DIRE:
A Neural Approach to Decompiled Identifier Renaming

Jeremy Lacomis, Pengcheng Yin,
Edward J. Schwartz, Miltiadis Allamanis,
Claire Le Goues, Graham Neubig, Bogdan Vasilescu

Carnegie Mellon University

STRUDEL
SOCIO-TECHNICAL RESEARCH USING DATA EXCAVATION LAB

squaresLab
Software Quality in Real Evolving Systems

Software Engineering Institute
Carnegie Mellon

NEULAB

Microsoft

Reverse Engineering

26 Feb 2013 | 14:00 GMT

The Real Story of Stuxnet

How Kaspersky Lab tracked down the malware that stymied Iran's nuclear-fuel enrichment program

By **David Kushner**

Computer cables snake across the floor. Cryptic flowcharts are scrawled across various whiteboards adorning the walls. A life-size Batman doll stands in the hall. This office might seem no different than any other geeky workplace, but in fact it's the front line of a war—a cyberwar, where most battles play out not in remote jungles or deserts but in

2014 IEEE International Conference on Software Maintenance and Evolution

Reverse Engineering PL/SQL Legacy Code: An Experience Report

Martin Habringer
voestalpine Stahl GmbH
4020 Linz, Austria
martin.habringer@voestalpine.com

Michael Moser and Josef Pichler
Software Analytics and Evolution
Software Competence Center Hagenberg GmbH
4232 Hagenberg, Austria
michael.moser@scch.at, josef.pichler@scch.at

***Abstract*—The reengineering of legacy code is a tedious endeavor. Automatic transformation of legacy code from an old technology to a new one preserves potential problems in legacy code with respect to obsolete, changed, and new business cases. On the other hand, manual analysis of legacy code without assistance of original developers is time consuming and error-prone. For the purpose of reengineering PL/SQL legacy code in the steel making domain, we developed tool support for the reverse engineering of PL/SQL code into a more abstract and comprehensive representation. This representation then serves as input for stakeholders to manually analyze legacy code, to identify obsolete and missing business cases, and, finally, to support the re-implementation of a new system. In this paper we briefly introduce the tool and present results of reverse engineering PL/SQL legacy code in the steel making domain. We show how stakeholders are supported in analyzing legacy code by means of general-purpose analysis techniques combined with domain-specific representations and conclude with some of the lessons learned.**

***Keywords*—reverse engineering; program comprehension; source code analysis**

- Changes in business cases over the last years were not reflected in verification logic of the legacy code.
- For a new production plant, additional requirements must be incorporated.
- The maintenance of the legacy programs was complicated by the retirement of original developers.
- Legacy code is not extensible in a safe and reliable way.
- Stakeholders estimated high effort for manual analysis of the legacy code.

The goal for the reverse engineering tool was to support stakeholders to comprehend the verification logic implemented in the legacy programs. Whereas, comprehension requires that stakeholders can (1) *identify* the business cases currently checked by the software as well as that stakeholders are able to (2) *extend* the verification logic with respect to new requirements.

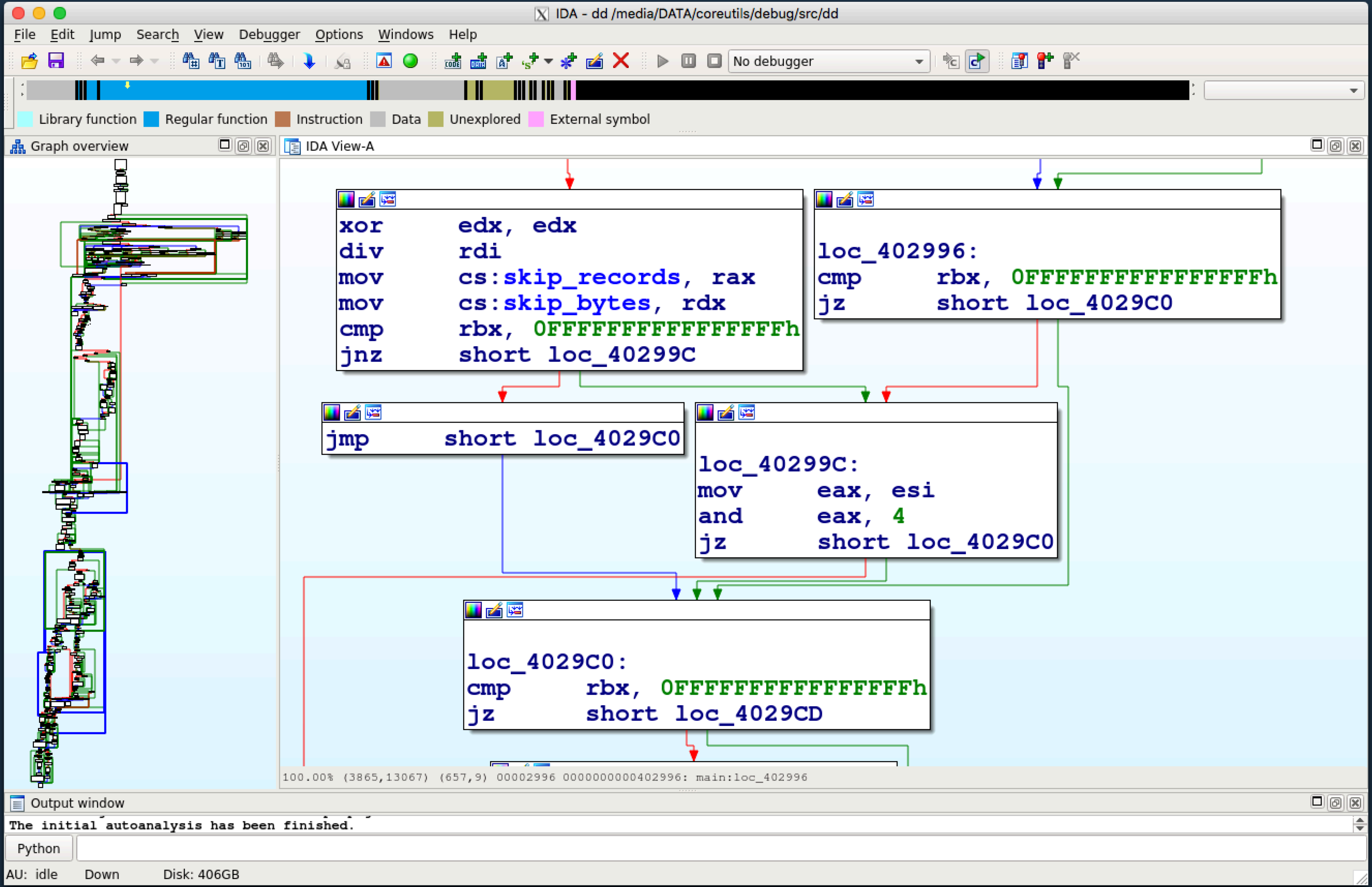
The contributions of this paper are:

Disassembler

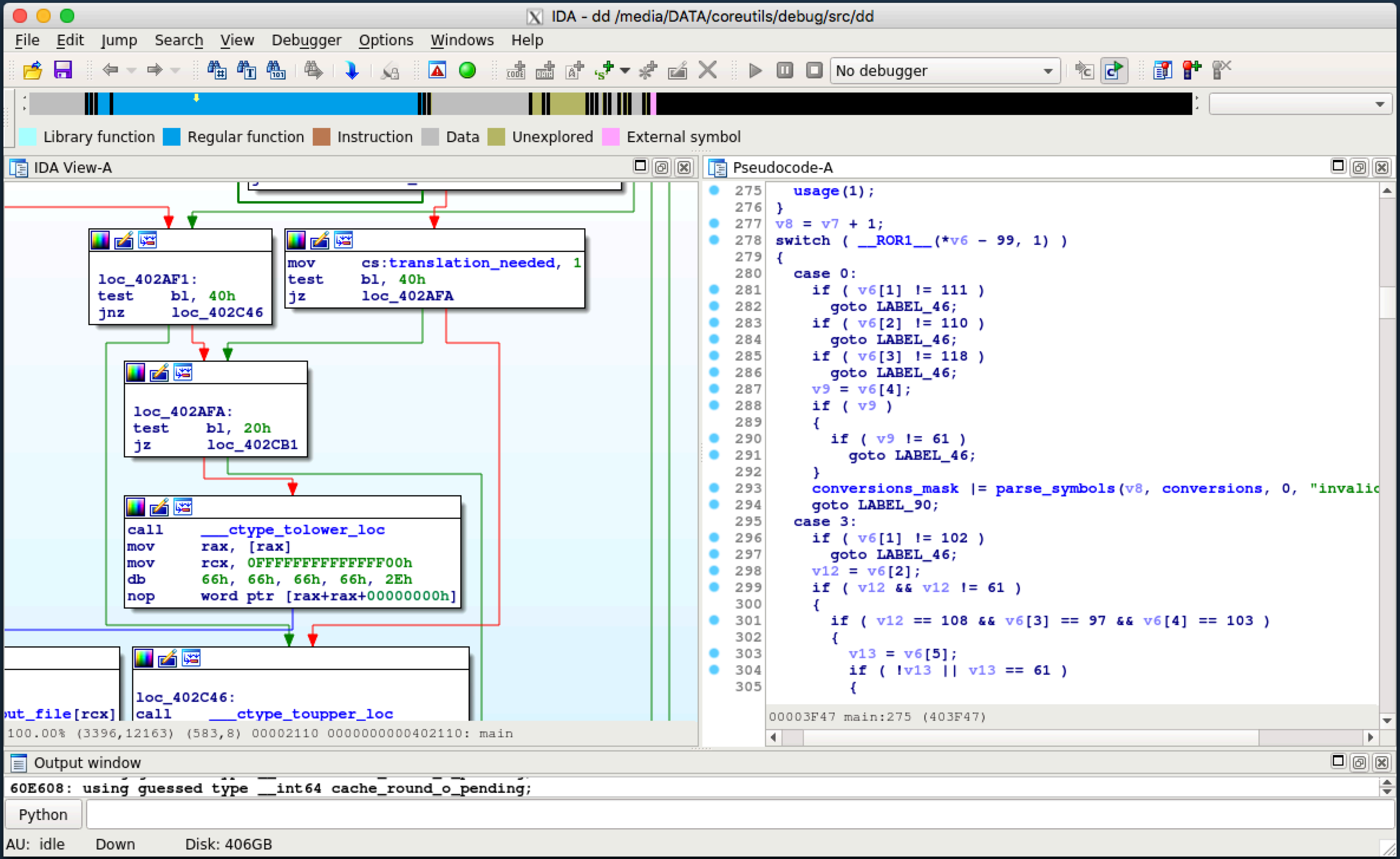
1. jlacomis@gs17931:~/Data/coreutils/debug/src (ssh)

```
40299c:   89 f0                   mov    %esi,%eax
40299e:   83 e0 04                and    $0x4,%eax
4029a1:   74 1d                   je     4029c0 <main+0x8b0>
4029a3:   31 d2                   xor    %edx,%edx
4029a5:   48 89 d8                mov    %rbx,%rax
4029a8:   48 f7 f7                div    %rdi
4029ab:   48 89 05 be b8 20 00    mov    %rax,0x20b8be(%rip)
4029b2:   48 89 15 ff ba 20 00    mov    %rdx,0x20baff(%rip)
4029b9:   4d 85 c0                test   %r8,%r8
4029bc:   75 14                   jne    4029d2 <main+0x8c2>
4029be:   eb 31                   jmp    4029f1 <main+0x8e1>
4029c0:   48 83 fb ff             cmp    $0xffffffffffffffff,%rbx
4029c4:   74 07                   je     4029cd <main+0x8bd>
4029c6:   48 89 1d a3 b8 20 00    mov    %rbx,0x20b8a3(%rip)
4029cd:   4d 85 c0                test   %r8,%r8
4029d0:   74 1f                   je     4029f1 <main+0x8e1>
4029d2:   89 c8                   mov    %ecx,%eax
4029d4:   83 e0 10                and    $0x10,%eax
4029d7:   74 18                   je     4029f1 <main+0x8e1>
```

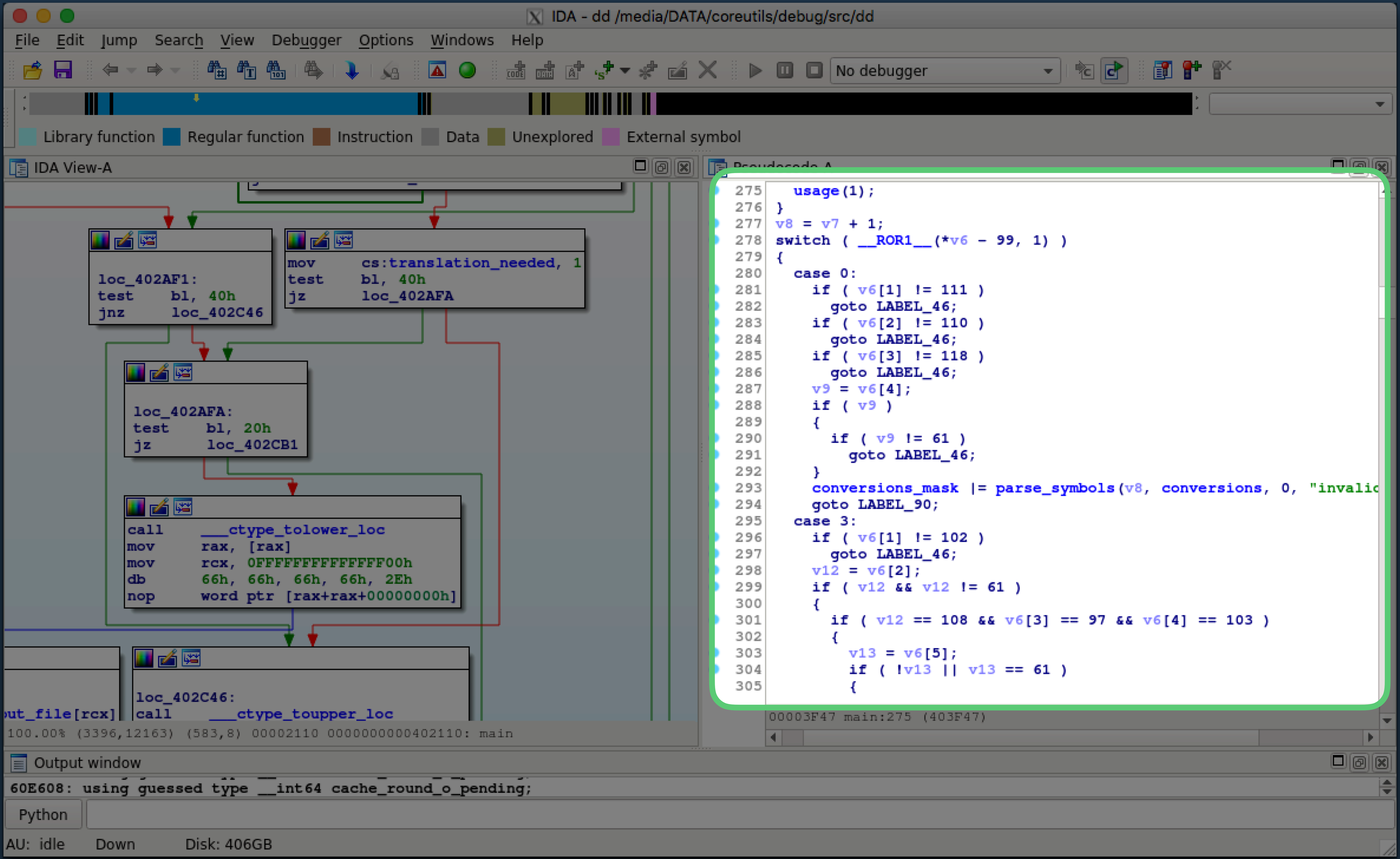
Disassembler



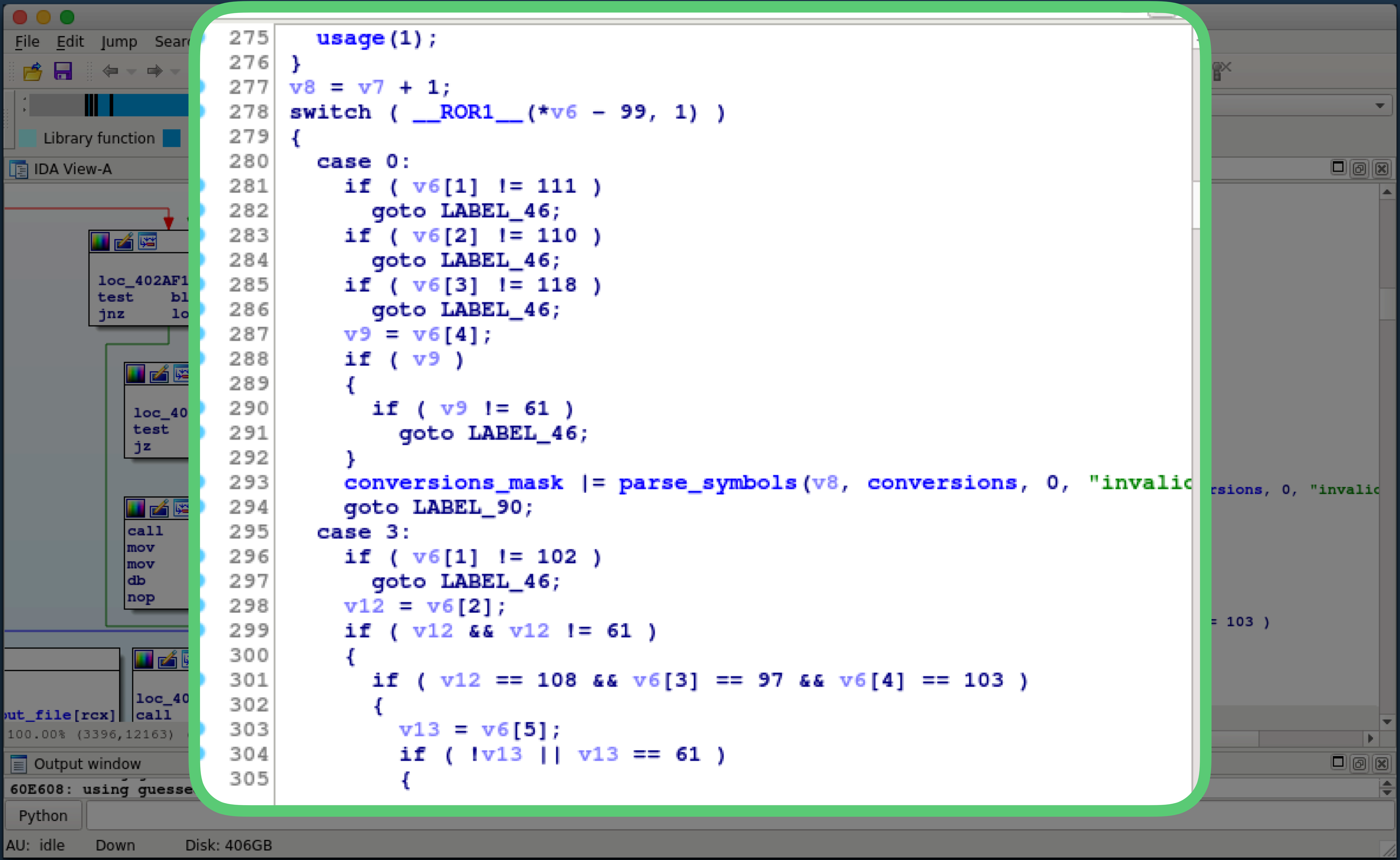
Decompiler


IDA - dd /media/DATA/coreutils/debug/src/dd
File Edit Jump Search View Debugger Options Windows Help
No debugger
Library function Regular function Instruction Data Unexplored External symbol
IDA View-A
loc_402AF1:
test bl, 40h
jnz loc_402C46
mov cs:translation_needed, 1
test bl, 40h
jz loc_402AFA
loc_402AFA:
test bl, 20h
jz loc_402CB1
call ___ctype_tolower_loc
mov rax, [rax]
mov rcx, 0FFFFFFFFFFFFFF00h
db 66h, 66h, 66h, 66h, 2Eh
nop word ptr [rax+rax+00000000h]
loc_402C46:
call ___ctype_toupper_loc
100.00% (3396,12163) (583,8) 00002110 0000000000402110: main
Pseudocode-A
usage(1);
}
v8 = v7 + 1;
switch (__ROR1__(*v6 - 99, 1))
{
case 0:
if (v6[1] != 111)
goto LABEL_46;
if (v6[2] != 110)
goto LABEL_46;
if (v6[3] != 118)
goto LABEL_46;
v9 = v6[4];
if (v9)
{
if (v9 != 61)
goto LABEL_46;
}
conversions_mask |= parse_symbols(v8, conversions, 0, "invalid
goto LABEL_90;
case 3:
if (v6[1] != 102)
goto LABEL_46;
v12 = v6[2];
if (v12 && v12 != 61)
{
if (v12 == 108 && v6[3] == 97 && v6[4] == 103)
{
v13 = v6[5];
if (!v13 || v13 == 61)
{
00003F47 main:275 (403F47)
Output window
60E608: using guessed type __int64 cache_round_o_pending;
Python
AU: idle Down Disk: 406GB

Decompiler


IDA - dd /media/DATA/coreutils/debug/src/dd
File Edit Jump Search View Debugger Options Windows Help
No debugger
Library function
Regular function
Instruction
Data
Unexplored
External symbol
IDA View-A
loc_402AF1:
test bl, 40h
jnz loc_402C46
mov cs:translation_needed, 1
test bl, 40h
jz loc_402AFA
loc_402AFA:
test bl, 20h
jz loc_402CB1
call ___ctype_tolower_loc
mov rax, [rax]
mov rcx, 0FFFFFFFFFFFFFF00h
db 66h, 66h, 66h, 66h, 2Eh
nop word ptr [rax+rax+00000000h]
loc_402C46:
call ___ctype_toupper_loc
100.00% (3396,12163) (583,8) 00002110 0000000000402110: main
usage(1);
}
v8 = v7 + 1;
switch (__ROR1__(*v6 - 99, 1))
{
case 0:
if (v6[1] != 111)
goto LABEL_46;
if (v6[2] != 110)
goto LABEL_46;
if (v6[3] != 118)
goto LABEL_46;
v9 = v6[4];
if (v9)
{
if (v9 != 61)
goto LABEL_46;
}
conversions_mask |= parse_symbols(v8, conversions, 0, "invalid
goto LABEL_90;
case 3:
if (v6[1] != 102)
goto LABEL_46;
v12 = v6[2];
if (v12 && v12 != 61)
{
if (v12 == 108 && v6[3] == 97 && v6[4] == 103)
{
v13 = v6[5];
if (!v13 || v13 == 61)
{
00003F47 main:275 (403F47)
Output window
60E608: using guessed type __int64 cache_round_o_pending;
Python
AU: idle
Down
Disk: 406GB

Decompiler



File Edit Jump Sear
Library function
IDA View-A
usage(1);
}
v8 = v7 + 1;
switch (__ROR1__(*v6 - 99, 1))
{
case 0:
if (v6[1] != 111)
goto LABEL_46;
if (v6[2] != 110)
goto LABEL_46;
if (v6[3] != 118)
goto LABEL_46;
v9 = v6[4];
if (v9)
{
if (v9 != 61)
goto LABEL_46;
}
conversions_mask |= parse_symbols(v8, conversions, 0, "invalic
goto LABEL_90;
case 3:
if (v6[1] != 102)
goto LABEL_46;
v12 = v6[2];
if (v12 && v12 != 61)
{
if (v12 == 108 && v6[3] == 97 && v6[4] == 103)
{
v13 = v6[5];
if (!v13 || v13 == 61)
{
Output window
60E608: using guesse
Python
AU: idle Down Disk: 406GB

The problem:

Decompilers are typically unable to assign meaningful names to variables

Today

Decompiler output → Refactored decompiler output

```
void *file_mmap(int V1, int V2)
{
  void *V3;
  V3 = mmap(0, V2, 1, 2, V1, 0);
  if (V3 == (void *) -1) {
    perror("mmap");
    exit(1);
  }
  return V3;
}
```

```
void *file_mmap(int fd, int size)
{
  void *ret;
  ret = mmap(0, size, 1, 2, fd, 0);
  if (ret == (void *) -1) {
    perror("mmap");
    exit(1);
  }
  return ret;
}
```

Today

Decompiler output → Refactored decompiler output

```c
void *file_mmap(int V1 int V2)
{
  void *V3;
  V3 = mmap(0, V2, 1, 2, V1 0);
  if (V3 == (void *) -1) {
    perror("mmap");
    exit(1);
  }
  return V3;
}
```

```c
void *file_mmap(int fd int size)
{
  void *ret;
  ret = mmap(0, size, 1, 2, fd 0);
  if (ret == (void *) -1) {
    perror("mmap");
    exit(1);
  }
  return ret;
}
```

Today

Decompiler output → Refactored decompiler output

```c
void *file_mmap(int V1, int V2)
{
  void *V3;
  V3 = mmap(0, V2, 1, 2, V1, 0);
  if (V3 == (void *) -1) {
    perror("mmap");
    exit(1);
  }
  return V3;
}
```

```c
void *file_mmap(int fd, int size)
{
  void *ret;
  ret = mmap(0, size, 1, 2, fd, 0);
  if (ret == (void *) -1) {
    perror("mmap");
    exit(1);
  }
  return ret;
}
```

Today

Decompiler output → Refactored decompiler output

```c
void *file_mmap(int V1, int V2)
{
  void *V3
  V3 = mmap(0, V2, 1, 2, V1, 0);
  if (V3 == (void *) -1) {
    perror("mmap");
    exit(1);
  }
  return V3
}
```

```c
void *file_mmap(int fd, int size)
{
  void *ret
  ret = mmap(0, size, 1, 2, fd, 0);
  if (ret == (void *) -1) {
    perror("mmap");
    exit(1);
  }
  return ret
}
```

up to **74%**

recovery of original source code names
on an open-source GitHub corpus

Why does it work?

Key principle: Software is "natural"

(2012 International Conference on Software Engineering)

On the Naturalness of Software

Abram Hindle, Earl Barr, Zhendong Su
Dept. of Computer Science
University of California at Davis
Davis, CA 95616 USA
{ajhindle,barr,su}@cs.ucdavis.edu

Mark Gabel
Dept. of Computer Science
The University of Texas at Dallas
Richardson, TX 75080 USA
mark.gabel@utdallas.edu

Prem Devanbu
Dept. of Computer Science
University of California at Davis
Davis, CA 95616 USA
devanbu@cs.ucdavis.edu

***Abstract*—Natural languages like English are rich, complex, and powerful. The highly creative and graceful use of languages like English and Tamil, by masters like Shakespeare and Avvaiyar, can certainly delight and inspire. But in practice, given cognitive constraints and the exigencies of daily life, most human utterances are far simpler and much more repetitive and predictable. In fact, these utterances can be very usefully modeled using modern statistical methods. This fact has led to the phenomenal success of statistical approaches to speech recognition, natural language translation, question-answering, and text mining and comprehension.**

We begin with the conjecture that most software is also

efforts in the 1960s. In the ’70s and ’80s, the field was re-animated with ideas from logic and formal semantics, which still proved too cumbersome to perform practical tasks at scale. Both these approaches essentially dealt with NLP from first principles—addressing *language*, in all its rich theoretical glory, rather than examining corpora of actual *utterances*, *i.e.*, what people actually write or say. In the 1980s, a fundamental shift to *corpus-based, statistically rigorous* methods occurred. The availability of large, on-line corpora of natural language text, including “aligned” text with translations in multiple

Recall

Decompiler output → Refactored decompiler output

```c
void *file_mmap(int V1, int V2)
{
  void *V3;
  V3 = mmap(0, V2, 1, 2, V1, 0);
  if (V3 == (void *) -1) {
    perror("mmap");
    exit(1);
  }
  return V3;
}
```

```c
void *file_mmap(int fd, int size)
{
  void *ret;
  ret = mmap(0, size, 1, 2, fd, 0);
  if (ret == (void *) -1) {
    perror("mmap");
    exit(1);
  }
  return ret;
}
```

Idea:

Learn typical variable names in a given context from examples ... many many examples

If software is repetitive, so are names

```
int main(int  ?
```

Idea:

Learn typical variable names in a given context from examples ... many many examples

If software is repetitive, so are names

```
int main(int banana
```

Idea:

Learn typical variable names in a given context from examples … many many examples

If software is repetitive, so are names

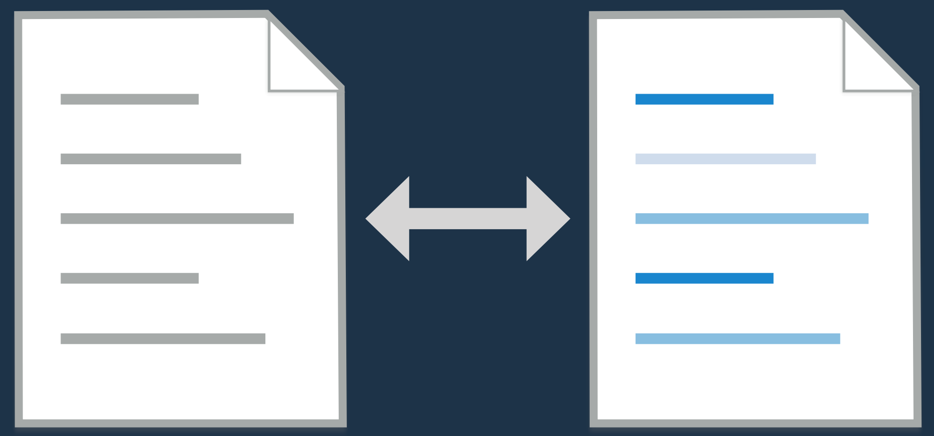
```
int main(int argc
```

Good news:

We can generate arbitrarily many examples

GitHub + Compiler/Decompiler + Time

Source code with meaningful names



Decompiler output with placeholder names

Corpus Construction

Original Source

emacs

```c
#include <stdio.h>

int main() {
  int x = 0;
  int y = 0;
  while (x < 100) {
    printf("%d\n", x);
    x++;
  }
  return y;
}
```

-UUU:**--F1 count.c All (11,1)

Decompiled Code

emacs

```c
#include <stdio.h>

int main() {
  int v1 = 0;
  int v2 = 0;
  while (v1 < 100) {
    printf("%d\n", v1);
    v1++;
  }
  return v2;
}
```

-UUU:**--F1 count.c All (11,1)

Corpus Construction

Original Source

emacs ⌥⌘1

```c
#include <stdio.h>

int main() {
  int x = 0;
  int y = 0;
  while (x < 100) {
    printf("%d\n", x);
    x++;
  }
  return y;
}
```

-UUU:**--F1 count.c All (11,1)

Decompiled Code

emacs ⌥⌘1

```c
#include <stdio.h>

int main() {
  int v1 = 0;
  int v2 = 0;
  while (v1 < 100) {
    printf("%d\n", v1);
    v1++;
  }
  return v2;
}
```

-UUU:**--F1 count.c All (11,1)

Difficulty: Decompilation Changes Structure

Original Source

emacs ⌥⌘1

```c
#include <stdio.h>

int main() {
  int x = 0;
  int y = 0;
  while (x < 100) {
    printf("%d\n", x);
    x++;
  }
  return y;
}
```

```
-UUU:**--F1  count.c       All (11,1)
```

Decompiled Code

Pseudocode-A

```c
int __cdecl main(int argc, const char **arg
{
  signed int i; // [rsp+8h] [rbp-8h]

  for ( i = 0; i < 100; ++i )
    printf("%d\n", (unsigned int)i, envp);
  return 0;
}
```

```
00000F44 _main:8 (100000F44)
```

Difficulty: Decompilation Changes Structure

Original Source

emacs

```c
#include <stdio.h>
int main() {
  int x = 0;
  int y = 0;
  while (x < 100) {
    printf("%d\n", x);
    x++;
  }
  return y;
}
```

```
-UUU:**--F1  count.c        All (11,1)
```

Decompiled Code

```c
int __cdecl main(int argc,
  signed int i; // [rsp+8h] [rbp-8h]

  for ( i = 0; i < 100; ++i )
    printf("%d\n", (unsigned int)i, envp);
  return 0;
}
```

```
00000F44 _main:8 (100000F44)
```

Different function signatures

Difficulty: Decompilation Changes Structure

Original Source

```
emacs
#include <stdio.h>

int x = 0;
int y = 0;
) {
    printf("%d\n", x);
    x++;
  }
  return y;
}
-UUU:**--F1  count.c        All (11,1)
```

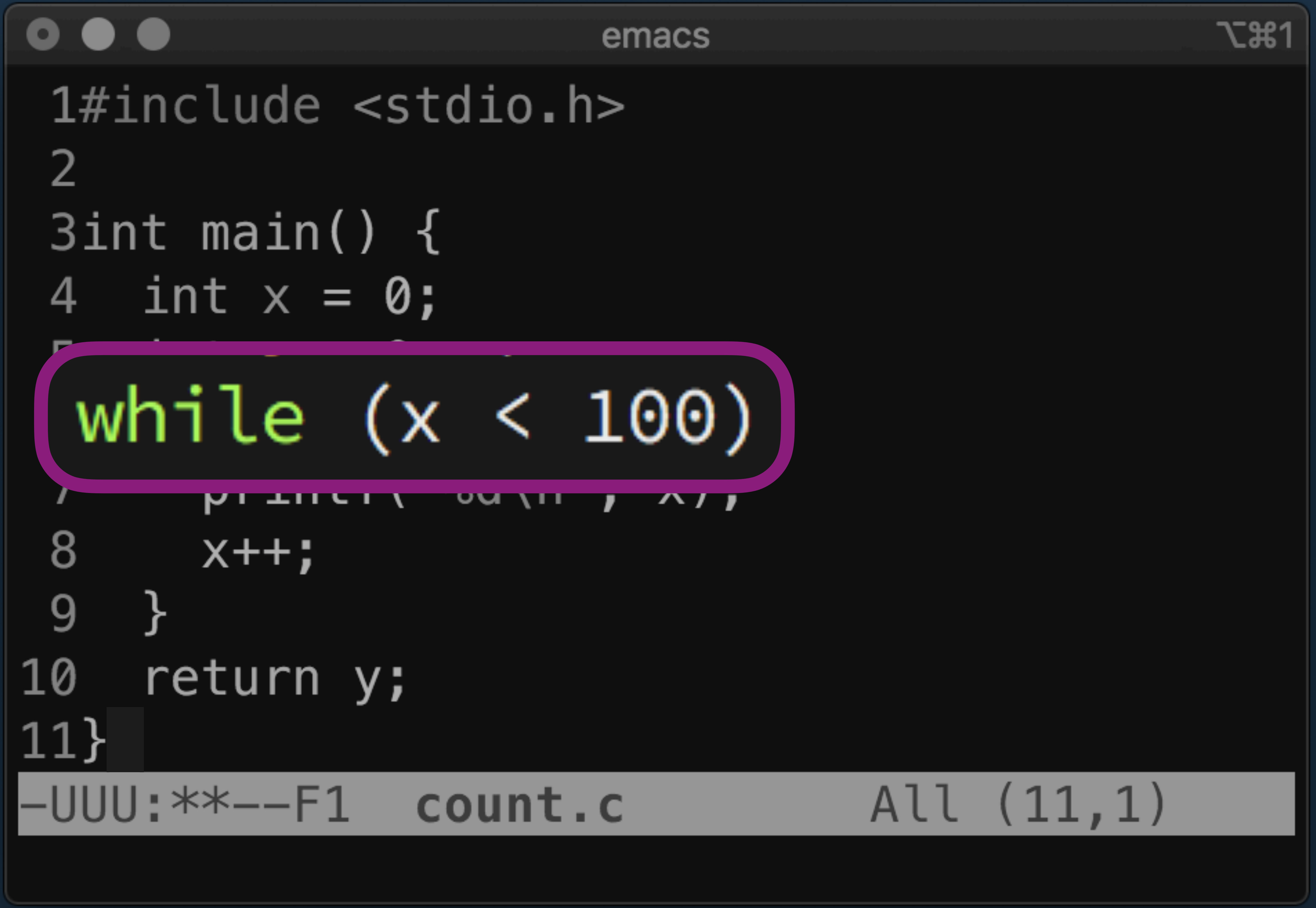
Decompiled Code

```
Pseudocode-A
int   cdecl main(int argc, const char **arg
signed int i;
8h] [rbp-8h]
  for ( i = 0; i < 100; ++i )
    printf("%d\n", (unsigned int)i, envp);
  return 0;
}
00000F44 _main:8 (100000F44)
```

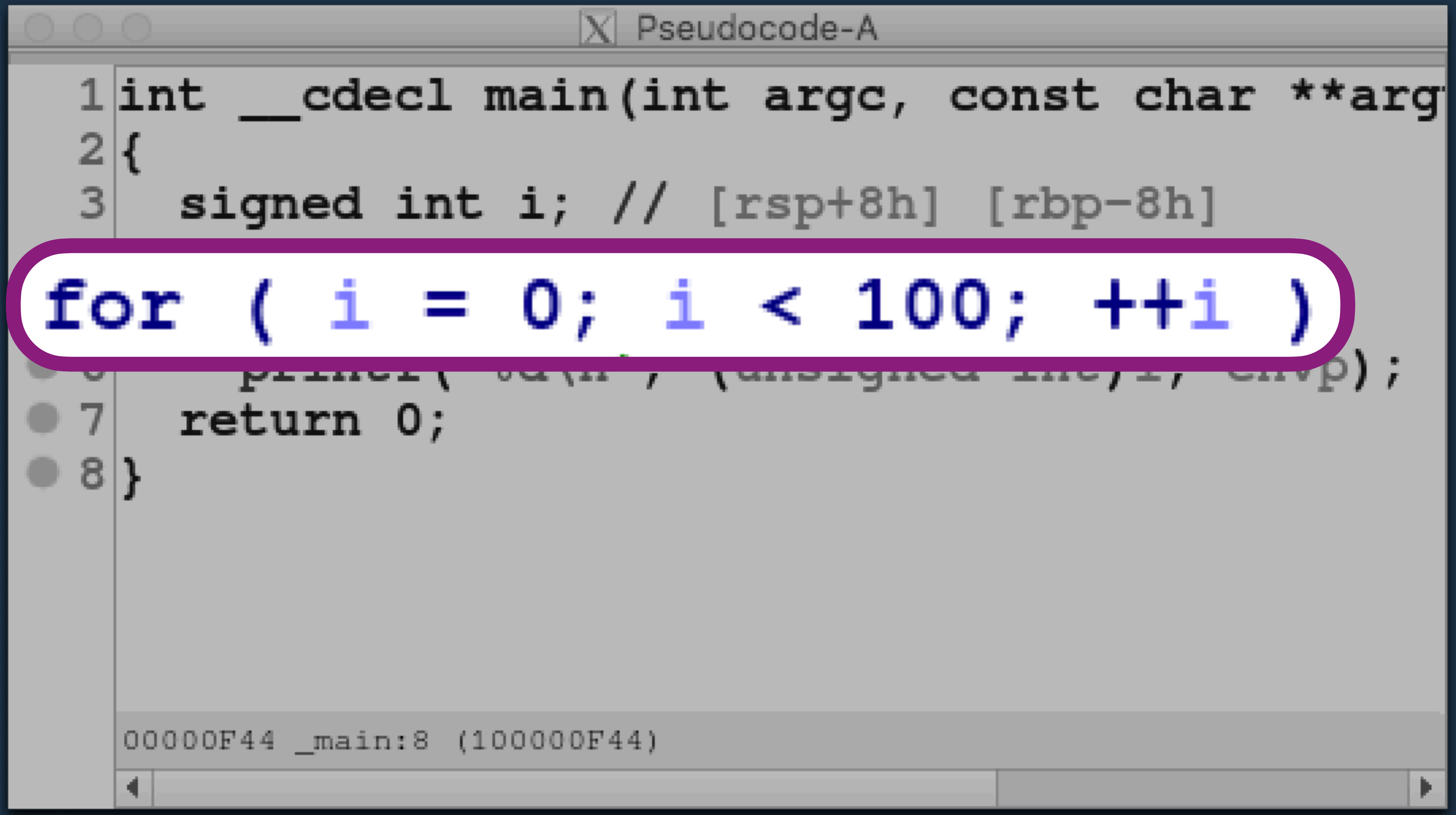
Different numbers of variables

Difficulty: Decompilation Changes Structure

Original Source



Decompiled Code



Different types of loops

Difficulty: Decompilation Changes Structure

Original Source

emacs ⌥⌘1

```c
#include <stdio.h>

int main() {
  int x = 0;
  int y = 0;
  while (x < 100) {
    printf("%d\n", x);
    x++;
  }
  return y;
}
```

-UUU:**--F1 count.c   All (11,1)

Decompiled Code

Pseudocode-A

```c
int __cdecl main(int argc, const char **arg
{
  signed int i; // [rsp+8h] [rbp-8h]

  for ( i = 0; i < 100; ++i )
    printf("%d\n", (unsigned int)i, envp);
  return 0;
}
```

00000F44 _main:8 (100000F44)

Alignment

Two different loops.

```
int i, z;
for (i = 0; i < 10; i++) {
  z += i;
}
```

```
int v1, v2;
v1 = 0;
while (v1 < 10) {
  v2 += v1;
  v1++;
}
```

Alignment

Two different loops.

```
int i, z;
for (i = 0; i < 10; i++) {
  z += i;
}
```

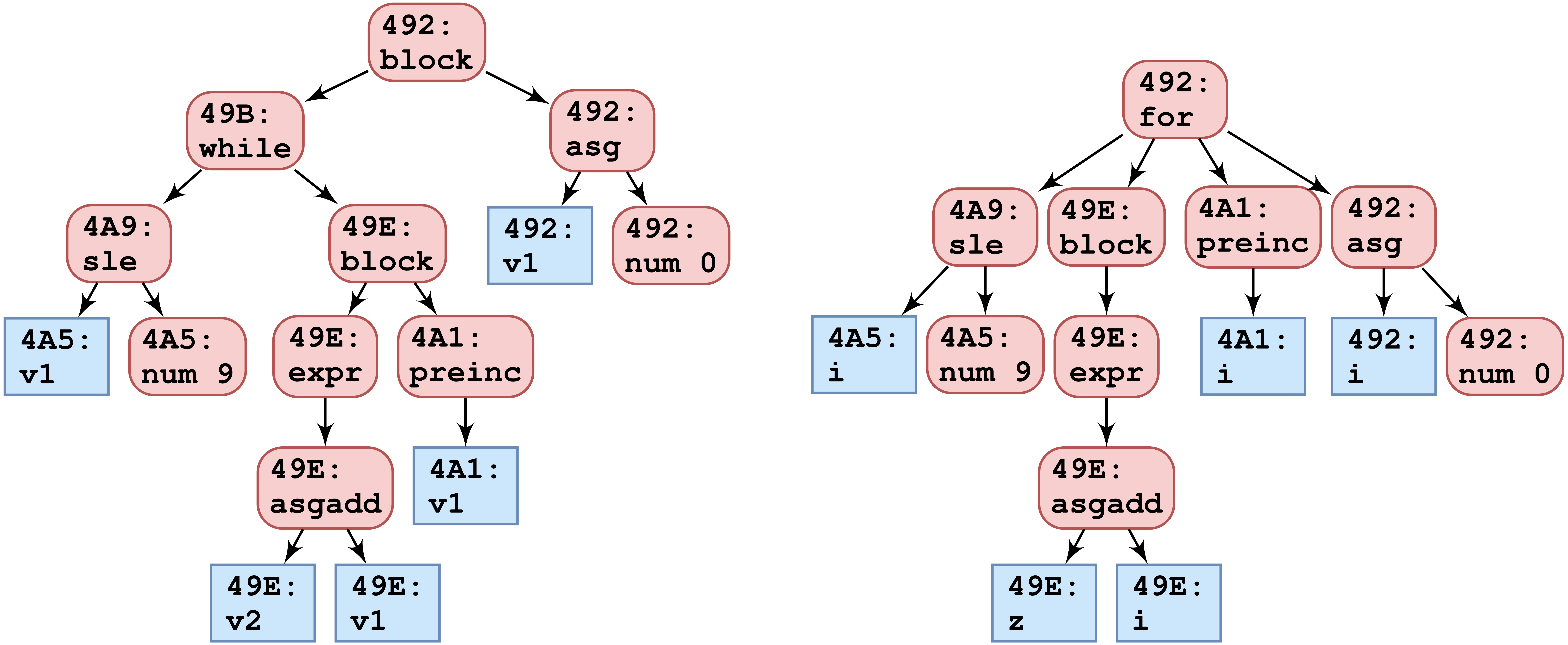
```
int v1, v2;
v1 = 0;
while (v1 < 10) {
  v2 += v1;
  v1++;
}
```

Same assembly code.

```
var1 = dword ptr -8
var2 = dword ptr -4
   ;;...
   mov [rbp+var2], 0
   jmp loc_4a5
loc_49B:
   mov eax, [rbp+var2]
   add [rbp+var1], eax
   add [rbp+var2], 1
loc_4a5:
   cmp [rbp+var2], 9
   jle loc_49b
```

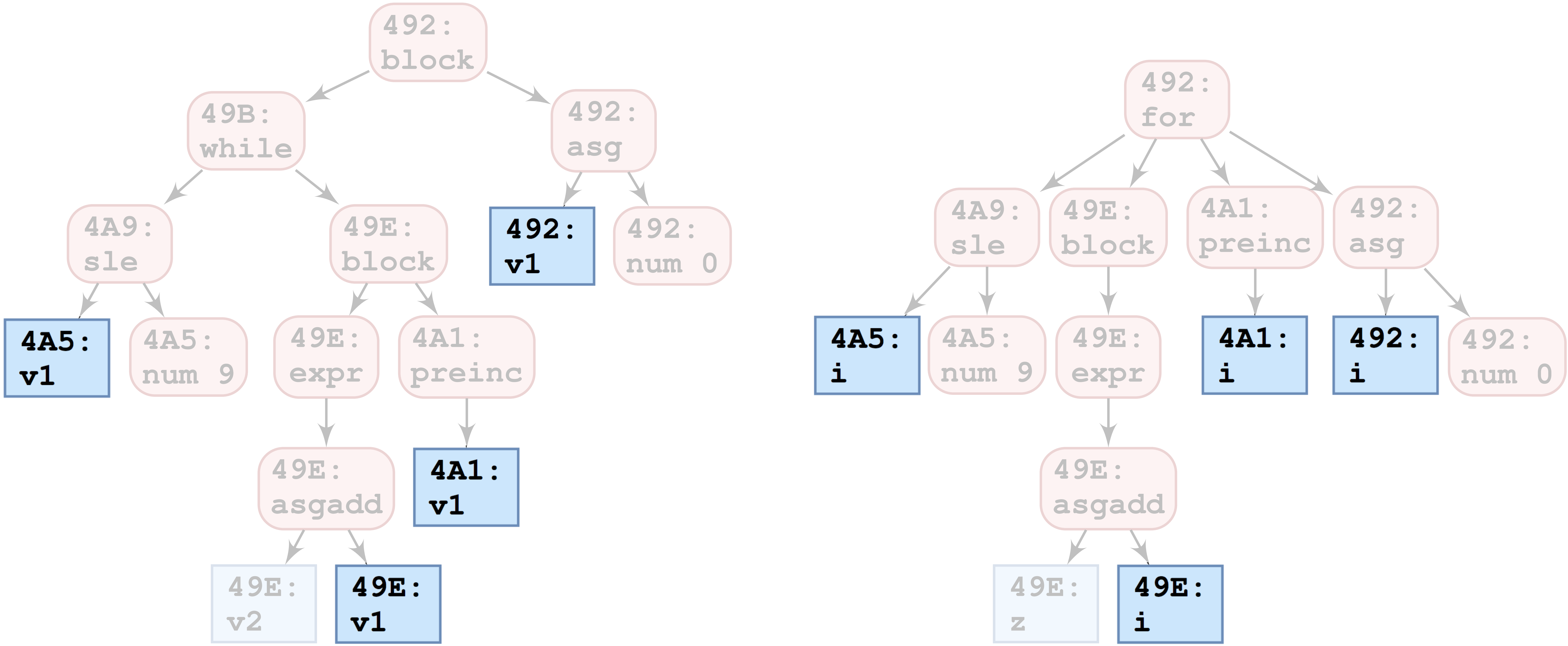
Alignment

Key insight: Operations on variables and their offsets are the same



Alignment

Key insight: Operations on variables and their offsets are the same



Learning from examples

Decompiled
Identifier
Renaming
Engine

Recall:

Names are repetitive in a given context

```
int main(int argc
```

Running example

```
char* mystrcopy(char *VAR1, char *VAR2){
  char *result;
  if (VAR1 && VAR2)
    result = strcopy(VAR1, VAR2);
  else
    result = 0LL;
  return result;
}
```

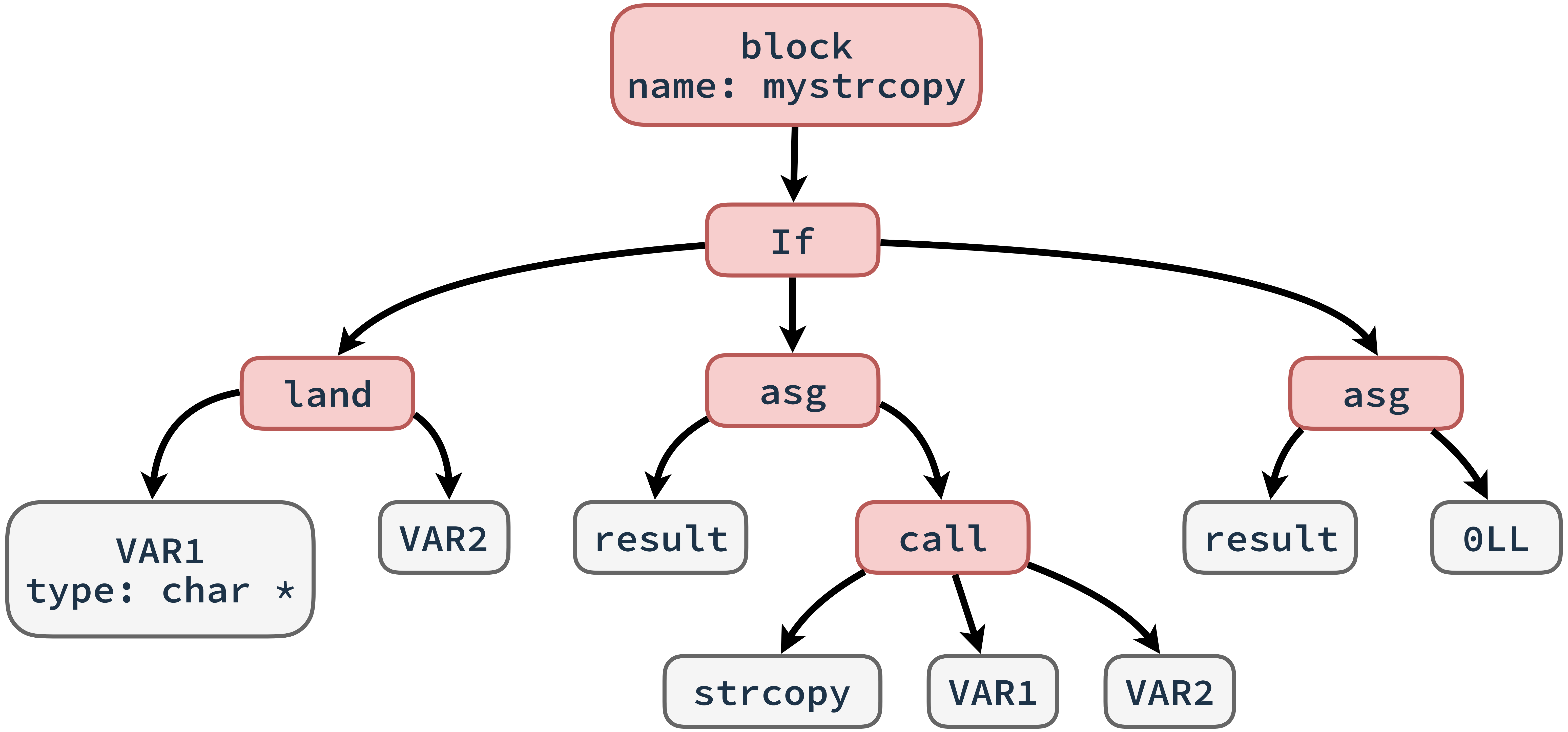
Code can be a sequence of *lexical tokens …*

… or a *syntax tree*

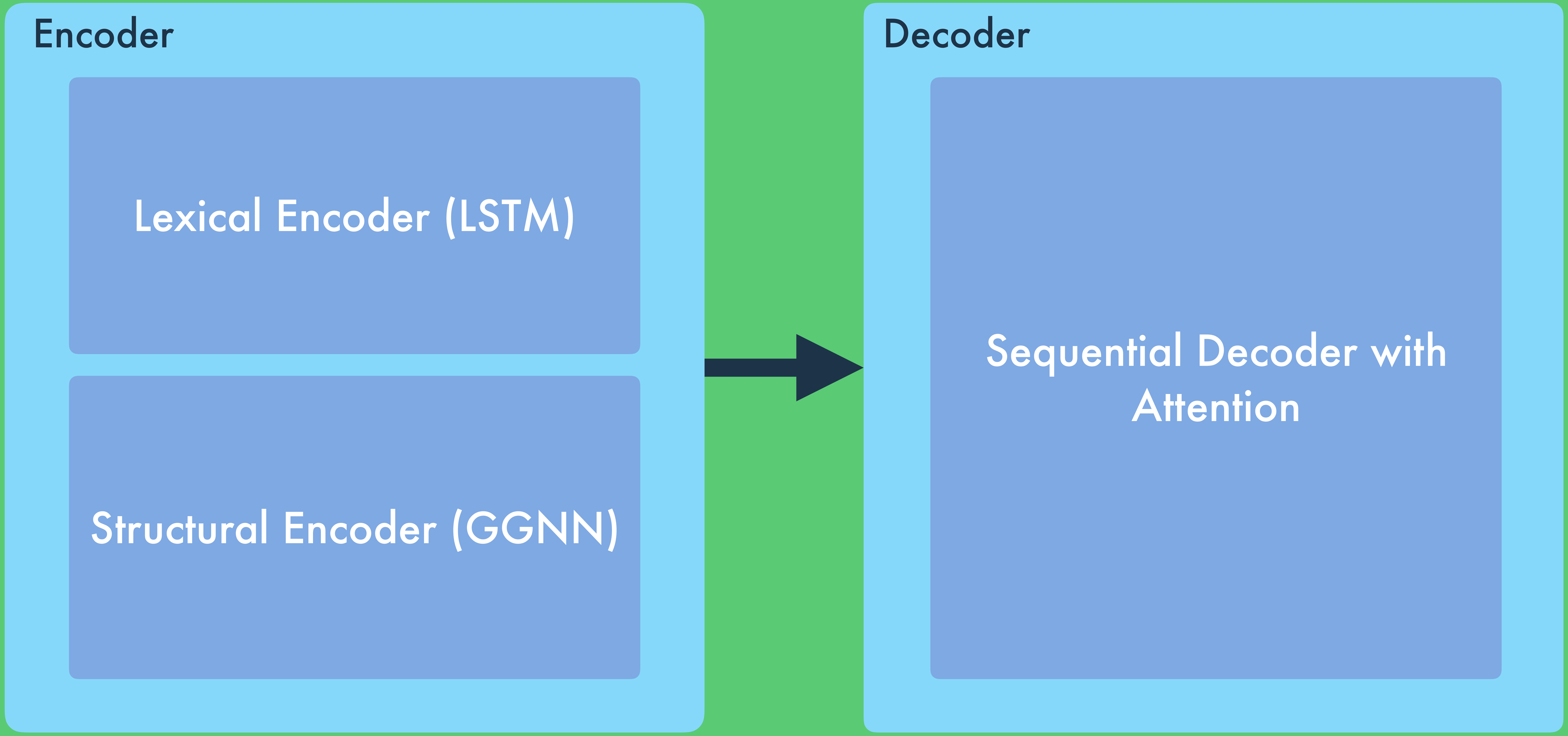
```c
char* mystrcopy(char *VAR1, char *VAR2){
  char *result;
  if (VAR1 && VAR2)
    result = strcopy(VAR1, VAR2);
  else

    result = 0LL;
  return result;
}
```

```
char * mystrcopy
( char * VAR1 , ...
```



DIRE Neural Architecture



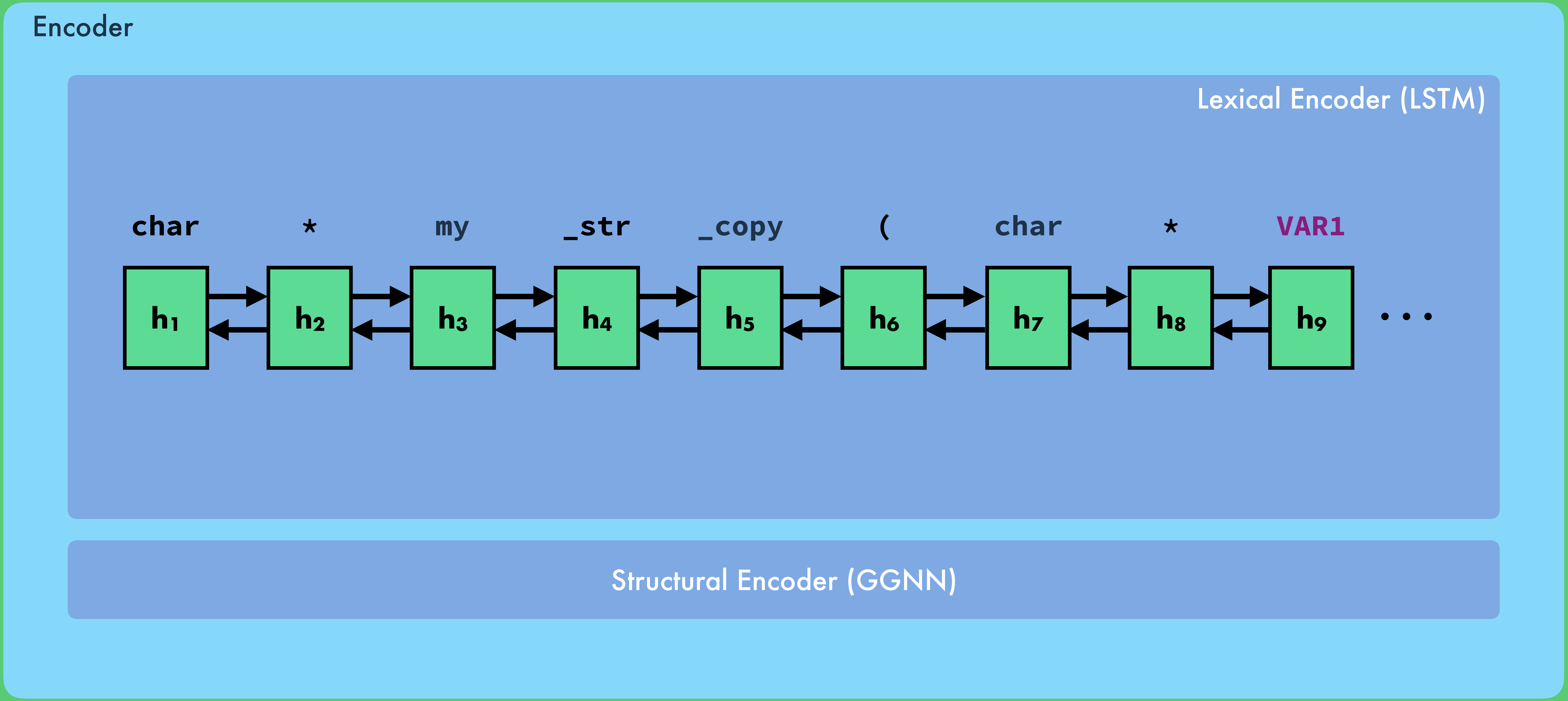
DIRE Neural Architecture

Encoder

Lexical Encoder (LSTM)

Structural Encoder (GGNN)

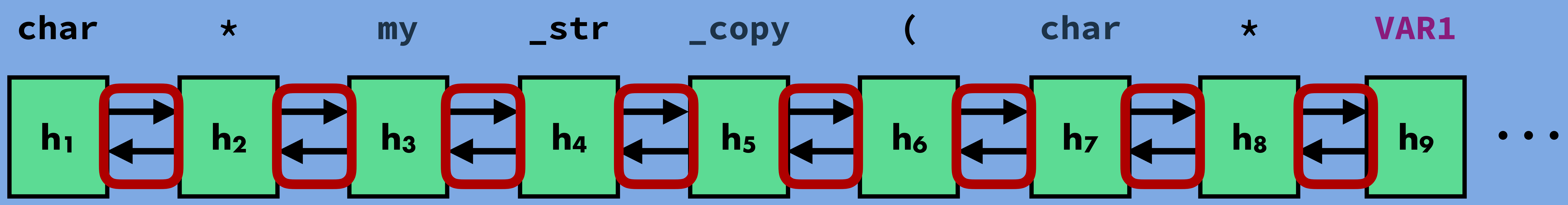
DIRE Neural Architecture



DIRE Neural Architecture

Encoder

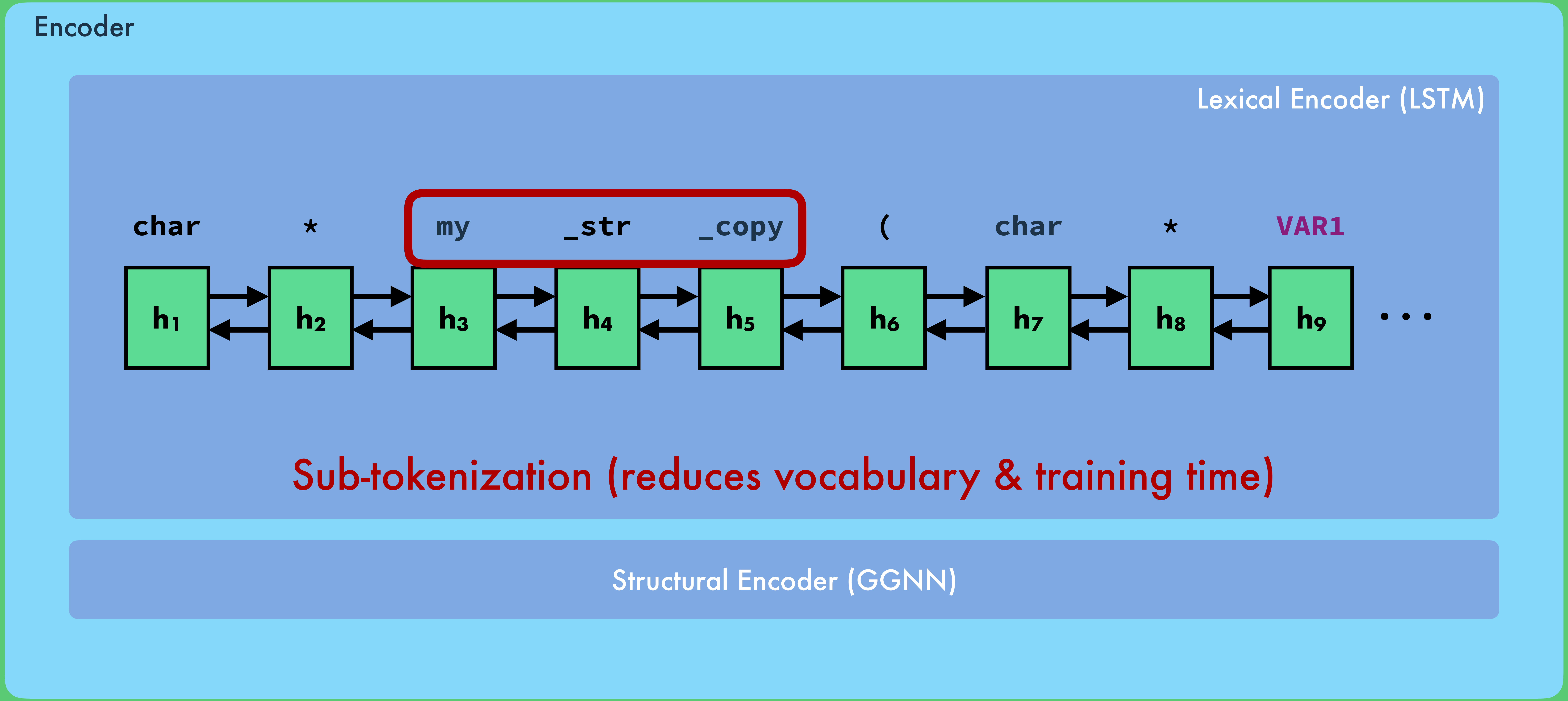
Lexical Encoder (LSTM)



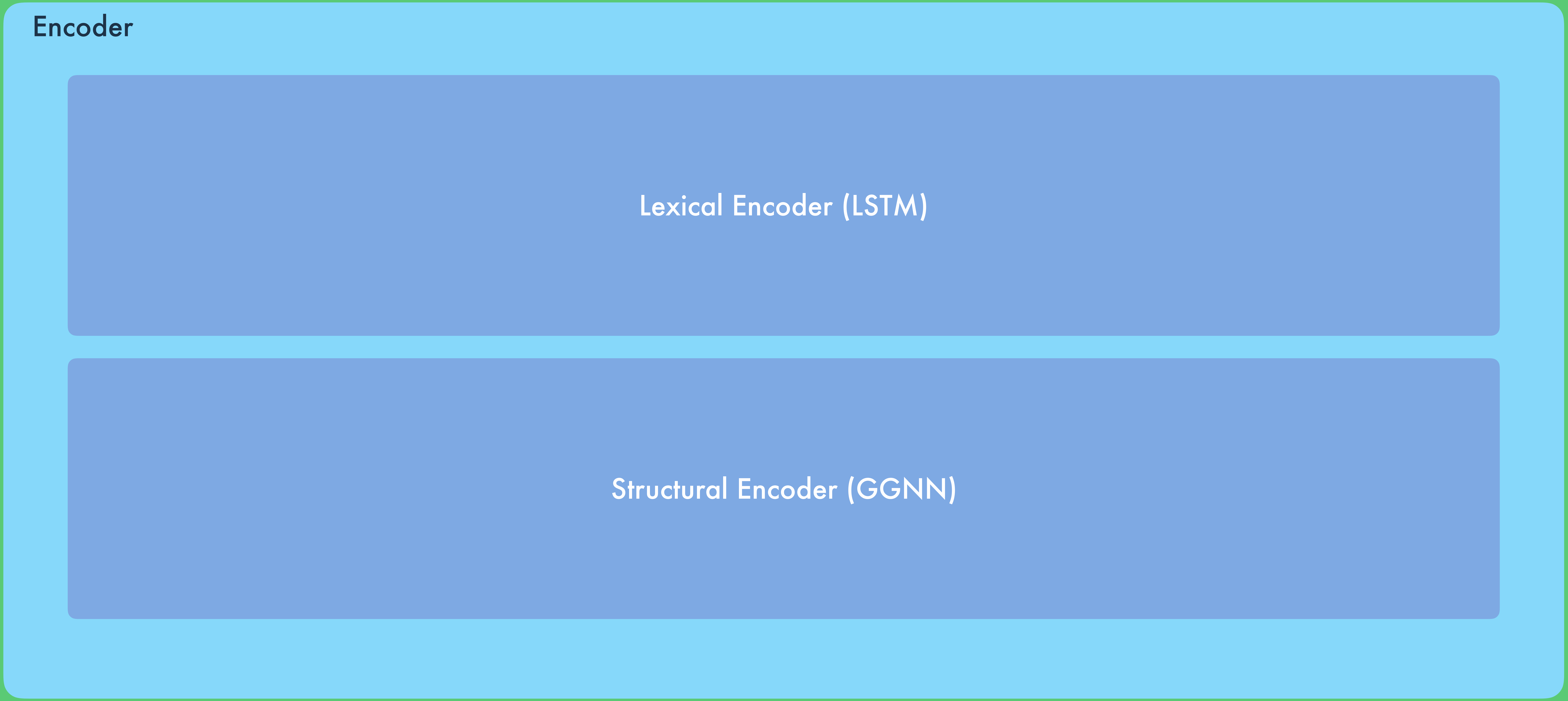
Look behind and ahead for more context

Structural Encoder (GGNN)

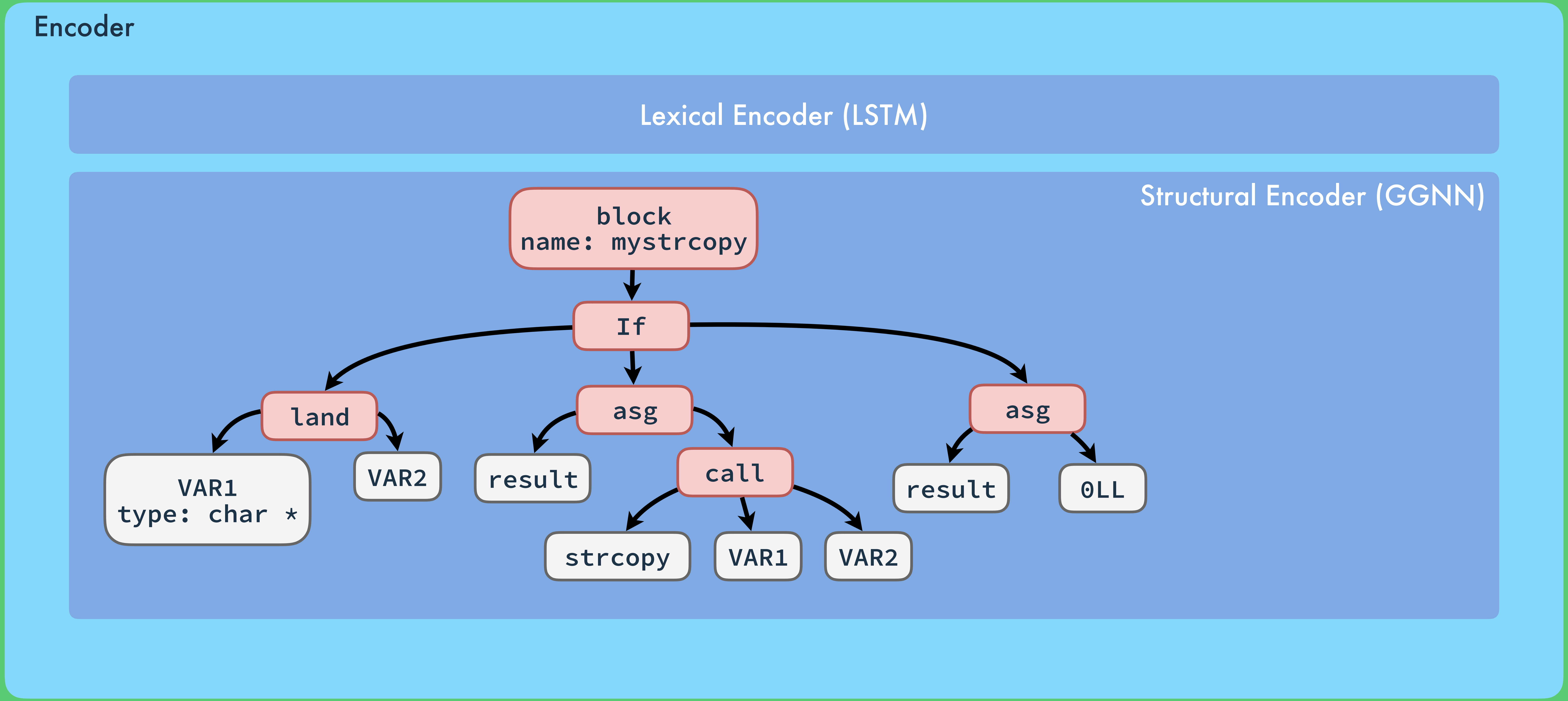
DIRE Neural Architecture



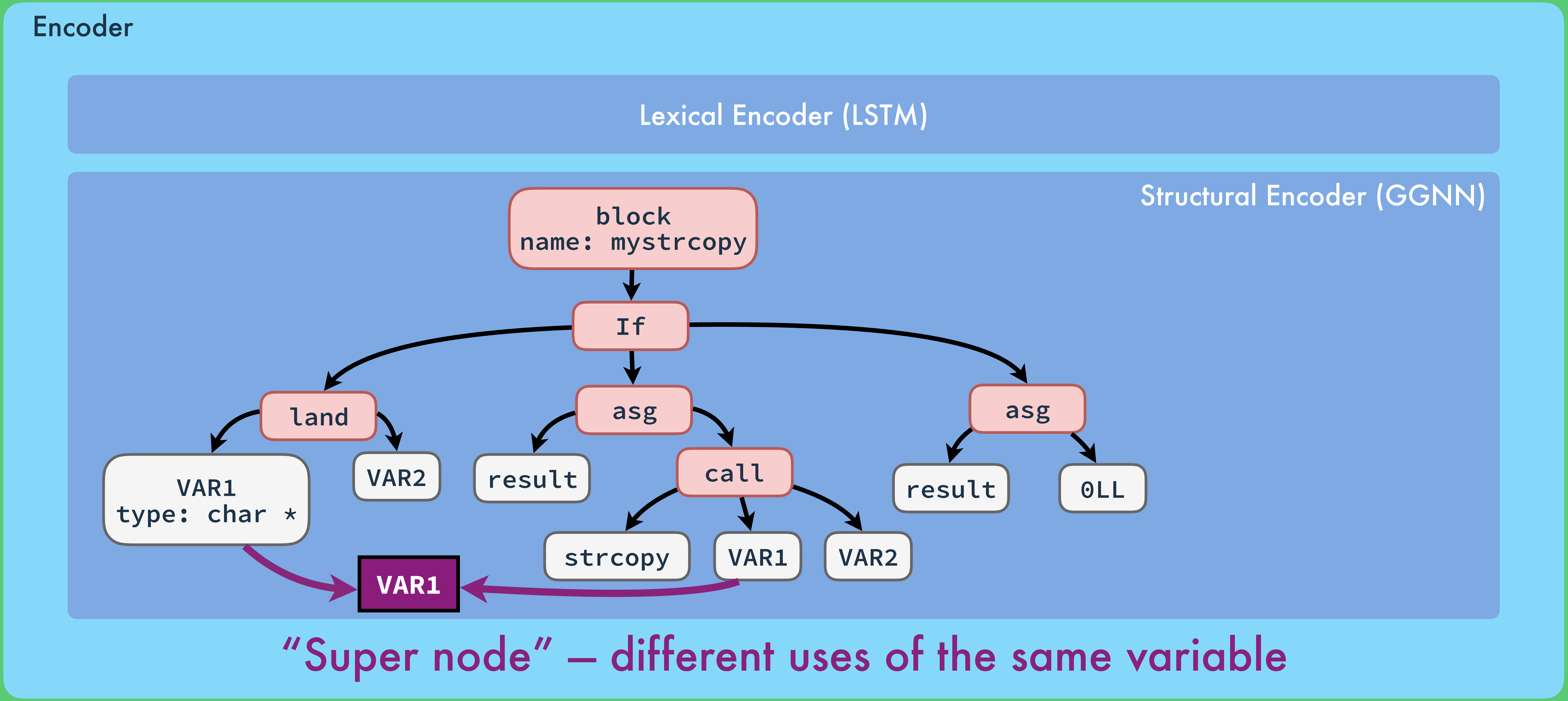
DIRE Neural Architecture



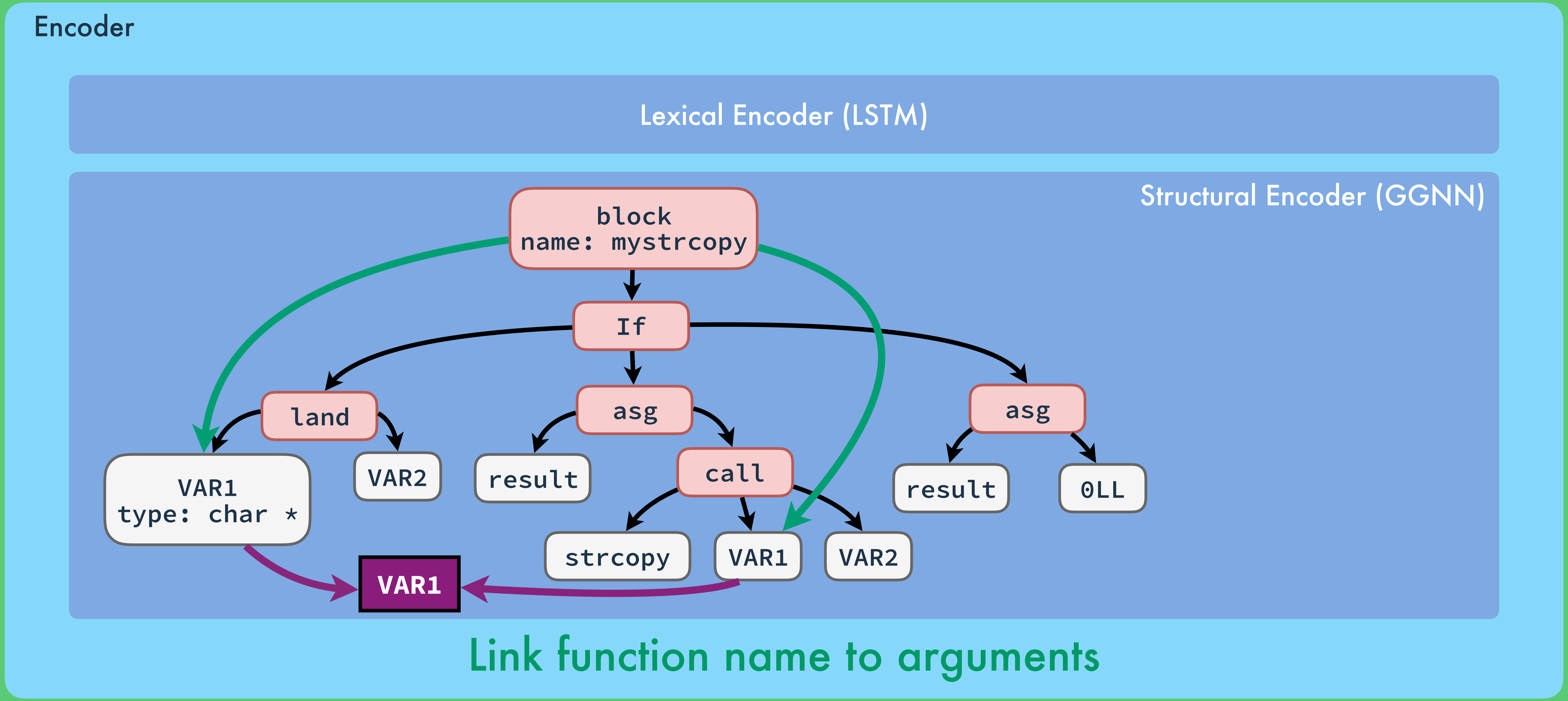
DIRE Neural Architecture



DIRE Neural Architecture



DIRE Neural Architecture



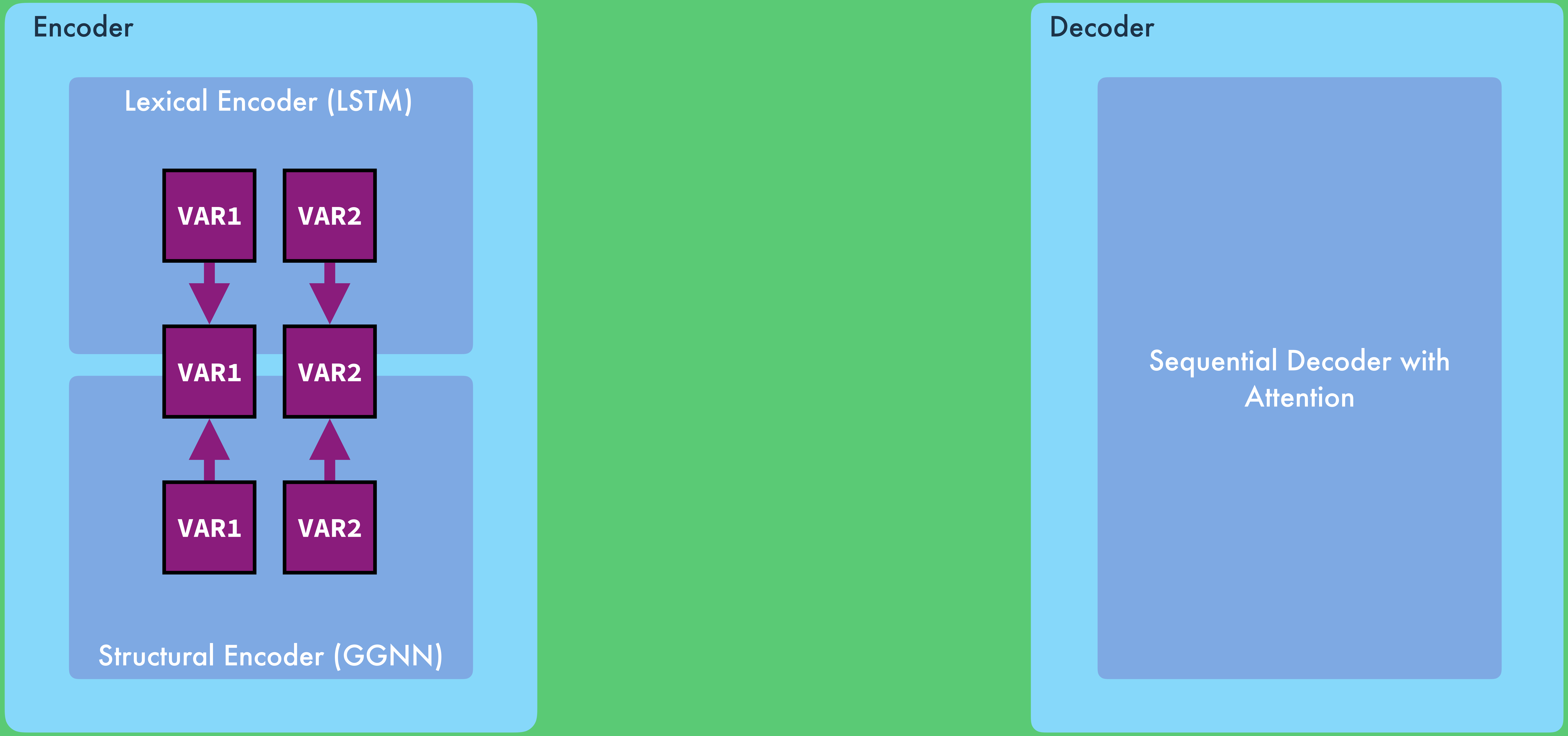
DIRE Neural Architecture

Encoder

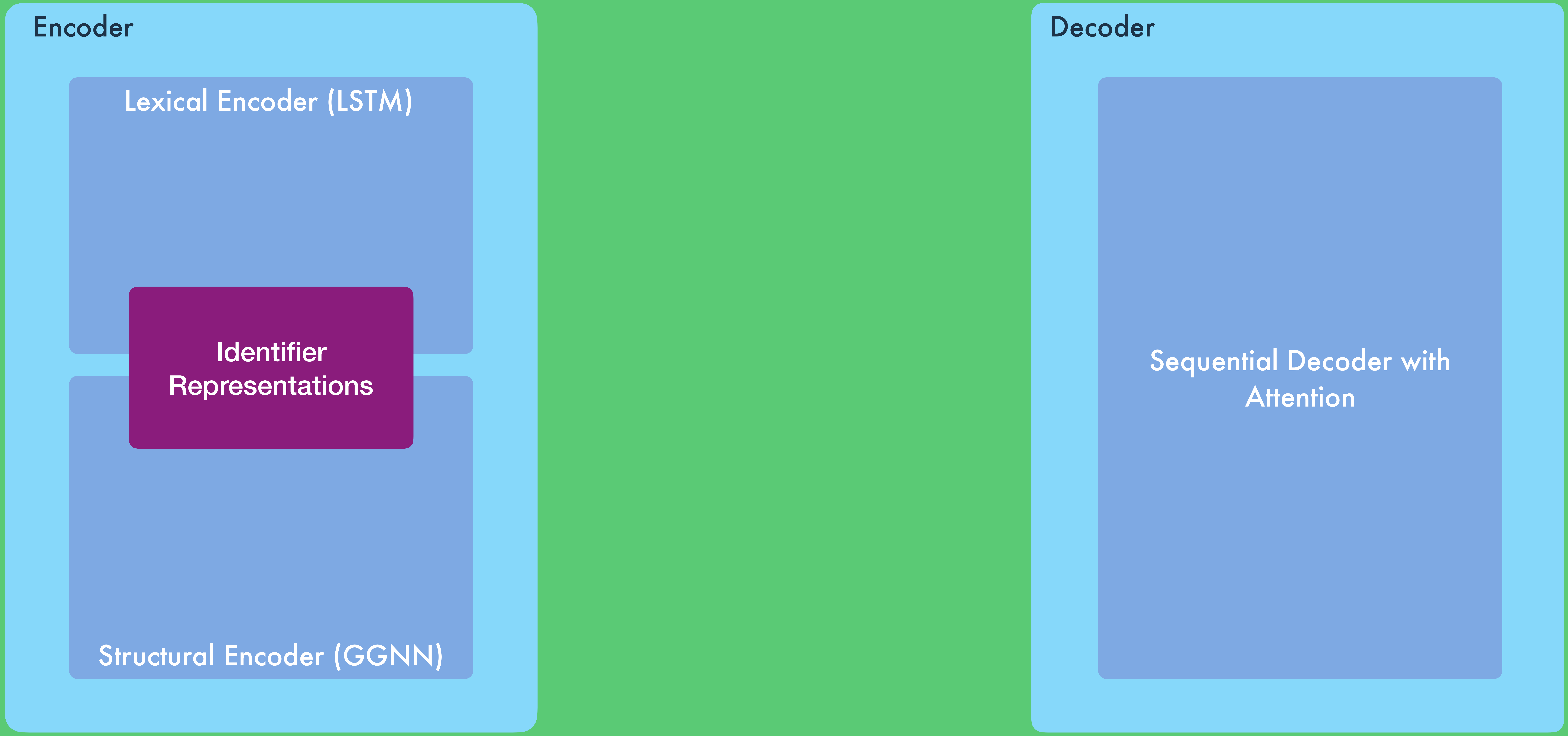
Lexical Encoder (LSTM)

Structural Encoder (GGNN)

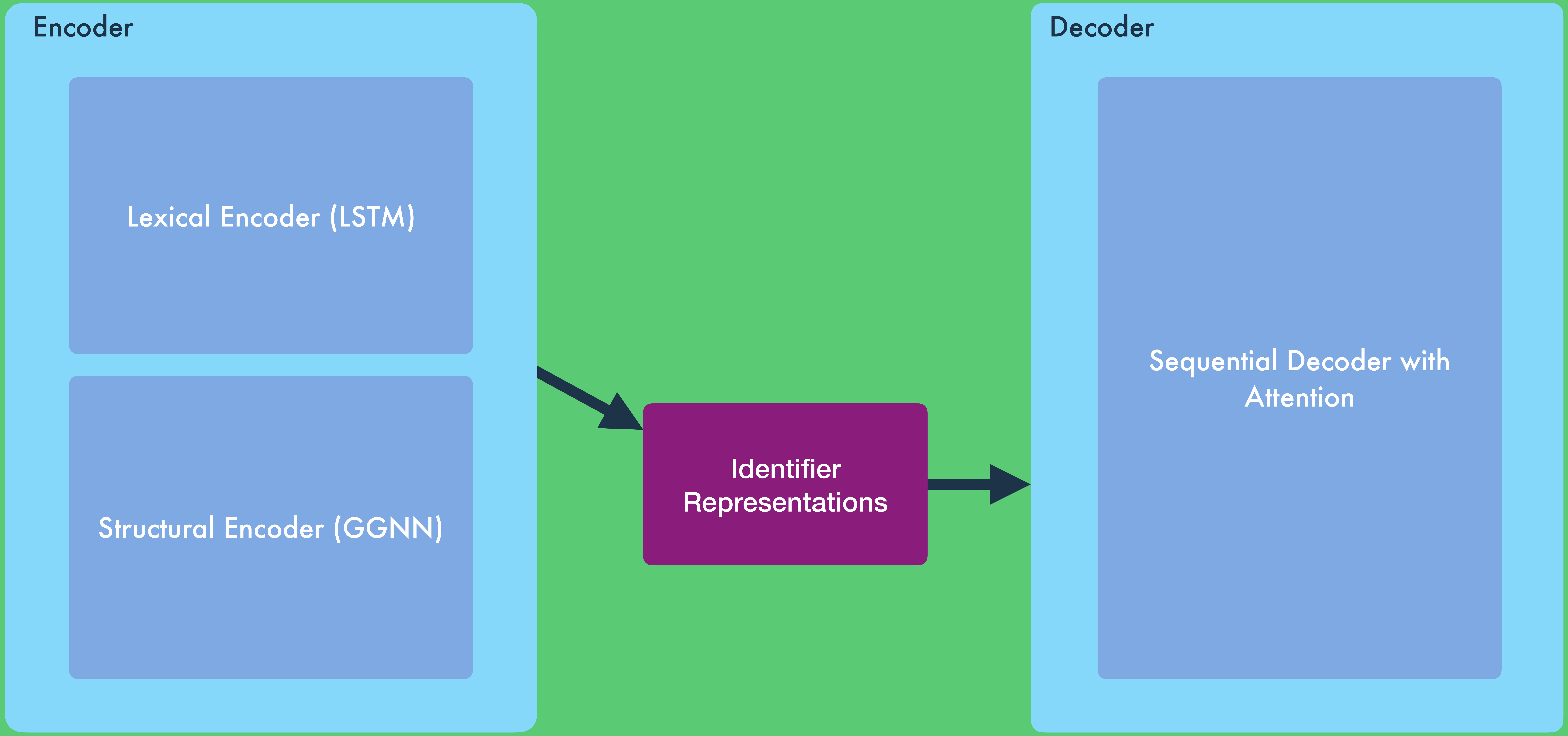
DIRE Neural Architecture



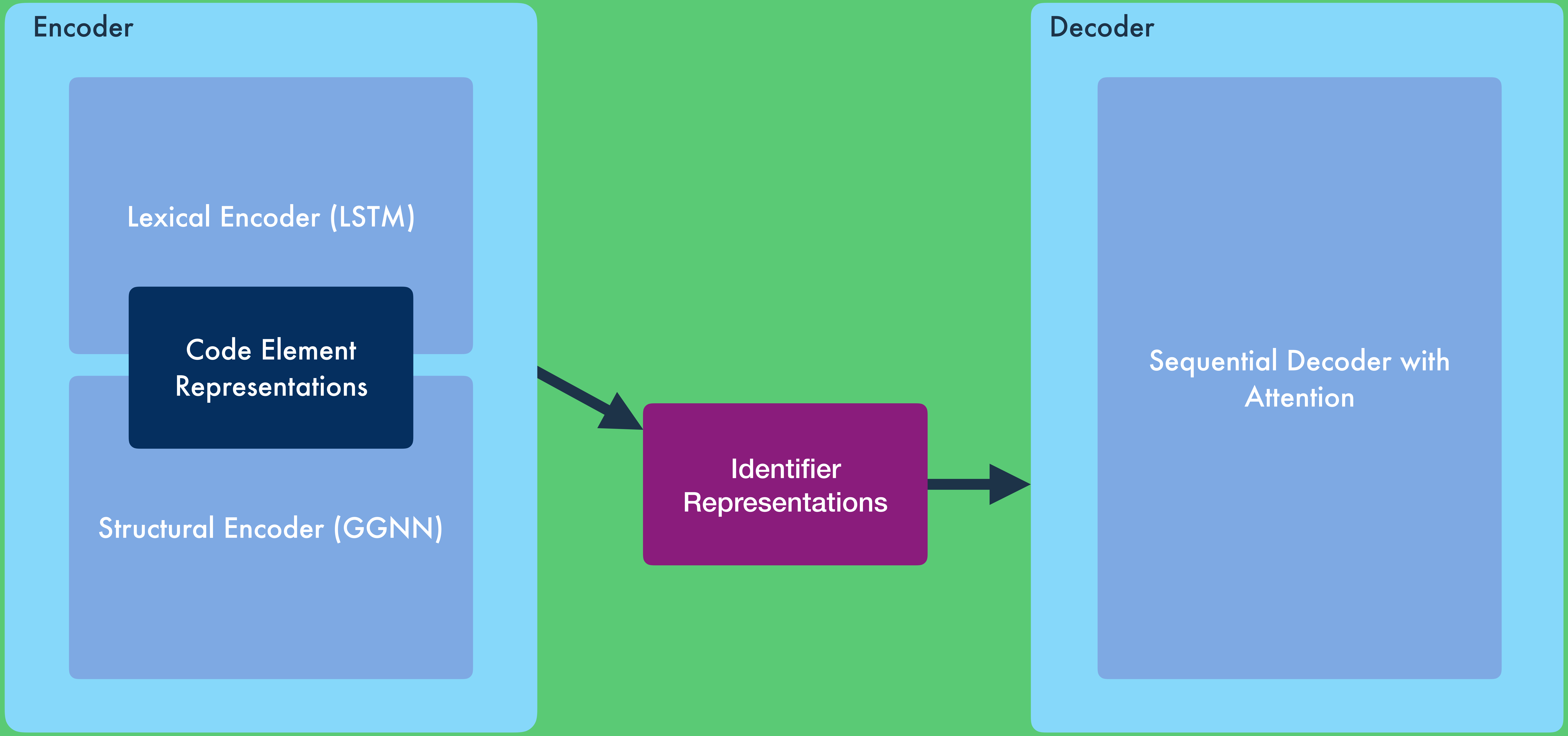
DIRE Neural Architecture



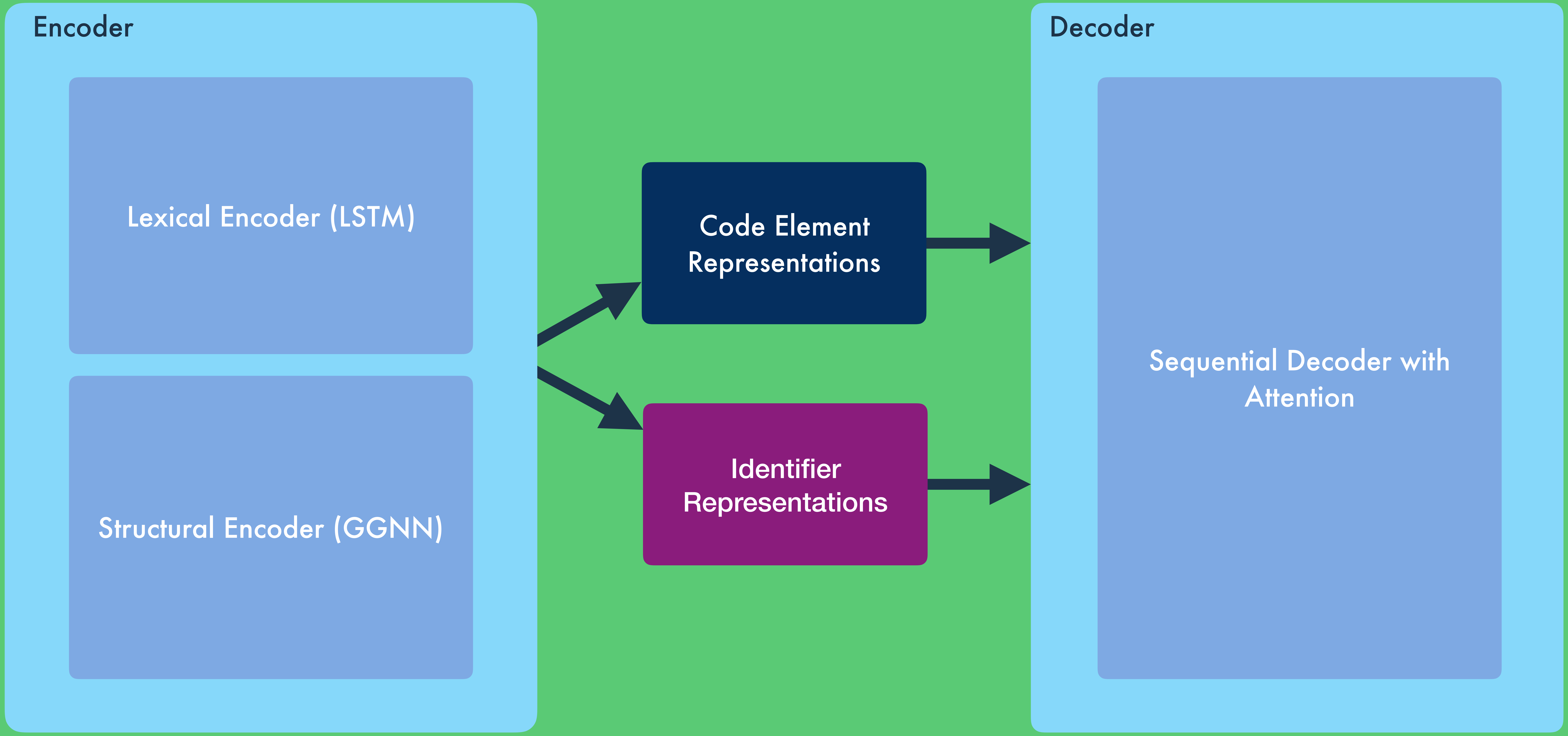
DIRE Neural Architecture



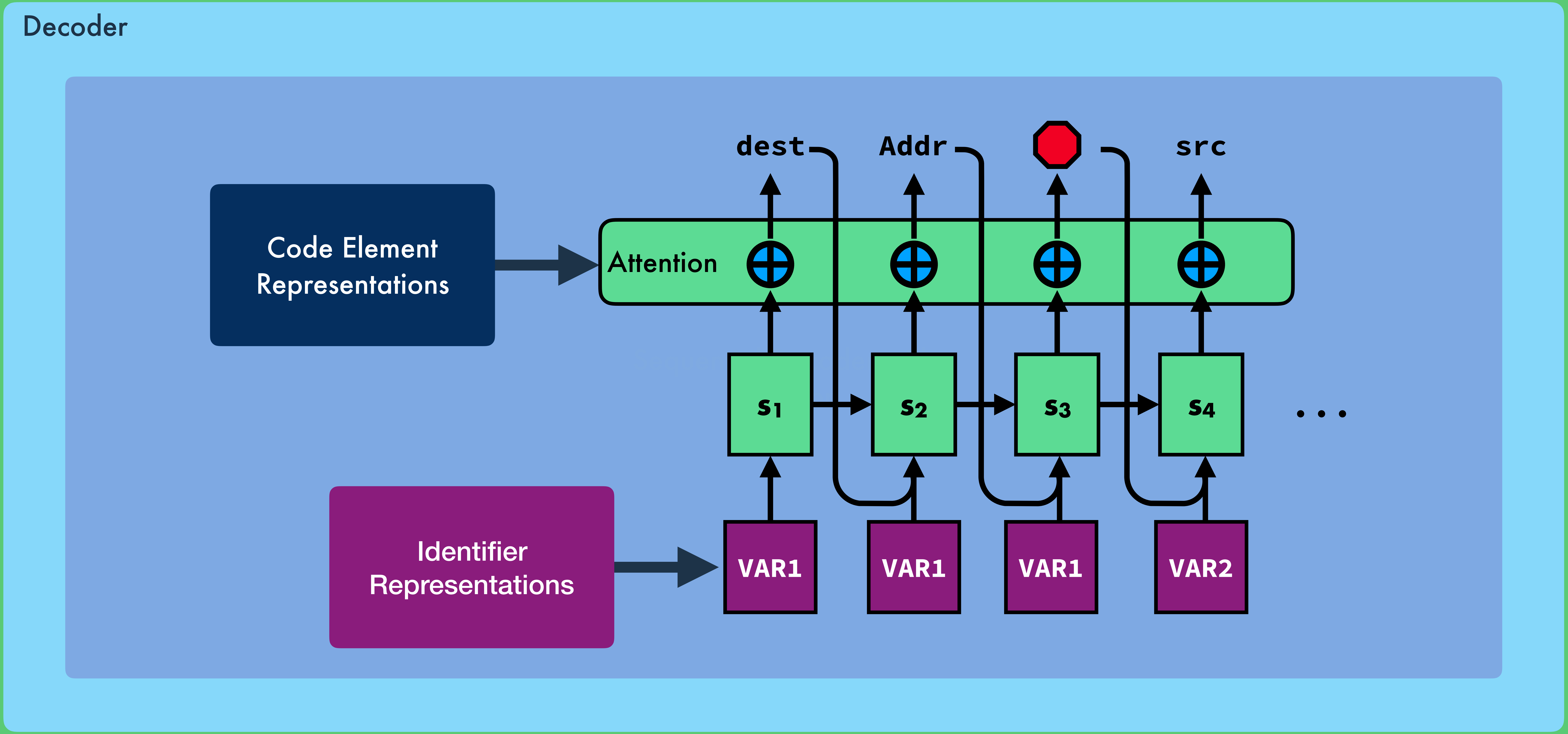
DIRE Neural Architecture



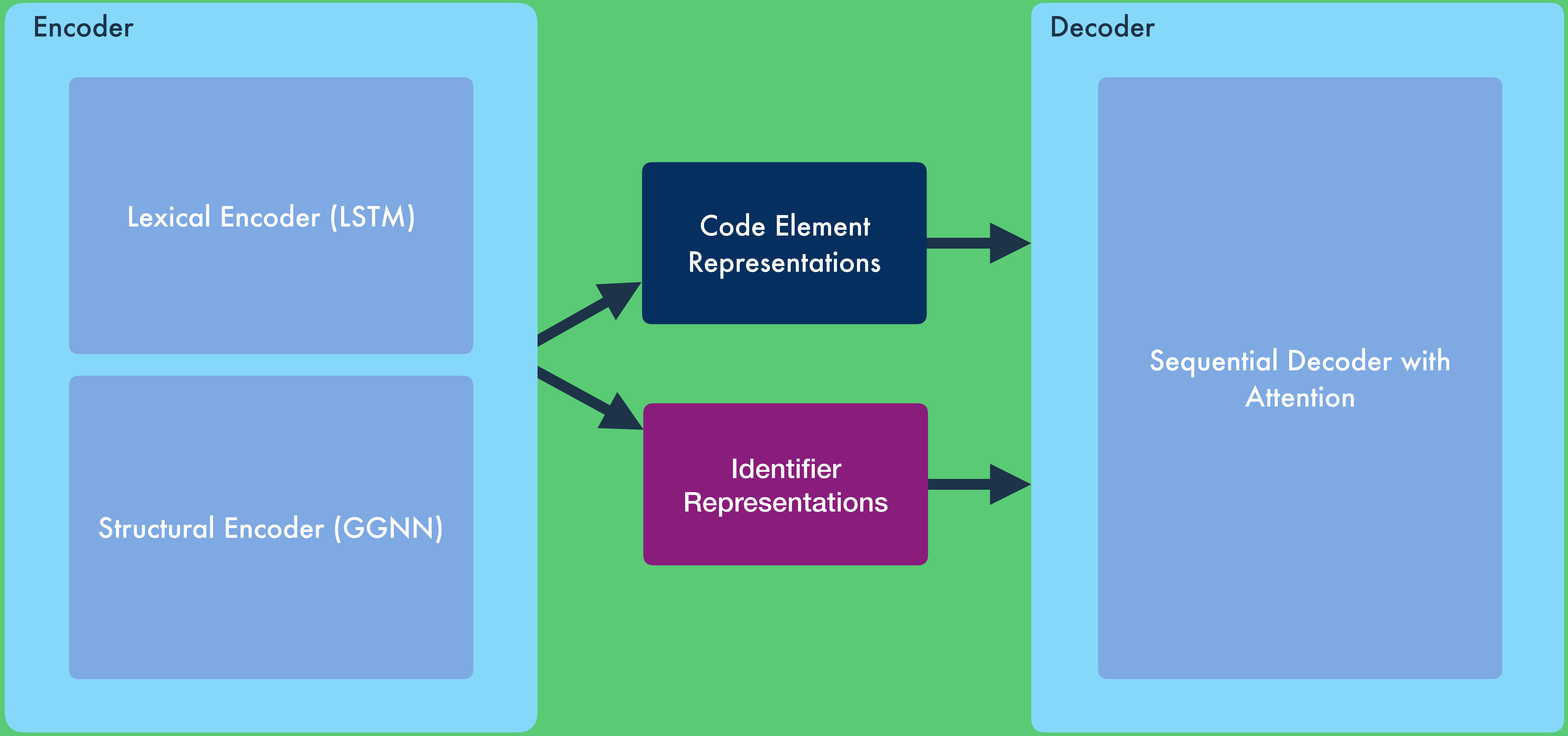
DIRE Neural Architecture

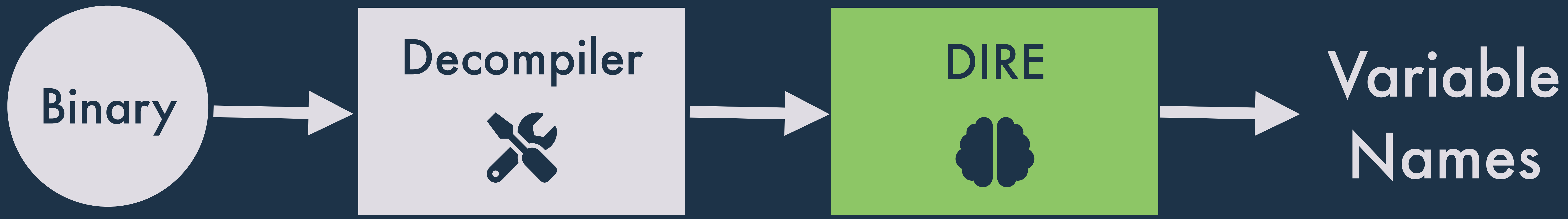


DIRE Neural Architecture

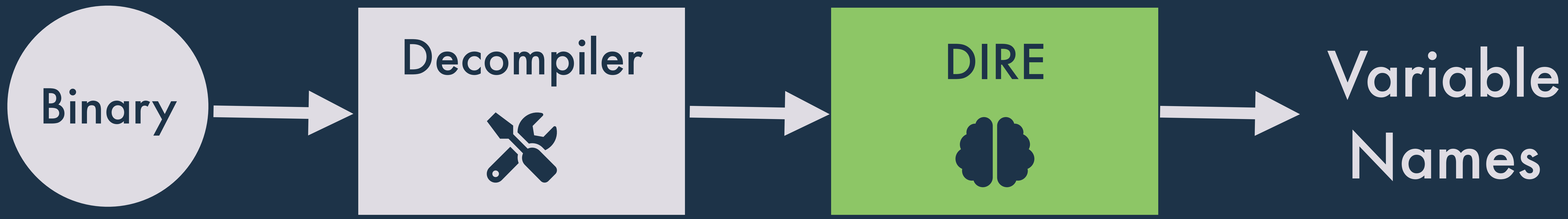


DIRE Neural Architecture





How good are the renamings?



Assumption:

Original (human-written) names are good

How many can we recover?

Dataset

- 164,632 unique x86-64 binaries
- 1,259,935 decompiled functions
- Split by binary into test/training/validation
- Open dataset, link in paper/on ASE site

Variable Recovery Rate (%)

DIRE	Lexical	Structural	Prior Work*
74.3	72.9	64.6	16.2

*__Meaningful Variable Names for Decompiled Code: A Machine Translation Approach,__
A. Jaffe, J. Lacomis, E. J. Schwartz, C. Le Goues, and B. Vasilescu, in *ICPC*, 2018

Variable Recovery Rate (%)

DIRE	Lexical	Structural	Prior Work*
74.3	72.9	64.6	16.2

* **Meaningful Variable Names for Decompiled Code: A Machine Translation Approach,**
A. Jaffe, J. Lacomis, E. J. Schwartz, C. Le Goues, and B. Vasilescu, in *ICPC*, 2018

Variable Recovery Rate (%)

DIRE	Lexical	Structural	Prior Work*
74.3	72.9	64.6	16.2

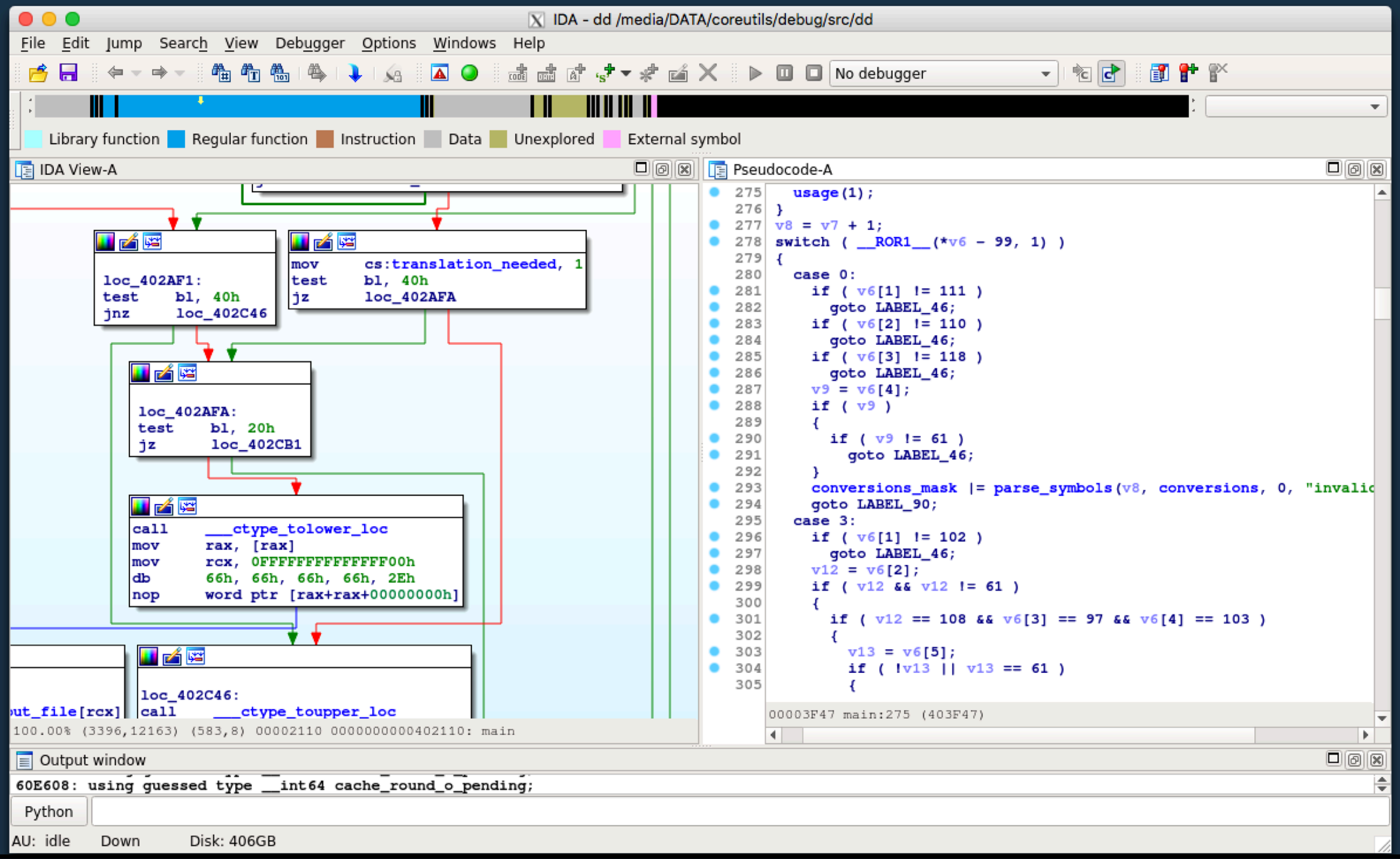
* **Meaningful Variable Names for Decompiled Code: A Machine Translation Approach,** A. Jaffe, J. Lacomis, E. J. Schwartz, C. Le Goues, and B. Vasilescu, in *ICPC*, 2018

Example

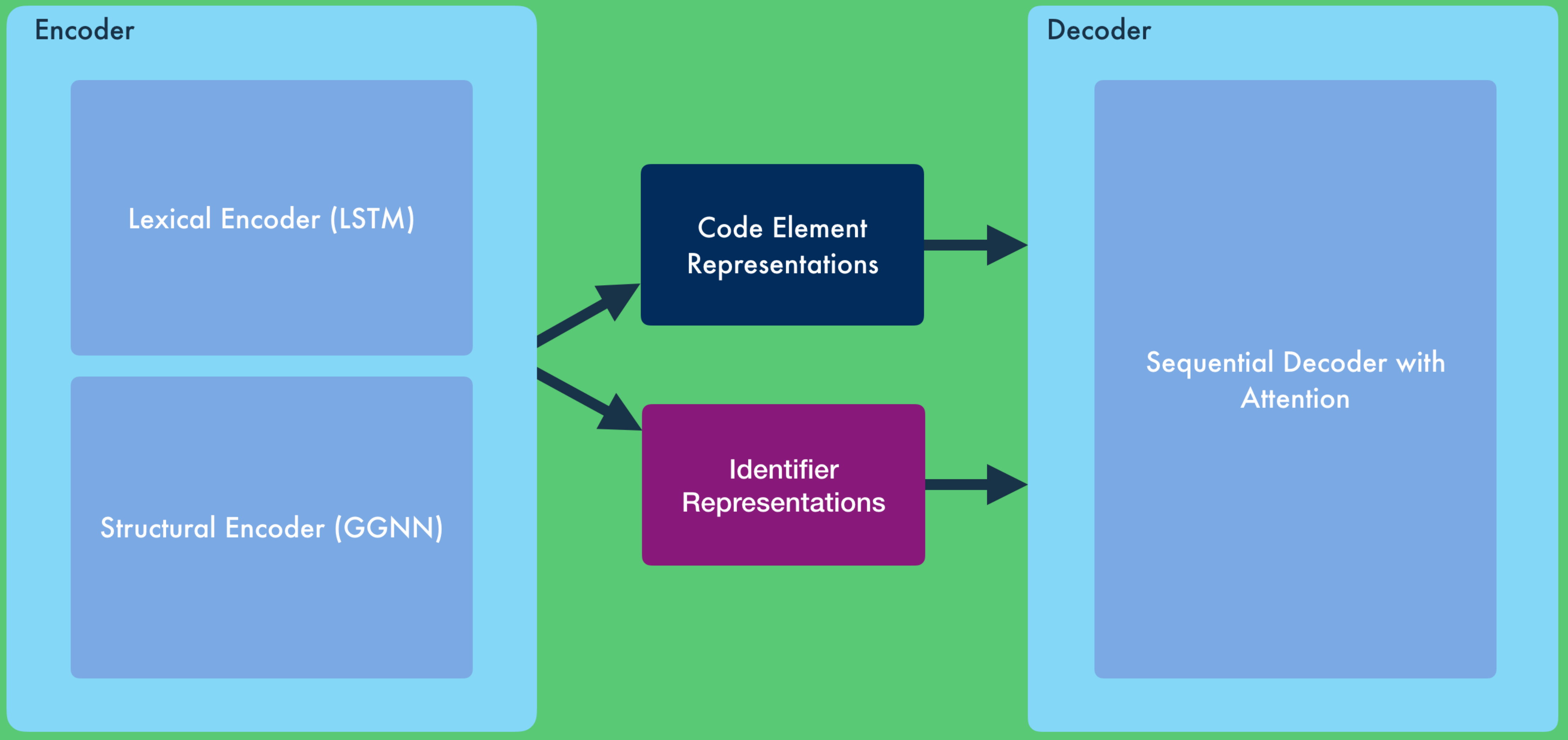
```
file *f_open(char **V1, char *V2, int V3) {
  int fd;
  if (!V3)
    return fopen(*V1, V2);
  if (*V2 != 119)
    assert_fail("fopen");
  fd = open(*V1, 577, 384);
  if (fd >= 0)
    return reopen(fd, V2);
  else
    return 0;
}
```

	Developer	Lexical	Structural	DIRE
V1	filename	file	fname	filename
V2	mode	name	oname	mode
V3	is_private	mode	flags	create

Decompiler



DIRE Neural Architecture



Today

Decompiler output → Refactored decompiler output

```c
void *file_mmap(int V1, int V2)
{
  void *V3;
  V3 = mmap(0, V2, 1, 2, V1, 0);
  if (V3 == (void *) -1) {
    perror("mmap");
    exit(1);
  }
  return V3;
}
```

```c
void *file_mmap(int fd, int size)
{
  void *ret;
  ret = mmap(0, size, 1, 2, fd, 0);
  if (ret == (void *) -1) {
    perror("mmap");
    exit(1);
  }
  return ret;
}
```


Jeremy Lacomis
jlacomis@cmu.edu
@jlacomis
jeremylacomis.com

STRUDEL
SOCIO-TECHNICAL RESEARCH USING DATA EXCAVATION LAB

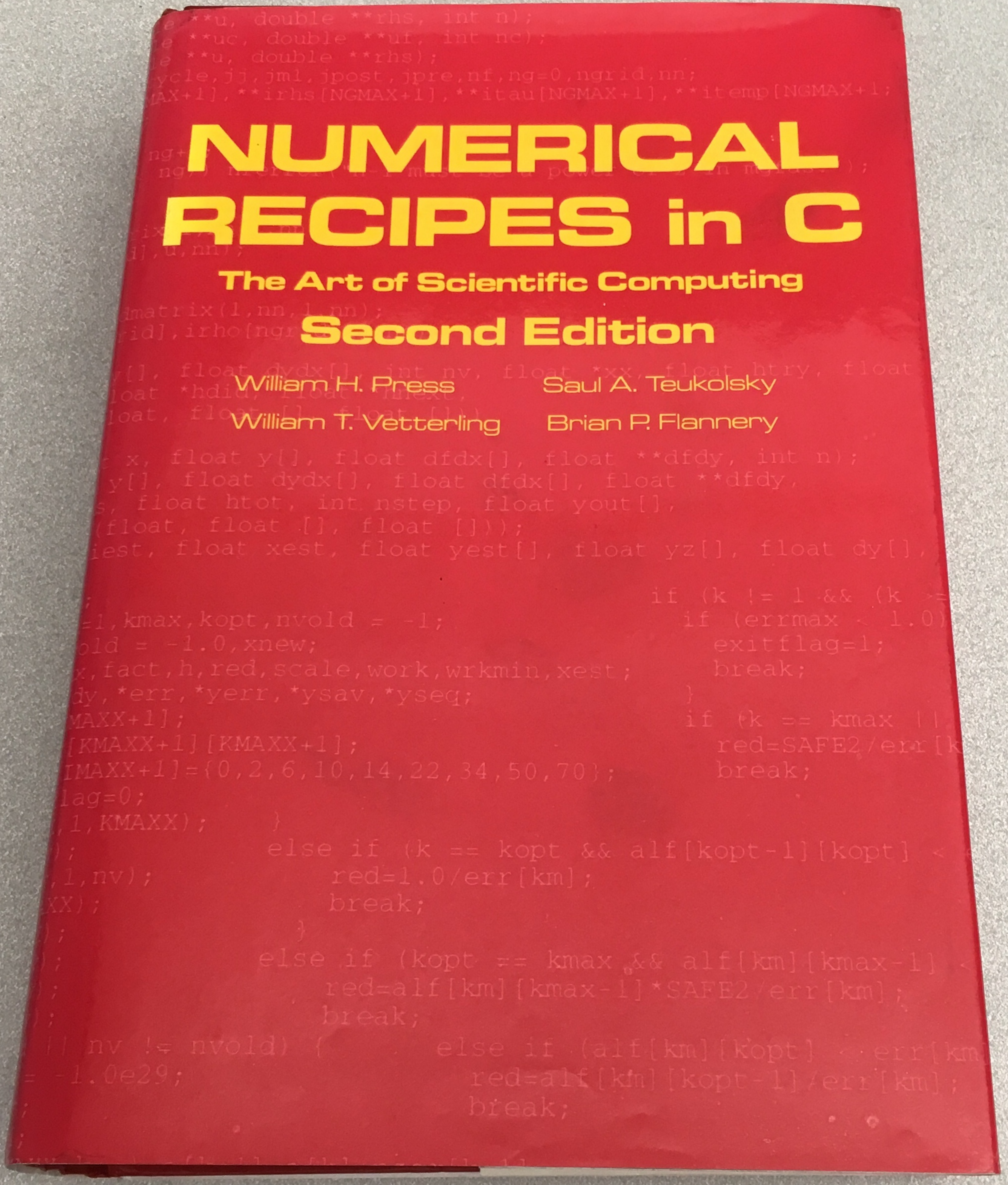
squaresLab
Software Quality in Real Evolving Systems

Software Engineering Institute
Carnegie Mellon

NEULAB

Microsoft Research

BONUS SLIDES


NUMERICAL
RECIPES in C
The Art of Scientific Computing
Second Edition
William H. Press
Saul A. Teukolsky
William T. Vetterling
Brian P. Flannery

will always converge, *provided* that the initial guess is good enough. Indeed one can even determine in advance the rate of convergence of most algorithms.

It cannot be overemphasized, however, how crucially success depends on having a good first guess for the solution, especially for multidimensional problems. This crucial beginning usually depends on analysis rather than numerics. Carefully crafted initial estimates reward you not only with reduced computational effort, but also with understanding and increased self-esteem. Hamming's motto, "the purpose of computing is insight, not numbers," is particularly apt in the area of finding roots. You should repeat this motto aloud whenever your program converges, with ten-digit accuracy, to the wrong root of a problem, or whenever it fails to converge because there is actually *no* root, or because there is a root but your initial estimate was not sufficiently close to it.

"This talk of insight is all very well, but what do I actually do?" For one-dimensional root finding, it is possible to give some straightforward answers: You should try to get some idea of what your function looks like before trying to find its roots. If you need to mass-produce roots for many different functions, then you should at least know what some typical members of the ensemble look like. Next, you should always bracket a root, that is, know that the function changes sign in an identified interval, before trying to converge to the root's value.

Finally (this is advice with which some daring souls might disagree, but we give it nonetheless) never let your iteration method get outside of the best bracketing bounds obtained at any stage. We will see below that some pedagogically important algorithms, such as *secant method* or *Newton-Raphson*, can violate this last constraint, and are thus not recommended unless certain fixups are implemented.

Multiple roots, or very close roots, are a real problem, especially if the multiplicity is an even number. In that case, there may be no readily apparent sign change in the function, so the notion of bracketing a root — and maintaining the bracket — becomes difficult. We are hard-liners: we nevertheless insist on bracketing a root, even if it takes the minimum-searching techniques of Chapter 10 to determine whether a tantalizing dip in the function really does cross zero or not. (You can easily modify the simple golden section routine of §10.1 to return early if it detects a sign change in the function. And, if the minimum of the function is exactly zero, then you have found a *double* root.)

As usual, we want to discourage you from using routines as black boxes without understanding them. However, as a guide to beginners, here are some reasonable starting points:

- Brent's algorithm in §9.3 is the method of choice to find a bracketed root of a general one-dimensional function, when you cannot easily compute the function's derivative. Ridders' method (§9.2) is concise, and a close competitor.
- When you can compute the function's derivative, the routine `rtsafe` in §9.4, which combines the Newton-Raphson method with some bookkeeping on bounds, is recommended. Again, you must first bracket your root.
- Roots of polynomials are a special case. Laguerre's method, in §9.5, is recommended as a starting point. Beware: Some polynomials are ill-conditioned!
- Finally, for multidimensional problems, the only elementary method is Newton-Raphson (§9.6), which works *very* well if you can supply a

good first guess of the solution. Try it. Then read the more advanced material in §9.7 for some more complicated, but globally more convergent, alternatives.

Avoiding implementations for specific computers, this book must generally steer clear of interactive or graphics-related routines. We make an exception right now. The following routine, which produces a crude function plot with interactively scaled axes, can save you a lot of grief as you enter the world of root finding.

```
#include <stdio.h>
#define ISCR 60                     Number of horizontal and vertical positions in display.
#define JSCR 21
#define BLANK ' '
#define ZERO '-'
#define YY 'l'
#define XX '-'
#define FF 'x'

void scrsho(float (*fx)(float))
For interactive CRT terminal use. Produce a crude graph of the function fx over the prompted-
for interval x1,x2. Query for another plot until the user signals satisfaction.
{
    int jz,j,i;
    float ysml,ybig,x2,x1,x,dyj,dx,y[ISCR+1];
    char scr[ISCR+1][JSCR+1];

    for (;;) {
        printf("\nEnter x1 x2 (x1=x2 to stop):\n");    Query for another plot, quit
        scanf("%f %f",&x1,&x2);                           if x1=x2.
        if (x1 == x2) break;
        for (j=1;j<=JSCR;j++)                           Fill vertical sides with character 'l'.
            scr[1][j]=scr[ISCR][j]=YY;
        for (i=2;i<=(ISCR-1);i++) {
            scr[i][1]=scr[i][JSCR]=XX;                  Fill top, bottom with character '-'.
            for (j=2;j<=(JSCR-1);j++)                   Fill interior with blanks.
                scr[i][j]=BLANK;
        }
        dx=(x2-x1)/(ISCR-1);
        x=x1;
        ysml=ybig=0.0;                                  Limits will include 0.
        for (i=1;i<=ISCR;i++) {                         Evaluate the function at equal intervals.
            y[i]=(*fx)(x);                                 Find the largest and smallest val-
            if (y[i] < ysml) ysml=y[i];                    ues.
            if (y[i] > ybig) ybig=y[i];
            x += dx;
        }
        if (ybig == ysml) ybig=ysml+1.0;                Be sure to separate top and bottom.
        dyj=(JSCR-1)/(ybig-ysml);
        jz=1-(int) (ysml*dyj);                          Note which row corresponds to 0.
        for (i=1;i<=ISCR;i++) {                         Place an indicator at function height and
            scr[i][jz]=ZERO;                               0.
            j=1+(int) ((y[i]-ysml)*dyj);
            scr[i][j]=FF;
        }
        printf(" %10.3f ",ybig);
        for (i=1;i<=ISCR;i++) printf("%c",scr[i][JSCR]);
        printf("\n");
        for (j=(JSCR-1);j>=2;j--) {                     Display.
            printf("%12s"," ");
            for (i=1;i<=ISCR;i++) printf("%c",scr[i][j]);
            printf("\n");
        }
        printf(" %10.3f ",ysml);
```

will always converge, *provided* that the initial guess is good enough. Indeed one can even determine in advance the rate of convergence of most algorithms.

It cannot be overemphasized, however, how crucially success depends on having a good first guess for the solution, especially for multidimensional problems. This crucial beginning usually depends on analysis rather than numerics. Carefully crafted initial estimates reward you not only with reduced computational effort, but also with understanding and increased self-esteem. Hamming's motto, "the purpose of computing is insight, not numbers," is particularly apt in the area of finding roots. You should repeat this motto aloud whenever your program converges, with ten-digit accuracy, to the wrong root of a problem, or whenever it fails to converge because there is actually *no* root, or because there is a root but your initial estimate was not sufficiently close to it.

"This talk of insight is all very well, but what do I actually do?" For one-dimensional root finding, it is possible to give some straightforward answers: You should try to get some idea of what your function looks like before trying to find its roots. If you need to mass-produce roots for many different functions, then you should at least know what some typical members of the ensemble look like. Next, you should always bracket a root, that is, know that the function changes sign in an identified interval, before trying to converge to the root's value.

Finally (this is advice with which some daring souls might disagree, but we give it nonetheless) never let your iteration method get outside of the best bracketing bounds obtained at any stage. We will see below that some pedagogically important algorithms, such as *secant method* or *Newton-Raphson*, can violate this last constraint, and are thus not recommended unless certain fixups are implemented.

Multiple roots, or very close roots, are a real problem, especially if the multiplicity is an even number. In that case, there may be no readily apparent sign change in the function, so the notion of bracketing a root — and maintaining the bracket — becomes difficult. We are hard-liners: we nevertheless insist on bracketing a root, even if it takes the minimum-searching techniques of Chapter 10 to determine whether a tantalizing dip in the function really does cross zero or not. (You can easily modify the simple golden section routine of §10.1 to return early if it detects a sign change in the function. And, if the minimum of the function is exactly zero, then you have found a *double* root.)

As usual, we want to discourage you from using routines as black boxes without understanding them. However, as a guide to beginners, here are some reasonable starting points:

- Brent's algorithm in §9.3 is the method of choice to find a bracketed root of a general one-dimensional function, when you cannot easily compute the function's derivative. Ridders' method (§9.2) is concise, and a close competitor.
- When you can compute the function's derivative, the routine `rtsafe` in §9.4, which combines the Newton-Raphson method with some bookkeeping on bounds, is recommended. Again, you must first bracket your root.
- Roots of polynomials are a special case. Laguerre's method, in §9.5, is recommended as a starting point. Beware: Some polynomials are ill-conditioned!
- Finally, for multidimensional problems, the only elementary method is Newton-Raphson (§9.6), which works *very* well if you can supply a

good first guess of the solution. Try it. Then read the more advanced material in §9.7 for some more complicated, but globally more convergent, alternatives.

Avoiding implementations for specific computers, this book must generally steer clear of interactive or graphics-related routines. We make an exception right now. The following routine, which produces a crude function plot with interactively scaled axes, can save you a lot of grief as you enter the world of root finding.

```
#include <stdio.h>
#define ISCR 60                              Number of horizontal and vertical positions in display.
#define JSCR 21
#define BLANK ' '
#define ZERO '-'
#define YY '1'
#define XX '-'
#define FF 'x'

void scrsho(float (*fx)(float))
For interactive CRT terminal use. Produce a crude graph of the function fx over the prompted-
for interval x1,x2. Query for another plot until the user signals satisfaction.
{
    int jz,j,i;
    float ysml,ybig,x2,x1,x,dyj,dx,y[ISCR+1];
    char scr[ISCR+1][JSCR+1];

    for (;;) {
        printf("\nEnter x1 x2 (x1=x2 to stop):\n");     Query for another plot, quit
        scanf("%f %f",&x1,&x2);                              if x1=x2.
        if (x1 == x2) break;
        for (j=1;j<=JSCR;j++)                           Fill vertical sides with character '1'.
            scr[1][j]=scr[ISCR][j]=YY;
        for (i=2;i<=(ISCR-1);i++) {
            scr[i][1]=scr[i][JSCR]=XX;                  Fill top, bottom with character '-'.
            for (j=2;j<=(JSCR-1);j++)                   Fill interior with blanks.
                scr[i][j]=BLANK;
        }
        dx=(x2-x1)/(ISCR-1);
        x=x1;
        ysml=ybig=0.0;                                  Limits will include 0.
        for (i=1;i<=ISCR;i++) {                         Evaluate the function at equal intervals.
            y[i]=(*fx)(x);                                  Find the largest and smallest val-
            if (y[i] < ysml) ysml=y[i];                     ues.
            if (y[i] > ybig) ybig=y[i];
            x += dx;
        }
        if (ybig == ysml) ybig=ysml+1.0;                Be sure to separate top and bottom.
        dyj=(JSCR-1)/(ybig-ysml);
        jz=1-(int) (ysml*dyj);                          Note which row corresponds to 0.
        for (i=1;i<=ISCR;i++) {                         Place an indicator at function height and
            scr[i][jz]=ZERO;                                0.
            j=1+(int) ((y[i]-ysml)*dyj);
            scr[i][j]=FF;
        }
        printf(" %10.3f ",ybig);
        for (i=1;i<=ISCR;i++) printf("%c",scr[i][JSCR]);
        printf("\n");
        for (j=(JSCR-1);j>=2;j--) {                     Display.
            printf("%12s"," ");
            for (i=1;i<=ISCR;i++) printf("%c",scr[i][j]);
            printf("\n");
        }
        printf(" %10.3f ",ysml);
```

will always converge, *provided* that the initial guess is good enough. Indeed one can even determine in advance the rate of convergence of most algorithms.

It cannot be overemphasized, however, how crucially success depends on having a good first guess for the solution, especially for multidimensional problems. This crucial beginning usually depends on analysis rather than numerics. Carefully crafted initial estimates reward you not only with reduced computational effort, but also with understanding and increased self-esteem. Hamming's motto, "the purpose of computing is insight, not numbers," is particularly apt in the area of finding roots. You should repeat this motto aloud whenever your program converges, with ten-digit accuracy, to the wrong root of a problem, or whenever it fails to converge because there is actually *no* root, or because there is a root but your initial estimate was not sufficiently close to it.

"This talk of insight is all very well, but what do I actually do?" For one-dimensional root finding, it is possible to give some straightforward answers: You should try to get some idea of what your function looks like before trying to find its roots. If you need to mass-produce roots for many different functions, then you should at least know what some typical members of the ensemble look like. Next,

```
int jz,j,i;
float ysml,ybig,x2,x1,x,dyj,dx,y[ISCR+1];
char scr[ISCR+1][JSCR+1];
```

understanding them. However, as a guide to beginners, here are some reasonable starting points:

- Brent's algorithm in §9.3 is the method of choice to find a bracketed root of a general one-dimensional function, when you cannot easily compute the function's derivative. Ridders' method (§9.2) is concise, and a close competitor.
- When you can compute the function's derivative, the routine `rtsafe` in §9.4, which combines the Newton-Raphson method with some bookkeeping on bounds, is recommended. Again, you must first bracket your root.
- Roots of polynomials are a special case. Laguerre's method, in §9.5, is recommended as a starting point. Beware: Some polynomials are ill-conditioned!
- Finally, for multidimensional problems, the only elementary method is Newton-Raphson (§9.6), which works *very* well if you can supply a

good first guess of the solution. Try it. Then read the more advanced material in §9.7 for some more complicated, but globally more convergent, alternatives.

Avoiding implementations for specific computers, this book must generally steer clear of interactive or graphics-related routines. We make an exception right now. The following routine, which produces a crude function plot with interactively scaled axes, can save you a lot of grief as you enter the world of root finding.

```
#include <stdio.h>
#define ISCR 60          Number of horizontal and vertical positions in display.
#define JSCR 21
#define BLANK ' '
#define ZERO '-'
#define YY 'l'
#define XX '-'
#define FF 'x'

void scrsho(float (*fx)(float))
For interactive CRT terminal use. Produce a crude graph of the function fx over the prompted-
for interval x1,x2. Query for another plot until the user signals satisfaction.
{
```

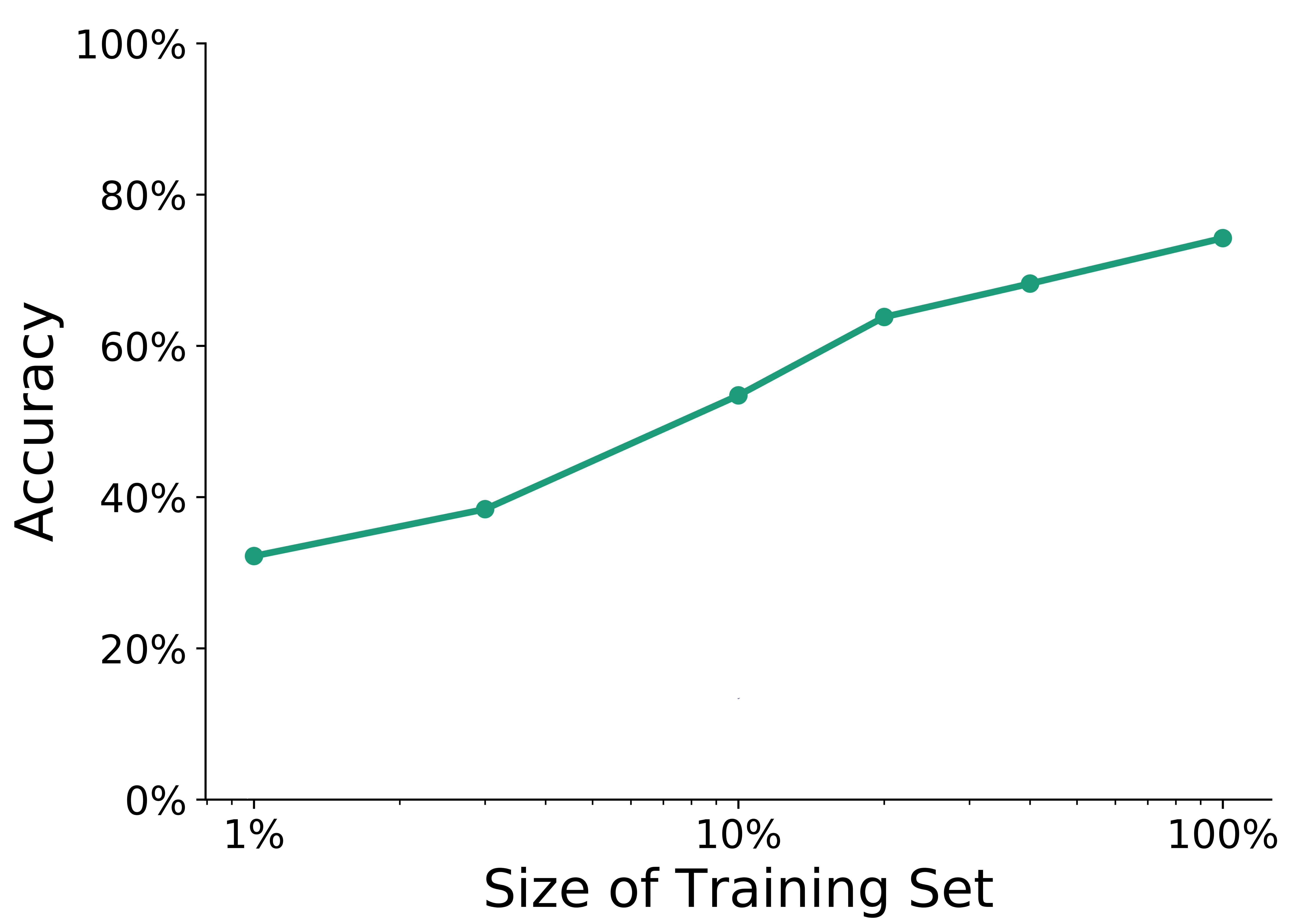
```
        x += dx;
    }
    if (ybig == ysml) ybig=ysml+1.0;        Be sure to separate top and bottom.
    dyj=(JSCR-1)/(ybig-ysml);
    jz=1-(int) (ysml*dyj);                  Note which row corresponds to 0.
    for (i=1;i<=ISCR;i++) {                 Place an indicator at function height and
        scr[i][jz]=ZERO;                        0.
        j=1+(int) ((y[i]-ysml)*dyj);
        scr[i][j]=FF;
    }
    printf(" %10.3f ",ybig);
    for (i=1;i<=ISCR;i++) printf("%c",scr[i][JSCR]);
    printf("\n");
    for (j=(JSCR-1);j>=2;j--) {             Display.
        printf("%12s"," ");
        for (i=1;i<=ISCR;i++) printf("%c",scr[i][j]);
        printf("\n");
    }
    printf(" %10.3f ",ysml);
```

Preliminary Human Study

```
1 int x = 1;
2 int y = 0;
3 while (x<= 5) {
4   y += 2;
5   x += 1;
6 }
7 printf("%d", y);
```

What is the value of the variable y on line 7?

The amount of training data matters



The uniqueness of the functions matters