Genotype Specification Language

"A compiler tool-chain whose target architecture is life itself"

Darren Platt
Email: darren@demetrixbio.com
Twitter: @dplattsf

Outline

- Introductions and acknowledgements
- Why a compiler for DNA
- Metabolic / Genetic Engineering 101
- Why F# for a DNA compiler
- Crash course on GSL itself
- Code vignettes
  - Parser
  - Abstract Syntax Tree
  - Putting Railroad Oriented Programming to use
  - Units of measure and reagent design

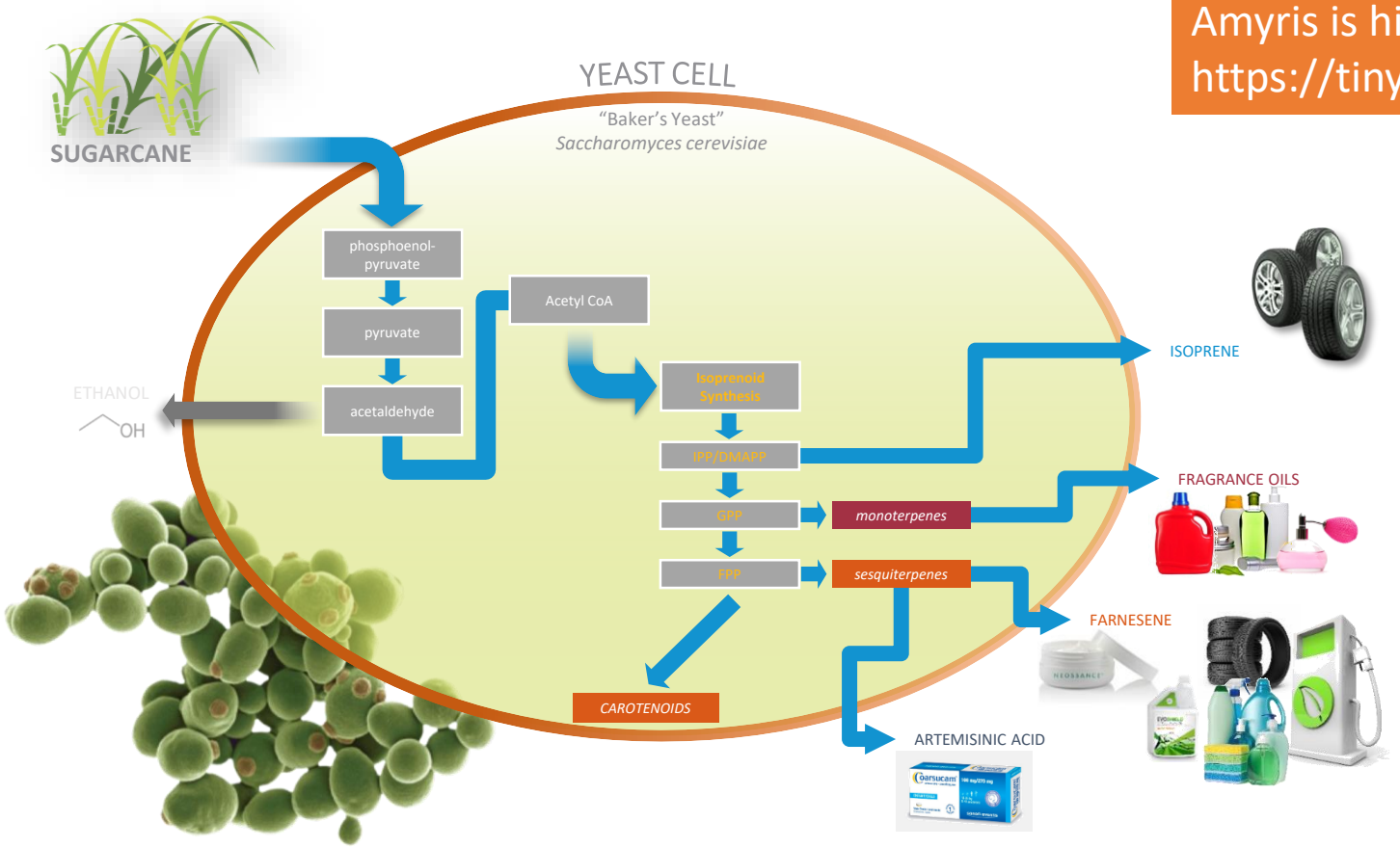
Introductions and Acknowledgements

It takes a village to raise a DNA compiler

- Chris Macklin
- Erin H. Wilson
- Shiori Sagawa
- James Weis
- Michael Bissell
- Brian Hawthorne
- Autodesk NanoBio
- Christopher Reeves
- Jed Dean
- Max G. Schubert
- Andrew Horwitz
- Svetlana Borisova

Amyris engineers living factories that produce chemicals from sustainable sources

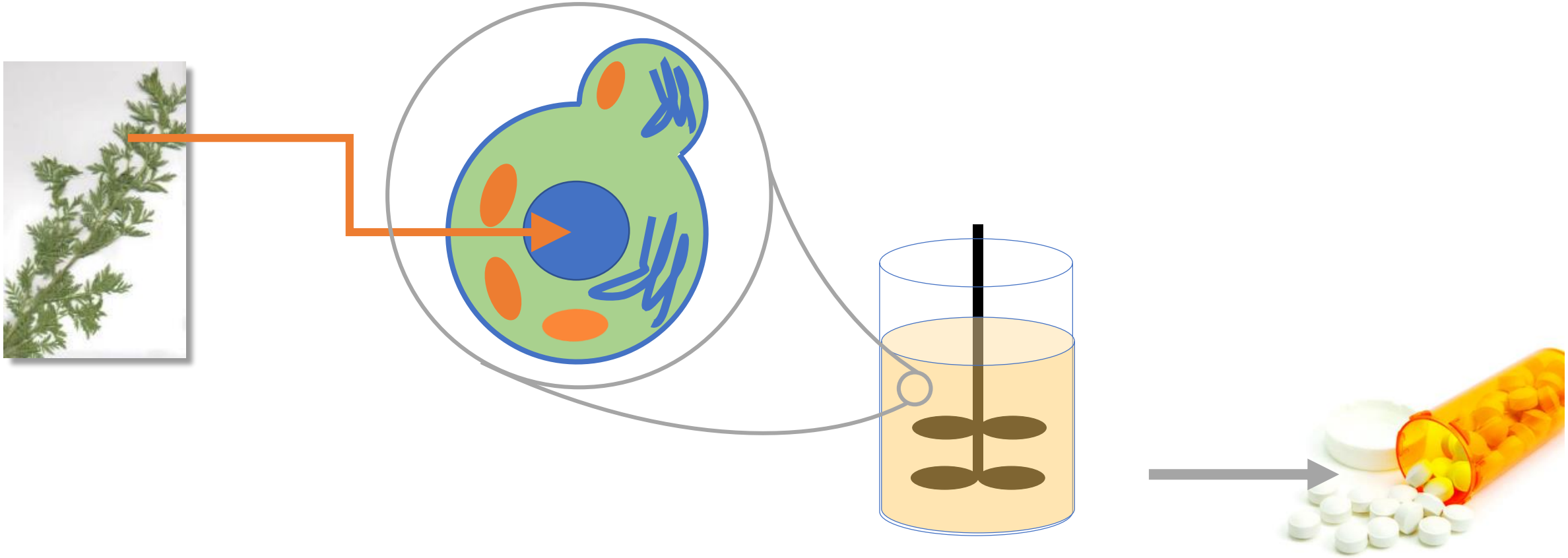
Amyris is hiring F# devs
https://tinyurl.com/fsharpamyris



Courtesy: Amyris inc. http://amyris.com

DEMETRIX

Nature's medicines through synthetic biology



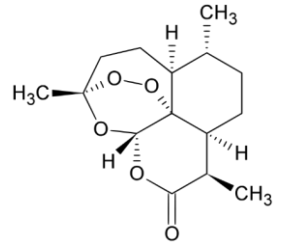
Why a compiler for DNA?

DNA appealing like a binary language

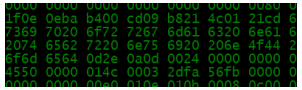
```
daz@platt-surface  /c/seq/GSL/gslc/bin/Debug
$ od -x gslc.exe | head -50
0000000 5a4d 0090 0003 0000 0004 0000 ffff 0000
0000020 00b8 0000 0000 0000 0040 0000 0000 0000
0000040 0000 0000 0000 0000 0000 0000 0000 0000
0000060 0000 0000 0000 0000 0000 0000 0080 0000
0000100 1f0e 0eba b400 cd09 b821 4c01 21cd 6854
0000120 7369 7020 6f72 7267 6d61 6320 6e61 6f6e
0000140 2074 6562 7220 6e75 6920 206e 4f44 2053
0000160 6f6d 6564 0d2e 0a0d 0024 0000 0000 0000
0000200 4550 0000 014c 0003 2dfa 56fb 0000 0000
0000220 0000 0000 00e0 010e 010b 0008 0c00 0012
0000240 0400 0000 0000 0000 2b2e 0012 2000 0000
0000260 4000 0012 0000 0040 2000 0000 0200 0000
0000300 0004 0000 0000 0000 0004 0000 0000 0000
0000320 8000 0012 0200 0000 0000 0000 0003 0540
0000340 0000 0010 1000 0000 0000 0010 1000 0000
0000360 0000 0000 0010 0000 0000 0000 0000 0000
0000400 2acb 0012 0060 0000 4000 0012 0248 0000
0000420 0000 0000 0000 0000 0000 0000 0000 0000
0000440 6000 0012 000c 0000 2b34 0012 001c 0000
0000460 0000 0000 0000 0000 0000 0000 0000 0000
*
0000520 0000 0000 0000 0000 2000 0000 0008 0000
0000540 0000 0000 0000 0000 2008 0000 0048 0000
0000560 0000 0000 0000 0000 742e 7865 0074 0000
0000600 0bd4 0012 2000 0000 0c00 0012 0200 0000
0000620 0000 0000 0000 0000 0000 0000 0020 6000
0000640 722e 7273 0063 0000 0248 0000 4000 0012
0000660 0400 0000 0e00 0012 0000 0000 0000 0000
0000700 0000 0000 0040 4000 722e 6c65 636f 0000
0000720 000c 0000 6000 0012 0200 0000 1200 0012
0000740 0000 0000 0000 0000 0000 0000 0040 4200
```

```
ATGTCATTCGACGACTTACACAAAGCCACTGAGAGAGCGGTCATCCAGGCCGTGGACCAG
ATCTGCGACGATTTCGAGGTTACCCCCGAGAAGCTGGACGAATTAACTGCTTACTTCATC
GAACAAATGGAAAAAGGTCTAGCTCCACCAAAGGAAGGCCACACATTGGCCTCGGACAAA
GGTCTTCCTATGATTCCGGCGTTCGTCACCGGGTCACCCAACGGGACGGAGCGCGGTGTT
TTACTAGCCGCCGACCTGGGTGGTACCAATTTCCGTATATGTTCTGTTAACTTGCATGGA
GATCATACTTTCTCCATGGAGCAAATGAAGTCCAAGATTCCCGATGATTTGCTAGACGAT
GAGAACGTCACATCTGACGACCTGTTTGGGTTTCTAGCACGTCGTACACTGGCCTTTATG
AAGAAGTATCACCCGGACGAGTTGGCCAAGGGTAAAGACGCCAAGCCCATGAAACTGGGG
TTCACTTTCTCATACCCTGTAGACCAGACCTCTCTAAACTCCGGGACATTGATCCGTTGG
ACCAAGGGTTTCCGCATCGCGGACACCGTCGGAAAGGATGTCGTGCAATTGTACCAGGAG
CAATTAAGCGCTCAGGGTATGCCTATGATCAAGGTTGTTGCATTAACCAACGACACCGTC
GGAACGTACCTATCGCATTGCTACACGTCCGATAACACGGACTCAATGACGTCCGGAGAA
ATCTCGGAGCCGGTCATCGGATGTATTTTCGGTACCGGTACCAATGGGTGCTATATGGAG
GAGATCAACAAGATCACGAAGTTGCCACAGGAGTTGCGTGACAAGTTGATAAAGGAGGGT
AAGACACACATGATCATCAATGTCGAATGGGGGTCCTTCGATAATGAGCTCAAGCACTTG
CCTACTACTAAGTATGACGTCGTAATTGACCAGAAACTGTCAACGAACCCGGGATTTCAC
TTGTTTGAAAAACGTGTCTCAGGGATGTTCTTGGGTGAGGTGTTGCGTAACATTTTAGTG
GACTTGCACTCGCAAGGCTTGCTTTTGCAACAGTACAGGTCCAAGGAACAACTTCCTCGC
CACTTGACTACACCTTTCCAGTTGTCATCCGAAGTGCTGTCGCATATTGAAATTGACGAC
TCGACAGGTCTACGTGAAACAGAGTTGTCATTATTACAGAGTCTCAGACTGCCCACCACT
CCAACAGAGCGTGTTCAAATTCAAAAATTGGTGCGCGCGATTTCTAGGAGATCTGCGTAT
TTAGCCGCCGTGCCGCTTGCCGCGATATTGATCAAGACAAATGCTTTGAACAAGAGATAT
CATGGTGAAGTCGAGATCGGTTGTGATGGTTCCGTTGTGGAATACTACCCCGGTTTCAGA
TCTATGCTGAGACACGCCTTAGCCTTGTCACCCTTGGGTGCCGAGGGTGAGAGGAAGGTG
```

The Application Stack



e.g. Flinging birds at pigs

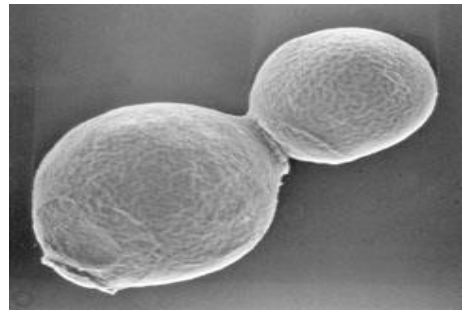


PostgreSQL

- Application
- Database
- Operating System
- Virtualization
- Physical server
- Storage

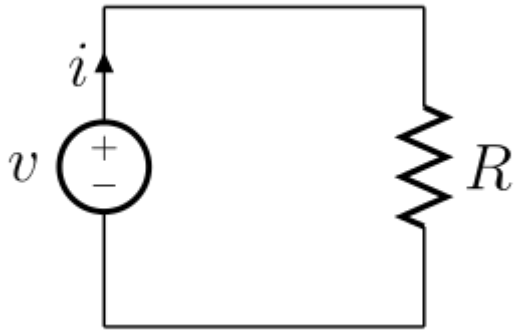
e.g. Anti malarial drug production

- Application
- Micro-organism
- Fermentor
- Manufacturing Facility

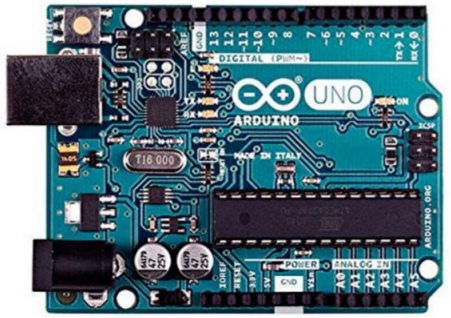


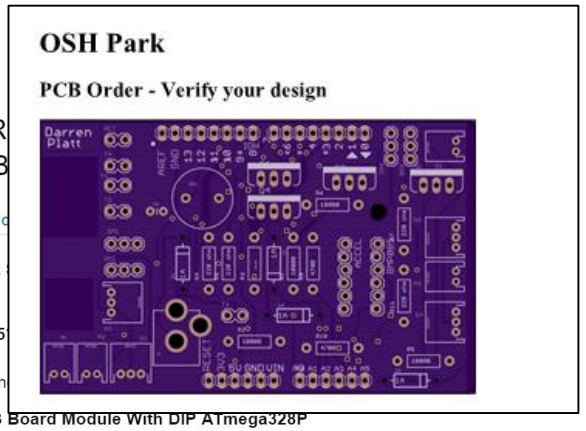
Amyris production facility
Brazil

Lessons from Electronics and Tech



Components and Circuits

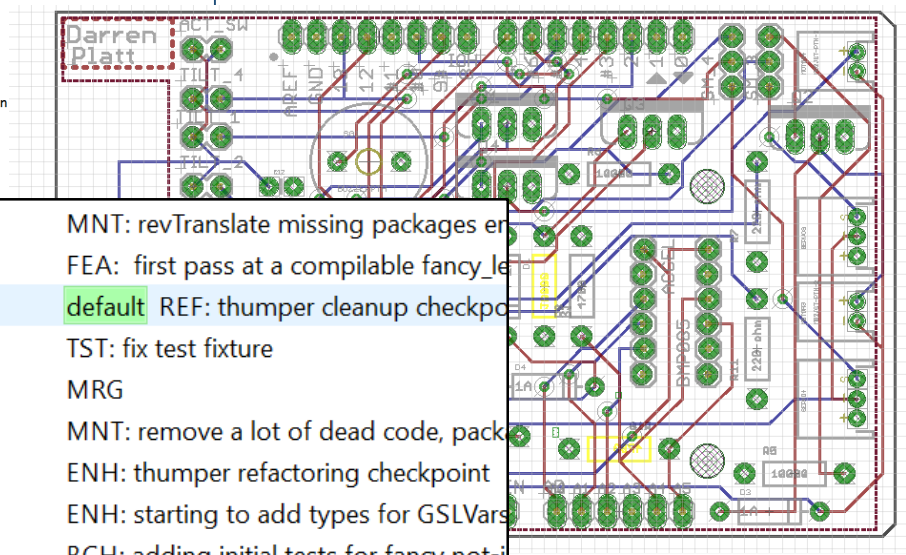




Arduino UNO R
ATmega328P(B
by Arduino Org
590 c
Price: $29.99 & FREE
In Stock.
Want it Tuesday, April 5
checkout. Details
Sold by Mp3Car Store an
Color: Arduino UNO R3 Board Module With DIP ATmega328P
Price For: Each
65 new from $10.54 1 used from $22.00

Commerce / Ecosystem

```
if not (codon2aa currentCodon = m.f) && asAACheck then// Ensure we are in the right place in the gene
    printfn "ORF: %s" (arr2seq orf)

    failwithf "ERROR: for mutation %c%d%c , gene g%s has amino acid %c (%s) in that position not %c"
                m.f m.pos m.t name (codon2aa currentCodon) (arr2seq currentCodon) m.f
// If mutation is relatively far from 3' end and close enough to 5' end, use 5' design
// but we prefer 3' designs since they are less disruptive
// Can't get too close to 5' end and still insert hB on left, so will need a 5' design
// 20120204:  Chris suggested changing 50 -> 30bp as the proximity limit for 5' design
if endPref = NTERM (*endPref <> CTERM || endPref = NTERM ||*) ||   (b < 30<ZeroOffset> || ( b < 1000<ZeroOffset> && len
    let mutSeq = selectMutCodonRight codonLookup minFreq currentCodon m.t |> arr2seq
    assert( (mutSeq.ToCharArray() |> translate).[0] = m.t)
    // 5' end design, need to rewrite promoter and put in marker
    //                              b
    // a......-1; marker ; a...ATG.....Mutation ; HB ...  HB + 700
    let a = -1000<ZeroOffset> // TODO
    let a2 = -600<ZeroOffset> // Promo

    sprintf "%s[~%A:~-100] {#name %s.5
                gene (zero2One a) name
                mutSeq gene (b+3<ZeroO
else
    let mutSeq = selectMutCodonLeft co
    assert( (mutSeq.ToCharArray() |> t

    //  a           b
    //  US700.......Mutation; HB ; dna
    let a' = b-700<ZeroOffset> // Pick
    let a = if (zero2One a') = 0<OneOf
    let c = 200<OneOffset>
    sprintf "%s[~%A:%A] {#name %s.hb}
            gene a ((zero2One (b-1<Zer
```



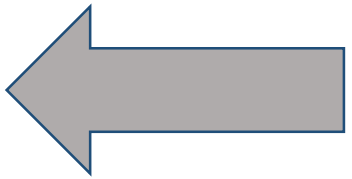
617	fancy_lets	MNT: revTranslate missing packages er
616	fancy_lets	FEA: first pass at a compilable fancy_le
615	default	default REF: thumper cleanup checkpo
614	default	TST: fix test fixture
613	default	MRG
612	default	MNT: remove a lot of dead code, pack
611	default	ENH: thumper refactoring checkpoint
610	fancy_lets	ENH: starting to add types for GSLVars
609	fancy_lets	BCH: adding initial tests for fancy not-j
608		

Tooling / Design Automation

If this is how you were writing your software

```
0001320 111e 2805 000e 0a00 0511 0f28 0000 110a
0001340 2805 0010 0a00 0511 1128 0000 110a 2805
0001360 0012 0a00 0511 1328 0000 110a 2805 0014
0001400 0a00 1573 0000 130a 0004 2c09 2b02 2b02
0001420 7208 0043 7000 2b00 7206 0049 7000 0600
0001440 1628 0000 280a 0017 0a00 2816 12b2 0600
0001460 1873 0000 130a 1706 0411 0611 1928 0000
0001500 130a 7207 004f 7000 1a73 0000 280a 0003
0001520 2b00 0813 0811 0d73 0000 1106 1104 6f07
0001540 001c 0a00 1d28 0000 110a 6f07 001c 0a00
0001560 1e28 0000 280a 001f 0a00 0711 1c6f 0000
0001600 280a 001e 0a00 698e 0428 0000 262b 002a
0001620 301b 0004 0319 0000 0012 1100 0000 2128
0001640 0000 280a 0005 2b00 0628 0000 0b2b 2807
0001660 0024 0a00 2c2c 0c07 2808 0025 0a00 7572
0001700 0000 2870 0026 0a00 022c 162b 2808 0025
0001720 0a00 8372 0000 2870 0026 0a00 022c 112b
0001740 202b 1528 0000 0006 2817 0007 2b00 2b00
0001760 2815 0001 0600 1400 1726 0728 0000 002b
0002000 042b 0d07 0009 000a 0600 1628 0000 1306
0002020 de05 7435 0038 0100 0613 9172 0000 7370
0002040 0028 0a00 0828 0000 132b 1107 6f06 0029
0002060 0a00 0813 0711 0811 2a6f 0000 260a 2817
0002100 0009 2b00 0513 00de 0511 0413 0411 2b28
0002120 0000 130a 1109 2804 002c 0a00 0a13 1100
0002140 130a 110c 280c 0024 0a00 162c 0c11 0d13
0002160 0d11 2428 0000 280a 0024 0a00 022d 1b2b
0002200 272b a572 0000 7370 002d 0a00 0a28 0000
0002220 262b 2817 000b 2b00 2b00 1138 280d 0025
0002240 0a00 0e13 0e11 2b00 112a 130c 720f 00e5
0002260 7000 2e73 0000 280a 000c 2b00 1013 0f11
0002300 1113 1011 1111 2f6f 0000 260a 2817 000b
0002320 2b00 1300 110b 280b 0030 0a00 022d 022b
0002340 292b 5b72 0001 7370 0028 0a00 0d28 0000
0002360 132b 1112 130b 1113 1112 6f13 002a 0a00
0002400 1826 0e28 0000 262b 2b00 0001 0911 8d7b
0002420 0008 2c04 2b02 2b02 1114 1109 280b 0066
0002440 0600 1600 0e28 0000 262b 2b00 0001 0911
0002460 7d7b 0008 2d04 2b02 2b02 7225 0195 7000
0002500 2873 0000 280a 000d 2b00 1413 1028 0000
0002520 1306 1115 1114 6f15 002a 0a00 0026 012b
0002540 0000 0911 0b11 6828 0000 0006 2816 000f
0002560 2b00 1613 39dd 0001 7400 0038 0100 1713
0002600 cd72 0001 7370 0028 0a00 0828 0000 132b
0002620 1118 6f17 0029 0a00 1913 1811 1911 2a6f
0002640 0000 260a 1711 316f 0000 130a 141a 1b13
0002660 1a11 1b11 1028 0000 162b 01fe 022c 052b
0002700 ae38 0000 1100 6f17 0031 0a00 296f 0000
0002720 720a 01d3 7000 336f 0000 2c0a 2b02 2b02
0002740 722c 01cd 7000 2873 0000 280a 0008 2b00
0002760 1c13 1711 316f 0000 6f0a 0029 0a00 1d13
0003000 1c11 1d11 2a6f 0000 260a 2b00 722a 0091
0003020 7000 2873 0000 280a 0008 2b00 1e13 1711
0003040 316f 0000 6f0a 0029 0a00 1f13 1e11 1f11
0003060 2a6f 0000 260a 1100 7b09 0886 0400 022c
```

Impressive, but you should be fired..

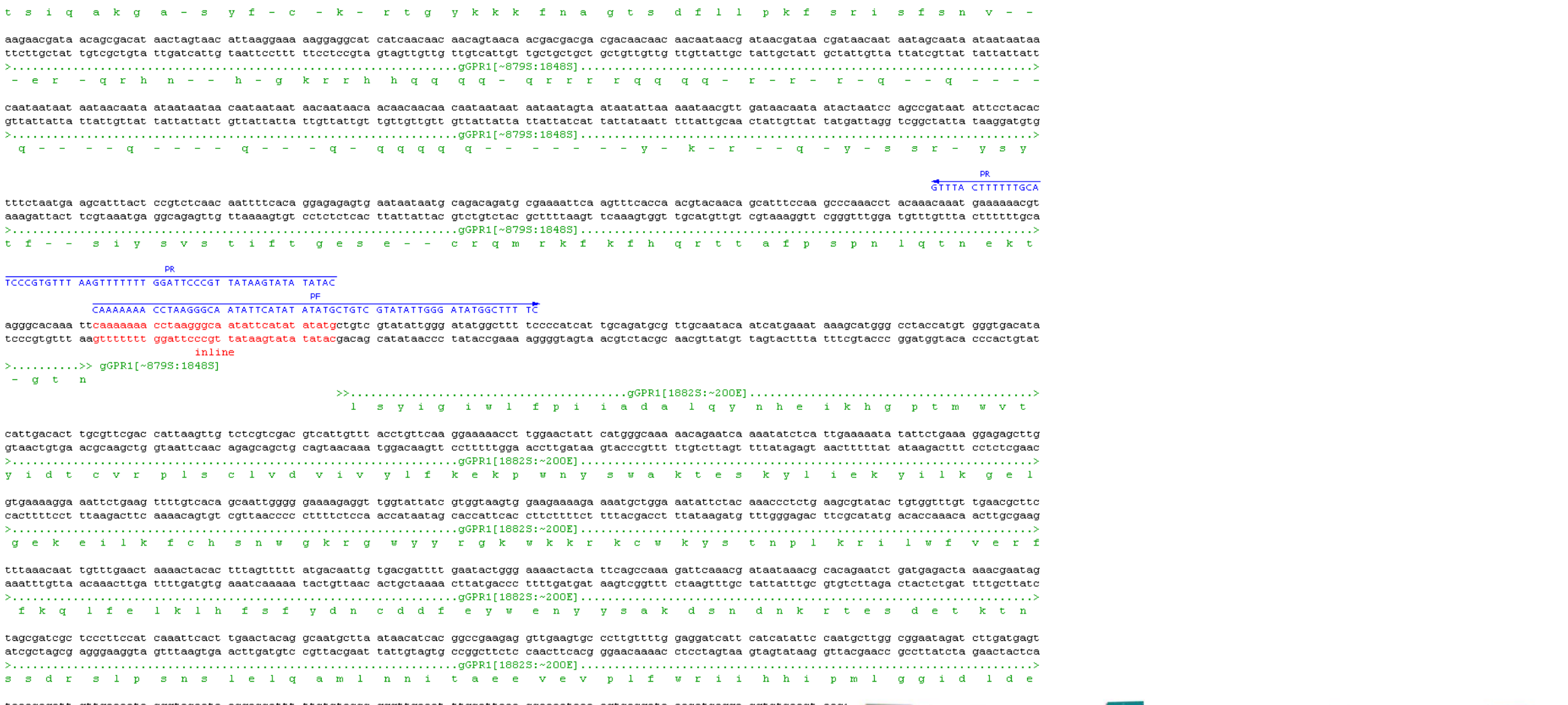


Compilers translate into machine language

Humans write in high level languages

```
if not (codon2aa currentCodon = m.f) && asAACheck then// Ensure we are in the right place in the gene
    printfn "ORF: %s" (arr2seq orf)

    failwithf "ERROR: for mutation %c%d%c , gene g%s has amino acid %c (%s) in that position not %c"
                m.f m.pos m.t name (codon2aa currentCodon) (arr2seq currentCodon) m.f
// If mutation is relatively far from 3' end and close enough to 5' end, use 5' design
// but we prefer 3' designs since they are less disruptive
// Can't get too close to 5' end and still insert hB on left, so will need a 5' design
// 20120204:  Chris suggested changing 50 -> 30bp as the proximity limit for 5' design
if endPref = NTERM (*endPref <> CTERM || endPref = NTERM ||*) ||   (b < 30<ZeroOffset> || ( b < 1000<ZeroOffset> && len-b > 25
    let mutSeq = selectMutCodonRight codonLookup minFreq currentCodon m.t |> arr2seq
    assert( (mutSeq.ToCharArray() |> translate).[0] = m.t)
    // 5' end design, need to rewrite promoter and put in marker
    //                                b
    // a......-1; marker ; a...ATG.......Mutation ; HB ...  HB + 700
    let a = -1000<ZeroOffset> // TODO - could adjust this based on intergenic distance
    let a2 = -600<ZeroOffset> // Promoter region

    sprintf "%s[~%A:~-100] {#name %s.5us} ;### ;%s[~%A:%A] {#name %s.hb} ;/%s/ {#inline };~ ;%s[%A:~%A]"
                gene (zero2One a) name gene (zero2One a2) (zero2One (b-1<ZeroOffset>)) name
                mutSeq gene (b+3<ZeroOffset> |> zero2One) (b+703<ZeroOffset> |> zero2One)    // 3' end design, classic
else
    let mutSeq = selectMutCodonLeft codonLookup minFreq currentCodon m.t |> arr2seq
    assert( (mutSeq.ToCharArray() |> translate).[0] = m.t)

    //  a           b                       c
    //  US700.......Mutation; HB ; dnaToDS200 ; marker ; 800bp downstream including 200bp repeat
    let a' = b-700<ZeroOffset> // Pick 700 so we can still sequence through
    let a = if (zero2One a') = 0<OneOffset> then 1<OneOffset> else zero2One a'
    let c = 200<OneOffset>
    sprintf "%s[~%A:%A] {#name %s.hb} ;~ ;/%s/ {#inline };%s[%A:~%AE] ;### ;%s[1E:~%AE] {#name %s.3ds} "
            gene a ((zero2One (b-1<ZeroOffset>)))  name  mutSeq     gene  (b+3<ZeroOffset> |> zero2One) c         gene     (c+
```

Common in our field to see this



Genetics sort of has a written notation

$$erg9\Delta :: kanMX\_P_{MET3}\text{-}ERG9$$

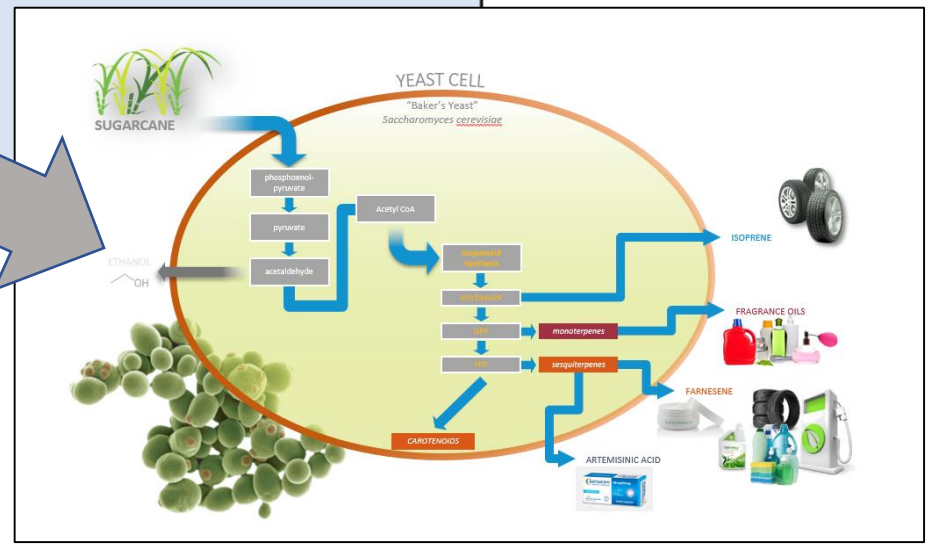
$$his3\Delta 1 :: hisMX\_P_{GAL1}\text{-}ERG12\_P_{GAL10}\text{-}ERG10$$

```
// Yeast terpene design
// Level 1 Syntax
// Part definitions
let pGAL1_10 = gGAL1[-668:-1]
let truncHMGR = /ATGG/ {#rabitstart }; gHMG1[1586:~200E]

// Locus engineering
uERG9 ; ### ; pMET3 ; gERG9[1:~500]
uLEU2 ; ### ;  !mERG8 ; @pGAL1_10 ; mMVD1 ; dLEU2
gHIS3[~-1000:-500]; ### ; !mERG10 ; @pGAL1_10 ; mERG12 ; gHIS3[-500:~500]
gADE1[~-1000:-500]; ### ; !mIDI1  ; @pGAL1_10 ; @truncHMGR ; gADE1[-500:~500]
gURA3[~-1000:-500]; ### ; !mERG13 ; @pGAL1_10 ; @truncHMGR ; gURA3[-500:~500]
gTRP1[~-1000:-500]; ### ; !@truncHMGR ;  @pGAL1_10 ; mERG20 ;  gTRP1[-500:~500]

// Level 2 syntax
pMET3>gERG9
pGAL1>mERG10; pGAL10>mERG12
pGAL1>mIDI1; pGAL10>@truncHMGR
pGAL1>@truncHMGR; pGAL10>mERG20 pGAL1>mERG8; pGAL10>mMVD1
```

Genotype Specification Language

Figure 7. Example yeast design for terpene production.

GSL Compiler

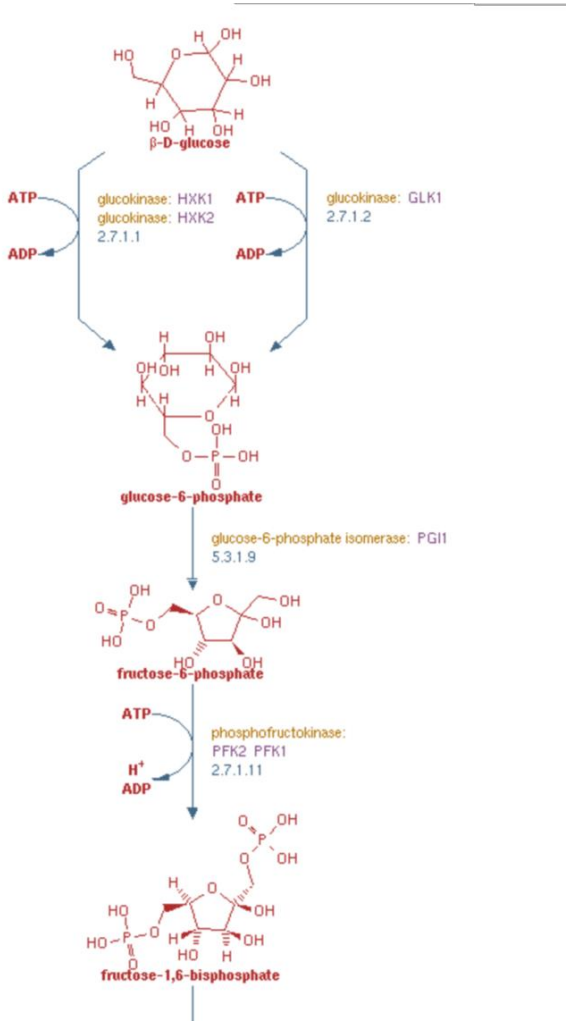


Genotype Specification Language

10.1021/acssynbio.5b00194

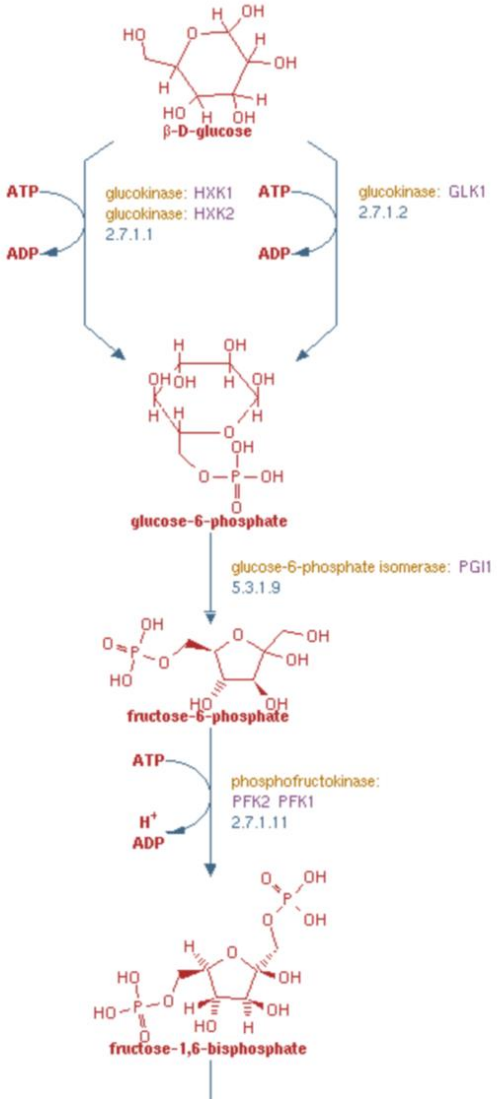
Metabolic / Genetic Engineering 101

Biochemical reactions in cell create many molecules from starting material e.g. sugar

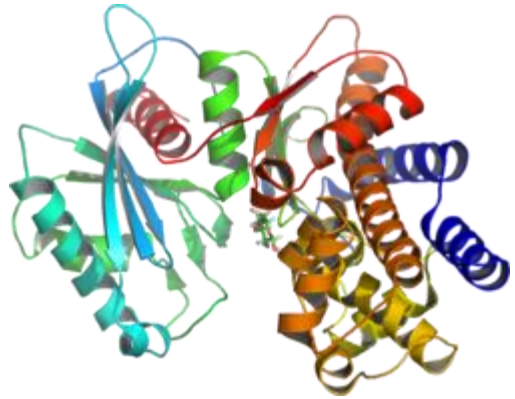


Credit: http://pathway.yeastgenome.org/YEAST/NEW-IMAGE?type=PATHWAY&object=GLUCFERMEN-PWY&detail-level=3&detail-level=2

Reactions are carried out by enzymes

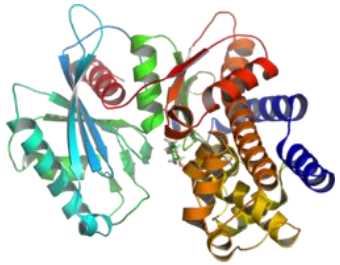


https://en.wikipedia.org/wiki/Glucokinase



1 MVHLGPKKPQ AR**K**G**S**MADVP KELMDEIHQL EDMFTVDSET LRKVVKHFID ELNKGLTKKG
61 GNIPMIPGWV MEFPTGKESG NYLAIDLGGT NLRVVLVKLS GNH**T**FD**T**TQS **K**YKLPHDMRT
121 TKHQEELWSF IADSLKDFMV EQELLNTKDT LPLGFTF**S**YP ASQN**K**INEGI LQRWTKGFDI
181 PNVEGHDVVP LLQNEISKRE LPIEIVALIN DTVGTLIASY YTDPETKMGV IFGTGVNGAF
241 YDVV**S**DIEKL EGKLADDIPS N**S**PMAINCEY G**S**FDNEHLVL PRT**K**YDVAVD EQ**S**PRPGQQA
301 FEKMTSGYYL GELLRLVLLE LNEKGLMLKD QDLSKLKQPY IMDTSYPARI EDDPFENLED
361 TDDIFQ**K**DFG V**K**TTLPERKL IRRLCELIGT RAARLAVCGI AAICQKRGYK TGHIAADG**S**V
421 YN**K**YPGFKEA AAKGLRDIYG WTGDASKDPI TIVPAEDGSG AGAAVIAALS EKRIAEGKSL
481 GIIGA

Instructions to build the Enzyme (which is a protein) are encoded in DNA in a gene



enzyme

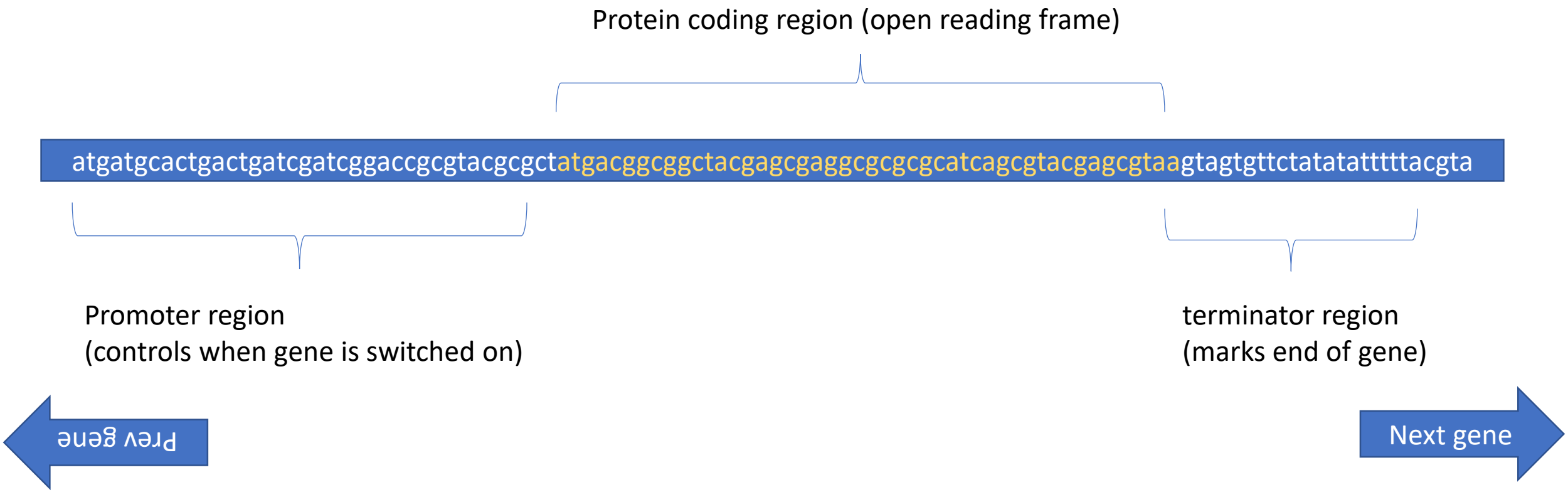
```
ATGGTTCATTTAGGTCCAAAGAAACCACAGGCTAGAAAGGGTTCCATGGCTGATGTGCCCAAGGAATTGATGGATGAAATTCATCAG
TTGGAAGATATGTTTACAGTTGACAGCGAGACCTTGAGAAAGGTTGTTAAGCACTTTATCGACGAATTGAATAAAGGTTTGACAAAG
AAGGGAGGTAACATTCCAATGATTCCCGGTTGGGTCATGGAATTCCCAACAGGTAAAGAATCTGGTAACTATTTGGCCATTGATTTG
GGTGGTACTAACTTAAGAGTCGTGTTGGTCAAGTTGAGCGGTAACCATACCTTTGACACCACTCAATCCAAGTATAAACTACCACAT
GACATGAGAACCACTAAGCACCAAGAGGAGTTATGGTCCTTTATTGCCGACTCTTTGAAGGACTTTATGGTCGAGCAAGAATTGCTA
AACACCAAGGACACCTTACCATTAGGTTTCACCTTCTCGTACCCAGCTTCCCAAAACAAGATTAACGAAGGTATTTTGCAAAGATGG
ACCAAGGGTTTCGATATTCCAAATGTCGAAGGCCACGATGTCGTCCCATTGCTACAAAACGAAATTTCCAAGAGAGAGTTGCCTATT
GAAATTGTAGCATTGATTAATGATACTGTTGGTACTTTAATTGCCTCATACTACACTGACCCAGAGACTAAGATGGGTGTGATTTTC
GGTACTGGTGTCAACGGTGCTTTCTATGATGTTGTTTCCGATATCGAAAAGTTGGAGGGCAAATTAGCAGACGATATTCCAAGTAAC
TCTCCAATGGCTATCAATTGTGAATATGGTTCCTTCGATAATGAACATTTGGTCTTGCCAAGAACCAAGTACGATGTTGCTGTCGAC
GAACAATCTCCAAGACCTGGTCAACAAGCTTTTGAAAAGATGACCTCCGGTTACTACTTGGGTGAATTGTTGCGTCTAGTGTTACTT
GAATTAAACGAGAAGGGCTTGATGTTGAAGGATCAAGATCTAAGCAAGTTGAAACAACCATACATCATGGATACCTCCTACCCAGCA
AGAATCGAGGATGATCCATTTGAAAACTTGGAAGATACTGATGACATCTTCCAAAAGGACTTTGGTGTCAAGACCACTCTGCCAGAA
CGTAAGTTGATTAGAAGACTTTGTGAATTGATCGGTACCAGAGCTGCTAGATTAGCTGTTTGTGGTATTGCCGCTATTTGCCAAAAG
AGAGGTTACAAGACTGGTCACATTGCCGCTGACGGTTCTGTCTATAACAAATACCCAGGTTTCAAGGAAGCCGCCGCTAAGGGTTTG
AGAGATATCTATGGATGGACTGGTGACGCAAGCAAAGATCCAATTACGATTGTTCCAGCTGAGGATGGTTCAGGTGCAGGTGCTGCT
GTTATTGCTGCATTGTCCGAAAAAAGAATTGCCGAAGGTAAGTCTCTTGGTATCATTGGCGCTTAA
```

DNA (4 letter alphabet)

```
1 MVHLGPKKPQ ARKGSMADVP KELMDEIHQL EDMFTVDSET LRKVVKHFID ELNKGLTKKG
61 GNIPMIPGWV MEFPTGKESG NYLAIDLGGT NLRVVLVKLS GNHTFDTTQS KYKLPHDMRT
121 TKHQEELWSF IADSLKDFMV EQELLNTKDT LPLGFTFSYP ASQNKINEGI LQRWTKGFDI
181 PNVEGHDVVP LLQNEISKRE LPIEIVALIN DTVGTLIASY YTDPETKMGV IFGTGVNGAF
241 YDVVSDIEKL EGKLADDIPS NSPMAINCEY GSFDNEHLVL PRTKYDVAVD EQSPRPGQQA
301 FEKMTSGYYL GELLRLVLLE LNEKGLMLKD QDLSKLKQPY IMDTSYPARI EDDPFENLED
361 TDDIFQKDFG VKTTLPERKL IRRLCELIGT RAARLAVCGI AAICQKRGYK TGHIAADGSV
421 YNKYPGFKEA AAKGLRDIYG WTGDASKDPI TIVPAEDGSG AGAAVIAALS EKRIAEGKSL
481 GIIGA
```

Protein (20 letter alphabet)

Simple gene architecture (like a subroutine)



Genes are organized into long strings of DNA called chromosomes

F# genome viewer (Courtesy: Amyris)

Yeast genome has
~ 6000 genes organized into 17 separate chromosomes (volumes) of data (=~ 17 DLLs)

Entire genome (all DNA)
has 12 million
letters (ATCGs).
(3Mb with trivial compression)

Smaller than this
PowerPoint presentation

Metabolic engineering problem ..

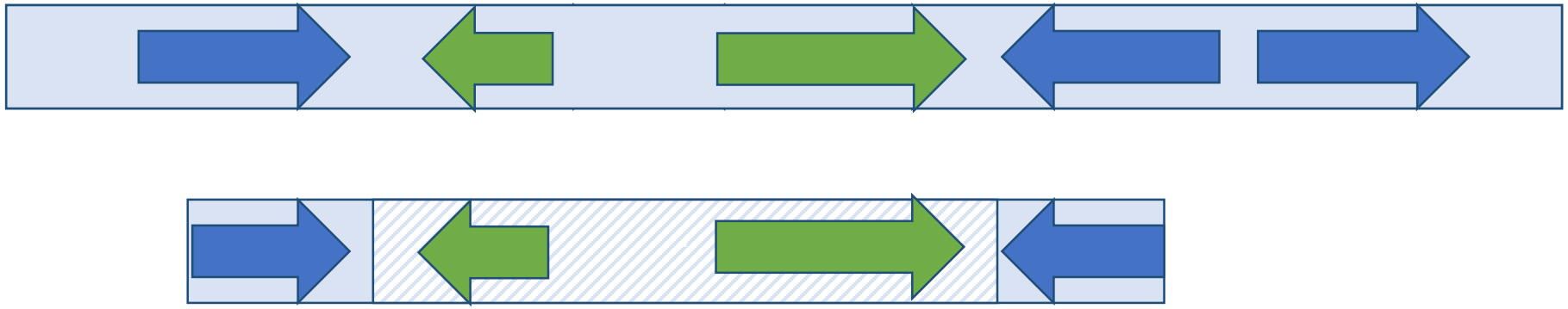
We can reduce the engineering problem to the application of a set of changes to the yeast genome (DNA sequence). We typically

- delete genes
- introduce genes (enzymes) from other sources
- increase gene activation levels "upregulate"
- decrease gene activation (expression) levels "down regulate"
- edit genes/enzymes to slightly change their reaction

This can all be implemented as patches to the yeast binary…

Yeast has a pretty sweet patch process

ATGACTAGTCTGATGTGATCTGATCTGATAT….



So you’re writing a genetic programming domain specific language, why F#?

Respectable but not most popular

Programming Language Interview

“cat” + 123?

“cat”+123 Brings out personality of a language

```
print "cat" + 12

perl t.pl
12
```

```
print "cat" + 12

python t.py
Traceback (most recent call last):
  File "t.py", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

```
"cat"+123
"cat123"

"cat"-123
NaN
```

```
using namespace std;
#include <iostream>

int main(int,char **)
{
        cout << "cat" + 12 << endl;
        return 0;
}

dmplatt@nirvana~> g++ t.cc
dmplatt@nirvana~> ./a.out

dmplatt@nirvana~>
```

```
module fs4p

let x = "cat" + 123
```

The type 'int' does not match the type 'string'

My minimal shopping list for a programming language

• Strongly typed	- reduces errors
• Semantic whitespace	- reduces errors
• Functional first	- reduces errors
• Option types	- reduces errors
• Units of measure	- reduces errors
• Type inference	- reduces RSI, makes #1 palatable
• Static typing	- fast
• Good parallel primitives	- lazy fast
• Rich built in data types	- reduces development time
• Type providers	- reduces development time
• Open source	- reduces dependency
• Decent library support	- don’t want to roll my own db driver
• Language iterop (*)	- increases library support

Genotype Specification Language Crash Course

GSL hierarchy

Conceptually want different languages for different levels of specificity.
Current implementation is mostly level 1 with some level 2 concepts. Can implement higher level concepts by translation into lower level designs.

Level 3: gNeutral^ ; pStrong>@EC2.3.1.9

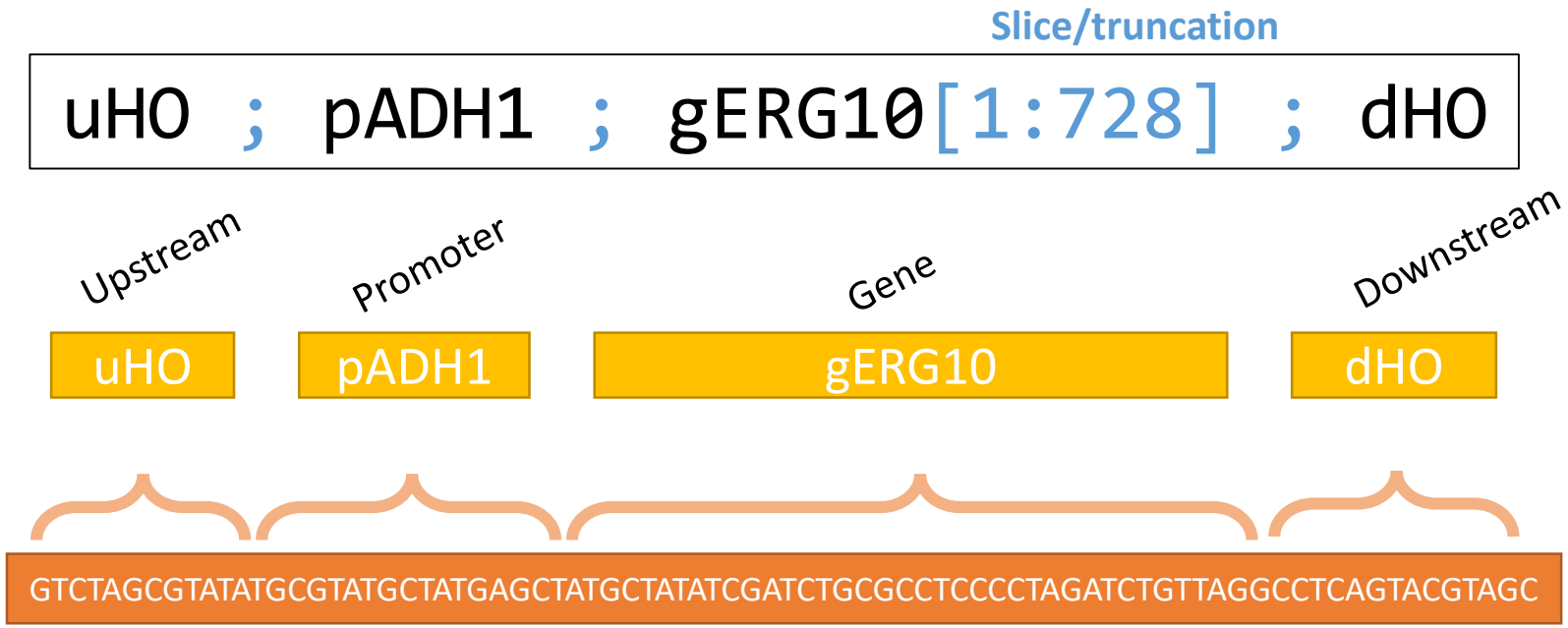
Level 2: gHO^ ; pTDH3 > gERG10

Level 1: uHO ; pTDH3 ; mERG10 ; ### ; dHO

Level 0: DNA ACCTTTTTTGTGCGTGTATT..

GSL: A simple example

- Default parts from the native genome of the target organism are available in the name space.



More complex example

- State of the art is cut and paste or drag and drop

```
// Yeast terpene design
// Level 1 Syntax
// Part definitions
let pGAL1_10 = gGAL1[-668:-1]
let truncHMGR = /ATGG/ {#rabitstart }; gHMG1[1586:~200E]

// Locus engineering
uERG9 ; ### ; pMET3 ; gERG9[1:~500]
uLEU2 ; ### ;  !mERG8 ; @pGAL1_10 ; mMVD1 ; dLEU2
gHIS3[~-1000:-500]; ### ; !mERG10 ; @pGAL1_10 ; mERG12 ; gHIS3[-500:~500]
gADE1[~-1000:-500]; ### ; !mIDI1  ; @pGAL1_10 ; @truncHMGR ; gADE1[-500:~500]
gURA3[~-1000:-500]; ### ; !mERG13 ; @pGAL1_10 ; @truncHMGR ; gURA3[-500:~500]
gTRP1[~-1000:-500]; ### ; !@truncHMGR ;  @pGAL1_10 ; mERG20 ;  gTRP1[-500:~500]

// Level 2 syntax
pMET3>gERG9
pGAL1>mERG10; pGAL10>mERG12
pGAL1>mIDI1; pGAL10>@truncHMGR
pGAL1>@truncHMGR; pGAL10>mERG20 pGAL1>mERG8; pGAL10>mMVD1
```

Figure 7. Example yeast design for terpene production.

More detail of GSL compiler available in this publication

http://pubs.acs.org/doi/abs/10.1021/acssynbio.5b00194
http://bit.ly/2yG5hDt

ACS SyntheticBiology

Research Article

pubs.acs.org/synthbio

Genotype Specification Language

Erin H. Wilson,[†] Shiori Sagawa,[†] James W. Weis,[†] Max G. Schubert,[†] Michael Bissell, Brian Hawthorne, Christopher D Reeves, Jed Dean, and Darren Platt*

Amyris, Inc., 5885 Hollis Street, Suite 100, Emeryville, California 94608, United States

ABSTRACT: We describe here the Genotype Specification Language (GSL), a language that facilitates the rapid design of large and complex DNA constructs used to engineer genomes. The GSL compiler implements a high-level language based on traditional genetic notation, as well as a set of low-level DNA manipulation primitives. The language allows facile incorporation of parts from a library of cloned DNA constructs and from the "natural" library of parts in fully sequenced and annotated genomes. GSL was designed to engage genetic engineers in their native language while providing a framework for higher level abstract tooling. To this end we define four language levels, Level 0 (literal DNA sequence) through Level 3, with increasing abstraction of part selection and construction paths. GSL targets an intermediate language based on DNA slices that translates efficiently into a wide range of final output formats, such as FASTA and GenBank, and includes formats that specify instructions and materials such as oligonucleotide primers to allow the physical construction of the GSL designs by individual strain engineers

```
gHO^ ; pTDH3 > gERG10
uHO ; pTDH3 ; mERG10 ; ### ; dHO
gYNG2$C227Y
```

GSL Internals / Code Vignettes

https://github.com/Amyrisinc

GslCore

Core library and basic plug-ins for the Amyris Genotype Specification Language (GSL) compiler.

F# ★ 1 Updated 7 days ago

Gslc

Genotype Specification Language command-line compiler application.

F# 3 Updated on Jun 26

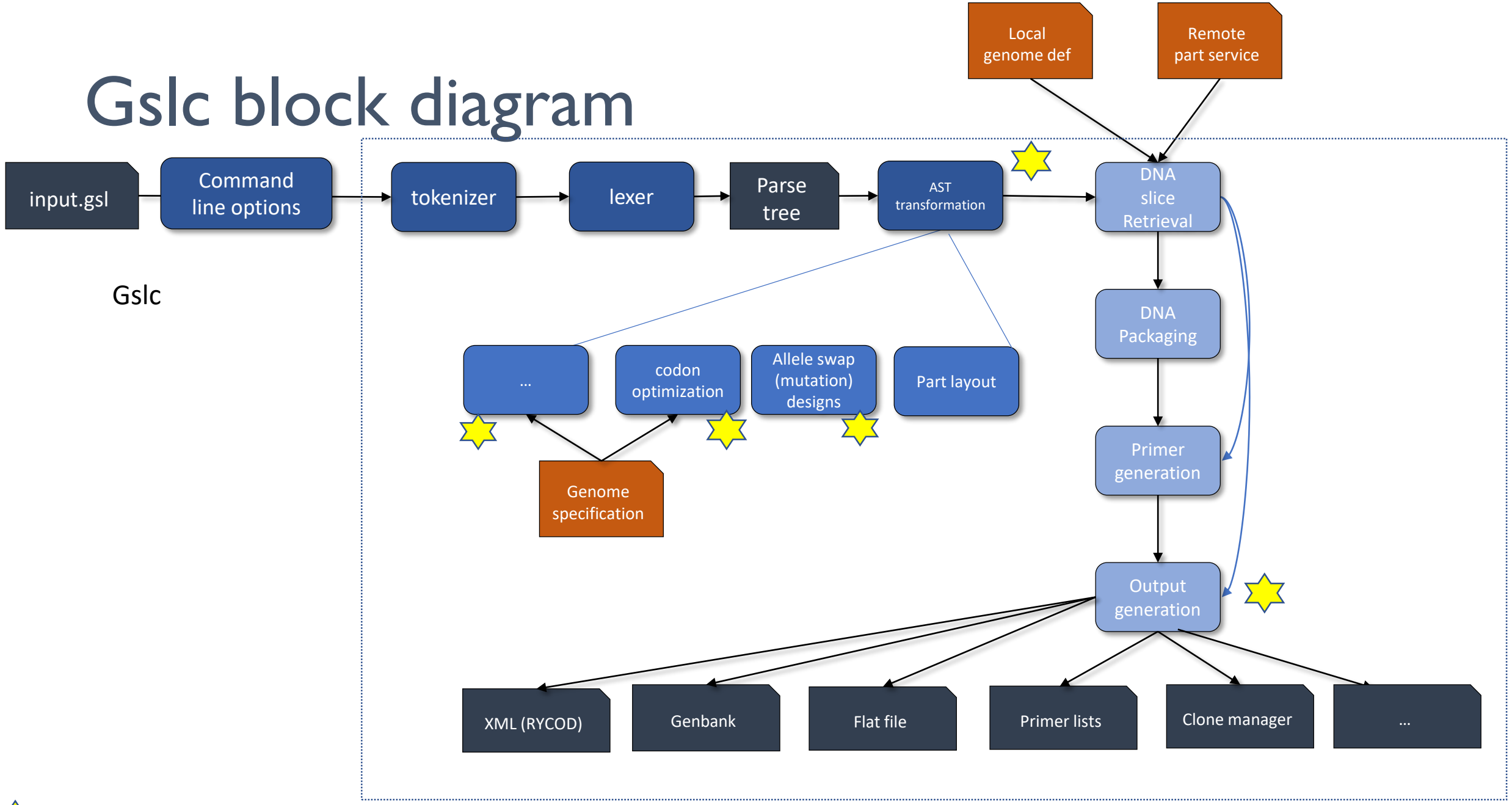
https://github.com/Amyris/AmyrisBio

AmyrisBio

Amyris F# Computational Biology Library

F# ★ 2 1

Gslc block diagram



Code Vignettes
Fsyacc/Fslex parser

Sample of FSLex (lexer)

```
19  let num = ['0'-'9']+
20  let intNum = '-'? num
21  let ident = ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*
22  let pval = [^ ';' '#' '}' ' ' '\t' '\n' '\r'][^ ';' '}' ' ' '\t' '\n' '\r']*
23  let pvalVariable = '&' ident
24  let pvalAllowSemicolons = [^ '#' '}' ' ' '\t' '\n' '\r'][^ '}' ' ' '\t' '\n' '\r']*
25  let whitespace = ' ' | '\t'
26  let newline = '\n' | '\r' '\n' | '\r'
27  let pname =  '#'['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*
28  let aa = 'A'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'K'|'L'|'M'|'N'|'P'|'Q'|'R'|'S'|'T'|'V'|'W'|'Y'|'*'  // Amino acids
29  let dna = 'A' | 'T' | 'C' | 'G' | 'a' | 't' | 'c' | 'g'
30  let linker = ['0'-'9' 'A'-'Z']
31  let string = ['\"'][^ '\"']*['\"']  // Quoted strings
32
33  // Each set of lexer rules also accepts a function that sets which tokenizer the running lexer is using.
```

Sample of FSYacc (grammar)

```
234  // name mangle to keep future refactorings clean
235  CompletePart:
236      | PartFwdRev {$1}
237
238  // ==================
239  // modifications to parts
240  // ==================
241
242  RelPos:
243      | IntExp      { ($1, None) } // rather than creating these as AST nodes here, we pass them on to Slice so it can
244      | IntExp ID   { ($1, (Some $2)) }
245
246  Slice: // Slice ranges of form  [ a : b] where a or b may be qualified with ~ to be approximate
247      | OPENSQBRACKET RelPos COLON RelPos CLOSESQBRACKET                { createParseSlice $2 $4 false false }
248      | OPENSQBRACKET TILDE RelPos COLON RelPos CLOSESQBRACKET          { createParseSlice $3 $5 true false }
249      | OPENSQBRACKET RelPos COLON TILDE RelPos CLOSESQBRACKET          { createParseSlice $2 $5 false true }
250      | OPENSQBRACKET TILDE RelPos COLON TILDE RelPos CLOSESQBRACKET    { createParseSlice $3 $6 true true }
251
252  Mod:
253      | DNAMUTATION    { createMutation $1 NT }
254      | AAMUTATION     { createMutation $1 AA }
```

Representing the parse tree

- Want every tree node to have source code position information
- Otherwise heterogeneous types
- Node definition:

```fsharp
// ============================
// Wrapper type for all AST nodes.
// ============================

/// Wrapper type for every AST node.
/// This enables adding extensible metadata to the AST for tracking things such as source code position.
[<CustomEquality>][<NoComparison>]
type Node<'T when 'T: equality> = {x: 'T; pos: SourcePosition option}
    with
    /// Override equality to ignore source code position.  We just care about semantic comparison.
    /// This is mostly to aid in testing.  We shouldn't need to be comparing generic AST nodes during parsing.
    override this.Equals other =
        match other with
        | :? Node<'T> as o -> this.x = o.x
        | _ -> false
```

Tree build from type parameterized Nodes

```
and AstNode =
    // leaf nodes that hold values
    | Int of Node<int>
    | Float of Node<float>
    | String of Node<string>
    // docstrings
    | Docstring of Node<string>
    // variable leaf node
    | TypedVariable of Node<string*GslVarType>
    // variable binding
    | VariableBinding of Node<VariableBinding>
    // typed value
    | TypedValue of Node<GslVarType*AstNode>
    // Simple operations on values
    | BinaryOperation of Node<BinaryOperation>
    | Negation of Node<AstNode>
    // Slicing
    | ParseRelPos of Node<ParseRelPos> // parsed relative position, may contain variables and hasn't been built yet
    | RelPos of Node<RelPos> // built relative position, fully specified
    | Slice of Node<ParseSlice>
    // non-slice part mods
    | Mutation of Node<Mutation>
    | DotMod of Node<string>
    | HetBlock of Node<unit>
    | Gene of Node<ParseGene>
    | Assembly of Node<AstNode list>
    // AST nodes for Level 2 syntax support
    | L2Id of Node<L2Id>
    | L2Element of Node<L2Element>
    | L2Expression of Node<L2Expression>
```

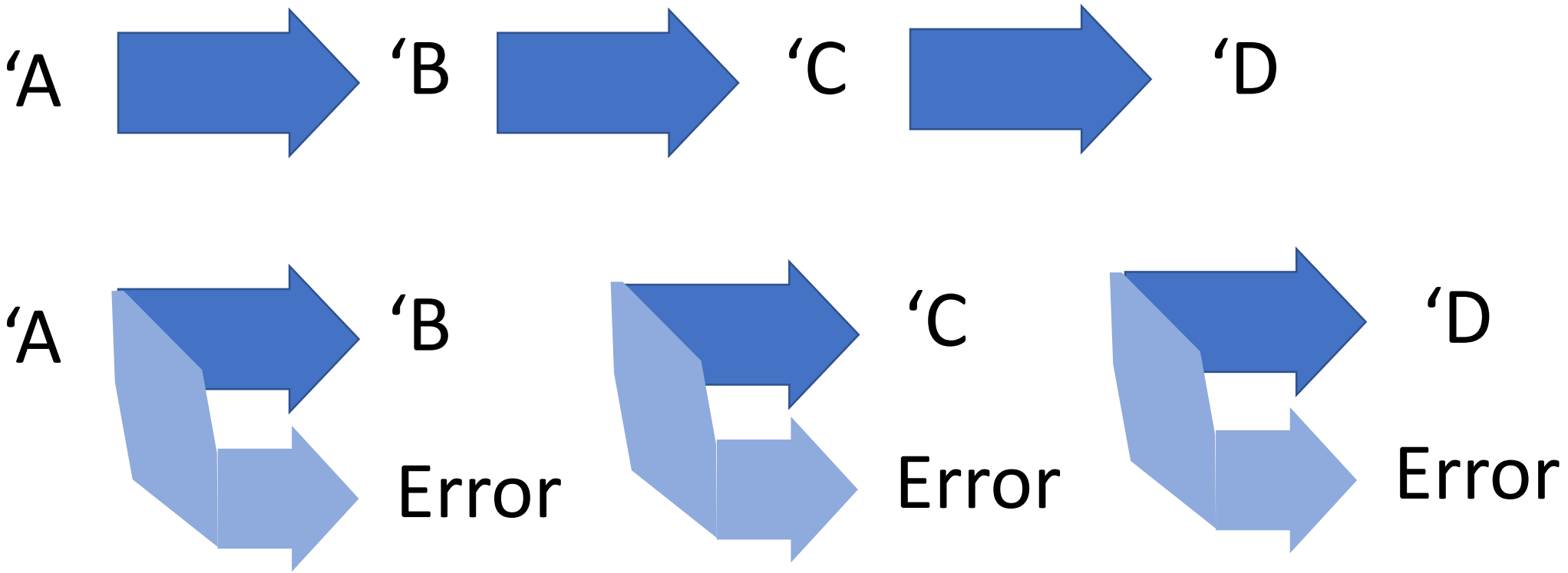
Some recursive

Railroad Oriented Programming

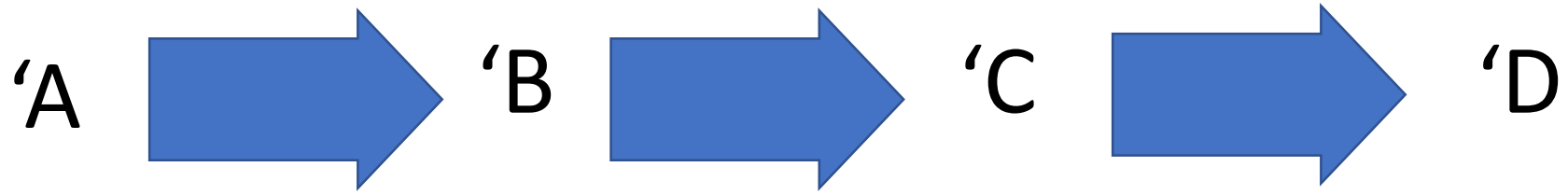
https://fsharpforfunandprofit.com/rop/
(Scott Wlaschin)

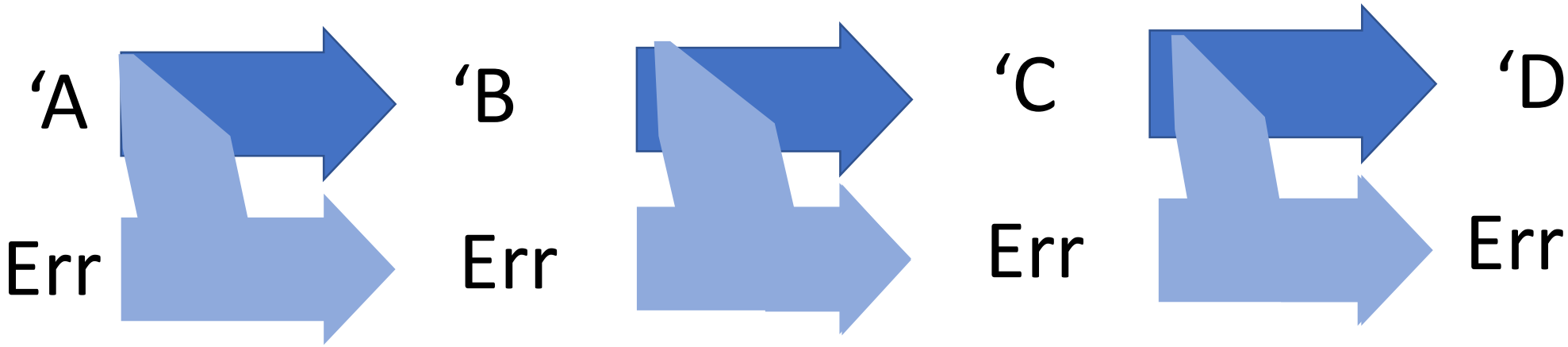
Railroad Oriented Programming

https://fsharpforfunandprofit.com/rop/ (Scott Wlaschin)



Railroad Oriented Programming





In practice

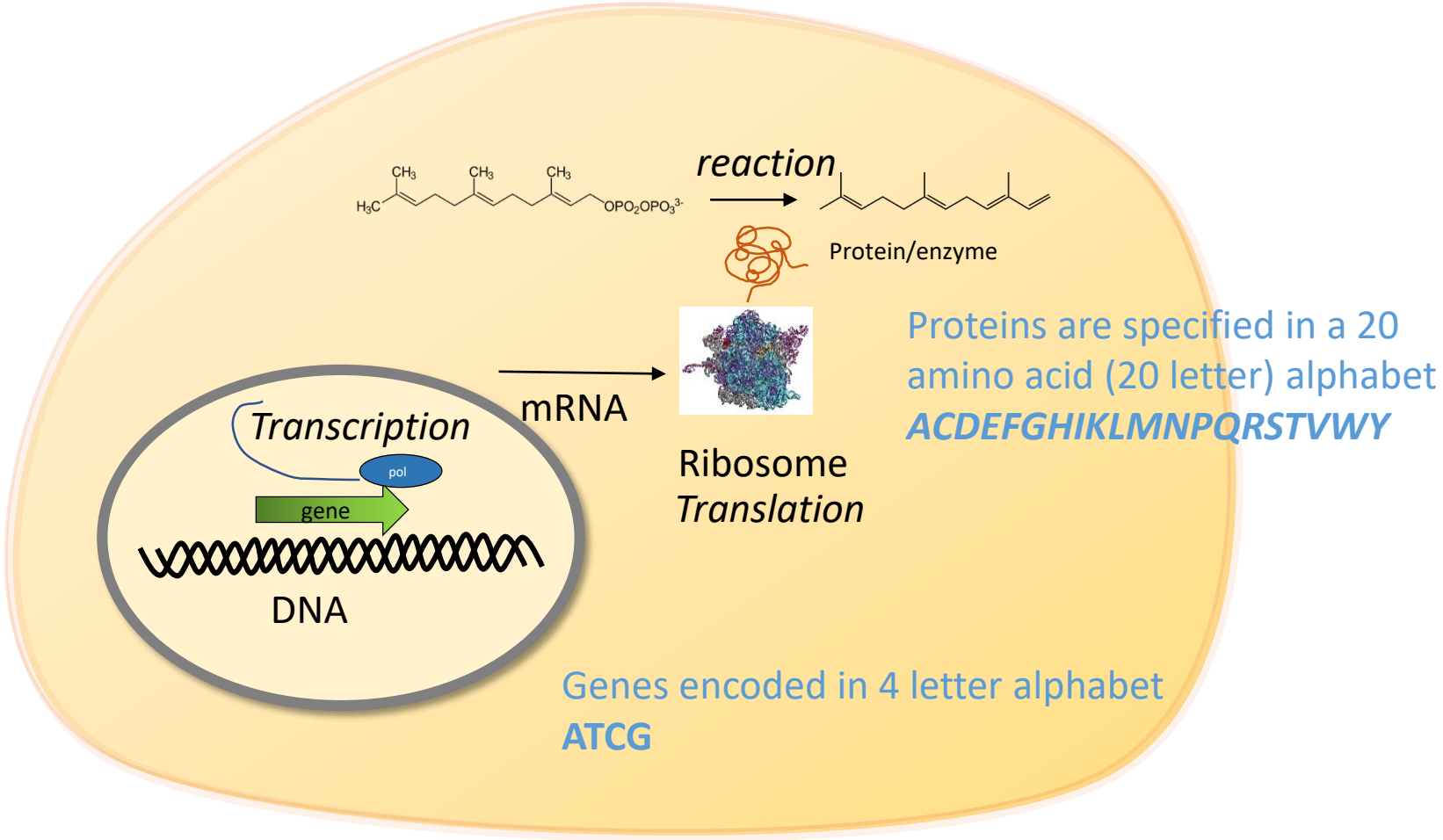
```
/// Phase 1 is everything before bioinformatics really gets involved.
let phase1 legalCapas =
    linters
    >=> immediateValidations
    >=> checkRecursiveCalls
    >=> resolveVariables
    >=> inlineFunctionCalls
    >=> stripFunctions
    >=> resolveVariablesStrict
    >=> stripVariables
    >=> reduceMathExpressions
    >=> buildPragmas legalCapas
    >=> collectWarnings
    >=> buildRelativePositions
    >=> expandRoughageLines // inline roughage expansion is pretty simple so we always do it
    >=> flattenAssemblies
    >=> (validate checkMods)
    >=> stuffPragmasIntoAssemblies
```

- Chessie ROP implementation
- Compose pipeline steps with >=> operator
- c.f. >> operator for function composition
- Error handling consistent
- Easy to unit test components
- Easy to customize compiler

```
[<TestFixture>]
type TestPragmasAST() =

    let pragmaBuildPipeline =
        resolveVariables
        >=> inlineFunctionCalls
        >=> stripFunctions
        >=> resolveVariablesStrict
        >=> stripVariables
        >=> reduceMathExpressions
        >=> (buildPragmas (["capa1"; "capa2"] |> Set.ofList))

    let compilePragmas = compile pragmaBuildPipeline
```

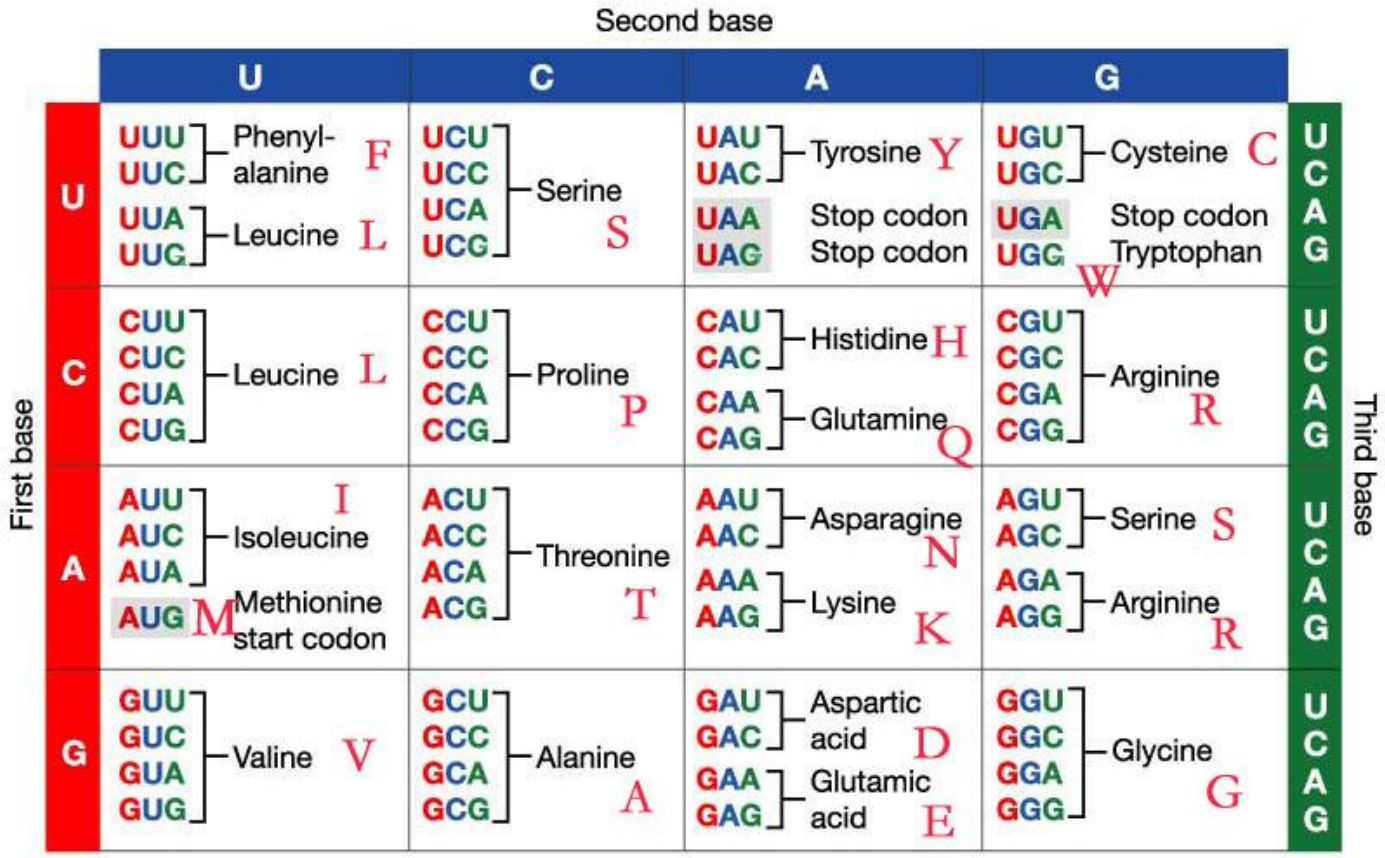
Code Vignettes
Genetic Algorithms for Reverse Translation

Central Dogma DNA -> RNA -> Protein -> Reaction



20 amino acids encoded by 3 DNA letters (codon)

- Redundant code, which means we have choices when we pick a codon to represent an amino acid
- Choices widely believed (known) to affect efficiency



Yeast Uses the 64 possibilities differentially

Codon (count/1000) (absolute # uses in proteins)

TTT 26.1(170666)	TCT 23.5(153557)	TAT 18.8(122728)	TGT 8.1(52903)
TTC 18.4(120510)	TCC 14.2(92923)	TAC 14.8(96596)	TGC 4.8(31095)
TTA 26.2(170884)	TCA 18.7(122028)	TAA 1.1(6913)	TGA 0.7(4447)
TTG 27.2(177573)	TCG 8.6(55951)	TAG 0.5(3312)	TGG 10.4(67789)
CTT 12.3(80076)	CCT 13.5(88263)	CAT 13.6(89007)	CGT 6.4(41791)
CTC 5.4(35545)	CCC 6.8(44309)	CAC 7.8(50785)	CGC 2.6(16993)
CTA 13.4(87619)	CCA 18.3(119641)	CAA 27.3(178251)	CGA 3.0(19562)
CTG 10.5(68494)	CCG 5.3(34597)	CAG 12.1(79121)	CGG 1.7(11351)
ATT 30.1(196893)	ACT 20.3(132522)	AAT 35.7(233124)	AGT 14.2(92466)
ATC 17.2(112176)	ACC 12.7(83207)	AAC 24.8(162199)	AGC 9.8(63726)
ATA 17.8(116254)	ACA 17.8(116084)	AAA 41.9(273618)	AGA 21.3(139081)
ATG 20.9(136805)	ACG 8.0(52045)	AAG 30.8(201361)	AGG 9.2(60289)
GTT 22.1(144243)	GCT 21.2(138358)	GAT 37.6(245641)	GGT 23.9(156109)
GTC 11.8(76947)	GCC 12.6(82357)	GAC 20.2(132048)	GGC 9.8(63903)
GTA 11.8(76927)	GCA 16.2(105910)	GAA 45.6(297944)	GGA 10.9(71216)
GTG 10.8(70337)	GCG 6.2(40358)	GAG 19.2(125717)	GGG 6.0(39359)

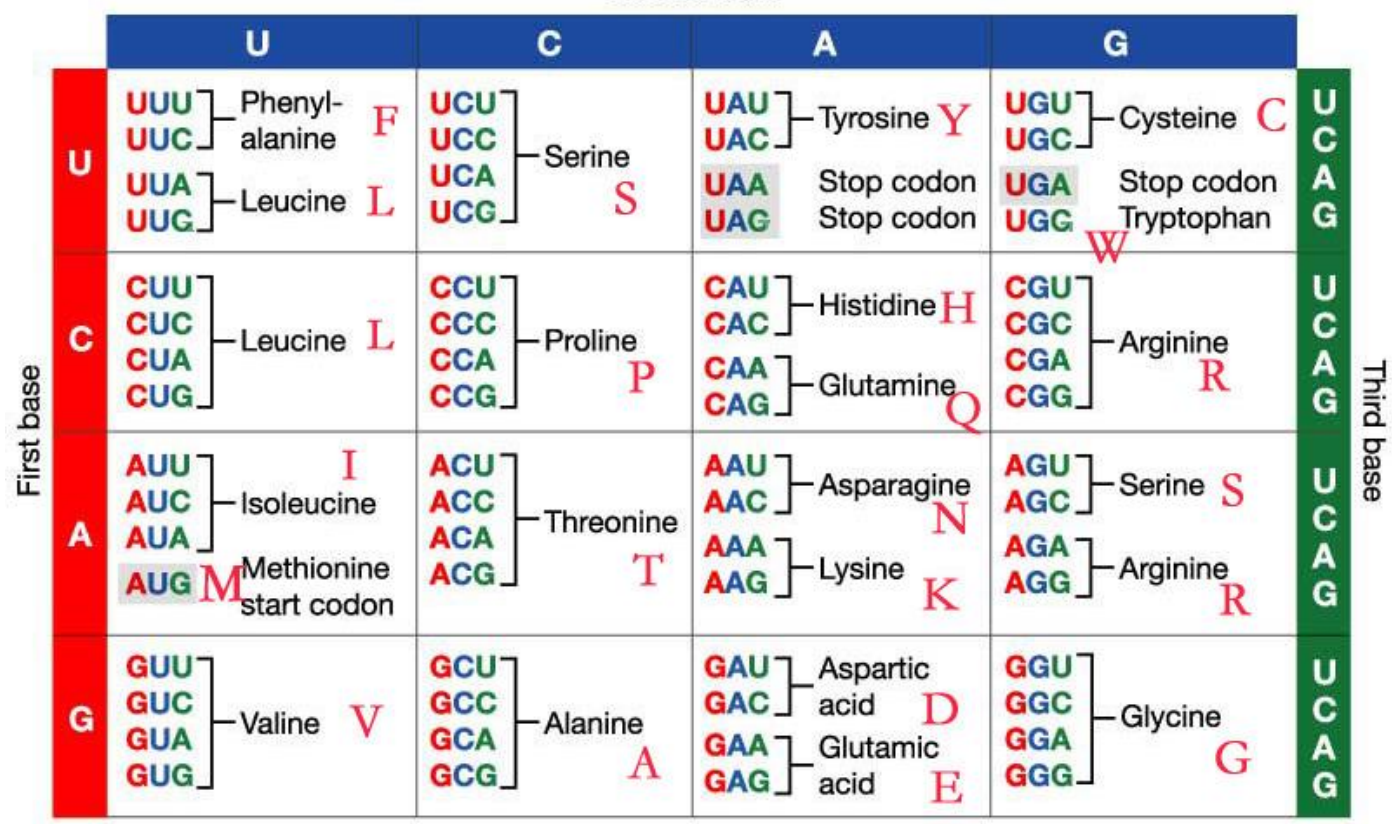
e.g. Glycine (G)

Reverse Translation Problem in a Nutshell

- Pick a codon for each amino acid in a protein

M V S V

ATG GTG AGC



T=U

Generating a DNA sequence from a protein

- Typical rules
  - Pick codons matching the desired protein sequence
  - Use only codons with minimum frequency >= M
  - Avoid strings of single nucleotides longer than L
    e.g. AAAAAAAAAAAAA

  - Avoid a list of specific short sequences e.g. GATTACCA or GGTTCGT
  - Avoid tandem repeats of length > R

  E.g.
  TGACTAGCTCAGgatgctagctagctagcTGACTGATCGATCGTACGTAGCTACGATCGACGGGCTAgatgctagctagctagcTAGCTAGCAC

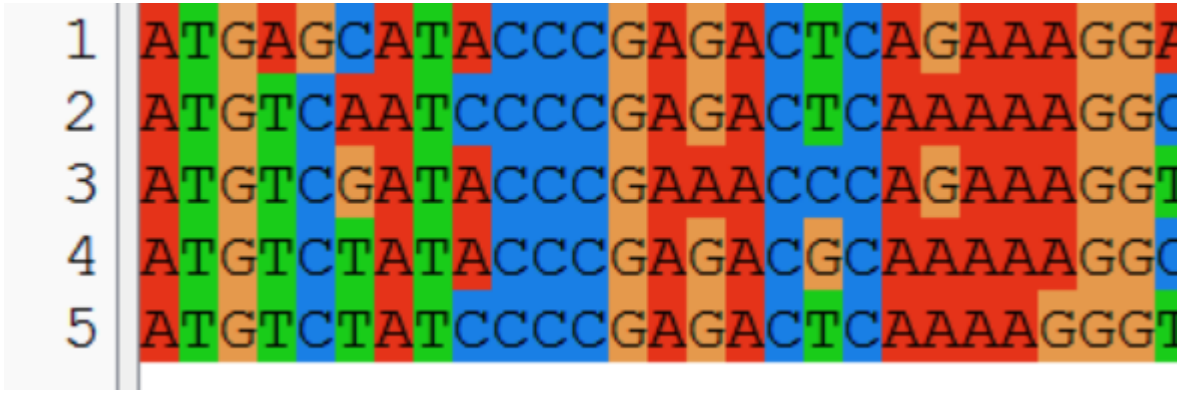
- Solution

  Use a genetic algorithm to evolve desired sequence

Genetic Algorithm

MSIPETQKGVIFYESHGKLE

1. Generate initial random population by sampling



.....

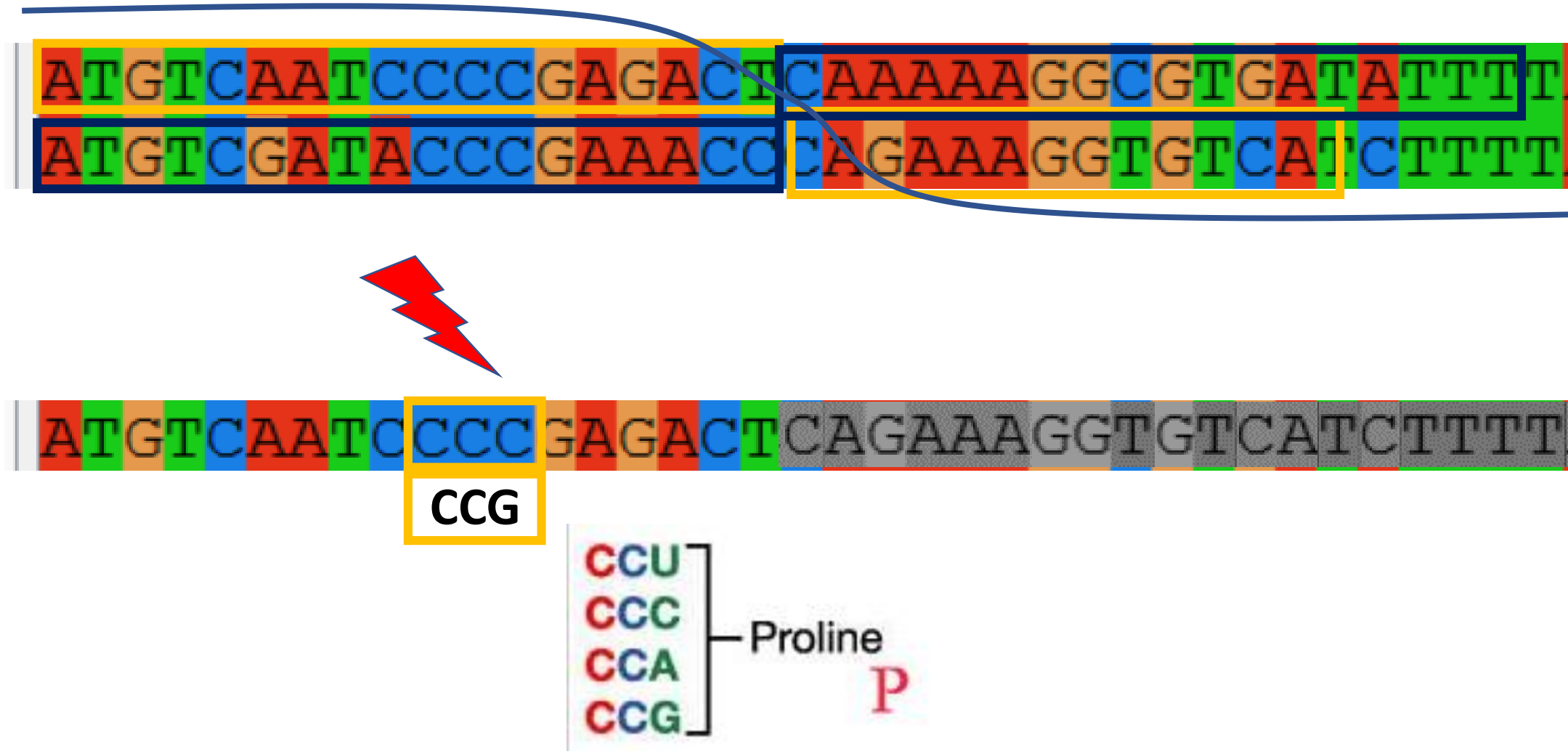
2. Score population for rule breaking instances, ranking candidates by fitness

3. Select pairs of sequences according to fitness

4. Cross over sequences, mutate problematic areas

5. Generate new population from mating pairs of sequence

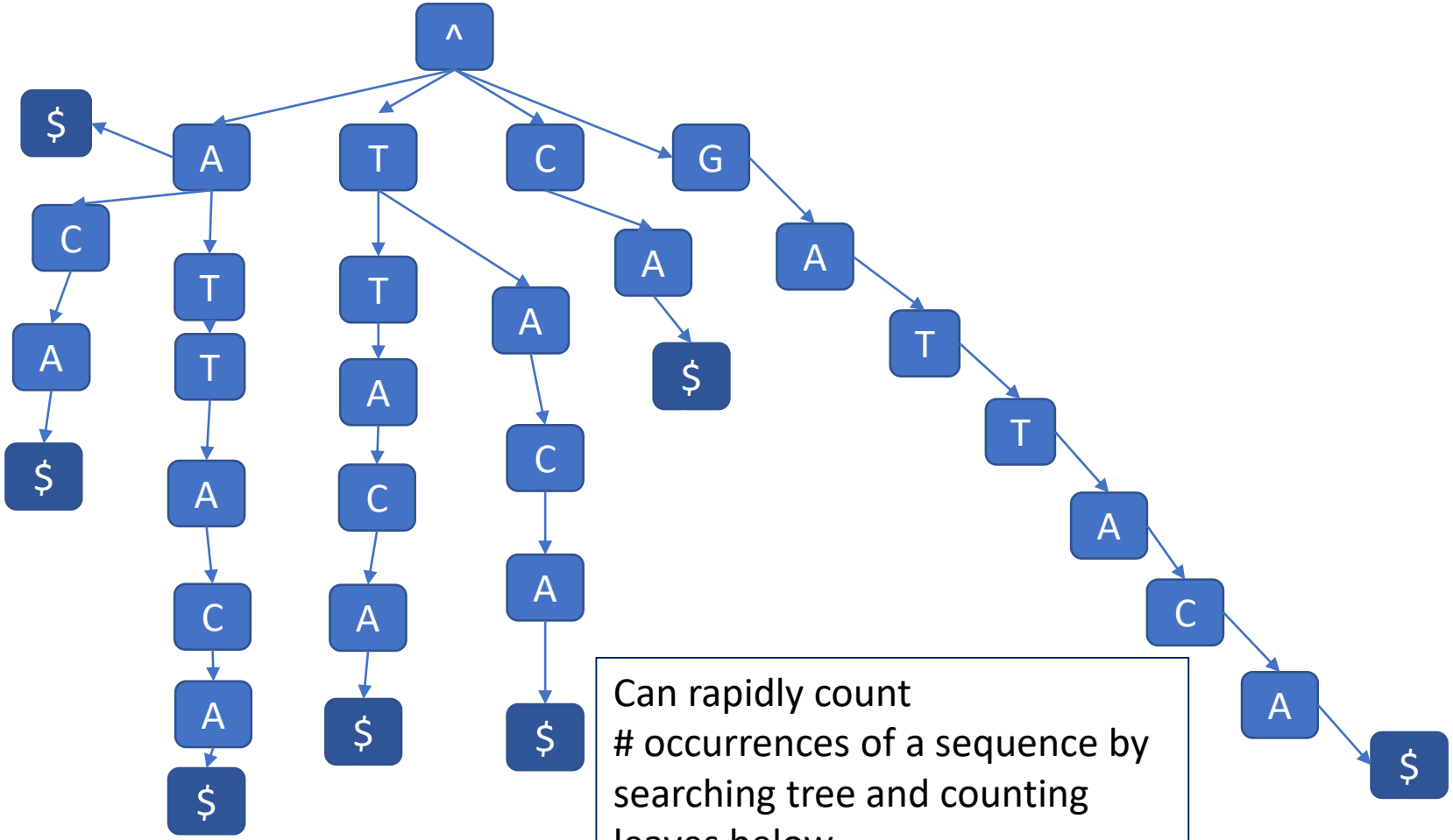
Crossover and mutation



Rapid repeat detection

- Build suffix tree
  - E.g given sequence GATTACCA, tree should contain

GATTACA
ATTACA
TTACA
TACA
ACA
CA
A



Rapid repeat detection

- High performance F# suffix tree implementation
- Indexing whole yeast genome of ~ 12 million letters for a 40 million node tree in 79 seconds including creation of binary output file
- Core construction and search algorithm just 375 lines
- Full implementation including memory mapped binary file format to load tree on demand is 679 lines.
- Uses structs heavily to get compact memory usage

```
$ time indexing
REF project = cenpk
load fasta sequence from c:\seq\GSL\gslc\lib\cenpk\cenpk.fsa
build full reference string
Indexed 12,156,182 bases with spacers
build suffix tree
saving .\suffixTree.st
Suffix tree of 40,209,386 nodes and 40,209,385 edges obtained in 78.993999 seconds
done..

real    1m19.349s
user    0m0.000s
sys     0m0.015s
```

```
[<Struct>]
type Edge(_firstChar:int,_lastChar:int,_startNode:int,_endNode:int) =
    member x.firstChar = _firstChar
    member x.lastChar = _lastChar
    member x.startNode = _startNode
    member x.endNode = _endNode
```

https://github.com/Amyris/AmyrisBio/blob/master/src/AmyrisBio/ukk.fs

Code Vignettes
Melting temperature calculation
F# Units of Measure

A DNA duplex will "melt" at a certain temperature

- There are reasonably well established mathematical models to estimate this temperature for a given DNA sequence

```
5' CTAGCCATCACAAATCGGTAT 3'
   | | | | | | | | | | | | | | | | | | | |
3' GATCGGTAGTGTTTAGCCATA 5'
```

```
CTAGCCATCACAAATCGGTAT

GATCGGTAGTGTTTAGCCATA
```

- Surprisingly hard to get the calculation right, and there are many broken Tm (temperature of melting) calculators on the internet.

https://en.wikipedia.org/wiki/Nucleic_acid_thermodynamics

GSL Tm calculation using units of measure

```
/// Tools for oligonucleotide design
namespace Amyris
open System

/// Core primer Tm calculation
module tmcore =
    [<Measure>] type M // Molar
    [<Measure>] type uM // micromolar
    [<Measure>] type mM // millimolar
    [<Measure>] type nM // nanomolar
    [<Measure>] type K // kelvin
    [<Measure>] type C // Celsius
    [<Measure>] type Cal // Calories
    let inline K2C (k:float<K>) = (k*1.0<C/K>) - 273.5<C>
    let inline C2K (c:float<C>) = (c*1.0<K/C>) + 273.5<K>
    let inline mM2M (c:float<mM>) = c*0.001<M/mM>
    let inline uM2M (c:float<uM>) = c/1000000.0<uM/M>
    let inline nM2M (c:float<nM>) = c/1000000000.0<nM/M> // Table 2: http://pubs.acs.org/doi/pdf/10.1021/bi702363u
    let private a = 3.92E-5<1/K>
    let private b = -9.11E-6<1/K>
    let private c = 6.26E-5<1/K>
    let private d = 1.42E-5<1/K>
    let private e = -4.82E-4<1/K>
    let private f = 5.25E-4<1/K>
    let private g = 8.31E-5<1/K>
```

http://bit.ly/2wVJWDM

delta H and delta S calculations for nucleotide pairs

```
type HS = { h : float<Cal/M> ; s: float<Cal/K/M>}
/// Santa lucia PNAS Feb 17 1998 vol 95 no 3. 1460-1465
let HSpair (a:char) (b:char) : HS =
        match a,b with
            | 'A','A' | 'T','T'     -> {h= -7.9<Cal/M>; s= -22.2<Cal/M/K> }
            | 'A','G' | 'C','T'     -> {h= -7.8<Cal/M>; s= -21.0<Cal/M/K>}
            | 'A','T'               -> {h= -7.2<Cal/M>; s= -20.4<Cal/M/K> }
            | 'A','C'| 'G','T'      -> {h= -8.4<Cal/M>; s= -22.4<Cal/M/K>}
            | 'G','A' | 'T','C'     -> {h= -8.2<Cal/M>; s= -22.2<Cal/M/K> }
            | 'G','G' | 'C','C'     -> {h= -8.0<Cal/M>; s= -19.9<Cal/M/K> }
            | 'G','C'               -> {h= -9.8<Cal/M>; s= -24.4<Cal/M/K>}
            | 'T','A'               -> {h= -7.2<Cal/M>; s= -21.3<Cal/M/K> }
            | 'T','G' | 'C','A'     -> {h= -8.5<Cal/M>; s= -22.7<Cal/M/K>}
            | 'C','G'               -> {h= -10.6<Cal/M>; s= -27.2<Cal/M/K>}
            | _ -> failwith "bad nucpair"
```

Inter unit conversion

```
// eq 16 table 2
let eq16Table2Mod (div:float<M>) (mon:float<M>) (Ca:float<M>) (Cb:float<M>) (dNTP:float<M>) (primer: char array) (N:int) =
    let mgConc = max 0.0<M> (div-dNTP) // NTPs bind divalent ions, so remove that from further consideration
    let l = logMolar mgConc
    let lm = logMolar mon
    let tm1000 = wikiTemp1000 Ca Cb primer N

    let modA  = 3.92E-5<1/K> * (0.843 - 0.352 * sqrt(mon / 1.0<M> (* UNITHACK *) ) * lm)
    let modD  = 1.42E-5<1/K> *(1.279-4.03E-3*lm-8.03E-3*lm*lm)
    let modG  = 8.31E-5<1/K> *(0.486-0.258*lm+5.25E-3*lm*lm*lm)

    // fGC is the fraction of residues that are G or C
    let fGC = gcN primer N

    let mgRecip = 1.0 / tm1000 + modA + b*l + fGC * (c+modD*l) + (1.0/(2.0 * float (N-1)) * (e+f*l+modG*l*l) )
    1.0/mgRecip |> K2C
```

```
let inline K2C (k:float<K>) = (k*1.0<C/K>) - 273.5<C>
let inline C2K (c:float<C>) = (c*1.0<K/C>) + 273.5<K>
let inline mM2M (c:float<mM>) = c*0.001<M/mM>
let inline uM2M (c:float<uM>) = c/1000000.0<uM/M>
let inline nM2M (c:float<nM>) = c/1000000000.0<nM/M>
```

Conclusions

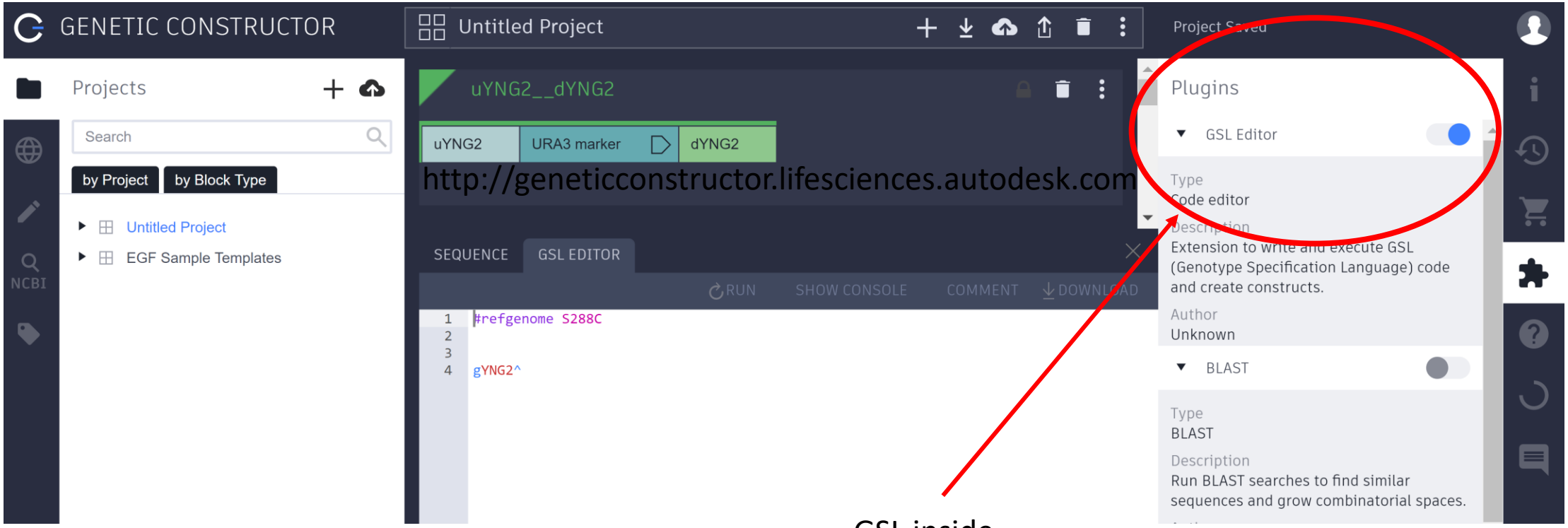
- Biologists are ready for software-like tool chains
- GSL is a viable path for improving Biologist productivity
- F# is a great language for compilers and DNA algorithms

More?

- https://github.com/AmyrisInc/GslCore
- https://github.com/AmyrisInc/Gslc
- https://github.com/Amyris/AmyrisBio
- http://geneticconstructor.lifesciences.autodesk.com

Genetic Constructor

http://geneticconstructor.lifesciences.autodesk.com



GSL inside

Can make introductions if anyone is interested

Extra Slides

Abstract: GSL: A compiler tool-chain whose target architecture is life itself

At Amyris, we are taking advantage of rapid advances in automation, biology and computing to engineer microorganisms into platforms that efficiently manufacture molecules for a wide range of industrial, consumer and pharmaceutical applications. We are currently working on hundreds of molecules and as the rate of engineering has increased, biologists are facing many of the same issues faced by early software pioneers. We have been introducing many software practices to the biology community at Amyris to increase efficiency and accuracy of genetic designs.

One of these innovations is the Genotype Specification Language (GSL) which was published and released as open source in 2016. GSL takes a high level genetic design and reduces it to DNA sequences that can "run" on a micro-organism of choice. The input language is modeled on the natural language Geneticists use, with extensions borrowed from software programming languages. I will describe the social and technical challenges we have faced with this revolution and explore the implementation of the GSL compiler and how F# and its excellent compiler tools are well suited to this type of application. The talk will assume minimal background in genetics or F# and should be accessible to any bio/functional curious types.

Xml Type Provider

- Move quickly from example of XML to data structures
- http://fsharp.github.io/FSharp.Data/library/XmlProvider.html

```
module rycodExample
open FSharp.Data

/// Example XMLto train FSharp type provider.  NB: not a valid rycod example, has been simplified for brevity
type ThumperRycod = FSharp.Data.XmlProvider<"""<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<ryseComponentRequest xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://thumper.amyris.local" xsi:schemaLocation="http://thumper.amyri
<rabitSpec name="gYNG2$C227C.hb" breed="X" upstreamLink="0" downstreamLink="2" direction="FWD" id="R.31277" creator="platt">
<dnaElementSpec speciesVariant="CENPK2">
<upstreamPrimerSpec>
<tail>gacggcacggccacgcgtttaaaccgcc</tail>
<body>ggacaacataaccaatagaagatg</body>
</upstreamPrimerSpec>
<downstreamPrimerSpec>
<tail>aggtccgccggcgttggacgagcg</tail>
<body>ggaatcatatcttagtttgattcacaag</body>
</downstreamPrimerSpec>
<dnaSequence>
GGACAACATAACCAATAGAAGATGGATCCAAGTTTAGTTTTAGAGCAAACGATACAAGATGTGTCCAACCTCCCATCAGAATTTCGTTACCTCTTAGAGGAGATCGGTTCAAATGATTTGAAGCTCATCGAAGAAAAAAAGAAATACGAGCA
</dnaSequence>
<quickChangeSpec>
```

Xml Type Provider: Consuming web services

Type provider generates (provides!) all the types associated with the XML definition.
Here we fetch and cache definitions of parts from a web service for DNA parts using the RYCOD XML protocol in the previous example. Actual retrieve and parse can be written in a single line.

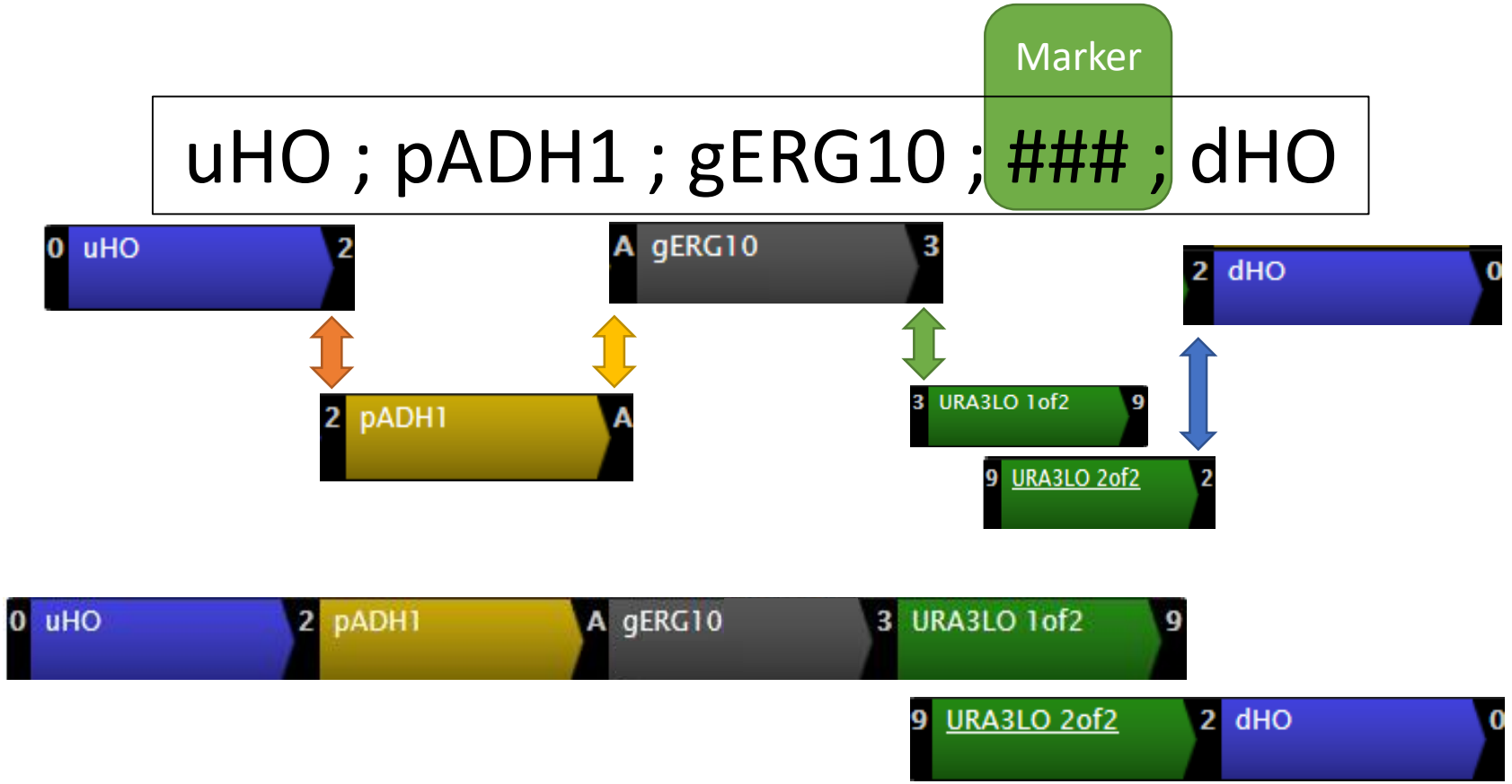
```
/// Hutch interaction - fetch part defs from RYCOd service, cache them
let fetch (url:string) =
    match fetchCache.TryGetValue(url) with
        | true,x -> Some x
        | false,_ ->
            let v=
                try
                    use wc = new System.Net.WebClient()
                    let s = wc.DownloadString(url)
                    let res = rycodExample.ThumperRycod.Parse(s)
                    res
                with _ as ex ->
                    printf "ERROR: from thumper %s\nERROR: Might be rabit id that does not exist.\nERROR: %s\n" url ex.Message
                    exit 1
            fetchCache.Add(url,v)
            Some v
```

Xml Type Provider: Generating output

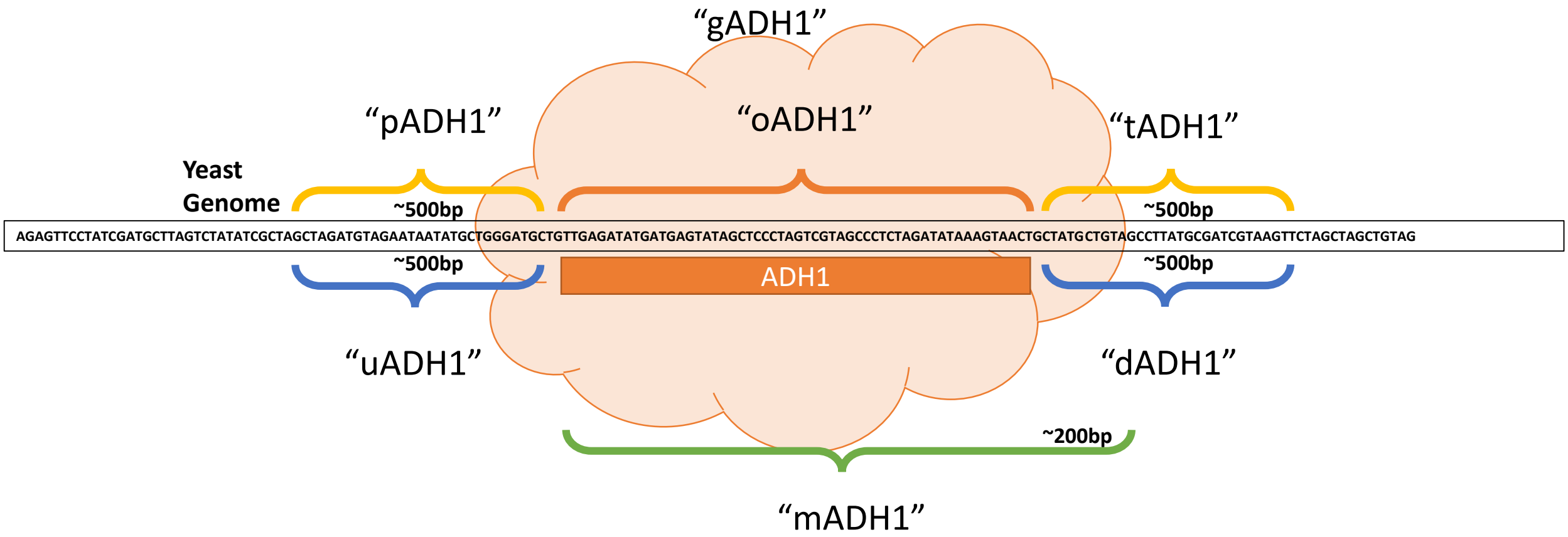
Type provider also creates all the data structures to make construction of XML content simple

```
let stitchSpecs = seq {
                  for myS in  seq { for m in ms do
                                       yield m.a
                                       if m.b.rabits.Length > 0 then
                                           yield m.b }
                                    |> Seq.filter (fun s -> keepStitch.Contains(s.sId))
                              do
                                  yield new rycodExample.ThumperRycod.StitchSpec(
                                                   sprintf "S%d" myS.sId,
                                                   user,
                                                   [| for myR in myS.rabits ->
                                                           new rycodExample.ThumperRycod.RabitRef(
                                                                          sprintf "R%d" (rabitMap.[myR.signature()])
                                                           )
                                                   |]

                                        )
                } |> Array.ofSeq

let rcr = new rycodExample.ThumperRycod.RyseComponentRequest(schemaLocation,rabitSpecs,stitchSpecs,megastitchSpecs)

use x = if proj = "-" then
                    new XmlTextWriter(stdout)
                else
                    new XmlTextWriter(new StreamWriter(proj + ".xml"))

x.Formatting<-Formatting.Indented
rcr.XElement.WriteTo(x)
```

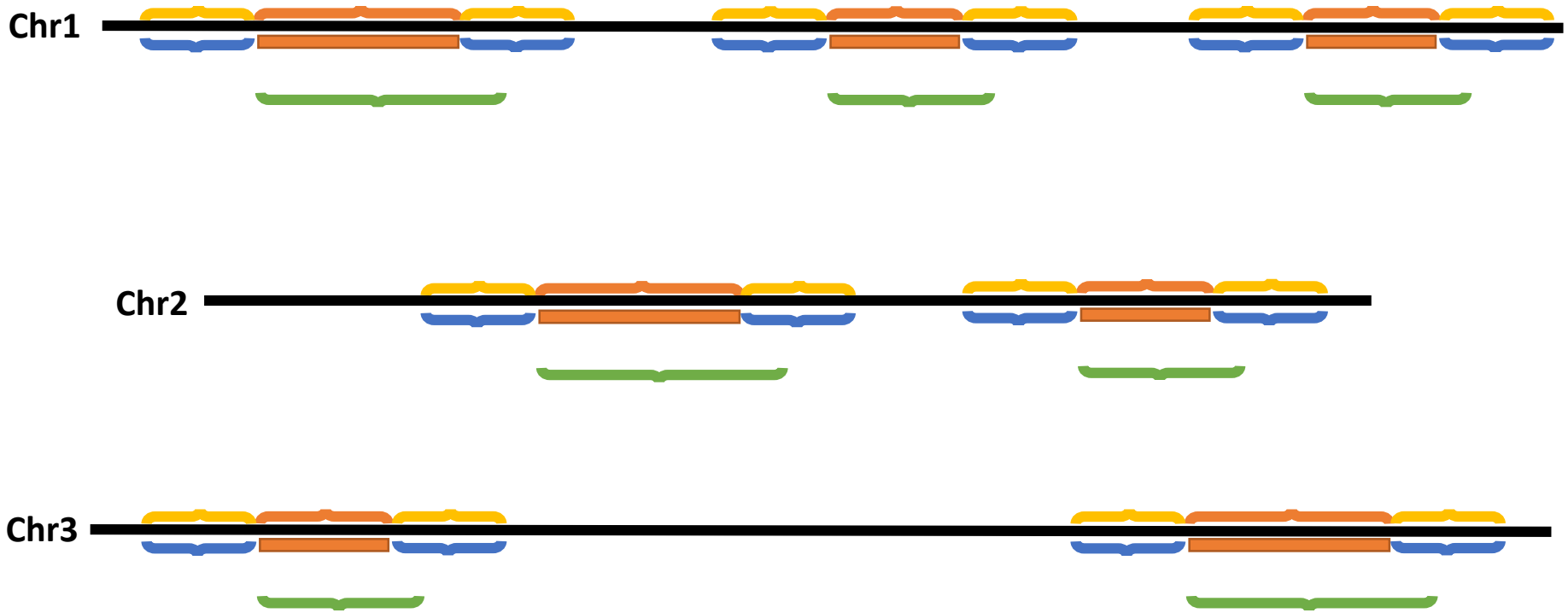
Basic arrangement of parts in GSL



GSL is oriented around genes and gene parts



GSL is oriented around genes



Code from Interview Candidate

Type safety can be costly and it doesn't always help….

```
]char *stripString(const char *input)
 {
    char *intermediate;
    char *resultant;
    int  inputLength,outputLength;
    char singlet;
    inputLength = strlen(input);
    for(int i =0;i<inputLength;i++)  {
        singlet = input[i];
        if(!illegalChar( singlet ))
            intermediate = intermediate + singlet;
        else
            intermediate = intermediate + '_';
    }
    resultant = new char(inputLength);
    strcpy(resultant,intermediate);
    outputLength = strlen(resultant);
    return resultant;
 }
```