DO FIFTY-TWO

Sinclair Target
Yichi Liu
Yunfei Wang
Jayson Ng
Josephine Tirtanata

Motivation

Situation:

- Card game languages are popular
- But the available languages are C-like syntax
- So who were those languages for?

Goal:

- A card game language for beginner programmers
- Friendly syntax influenced by Python
- Conceptually simple

Overview of the Language

What is Do Fifty-Two?

- An imperative, (mostly) procedural, statically and strongly typed language
- An ***IMPERATIVE***, (mostly) procedural, statically and strongly typed language
- Not functions. Procedures!
- No return values!
- Pass by reference
- Procedures are not first-class
- “A program does things to data.”

Tutorial

Operator	Meaning
+ - * /	Standard arithmetic operators. Integer arithmetic only. Standard precedence.
= != < > >= <=	Standard relational operators, except that the equivalence operator is = not ==. Standard precedence.
| & !	Logical operators and the unary NOT operator. No bitwise operators.
:	Assignment operator
t> b> <t <b	Prepend and append operators. Take a **Set** (see below)and a **Card** and either adds the **Card** to the front or the back of the **Set**. If **Card** is null, does nothing.

tutorial

Operator	Meaning
.	Dot operator, Accesses field within an variable
+	String concatenation operator

tutorial

Primitive Type	Meaning
Number	Integer
String	String
Boolean	Boolean

Tutorial

Composite Type	Meaning
Card	A data type representing a card. **Fields**: ***Number*** rank ***Number*** suit ***String*** desc
Set	A data type representing an ordered collection of cards, can be a deck, hand, or something else. **Field**s: ***Number*** size ***String*** desc ***Card*** top ***Card*** bottom
Player	A data type representing a player, or possibly a dealer. **Fields**: ***Set*** hand ***String*** desc

Tutorial

Action	Function
Declare a new variable	**new *typeName variableName*: *value*** i.e. new *Number num: 0*
Add new fields to existing variable	***typeName* has *fieldType* called *fieldName*** i.e. *Player* has *Number* called *score* Now Player has field "*score*"
Redefine default environmental variables	**configure *variableName*: *value*** i.e. configure *numberOfPlayers*: 2

Tutorial

Action	Statement
Declare a function	***procedureName* with *Type parameterName:*** i.e. do *Sum* with *Number n1* and *Number n2:* *<function body>*
Call a function	**do *procedureName* with *argumentName*** i.e. do *Sum* with *5* and *6*
Quit a loop/conditional statement	**do quit**

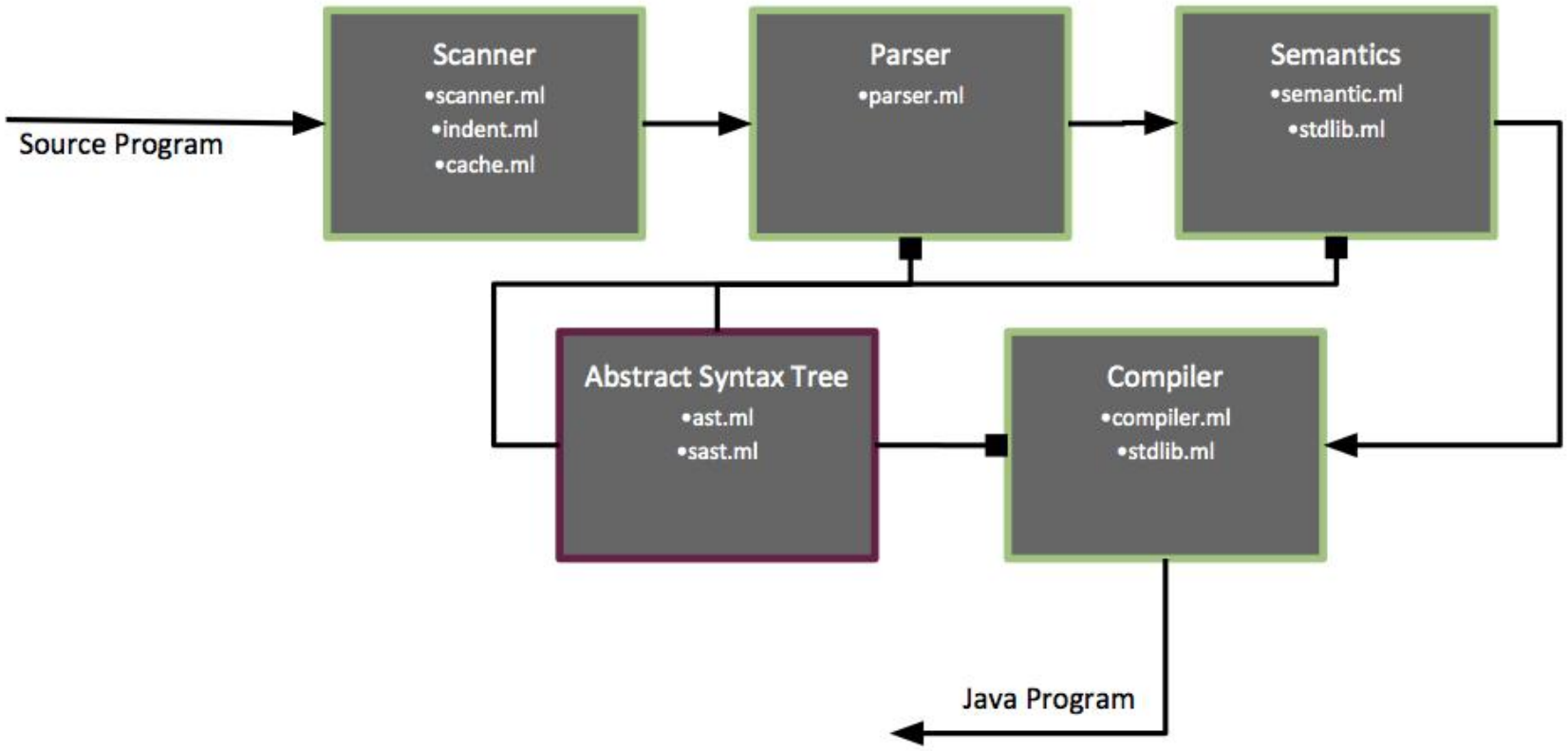
```
//A Sample Program

 configure numberOfPlayers: 2
 configure highestCard: ace
 configure ascendingOrder: true

 Player has Number called x // declare new field “x” for
  Player

 new Number warCount: 0 // declare new variable “warCount”

 setup:
        // deal cards
        do turn with player1
        do output with "The score of player1 before: " +
  player1.x
        do output with "The score of player1 after: " +
  player1.x
        do output with "warCount: " + x

 round:
        do output with "number of players: " +
  numberOfPlayers
        do quit

 turn with Player player:
        if p.x = 0:
            p.x : p.x + 1
            do quit
```

Behind-the-scenes

Implementation and Testing

Implementation

Compiler Work Flow



Implementation

Indentation-based Blocks

```
do output with "Hello"

hello:
    do output with "Hello"
```

(vim) set: list

```
do output with "Hello"$
$
hello:$
^Ido output with "Hello"$
```

Implementation

Indentation-based Blocks

Is this even Context-Free?

```
hello with Number n
    if n > 5:
        if num > 10:
            do output with “Hello!”
        else:
            do output with “Bye!”
```

```
hello with Number n$
^Iif n > 5:$
^I^Iif num > 10:$
^I^I^Ido output with “Hello!”$
^I^Ielse:$
^I^I^Ido output with “Bye!”$
```

Implementation

Indentation-based Blocks

scanner.ml:

```
let cur_depth = ref 0
```

If depth > cur_depth, <INDENT>
else if depth < cur_depth, <DEDENT>
else <NEWLINE>

```
hello:
<INDENT>do output with "Hello"
<DEDENT>
```

Just as if we had used braces!

```
hello:
<LBRACE>do output with "Hello"
<RBRACE>
```

Implementation

Indentation-based Blocks

But what about:

```
hello with Number n:
<INDENT>if n > 2:
    <INDENT>if n > 4:
<DEDENT??>
```

Does not parse. Second “if” never closed!

Need to spit out *multiple* <DEDENT> tokens for a single regex in scanner.ml. Can we even do that in Ocamllex?

Implementation

Indentation-based Blocks

No. Parser takes one token at a time—we can't just pass it several at once.

<DEDENT_MULT(n)>

```
if cache is not empty
        return front of cache
else get token
        if token is <DEDENT_MULT(n)>
                cache n <DEDENT> tokens
                return front of cache
        else
                return token
```

Implementation

A Small Standard Library

What happens when someone uses one of our “standard” procedures?

```
do output with a
```

```
UndeclaredID(“The procedure output has not been declared.”)
```

It might be implemented in our runtime, but the semantic analyzer doesn’t know about it.

Implementation

A Small Standard Library

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

int main … etc.
```

Standard library inserted at the top of the file. All function and variable declarations added to environment before the rest of the program goes through the semantic analyzer.

How to do this without implementing a preprocessor?

Implementation

A Small Standard Library

Another thing to consider:

Do:	Java:
`card.suit`	`-> card.suit`
`deck.size`	`-> deck.size()`

How do we address what could be lots and lots of special cases?

Implementation

A Small Standard Library

stdlib.ml

```
vars = [ (var_decl, java) … ]
configs = [ config_decl … ]
fields = [ (field_decl, java) … ]
funcs = [ func_decl … ]
```

semantic.ml

starting scope includes vars
starting environment includes
configs, fields, funcs

compile.ml

```
let java_of_var env var =
```

if var is in stdlib, use the java there
else use the var_id

Implementation

Translating to Java

- Issues
  - abstraction: designing a Player/Game/Card/Set/Deck class
  - What should be default classes?
  - What should the compiler generate?
- Making it user friendly
  - Defining Ace, Jack, King, Queen, Spade, Heart, Club, and Diamond
  - Card values
  - Card related functions- shuffle and draw
  - What would be the best data structure?
    - Deque is most suitable
    - Implemented by ArrayList

Implementation

Translating to Java

- Implementing a seamless and error prone output and input function
  - recognizing data type of variable to match with the correct input functions (Boolean/String/Number)
  - error handling
- Simplifying logical operators
  - The ability to compare various data types such as String, Card, and Set
  - `if player1_hand_top > player2_hand_top`
- A lot of these issues were solved after the semantic analysis of the compiler, attaching the data types to variables

Implementation

Translating to Java

Card A card contains a face number, a value, a suit and a suit value.	Set A set is an ordered group of cards. Set functions are fixed: shuffle, append, prepend, select, and peek.	Player A player has a hand.	*Game* The game class lists out the functions that are defined by the user. The setup() and round() methods must be defined.	Main The main function creates an instance of a game and invoke setup(). After setup completes, it calls round() in an infinite loop.
	Deck A deck extends a Set and creates a deck of French Playing cards (Standard 52 cards with 4 suits)	*MyPlayer* My player extends player and has additional instance variables that are defined by the source program	Utility The utility class aids the game class by supplying functions for logical operations and I/O	

Implementation

Semantic Checking

1. ENV contains
    - symbol table
    - Configuration Variables
    - Function Declarations
    - Function Calls
    - Variable Declarations
    - Field Declarations
    - break/continue

Implementation

Semantic Checking

2. Check Unknown Data Types

3. Check Undeclared IDs

4. Check Mismatched Types

5. Check Wrong Types

6. Check if any function/variable/field is redeclared

7. check if a program has no setup

8. check if a program has no “round” procedure

(corresponding to “main” in Java)

Testing

Performed on-going White box testing.

Test Plan:

3 Phase Testing:

- Parsing / Semantics
- Logic
- Game Play

Two automated test suites:

- testLogic.sh
- testParse.sh

Testing

Testing Script

```
Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                              s/.do//'`
    reffile=`echo $1 | sed 's/.do$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."
    javafile=`echo $basename |sed -e 's/^//g' -e 's/-/_/g'`
    ajavafile=`echo $javafile | perl -pe 's/\S+/\u$&/g'`
    newjavafile=`echo $ajavafile | perl -pe 's/([^ ])_([a-z])
/\\1\\u\\2/g'`
    echo 1>&2
    echo "###### Testing $basename" 1>&2
    generatedfiles="$generatedfiles tests/${newjavafile}.java
tests/${basename}.diff tests/${basename}.out" &&
    Run "$DO_FIFTY_TWO" $1 &&
    Run "mv Game.java MyPlayer.java $RUNTIME" &&
    Run "cd runtime/" &&
    CompileRunTime &&
    Run "java -cp . $MAIN >" ../tests/${basename}.out &&
    Run "make clean" &&
    Run "cd .." &&
    Compare tests/${basename}.out tests/${basename}.gold
tests/${basename}.diff
    # Report the status and clean up the generated files
}
```

Testing

Testing Script

```
CheckFail() {
     error=0
     basename=`echo $1 | sed 's/.*\\///
                               s/.do//'`
     reffile=`echo $1 | sed 's/.do$//'`
     basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."
     javafile=`echo $basename |sed -e 's/^//g' -e 's/-
 /_/g'`
     ajavafile=`echo $javafile | perl -pe 's/\S+/\u$&/g'`
     newjavafile=`echo $ajavafile | perl -pe 's/([^ ])_
 ([a-z])/\\1\\u\\2/g'`
     echo 1>&2
     echo "###### Testing $basename" 1>&2
     generatedfiles="$generatedfiles
 tests/test_failure/${newjavafile}.java
 tests/test_failure/${basename}.diff
 tests/test_failure/${basename}.out" &&
     RunFail "$DO_FIFTY_TWO" $1 ">"
 tests/test_failure/${basename}.out
     # Report the status and clean up the generated files
 }
```

Testing

Manual gameplay testing:

1. Created simple game called War where 2 players are given half a deck each.
2. Each player then plays the top card of their deck who ever has the lower of the two picks up the played cards
3. Winner is decided based on who has no more cards in their deck

summary

Summary of the language and lessons learned

summary

- Goal: to create a card game language that is user-friendly to average non-programmers
- We successfully provided an English language style syntax that are easy for novice programmers to learn
- Out compiler for *do fifty-two* parses a user's program, builds a AST, and semantically checks the AST with a Semantic Analyzed AST
- Our compiler is able to generate a running java program
- Manage to get all team members to contribute in the implementation of the compiler using multiple tools and languages throughout the semester
- Highlight: compiling hello_world.do, and eventually, war.do

Lessons Learned

"Split your project up into discrete parts with the smallest number of interdependencies possible. Then give each part to someone and make them own it. That way nobody has to build a whole compiler."

Sinclair Target

"Always have the big picture of your language design in mind. Keep modifying as you move along and discover dark corners of your original design" *Yichi Liu*

"Understand the concept of a compiler well. Grasp why each stage is required. Make sure you know OCAML well. And use tools such as Slack and Trello to help with communication and task management." *Josephine Tirtanata*

"I used CLIC machines while the rest of the team members used Mac OS X, and it gave me some compatibility problems when putting together all the codes to compile, so I think having integrated IDE environment is important."

Yunfei Wang

"Communication and task management was incredibly important to us.
The use of trello to monitor bugs and roque features greatly helped our efficiency.
Making tests with the intention of failure was important since it was an outlier case that wasn't immediately apparent." *Jayson Ng*

credits