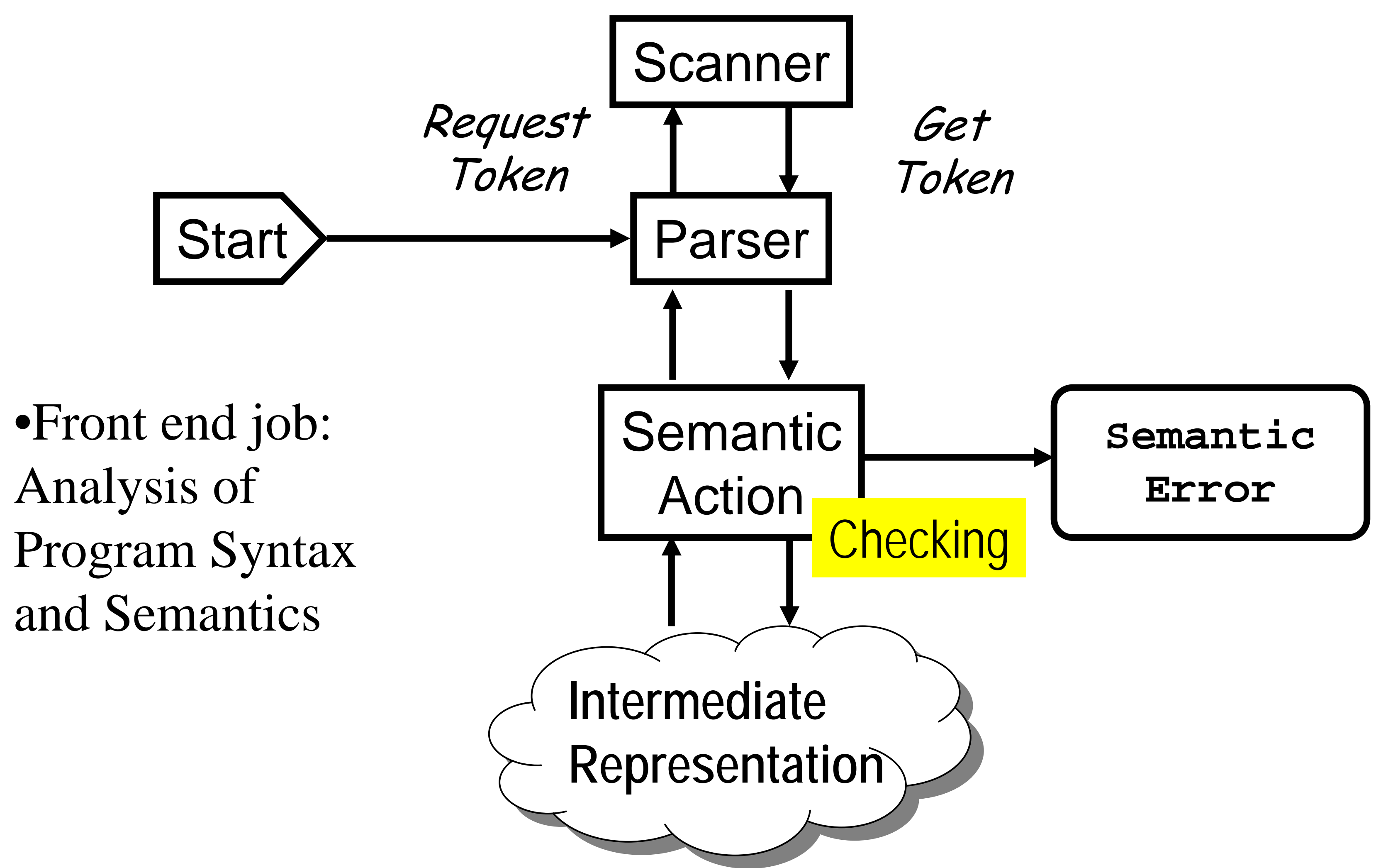
CS 3240

Presentation 2

Compiler Introduction

Compiler Phases – Front End



•Front end job: Analysis of Program Syntax and Semantics

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

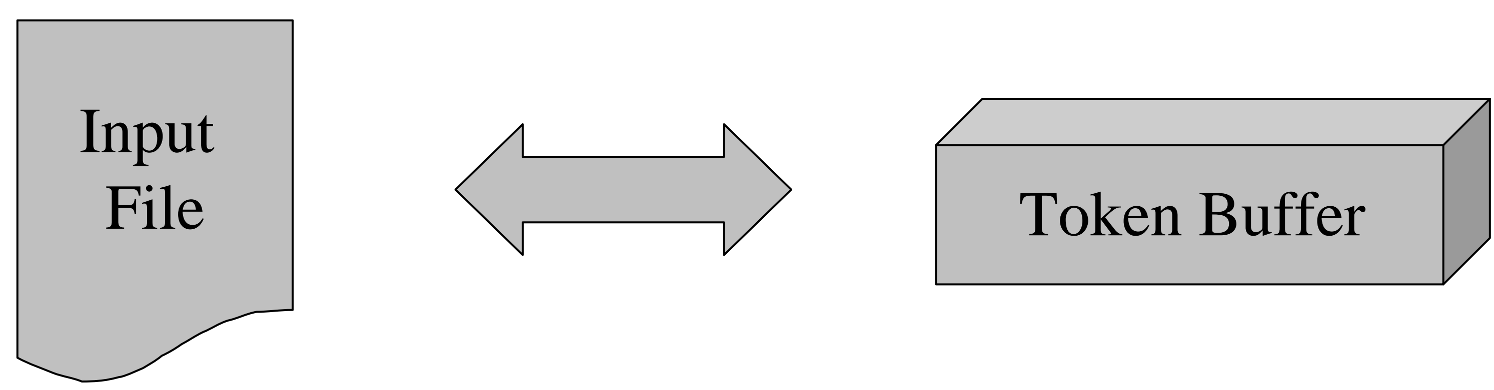
- Parser is in control of the overall operation
- Demands scanner to produce a token
- Scanner reads input file into token buffer & forms a token (How?)
- Token is returned to parser
- Parser attempts to match the token (How?)
- Failure: Syntax Error!
- Success:
  - Does nothing and returns to get next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

    or
  - Takes Semantic Action

Overall Operation

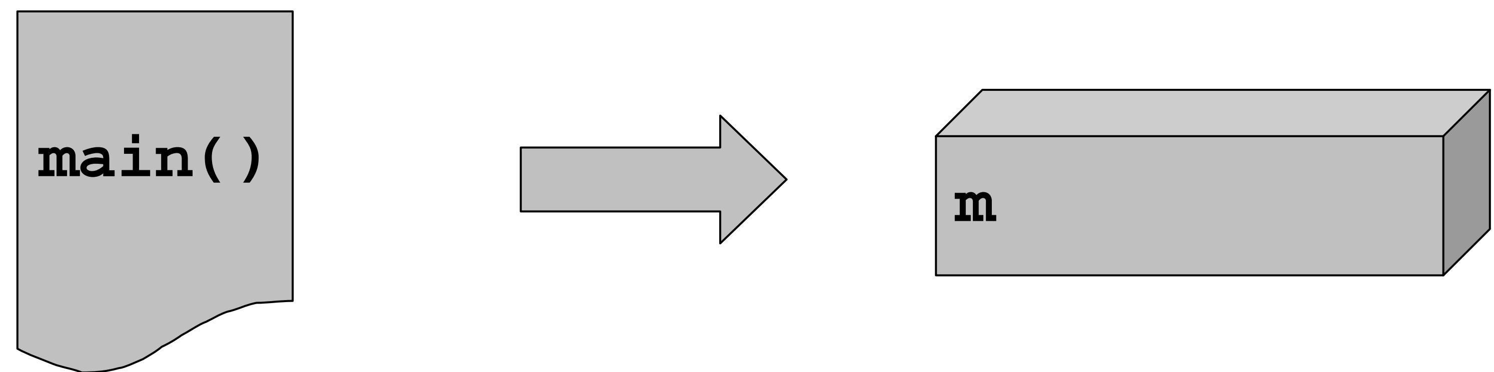
- Semantic Action: Lookup variable name
  - If found okay
  - If not: Put in symbol table
- If semantic checks succeed, do code-generation (How?)
- Return to get next token
- No more tokens? Done!

Tokenization

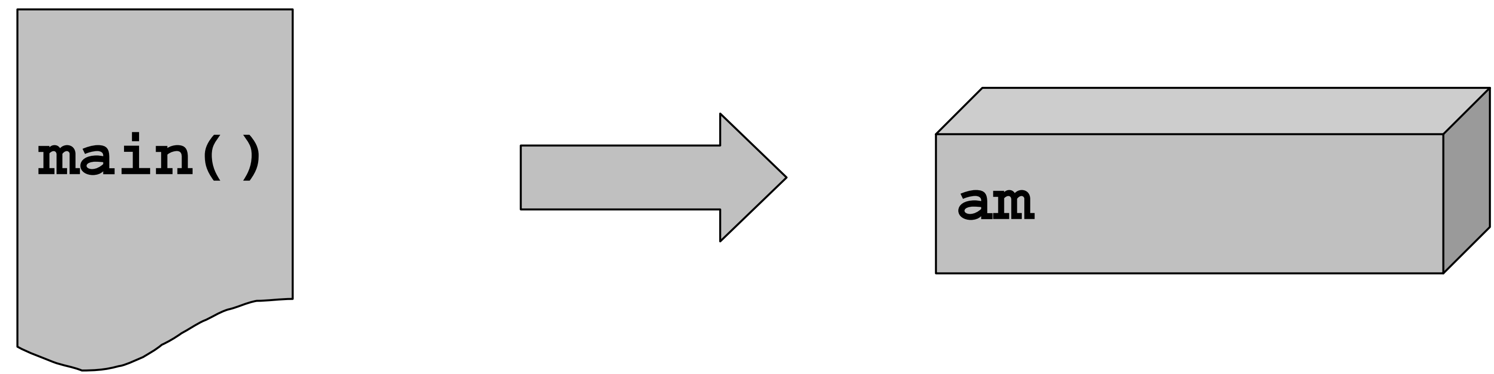


- What does the Token Buffer contain?
  - Token being identified
- Why a two-way (⟺) street?
  - Characters can be read ⟹
  - and unread ⟸
  - Termination of a token

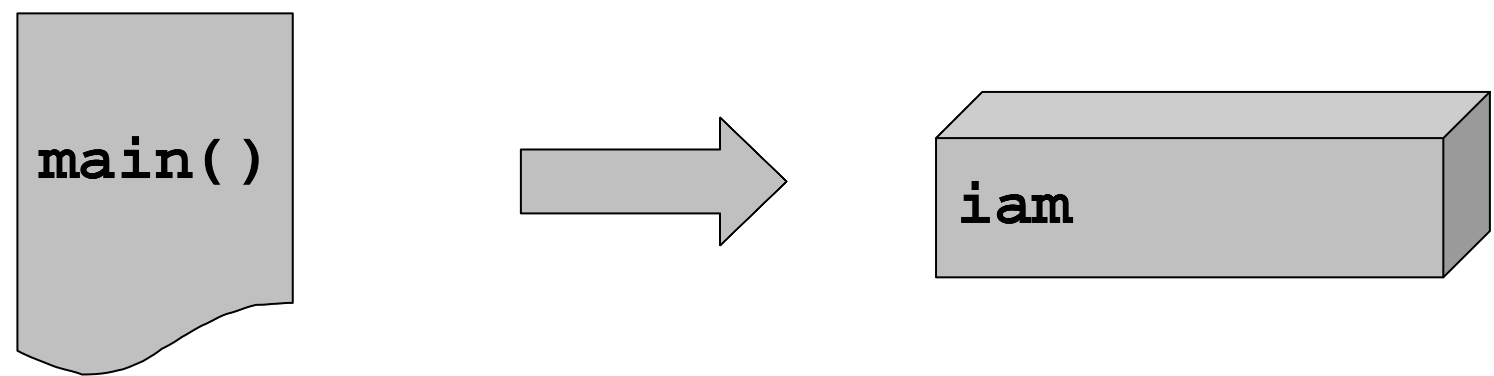
Example



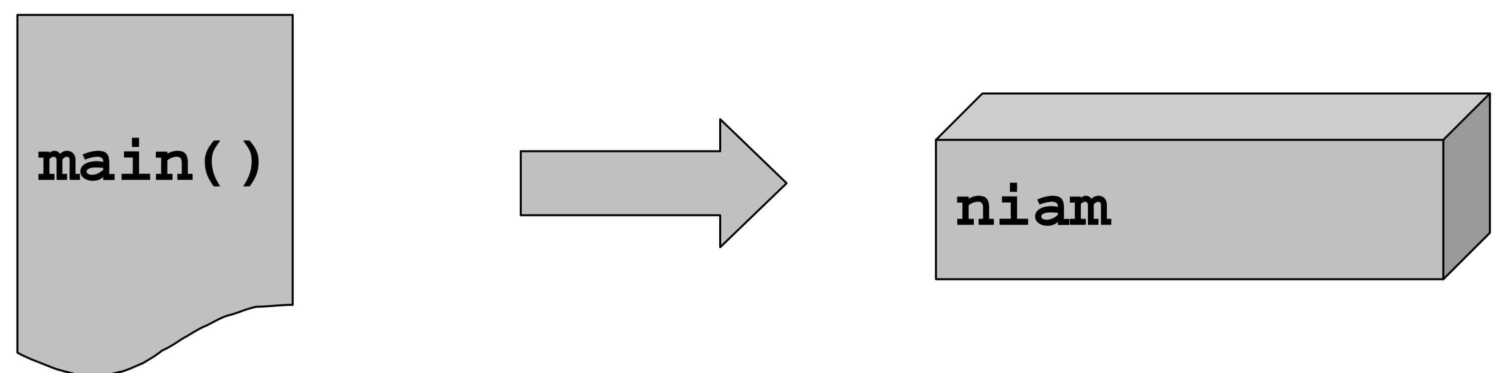
Example



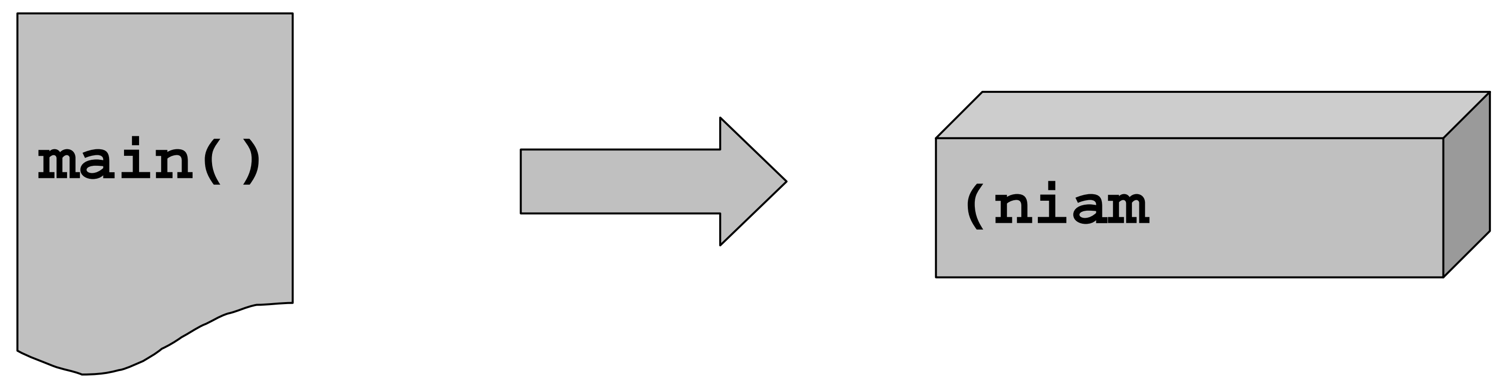
Example



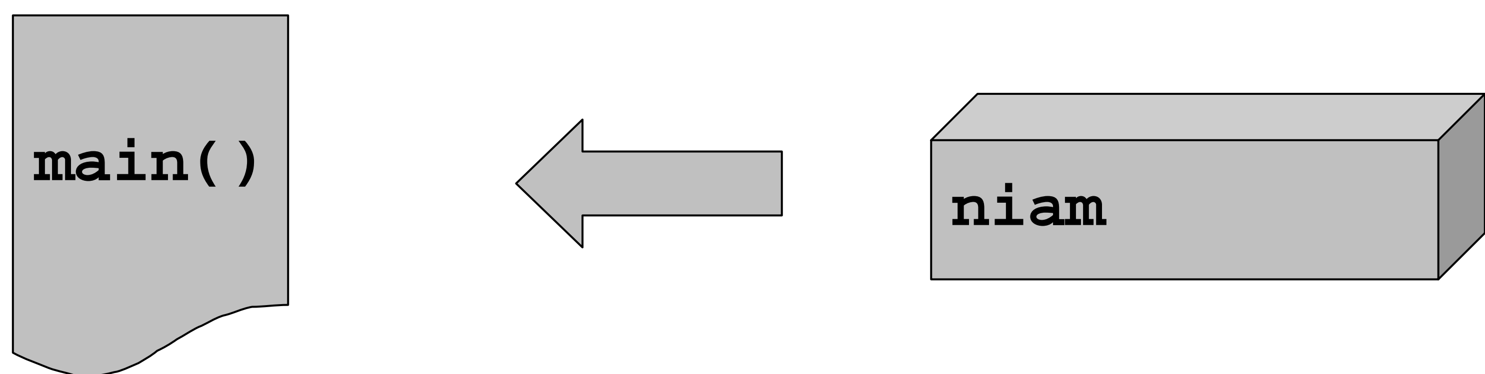
Example



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 (How?)
- Token is returned to parser
- Parser attempts to match the token (How?)
- 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 (How?)
- 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;
}
```

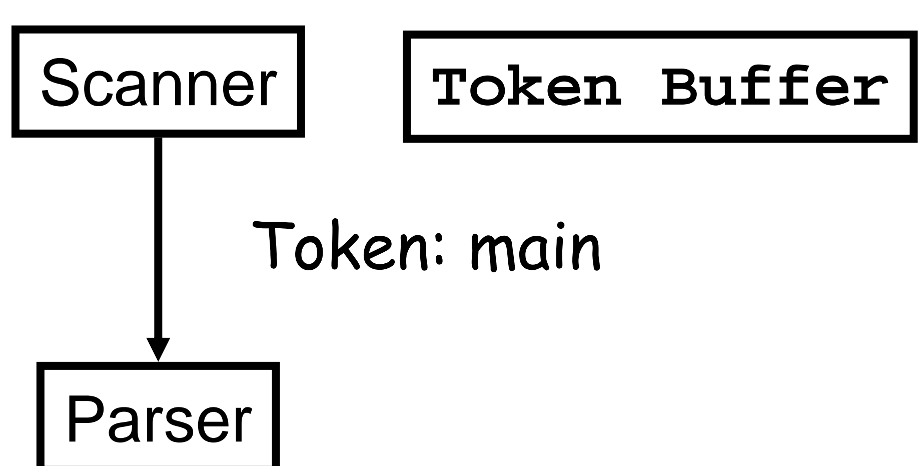
Scanner

```
niam
```

Parser

Demo

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



Demo

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

Scanner

Token Buffer

Parser

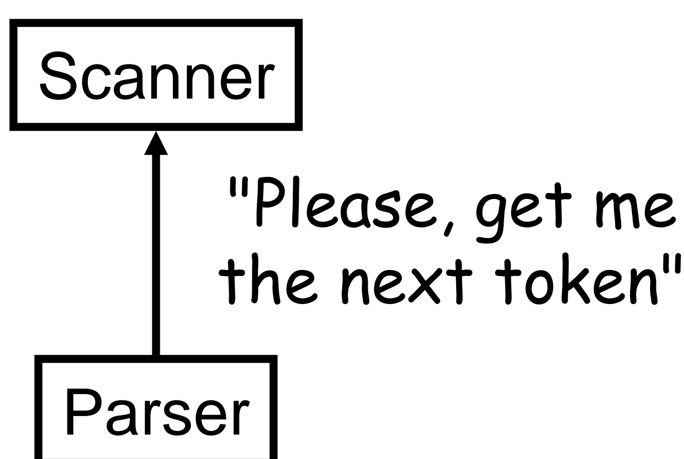
"I recognize this"

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?

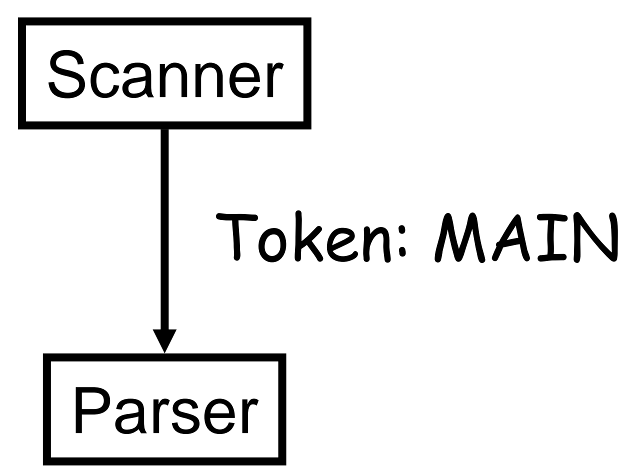
Scanning & Parsing Combined

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



Scanning & Parsing Combined

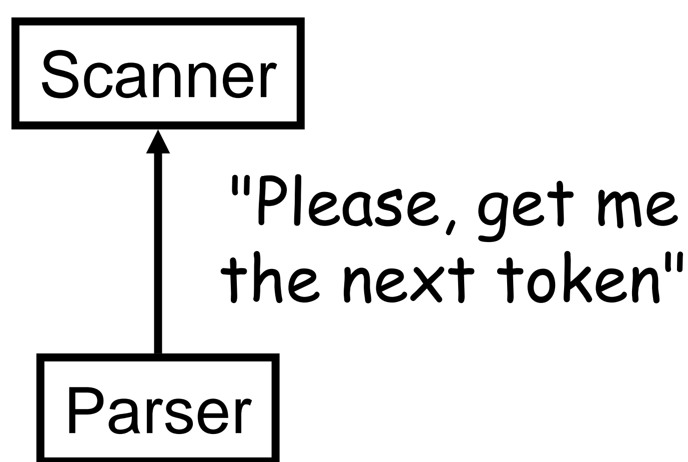
```
main() {
   int a,b;
   a = b;
}
```



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

Scanning & Parsing Combined

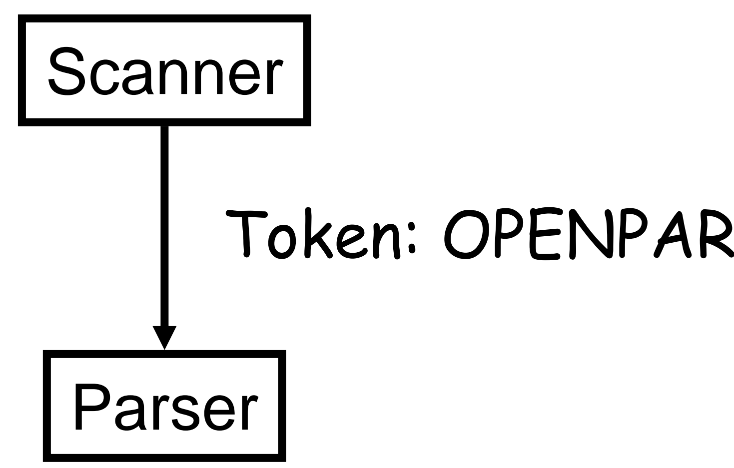
```
main() {
   int a,b;
   a = b;
}
```



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

Scanning & Parsing Combined

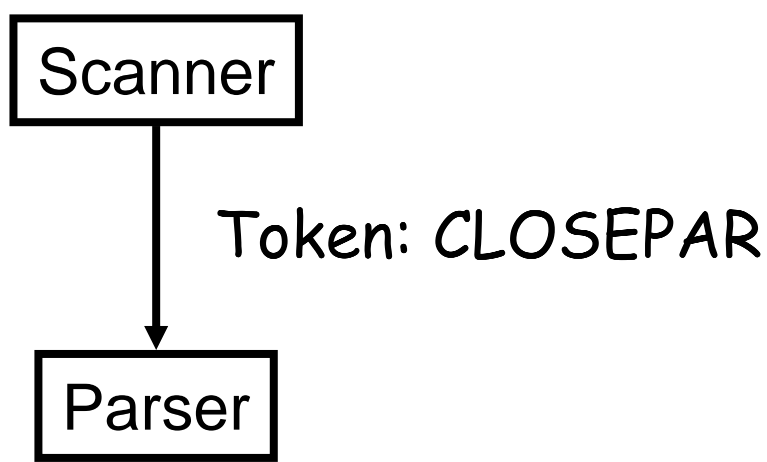
```
main() {
   int a,b;
   a = b;
}
```



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

Scanning & Parsing Combined

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

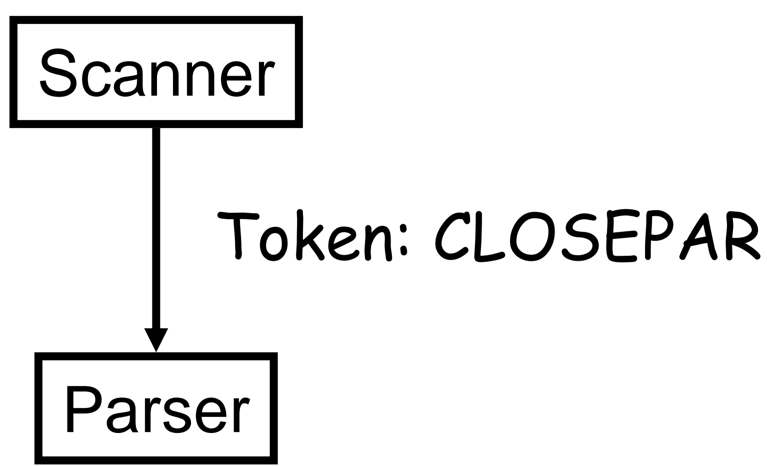


<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>

<PARAMETERS> → NULL

Scanning & Parsing Combined

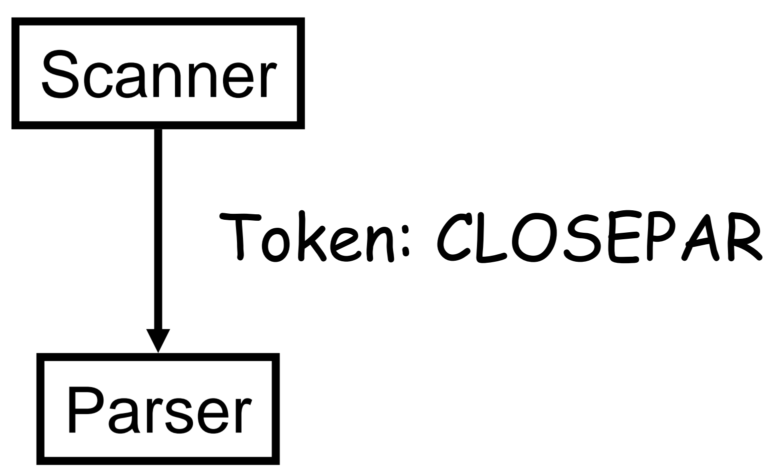
```
main() {
   int a,b;
   a = b;
}
```



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

Scanning & Parsing Combined

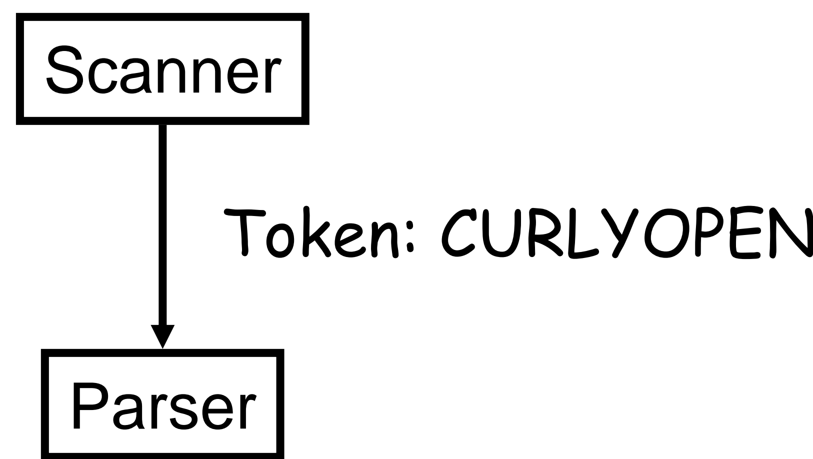
```
main() {
   int a,b;
   a = b;
}
```



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

Scanning & Parsing Combined

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

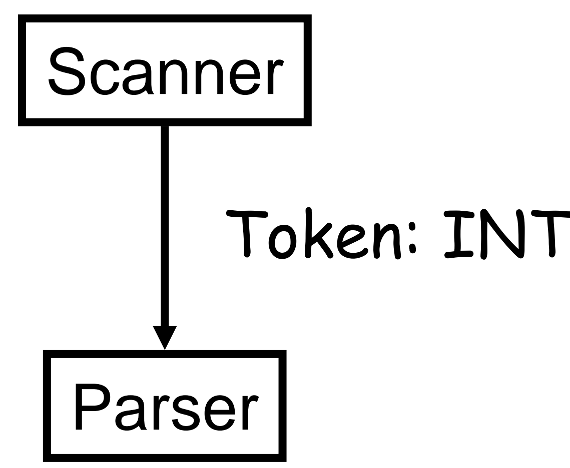


<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>

<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>

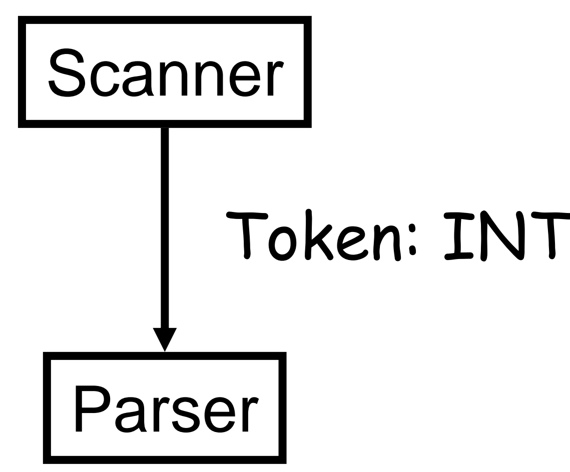
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<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>

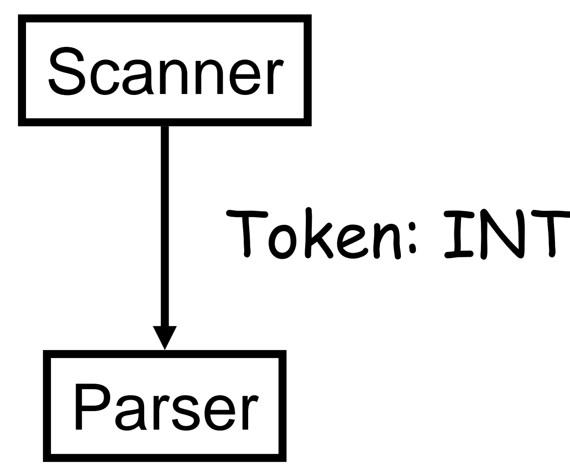
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<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>

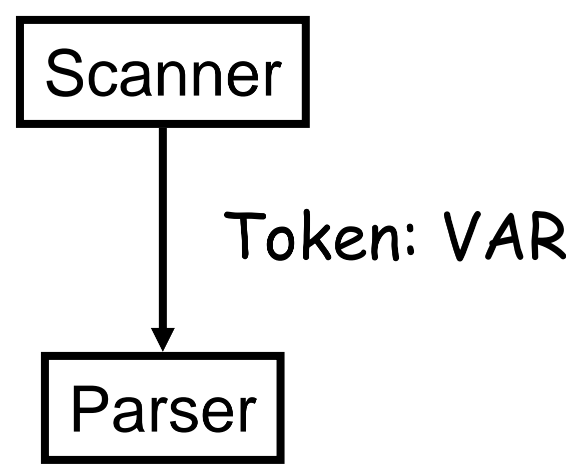
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<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

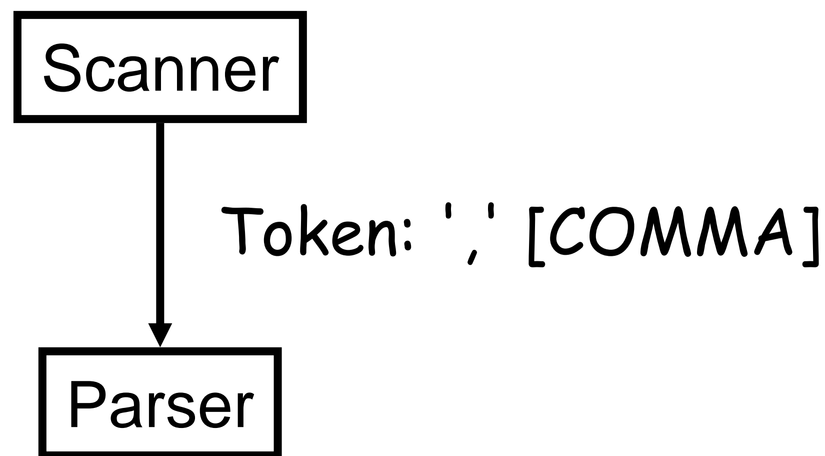
<DECL-STMT> → <TYPE>VAR<VAR-LIST>;

<VARLIST> → , VAR <VARLIST>

<VARLIST> → NULL

Scanning & Parsing Combined

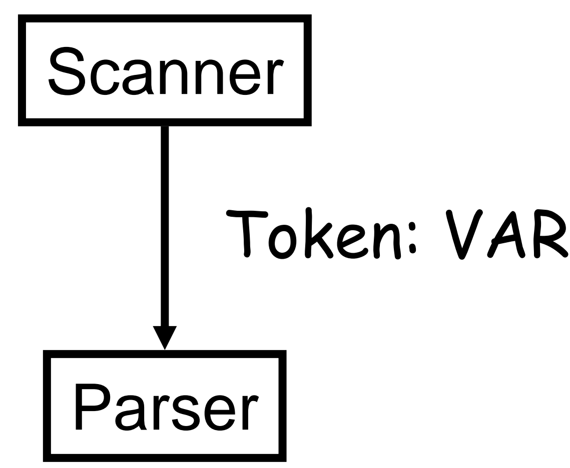
```
main() {
   int a,b;
   a = b;
}
```




```
<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

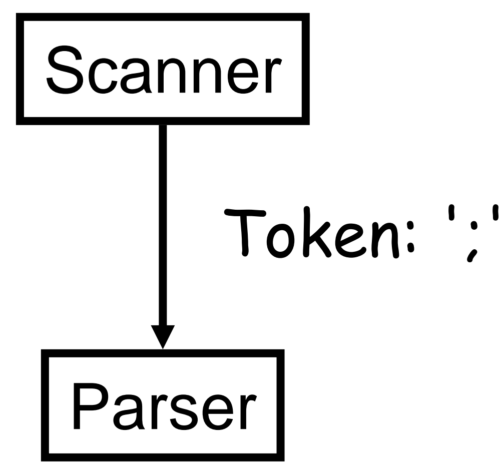
```
main() {
   int a,b;
   a = b;
}
```




```
<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

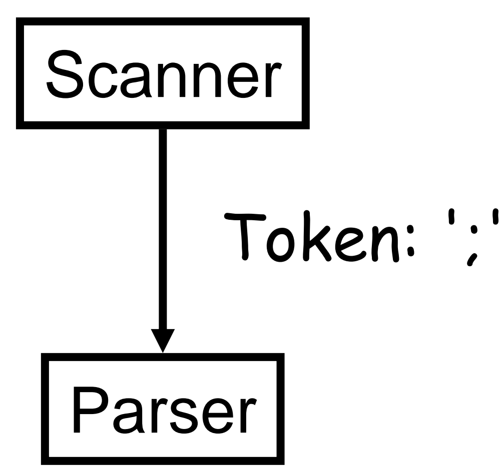
<DECL-STMT> → <TYPE>VAR<VAR-LIST>;

<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

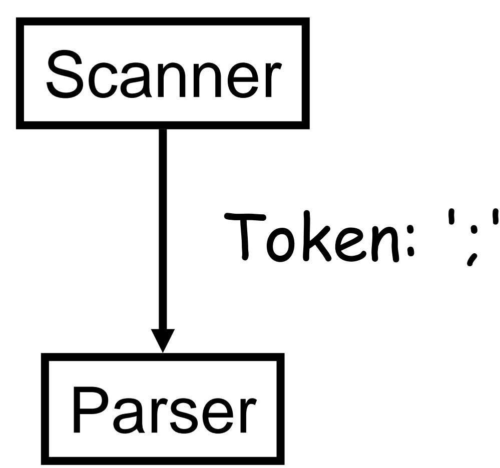
<DECL-STMT> → <TYPE>VAR<VAR-LIST>;

<VARLIST> → , VAR <VARLIST>

<VARLIST> → NULL

Scanning & Parsing Combined

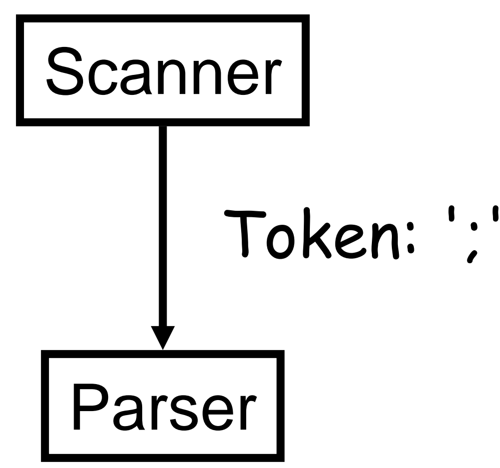
```
main() {
   int a,b;
   a = b;
}
```




```
<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

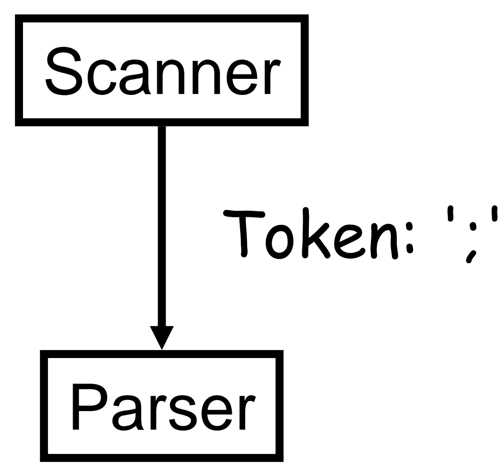
<DECL-STMT> → <TYPE>VAR<VAR-LIST>;

<VARLIST> → , VAR <VARLIST>

<VARLIST> → NULL

Scanning & Parsing Combined

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



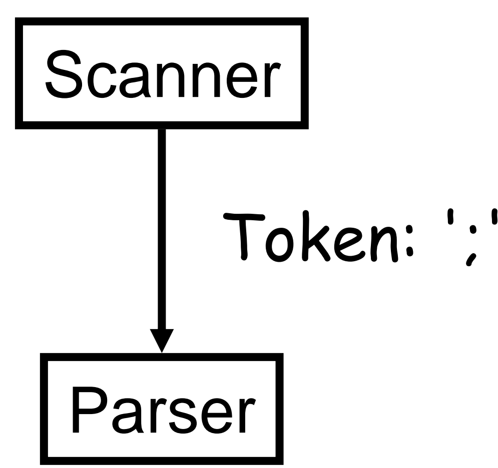
<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>

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

<DECL-STMT> → <TYPE>VAR<VAR-LIST>;

Scanning & Parsing Combined

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



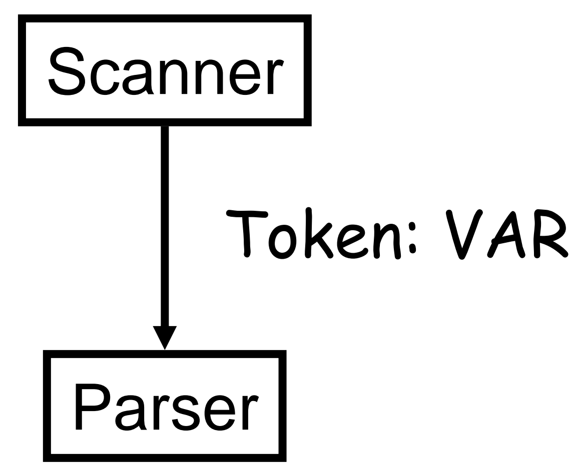
<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>

<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>

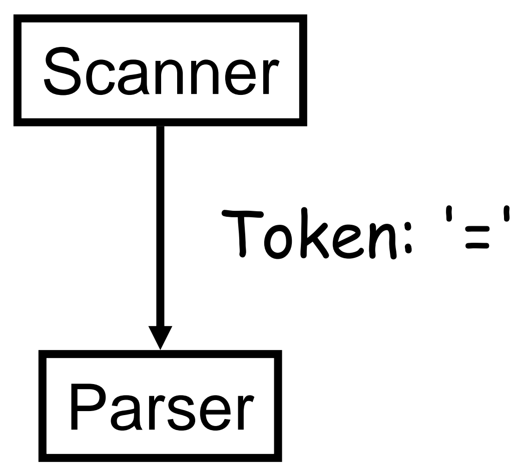
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<ASSIGN-STMT> → VAR = <EXPR>;

<EXPR> → VAR

Scanning & Parsing Combined

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



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

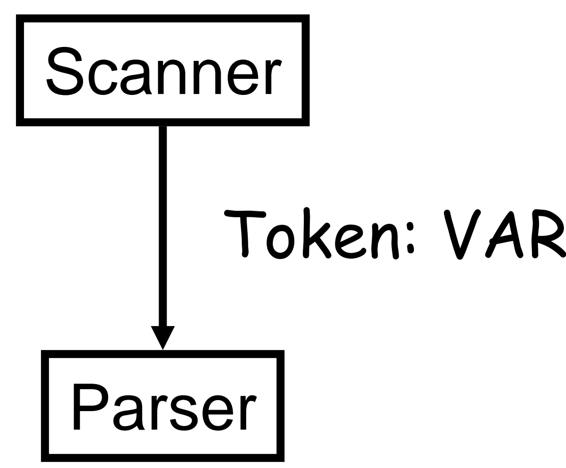
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<ASSIGN-STMT> → VAR = <EXPR>;

<EXPR> → VAR

Scanning & Parsing Combined

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



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

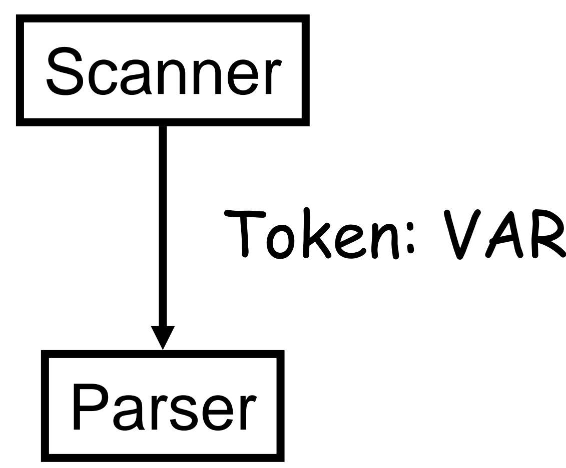
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<ASSIGN-STMT> → VAR = <EXPR>;

<EXPR> → VAR

Scanning & Parsing Combined

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



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

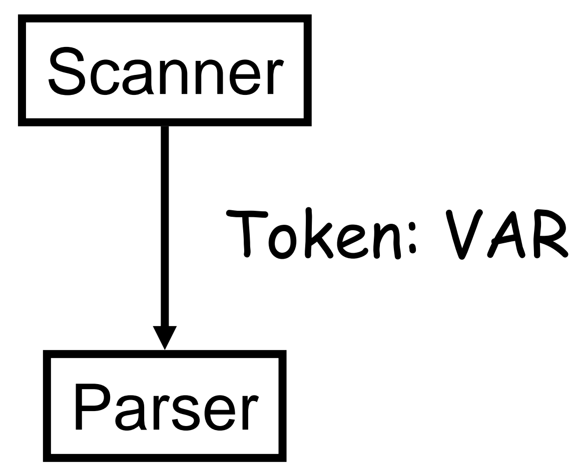
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<ASSIGN-STMT> → VAR = <EXPR>;

<EXPR> → VAR

Scanning & Parsing Combined

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



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

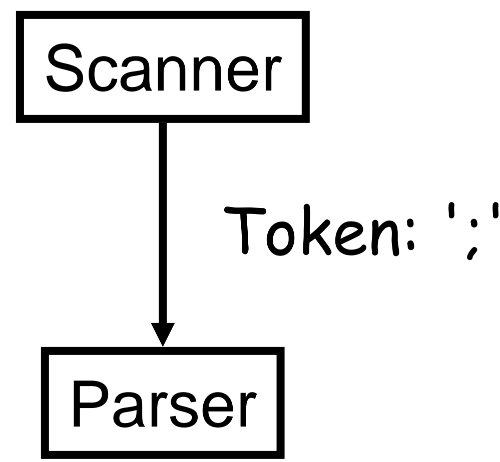
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE

<ASSIGN-STMT> → VAR = <EXPR>;

<EXPR> → VAR

Scanning & Parsing Combined

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



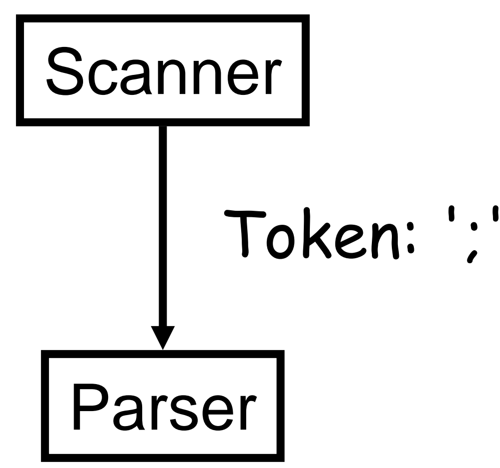
<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>

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

<ASSIGN-STMT> → VAR = <EXPR>;

Scanning & Parsing Combined

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



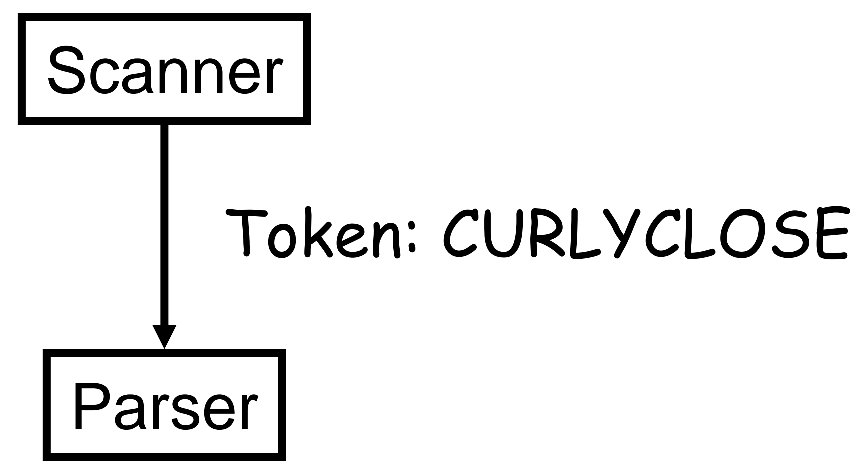
<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>

<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

What happens?

- During/after parsing?
  - Tokens get gobbled
  - Screen blinks?
  - Smoke comes out?
- Further checking and setup for checks
- Semantic actions
- Semantic checks
- Symbol tables
- What are semantic actions?
- Variables have attributes
- Declaration attached attributes to variables

Symbol Table

- int a,b;
- Declares a and b
  - Within current scope
  - Of type integer
- Use of a and b now legal

Basic Symbol Table		
Name	Type	Scope
a	int	"main"
b	int	"main"

Semantic Actions

- What are typical Semantic Actions?
- How do they get invoked?
- What happens after a Semantic Action?

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
```

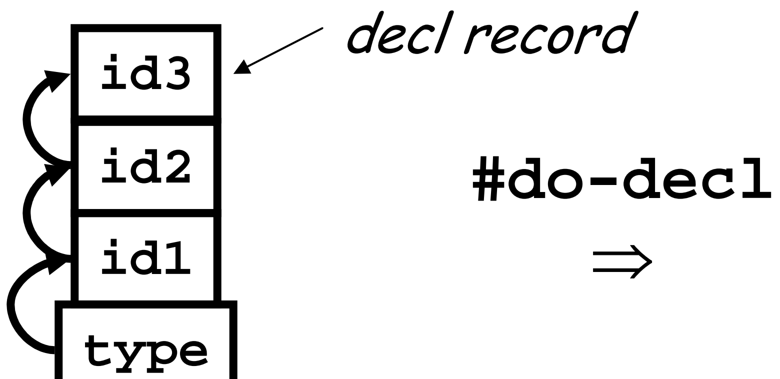
How are Semantic Actions Called?

- Action symbols embedded in the grammar
- Each action symbol represents a semantic procedure
- Semantic procedures are called by parser at appropriate places during parsing
- These procedures do things and/or return values
- Semantic stack implements & stores semantic records
- Semantic actions could do checking and/or storage or retrieval of information

Semantic Actions

<decl-stmt> → <type>#put-type<var-list>#do-decl
<type> → int | float
<var-list> → <var>#add-decl<, <var-list>
var-list> → <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



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?
- Do type checking for type compatibility & assignment
- Two type of Semantic Actions
  - Checking (Binding, Type Compatibility, Scoping, etc.)
  - Translation (Generate temporary values, propagate them to keep semantic context).

Semantic Actions (Translation)

- Consider `a = b + c + d;`

- Grammar:

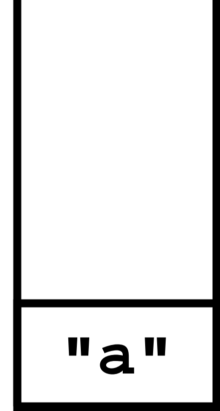
```
<ASSGNSTMT> → <VAR> = <EXPR>#do-assign;
<EXPR>      → <VAR><EXPRTAIL>
<VAR>       → ID#process-id
<EXPRTAIL>  → <OP>#process-op<VAR>#do-infix<EXPRTAIL>
<EXPRTAIL>   → NULL
```

```
a = b + c + d;
```

Call Chain to Semantic Actions

- **`#process-id`**: Puts semantic record for "a" on stack

Q. Checking or Translation?




```
<ASSGNSTMT> → <VAR> = <EXPR>#do-assign;
<EXPR>      → <VAR><EXPRTAIL>
<VAR>       → ID#process-id
<EXPRTAIL>  → <OP>#process-op<VAR>#do-infix<EXPRTAIL>
<EXPTAIL>   → NULL
```

```
a = b + c + d;
```

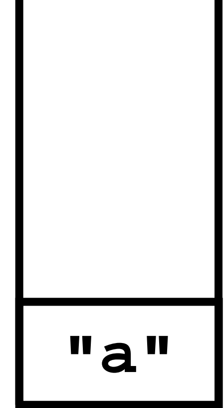
Call Chain to Semantic Actions

- **`#process-id`**: Puts semantic record for "a" on stack

Q. Checking or Translation?

A. Checking

```
<ASSGNSTMT> → <VAR> = <EXPR>#do-assign;
<EXPR>      → <VAR><EXPRTAIL>
<VAR>       → ID#process-id
<EXPRTAIL>  → <OP>#process-op<VAR>#do-infix<EXPRTAIL>
<EXPTAIL>   → NULL
```



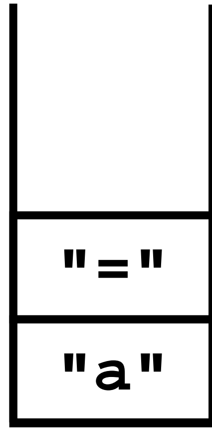
`a = b + c + d;`

Call Chain to Semantic Actions

- **`#process-op`**: Puts semantic record for "=" on stack

Q. Checking or Translation?

```
<ASSGNSTMT> → <VAR> = <EXPR>#do-assign;
<EXPR>      → <VAR><EXPRTAIL>
<VAR>       → ID#process-id
<EXPRTAIL>  → <OP>#process-op<VAR>#do-infix<EXPRTAIL>
<EXPTAIL>   → NULL
```



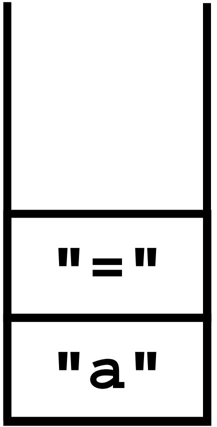
```
a = b + c + d;
```

Call Chain to Semantic Actions

- **`#process-op`**: Puts semantic record for "=" on stack

Q. Checking or Translation?

A. Checking

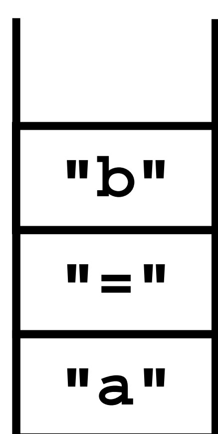



```
<ASSGNSTMT> → <VAR> = <EXPR>#do-assign;
<EXPR>      → <VAR><EXPRTAIL>
<VAR>       → ID#process-id
<EXPRTAIL>  → <OP>#process-op<VAR>#do-infix<EXPRTAIL>
<EXPTAIL>   → NULL
```

```
a = b + c + d;
```

Call Chain to Semantic Actions

- **#process-id**: Puts semantic record for "b" on stack



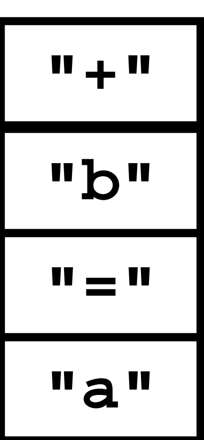

```
<ASSGNSTMT> → <VAR> = <EXPR>#do-assign;
<EXPR>      → <VAR><EXPRTAIL>
<VAR>       → ID#process-id
<EXPRTAIL>  → <OP>#process-op<VAR>#do-infix<EXPRTAIL>
<EXPTAIL>   → NULL
```

`a = b + c + d;`

Call Chain to Semantic Actions

- **`#process-id`**: Puts semantic record for "+" on stack

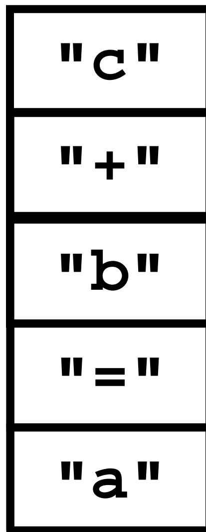
```
<ASSGNSTMT> → <VAR> = <EXPR>#do-assign;
<EXPR>      → <VAR><EXPRTAIL>
<VAR>       → ID#process-id
<EXPRTAIL>  → <OP>#process-op<VAR>#do-infix<EXPRTAIL>
<EXPTAIL>   → NULL
```



`a = b + c + d;`

Call Chain to Semantic Actions

- **`#process-id`**: Puts semantic record for "c" on stack

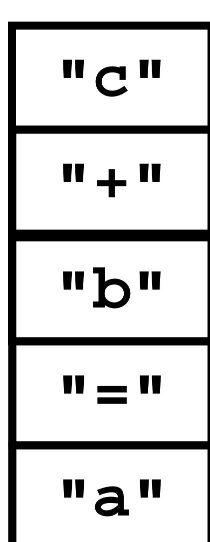



```
<ASSGNSTMT> → <VAR> = <EXPR>#do-assign;
<EXPR>      → <VAR><EXPRTAIL>
<VAR>       → ID#process-id
<EXPRTAIL>  → <OP>#process-op<VAR>#do-infix<EXPRTAIL>
<EXPTAIL>   → NULL
```

`a = b + c + d;`

Call Chain to Semantic Actions

- **`#do-infix`**:
  - Get temporary (say t1)
  - Evaluate
  - IR: `t1 + b c`

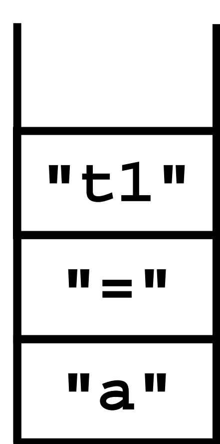



```
<ASSGNSTMT> → <VAR> = <EXPR>#do-assign;
<EXPR>      → <VAR><EXPRTAIL>
<VAR>       → ID#process-id
<EXPRTAIL>  → <OP>#process-op<VAR>#do-infix<EXPRTAIL>
<EXPTAIL>   → NULL
```

`a = b + c + d;`

Call Chain to Semantic Actions

- **`#do-infix`**:
  - Get temporary (say t1)
  - Evaluate
  - IR: `t1 + b c`

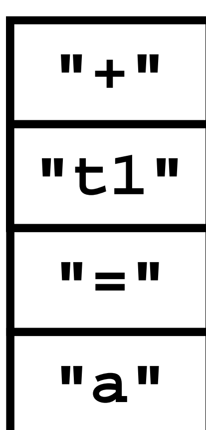



```
<ASSGNSTMT> → <VAR> = <EXPR>#do-assign;
<EXPR>      → <VAR><EXPRTAIL>
<VAR>       → ID#process-id
<EXPRTAIL>  → <OP>#process-op<VAR>#do-infix<EXPRTAIL>
<EXPTAIL>   → NULL
```

```
a = b + c + d;
```

Call Chain to Semantic Actions

- `#process-id`: Puts semantic record for "+" on stack

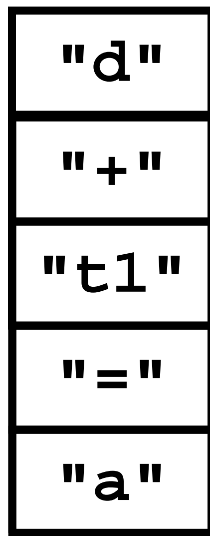



```
<ASSGNSTMT> → <VAR> = <EXPR>#do-assign;
<EXPR>      → <VAR><EXPRTAIL>
<VAR>       → ID#process-id
<EXPRTAIL>  → <OP>#process-op<VAR>#do-infix<EXPRTAIL>
<EXPTAIL>   → NULL
```

```
a = b + c + d;
```

Call Chain to Semantic Actions

- **`#process-id`**: Puts semantic record for "d" on stack

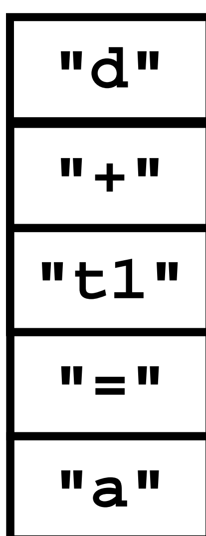



```
<ASSGNSTMT> → <VAR> = <EXPR>#do-assign;
<EXPR>      → <VAR><EXPRTAIL>
<VAR>       → ID#process-id
<EXPRTAIL>  → <OP>#process-op<VAR>#do-infix<EXPRTAIL>
<EXPTAIL>   → NULL
```

`a = b + c + d;`

Call Chain to Semantic Actions

- **`#do-infix`**:
  - Get temporary (say t2)
  - Evaluate
  - IR: `t2 + t1 d`

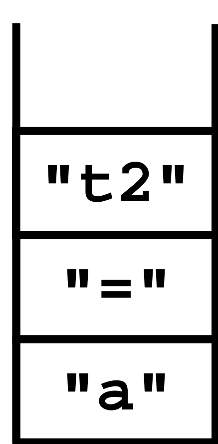



```
<ASSGNSTMT> → <VAR> = <EXPR>#do-assign;
<EXPR>      → <VAR><EXPRTAIL>
<VAR>       → ID#process-id
<EXPRTAIL>  → <OP>#process-op<VAR>#do-infix<EXPRTAIL>
<EXPTAIL>   → NULL
```

`a = b + c + d;`

Call Chain to Semantic Actions

- **`#do-infix`**:
  - Get temporary (say t2)
  - Evaluate
  - IR: `t2 + t1 d`

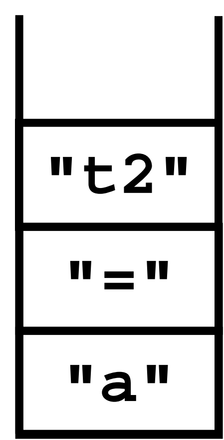



```
<ASSGNSTMT> → <VAR> = <EXPR>#do-assign;
<EXPR>      → <VAR><EXPRTAIL>
<VAR>       → ID#process-id
<EXPRTAIL>  → <OP>#process-op<VAR>#do-infix<EXPRTAIL>
<EXPTAIL>   → NULL
```

`a = b + c + d;`

Call Chain to Semantic Actions

- **`#do-assign`**:
  - Put value back into variable
  - IR: `a` ← `t2`




```
<ASSGNSTMT> → <VAR> = <EXPR>#do-assign;
<EXPR>      → <VAR><EXPRTAIL>
<VAR>       → ID#process-id
<EXPRTAIL>  → <OP>#process-op<VAR>#do-infix<EXPRTAIL>
<EXPTAIL>   → NULL
```

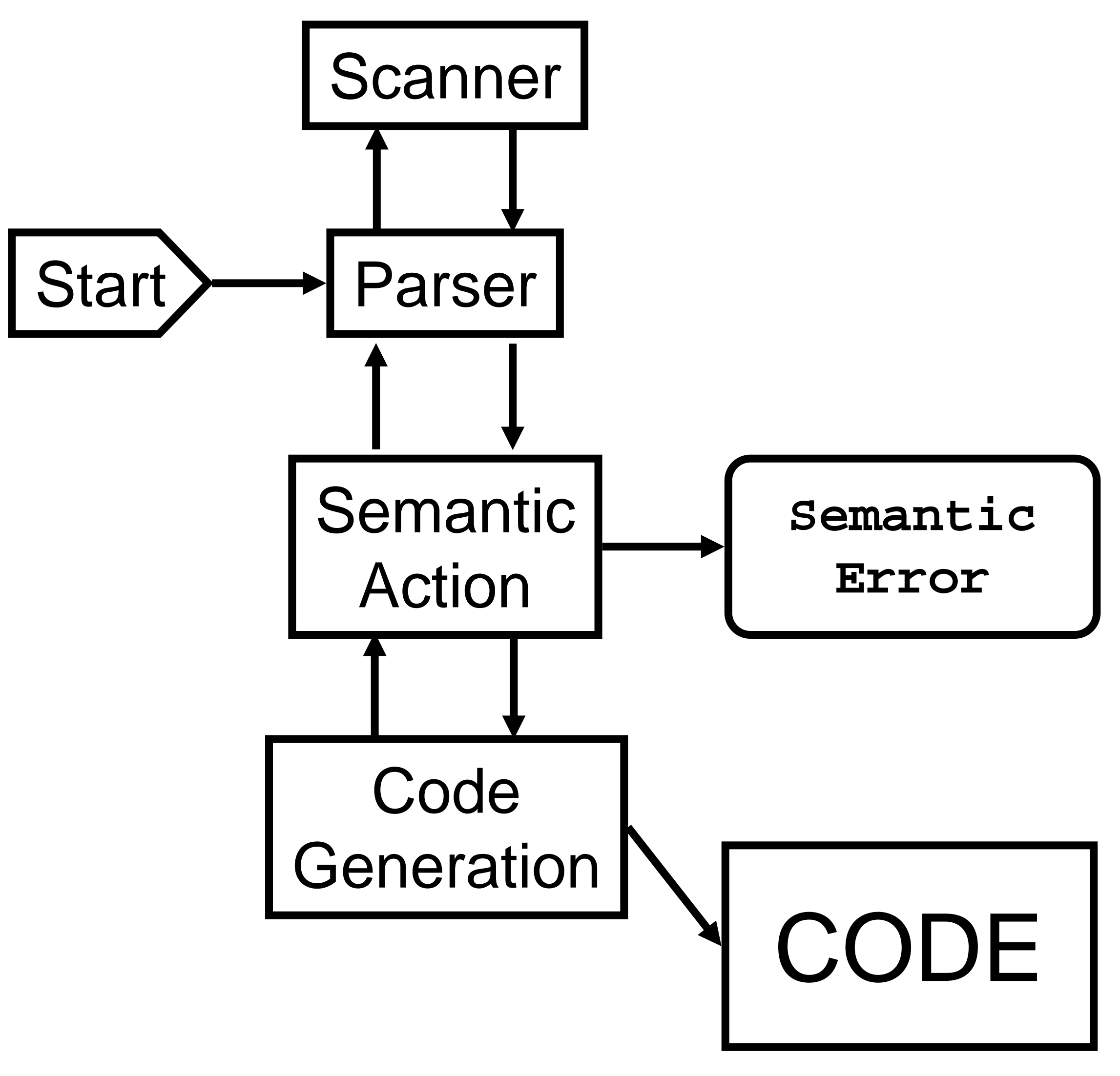
```
a = b + c + d;
```

Call Chain to Semantic Actions

- **`#do-assign`**:
  - Put value back into variable
  - IR: `a ← t2`

```
<ASSGNSTMT> → <VAR> = <EXPR>#do-assign;
<EXPR>      → <VAR><EXPRTAIL>
<VAR>       → ID#process-id
<EXPRTAIL>  → <OP>#process-op<VAR>#do-infix<EXPRTAIL>
<EXPTAIL>   → NULL
```

Full Compiler Structure



- Most compilers are two pass

Summary

- Parser is the brain of the compiler
- Controls everything
- Most of the time is spent in keeping & checking information (front-end job)
- Translation takes place in back end
- Scanner, parser and code generator are automated (How and Why?)
- A lot of string processing
- How to express?
- How to perform?
- Can it be automated (generate a machine which can do the computation automatically?)

Questions?