CS 536

Introduction to Programming Languages and Compilers

Charles N. Fischer

Lecture 9

- Midterm Exam #1:
  Monday, November 12,
  5:30 – 7:30 PM
  Covers LL(1) parsing.

Reading Assignment

Read Sections 6.1 to 6.5.1 of Crafting a Compiler.

How does JavaCup Work?

The main limitation of LL(1) parsing is that it must predict the correct production to use when it first starts to match the production's righthand side.

An improvement to this approach is the *LALR(1) parsing method* that is used in JavaCUP (and Yacc and Bison too).

The LALR(1) parser is bottom- up in approach. It tracks the portion of a righthand side already matched as tokens are scanned. It may not know immediately which is the correct production to choose, so it tracks *sets* of possible matching productions.

Configurations

We'll use the notation

X → A B • C D

to represent the fact that we are trying to match the production **X → A B C D** with **A** and **B** matched so far.

A production with a "•" somewhere in its righthand side is called a *configuration*.

Our goal is to reach a configuration with the "dot" at the extreme right:

X → A B C D •

This indicates that an entire production has just been matched.

Since we may not know which production will eventually be fully matched, we may need to track a *configuration set*. A configuration set is sometimes called a *state*.

When we predict a production, we place the "dot" at the beginning of a production:

X → • A B C D

This indicates that the production may possibly be matched, but no symbols have actually yet been matched.

We may predict a λ- production:

X → λ •

When a λ- production is predicted, it is immediately matched, since λ can be matched at any time.

Starting the Parse

At the start of the parse, we know some production with the start symbol must be used initially. We don't yet know which one, so we predict them *all*:

S → • A B C D

S → • e F g

S → • h I

...

Closure

When we encounter a configuration with the dot to the left of a non- terminal, we know we need to try to match that non-terminal.

Thus in

X → • A B C D

we need to match some production with A as its left hand side.

Which production?

We don't know, so we predict *all* possibilities:

A → • P Q R

A → • s T

...

The newly added configurations may predict other non- terminals, forcing additional productions to be included. We continue this process until no additional configurations can be added.

This process is called *closure* (of the configuration set).

Here is the closure algorithm:

```
ConfigSet Closure(ConfigSet C)
  { repeat
      if (X → a •B d is in C &&
              B is a non-terminal)
        Add all configurations of
           the form   B → •g to C)
  until (no more configurations
                can be added);
  return C;
}
```

Example of Closure

Assume we have the following grammar:

S → A b

A → C D

C → D

C → c

D → d

To compute Closure(**S → • A b**) we first include all productions that rewrite A:

A → •C D

Now **C** productions are included:

C → •D

C → •c

Finally, the D production is added:

D → • d

The complete configuration set is:

S → • A b

A → • C D

C → • D C

C → • c D

D → • d

This set tells us that if we want to match an **A**, we will need to match a **C**, and this is done by matching a **c** or **d** token.

Shift Operations

When we match a symbol (a terminal or non- terminal), we *shift* the "dot" past the symbol just matched. Configurations that don't have a dot to the left of the matched symbol are *deleted* (since they didn't correctly anticipate the matched symbol).

The **GoTo** function computes an updated configuration set after a symbol is shifted:

```
ConfigSet GoTo(ConfigSet C,Symbol X)
  { B=ϕ;
  for each configuration f in C {
      if (f is of the form A→ α•Xδ)
        Add A → α X •δ to B;
  }
  return Closure(B);
}
```

For example, if the set is

S → · A b

A → · C D

C → · D

C → · c

D → · d

and **x** is **C**, then **GoTo** returns

A → C · D

D → · d

Reduce Actions

When the dot in a configuration reaches the rightmost position, we have matched an entire righthand side. We are ready to replace the righthand side symbols with the lefthand side of the production. The lefthand side symbol can now be considered matched.

If a configuration set can shift a token and also reduce a production, we have a potential *shift/reduce error*.

If we can reduce more than one production, we have a potential *reduce/reduce error*.

How do we decide whether to do a shift or reduce? How do we choose among more than one reduction?

We examine the *next token* to see if it is consistent with the potential reduce actions.

The simplest way to do this is to use Follow sets, as we did in LL(1) parsing.

If we have a configuration

$$\mathbf{A} \rightarrow \alpha \cdot$$

we will reduce this production *only if* the current token, **CT**, is in Follow(**A**).

This makes sense since if we reduce α to **A**, we can't correctly match **CT** if **CT** can't follow **A**.

Shift/Reduce and Reduce/ Reduce Errors

If we have a parse state that contains the configurations

A $\rightarrow$ α $\bullet$

B $\rightarrow$ β $\bullet$ **a** γ

and **a** in Follow(**A**) then there is an *unresolvable* shift/reduce conflict. This grammar can't be parsed.

Similarly, if we have a parse state that contains the configurations

A $\rightarrow$ α $\bullet$

B $\rightarrow$ β $\bullet$

and Follow(**A**) $\cap$ Follow(**B**) $\neq \phi$, then the parser has an unresolvable reduce/reduce conflict. This grammar can't be parsed.

Building Parse States

All the manipulations needed to build and complete configuration sets suggest that parsing may be slow—configuration sets need to be updated after each token is matched.

Fortunately, all the configuration sets we ever will need can be computed and tabled *in advance*, when a tool like Java Cup builds a parser.

The idea is simple. We first compute an initial parse state, s_0, that corresponds to predicting productions that expand the start symbol. We then just compute successor states for each token that might be scanned. A complete set of states can be computed. For typical

programming language grammars, only a few hundred states are needed.

Here is the algorithm that builds a complete set of parse states for a grammar:

```
StateSet BuildStates(){
 Let s0=Closure({S→ •α, S→ •β, ...});
 C={s0};
 while (not all states in C are marked)
  { Choose any unmarked state, s, in C
  Mark s;
  For each X in
      terminals U nonterminals
    { if (GoTo(s,X) is not in C)
       Add GoTo(s,X) to C;
  }
 }
 return C;
}
```

Configuration Sets for CSX-Lite

State	Cofiguration Set
s_0	Prog → •{ Stmts } Eof
s_1	Prog → { • Stmts } Eof Stmts → •Stmt Stmts Stmts → λ • Stmt → • id = Expr ; Stmt → • if (Expr) Stmt
s_2	Prog → { Stmts •} Eof
s_3	Stmts → Stmt • Stmts Stmts → •Stmt Stmts Stmts → λ • Stmt → • id = Expr ; Stmt → • if (Expr) Stmt
s_4	Stmt → id • = Expr ;
s_5	Stmt → if • (Expr) Stmt

State	Cofiguration Set
s_6	Prog → { Stmts } • Eof
s_7	Stmts → Stmt Stmts •
s_8	Stmt → id = • Expr ; Expr → • Expr + id Expr → • Expr - id Expr → • id
s_9	Stmt → if (• Expr) Stmt Expr → • Expr + id Expr → • Expr - id Expr → • id
s_{10}	Prog → { Stmts } Eof •
s_{11}	Stmt → id = Expr • ; Expr → Expr • + id Expr → Expr • - id
s_{12}	Expr → id •
s_{13}	Stmt → if (Expr •) Stmt Expr → Expr • + id Expr → Expr • - id

State	Cofiguration Set
S_{14}	Stmt → id = Expr ; •
S_{15}	Expr → Expr + • id
S_{16}	Expr → Expr - • id
S_{17}	Stmt → if (Expr) • Stmt Stmt → • id = Expr ; Stmt → • if (Expr) Stmt
S_{18}	Expr → Expr + id •
S_{19}	Expr → Expr - id •
S_{20}	Stmt → if (Expr) Stmt •

Parser Action Table

We will table possible parser actions based on the current state (configuration set) and token.

Given configuration set C and input token T four actions are possible:

- Reduce i: The i- th production has been matched.
- Shift: Match the current token.
- Accept: Parse is correct and complete.
- Error: A syntax error has been discovered.

We will let A[C][T] represent the possible parser actions given configuration set C and input token T.

A[C][T] =

{Reduce i | i- th production is **A**→ α
and **A** → α • is in C
and T in Follow(A) }
U (If (**B** → β • **T** γ is in C)
{Shift} else ϕ)

This rule simply collects all the actions that a parser might do given C and T.

But we want parser actions to be unique so we require that the parser action always be *unique* for any C and T.

If the parser action isn't unique, then we have a shift/reduce error or reduce/reduce error. The grammar is then rejected as unparsable.

If parser actions are always unique then we will consider a shift of EOF to be an accept action.

An empty (or undefined) action for C and T will signify that token T is illegal given configuration set C.
A syntax error will be signaled.

LALR Parser Driver

Given the GoTo and parser action tables, a Shift/Reduce (LALR) parser is fairly simple:

```
void LALRDriver()
  { Push(S0);
  while(true){
   //Let S = Top state on parse stack
   //Let CT = current token to match
     switch (A[S][CT]) {
      case error:
         SyntaxError(CT);return;
      case accept:
         return;
      case shift:
         push(GoTo[S][CT]);
         CT= Scanner();
         break;
      case reduce i:
         //Let prod i = A→Y1...Ym
         pop m states;
         //Let S' = new top state
         push(GoTo[S'][A]);
         break;
} } }
```

Action Table for CSX-Lite

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
{	S																				
}		R3	S	R3				R2							R4						R5
if		S		S											R4			S			R5
(						S															
)													R8	S					R6	R7	
id		S		S					S	S					R4	S	S	S			
=					S																
+												S	R8	S					R6	R7	
-												S	R8	S					R6	R7	
;												S	R8						R6	R7	R5
eof							A														

GoTo Table for CSX-Lite

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
{	1																				
}			6																		
if		5		5														5			
(						9															
)														17							
id		4		4					12	12						18	19	4			
=					8																
+												15		15							
-												16		16							
;												14									
eof							10														
stmts		2		7																	
stmt		3		3																	
expr									11	13											

Example of LALR(1) Parsing

We'll again parse

{ a = b + c; } Eof

We start by pushing state 0 on the parse stack.

Parse Stack	Top State	Action	Remaining Input
0	Prog → •{ Stmts } Eof	Shift	{ a = b + c; } Eof
1 0	Prog → { • Stmts } Eof Stmts → • Stmt Stmts Stmts → λ • Stmt → • id = Expr ; Stmt → • if (Expr)	Shift	a = b + c; } Eof
4 1 0	Stmt → id • = Expr ;		= b + c; } Eof
8 4 1 0	Stmt → id = • Expr ; Expr → • Expr + id Expr → • Expr - id Expr → • id	Shift	b + c; } Eof

Parse Stack	Top State	Action	Remaining Input
12 8 4 1 0	Expr → id •	Reduce 8	+ c; } Eof
11 8 4 1 0	Stmt → id = Expr • ; Expr → Expr • + id Expr → Expr • - id	Shift	+ c; } Eof
15 11 8 4 1 0	Expr → Expr + • id	Shift	c; } Eof

Parse Stack	Top State	Action	Remaining Input
18 15 11 8 4 1 0	Expr → Expr + id •	Reduce 6	; } Eof
11 8 4 1 0	Stmt → id = Expr • ; Expr → Expr • + id Expr → Expr • - id	Shift	; } Eof
14 11 8 4 1 0	Stmt → id = Expr ; •	Reduce 4	} Eof

Parse Stack	Top State	Action	Remaining Input
3 1 0	Stmts → Stmt • Stmts Stmts → •Stmt Stmts Stmts → λ • Stmt → • id = Expr ; Stmt → • if (Expr) Stmt	Reduce 3	} Eof
7 3 1 0	Stmts → Stmt Stmts •	Reduce 2	} Eof
2 1 0	Prog → { Stmts •} Eof	Shift	} Eof
6 2 1 0	Prog → { Stmts } •Eof	Accept	Eof

Error Detection in LALR Parsers

In bottom- up, LALR parsers syntax errors are discovered when a blank (error) entry is fetched from the parser action table.

Let's again trace how the following illegal CSX- lite program is parsed:

```
{ b + c = a; } Eof
```

Parse Stack	Top State	Action	Remaining Input
0	Prog → •{ Stmts } Eof	Shift	{ b + c = a; } Eof

Parse Stack	Top State	Action	Remaining Input
1 0	Prog → { • Stmts } Eof Stmts → • Stmt Stmts Stmts → λ • Stmt → • id = Expr ; Stmt → • if (Expr)	Shift	`b + c = a; } Eof`
4 1 0	Stmt → id • = Expr ;	Error (blank)	`+ c = a; } Eof`

LALR is More Powerful

Essentially all LL(1) grammars are LALR(1) plus many more. Grammar constructs that confuse LL(1) are readily handled.

- Common prefixes are no problem. Since sets of configurations are tracked, more than one prefix can be followed. For example, in

  Stmt → id = Expr ;

  Stmt → id (Args) ;

  after we match an id we have

  Stmt → id · = Expr ;

  Stmt → id · (Args) ;

  The next token will tell us which production to use.

- Left recursion is also not a problem. Since sets of configurations are tracked, we can follow a left- recursive production *and* all others it might use. For example, in

  Expr → · Expr + id
  Expr → · id

  we can first match an **id**:

  Expr → id ·

  Then the **Expr** is recognized:

  Expr → Expr · + id

  The left- recursion is handled!

- But ambiguity will still block construction of an LALR parser. Some shift/reduce or reduce/ reduce conflict must appear. (Since two or more distinct parses are possible for some input). Consider our original productions for if- then and if- then- else statements:

Stmt → if (Expr) Stmt •

Stmt → if (Expr) Stmt • else Stmt

Since **else** can follow **Stmt**, we have an unresolvable shift/reduce conflict.

Grammar Engineering

Though LALR grammars are very general and inclusive, sometimes a reasonable set of productions is rejected due to shift/reduce or reduce/reduce conflicts.

In such cases, the grammar may need to be "engineered" to allow the parser to operate.

A good example of this is the definition of **MemberDecls** in CSX. A straightforward definition is

MemberDecls → FieldDecls MethodDecls
FieldDecls → FieldDecl FieldDecls
FieldDecls → λ
MethodDecls → MethodDecl MethodDecls
MethodDecls → λ
FieldDecl → int id ;
MethodDecl → int id ( ) ; Body

When we predict **MemberDecls** we get:

MemberDecls → • **FieldDecls MethodDecls**
FieldDecls → • **FieldDecl FieldDecls**
FieldDecls → λ•
FieldDecl → • **int id ;**

Now **int** follows **FieldDecls** since

MethodDecls $\Rightarrow^{+}$ **int …**

Thus an unresolvable shift/reduce conflict exists.

The problem is that **int** is derivable from both **FieldDecls** and **MethodDecls**, so when we see an **int**, we can't tell which way to parse it (and **FieldDecls** → λ requires we make an immediate decision!).

If we rewrite the grammar so that we can *delay* deciding from where the int was generated, a valid LALR parser can be built:

MemberDecls → FieldDecl MemberDecls
MemberDecls → MethodDecls
MethodDecls → MethodDecl MethodDecls
MethodDecls → λ
FieldDecl → int id ;
MethodDecl → int id ( ) ; Body

When **MemberDecls** is predicted we have

MemberDecls → • FieldDecl MemberDecls
MemberDecls → • MethodDecls
MethodDecls → •MethodDecl MethodDecls
MethodDecls → λ •
FieldDecl → • int id ;
MethodDecl → • int id ( ) ; Body

Now **Follow(MethodDecls)** = **Follow(MemberDecls)** = "}", so we have no shift/reduce conflict. After **int id** is matched, the next token (a ";" or a "(") will tell us whether a **FieldDecl** or a **MethodDecl** is being matched.

Properties of LL and LALR Parsers

- Each prediction or reduce action is *guaranteed* correct. Hence the entire parse (built from LL predictions or LALR reductions) must be correct.

  This follows from the fact that LL parsers allow only one valid prediction per step. Similarly, an LALR parser never skips a reduction if it is consistent with the current token (and *all* possible reductions are tracked).

- LL and LALR parsers detect an syntax error *as soon as* the first invalid token is seen.

  Neither parser can match an invalid program prefix. If a token is matched it *must be* part of a valid program prefix. In fact, the prediction made or the stacked configuration sets *show* a possible derivation of the token accepted so far.

- All LL and LALR grammars are *unambiguous*.

  LL predictions are always unique and LALR shift/reduce or reduce/reduce conflicts are disallowed. Hence only one valid derivation of any token sequence is possible.

- All LL and LALR parsers require only *linear time and space* (in terms of the number of tokens parsed).

  The parsers do only fixed work per node of the concrete parse tree, and the size of this tree is linear in terms of the number of leaves in it (even with λ- productions included!).

Reading Assignment

Read Chapter 8 of Crafting a Compiler.

Symbol Tables in CSX

CSX is designed to make symbol tables easy to create and use.

There are three places where a new scope is opened:

- In the class that represents the program text. The scope is opened as soon as we begin processing the **`classNode`** (that roots the entire program). The scope stays open until the entire class (the whole program) is processed.
- When a **`methodDeclNode`** is processed. The name of the method is entered in the top- level (global) symbol table. Declarations of parameters and locals are placed in the method's symbol table. A method's symbol table is closed after all the statements in its body are type checked.

- When a **`blockNode`** is processed. Locals are placed in the block's symbol table. A block's symbol table is closed after all the statements in its body are type checked.

CSX Limits
Forward References

Except for method references, we can do type-checking in *one pass* over the AST. As declarations are processed, their identifiers are added to the current (innermost) symbol table. When a use of an identifier occurs, we do an ordinary block- structured lookup, always using the innermost declaration found. Hence in

```
int i = j;
int j = i;
```

the first declaration initializes **`i`** to the nearest non- local definition of **`j`**.

The second declaration initializes **`j`** to the current (local) definition of **`i`**.

Forward References to Methods Require Two Passes

Since forward references to methods are allowed, we process method declarations in *two passes*.

First we walk the methodDecls AST to establish symbol tables entries for all method declarations. No calls (lookups) are handled in this passes.

On a second pass, all calls are processed, using the symbol table entries built on the first pass.

Forward references make type checking a bit trickier, as we may reference a declaration not yet fully processed.

In Java, forward references to fields within a class are allowed.

Thus in

```
class Duh {
 int i = j;
 int j = i;
}
```

a Java compiler must recognize that the initialization of **`i`** is to the **`j`** field and that the **`j`** declaration is incomplete (Java forbids uninitialized fields or variables).

Forward references allow methods to be mutually recursive. That is, we can let method **`a`** call **`b`**, while **`b`** calls **`a`**.

Incomplete Declarations

Some languages, like C++, allow *incomplete* declarations.

First, part of a declaration (usually the header of a procedure or method) is presented.

Later, the declaration is completed. In C++:

```
class C
  { int I;

   public:
     int f();
  };
  int C::f(){return i+1;}
```

Incomplete declarations solve potential forward reference problems, as you can declare method headers first, and bodies that use the headers later.

Headers support abstraction and separate compilation too.

In C and C++, it is common to use a **`#include`** statement to add the headers (but not bodies) of external or library routines you wish to use.

C++ also allows you to declare a class by giving its fields and method headers first, with the bodies of the methods declared later. This is good for users of the class, who don't always want to see implementation details.

Classes, Structs and Records

The fields and methods declared within a class, struct or record are stored within a individual symbol table allocated for its declarations.

Member names must be unique within the class, record or struct, but may clash with other visible declarations. This is allowed because member names are qualified by the object they occur in.

Hence the reference **`x.a`** means look up **`x`**, using normal scoping rules. Object **`x`** should have a type that includes local fields. The type of **`x`** will include a pointer to the symbol table containing the field declarations. Field **`a`** is looked up in that symbol table.

Chains of field references are no problem.

For example, in Java

`System.out.println`

is commonly used.

`System` is looked up and found to be a class in one of the standard Java packages (**`java.lang`**). Class **`System`** has a static member **`out`** (of type **`PrintStream`**) and **`PrintStream`** has a member **`println`**.

Internal and External Field Access

Within a class, members may be accessed without qualification. Thus in

```
class C {
   static int i;
   void subr() {
      int j = i;
   }
}
```

field `i` is accessed like an ordinary non- local variable.

To implement this, we can treat member declarations like an ordinary scope in a block-structured symbol table.

When the class definition ends, its symbol table is popped and members are referenced through the symbol table entry for the class name.

This means a simple reference to **`i`** will no longer work, but **`C.i`** will be valid.

In languages like C++ that allow incomplete declarations, symbol table references need extra care. In

```
class C
 { int i;
 public:
 int f();
};
int C::f(){return i+1;}
```

when the definition of **`f()`** is completed, we must restore **`C`**'s field definitions as a containing scope so that the reference to **`i`** in **`i+1`** is properly compiled.

Public and Private Access

C++ and Java (and most other object- oriented languages) allow members of a class to be marked **`public`** or **`private`**.

Within a class the distinction is ignored; all members may be accessed.

Outside of the class, when a qualified access like **`C.i`** is required, only **`public`** members can be accessed.

This means lookup of class members is a two- step process. First the member name is looked up in the symbol table of the class. Then, the **`public`**/**`private`** qualifier is checked. Access to **`private`** members from outside the class generates an error message.

C++ and Java also provide a **`protected`** qualifier that allows access from subclasses of the class containing the member definition.

When a subclass is defined, it "inherits" the member definitions of its ancestor classes. Local definitions may hide inherited definitions. Moreover, inherited member definitions must be **`public`** or **`protected`**; **`private`** definitions may not be directly accessed (though they are still inherited and may be indirectly accessed through other inherited definitions).

Java also allows "blank" access qualifiers which allow **`public`** access by all classes within a package (a collection of classes).

Packages and Imports

Java allows packages which group class and interface definitions into named units.

A package requires a symbol table to access members. Thus a reference

`java.util.Vector`

locates the package **`java.util`** (typically using a **`CLASSPATH`**) and looks up **`Vector`** within it.

Java supports **`import`** statements that modify symbol table lookup rules.

A single class import, like

`import java.util.Vector;`

brings the name **`Vector`** into the current symbol table (unless a

definition of **`Vector`** is already present).

An “import on demand” like

```
import java.util.*;
```

will lookup identifiers in the named packages after explicit user declarations have been checked.

Classfiles and Object Files

Class files (".class" files, produced by Java compilers) and object files (".o" files, produced by C and C++ compilers) contain internal symbol tables.

When a field or method of a Java class is accessed, the JVM uses the classfile's internal symbol table to access the symbol's value and verify that type rules are respected.

When a C or C++ object file is linked, the object file's internal symbol table is used to determine what external names are referenced, and what internally defined names will be exported.

C, C++ and Java all allow users to request that a more complete symbol table be generated for debugging purposes. This makes internal names (like local variable) visible so that a debugger can display source level information while debugging.

Overloading

A number of programming languages, including CSX, Java and C++, allow method and subprogram names to be *overloaded*.

This means several methods or subprograms may share the same name, as long as they differ in the number or types of parameters they accept. For example,

```
class C
   {int x;
   public static int sum(int v1,
                         int v2) {
      return v1 + v2;
   }
   public int sum(int v3) {
      return x + v3;
   }
}
```

For overloaded identifiers the symbol table must return a *list* of valid definitions of the identifier. Semantic analysis (type checking) then decides which definition to use.

In the above example, while checking

```
(new C()).sum(10);
```

both definitions of **`sum`** are returned when it is looked up. Since one argument is provided, the definition that uses one parameter is selected and checked.

A few languages (like Ada) allow overloading to be disambiguated on the basis of a method's result type. Algorithms that do this analysis are known, but are fairly complex.

Overloaded Operators

A few languages, like C++, allow operators to be overloaded.

This means users may add new definitions for existing operators, though they may not create new operators or alter existing precedence and associativity rules.

(Such changes would force changes to the scanner or parser.)

For example,

```
Class complex{
  float re, im;
  complex operator+(complex d)
    { complex ans;
      ans.re = d.re+re;
      ans.im = d.im+im;
      return ans;
  } }
  complex c,d; c=c+d;
```

During type checking of an operator, all visible definitions of the operator (including predefined definitions) are gathered and examined.

Only one definition should successfully pass type checks.

Thus in the above example, there may be many definitions of **+**, but only one is defined to take **`complex`** operands.

Contextual Resolution

Overloading allows multiple definitions of the same kind of object (method, procedure or operator) to co- exist.

Programming languages also sometimes allow reuse of the same name in defining different kinds of objects. Resolution is by context of use.

For example, in Java, a class name may be used for *both* the class and its constructor. Hence we see

```
C cvar = new C(10);
```

In Pascal, the name of a function is also used for its return value.

Java allows rather extensive reuse of an identifier, with the same identifier potentially denoting a class (type), a class constructor, a

package name, a method and a field.

For example,

```
Class C{
   double v;
  C(double f) {v=f;}
 }
 class D {
   int C;
   double C() {return 1.0;}
  C cval =  new C(C+C());
}
```

At type- checking time we examine all potential definitions and use that definition that is consistent with the context of use. Hence new `C()` must be a constructor, `+C()` must be a function call, etc.

Allowing multiple definitions to co- exist certainly makes type checking more complicated than in other languages.

Whether such reuse benefits programmers is unclear; it certainly violates Java's "keep it simple" philosophy.

In CSX we allow overloading of methods (same name, different parameter combinations).

CSX also allows a label to use the same name as any other identifier.