Introduction to Translators & Domain Specific Languages

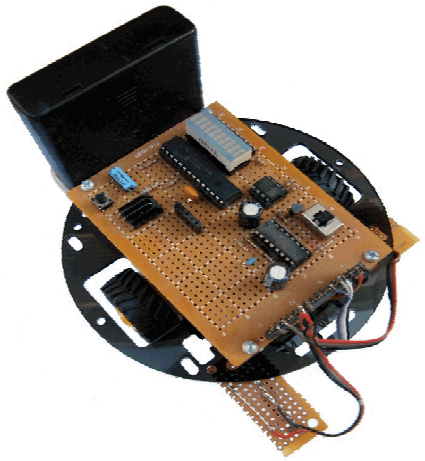
Ralph Iden, Principal Developer
Follett Software Company

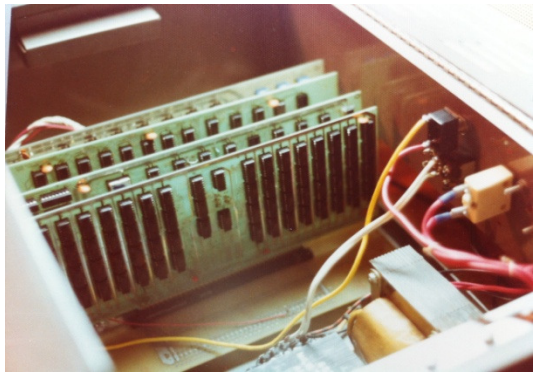
Introduction and Objectives

- Who am I?
- What do I know and when did I learn it?
- Three parts with time for Q & A
- What we'll cover and what we'll be glossing over

Part 1: What's this all about?

- How does a program get from here to there?





Computer Languages

- Why do we have them?
- Why are there so many?
- “I like feature x of a language, but other parts of the language are for junk.”
- Isn’t there a way to easily have my customer write her/his own scripts or rules?

Enter Domain Specific Languages (DSL)

Domain-specific language (noun): a computer programming language of limited expressiveness focused on a particular domain.

- Martin Fowler's DSL book: http://martinfowler.com/dsl.html
- DSLs abundant and found everywhere

Domain Specific Languages

- Developed to make it express solutions simpler
  - Keeps program focused on the specifics of the domain and not the underpinnings of a general purpose language
  - Can implement a full set of functionality, but generally a very small, targeted set of statements.
- FORTRAN arguably an early DSL
- make, ant, even regular expressions are also considered DSLs
- Often they are declarative focused (over procedural)
- Natural languages are in a separate category and generally not considered a DSL per se. DSLs targeted for customers can tend to have a natural language bent, but are not truly natural.
- DSLs can be categorized as internal or external (Fowler)

DSL Examples - RSpec

Rspec

```
# bowling_spec.rb
require 'bowling'

describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

DSL Examples - CoffeeScript

CoffeeScript

```
mood = greatlyImproved if singing

if happy and knowsIt
   clapsHands()
   chaChaCha()
else
   showIt()

date = if friday then sue else jill
```

DSL Examples - ha!

Solar Controller

```
sensor 1 is 1820b
sensor 2 is SHT11
sensor 3 is SL322
sensor 4 is US1

zones = 3

while 1 == 1
  // soil
  if SENSOR 3 < .33 then activate relay1 until sensor3 > .8

  at dusk activate relay2

  at dawn deactivate relay2

  if night and sensor 4 > 5 then
    for i = 1 to zones
      activate zone i for 120 seconds
    next
    log "Motion detected"
  end

  if panic then CallForHelp end

end
```

DSL Examples - DBPopulate

```
person_type:? super-user: false
+ [hourly] id: ?, desc: "Employee-Hourly", added: "now(-7,BOD)“
+ [salaried] id: ? desc: "Employee-Salaried" added: "now(-3,EOD)“

# Additional person_type row added after the first group
person_type:
+ [admin] id: ?, desc: "Administrator", super-user: true, added: now()

person:
+ id: 4, person_type_id: person_type(admin:id) last_name: "Admin"
```

The Basics

- Three stages of language processing
  - Lexical analysis (lexing, scanning or tokenizing) of the input stream
  - Parsing or "recognizing" the language grammar
  - Back end processing
    - *Program execution (interpreter)*
    - *Code generation (compiler)*
    - *Translation or processing of data*

Part 2: Tokens, Lexers, Scanners, oh my

- First step in process is to tokenize the input stream (the program)
- The tokenizer scans the input stream looking for symbols or tokens
- Token types fall into a number of categories
  - Keyword
  - Constant
  - Identifier
  - Number
  - String
  - Operator
  - Separator
- Tokenizers often use regular expressions for input processing
  - regex is available in most languages in use
  - Simplifies tokenizing code, but can be difficult to create correct matching rules -- examples to follow

Example: Simple 4 function calc (the HelloWorld of the parser world)

Consider a simple 4 function calculator

NUMBER

OPERATOR + - * /

RESULT =

1 + 2 =	
	NUMBER 1
	OPERATOR +
	NUMBER 2
	OPERATOR =

Simplest thing that could work approach

- Write a few lines of code to tokenize the input
- Examine the token stream looking for a NUMBER OPERATOR NUMBER =
- Switch on the type of operator and apply the calculation
- We're good to go

Not quite there yet

- What about
  1 + 3 / 4 =
  or
  (1 + 3) / 4 =
  or
  -1 - 4 =
- More than a single calculation
- Operator precedence
- Negation
- Write a little more code and now we're done

What if your grammar is associative?

- Associatively defines behavior between 2 operators of equal precedence
  - LEFT-ASSOCIATIVE (working left to right)
  - RIGHT-ASSOCIATIVE (working right to left)
  - NON-ASSOCIATIVE (can work LR or RL)

A = 1 + 3

A = B = 1 + 3

5! (factorial)

!5 (error)

Other things to consider

- Error handling

  1 / 0 =

  Parameter type checking (if grammar supports types)

  Syntax and ordering issues (natural language this is not)

- Scopes and contexts

Is there a better way?

- It is apparent that things can get quite ugly in a hurry if starting from scratch
- Developing a system that defines the legal tokens as well as a set of syntax rules (productions, reductions, rules), allows the use of standardized tools that take care of much of the drudgery.

Demo
expression-evaluator

Part 3: Heading towards home

- expression-evaluator is a pretty useful tool
  - Expression caching
  - Variables
  - Multiple statements per parse call
  - Data sources
  - Extensible
- But it lacks things like conditionals and flow-control (loops, etc.)

Back-end processing: Productions and Rules

- Produces final output (asm src, bytecode, or whatever you want) from an abstract syntax tree (AST) or node list.
- Different back-ends can be dropped in for different results with the same grammar (interpreter versus compile, etc.)
- Optimizations usually occur here
  - Optimize for execution time
  - Optimize for code size
  - peephole optimization to simplify the code generated
  - constant folding (evaluating static expressions at compile time)
  - short-circuit evaluation

Demo
GOLDBuilder and GOLDengine

Resources - Books

- "Engineering a Compiler" by Cooper and Torczon (Theory)
- "Compilers: Principals, Techniques, and Tools" aka "The Dragon Book" by Aho, Lam, Sethi, and Ullman (Theory)
- "Domain Specific Languages" by Martin Fowler http://martinfowler.com/books/dsl.html (Theory)
- "How to Create Your Own Freaking Awesome Programming Language" by Marc-Andre Cournoyer (eBook, code, and screencast) (Ruby/Java)
- Free: "Let's Build a Compiler" by Jack Crenshaw http://compilers.iecc.com/crenshaw (Pascal)

Resources - Software demonstrated

- expression-evaluator by Ralph Iden https://github.com/ridencww/expression-evaluator (Java)
- GOLDBuilder/GOLDParser by Devin Cook http://goldparser.org (.NET, Java, others)
- GOLDEngine by Ralph Iden's implementation of GOLD runtime https://github.com/ridencww/goldengine (Java)

Resources - Other

- Lexers
  - Lex (co-written by Google's Eric Schmidt)
  - Flex
  - JFlex (Java)
  - Rex (Ruby)
- Parser builders
  - YACC
  - Bison
  - Ragel
  - JavaCC (Java)
  - RAcc (Ruby)
  - Ply (Python)
- Lexer/Parser builder combinations
  - ANTLR
  - GOLDParser
  - TreeTop