

# Parsing using CYK Algorithm

- Transform grammar into Chomsky Form:
  1. remove unproductive symbols
  2. remove unreachable symbols
  3. remove epsilons (no non-start nullable symbols)
  4. remove single non-terminal productions  $X ::= Y$
  5. transform productions of arity more than two
  6. make terminals occur alone on right-hand side

Have only rules  $X ::= Y Z$ ,  $X ::= t$

- Apply CYK dynamic programming algorithm

## Questions:

- With steps in the order above, what is the worst-case increase in grammar size, in each step and overall?
- Does any step break the effect of a previous one?
- Propose alternative step order and answer again the above.
- Which steps could we omit and still have CYK working?

# Suggested Order

- Removing epsilons (3) can increase grammar size exponentially
- This problem is avoided if we make rules binary first (5).
- Removing epsilons can make some symbols unreachable, so we can repeat 2
- Resulting order:  
1,2,5,3,4,2,6

# A CYK for Any Grammar

grammar  $G$ , non-terminals  $A_1, \dots, A_K$ , tokens  $t_1, \dots, t_L$

input word:  $w = w_{(0)} w_{(1)} \dots w_{(N-1)}$

$w_{p..q} = w_{(p)} w_{(p+1)} \dots w_{(q-1)}$

Triple  $(A, p, q)$  means:  $A \Rightarrow^* w_{p..q}$ ,  $A$  can be:  $A_i$ ,  $t_j$ , or  $\varepsilon$

$P = \{(w_{(i)}, i, i+1) \mid 0 \leq i < N-1\}$

repeat {

    choose rule  $(A ::= B_1 \dots B_m) \in G$

    if  $((A, p_0, p_m) \notin P \ \&\&$

$((m=0 \ \&\& \ p_0=p_m) \ || \ (B_1, p_0, p_1), \dots, (B_m, p_{m-1}, p_m) \in P))$

$P := P \cup \{(A, p_0, p_m)\}$

    } until no more insertions possible

What is the maximal number of steps?

How long does it take to check step for a rule?

} for grammar in  
given normal form

# Observation

- How many ways are there to split a string of length  $Q$  into  $m$  segments?

$$\binom{Q+m}{m} = \frac{(Q+m)!}{Q!m!}$$

- Exponential in  $m$ , so algorithm is exponential.
- For binary rules,  $m=2$ , so algorithm is efficient.

# A CYK Algorithm Producing Results

input word:  $w = w_{(0)}w_{(1)} \dots w_{(N-1)}$  ,  $w_{p..q} = w_{(p)}w_{(p+1)} \dots w_{(q-1)}$

Non-terminals  $A_1, \dots, A_K$ , tokens  $t_1, \dots, t_L \in T$

Rule  $(A ::= B_1 \dots B_m, f) \in G$  with semantic action  $f$ .  $R$  - result (e.g. tree)

$f : ((A \times N \times N) \times (RUT))^m \rightarrow R$

Useful parser: returning a set of result (e.g. syntax trees)

$((A, p, q), r)$ :  $A \Rightarrow^* w_{p..q}$  and the result can be interpreted as  $r$

Let  $f$  be **partial** function, we apply it only if the result is defined

$P = \{((w_{(i)}, i, i+1), w_{(i)}) \mid 0 \leq i < N-1\}$  // set of  $((A, p, q), r)$

repeat {

  choose rule  $(A ::= B_1 \dots B_m, f) \in G$

  choose  $((B_1, p_0, p_1), r_1), \dots, ((B_m, p_{m-1}, p_m), r_2) \in P$

$P := P \cup \{((A, p_0, p_m), f(((B_1, p_0, p_1), r_1), \dots, ((B_m, p_{m-1}, p_m), r_2)))\}$

  // do nothing if  $f$  is not defined on those args

} until no more insertions into  $P$  possible

# Simple Application: Associativity

$e ::= e - e \mid ID$

**abstract class** Tree

**case class** ID(s:String) **extends** Tree

**case class** Minus(e1:Tree,e2:Tree) **extends** Tree

Define rules that will return only

Suppose minus is left associative

Result attribute type: Tree. Defining semantic action:

General rule: if can parse e from i to j, then minus, then another e from j+1 to k, then can parse e from i to k

$f((e,i,j), \underline{t_1}, (-, j, j+1), \_, ((e,j+1,k), \underline{t_2})) = \text{Minus}(t_1, t_2)$

**Restriction:** only if  $t_2$  is not  $\text{Minus}(\dots, \dots)$  otherwise undefined

Discuss: right associativity, priority of \* over - , parentheses.

$ID - ID - ID$

$(ID - ID) - ID$

$ID - (ID - ID)$



# Priorities

- In addition to the tree, return the priority of the tree
  - usually the priority is the top-level operator
  - parenthesized expressions have high priority, as do other 'atomic' expressions (identifiers, literals)
- Disallow combining trees if the priority of current right-hand-side is higher than priority of results being combining
- Given:  $x - y * z$  with priority of  $*$  higher than of  $-$ 
  - disallow combining  $x-y$  and  $z$  using  $*$
  - allow combining  $x$  and  $y*z$  using  $-$

# Probabilities: Natural Language Processing

Represent the set of tuples  $((A, p, q), r_1), \dots, ((A, p, q), r_n)$   
as a map from  $(A, p, q)$  to ranked priority queue  $r_1, \dots, r_n$   
Example application: probabilistic context-free grammars  
(can be learned from corpus).

Each rule has a probability  $p$

This assigns probability to the space of all possible parse trees  
 $r$  stores pointers to sub-trees and probability of the parse tree  $q$

$$\begin{aligned} &f( ((B_1, p_0, p_1), (\_, q_1)), \dots, ((B_m, p_{m-1}, p_m), (\_, q_m)) ) \\ &= ( (B_1, p_0, p_1) \dots (B_m, p_{m-1}, p_m) , p \cdot q_1 \cdot \dots \cdot q_m ) \end{aligned}$$

For binary rules: how we split it, and what probability we got.

more (optional) in book: **Daniel Jurafsky, James H. Martin:**  
**Speech and Language Processing, Pearson (2nd edition), (Part III)**

<http://www.cs.colorado.edu/~martin/SLP/>



# Attribute Grammars

- They extend context-free grammars to give parameters to non-terminals, have rules to combine attributes
- Attributes can have any type, but often they are trees
- Example:

- context-free grammar rule:

$A ::= B C$

- attribute grammar rules:

$A ::= B C \{ \text{Plus}(\$1, \$2) \}$

or, e.g.

$A ::= B:x C:y \{ \text{RESULT} := \text{new Plus}(x.v, y.v) : \}$

**Semantic actions** indicate how to compute attributes

- attributes computed bottom-up, or in more general way

# Parser Generators:

## Attribute Grammar -> Parser

1) Embedded: parser combinators (Scala, Haskell)

They *are* code in some (functional) language

```
def ID : Parser = "x" | "y" | "z"
def expr : Parser = factor ~ (( "+" ~ factor | "-" ~ factor )
                             | epsilon)
def factor : Parser = term ~ (( "*" ~ term | "/" ~ term )
                              | epsilon)
def term : Parser = ( "(" ~ expr ~ ")" | ID | NUM )
```

*implicit conversion: string s to skip(s)*  
*concatenation*  
*<- often not really LL(1) but "try one by one", must put first non-empty, then epsilon*

implementation in Scala: use **overloading** and **implicit**s

2) Standalone tools: JavaCC, Yacc, ANTLR, CUP

- ***generate*** code in a conventional programming languages (e.g. Java)

# Example in CUP - LALR(1) (not LL(1) )

precedence left PLUS, MINUS;  
precedence left TIMES, DIVIDE, MOD; // priorities disambiguate  
precedence left UMINUS;

expr ::= expr PLUS expr // ambiguous grammar works here  
| expr MINUS expr  
| expr TIMES expr  
| expr DIVIDE expr  
| expr MOD expr  
| MINUS expr %prec UMINUS  
| LPAREN expr RPAREN  
| NUMBER ;

# Adding Java Actions to CUP Rules

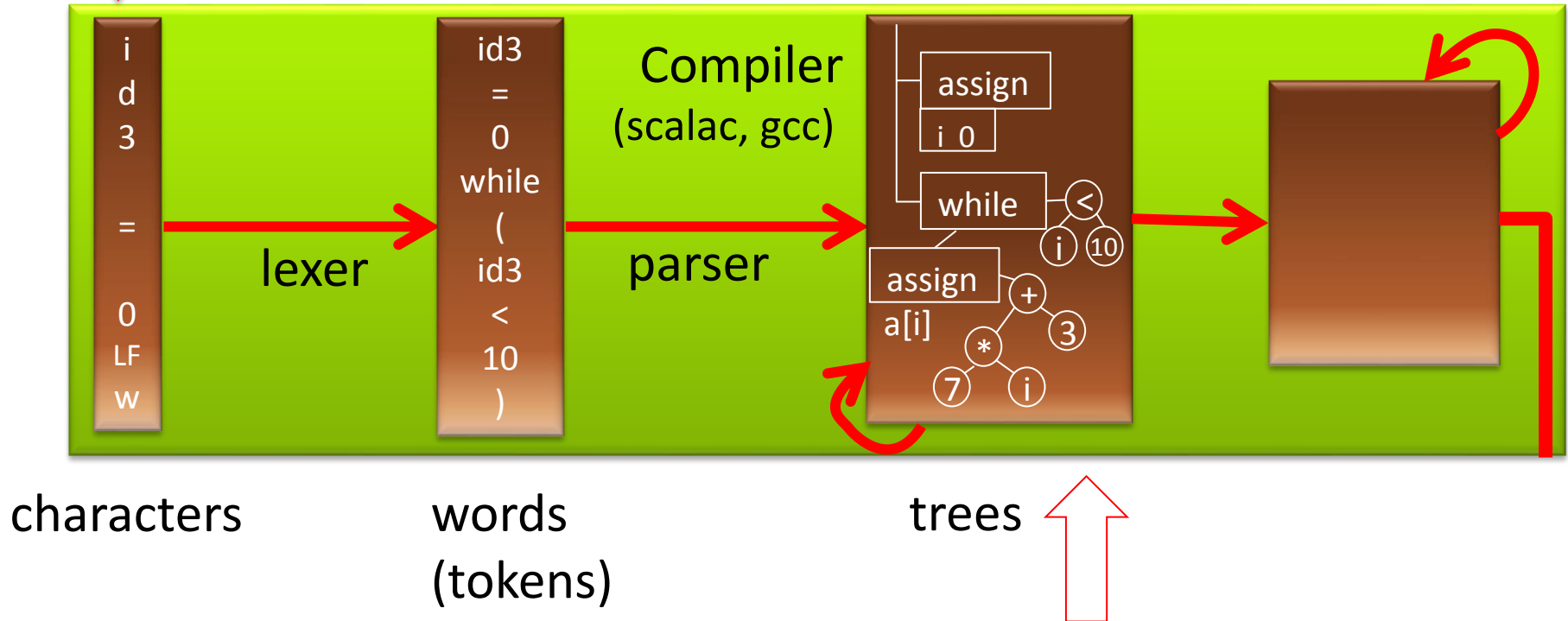
```
expr ::= expr:e1 PLUS expr:e2
      { : RESULT = new Integer(e1.intValue() + e2.intValue()); : }
      | expr:e1 MINUS expr:e2
      { : RESULT = new Integer(e1.intValue() - e2.intValue()); : }
      | expr:e1 TIMES expr:e2
      { : RESULT = new Integer(e1.intValue() * e2.intValue()); : }
      | expr:e1 DIVIDE expr:e2
      { : RESULT = new Integer(e1.intValue() / e2.intValue()); : }
      | expr:e1 MOD expr:e2
      { : RESULT = new Integer(e1.intValue() % e2.intValue()); : }
      | NUMBER:n { : RESULT = n; : }
      | MINUS expr:e
      { : RESULT = new Integer(0 - e.intValue()); : } %prec UMINUS
      | LPAREN expr:e RPAREN { : RESULT = e; : } ;
```

# Compiler

source code

```
Id3 = 0
while (id3 < 10) {
  println("",id3);
  id3 = id3 + 1 }

```

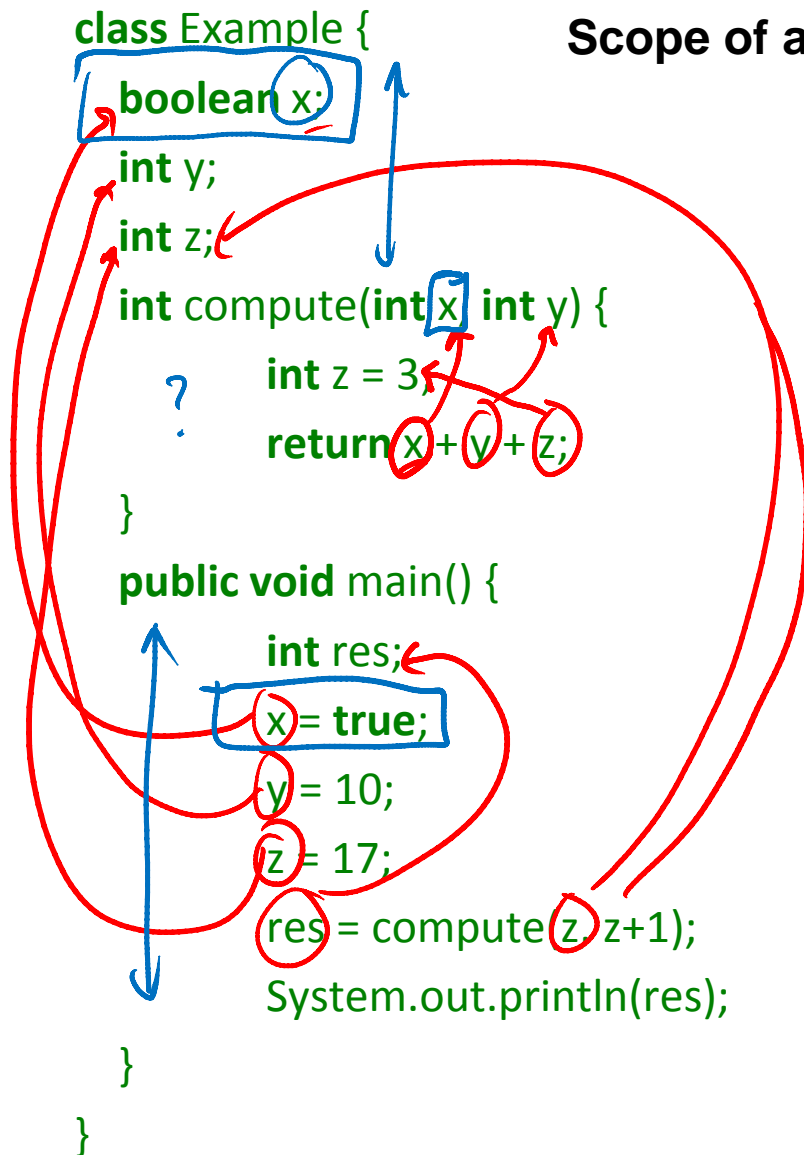


**Name Analysis**  
(step towards type checking)

# Name Analysis: Problems it Detects

- a class is defined more than once: **class A { ... } class B { ... } class A { ... }**
  - a variable is defined more than once: **int x; int y; int x;**
  - a class member is overloaded (forbidden in Tool, requires **override** keyword in Scala):  
**class A { int x; ... } class B extends A { int x; ... }**
  - a method is overloaded (forbidden in Tool, requires **override** keyword in Scala):  
**class A { int x; ... } class B extends A { int x; ... }**
  - a method argument is shadowed by a local variable declaration (forbidden in Java, Tool):  
**def (x:Int) { var x : Int; ... }**
  - two method arguments have the same name: **def (x:Int,y:Int,x:Int) { ... }**
  - a class name is used as a symbol (as parent class or type, for instance) but is not declared:  
**class A extends Objekt {}**
  - an identifier is used as a variable but is not declared:  
**def(amount:Int) { total = total + ammount }**
  - the inheritance graph has a cycle: **class A extends B {} class B extends C {} class C extends A**
- To make it efficient and clean to check for such errors, we associate mapping from each identifier to the symbol that the identifier represents.
- We use Map data structures to maintain this mapping (Map, what else?)
  - The rules that specify how declarations are used to construct such maps are given by ***scoping rules of the programming language.***

# Example: program result, symbols, scopes



**Scope of a variable** = part of program where it is visible

Draw an arrow from occurrence of each identifier to the point of its declaration.

For each declaration of identifier, identify where the identifier can be referred to (its scope).

Name analysis:

- compute those arrows
  - = maps, partial functions (math)
  - = environments (PL theory)
  - = symbol table (implementation)
- report some simple semantic errors

Usually introduce **symbols** for things denoted by identifiers.

Symbol tables map identifiers to symbols.

# Usual static scoping: What is the result?

```
class World {  
    int sum;  
    int value;  
    void add() {  
        sum = sum + value;  
        value = 0;  
    }  
    void main() {  
        sum = 0;  
        value = 10;  
        add();  
        if (sum % 3 == 1) {  
            int value;  
            value = 1;  
            add();  
            print("inner value = ", value); 1  
            print("sum = ", sum); 10  
        }  
        print("outer value = ", value); 0  
    }  
}
```

Identifier refers to the symbol that was declared closest to the place in program text (thus "static").

**We will assume static scoping** unless otherwise specified.

Cool property: we could always rename variables to avoid any shadowing (make all vars unique).



# Renaming Statically Scoped Program

```
class World {  
  int sum;  
  int value;  
  void add(int foo) {  
    sum = sum + value;  
    value = 0;  
  }  
  void main() {  
    sum = 0;  
    value = 10;  
    add();  
    if (sum % 3 == 1) {  
      int value1;  
      value1 = 1;  
      add(); // cannot change value1  
      print("inner value = ", value1); 1  
      print("sum = ", sum); 10  
    }  
    print("outer value = ", value); 0  
  }  
}
```

Identifier refers to the symbol that was declared closest to the place in program text (thus "static").

**We will assume static scoping** unless otherwise specified.

Cool property: we could always rename variables to avoid any shadowing (make all vars unique).

# Dynamic scoping: What is the result?

```
class World {  
  int sum;  
  int value;  
  void add() {  
    sum = sum + value;  
    value = 0;  
  }  
  void main() {  
    sum = 0;  
    value = 10;  
    add();  
    if (sum % 3 == 1) {  
      int value;  
      value = 1;  
      add();  
      print("inner value = ", value); 0  
      print("sum = ", sum); 11  
    }  
    print("outer value = ", value); 0  
  }  
}
```

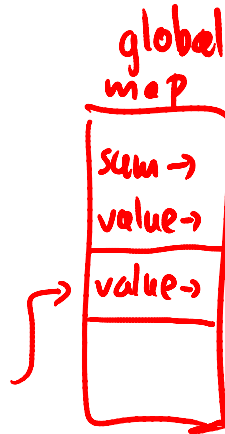
Symbol refers to the variable that was most **recently declared within program execution**.

Views variable declarations as executable statements that establish which symbol is considered to be the 'current one'. (Used in old LISP interpreters.)

Translation to normal code: access through a dynamic environment.

# Dynamic scoping translated using global map, working like stack

```
class World {  
  int sum;  
  int value;  
  void add() {  
    sum = sum + value;  
    value = 0;  
  }  
  void main() {  
    sum = 0;  
    value = 10;  
    add();  
    if (sum % 3 == 1) {  
      int value;  
      value = 1;  
      add();  
      print("inner value = ", value); 0  
      print("sum = ", sum); 11  
    }  
    print("outer value = ", value); 0  
  }  
}
```



```
class World {  
  pushNewDeclaration('sum');  
  pushNewDeclaration('value');  
  void add(int foo) {  
    update('sum, lookup('sum) + lookup('value));  
    update('value, 0);  
  }  
  void main() {  
    update('sum, 0);  
    update('value, 10);  
    add();  
    if (lookup('sum) % 3 == 1) {  
      → pushNewDeclaration('value);  
      update('value, 1);  
      add();  
      print("inner value = ", lookup('value));  
      print("sum = ", lookup('sum));  
      popDeclaration('value)  
    }  
    print("outer value = ", lookup('value));  
  }  
}
```

## How map changes with **static** scoping

Outer declaration  
**int value** is shadowed by  
inner declaration **string value**

Map becomes bigger as  
we enter more scopes,  
later becomes smaller again.  
**Imperatively:** need to make  
maps bigger, later smaller again.  
**Functionally:** immutable maps,  
keep old versions.

```
class World {  
    int sum; int value;  
    // value → int, sum → int  
    void add(int foo) {  
        // foo → int, value → int, sum → int  
        string z;  
        // z → string, foo → int, value → int, sum → int  
        sum = sum + value; value = 0;  
    }  
    // value → int, sum → int  
    void main(string bar) {  
        // bar → string, value → int, sum → int  
        int y;  
        // y → int, bar → string, value → int, sum → int  
        sum = 0;  
        value = 10;  
        add();  
        // y → int, bar → string, value → int, sum → int  
        if (sum % 3 == 1) {  
            string value;  
            // value → string, y → int, bar → string, sum → int  
            value = 1;  
            add();  
            print("inner value = ", value);  
            print("sum = ", sum); }  
        // y → int, bar → string, value → int, sum → int  
        print("outer value = ", value);  
    }  
}
```

# Abstract Trees of Simple Language with Arbitrarily Nested Blocks

program ::= **class World** { varDecl\* method\* }  
method ::= varDecl ( varDecl\* ) { thing\* **return** expr }  
varDecl ::= type ID  
type ::= **int** | **boolean** | **void**  
thing ::= varDecl | stmt  
stmt ::= expr | if | while | block  
if ::= **if** (expr) stmt **else** stmt  
while ::= **while** expr stmt  
block ::= { thing\* }  
expr ::= **ID** | expr + expr | expr <= expr | assign | call | condExpr  
assign ::= **ID** = expr  
condExpr ::= expr ? expr : expr  
call ::= **ID** ( expr\* )

# NOTATION FOR MAPS

Mathematical notion of map  $f : A \rightarrow B$  is a partial function, that is, a function from a subset of  $A$  to  $B$ .

- $f \subseteq A \times B$
- $\forall x. \forall y_1. \forall y_2. (x, y_1) \in f \wedge (x, y_2) \in f \rightarrow y_1 = y_2$

We define  $\text{dom}(f) = \{x \mid \exists y. (x, y) \in f\}$

Key operation is function update

$$f[k := v] = \{(x, y) \mid (x = k \wedge y = v) \vee (x \neq k \wedge (x, y) \in f)\}$$

If the value was defined before, now we redefine it.

$$f[k := v] = f \oplus \{k \mapsto v\}$$

A generalization of update is overriding one map by another:

$$f \oplus g = \{(x, y) \mid (x, y) \in g \vee (x \notin \text{dom}(g) \wedge (x, y) \in f)\}$$

Sometimes we denote map  $\{(k_1, v_1), \dots, (k_n, v_n)\}$  by  $\{k_1 \mapsto v_1, \dots, k_n \mapsto v_n\}$

Is  $f \oplus g = g \oplus f$ ?

- $\{x \mapsto b, z \mapsto a\} \oplus \{x \mapsto c\} = \{x \mapsto c, z \mapsto a\}$
- $\{x \mapsto c\} \oplus \{x \mapsto b, z \mapsto a\} = \{x \mapsto b, z \mapsto a\}$

# Rules to check that each variable used in a statement is declared

$\Gamma = \{ (x_1, T_1), \dots (x_n, T_n) \}$  - environment (symbol table)

$x_1 \mapsto T_1$

identifier      symbol (type, ...)

$\Gamma \vdash \underline{e}$

e uses only variables declared in  $\Gamma$

$\Gamma = \{ (x, \text{int}), (y, \text{string}), (z, \text{int}) \}$

then:

$\Gamma \vdash x + (z + 1)$

$\Gamma \vdash x = x + 1$

# Rules for Checking Variable Use

$$\frac{\Gamma \vdash e_1, \Gamma \vdash e_2}{\Gamma \vdash e_1 + e_2} \quad \leftarrow$$

$$\frac{\Gamma \vdash e_1 \quad \Gamma \vdash e_2}{\Gamma \vdash e_1 * e_2}$$

$$\frac{\begin{array}{c} x \in \text{dom}(\Gamma) \\ (x, -) \in \Gamma \end{array} \quad \Gamma \vdash e}{\Gamma \vdash \underbrace{x = e}}$$

$$\begin{array}{c} \downarrow \\ \exists \underline{t}. (x, t) \in \Gamma \\ (x, -) \in \Gamma \end{array}$$

$$\frac{\Gamma \vdash s \quad \Gamma \vdash \bar{s}}{\Gamma \vdash s; \bar{s}}$$

s - statement

$\bar{s}$  - statement  
sequence



# Local Block Declarations Change $\Gamma$

$$\Gamma \oplus \{(x, \text{int})\}$$

$$\frac{\Gamma[x := \text{int}] \vdash \bar{5}}{\Gamma \vdash (\text{int } x); \bar{5}}$$

$$\rightarrow \frac{\Gamma \vdash e, \quad \Gamma \vdash s_1, \quad \Gamma \vdash s_2}{\Gamma \vdash \text{if } (e) s_1 \text{ else } s_2}$$

$$\Gamma = \{(z, \text{int})\}$$

$$\Gamma[x := \text{int}] = \{(z, \text{int}), (x, \text{int})\}$$

$$\Gamma \rightarrow \left[ \begin{array}{l} \text{if } (x > 0) \{ \\ \quad \rightarrow \text{int } y; \quad y = x + 1 \\ \} \text{ else } \{ \text{int } z; \quad z = x + 1 \} \end{array} \right.$$

$$\frac{\Gamma[x := \text{int}] \vdash x = z + 2}{\Gamma \vdash \text{int } x; x = z + 2}$$

# Method Parameters are Similar

$$\frac{\Gamma \oplus \overbrace{\{(x_1, T_1), \dots, (x_n, T_n)\}} \vdash \underline{\bar{s}}}{\Gamma \vdash \underline{Tm(T_1 x_1, \dots, T_n x_n) \{ \bar{s} \}}}$$

$\Gamma = \{(sum, int), (value, int)\}$

$\Gamma \oplus \{(foo, int)\} \vdash sum = sum + foo;$

$\Gamma \vdash \text{void add(int foo) {  
    sum = sum + foo;  
}}$

```
class World {  
    int sum;  
    int value;  
    void add(int foo) {  
        sum = sum + foo;  
    }  
}
```