Matching Logic

Grigore Rosu
University of Illinois at Urbana-Champaign

Joint work with Andrei Stefanescu and Chucky Ellison. Started with Wolfram Schulte at Microsoft Research in 2009

Question

… could it be that, after 40 years of program verification, we still lack the right semantically grounded program verification foundation?

Hoare logic

$$\{\pi_{\text{pre}}\}\ \texttt{code}\ \{\pi_{\text{post}}\}$$

Current State-of-the-Art in Program Analysis and Verification

Consider some programming language, L

- Formal semantics of L
  - Typically skipped: considered expensive and useless
- Model checkers for L
  - Based on some adhoc encodings of L
- Program verifiers for L
  - Based on some other adhoc encodings of L
- Runtime verifiers for L
  - Based on yet another adhoc encodings of L
- …

Example of C Program

- What should the following program evaluate to?

```
int main(void) {
  int x = 0;
  return (x = 1) + (x = 2);
}
```

- According to C “standard”, it is **undefined**
- GCC4, MSVC: it returns **4**
  GCC3, ICC, Clang: it returns **3**
  Both Frama-C and Havoc "prove" it returns **4**

A Formal Semantics Manifesto

- Programming languages must have formal semantics! (period)
- Informal manuals are not sufficient
  - Manuals typically have a formal syntax of the language (in an appendix)
  - Why not a formal semantics as well?

Motivation and Goal

- We are facing a semantic chaos
  - Operational, denotational, axiomatic, etc.
  - Problematic when dealing with large languages
- Why so many semantic styles?
  - Since none of them is ideal, they have limitations

- We want a powerful, unified foundation for programming language semantics and verification
  - One semantics to serve all the purposes!

Minimal Requirements for an Ideal Language Semantic Framework

- Should be expressive
  - Substitution or environment-based definitions, abrupt control changes (callcc), concurrency, etc.
- Should be executable
  - So we can test it and use it in tools (symb. exec.)
- Should be modular (and thus scale)
  - So each feature is defined once and for all
- Should serve as a program logic
  - So we can also prove programs correct with it

Current Semantic Approaches

- Structural Operational Semantics
  - Executable; not very modular; not appropriate for verification; only interleaving semantics
- Denotational Semantics
  - Reasonable trade-offs; not very executable (Norish’s C semantics is not executable, and factorial(5) crushes Papaspyrou’s C semantics); not very good for verification; bad for concurrency; expert knowledge
- Axiomatic Semantics (Floyd/Hoare logic)
  - Good for verification (sort of); not executable; not very expressive (extensions typically needed)
- Etc.

Towards a Better Semantic Approach

Starting Point: Rewriting Logic

Meseguer (late 80s, early 90s)

- Expressive
  - Any logic can be represented in RL (it is reflective)
- Executable
  - Quite efficiently; Maude often outperforms SML
- Modular
  - Allows rules to only “match” what they need
- Can potentially serve as a program logic
  - Admits initial model semantics, so it is amenable for inductive or fixed-point proofs

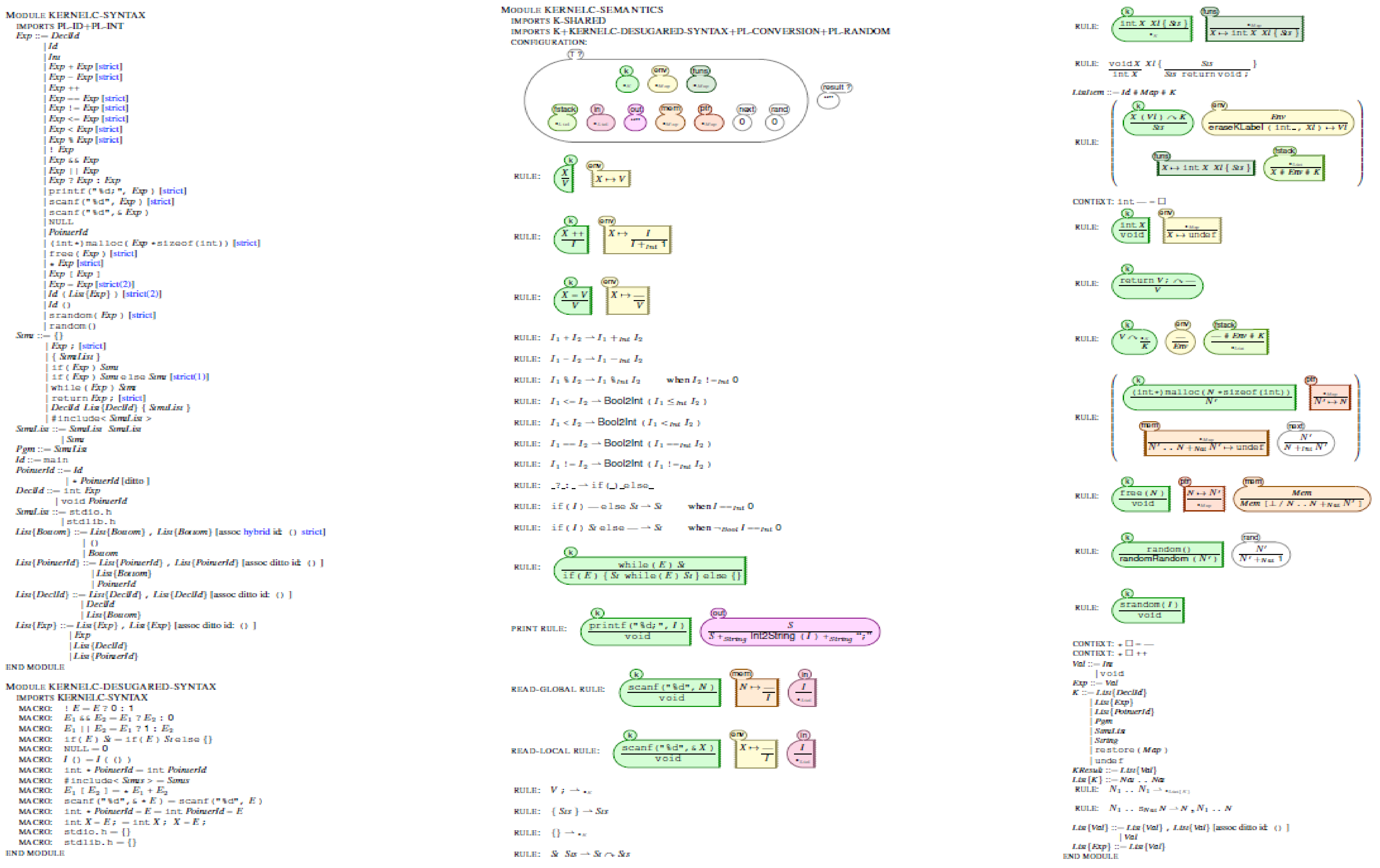
Rewriting Logic Semantics Project

- Project started jointly with Meseguer in 2003-4
- Idea: Define the semantics of a programming language as a rewrite theory (set of rules)
- Showed that most executable semantics approaches can be framed as rewrite logic semantics (Modular/SmallStep/BigStep SOS, evaluation contexts, continuation-based, etc.)
  - But they still had their inherent limitations
- Appropriate techniques/methodologies needed

The K Framework

- A tool-supported rewrite-based framework for defining programming language semantics
- Inspired from rewriting logic
- Used regularly in teaching undergraduate courses
- Ideas:
  - Represent program configurations as a nested structure of cells (like in the CHAM)
  - Flatten syntax into special computational structures (like in refocusing for evaluation contexts)
  - Define the semantics of each language construct by rules (a small number, typically 1 or 2)

Complete K Definition of KernelC



```
MODULE KERNELC-SYNTAX
  IMPORTS PL-ID+PL-INT
  Exp ::= DeclId
        | Id
        | Int
        | Exp + Exp [strict]
        | Exp - Exp [strict]
        | Exp ++
        | Exp == Exp [strict]
        | Exp != Exp [strict]
        | Exp <= Exp [strict]
        | Exp < Exp [strict]
        | Exp % Exp [strict]
        | ! Exp
        | Exp && Exp
        | Exp || Exp
        | Exp ? Exp : Exp
        | printf("%d;", Exp ) [strict]
        | scanf("%d", Exp ) [strict]
        | scanf("%d",& Exp )
        | NULL
        | PointerId
        | (int*)malloc( Exp *sizeof(int)) [strict]
        | free( Exp ) [strict]
        | * Exp [strict]
        | Exp [ Exp ]
        | Exp = Exp [strict(2)]
        | Id ( List{Exp} ) [strict(2)]
        | Id ()
        | srandom( Exp ) [strict]
        | random()
  Stmt ::= {}
         | Exp ; [strict]
         | { StmtList }
         | if( Exp ) Stmt
         | if( Exp ) Stmt else Stmt [strict(1)]
         | while( Exp ) Stmt
         | return Exp ; [strict]
         | DeclId List{DeclId} { StmtList }
         | #include< StmtList >
  StmtList ::= StmtList StmtList
             | Stmt
  Pgm ::= StmtList
  Id ::= main
  PointerId ::= Id
              | * PointerId [ditto ]
  DeclId ::= int Exp
           | void PointerId
  StmtList ::= stdio.h
             | stdlib.h
  List{Bottom} ::= List{Bottom} , List{Bottom} [assoc hybrid id: () strict]
                 | ()
                 | Bottom
  List{PointerId} ::= List{PointerId} , List{PointerId} [assoc ditto id: () ]
                    | List{Bottom}
                    | PointerId
  List{DeclId} ::= List{DeclId} , List{DeclId} [assoc ditto id: () ]
                 | DeclId
                 | List{Bottom}
  List{Exp} ::= List{Exp} , List{Exp} [assoc ditto id: () ]
              | Exp
              | List{DeclId}
              | List{PointerId}
END MODULE
```

```
MODULE KERNELC-DESUGARED-SYNTAX
  IMPORTS KERNELC-SYNTAX
   MACRO:  ! E = E ? 0 : 1
   MACRO:  E1 && E2 = E1 ? E2 : 0
   MACRO:  E1 || E2 = E1 ? 1 : E2
   MACRO:  if( E ) St = if( E ) St else {}
   MACRO:  NULL = 0
   MACRO:  I () = I ( () )
   MACRO:  int * PointerId = int PointerId
   MACRO:  #include< Stmts > = Stmts
   MACRO:  E1 [ E2 ] = * E1 + E2
   MACRO:  scanf("%d",& * E ) = scanf("%d", E )
   MACRO:  int * PointerId = E = int PointerId = E
   MACRO:  int X = E ; = int X ; X = E ;
   MACRO:  stdio.h = {}
   MACRO:  stdlib.h = {}
END MODULE
```

MODULE KERNELC-SEMANTICS
IMPORTS K-SHARED
IMPORTS K+KERNELC-DESUGARED-SYNTAX+PL-CONVERSION+PL-RANDOM
CONFIGURATION:

$$\langle \langle \cdot_K \rangle_{k} \langle \cdot_{Map} \rangle_{env} \langle \cdot_{Map} \rangle_{funs} \langle \cdot_{List} \rangle_{fstack} \langle \cdot_{List} \rangle_{in} \langle \text{""} \rangle_{out} \langle \cdot_{Map} \rangle_{mem} \langle \cdot_{Map} \rangle_{ptr} \langle 0 \rangle_{next} \langle 0 \rangle_{rand} \rangle_{\top ?} \; \langle \text{""} \rangle_{result ?}$$

RULE: $\langle \frac{X}{V} \ldots \rangle_{k} \; \langle \ldots X \mapsto V \ldots \rangle_{env}$

RULE: $\langle \frac{X\,++}{I} \ldots \rangle_{k} \; \langle \ldots X \mapsto \frac{I}{I +_{Int} 1} \ldots \rangle_{env}$

RULE: $\langle \frac{X = V}{V} \ldots \rangle_{k} \; \langle \ldots X \mapsto \frac{\_}{V} \ldots \rangle_{env}$

RULE: $I_1 + I_2 \to I_1 +_{Int} I_2$

RULE: $I_1 - I_2 \to I_1 -_{Int} I_2$

RULE: $I_1 \% I_2 \to I_1 \%_{Int} I_2$ when $I_2 \; !=_{Int} 0$

RULE: $I_1 <= I_2 \to$ Bool2Int $(I_1 \leq_{Int} I_2)$

RULE: $I_1 < I_2 \to$ Bool2Int $(I_1 <_{Int} I_2)$

RULE: $I_1 == I_2 \to$ Bool2Int $(I_1 ==_{Int} I_2)$

RULE: $I_1 \; != I_2 \to$ Bool2Int $(I_1 \; !=_{Int} I_2)$

RULE: `_?_:_` → `if(_)_else_`

RULE: if(I) _ else $St \to St$ when $I ==_{Int} 0$

RULE: if(I) St else _ $\to St$ when $\neg_{Bool} I ==_{Int} 0$

RULE: $\langle \frac{\texttt{while(}E\texttt{)}\ St}{\texttt{if(}E\texttt{) \{}St\ \texttt{while(}E\texttt{)}\ St\texttt{\} else \{\}}} \ldots \rangle_{k}$

PRINT RULE: $\langle \frac{\texttt{printf("\%d;",}\ I\texttt{)}}{\texttt{void}} \ldots \rangle_{k} \; \langle \frac{S}{S +_{String} \text{Int2String}(I) +_{String} \text{";"}} \rangle_{out}$

READ-GLOBAL RULE: $\langle \frac{\texttt{scanf("\%d",}\ N\texttt{)}}{\texttt{void}} \ldots \rangle_{k} \; \langle \ldots N \mapsto \frac{\_}{I} \ldots \rangle_{mem} \; \langle \frac{I}{\cdot_{List}} \ldots \rangle_{in}$

READ-LOCAL RULE: $\langle \frac{\texttt{scanf("\%d",\&}\ X\texttt{)}}{\texttt{void}} \ldots \rangle_{k} \; \langle \ldots X \mapsto \frac{\_}{I} \ldots \rangle_{env} \; \langle \frac{I}{\cdot_{List}} \ldots \rangle_{in}$

RULE: $V\ ; \to \cdot_K$

RULE: $\{\ Sts\ \} \to Sts$

RULE: $\{\} \to \cdot_K$

RULE: $St\ \ Sts \to St \curvearrowright Sts$

RULE: $\langle \frac{\texttt{int}\ X\ \ Xl\ \{\ Sts\ \}}{\cdot_K} \ldots \rangle_{k} \; \langle \ldots \frac{\cdot_{Map}}{X \mapsto \texttt{int}\ X\ \ Xl\ \{\ Sts\ \}} \ldots \rangle_{funs}$

RULE: $\frac{\texttt{void}\ X}{\texttt{int}\ X}\ Xl\ \{ \frac{Sts}{Sts\ \texttt{return void ;}} \}$

$ListItem ::= Id \# Map \# K$

RULE: $\left(\langle \frac{X\ (\ Vl\ ) \curvearrowright K}{Sts} \rangle_{k} \; \langle \frac{Env}{\text{eraseKLabel}(\texttt{int\_}, Xl) \mapsto Vl} \rangle_{env} \; \langle \ldots X \mapsto \texttt{int}\ X\ \ Xl\ \{\ Sts\ \} \ldots \rangle_{funs} \; \langle \frac{\cdot_{List}}{X \# Env \# K} \ldots \rangle_{fstack} \right)$

CONTEXT: int _ = $\square$

RULE: $\langle \frac{\texttt{int}\ X}{\texttt{void}} \ldots \rangle_{k} \; \langle \ldots \frac{\cdot_{Map}}{X \mapsto \texttt{undef}} \ldots \rangle_{env}$

RULE: $\langle \frac{\texttt{return}\ V\ ; \curvearrowright \_}{V} \rangle_{k}$

RULE: $\langle V \curvearrowright \frac{\cdot_K}{K} \rangle_{k} \; \langle \frac{\_}{Env} \rangle_{env} \; \langle \frac{\_ \# Env \# K}{\cdot_{List}} \ldots \rangle_{fstack}$

RULE: $\left(\langle \frac{\texttt{(int*)malloc(}N\ \texttt{*sizeof(int))}}{N'} \ldots \rangle_{k} \; \langle \ldots \frac{\cdot_{Map}}{N' \mapsto N} \ldots \rangle_{ptr} \; \langle \ldots \frac{\cdot_{Map}}{N' \ldots N +_{Nat} N' \mapsto \texttt{undef}} \ldots \rangle_{mem} \; \langle \frac{N'}{N +_{Int} N'} \rangle_{next} \right)$

RULE: $\langle \frac{\texttt{free(}N\texttt{)}}{\texttt{void}} \ldots \rangle_{k} \; \langle \ldots \frac{N \mapsto N'}{\cdot_{Map}} \ldots \rangle_{ptr} \; \langle \frac{Mem}{Mem\ [\perp / N \ldots N +_{Nat} N']} \rangle_{mem}$

RULE: $\langle \frac{\texttt{random()}}{\text{randomRandom}(N')} \ldots \rangle_{k} \; \langle \frac{N'}{N' +_{Nat} 1} \rangle_{rand}$

RULE: $\langle \frac{\texttt{srandom(}I\texttt{)}}{\texttt{void}} \ldots \rangle_{k}$

CONTEXT: $* \square = \_$
CONTEXT: $* \square ++$

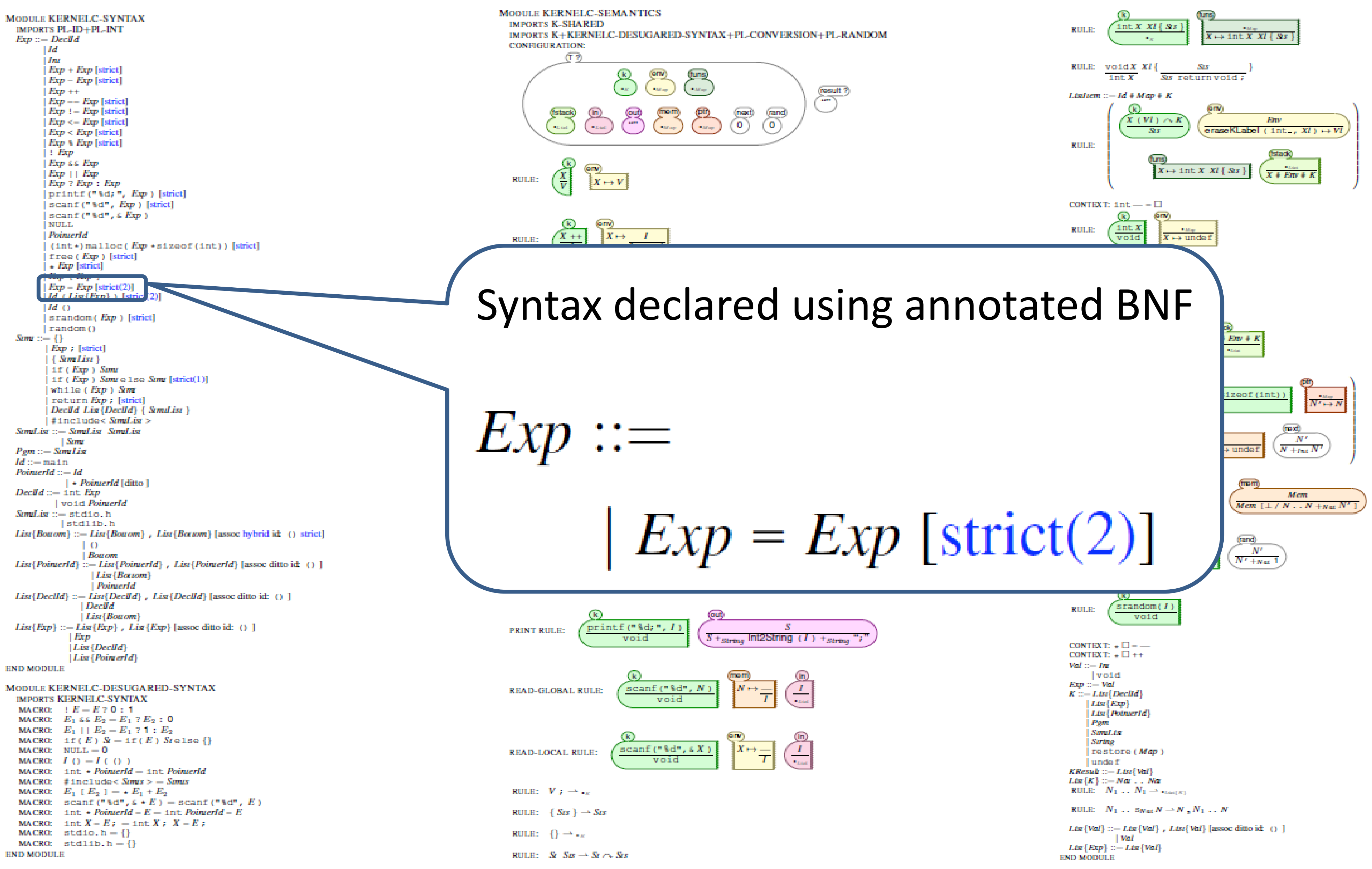
```
Val ::= Int
      | void
Exp ::= Val
K ::= List{DeclId}
    | List{Exp}
    | List{PointerId}
    | Pgm
    | StmtList
    | String
    | restore( Map )
    | undef
KResult ::= List{Val}
List{K} ::= Nat .. Nat
```

RULE: $N_1 \ldots N_1 \to \cdot_{List\{K\}}$

RULE: $N_1 \ldots s_{Nat} N \to N\ ,\ N_1 \ldots N$

```
List{Val} ::= List{Val} , List{Val} [assoc ditto id: () ]
            | Val
List{Exp} ::= List{Val}
END MODULE
```

Complete K Definition of KernelC



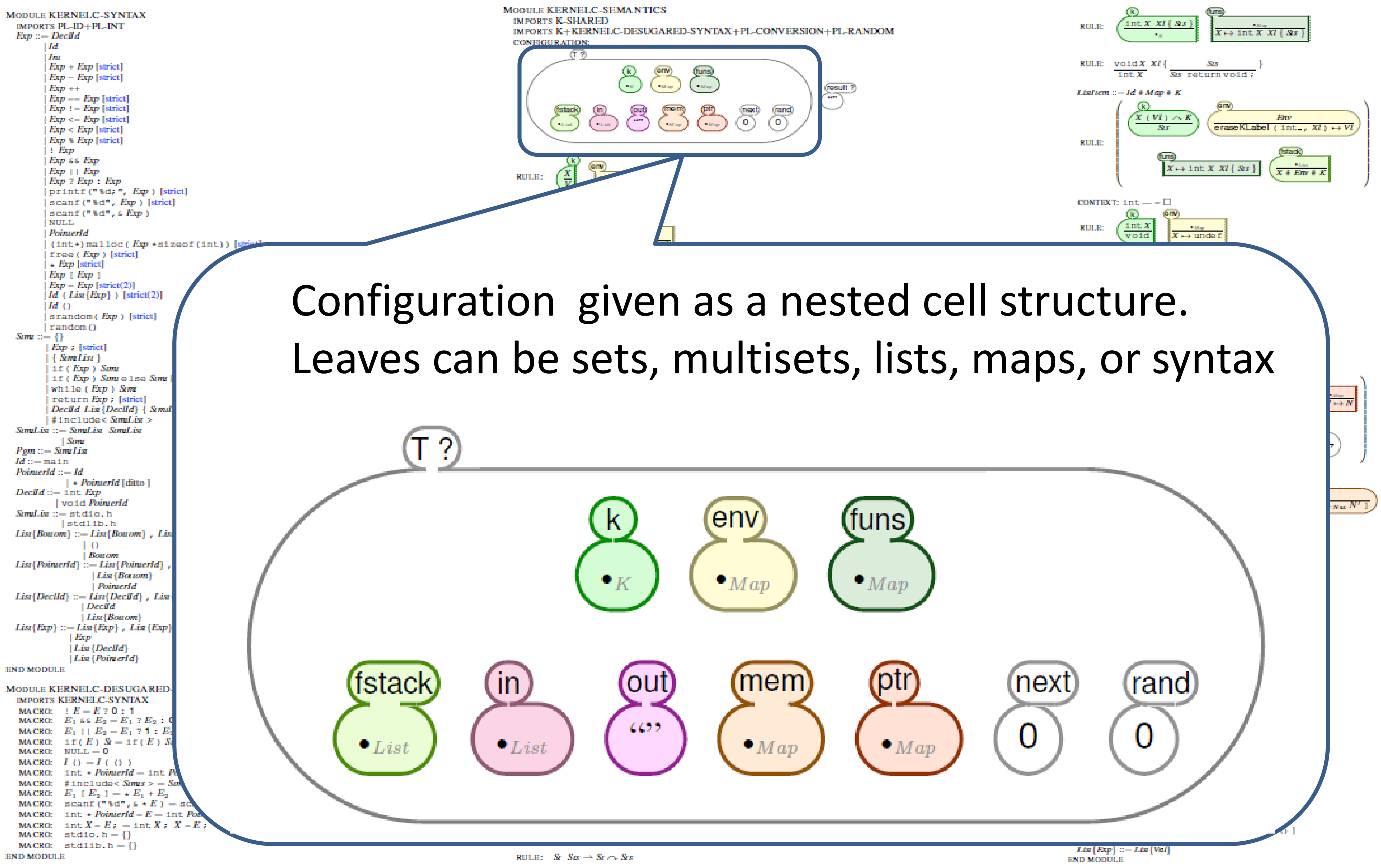
```
MODULE KERNELC-SYNTAX
  IMPORTS PL-ID+PL-INT
  Exp ::= DeclId
        | Id
        | Int
        | Exp + Exp [strict]
        | Exp - Exp [strict]
        | Exp ++
        | Exp == Exp [strict]
        | Exp != Exp [strict]
        | Exp <= Exp [strict]
        | Exp < Exp [strict]
        | Exp % Exp [strict]
        | ! Exp
        | Exp && Exp
        | Exp || Exp
        | Exp ? Exp : Exp
        | printf("%d;", Exp ) [strict]
        | scanf("%d", Exp ) [strict]
        | scanf("%d",& Exp )
        | NULL
        | PointerId
        | (int*)malloc( Exp *sizeof(int)) [strict]
        | free( Exp ) [strict]
        | * Exp [strict]
        | Exp [ Exp ]
        | Exp = Exp [strict(2)]
        | Id ( List{Exp} ) [strict(2)]
        | Id ()
        | srandom( Exp ) [strict]
        | random()
  Stmt ::= {}
         | Exp ; [strict]
         | { StmtList }
         | if( Exp ) Stmt
         | if( Exp ) Stmt else Stmt [strict(1)]
         | while( Exp ) Stmt
         | return Exp ; [strict]
         | DeclId List{DeclId} { StmtList }
         | #include< StmtList >
  StmtList ::= StmtList StmtList
             | Stmt
  Pgm ::= StmtList
  Id ::= main
  PointerId ::= Id
              | * PointerId [ditto]
  DeclId ::= int Exp
           | void PointerId
  StmtList ::= stdio.h
             | stdlib.h
  List{Bottom} ::= List{Bottom} , List{Bottom} [assoc hybrid id: () strict]
                 | ()
                 | Bottom
  List{PointerId} ::= List{PointerId} , List{PointerId} [assoc ditto id: () ]
                    | List{Bottom}
                    | PointerId
  List{DeclId} ::= List{DeclId} , List{DeclId} [assoc ditto id: () ]
                 | DeclId
                 | List{Bottom}
  List{Exp} ::= List{Exp} , List{Exp} [assoc ditto id: () ]
              | Exp
              | List{DeclId}
              | List{PointerId}
END MODULE

MODULE KERNELC-DESUGARED-SYNTAX
  IMPORTS KERNELC-SYNTAX
   MACRO:  ! E = E ? 0 : 1
   MACRO:  E1 && E2 = E1 ? E2 : 0
   MACRO:  E1 || E2 = E1 ? 1 : E2
   MACRO:  if( E ) St = if( E ) St else {}
   MACRO:  NULL = 0
   MACRO:  I () = I ( () )
   MACRO:  int * PointerId = int PointerId
   MACRO:  #include< Stmts > = Stmts
   MACRO:  E1 [ E2 ] = * E1 + E2
   MACRO:  scanf("%d",& * E ) = scanf("%d", E )
   MACRO:  int * PointerId = E = int PointerId = E
   MACRO:  int X = E ; = int X ; X = E ;
   MACRO:  stdio.h = {}
   MACRO:  stdlib.h = {}
END MODULE

MODULE KERNELC-SEMANTICS
  IMPORTS K-SHARED
  IMPORTS K+KERNELC-DESUGARED-SYNTAX+PL-CONVERSION+PL-RANDOM
  CONFIGURATION:
```

```
  RULE:  V ; → ·K
  RULE:  { Sts } → Sts
  RULE:  {} → ·K
  RULE:  St Sts → St ↷ Sts

  CONTEXT: * □ = —
  CONTEXT: * □ ++
  Val ::= Int
        | void
  Exp ::= Val
  K ::= List{DeclId}
      | List{Exp}
      | List{PointerId}
      | Pgm
      | StmtList
      | String
      | restore( Map )
      | undef
  KResult ::= List{Val}
  List[K] ::= Nat .. Nat
   RULE:  N1 .. N1 → ·List[K]

   RULE:  N1 .. sNat N → N , N1 .. N

  List[Val] ::= List[Val] , List{Val} [assoc ditto id: () ]
              | Val
  List[Exp] ::= List[Val]
END MODULE
```

Syntax declared using annotated BNF

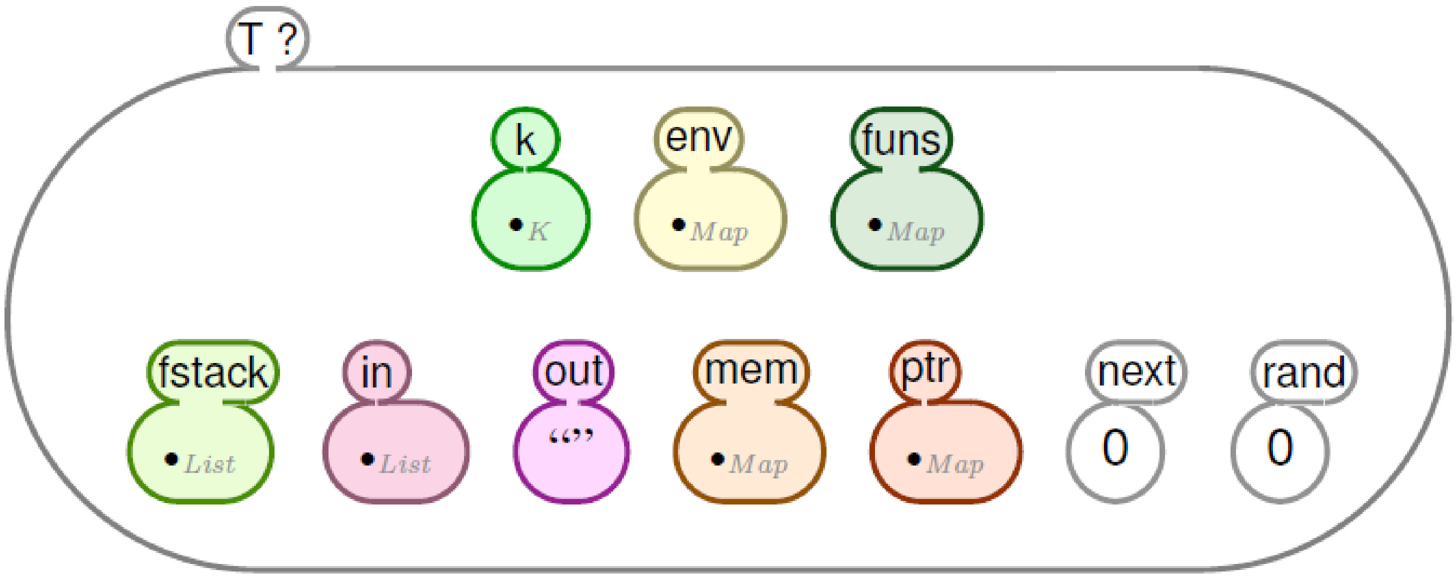
$$Exp ::= \ldots$$
$$\mid Exp = Exp \ [\text{strict}(2)]$$

Complete K Definition of KernelC

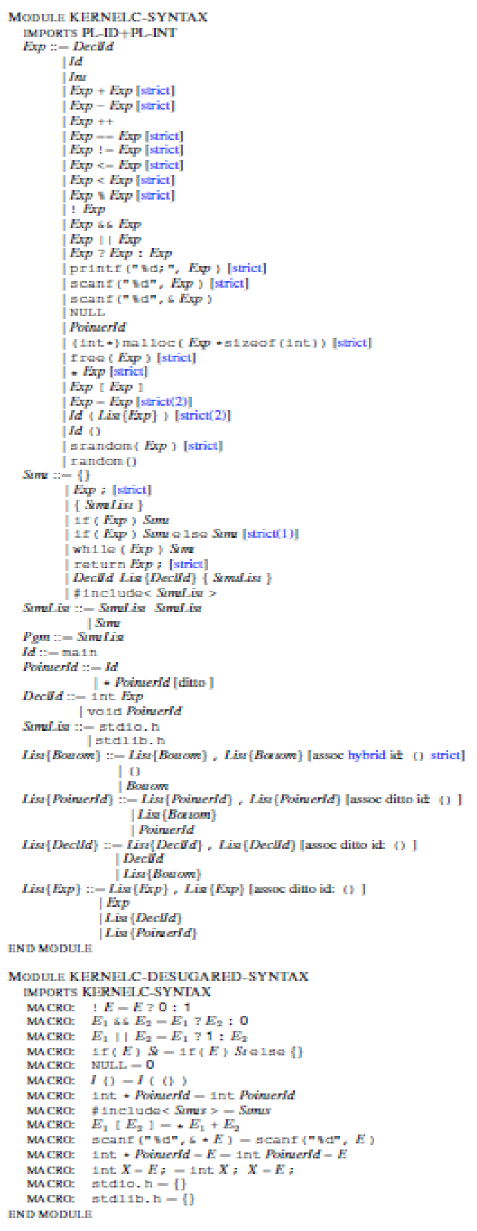


```
MODULE KERNELC-SYNTAX
  IMPORTS PL-ID+PL-INT
  Exp ::= DeclId
        | Id
        | Int
        | Exp + Exp [strict]
        | Exp - Exp [strict]
        | Exp ++
        | Exp == Exp [strict]
        | Exp != Exp [strict]
        | Exp <= Exp [strict]
        | Exp < Exp [strict]
        | Exp % Exp [strict]
        | ! Exp
        | Exp && Exp
        | Exp || Exp
        | Exp ? Exp : Exp
        | printf("%d;", Exp ) [strict]
        | scanf("%d", Exp ) [strict]
        | scanf("%d",& Exp )
        | NULL
        | PointerId
        | (int*)malloc( Exp *sizeof(int))
        | free( Exp ) [strict]
        | * Exp [strict]
        | Exp [ Exp ]
        | Exp = Exp [strict(2)]
        | Id ( List{Exp} ) [strict(2)]
        | Id ()
        | srandom( Exp ) [strict]
        | random()
  Stmt ::= {}
        | Exp ; [strict]
        | { StmtList }
        | if( Exp ) Stmt
        | if( Exp ) Stmt else Stmt
        | while( Exp ) Stmt
        | return Exp ; [strict]
        | DeclId List{DeclId} { StmtList
        | #include< StmtList >
  StmtList ::= StmtList StmtList
        | Stmt
  Pgm ::= StmtList
  Id ::= main
  PointerId ::= Id
        | * PointerId [ditto]
  DeclId ::= int Exp
        | void PointerId
  StmtList ::= stdio.h
        | stdlib.h
  List{Bottom} ::= List{Bottom} , List
        | ()
        | Bottom
  List{PointerId} ::= List{PointerId} ,
        | List{Bottom}
        | PointerId
  List{DeclId} ::= List{DeclId} , List
        | DeclId
        | List{Bottom}
  List{Exp} ::= List{Exp} , List{Exp}
        | Exp
        | List{DeclId}
        | List{PointerId}
END MODULE

MODULE KERNELC-DESUGARED-
  IMPORTS KERNELC-SYNTAX
  MACRO: ! E = E ? 0 : 1
  MACRO: E1 && E2 = E1 ? E2 : 0
  MACRO: E1 || E2 = E1 ? 1 : E2
  MACRO: if( E ) St = if( E ) St
  MACRO: NULL = 0
  MACRO: I () = I ( () )
  MACRO: int * PointerId = int Po
  MACRO: #include< Stmts > = St
  MACRO: E1 [ E2 ] = * E1 + E2
  MACRO: scanf("%d",& * E ) = sc
  MACRO: int * PointerId = E = int Poi
  MACRO: int X = E ; = int X ; X = E ;
  MACRO: stdio.h = {}
  MACRO: stdlib.h = {}
END MODULE

MODULE KERNELC-SEMANTICS
  IMPORTS K-SHARED
  IMPORTS K+KERNELC-DESUGARED-SYNTAX+PL-CONVERSION+PL-RANDOM
  CONFIGURATION:

RULE:
RULE: St Sts → St ↷ Sts

RULE:
RULE: void X XI { Sts / Sts return void ; }
ListItem ::= Id # Map # K
RULE:
CONTEXT: int — = □
RULE:
List{Exp} ::= List{Val}
END MODULE
```

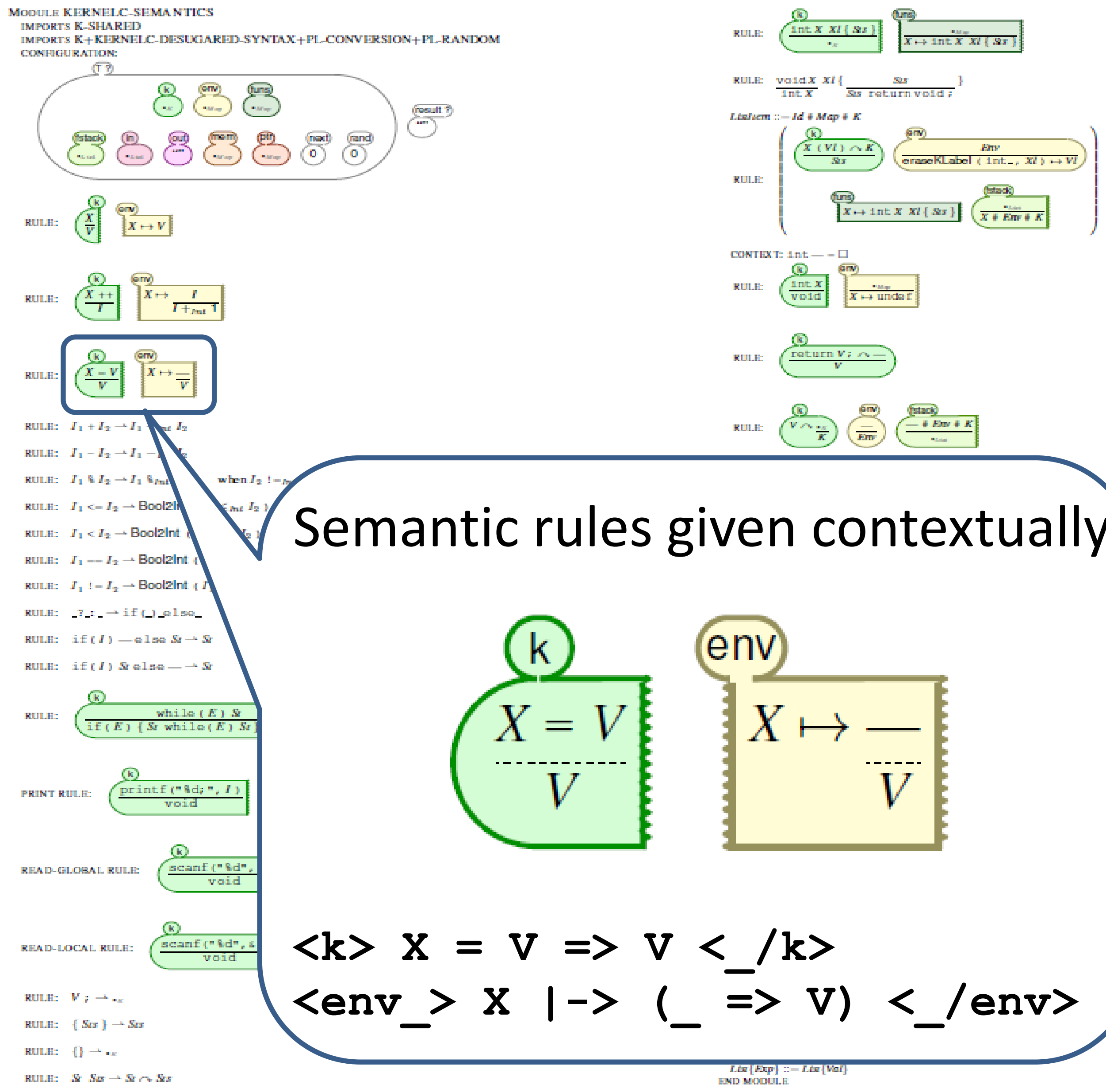
Configuration given as a nested cell structure.
Leaves can be sets, multisets, lists, maps, or syntax



Complete K Definition of KernelC



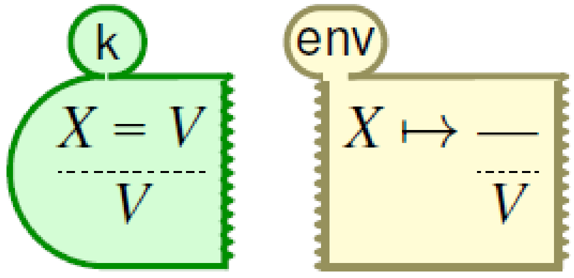
```
MODULE KERNELC-SYNTAX
  IMPORTS PL-ID+PL-INT
  Exp ::= DeclId
        | Id
        | Int
        | Exp + Exp [strict]
        | Exp - Exp [strict]
        | Exp ++
        | Exp == Exp [strict]
        | Exp != Exp [strict]
        | Exp <= Exp [strict]
        | Exp < Exp [strict]
        | Exp % Exp [strict]
        | ! Exp
        | Exp && Exp
        | Exp || Exp
        | Exp ? Exp : Exp
        | printf("%d;", Exp ) [strict]
        | scanf("%d", Exp ) [strict]
        | scanf("%d",& Exp )
        | NULL
        | PointerId
        | (int*)malloc( Exp *sizeof(int)) [strict]
        | free( Exp ) [strict]
        | * Exp [strict]
        | Exp [ Exp ]
        | Exp = Exp [strict(2)]
        | Id ( List{Exp} ) [strict(2)]
        | Id ()
        | srandom( Exp ) [strict]
        | random()
  Stmt ::= {}
         | Exp ; [strict]
         | { StmtList }
         | if( Exp ) Stmt
         | if( Exp ) Stmt else Stmt [strict(1)]
         | while( Exp ) Stmt
         | return Exp ; [strict]
         | DeclId List{DeclId} { StmtList }
         | #include< StmtList >
  StmtList ::= StmtList StmtList
             | Stmt
  Pgm ::= StmtList
  Id ::= main
  PointerId ::= Id
              | * PointerId [ditto]
  DeclId ::= int Exp
           | void PointerId
  StmtList ::= stdio.h
             | stdlib.h
  List{Bottom} ::= List{Bottom} , List{Bottom} [assoc hybrid id: () strict]
                 | ()
                 | Bottom
  List{PointerId} ::= List{PointerId} , List{PointerId} [assoc ditto id: () ]
                    | List{Bottom}
                    | PointerId
  List{DeclId} ::= List{DeclId} , List{DeclId} [assoc ditto id: () ]
                 | DeclId
                 | List{Bottom}
  List{Exp} ::= List{Exp} , List{Exp} [assoc ditto id: () ]
              | Exp
              | List{DeclId}
              | List{PointerId}
END MODULE

MODULE KERNELC-DESUGARED-SYNTAX
  IMPORTS KERNELC-SYNTAX
   MACRO:  ! E = E ? 0 : 1
   MACRO:  E1 && E2 = E1 ? E2 : 0
   MACRO:  E1 || E2 = E1 ? 1 : E2
   MACRO:  if( E ) St = if( E ) St else {}
   MACRO:  NULL = 0
   MACRO:  I () = I ( () )
   MACRO:  int * PointerId = int PointerId
   MACRO:  #include< Stmts > = Stmts
   MACRO:  E1 [ E2 ] = * E1 + E2
   MACRO:  scanf("%d",& * E ) = scanf("%d", E )
   MACRO:  int * PointerId = E = int PointerId = E
   MACRO:  int X = E ; = int X ; X = E ;
   MACRO:  stdio.h = {}
   MACRO:  stdlib.h = {}
END MODULE
```



```
MODULE KERNELC-SEMANTICS
  IMPORTS K-SHARED
  IMPORTS K+KERNELC-DESUGARED-SYNTAX+PL-CONVERSION+PL-RANDOM
  CONFIGURATION:
```

```
RULE:  I1 + I2 → I1 +Int I2
RULE:  I1 - I2 → I1 -Int I2
RULE:  I1 % I2 → I1 %Int        when I2 !=Int
RULE:  I1 <= I2 → Bool2Int        ≤Int I2 )
RULE:  I1 < I2 → Bool2Int (       I2 )
RULE:  I1 == I2 → Bool2Int (
RULE:  I1 != I2 → Bool2Int ( I
RULE:  _?_:_ → if(_)_else_
RULE:  if( I ) — else St → St
RULE:  if( I ) St else — → St
RULE:  V ; → •K
RULE:  { Sts } → Sts
RULE:  {} → •K
RULE:  St Sts → St ↷ Sts
```

Semantic rules given contextually



```
<k> X = V => V <_/k>
<env_> X |-> (_ => V) <_/env>
```

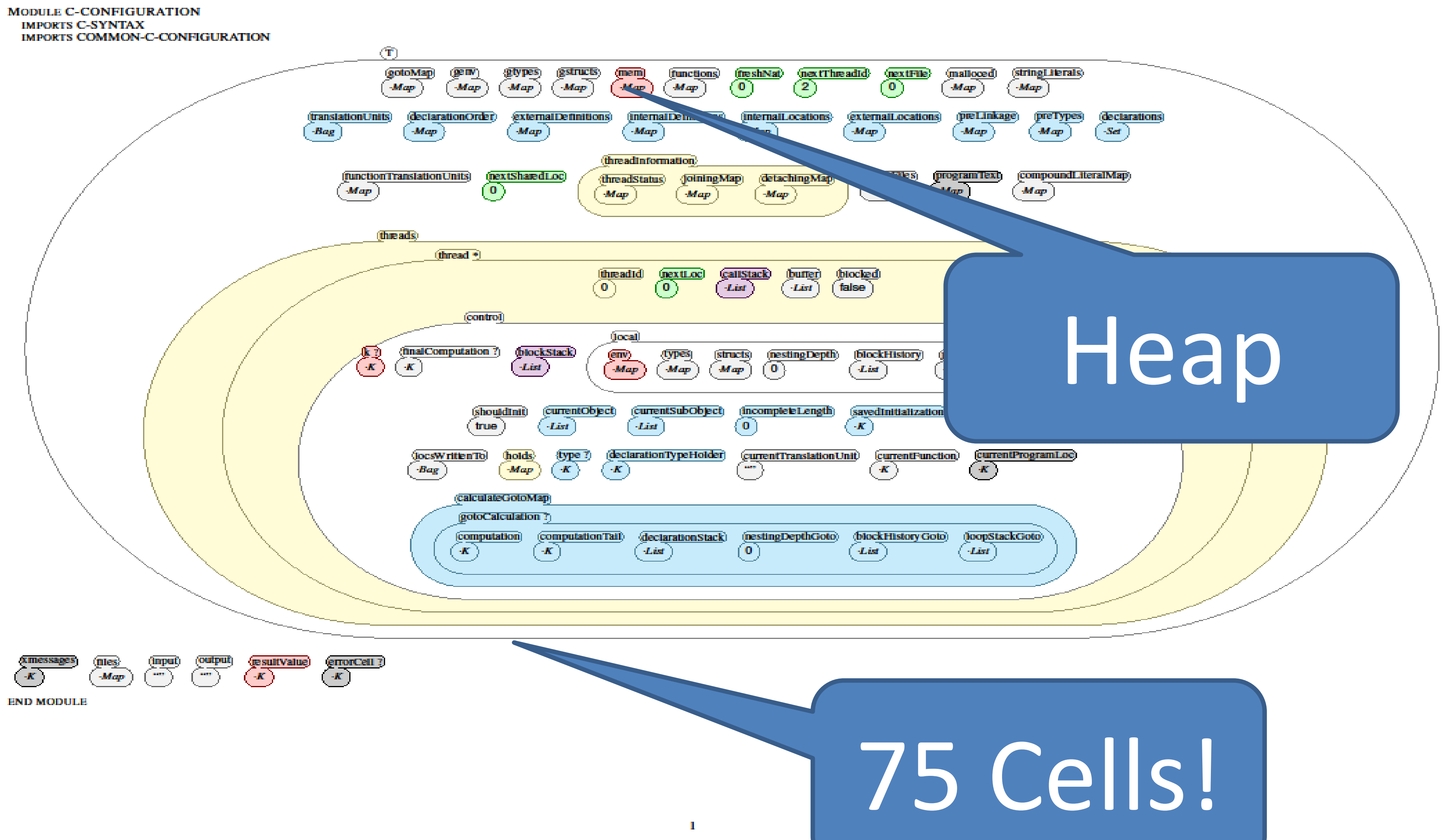
K Scales

Besides smaller and paradigmatic teaching languages, several larger languages were defined

- Scheme : by Pat Meredith
- Java 1.4 : by Feng Chen
- Verilog : by Pat Meredith and Mike Katelman
- C : by Chucky Ellison

etc.

The K Configuration of C



Heap

75 Cells!

Statistics for the C definition

Syntactic constructs:	173
Total number of rules:	812
Total number of lines:	4688

- Has been tested on thousands of C programs (several benchmarks, including the gcc torture test – passed 96% so far)
- The most complete formal C semantics
- Took more than 1 year to define …
  - Wouldn’t it be uneconomical to redefine it in each tool?

Executable Semantics are Useful

- Compiler certification (Leroy's talk)
- Help language designers
- K semantics are currently compiled into
  - Maude, for execution, debugging, model checking
  - Latex, for human inspection and understanding
  - Soon to OCAML, for fast execution
- Can we use K semantics for program verification?
  - E.g., we want to use the C semantics unchanged for program verification; defining an alternative (axiomatic) semantics is very inconvenient and error prone!

Matching Logic = K + FOL

- A logic for reasoning about configurations
- Formulae
  - FOL over configurations, called patterns
  - Configurations are allowed to contain variables
- Models
  - Ground configurations
- Satisfaction
  - Matching for configurations, plus FOL for the rest

Examples of Patterns I

- x is bound to 5 and is the only variable in environment

$$\langle \mathrm{x} \mapsto 5 \rangle_{\mathsf{env}}$$

- x is bound to 5 in the current environment

$$\exists \rho \; (\langle \mathrm{x} \mapsto 5, \;\; \rho \rangle_{\mathsf{env}})$$

- x is bound to a non-negative value

$$\exists a \exists \rho \; (\langle \mathrm{x} \mapsto a, \;\; \rho \rangle_{\mathsf{env}} \;\; \wedge \;\; a \geq 0)$$

- x and y hold equal positive values

$$\exists a \exists \rho \; (\langle \mathrm{x} \mapsto a, \;\; \mathrm{y} \mapsto a, \;\; \rho \rangle_{\mathsf{env}} \;\; \wedge \;\; a > 0)$$

Examples of Patterns II

- x and y are aliased

$$\exists a \exists \rho \exists u \exists \sigma\ (\langle \mathrm{x} \mapsto a,\ \mathrm{y} \mapsto a,\ \rho \rangle_{\mathsf{env}}\ \langle a \mapsto u,\ \sigma \rangle_{\mathsf{heap}})$$

- x and y not aliased, and x points to larger value

$$\exists a \exists b \exists \rho \exists u \exists v \exists \sigma\ (\langle \mathrm{x} \mapsto a,\ \mathrm{y} \mapsto b,\ \rho \rangle_{\mathsf{env}}\ \langle a \mapsto u,\ b \mapsto v,\ \sigma \rangle_{\mathsf{heap}}\ \wedge u > v)$$

- x points to a list containing sequence A

$$\exists a \exists \rho \exists \sigma\ (\langle \mathrm{x} \mapsto a,\ \rho \rangle_{\mathsf{env}}\ \langle \mathsf{list}(a, A),\ \sigma \rangle_{\mathsf{heap}})$$

Examples of Patterns III

- x points to sequence A, and the reversed sequence A has been output

$$\exists a \exists \rho \exists \sigma \exists \omega \ (\langle \mathtt{x} \mapsto a,\ \rho \rangle_{\mathsf{env}}\ \langle \mathsf{list}(a, A),\ \sigma \rangle_{\mathsf{heap}}\ \langle \omega, \mathsf{rev}(A) \rangle_{\mathsf{out}})$$

- **`untrusted()`** can only be from **`trusted()`**

$$\exists s_1 \exists s_2 \ (\langle \mathtt{untrusted()} \rangle_{\mathsf{k}}\ \langle s_1, \mathtt{trusted()}, s_2 \rangle_{\mathsf{fstack}})$$

- Read/Write datarace (simplified)

$$\exists X \exists a \ (\langle X\ \cdots \rangle_{\mathsf{k}}\ \langle X = a\ \cdots \rangle_{\mathsf{k}})$$

Matching Logic vs. Separation Logic

- Matching logic achieves separation naturally, through matching at the structural (term) level, not through special logical connectives (*)
- Matching logic realizes separation at all levels of the configuration, not only in the heap; recall that the heap was only 1 out of the 75 cells in C's def.
- Matching logic stays within FOL, while separation logic extends FOL
  - Thus, we can use the existing SMT solvers, etc.

Matching Logic as a Program Logic

- Hoare style - not recommended

$$\{\pi_{\mathsf{pre}}\}\ \mathsf{code}\ \{\pi_{\mathsf{post}}\}$$

  – One has to redefine the PL semantics – impractical

- Rewriting (or K) style – recommended

$$left[\mathsf{code}] \to right$$

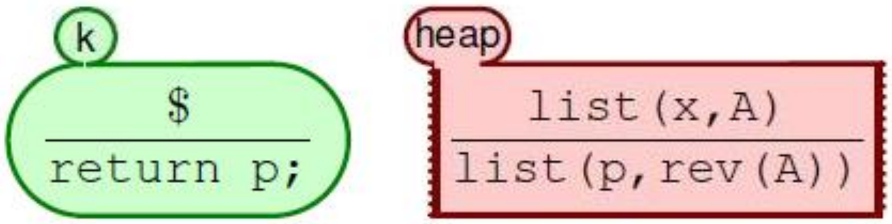
  – One can reuse existing K semantics – very good

Example – Reversing a list

```
struct listNode* reverse(struct listNode *x)
{
  struct listNode *p;
  struct listNode *y;
  p = 0 ;
  while(x) {
    y = x->next;
    x->next = p;
    p = x;
    x = y;
  }
  return p;
}
```

$

- What is the K semantics of the reverse function?
- Let $ be its body



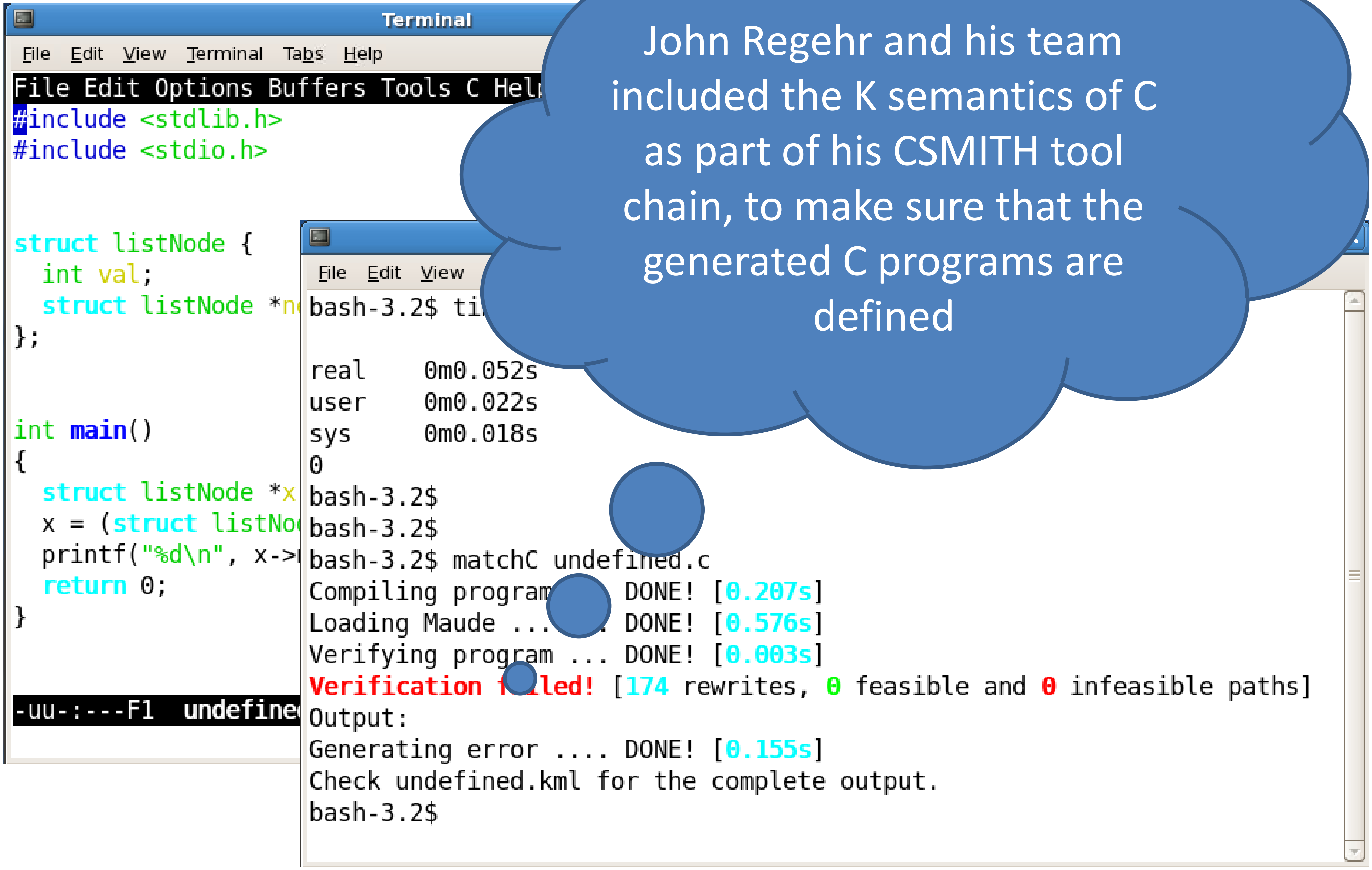
```
<k> $ => return p; </k>
<heap_> list(x,A) => list(p,rev(A)) <_/heap>
```

Partial Correctness

- We have two rewrite relations on configurations
  - $\rightarrow$ given by the language K semantics; safe
  - $\rightarrow$ given by specifications; unsafe, has to be proved
- Idea (simplified for deterministic languages):
  - Pick left $\rightarrow$ right. Show that always left $\rightarrow$ $(\rightarrow \cup \rightarrow)^*$ right modulo matching logic reasoning (between rewrite steps)
- Theorem (soundness):
  - If left $\rightarrow$ right and "config *matches* left" such that config has a normal form for $\rightarrow$, then "nf(config) *matches* right"

MatchC Tool DEMO (through slides)

Semantic Execution

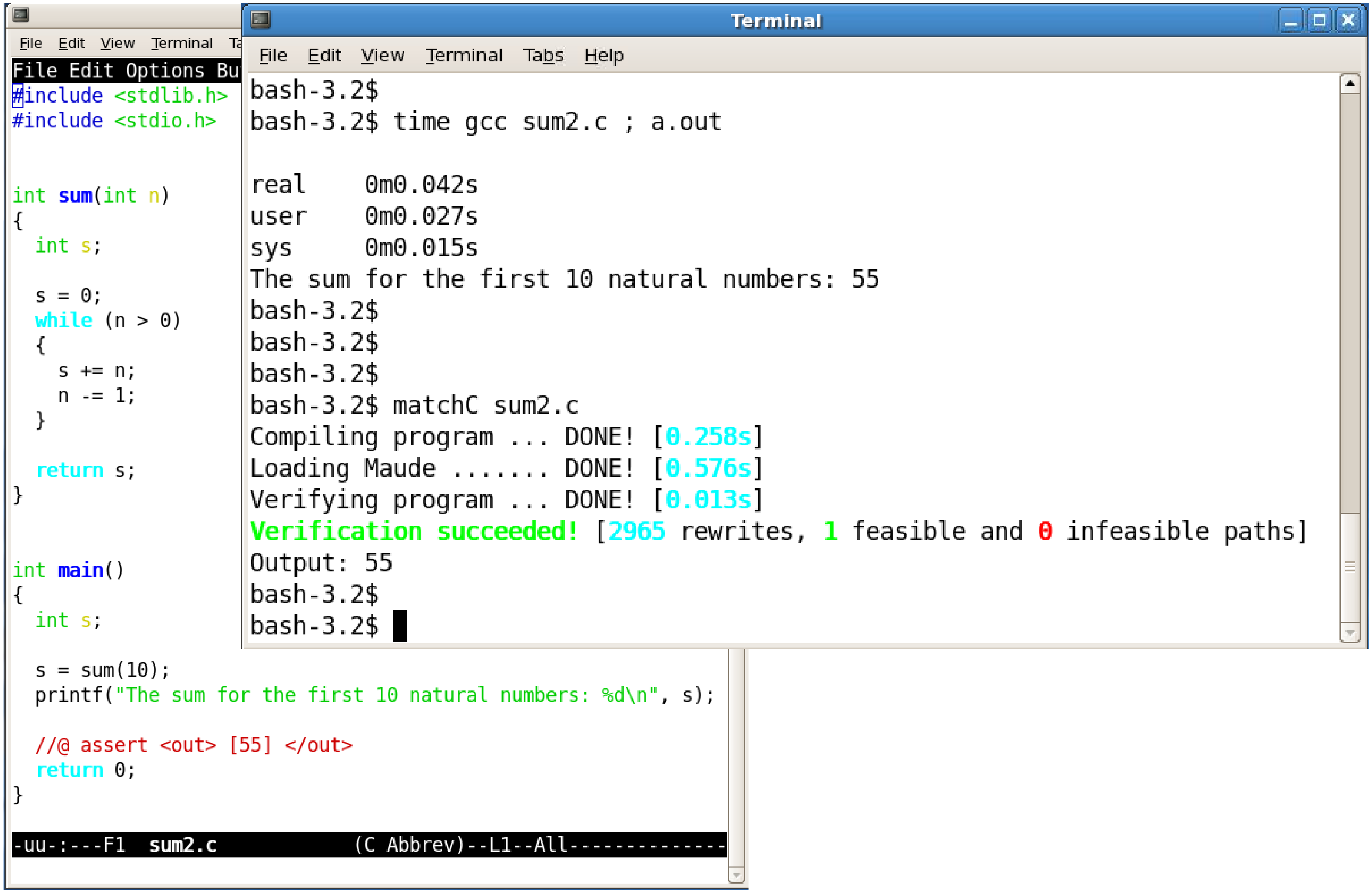


```
File Edit Options Buffers Tools C Help
#include <stdlib.h>
#include <stdio.h>


struct listNode {
  int val;
  struct listNode *n
};


int main()
{
  struct listNode *x
  x = (struct listNo
  printf("%d\n", x->
  return 0;
}


-uu-:---F1  undefined
```

```
bash-3.2$ ti

real    0m0.052s
user    0m0.022s
sys     0m0.018s
0
bash-3.2$
bash-3.2$
bash-3.2$ matchC undefined.c
Compiling program ... DONE! [0.207s]
Loading Maude ..... DONE! [0.576s]
Verifying program ... DONE! [0.003s]
Verification failed! [174 rewrites, 0 feasible and 0 infeasible paths]
Output:
Generating error .... DONE! [0.155s]
Check undefined.kml for the complete output.
bash-3.2$
```

John Regehr and his team included the K semantics of C as part of his CSMITH tool chain, to make sure that the generated C programs are defined

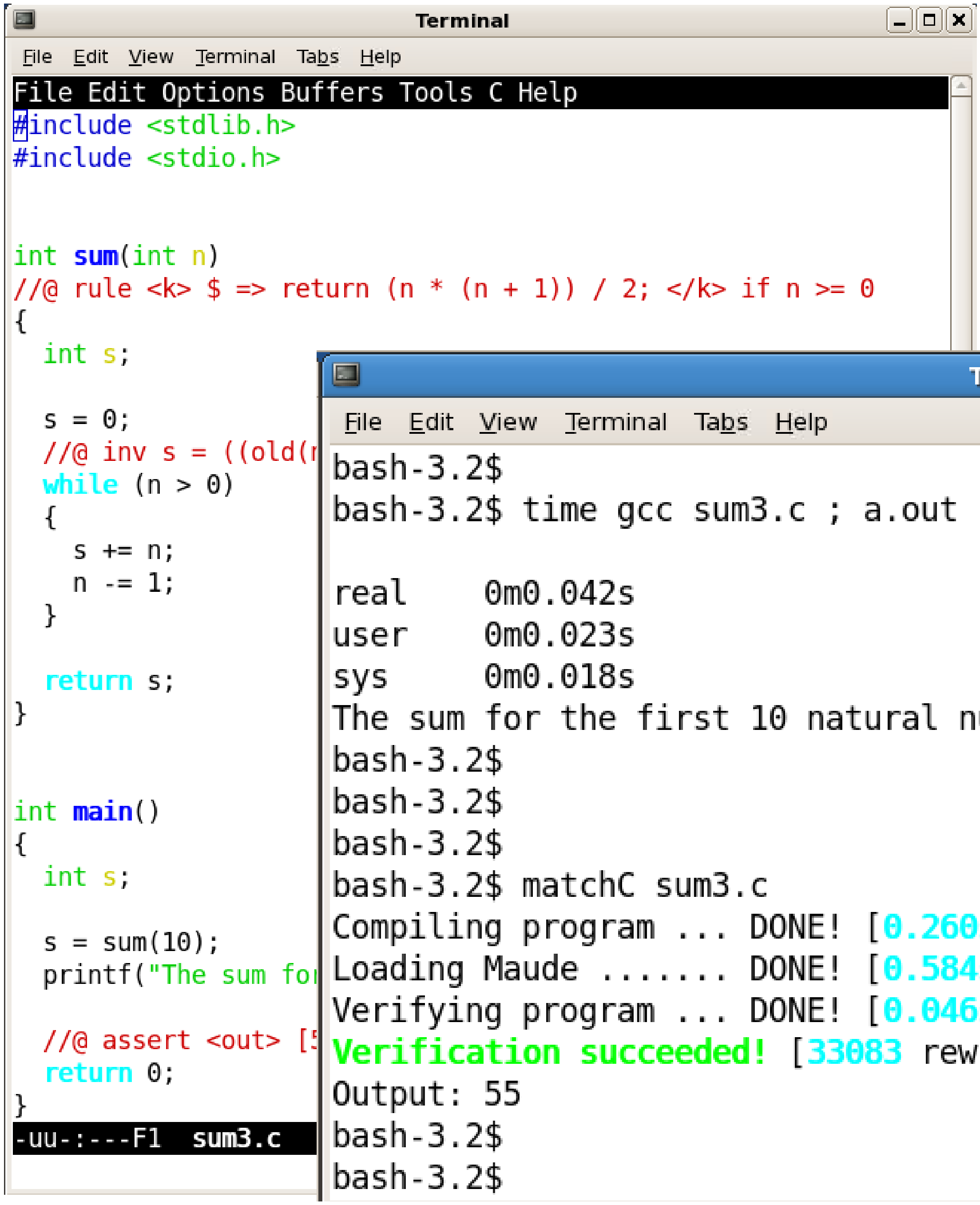
Assertion Checking



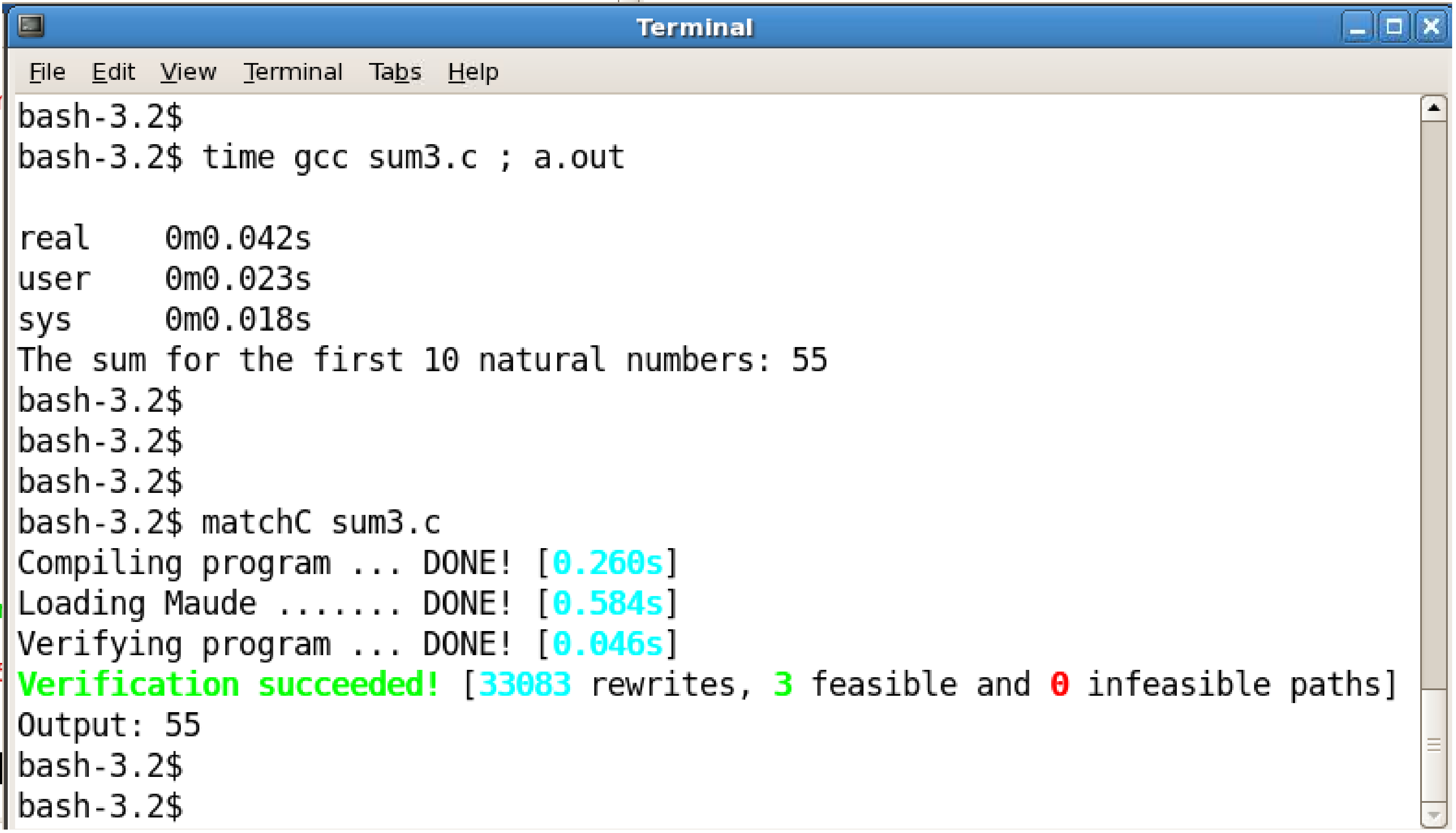
```
#include <stdlib.h>
#include <stdio.h>


int sum(int n)
{
  int s;

  s = 0;
  while (n > 0)
  {
    s += n;
    n -= 1;
  }

  return s;
}


int main()
{
  int s;

  s = sum(10);
  printf("The sum for the first 10 natural numbers: %d\n", s);

  //@ assert <out> [55] </out>
  return 0;
}

-uu-:---F1  sum2.c          (C Abbrev)--L1--All-------------
```

```
bash-3.2$
bash-3.2$ time gcc sum2.c ; a.out

real    0m0.042s
user    0m0.027s
sys     0m0.015s
The sum for the first 10 natural numbers: 55
bash-3.2$
bash-3.2$
bash-3.2$
bash-3.2$ matchC sum2.c
Compiling program ... DONE! [0.258s]
Loading Maude ....... DONE! [0.576s]
Verifying program ... DONE! [0.013s]
Verification succeeded! [2965 rewrites, 1 feasible and 0 infeasible paths]
Output: 55
bash-3.2$
bash-3.2$
```

Full Verification

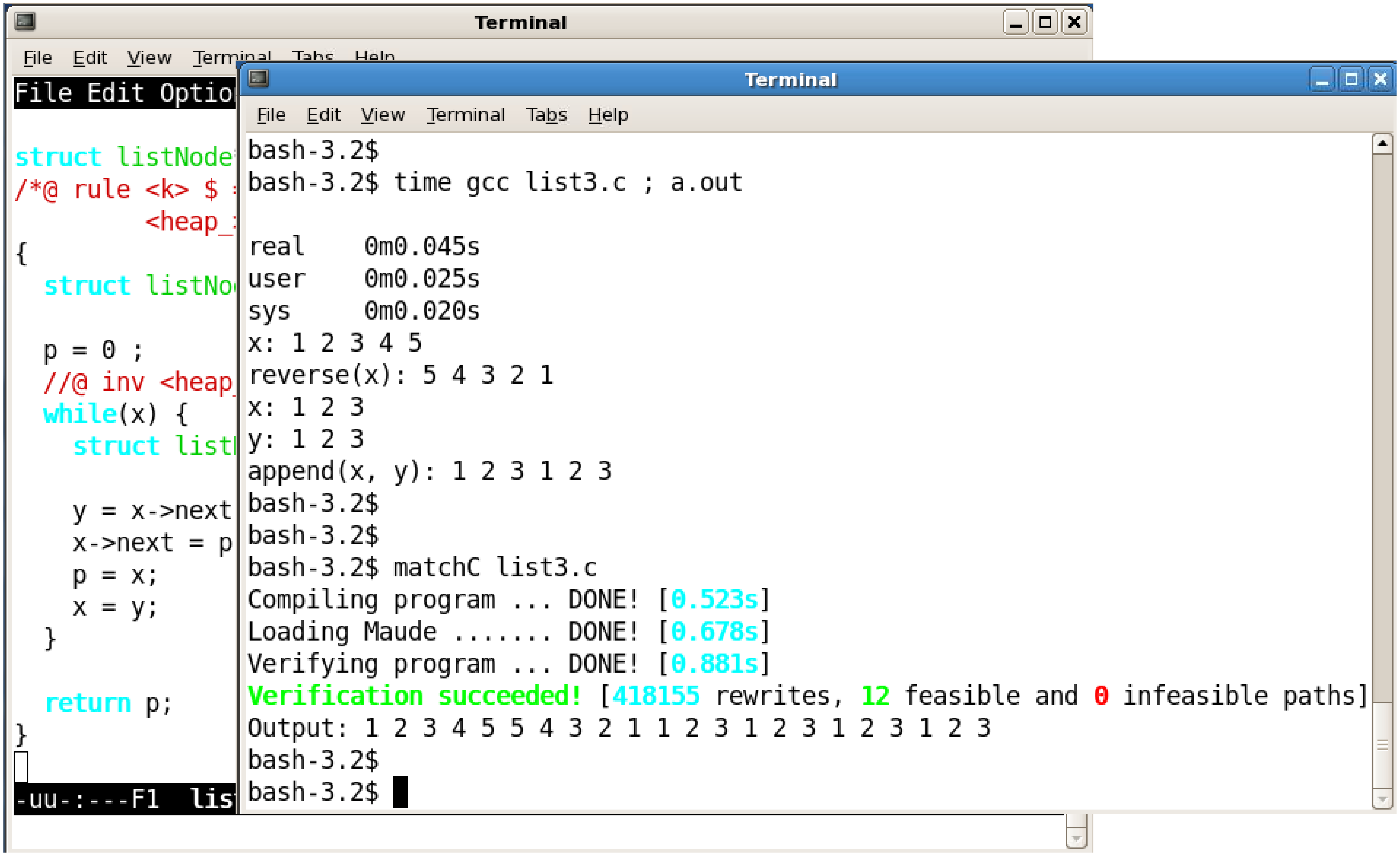


```
File Edit Options Buffers Tools C Help
#include <stdlib.h>
#include <stdio.h>


int sum(int n)
//@ rule <k> $ => return (n * (n + 1)) / 2; </k> if n >= 0
{
  int s;

  s = 0;
  //@ inv s = ((old(n
  while (n > 0)
  {
    s += n;
    n -= 1;
  }

  return s;
}


int main()
{
  int s;

  s = sum(10);
  printf("The sum for

  //@ assert <out> [5
  return 0;
}
-uu-:---F1  sum3.c
```



```
bash-3.2$
bash-3.2$ time gcc sum3.c ; a.out

real    0m0.042s
user    0m0.023s
sys     0m0.018s
The sum for the first 10 natural numbers: 55
bash-3.2$
bash-3.2$
bash-3.2$
bash-3.2$ matchC sum3.c
Compiling program ... DONE! [0.260s]
Loading Maude ....... DONE! [0.584s]
Verifying program ... DONE! [0.046s]
Verification succeeded! [33083 rewrites, 3 feasible and 0 infeasible paths]
Output: 55
bash-3.2$
bash-3.2$
```

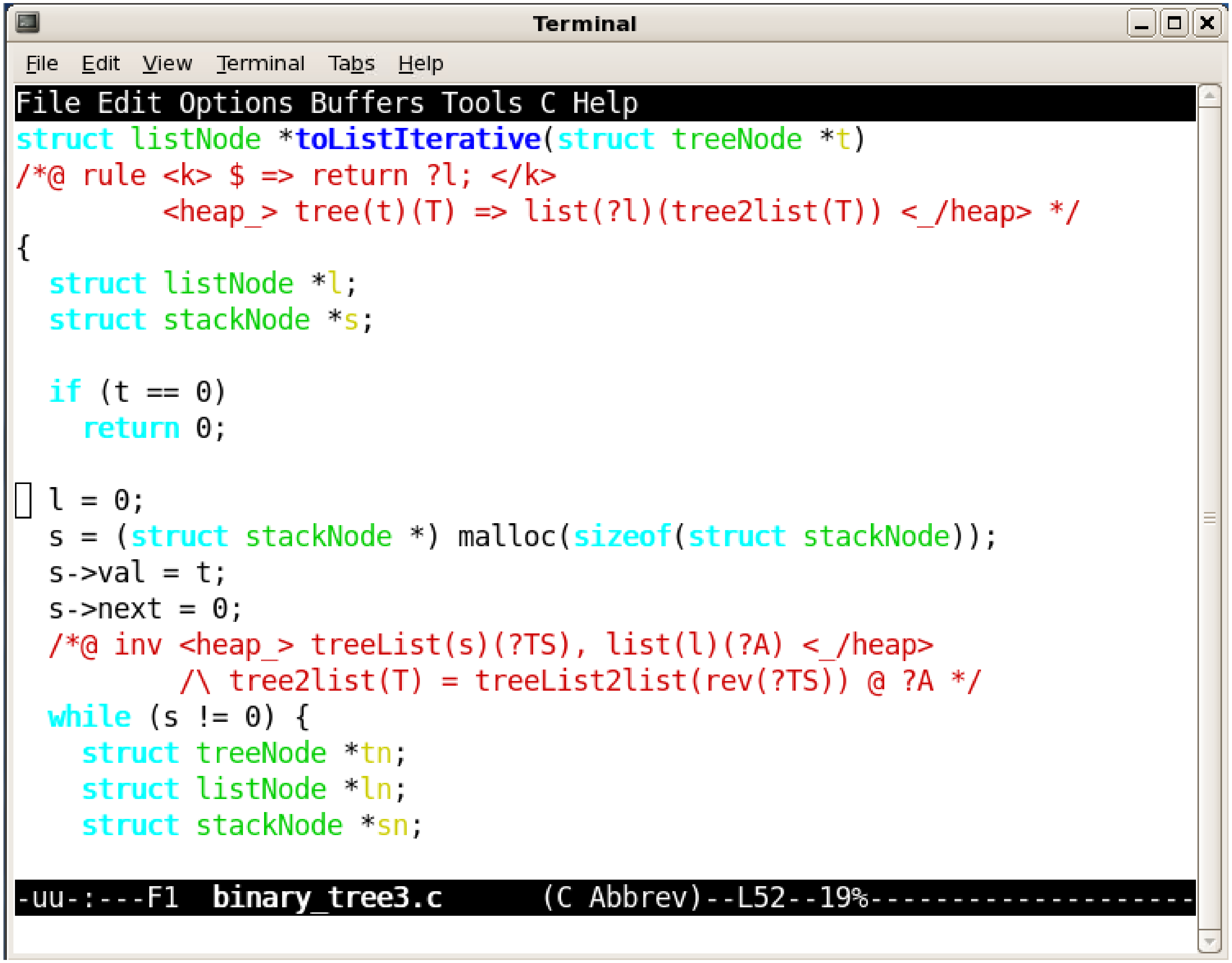
List Examples – Borrowed from SL tools


Terminal
File Edit View Terminal Tabs Help
Terminal
File Edit View Terminal Tabs Help

```
File Edit Optio

struct listNode
/*@ rule <k> $ 
        <heap_
{
  struct listNo

  p = 0 ;
  //@ inv <heap
  while(x) {
    struct list

    y = x->next
    x->next = p
    p = x;
    x = y;
  }

  return p;
}

-uu-:---F1  lis
```

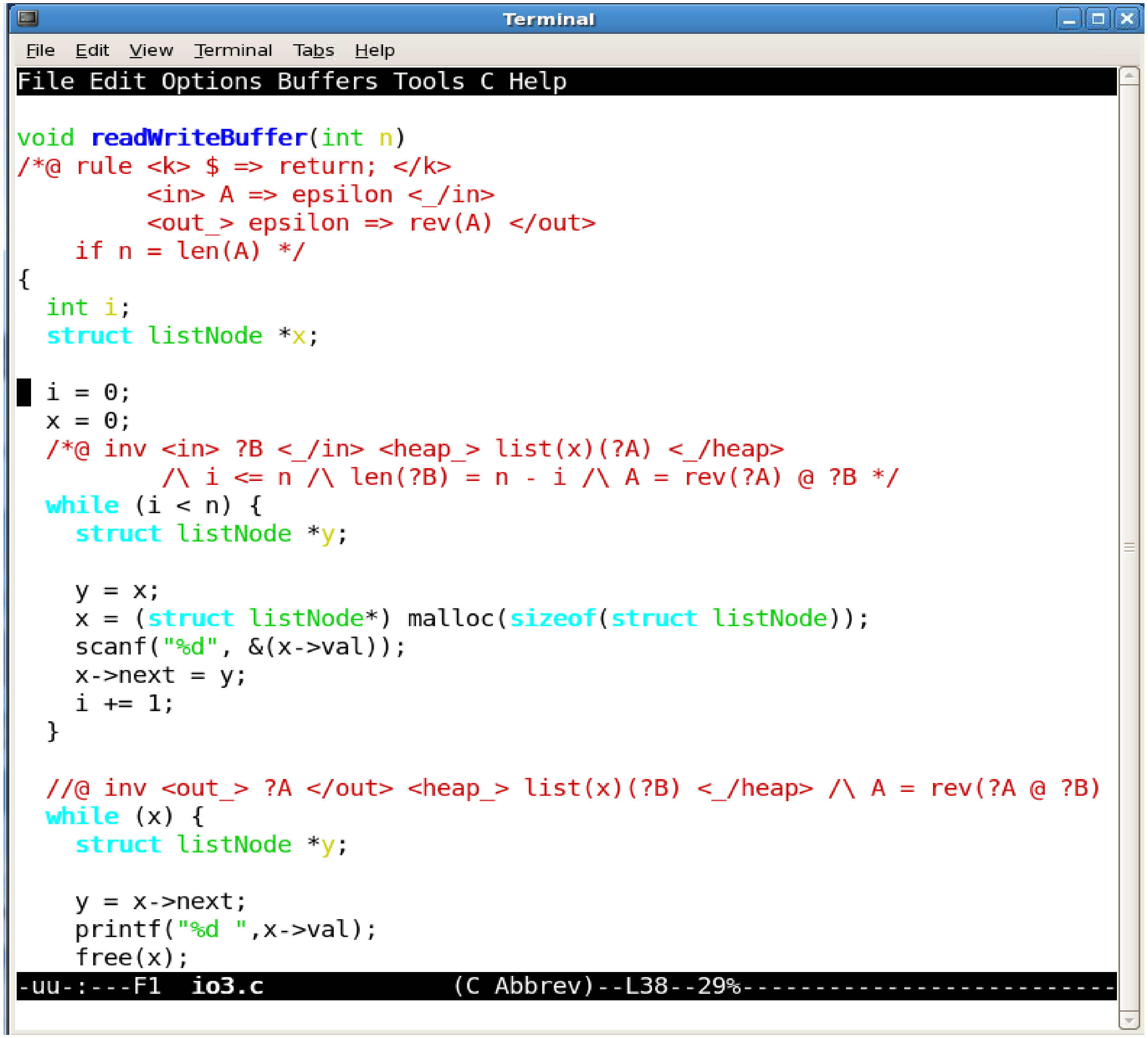
```
bash-3.2$
bash-3.2$ time gcc list3.c ; a.out

real    0m0.045s
user    0m0.025s
sys     0m0.020s
x: 1 2 3 4 5
reverse(x): 5 4 3 2 1
x: 1 2 3
y: 1 2 3
append(x, y): 1 2 3 1 2 3
bash-3.2$
bash-3.2$
bash-3.2$ matchC list3.c
Compiling program ... DONE! [0.523s]
Loading Maude ....... DONE! [0.678s]
Verifying program ... DONE! [0.881s]
Verification succeeded! [418155 rewrites, 12 feasible and 0 infeasible paths]
Output: 1 2 3 4 5 5 4 3 2 1 1 2 3 1 2 3 1 2 3 1 2 3
bash-3.2$
bash-3.2$
```

Beyond Separation Logic Tools


Terminal
File Edit View Terminal Tabs Help
File Edit Options Buffers Tools C Help

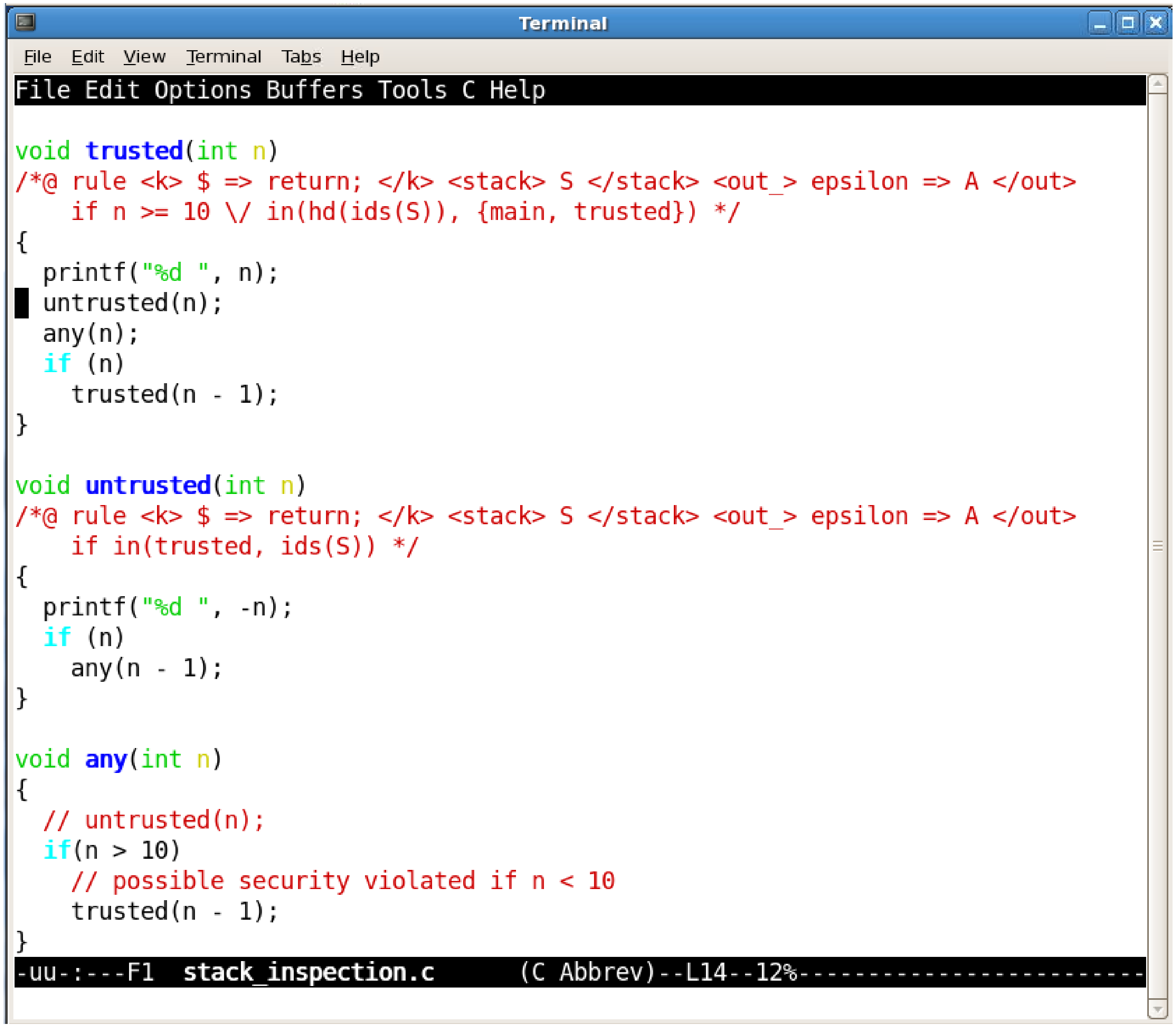
```
struct listNode *toListIterative(struct treeNode *t)
/*@ rule <k> $ => return ?l; </k>
         <heap_> tree(t)(T) => list(?l)(tree2list(T)) <_/heap> */
{
  struct listNode *l;
  struct stackNode *s;

  if (t == 0)
    return 0;

  l = 0;
  s = (struct stackNode *) malloc(sizeof(struct stackNode));
  s->val = t;
  s->next = 0;
  /*@ inv <heap_> treeList(s)(?TS), list(l)(?A) <_/heap>
          /\ tree2list(T) = treeList2list(rev(?TS)) @ ?A */
  while (s != 0) {
    struct treeNode *tn;
    struct listNode *ln;
    struct stackNode *sn;

-uu-:---F1  binary_tree3.c      (C Abbrev)--L52--19%-------------------
```

Beyond Separation Logic – I/O



```
File Edit Options Buffers Tools C Help

void readWriteBuffer(int n)
/*@ rule <k> $ => return; </k>
         <in> A => epsilon <_/in>
         <out_> epsilon => rev(A) </out>
    if n = len(A) */
{
  int i;
  struct listNode *x;

  i = 0;
  x = 0;
  /*@ inv <in> ?B <_/in> <heap_> list(x)(?A) <_/heap>
          /\ i <= n /\ len(?B) = n - i /\ A = rev(?A) @ ?B */
  while (i < n) {
    struct listNode *y;

    y = x;
    x = (struct listNode*) malloc(sizeof(struct listNode));
    scanf("%d", &(x->val));
    x->next = y;
    i += 1;
  }

  //@ inv <out_> ?A </out> <heap_> list(x)(?B) <_/heap> /\ A = rev(?A @ ?B)
  while (x) {
    struct listNode *y;

    y = x->next;
    printf("%d ",x->val);
    free(x);
-uu-:---F1  io3.c           (C Abbrev)--L38--29%------------------------
```

Beyond Separation Logic – Stack Inspection


Terminal
File Edit View Terminal Tabs Help
File Edit Options Buffers Tools C Help

```
void trusted(int n)
/*@ rule <k> $ => return; </k> <stack> S </stack> <out_> epsilon => A </out>
    if n >= 10 \/ in(hd(ids(S)), {main, trusted}) */
{
  printf("%d ", n);
  untrusted(n);
  any(n);
  if (n)
    trusted(n - 1);
}

void untrusted(int n)
/*@ rule <k> $ => return; </k> <stack> S </stack> <out_> epsilon => A </out>
    if in(trusted, ids(S)) */
{
  printf("%d ", -n);
  if (n)
    any(n - 1);
}

void any(int n)
{
  // untrusted(n);
  if(n > 10)
    // possible security violated if n < 10
    trusted(n - 1);
}
-uu-:---F1  stack_inspection.c      (C Abbrev)--L14--12%-----------------------
```

Conclusions

- Formal semantics is useful and practical!
- One can use an executable semantics of a language *as is* also for program verification
  - As opposed to redefining it as a Hoare logic
- Giving a formal semantics is not necessarily painful, it can be fun if one uses the right tools