Top-Down Parsing

Top down parsing can be viewed as an attempt to find a leftmost derivation for an input string. Equivalently, it can be viewed as an attempt to construct a parse tree for the input starting from the root and creating nodes of the parse tree in preorder.

Top-Down Parsing(cont'd)

We now consider a general form of top down parsing, called recursive descent, that may involve backtracking, that is making repeated scans of the input. However, backtracking parsers are not seen frequently

Top-Down Parsing(cont'd)

One reason is that backtracking is rarely needed to parse programming language constructs. In situations like natural language parsing, backtracking is still not very efficient, and tabular methods such as the dynamic programming algorithm or method of Earley are preferred.

Example.

Consider the grammar,

$S \longrightarrow cAd$

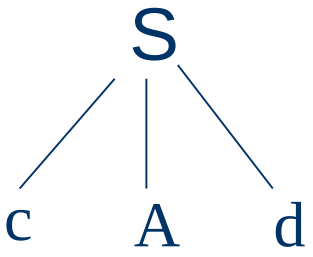
$A \longrightarrow ab|a$

and the input string **w** = cad. To construct a parse tree for this string top down, we initially create a tree consisting of a single node labeled **S.**

Example (cont'd)

An input pointer points to "**c**", the first symbol of **w**. We then use the first production for **S** to expand the tree and obtain the tree of the Fig.(a)

Fig(a)

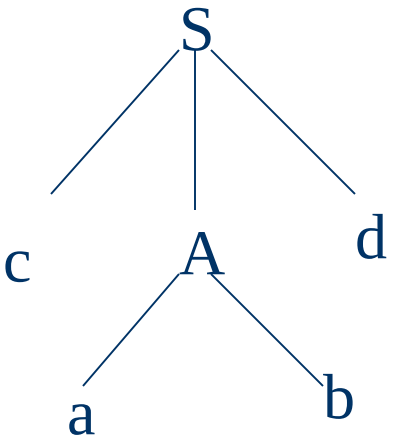


Example (cont'd)

The leftmost leaf, labeled "c", matches the first symbol of **w**,So we now advance the input pointer to "a",the second symbol of **w,** and consider the next leaf, labeled **A.** We can then expand **A** using the first alternative for **A** to obtain the tree of the fig(b).

Example (cont'd)

Fig(b)

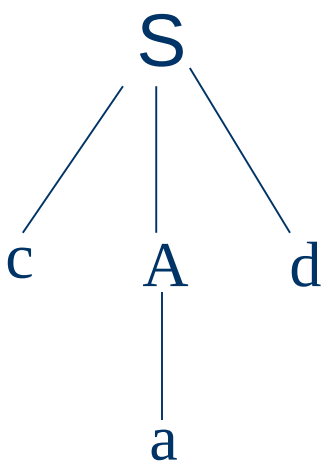


Example (cont'd)

We now have a match for the second input symbol so we advance the input pointer to "d", the third input symbol and compare "d" against the next leaf, labeled "b". Since "b" does not match "d", we report failure and go back to **A** to see whether there is another alternative for **A** that we have not tried but that might produce a match.

Example (cont'd)

Fig(c)



Example (cont'd)

In going back to **A**, we must reset the input pointer to position 2, the position it had when we first came to **A**, which means that the procedure for **A** must store the input pointer in a local variable. We now try the second alternative for **A** to obtain the tree of the fig(c). The leaf "a" matches the second symbol **w** and the leaf "d" matches the third symbol. Since we have produced a parse tree for **w**, we halt and announce successful completion of parsing.

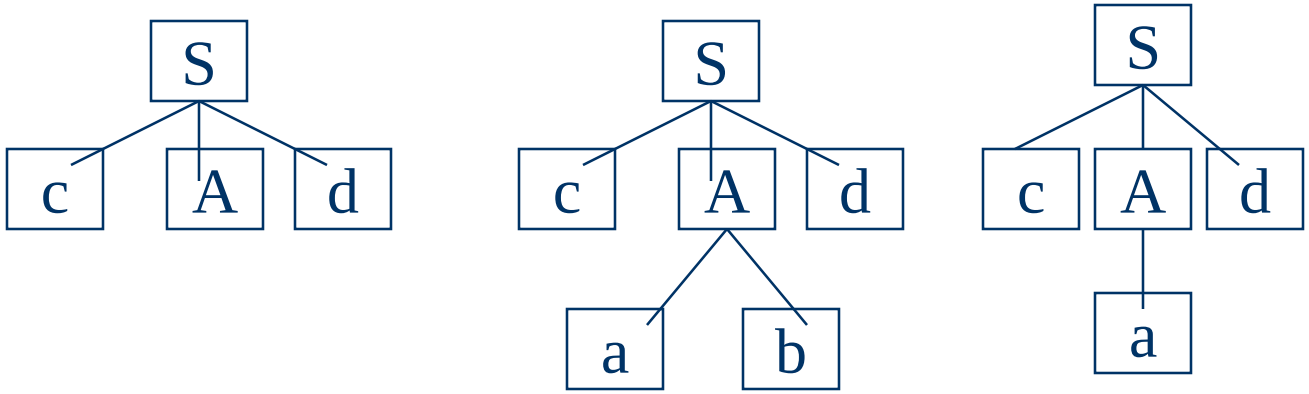
Top-down Parsing

- Construct a parse tree from the root
- Example

  S → cAd

  A → ab | a

  Input w = cad

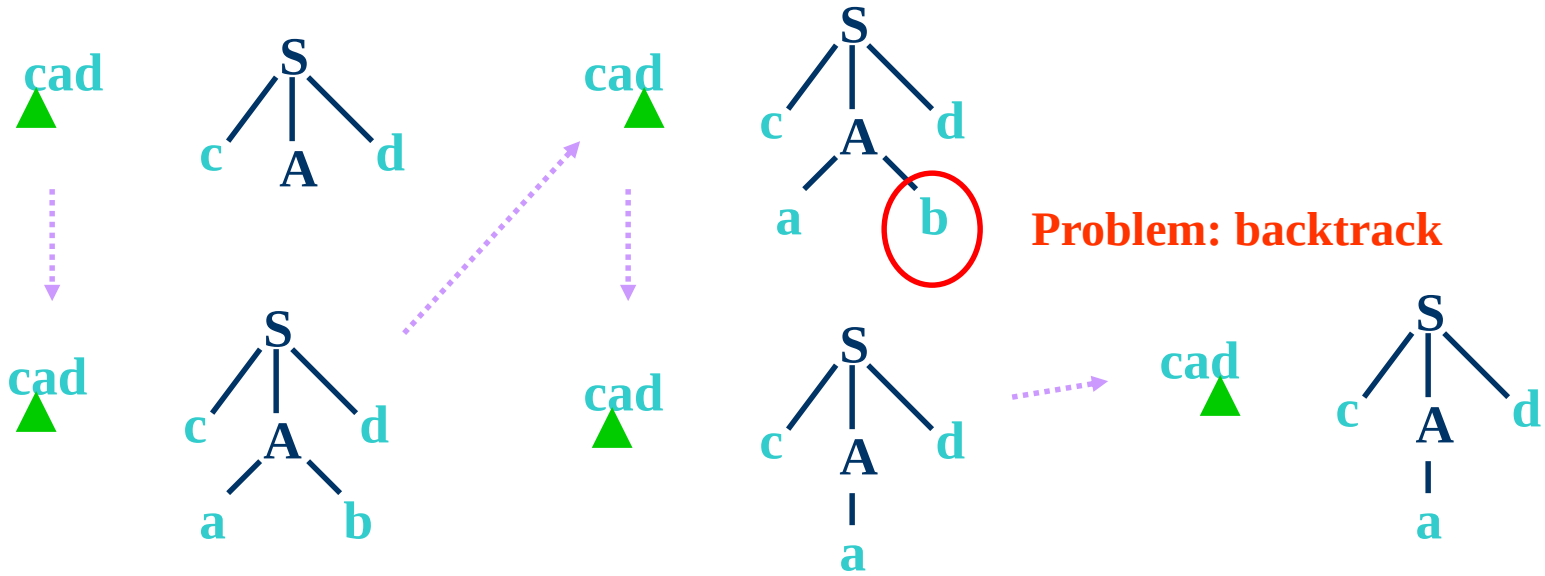


Example

Example: $S \rightarrow c\ A\ d$

$A \rightarrow ab \mid a$

input: cad



Parsing – Top-Down & Predictive

- Top-Down Parsing $\Rightarrow$ Parse tree / derivation of a token string occurs in a top down fashion.

- For Example, Consider:

Start symbol

$$
\begin{aligned}
\textit{type} &\rightarrow \textit{simple} \\
&\mid \uparrow \textbf{id} \\
&\mid \textbf{array} \; [\; \textit{simple} \;] \; \textbf{of} \; \textit{type} \\
\textit{simple} &\rightarrow \textbf{integer} \\
&\mid \textbf{char} \\
&\mid \textbf{num dotdot num}
\end{aligned}
$$

Suppose input is :

array [**num dotdot num**] **of integer**

Parsing would begin with

type $\rightarrow$ **???**

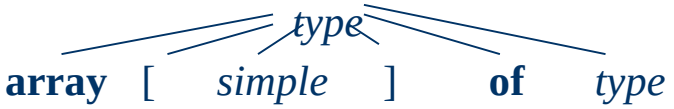
Top-Down Parse

Lookahead symbol

Input : **array** [**num dotdot num**] **of integer**

type

?



type → *simple* **Start symbol**

| ↑ **id**

| **array** [*simple*] **of** *type*

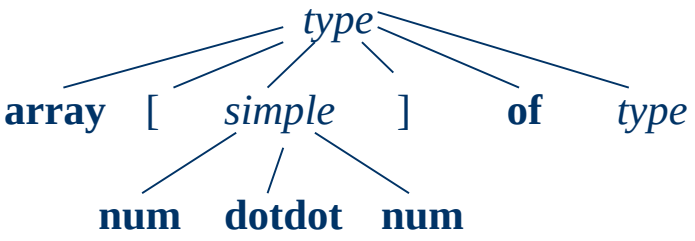
simple → **integer**

| **char**

| **num dotdot num**

Lookahead symbol

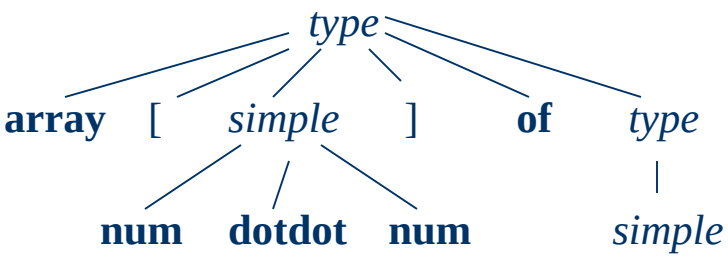
Input : **array** [**num dotdot num**] **of integer**

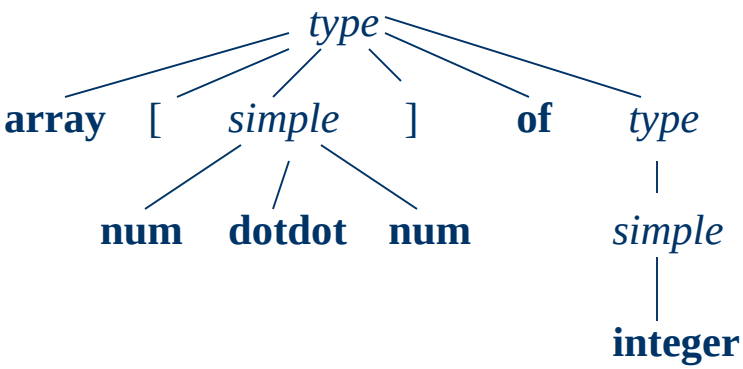


Top-Down Parse

Lookahead symbol

Input : **array** [**num dotdot num**] **of** **integer**





type → *simple* **Start symbol**
| ↑ **id**
| **array** [*simple*] **of** *type*
simple → **integer**
| **char**
| **num dotdot num**

Recursive Descent or Predictive Parsing

- Parser Operates by Attempting to Match Tokens in the Input Stream
- Utilize both Grammar and Input Below to Motivate Code for Algorithm

array [num dotdot num] of integer

type → *simple*
| ↑ **id**
| **array** [*simple*] **of** *type*
simple → **integer**
| **char**
| **num dotdot num**

```
procedure match ( t : token ) ;
begin
    if lookahead = t then
        lookahead := nexttoken
    else error
end ;
```

Top-down algorithm (continued)

```
procedure  simple ;
begin
        if  lookahead = integer  then   match ( integer );
        else if  lookahead = char  then  match ( char );
              else if  lookahead = num  then  begin
                        match (num); match (dotdot); match (num)
                    end
                    else  error
end ;
```

$type \rightarrow simple$

$\quad | \uparrow \mathbf{id}$

$\quad | \mathbf{array}\ [\ simple\ ]\ \mathbf{of}\ type$

$simple \rightarrow \mathbf{integer}$

$\quad | \mathbf{char}$

$\quad | \mathbf{num\ dotdot\ num}$

Top-Down Algorithm (Continued)

```
procedure  type ;
begin
        if  lookahead  is in { integer, char, num }  then  simple
        else if  lookahead  = '↑'  then  begin  match ('↑' ) ;  match( id ) end
           else if  lookahead = array  then  begin
                 match( array ); match('['); simple; match(']'); match(of); type
                end
                else  error
end ;
```

$type \rightarrow simple$

| ↑ **id**

| **array** [*simple*] **of** *type*

$simple \rightarrow$ **integer**

| **char**

| **num dotdot num**

Tracing

type → *simple*
| ↑ **id**
| **array** [*simple*] **of** *type*
simple → **integer**
| **char**
| **num dotdot num**

Input: **array** [**num dotdot num**] **of** **integer**

To initialize the parser:
set global variable : **lookahead = array**
call procedure: **type**

Procedure call to **type** *with* **lookahead = array** *results in the actions:*

match(**array**); *match*('['); simple; *match*(']'); *match*(**of**); *type*

Procedure call to **simple** *with* **lookahead = num** *results in the actions:*

match (**num**); *match* (**dotdot**); *match* (**num**)

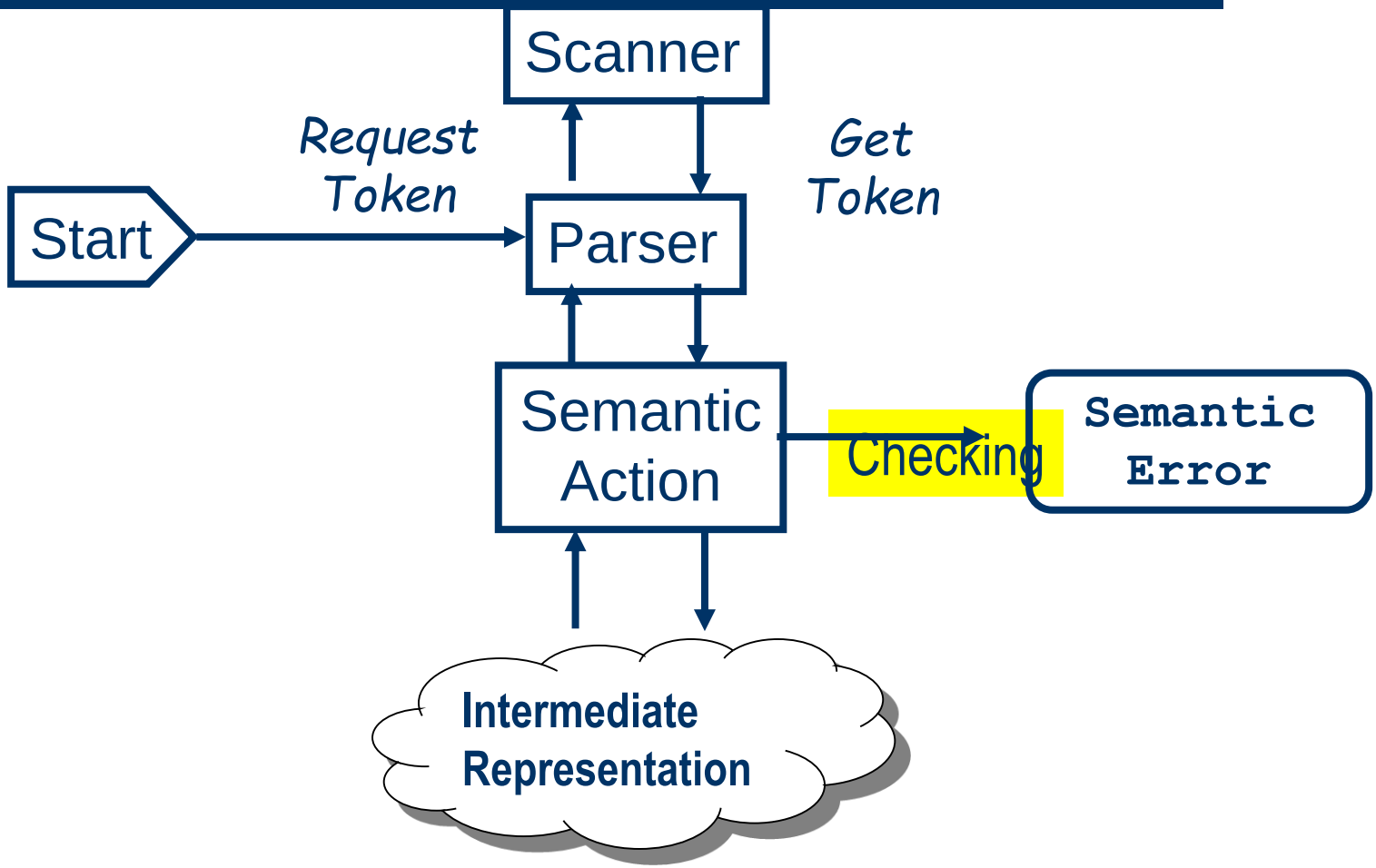
Procedure call to **type** *with* **lookahead = integer** *results in the actions:*

simple

Procedure call to **simple** *with* **lookahead = integer** *results in the actions:*

match (**integer**)

Compiler Phases – Front End



Big Picture

- Parsing: Matching code we are translating to rules of a grammar. Building a representation of the code.
- Scanning: An abstraction that simplifies the parsing process by converting the raw text input into a stream of known objects called tokens.

- Grammar dictates syntactic rules of a language ie, how a legal sentence in a language could be formed
- Lexical rules of a language dictate how a legal word in a language is formed by concatenating alphabet of the language.

Overall Operation

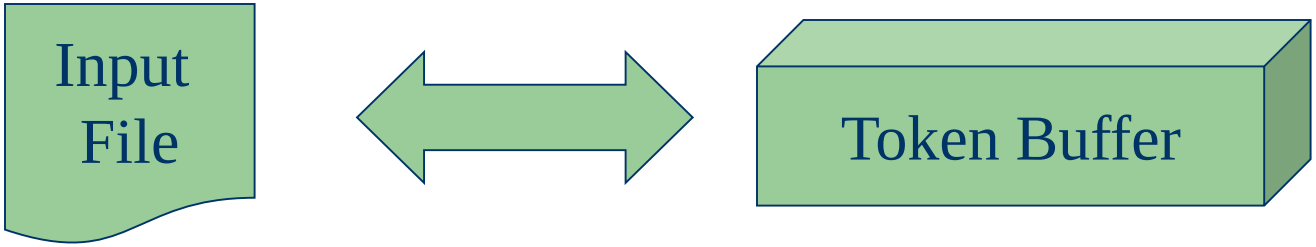
- Parser is in control of the overall operation
- Demands scanner to produce a token
- Scanner reads input file into token buffer & forms a token
- Token is returned to parser
- Parser attempts to match the token
- Failure: Syntax Error!
- Success:
  - Does nothing and returns to get next token

    or
  - Takes Semantic Action

Overall Operation

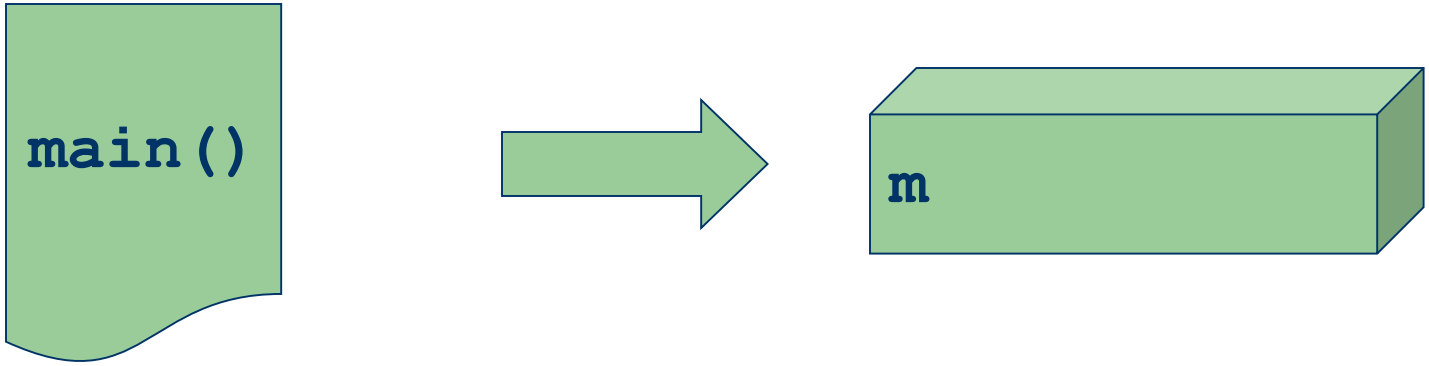
- Semantic Action: Lookup variable name
  - If found okay
  - If not: Put in symbol table
- If semantic checks succeed, do code-generation
- Return to get next token
- No more tokens? Done!

Tokenization

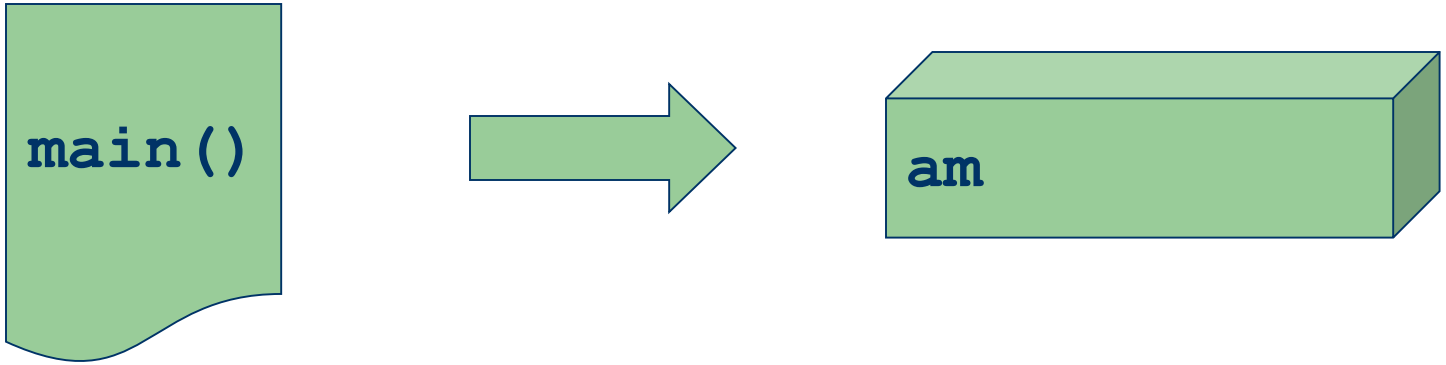


- What does the Token Buffer contain?
  - Token being identified
- Why a two-way (⟷) street?
  - Characters can be read →
  - and unread ←
  - Termination of a token

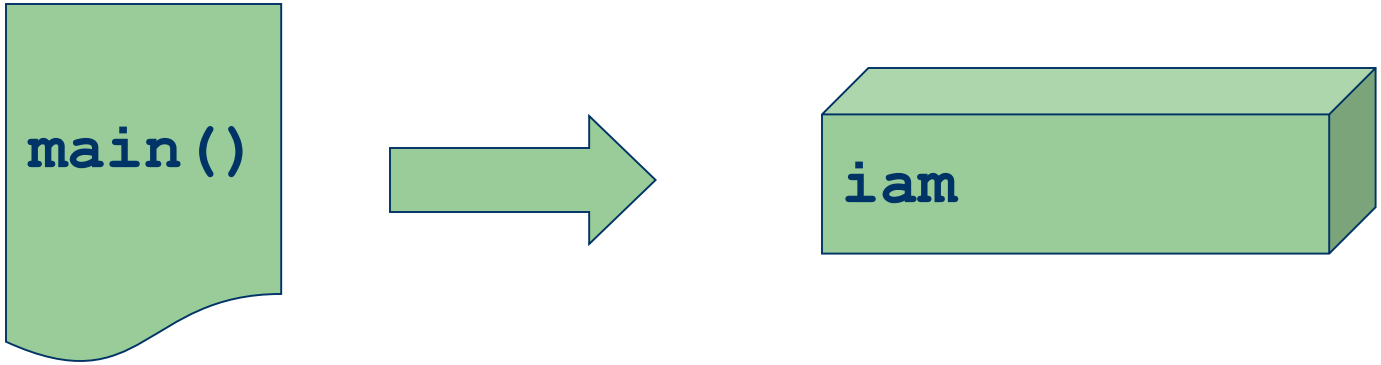
Example



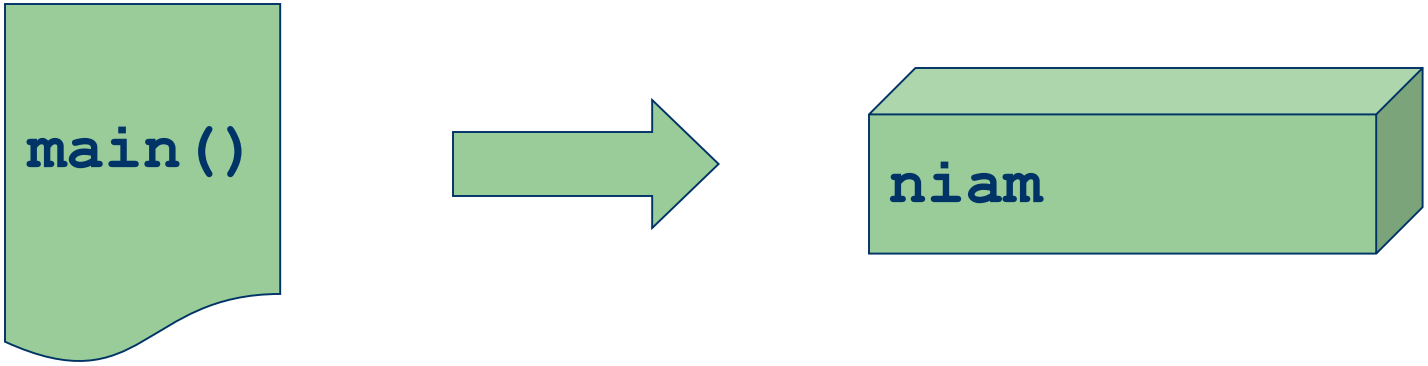
Example



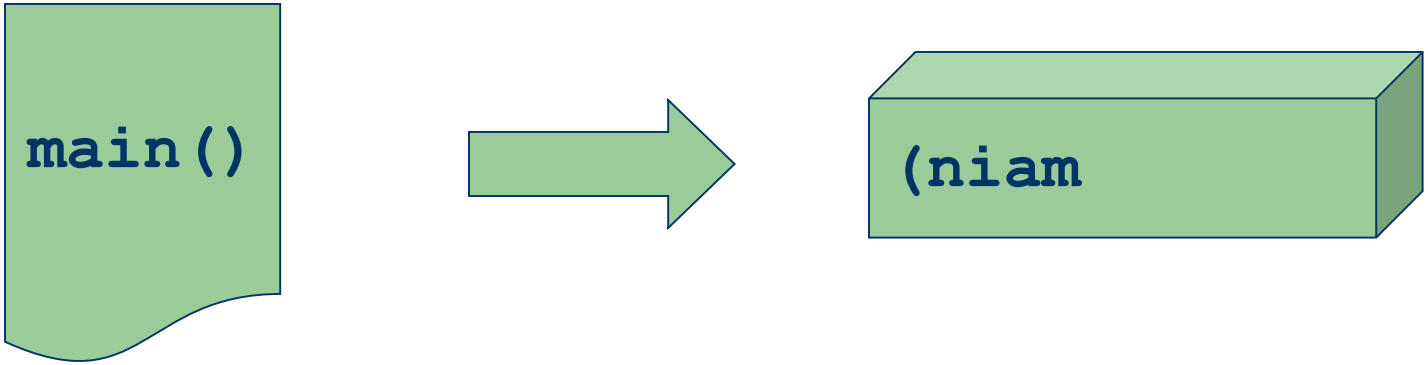
Example



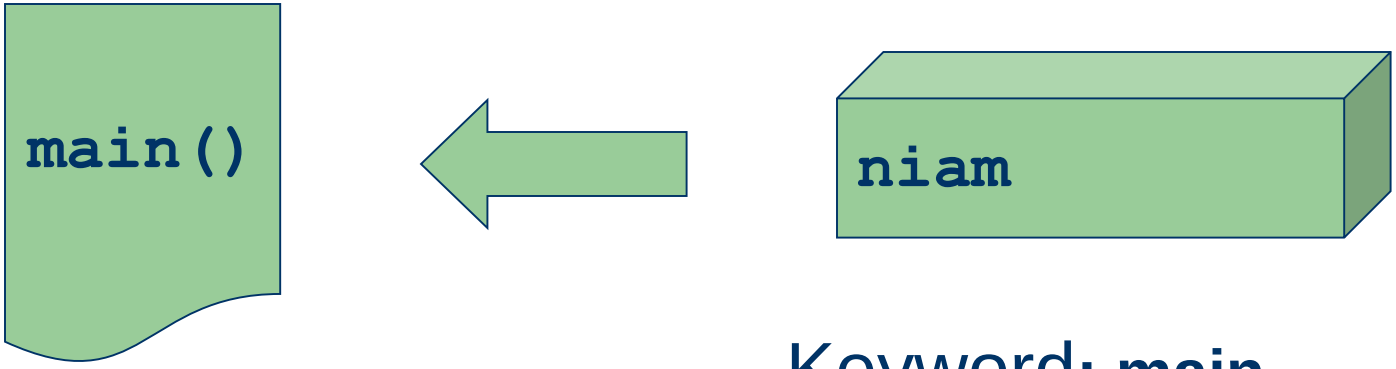
Example



Example



Example



Keyword: **main**

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
- Token is returned to parser
- Parser attempts to match the token
- Failure: Syntax Error!
- Success:
  - Does nothing and returns to get next token

    OR
  - Takes Semantic Action

Overall Operation

- Semantic Action: Lookup variable name
  - If found okay
  - If not: Put in symbol table
- If semantic checks succeed, do code-generation
- Return to get next token
- No more tokens? Done!

Grammar Rules

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

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

Scanner

```
m
```

Parser

Demo

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

Scanner

```
am
```

Parser

Demo

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

Scanner

```
iam
```

Parser

Demo

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

Scanner

```
niam
```

Parser

Demo

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

Scanner

```
(niam
```

Parser

Demo

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

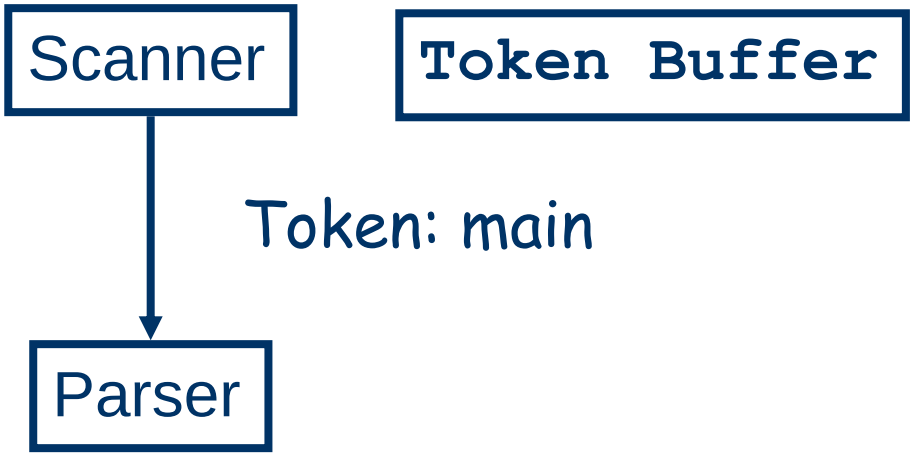
Scanner

```
niam
```

Parser

Demo

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



Demo

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

Scanner

Token Buffer

Parser

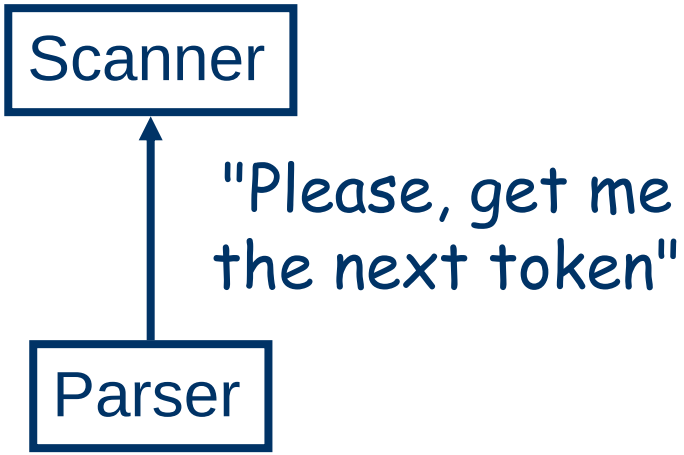
"I recognize this"

Parsing (Matching)

- Start matching using a rule
- When match takes place at a certain position, move further (get next token & repeat the process)
- 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
- Grammar ambiguous? Language ambiguous?

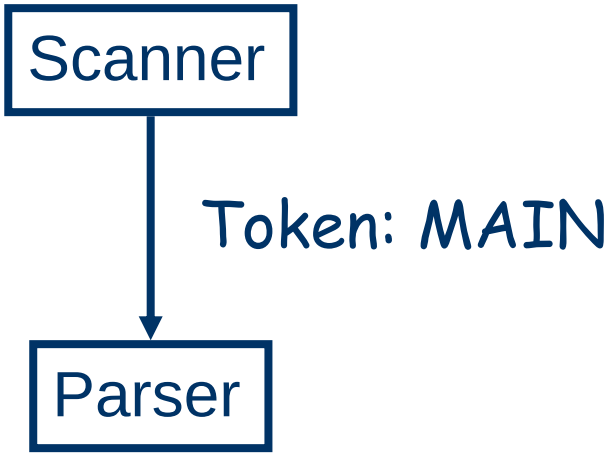
Scanning & Parsing Combined

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



Scanning & Parsing Combined

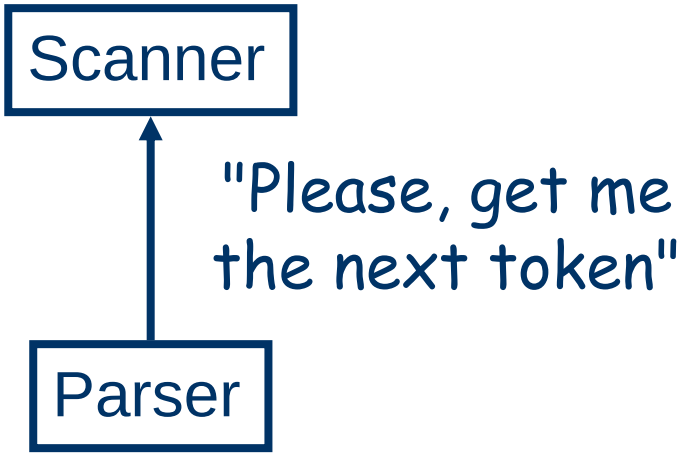
```
main() {
   int a,b;
   a = b;
}
```



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

Scanning & Parsing Combined

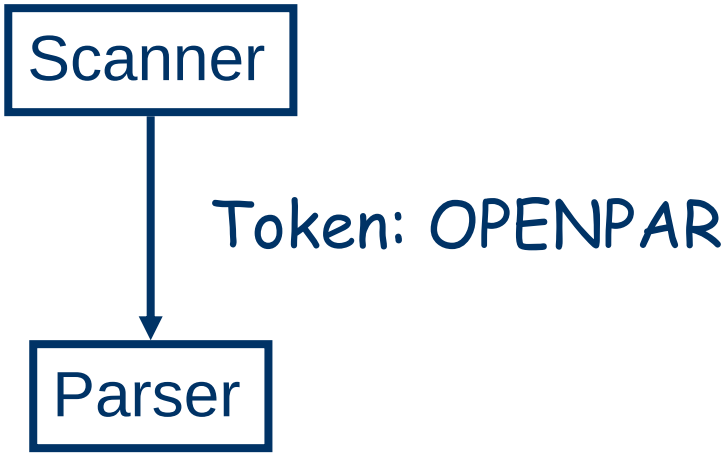
```
main() {
   int a,b;
   a = b;
}
```



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

Scanning & Parsing Combined

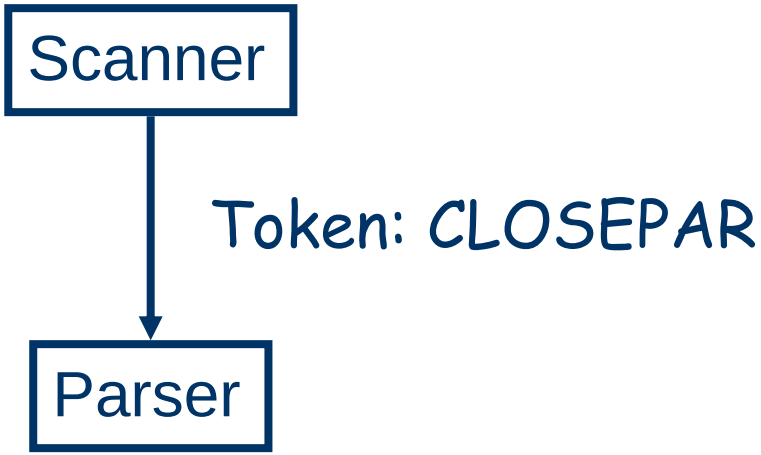
```
main() {
   int a,b;
   a = b;
}
```



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

Scanning & Parsing Combined

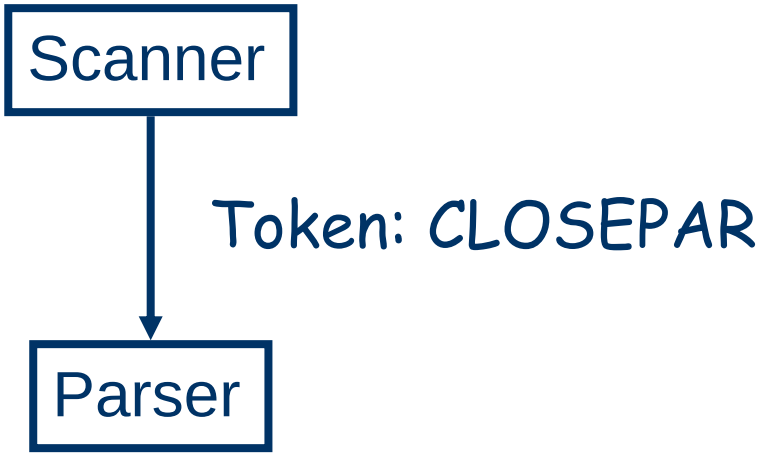
```
main() {
   int a,b;
   a = b;
}
```



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

Scanning & Parsing Combined

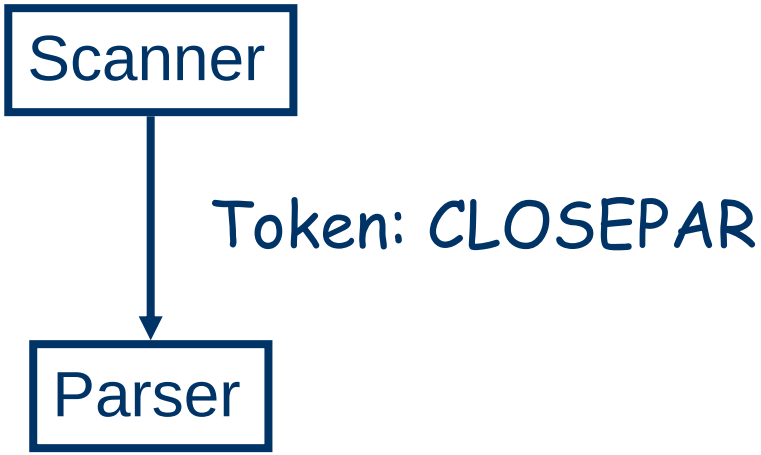
```
main() {
   int a,b;
   a = b;
}
```



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

Scanning & Parsing Combined

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



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

Scanning & Parsing Combined

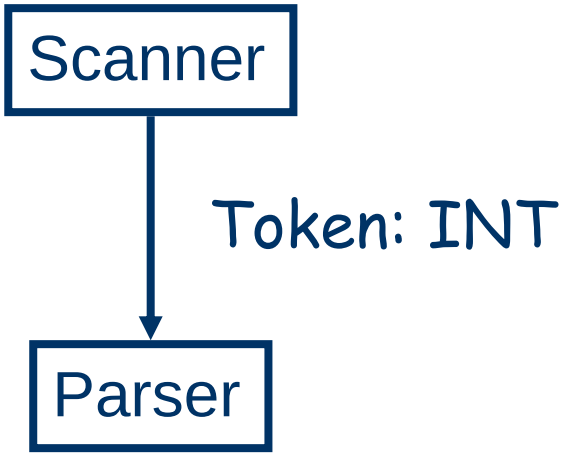
```
main() {
   int a,b;
   a = b;
}
```

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

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

Scanning & Parsing Combined

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



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

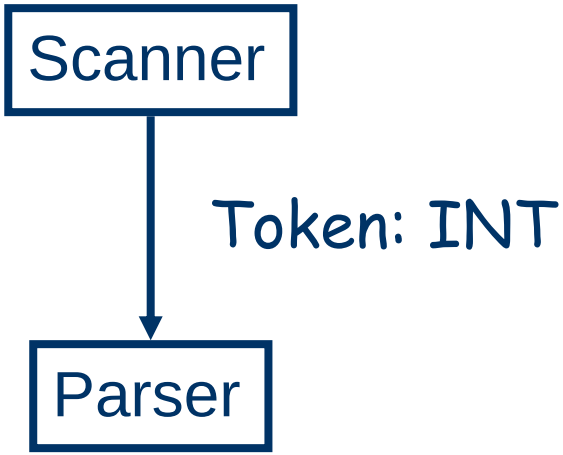
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<DECL-STMT> → <TYPE>VAR<VAR-LIST>;

<TYPE> → INT

Scanning & Parsing Combined

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



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

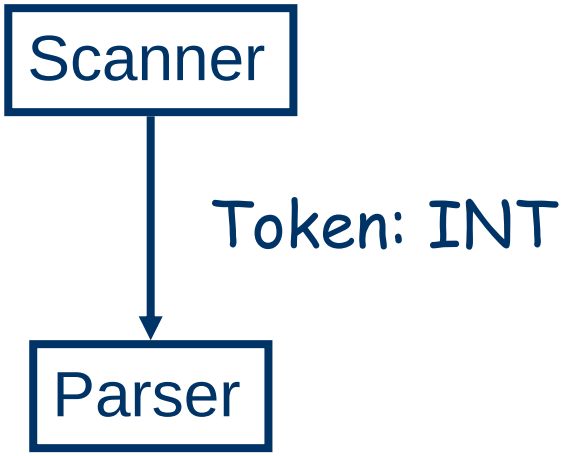
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<DECL-STMT> → <TYPE>VAR<VAR-LIST>;

<TYPE> → INT

Scanning & Parsing Combined

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



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

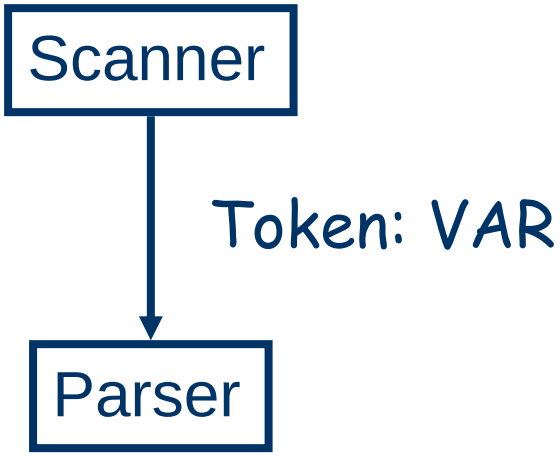
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<DECL-STMT> → <TYPE>VAR<VAR-LIST>;

<TYPE> → INT

Scanning & Parsing Combined

```
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
```

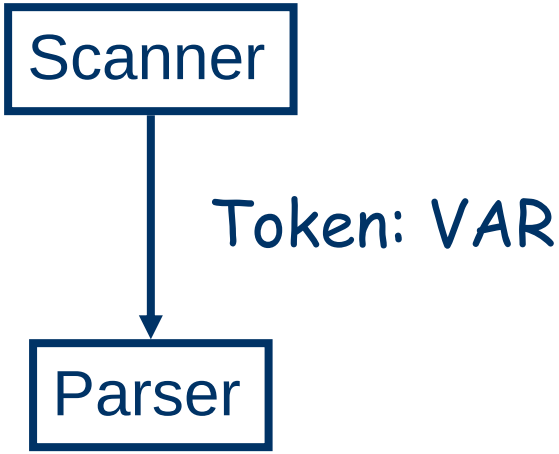
Scanning & Parsing Combined

```
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 & Parsing Combined

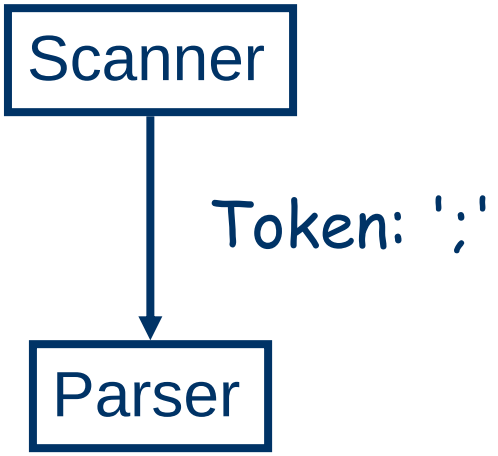
```
main() {
   int a,b;
   a = b;
}
```




```
<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 & Parsing Combined

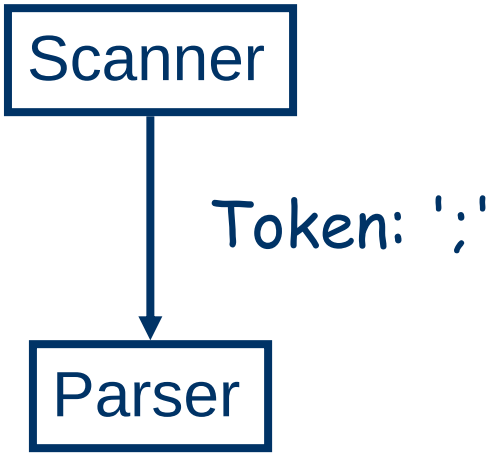
```
main() {
   int a,b;
   a = b;
}
```




```
<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 & Parsing Combined

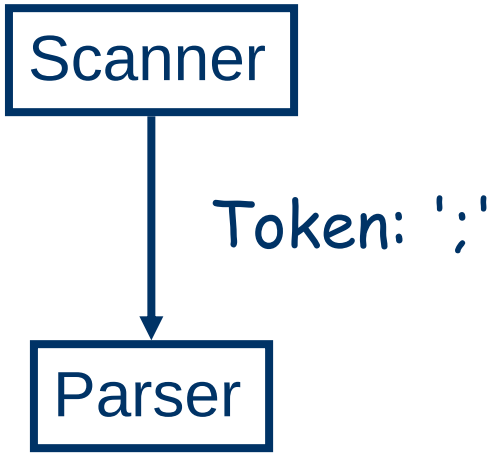
```
main() {
   int a,b;
   a = b;
}
```




```
<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 & Parsing Combined

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



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

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

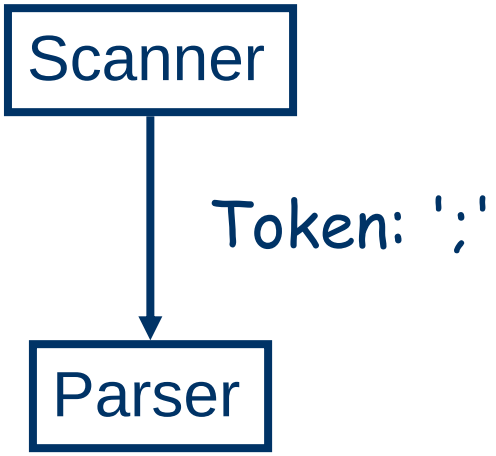
<DECL-STMT> → <TYPE>VAR<VAR-LIST>;

<VARLIST> → , VAR <VARLIST>

<VARLIST> → NULL

Scanning & Parsing Combined

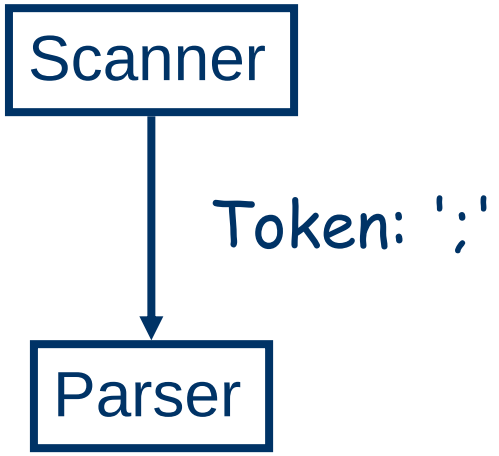
```
main() {
   int a,b;
   a = b;
}
```




```
<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 & Parsing Combined

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



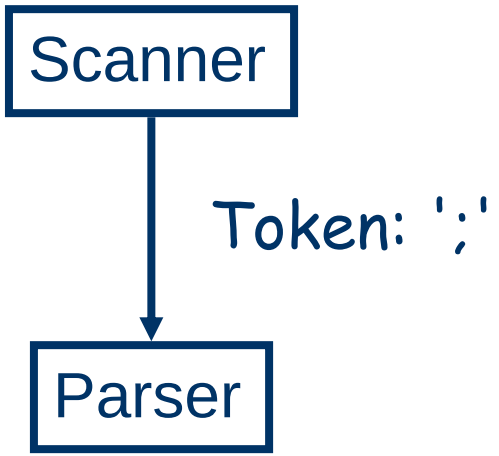
<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>

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

<DECL-STMT> → <TYPE>VAR<VAR-LIST>;

Scanning & Parsing Combined

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



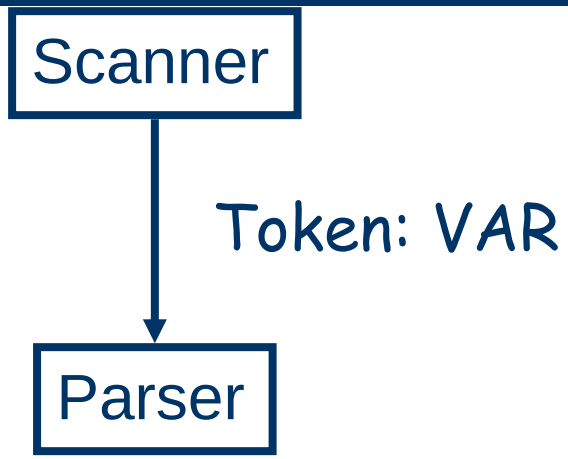
<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>

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

<DECL-STMT> → <TYPE>VAR<VAR-LIST>;

Scanning & Parsing Combined

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



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

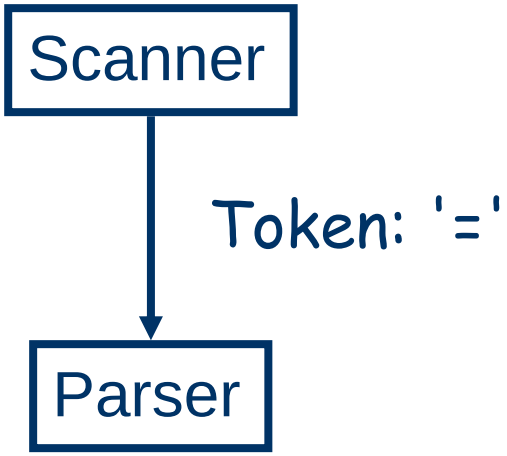
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<ASSIGN-STMT> → VAR = <EXPR>;

<EXPR> → VAR

Scanning & Parsing Combined

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



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

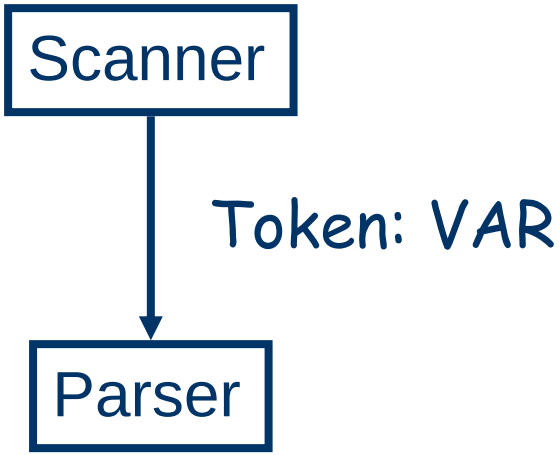
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<ASSIGN-STMT> → VAR = <EXPR>;

<EXPR> → VAR

Scanning & Parsing Combined

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



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

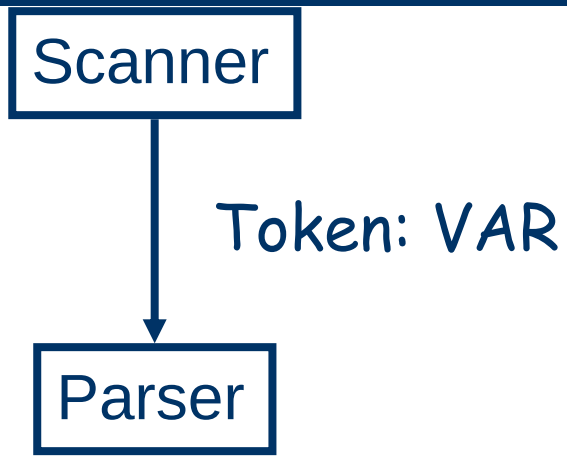
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<ASSIGN-STMT> → VAR = <EXPR>;

<EXPR> → VAR

Scanning & Parsing Combined

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



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

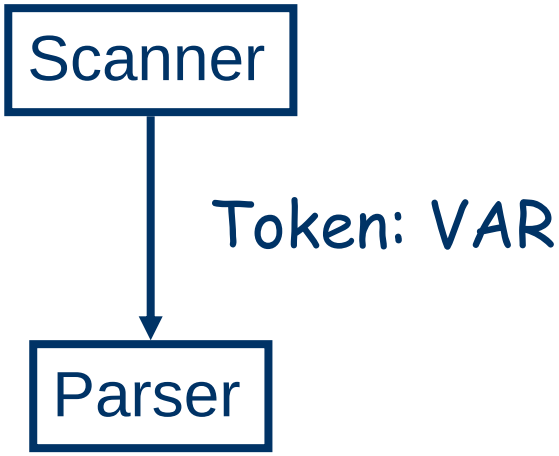
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<ASSIGN-STMT> → VAR = <EXPR>;

<EXPR> → VAR

Scanning & Parsing Combined

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



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

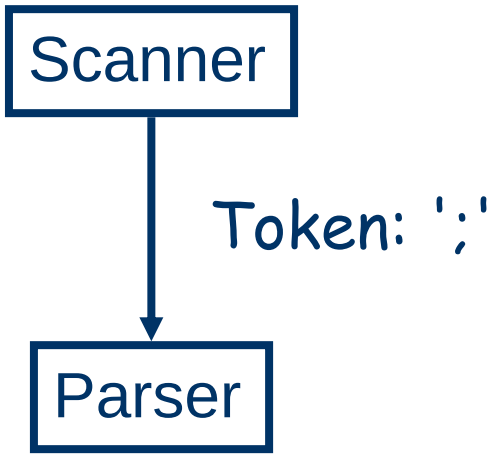
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<ASSIGN-STMT> → VAR = <EXPR>;

<EXPR> → VAR

Scanning & Parsing Combined

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



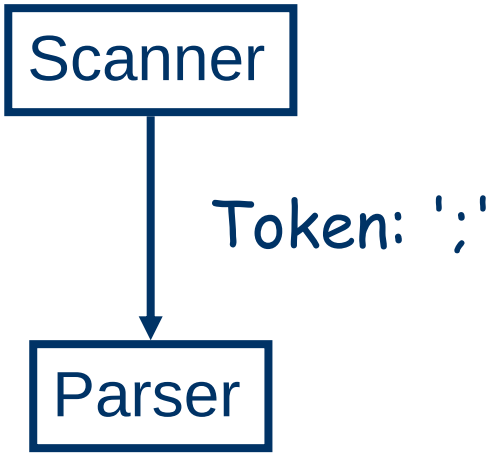
<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>

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

<ASSIGN-STMT> → VAR = <EXPR>;

Scanning & Parsing Combined

```
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 & Parsing Combined

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

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

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

THANKS