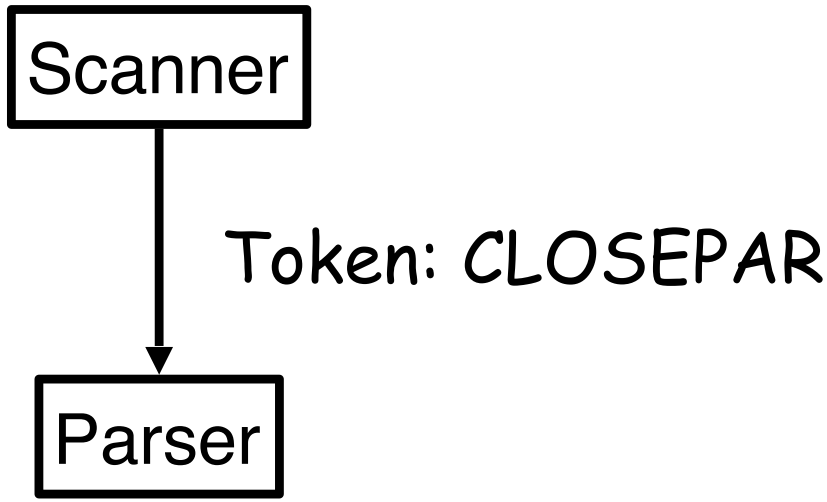
```
main() {
   int a,b;
   a = b;
}
```



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

Scanning & Parsing Combined

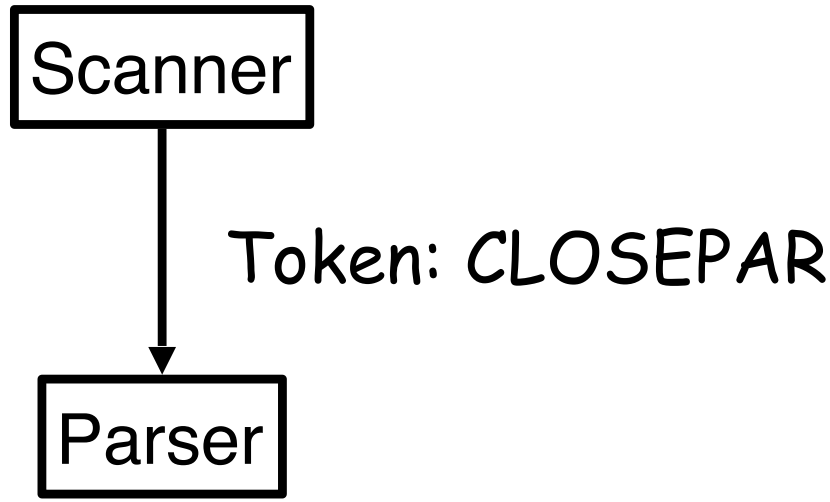
```
main() {
   int a,b;
   a = b;
}
```



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

Scanning & Parsing Combined

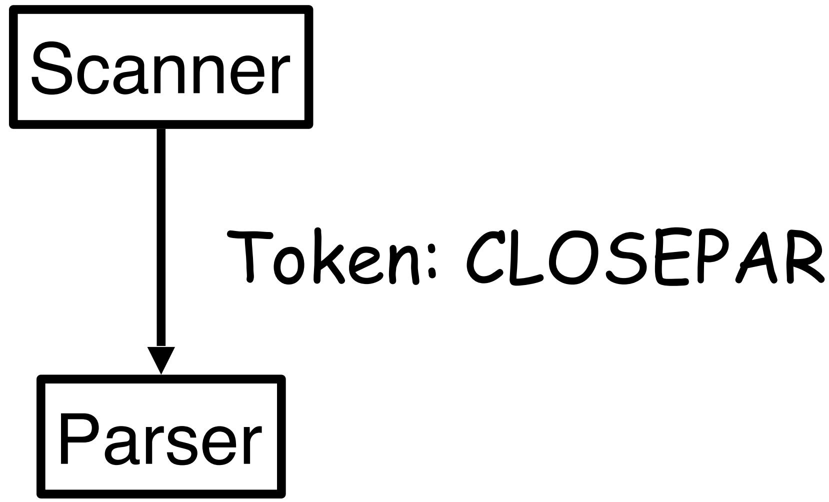
```
main() {
   int a,b;
   a = b;
}
```



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

Scanning & Parsing Combined

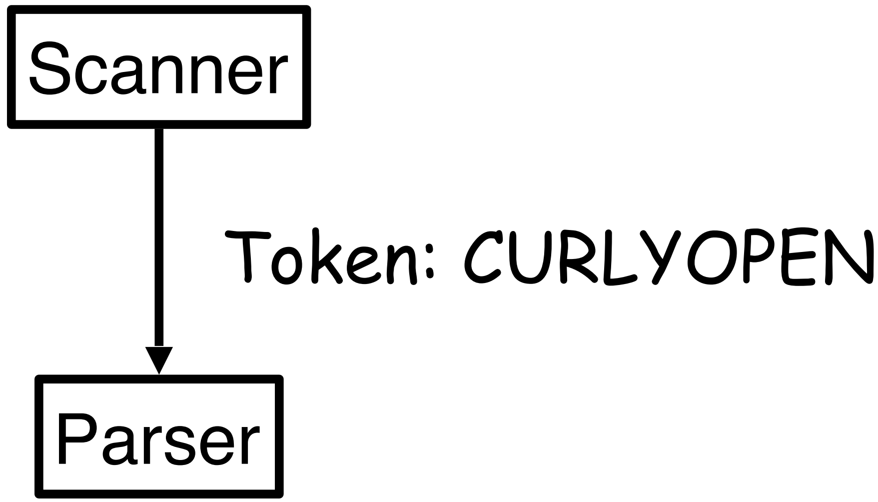
```
main() {
   int a,b;
   a = b;
}
```



<C-PROG> $\rightarrow$ **MAIN OPENPAR <PARAMS> CLOSEPAR <MAIN-BODY>**

Scanning & Parsing Combined

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

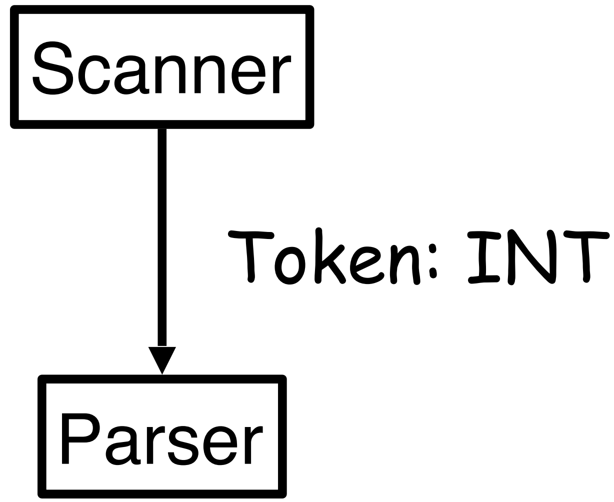


<C-PROG> → MAIN OPENPAR <PARAMS> CLOSEPAR <MAIN-BODY>

<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

Scanning & Parsing Combined

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



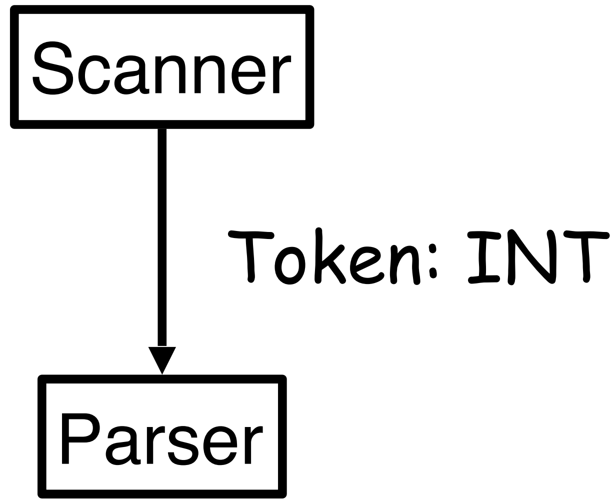
<C-PROG> → MAIN OPENPAR <PARAMS> CLOSEPAR <MAIN-BODY>

<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<DECL-STMT> → <TYPE>VAR<VAR-LIST>;
<TYPE> → INT

Scanning & Parsing Combined

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



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

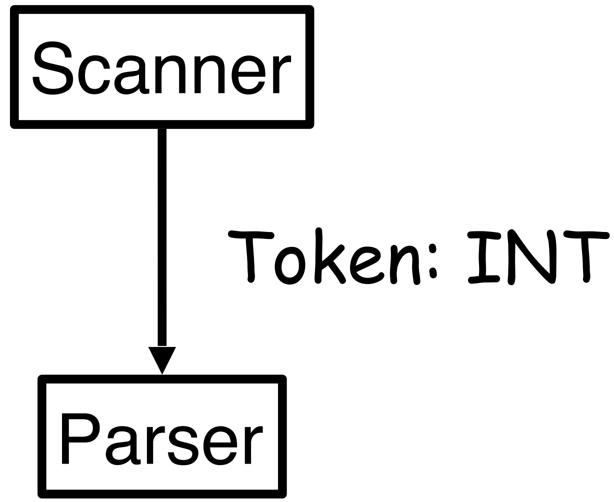
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<DECL-STMT> → <TYPE>VAR<VAR-LIST>;

<TYPE> → INT

Scanning & Parsing Combined

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



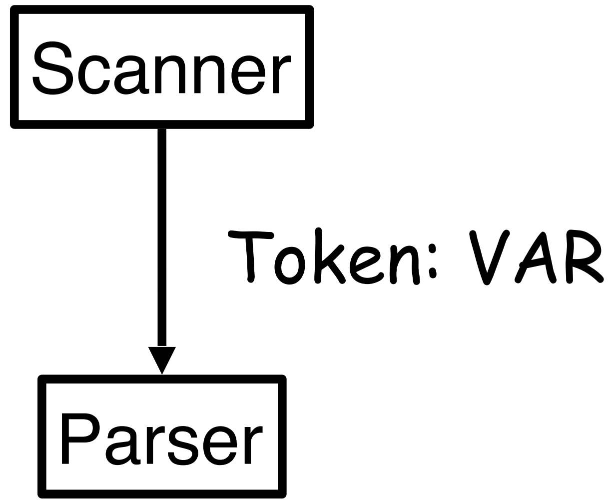
<C-PROG> → MAIN OPENPAR <PARAMS> CLOSEPAR <MAIN-BODY>

<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<DECL-STMT> → <TYPE>VAR<VAR-LIST>;
<TYPE> → INT

Scanning & Parsing Combined

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



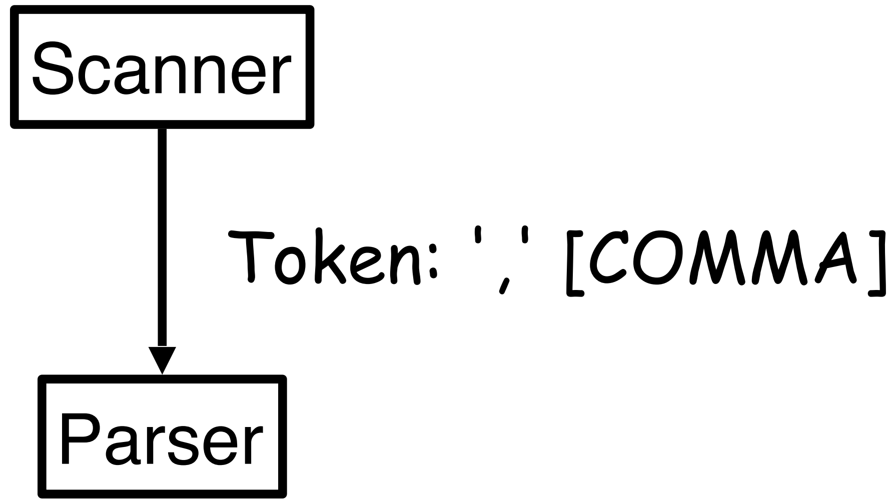
<C-PROG> → MAIN OPENPAR <PARAMS> CLOSEPAR <MAIN-BODY>

<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<DECL-STMT> → <TYPE>VAR<VARLIST>;
<VARLIST> → , VAR <VARLIST>
<VARLIST> → NULL

Scanning & Parsing Combined

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



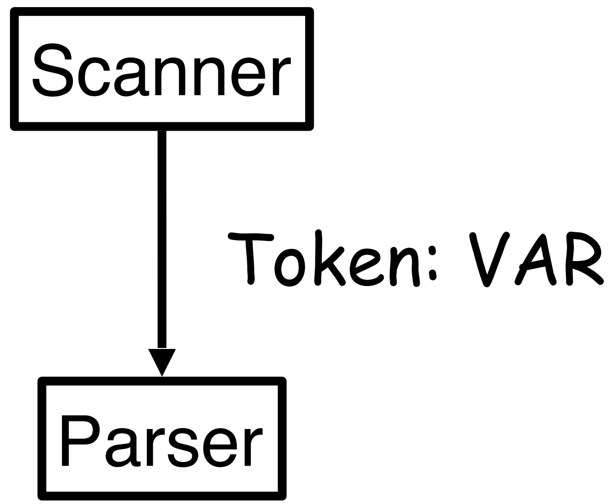
<C-PROG> → MAIN OPENPAR <PARAMS> CLOSEPAR <MAIN-BODY>

<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<DECL-STMT> → <TYPE>VAR<VARLIST>;
<VARLIST> → , VAR <VARLIST>
<VARLIST> → NULL

Scanning & Parsing Combined

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



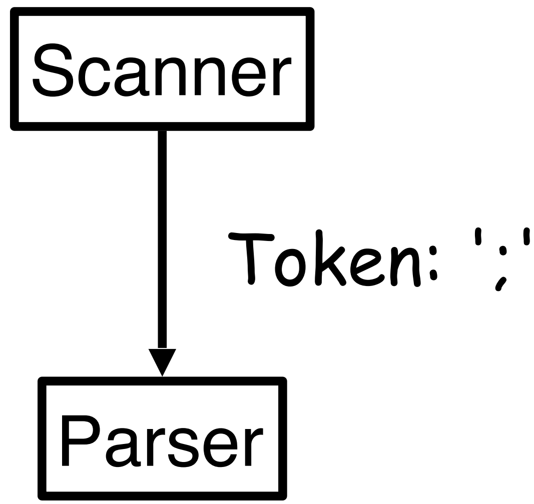
<C-PROG> → MAIN OPENPAR <PARAMS> CLOSEPAR <MAIN-BODY>

<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<DECL-STMT> → <TYPE>VAR<VARLIST>;
<VARLIST> → , VAR <VARLIST>
<VARLIST> → NULL

Scanning & Parsing Combined

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



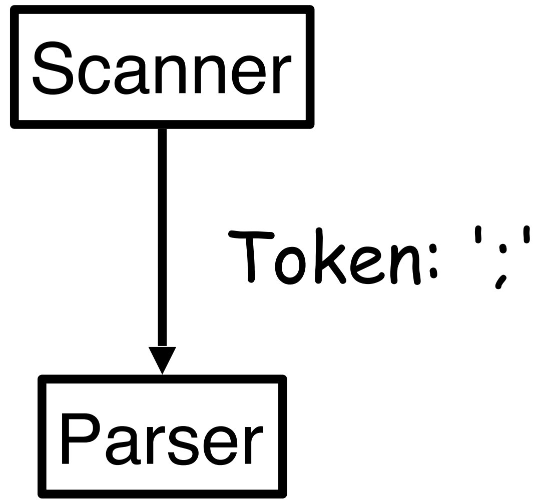
<C-PROG> → MAIN OPENPAR <PARAMS> CLOSEPAR <MAIN-BODY>

<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<DECL-STMT> → <TYPE>VAR<VARLIST>;
<VARLIST> → , VAR <VARLIST>
<VARLIST> → NULL

Scanning & Parsing Combined

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



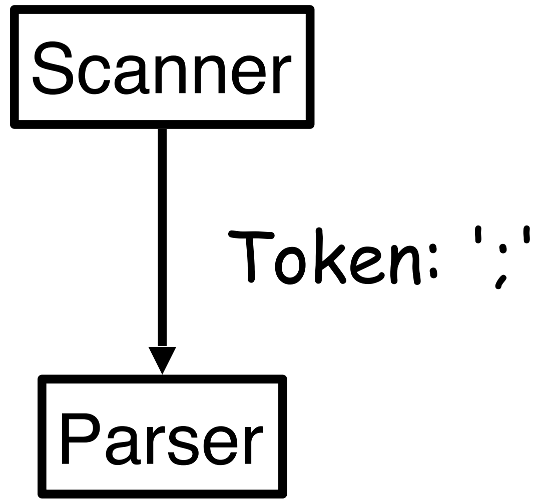
<C-PROG> → MAIN OPENPAR <PARAMS> CLOSEPAR <MAIN-BODY>

<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<DECL-STMT> → <TYPE>VAR<VARLIST>;
<VARLIST> → , VAR <VARLIST>
<VARLIST> → NULL

Scanning & Parsing Combined

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



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

<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

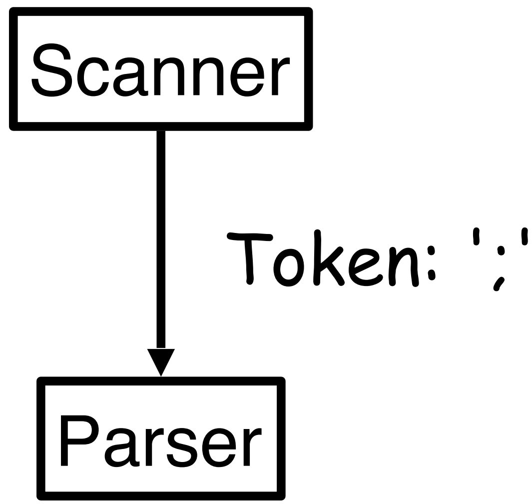
<DECL-STMT> → <TYPE>VAR<VARLIST>;

<VARLIST> → , VAR <VARLIST>

<VARLIST> → NULL

Scanning & Parsing Combined

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



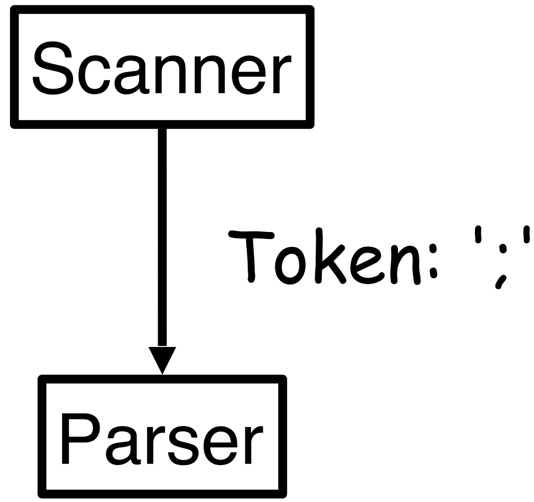
<C-PROG> → MAIN OPENPAR <PARAMS> CLOSEPAR <MAIN-BODY>

<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<DECL-STMT> → <TYPE>VAR<VARLIST>;
<VARLIST> → , VAR <VARLIST>
<VARLIST> → NULL

Scanning & Parsing Combined

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



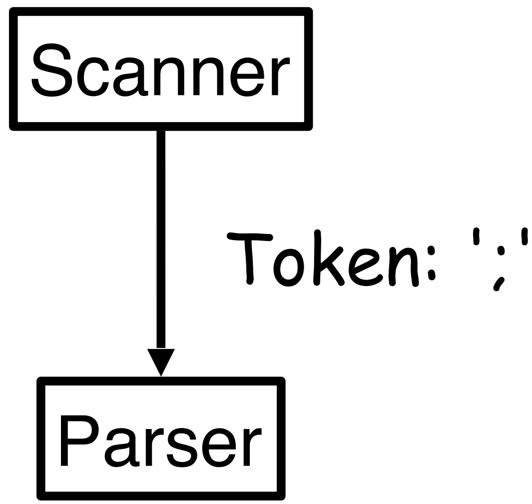
<C-PROG> → MAIN OPENPAR <PARAMS> CLOSEPAR <MAIN-BODY>

<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<DECL-STMT> → <TYPE>VAR<VARLIST>;

Scanning & Parsing Combined

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



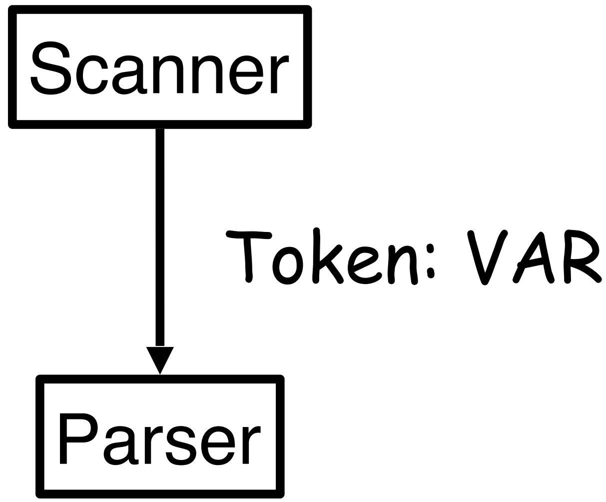
<C-PROG> → MAIN OPENPAR <PARAMS> CLOSEPAR **<MAIN-BODY>**

<MAIN-BODY> → CURLYOPEN <DECL-STMT> **<ASSIGN-STMT> CURLYCLOSE**

<DECL-STMT> → <TYPE>VAR<VARLIST>;

Scanning & Parsing Combined

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

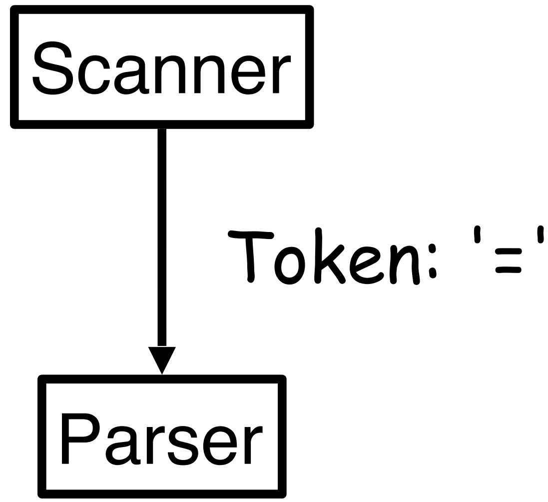


<C-PROG> → MAIN OPENPAR <PARAMS> CLOSEPAR <MAIN-BODY>

<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<ASSIGN-STMT> → VAR = <EXPR>;
<EXPR> → VAR

Scanning & Parsing Combined

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

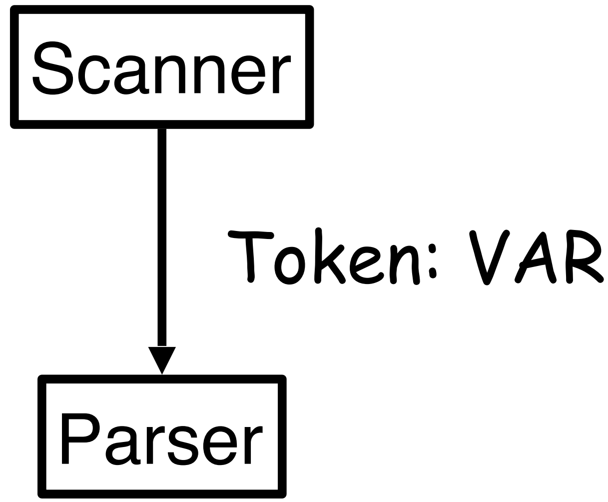


<C-PROG> → MAIN OPENPAR <PARAMS> CLOSEPAR <MAIN-BODY>

<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<ASSIGN-STMT> → VAR = <EXPR>;
<EXPR> → VAR

Scanning & Parsing Combined

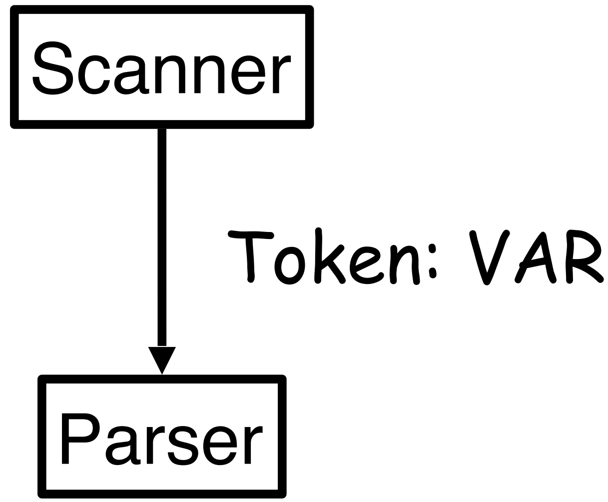
```
main() {
   int a,b;
   a = b;
}
```




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

Scanning & Parsing Combined

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

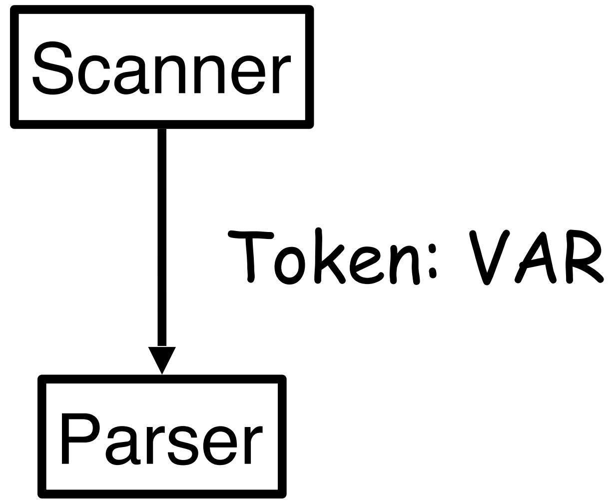



```
<C-PROG> → MAIN OPENPAR <PARAMS> CLOSEPAR <MAIN-BODY>

<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<ASSIGN-STMT> → VAR = <EXPR>;
<EXPR> → VAR
```

Scanning & Parsing Combined

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



<C-PROG> → MAIN OPENPAR <PARAMS> CLOSEPAR **<MAIN-BODY>**

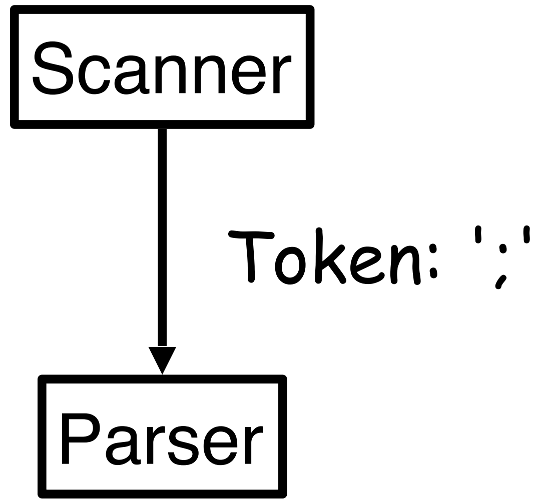
<MAIN-BODY> → CURLYOPEN <DECL-STMT> **<ASSIGN-STMT> CURLYCLOSE**

<ASSIGN-STMT> → VAR = <EXPR>;

<EXPR> → VAR

Scanning & Parsing Combined

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



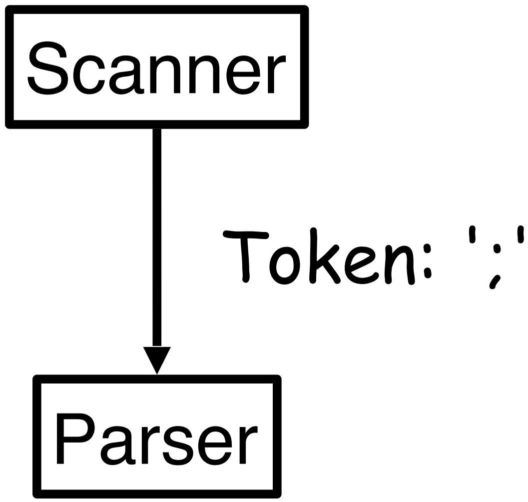
<C-PROG> → MAIN OPENPAR <PARAMS> CLOSEPAR **<MAIN-BODY>**

<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> **CURLYCLOSE**

<ASSIGN-STMT> → VAR = <EXPR>;

Scanning & Parsing Combined

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



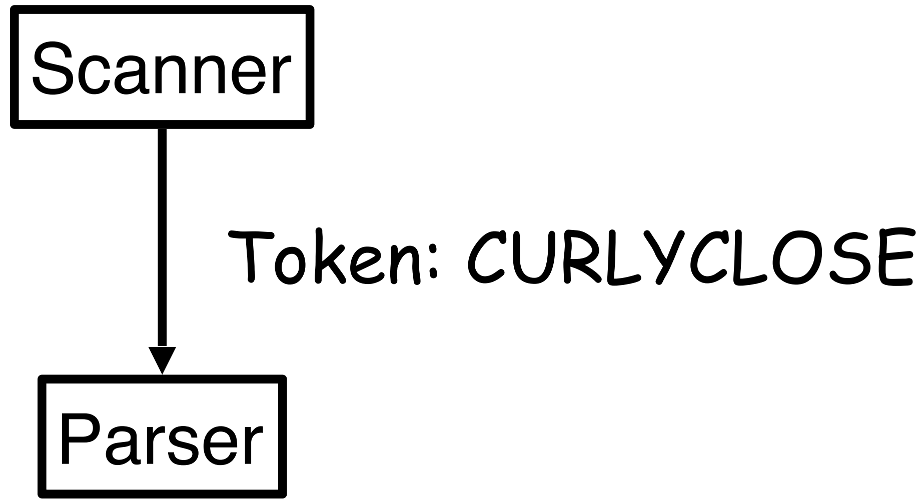
<C-PROG> → MAIN OPENPAR <PARAMS> CLOSEPAR **<MAIN-BODY>**

<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> **CURLYCLOSE**

<ASSIGN-STMT> → VAR = <EXPR>;

Scanning & Parsing Combined

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



<C-PROG> → MAIN OPENPAR <PARAMS> CLOSEPAR **<MAIN-BODY>**

<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

What Is Happening?

- During/after parsing?
  - Tokens get gobbled
- 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"

Typical 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
```

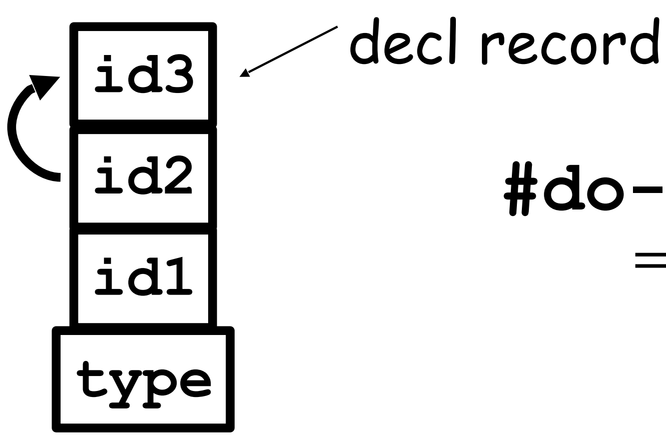
Semantic Context

How Are Semantic Actions Called?

- 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
```



#do-decl
⇒

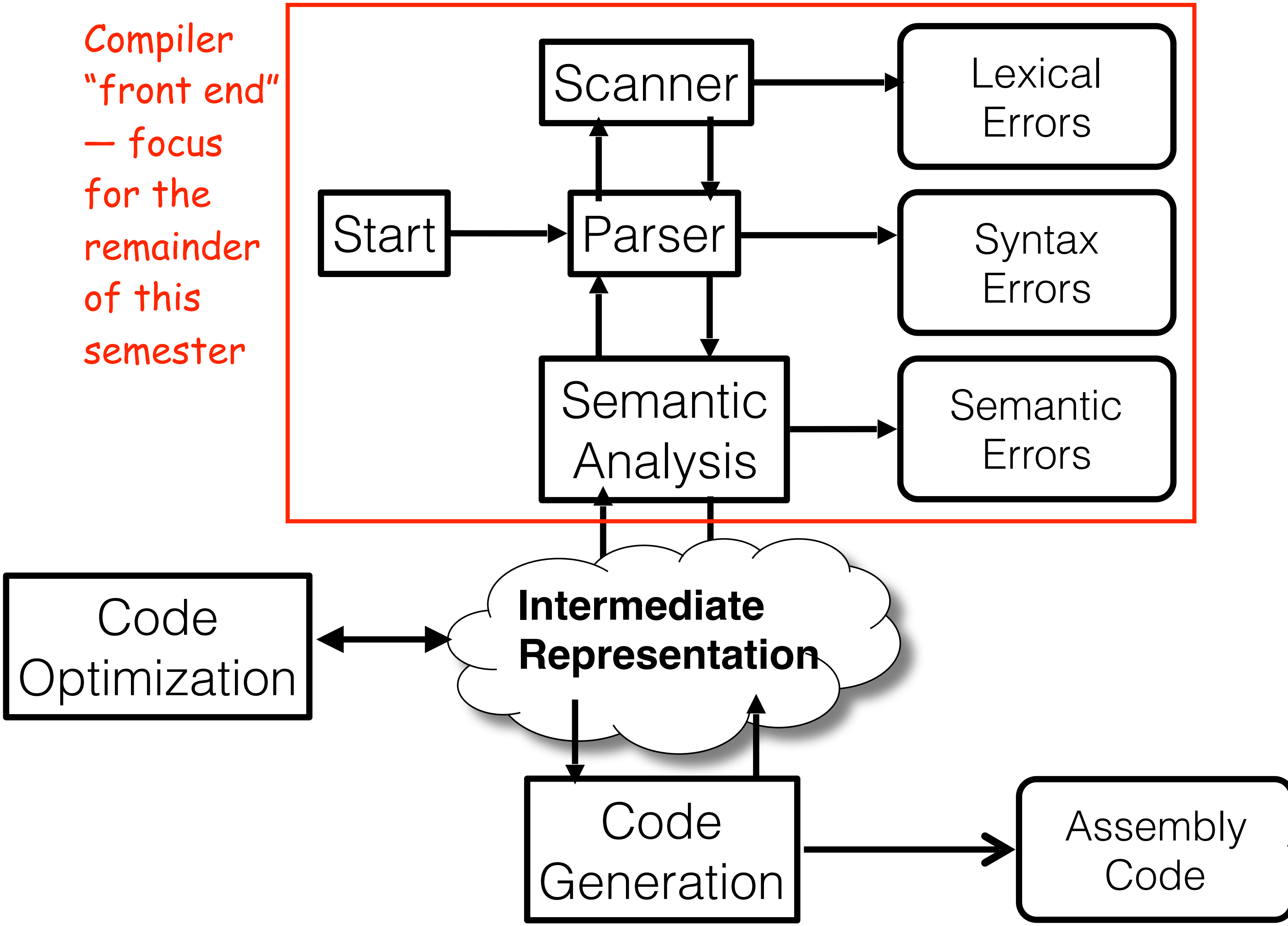
Name	Type	Scope
id1	1	3
id2	1	3
id3	1	3

Semantic Actions

- What else can semantic actions do in addition to storing and looking up names in a symbol table?
- Two type of semantic actions
  - Checking (binding, type compatibility, scoping, etc.)
  - Translation (generate temporary values, propagate them to keep semantic context).

Structure of a Full Compiler (Recap from Lecture 1)

Compiler "front end" — focus for the remainder of this semester



Summary

- Front-end of compiler: scanner and parser
- Scanner and parser generation are automated using regular expressions and context-free grammars
  - We will use the ANTLR framework for this automation in Project 3