School of Computing

CP SC 828: Theory of Programming Languages

Spring 2011

Instructor: Murali Sitaraman
Contact Information: McAdams 210; Phone: 656-6738; E-mail: murali@cs.clemson.edu
Web page: Listed under www.cs.clemson.edu/~murali
Lecture Hours: TTh 12:30PM – 1:45PM, Daniel 303
Office Hours: M: 1:00PM – 2:00PM; TW: 2:00PM – 3:00PM
other hours by appointment.
Prerequisite: CP SC 628 or equivalent

All notes have been posted. Blackboard has up to date grades.

Reading Assignment: Tony Hoare, *The verifying compiler: A grand challenge for computing research*, Journal of the ACM (JACM), 50(1), January 2003.

Homework Assignment #1 (due February 1)

1. What does a "verifying compiler" do according to Hoare and Misra's paper above? Explain any one point in the paper that you found to be especially interesting. Explain any one point that you found to be especially confusing.

2. Suppose that the alphabet set is {0, 1}. Draw an FSA to recognize only even numbers without any leading zeros.

3. Write a regular expression to generate a valid identifier in a programming language of your choice.

4. Can you draw an FSA to recognize prime numbers? Why or why not?

5. Develop a CFG to generate the language in question #2.

6. Write a CFG for necessary to generate a sequence of variable declarations of Boolean and Integer variables, followed by a sequence of one or more swap statements of the form <var_name> :=: <var_name>. The declarations and sequences should be terminated by semicolons. The language should also allow one or more curly braces at the beginning and require a matching set at the end. You can use the non-terminal <var_name> without expanding it. Draw a parse tree for an example sentence in your language. Give an example of a sentence not in your language.

7. Develop a "mini compiler" for the language in question #6, based on the principles discussed in the class. You may write it in any language. You may also use lexer/parser generator tools, such as flex/bison or antlr. (Due: Feb. 3)

Homework Assignment #2 (due February 8)

1. In your favorite object-based language, using an example interface, classes that implement that interface, and objects based on the interface and classes, give an example piece of code (i) that has no type errors; (ii) that has compile-time type errors; (iii) has run-time type-related errors. Which of these errors can be caught using attribute grammars? [You don't need to write interfaces or classes; you just need names.]

2. In the following grammar, Op_Name and Var_Name are just usual identifiers, and Param_Num is a number. Without changing the grammar, add suitable attributes, evaluation rules, and conditions to the following grammar so that the number of parameters to an operation is restricted to be at least 1 and the number of arguments in a call to that operation matches that number.

   S → <Op_Decls>< Calls>

   <Op_Decls> → <Op_Decl>; | <Op_Decl> ; <Op_Decls>

   <Op_Decl> → <Op_Name> <Param_Num>

   <Calls> → <Call>; | <Call> ; <Calls>

   <Call> → <Op_Name> (<Arguments>)

   <Arguments> → <Var_Name> | <Var_Name> , <Arguments>

4. Attribute grammars can be used to resolve ambiguities. For example, suppose that the Boolean expression rule is given as below.

<Bexp> → "True" | "False" | <Bvar> | <Bexp> "and" <Bexp> | <Bexp> "or" <Bexp>

   (i) Show that the above grammar is ambiguous using an example sentence.

   (ii) Without changing the grammar, insert suitable conditions (and new attributes as necessary to your non-terminals) so that the Boolean expressions are evaluated, giving precedence to "and" over "or".

5. Write an Attribute Translation Grammar (ATG) to evaluate the decimal value of binary strings given the following grammar:

   <bits> ::= <bit> | <bit><bits>; <bit> ::= 0 | 1.

Homework Assignment #3 (due February 17)

1. In a paragraph or two, discuss the key problems with informal reasoning using an example or otherwise.

2. Consider the following uses of the monogeneric predicate, and explain which properties are violated (if any) in each case.

    a. Is_Monogeneric_for({1, 2, 3}, 1, cyclic_f) where cyclic_f is a function that maps 1 to 2, 2 to 3, and 3 to 1.

    b. Is_Monogeneric_for({1, 2, 3}, 1, almost_cyclic_f) where almost_cyclic_f is a function that maps 1 to 2, 2 to 3, and 3 to 3.

    c. Is_Monogeneric_for(N, 1, positive_suc) where positive_suc is the successor function that gives the next Natural number, except that suc(0) is defined to be 0. N is the natural number set, including 0.

    d. Is_Monogeneric_for(N, 0, even_suc) where even_suc is the successor function that gives the next even Natural number.

3. If the third property of monogenerics is left out, would the definition of "+" apply to all Natural numbers? Explain your answer.

4. Write formal definitions of 1, 2, and 3, and prove 1 + 3 = 2 + 2 formally. Show all steps along with the justification for each step.

5. Write a formal inductive definition of the factorial function on natural numbers.

6. Show that natural numbers and the set of even natural numbers are isomorphic.

7. Prove formally Theorem N18 (that states that multiplication is commutative) in Basic_Natural_Number_Theory. Show *all* steps along the same lines done for the proof of Theorem N1 in the class. You may use any of the theorems given that precede N18, such as + is associative, commutative, etc. [You may not be able to answer this question until after the discussion in the class on Feb 15.]

Homework Assignment #4 (due March 8)

1. Explain and distinguish the notions of validity and correctness. Write an example of valid assertive code involving a loop and an invalid code involving a loop in your favorite language.

2. For each of the examples given below, state whether the assertive code is valid or invalid. Assume suitable context for all questions in this assignment. Use Venn diagrams to explain your answers to at least three questions. Prove the correctness of any one valid piece of code. Show all steps.

```
Assume true;
Confirm I >= 0;

Assume I > 0;
Assume I /= 0;
    I :=: J;
Confirm I = 0 or I /= 0;

Assume true;
    I :=: J;
Confirm I = J;

Assume I = K and K > 0;
    I :=: J;
Confirm J > 0 and K > 0;
```

3. Prove the correctness of any one example above. Show all your steps.

4. Explain the terms soundness, completeness, and relative completeness in your own words.

5. Write a proof rule for swap statement that is unsound. Show it is unsound. Write a rule for swap statement that is incomplete. Show it is incomplete. Your rules and examples need to be different from the ones discussed in the class.

Homework Assignment #5 (due March 17)

6. Are the following rules sound? If not, are they relatively complete? Justify your answers with suitable examples. No proofs are necessary.

   a. The swap statement rule below, where Q [X <~ Y][Y <~ X] means that X's are replaced followed by Y's:

      Context/assertive_code; **Confirm** Q[X <~ Y][Y <~ X];
      ____
      Context/assertive_code; X :=: Y; **Confirm** Q;

   b. The if-then statement rule below:

      Context/assertive_code; code; **Confirm** Q;
      Context/assertive_code; **Confirm not** B **implies** Q;
      ____
      Context/assertive_code; if B then code; **Confirm** Q;

7. Prove the correctness of the following code. Show all steps.
   Context/ **Assume** I /= J; **If** I = J **then** I :=: J; **end**; **Confirm** I /= J;

8. Show a sound and relatively complete proof rule for the if-then-else statement. Use a simple example in any language to show what would go wrong with your rule, if the condition had side-effects. Does your rule become unsound or incomplete if there are side-effects?

9. Show a sound and relatively complete proof rule for the case statement of the form given below, where only one of the clauses is executed; otherwise clause is executed only if everything else fails. Show a flowchart that you'd use to develop this rule.

   Context/assertive_code;
   Case B1 **then** code_1;
   B2 **then** code_2; …
   Bn **then** code_n;
   otherwise code_0;
   end case; **Confirm** Q;

10. Go to the link RESOLVE Tutor under the heading Education at this site: http://www.cs.clemson.edu/group/resolve/

    a. Select Mathematics, select String_Theory, and then select Introduction. Answer the True/False exercises at the end of the practice questions. You need to only write True/False.

    b. Select Specifications and select Introduction under Specifications and Parameter Modes. Answer the multiple choice questions at the end.

Homework Assignment #6 (due April 5)

11. Reduce the following assertive code by applying function assignment rule. Show all steps. What additional assumptions are necessary (at the beginning) for the assertive code to be provable?

Context/**Assume** X = X0 **and** Y = Y0;
X := Sum(X, Y); Y:= Difference(X, Y); X := Difference(X, Y);
Confirm X = Y0 **and** Y = X0;

where Context includes
Operation Sum (**restores** I, J: Integer): Integer;
requires min_int <= I + J <= max_int;
ensures Sum = (I + J);

Operation Difference (**restores** I, J: Integer): Integer;
requires min_int <= I - J <= max_int;
ensures Difference = (I - J);

12. Write an operation call rule and a procedure declaration rule suitable for an operation with the following specification:

Operation P(**alters** x: T1; **clears** y: T2; **replaces** z: T3);
requires pre_P/_x, y_\;
ensures z = post_P/_#x, #y_\;

13. Consider the following specification and code. You need to verify this code twice: (i) Using Pop specified with an explicit ensures clause and applying the simple operation call rule and (ii) using Pop with the implicit ensures clause and applying the general operation call rule. Your answer should begin with the application of the procedure declaration rule.

Operation Clear_2(**clears** S: Stack);
requires (|S| = 2);
Procedure
Var Temp: Entry;
Pop(Temp, S);
Pop(Temp, S);
end Clear_2;

14. What do you understand by modular verification? Using the examples here or others, give an example of a proof rule that is not modular.

Verification Mini-Project Overview (due April 26)

This is a one or two-person group project involving an existing software-verification or theorem-proving tool, a paper describing and evaluating it, and a presentation of the tool to the class. By next week, you need to decide about which tools you might be most interested in evaluating and which students you would like to be your partners (no guarantees, however). Team and project consent will be given by the instructor shortly.

Project Details

- Decide whether to investigate a tool that claims to do **software verification** or one that claims to do **theorem proving**;
- Clear your choice with the instructor;
- Download and install the tool, on your computer or on a department computer; in some cases, tool download information may be hard to find!
- Figure out how to use it;
- For a software verifier, use it to verify one of the RSRG benchmark problems, or at least get close, and try to "break" it by running it on a small example program that would reveal unsoundness (if present); for a theorem prover, use it to prove (automatically, if possible) a few non-trivial VCs from the RSRG benchmark problems or similar code;
- Report to the class by spending in about 20 minutes, explaining the tool in detail -- how it works -- and by giving a live demonstration;
- Submit a brief (2-3 page) description and critical evaluation of it -- with some serious technical content to get full credit one week before the final exams.

Possible choices

- HOL4
- Holfoot
- JML4- FSPV
- KeY
- ProofPower
- VCC
- VeriFast
- A comparable tool about which we know little (with instructor permission)

Homework Assignment #7 (due April 21)

15. Write a recursive procedure for the following operation. Prove its total correctness.

    Enhancement Flipping_Capability **for** Queue_Template;
        Operation Flip (**updates** Q: Queue);
            ensures Q = Reverse(#Q);
    end Flipping_Capability;

16. Refer to the web demo interface at www.cs.clemson.edu/group/resolve for this question. Go to the Help menu, click on Tutorials, and then select "How to Create an Enhancement for a Concept." Follow the instructions and create a Flipping_Capability enhancement and realization, using your answer from the previous question. (In doing this assignment, do NOT cut and paste the enhancement from this PDF file! Type it in.) Generate VCs. Write a proof for each VC. Compare your answer for the previous question with the VCs automatically generated, and report on the differences you see.

17. Explain the difference between partial and total correctness. On the web interface, make the following changes to your code. In each case make only the one change and answer, if the code is valid; Generate VCs and check if it is provably correct. Explain the answers in each case. (a): Change the decreasing clause to be |Q| + 1. (b) Change the decreasing clause to be Max_Length - |Q|. (c) Remove the if statement in your code and simply write the statements in the body of the if statement. (d) Remove the call to Dequeue.

18. Prove the correctness of Iterative_Realiz of Append_Capability of Queue_Template given at the web interface by applying the while loop rule. Generate VCs, but don't need to show their proofs. Compare your answer with the VCs automatically generated, and report on the differences you see.

19. Write an implementation of a Queue Rotate operation and annotate your code with suitable maintaining and decreasing clauses. No proofs are necessary.

    Operation Rotate (updates Q: Queue; evaluates n: Integer);

        Requires 0 <= n <= |Q|;

        Ensures Q = Prt_Btwn(n, |#Q|, #Q) o Prt_Btwn(0, n, #Q);

20. Develop a partial correctness proof rule for the classical "repeat…until" statement.

Materials

Lecture notes, hand-outs, and web information will be used to cover the topics. It is not necessary to buy any book. References will be posted at the course web site.

Course Description and Content

The course will introduce you to formal syntax and semantics of languages. Topics covered will include operational and denotational semantics, and proof systems for verification of program correctness. The emphasis will be on imperative and object-based language features, specification language features, and generic data abstractions. The course will also cover topics such as type checking and attribute grammars that form the basis for language implementation tools. There may be a few programming assignments targeted to reinforce specific concepts. Students will also be introduced to and experiment with actual verification systems.

Grading Policy

Performance in this course will be evaluated by exams, homework assignments, a verification project, and active class participation. Some assignments may require programming. There will be two mid-term exams and a final exam. Requests for makeup exams are discouraged. NO MAKEUP EXAMS will be given without prior approval or valid medical emergency.

For the two mid-term exams, the following additional policy will be in effect. You may resubmit revised answers to questions where you lost points, within a prescribed deadline. You will earn a maximum of 33% of lost points, if your revised answers are correct.

Homework assignments are due at the beginning of the class when they are due. Only selected parts may be graded from homework assignments; the entire grade for the assignments will be based on those parts.

Details of mini verification project will be posted at the course website. It will entail learning about a verification system, writing a short paper summarizing it in the context of this course, and a short demonstration/presentation in the class. Unlike HW assignments, which are individual activities, two students may work together on a project.

Breakdown of points is given below:

HW assignments and mini project	35%
Exam #1 (Week #6)	20%
Exam #2 (Week #11)	20%
Final Exam	25%

Letter grades will be assigned as shown below:

90% - 100%	A
80% - 89%	B
70% - 79%	C
60% - 69%	D
< 60%	F

Attendance Policy

Attendance is not mandatory, but you are responsible for all materials covered in lectures.

Academic Integrity

All exams and homework assignments are individual tasks, unless specifically designated as group tasks. It is expected that you will work ALONE on exams and homework. Evidence to the contrary will be regarded as academic dishonesty and will be dealt with according to the University policie. For details, please see: http://gradspace.editme.com/AcademicGrievancePolicyandProcedures#integritypolicy

Learning and Feedback

I expect to foster a nurturing learning environment based upon communication and mutual respect. I will give serious consideration to any suggestion as to how to further such a positive and open environment. I encourage you to give feedback on various aspects of the course, including but not limited to content, exams and labs, style, and treatment. Please feel free to express your opinions during the classes or in the office hours. Your feedback is important for improving the quality of this course and that of graduate education in computer science, in general.

If you have a special need, and feel that you need assistance with regard to lectures, reading assignments, or testing, please advise me of your needs as soon as possible.

Tentative Course Outline

Given below is the list of topics I expect to cover in this course. The instructor might change the ordering of topics at his discretion.

Formal Syntax

1. Context-free grammars
2. Capturing context-sensitivity using attribute grammars
3. Syntax to semantics: Translational semantics, operational semantics
4. Compiler generation tools

Program Verification

5. Formal systems and proofs (e.g., number theory and string theory)
6. Proof systems for mechanical verification
7. Assertive languages
8. Example proof rules for (expression) assignment and swap statements
9. Data abstraction specification
10. Modular verification principles
11. Verification of control statements
12. Verification of loops; partial and total correctness proofs
13. Verification of procedure calls, including recursion
14. Verification of arrays and other built-in types
15. Data abstraction verification
16. Other topics, including references
17. Verification and theorem proving tools

Formal Semantics

18. Soundness and completeness of formal systems
19. Denotational semantics of programs
20. Relational semantics for assertive languages
21. Fixed point theory

Notes for CpSc 828

January 18, 2011

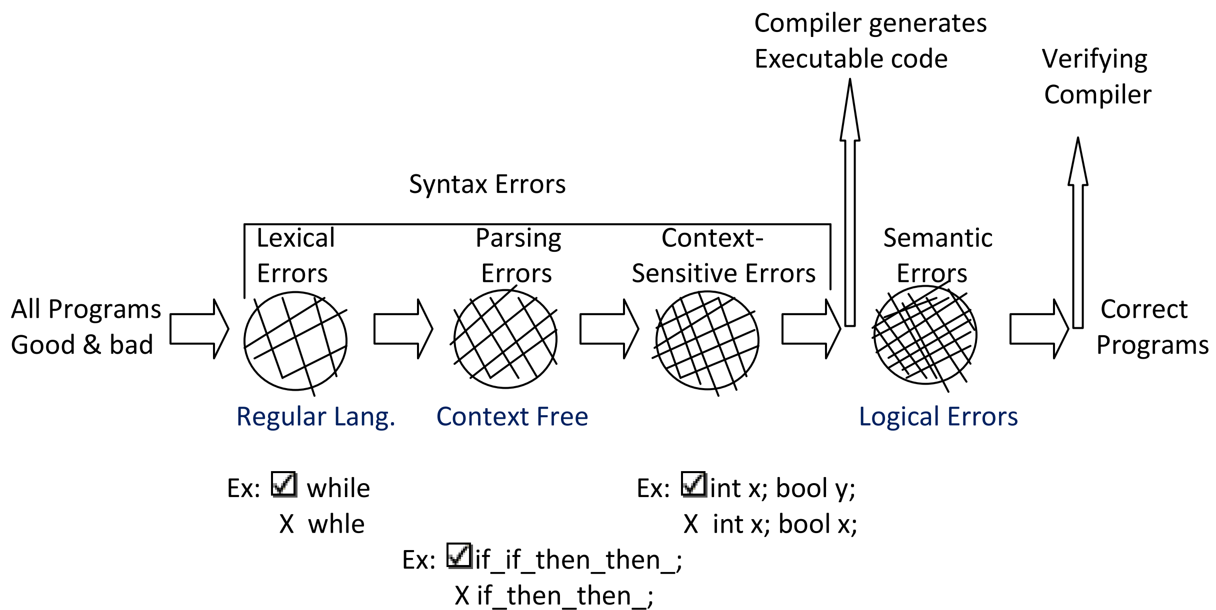
Code is **correct** if it meets specifications.

Compiler – generates executable code.

Verifier – Check is code meets specifications.

Verifying Compiler – Compiler + verifier.

Reading assignment: Tony Hoare: The Verifying Compiler: A Grand Challenge for Computing Research

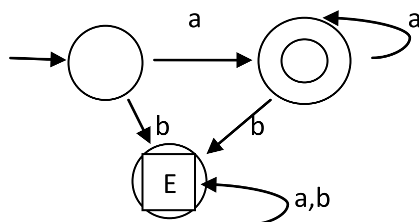


I. Lexical Analysis

Regular Languages & FSA's

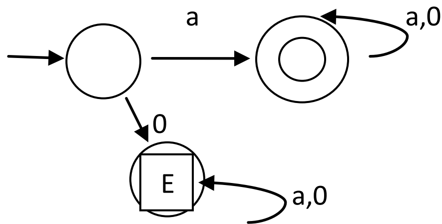
$\sum = \{a,b\}$
$L_a = \{a^n \mid n \geq 1\}$;

FSA for L_a:



$\sum = \{a,0\}$
$L_{id} = a(a|0)^*$; ** id's must start with an alphabet

FSA for L_{id}:



Example 3: Can you recognize this with an FSA?

$L_{ab'} = \{a^n b^n \mid 1 \le n \le 5\}$; Yes

Example 4: Can you recognize this with an FSA?

int a0, a15, a3; Yes

Example 5: Can you recognize this with an FSA?

$L_{ab} = \{a^n b^n \mid n \ge 1\}$; No

Write a CFG to generate L_{ab}:

$S \longrightarrow a\ S\ b$
$S \longrightarrow a\ b$

January 18, 2011

Parsing Errors (Context-Free)

$L_{ab} = \{a^n b^n | n \geq 1\}$ is not a regular language and thus cannot be recognized by an FSM.

We use context-free grammars (CFGs) to *generate* sentences for a context-free language. The CFG for L_{ab} is as follows:

$S \rightarrow ab | aSb$

Capital letters represent non-terminals (*S* is the starting non-terminal); lowercase letters represent terminals.

Write a CFG to generate: *int a0, a1, a15;*

$S \rightarrow$ "int" M ";"
$M \rightarrow ID$ "," $M \mid ID$

Pushdown automata (PDAs) *recognize* the languages generated by context-free grammars.

Common Compiler Tools
lex **–** lexical analyzer generator
yacc – parser generator
(Generated code for lex and yacc is C)
ANTLR – parser generator for Java

Context-Sensitive Errors

Example:
int x;
bool x; ←error

Context-sensitive errors include type errors, argument mismatch errors, inheritance-hierarchy errors, and declare-use errors.

To catch these errors, we can use an *attribute grammar*.

An attribute grammar consists of:
1. Context-free grammar
2. Attributes (two kinds: synthesized and inherited)
3. Attribute evaluation rules
4. Conditions

Example:

$L_{abc} = \{a^n b^n c^n | n \geq 1\}$

Attribute grammar:

$A \rightarrow a$
$aCount(A) \leftarrow 1$

$A \rightarrow aA_1$
$aCount(A) \leftarrow aCount(A_1) + 1$

$B \rightarrow b$
$aCount(B) \leftarrow 1$

$B \rightarrow bB_1$
$aCount(B) \leftarrow aCount(B_1) + 1$

$C \rightarrow c$
$aCount(C) \leftarrow 1$

$C \rightarrow cC_1$
$aCount(C) \leftarrow aCount(C_1) + 1$

$S \rightarrow ABC$
Condition:
$aCount(A) = bCount(B) = cCount(C)$

Attributes

Name	**Non-terminal**	**Type**	**Kind**
aCount	A	number	synthesized
bCount	B	number	synthesized
cCount	C	number	synthesized

Attribute Grammars (Manan Gupta's notes)

1. CFG
2. Attributes on non-terminals
3. Attribute evaluation rules
4. Condition

Example:

    int x, y, z;
    bool b1, b2;

Solution

S → <Int_Decls> <Bool_Decls>

Condition: Var_Names(Int_Decls) ∩ Var_names(Bool_Decls) = ∅

<Int_Decls> → "int" <Decls>

    Var_Names(Int_Decls) ← Var_Names(Decls);

<Bool_Decls> → "bool" <Decls>

    Var_Names(Bool_Decls) ← Var_Names(Decls);

<Decls> → <Id>";"

    Var_Names(Decls) ← {Var_Name(Id)};

<Decls> → <Id>","<Decls$_1$>

    Var_Names(Decls) ← Var_Names (Decls$_1$) U {Var_Name(Id)};

Attributes

NT	Name	Type	Kind
<Id>	Var_Name	Char string	Synthesized
<Decls>	Var_Names	Set of char strings	Synthesized
<Int_Decls>	Var_Names	Set of char strings	Synthesized
<Bool_Decls>	Var_Names	Set of char strings	Synthesized

Rule for Synthesized attribute:

- For every non-terminal,
  - For every production rule on which the non-terminal is on the left hand side,
    - For every synthesized attribute,
      - Be sure to give a value!

Inherited Attributes

Generate $L_{abc} = \{a^n b^n c^n \mid n \geq 1\}$ using inherited & synthesized attributes.

S → ABC

bExpCt(B) ← aCount(A);

cExpCt(C) ← aCount(A);

A → a

aCount(A) ← 1;

A → aA_1

aCount(A) ← aCount(A_1) + 1;

B → b

Condition: bExpCt(B) = 1;

B → bB_1

bExpCt(B_1) ← bExpCt(B) - 1;

C → c

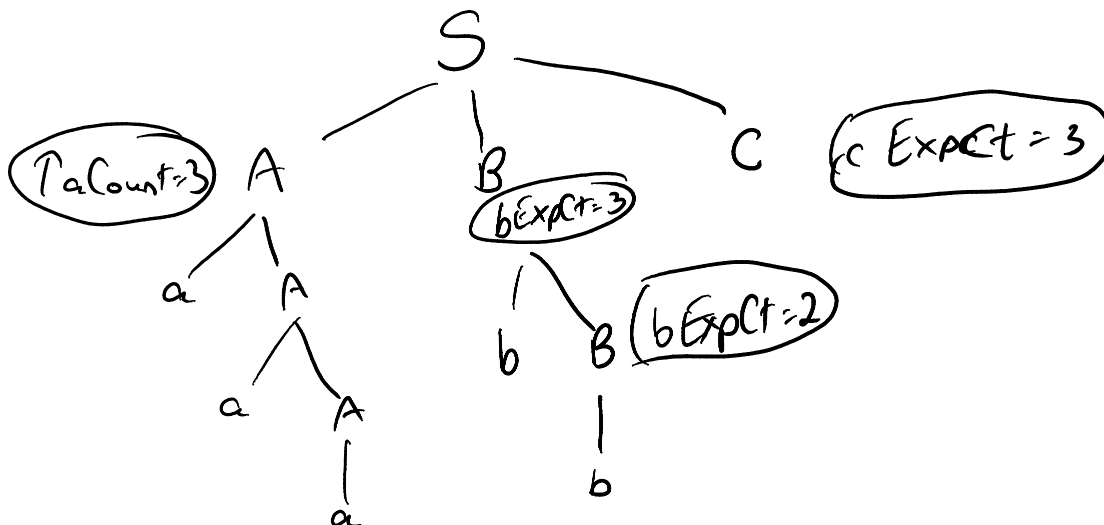
Condition: cExpCt(C) = 1;

C → cC_1

cExpCt(C_1) ← cExpCt(C) - 1;

Attributes

NT	Name	Type	Kind
A	aCount	Integer	Synthesized
B	bExpCt	Integer	Inherited
C	cExpCt	Integer	Inherited



CP SC 828

02-01-2011

Rules for inherited attributes

For every non terminal

For every production rule in which the NT on the RHS

For every inherit attribute,

be sure to give a value

synthesized: LHS

Ex:

S → <int_decls><bool_decls>

<int_decls> → “int” <decls>

<bool_decls> → “bool” <decls>

<decls> → <Id>”;” | <Id> “,” <decls>

Solution:

Attributes

Name	**NT**	**Type**	**Kind**
Vname	<Id>	Charstr	Synthesized
Vnames	<decls>	Set of charstr	Synthesized
Vnames	<int_decls>	Set of charstr	Synthesized
Vnames	<bool_decls>	Set of charstr	Inherited

<decls> →	<Id>”;” Vnames(<decls>) ← Vname(<Id>),	<Id> “,” <$decls_1$> Vnames(<decls>) ← { Vnames($decls_1$)} ∪ {Vname(<Id>};
<int_decls> →	“int” <decls> Vnames(<int_decls>) ← Vnames(<decls>);	
S →	<int_decls><bool_decls> Vnames(<bool_decls>) ← Vnames(<int_decls>);	
<bool_decls> →	“bool” <decls> Condition: Vnames(<bool_decls>) ∩ Vnames(<decls>) = ∅	

Attribute translation grammar (ATG)

An ATG is an AG, without condition

Ex: translate binary to decimal

1001 → 9

Attributes:

Name	NT	Type	Kind
Dval	<bits>	Number	Synthesized
Bval	<bit>	Number	Synthesized

<bits> →	$\langle bits_1 \rangle$<bit> Dval(<bits>) ← Dval(<bits>) * 2 + Bval(<bit>)	<bit> Dval(<bits>) ← Bval (<bit>)
<bit> →	0 Bval(<bit>) ← 0	1 Bval(<bit>) ← 1

Another ex:

<statements> → <statements>; | $\langle statement_1 \rangle$<statement>

<statement> → <assignment statement><if then statement>

Name	NT	Type	Kind
Code	<statement>	Charstr	Synthesized
Code	<statements>	Charstr	Synthesized

If <B> then <statements> end;

Code (<statement>) ←

EVAL B
BRF L1
Code (<statements>)
L1 : skip

Formal_Syntax and formal Semantics: basis for precise communication between programmers(language users) and compiler_writers (language implementers)

Example:

If B then <statements> end;
while B do <statements> end;   // you have to know the exact meaning of these two statements

"Formal" makes it machine-possible

Formal semantics:

1, operational sematics : assume you have known something and then explain the new things in form of what you have known

2, denotational sematics

Operational semantics "operational" description of semantics can be given, for example, using an ATG

benefits: "compiler friendly"

negatives: may not be " user friendly" ; need to understand the lower level system implemented biased

Denotational semantics : abstract semantics (desirable!)
doesn't get down to implementation details

From Syntax to Semantics

Example: specs : write code to exchange the values of I and J

```
I := sum (I, J);
J := Difference (I, J);
I := Difference (I, J);
```

Is this correct?

Several assumptions are involve: sum-overflow?
relationship between Min_int & Max_int
I, J are integers (so min_int <= I, J <= Max_int);
Min_int <= I + J <= Max_int;
Assumption that "sum" and "difference" behave in a certain way;
code runs top to bottom, one step at a time;
I and J are not coupled;

They all need to be formally expressed!

Theory of Programming Languages

February 8, 2011

Jerone Dunbar

Formal Syntax to Formal Specification

I := I + J ;		I := Sum (I, J);
J := I – J ;	≡	J := Diff (I, J);
I : = I – J ;		I := Diff (I, J);

We have to understand the explicit side of things.

Think of programming integers as math Integers, with constraints on bounds. With this understanding, Operation Sum is a function as specified below.

Operation Sum(x, y: integer): Integer;
    requires Min_Int <= X + Y <= Max_Int;
    ensures Sum = (x + y);

<u>What is the meaning of "+" in mathematics?</u>

Formalization of Number Theory -> Integers

Z - Set of Integers   Formalization of Number Theory

N - Natural Numbers

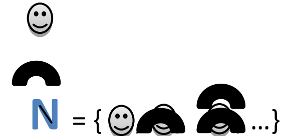
N = {0, 1, ….}

All natural numbers start at 0

0

Suc

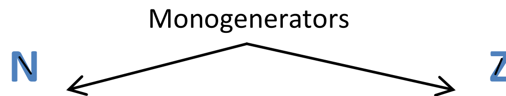
N = {0, Suc(0), Suc (Suc (0)),…}



{ } - The empty set

{{}} - The set containing the empty set

N = { {}, {{}}, {{{}}}, ,…}



Precis is a short version of a Theory.

Precis Natural_Number _Theory
uses Monogenerator theory,…

Assumption N : Set and Is_Monogeneric_for (N, 0, suc)=true;

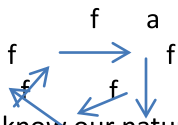
N is a number of the set
Successor of the set,
0 is a member of the set

…

Precis Monogenerator theory;

Definition Is Monogeneric – for (D: Set, a:D, f:D -> D) : B =
(Pty 1: forall x: D, f(x) notequalto a;
Pty 2: forall a, y:D ; f(x) = = f(y) -> (x=y)
f is injective
Pty 3: forall D(D), if a **E** S and forall x : S, f(x) **E** S then S = D)
Predicate is true of false, yes of no in this case
If the member, function and set satisfy certain condition.

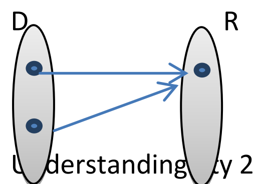
The verifier will assume that Set and B is built in. The S is so huge
Set of all sets that we want to talk about in Computer Science
a:S -> S is a function
Understand Pty 1;
it precludes =



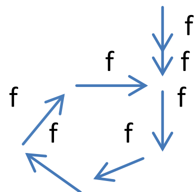
f 0 only one member in set

We know our natural number set is not finite
What can happen is

injective means one to one

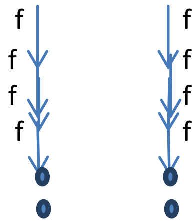


Understanding Pty 2
It precludes



Property #2 involves making sure that the set is finite

Understanding Pty 3



P-power set

S is a member

Forall s: P(D)

S is come set of subset of D (including D)

suppose D is this set

a is S

All the properties are Satisied

Pty 3 precludes D from being multiple disjoint infinite Set S.

Initial concern was is the Set is big enough, then it was about getting it right.

CpSc 828

Ryan Cooke

Notes for February 10. Note: [] before symbol or key terms means definition of that term. For example [for all] ∀ means ∀ is the symbol that represents for all.

Continuation of Monogenerators:

What was our motivation for learning monogenerators?

- We want to understand operations like + & -
- So we ask how do we know + & 0
  - For this we needed number theory
    - And for number theory we use monogenerators as a way to take out common elements
      - Working with a foundation of Boolean

[summary] Precis Mongenerator_theory:

Note there is an excellent handout with much of these notes on it. If you don't have that handout I suggest talking to Dr. Murali.

Definition IS_Monogeneric_for(D: **Set**, a: D, *f*:D -> D): (bool)**B =**

{

Pty 1: [for all]∀x: D, *f*(x) != a;

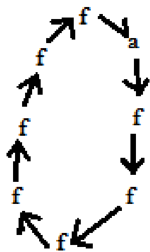
Pty 2: ∀x, y: D (*f*(x) = *f*(y)) => (x = y);

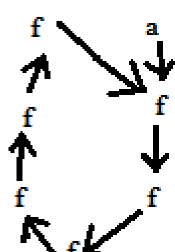
Pty 2 stated differently: Is_injective(*f*);

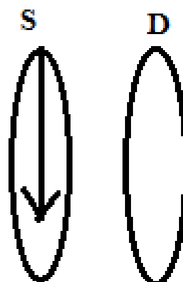
Pty 3: ∀s : [power set of]**P**(D),

If(a[element of]∈s *and* ∀x: s, x ∈ S, *then* S=D;

};







Précis: Natural_Number_Theory:

Uses: Monogenerator_Theory..

Assumption: N: **Set** and Is_monogeneric_for(N, 0, [successor]suc);

// we know there is a hidden function in F(n) -> the fibonnici sequence

Fib(0) = 0

Fib(1) = 1;

Fib(n) = fib(n-1) + fib(n-2)

“Now we are ready to say what plus is” (having covered monogenerators number theory and hidden functions)

Precis: Natural Number_Thoery:

Uses: Monogenerator_Theory, basic_binary_operation_properties

Assumption: N: **Set** and Is_Monogeneric_for(N, 0 Suc):

Inductive definition on n: N of (m: N) + (n:N) is

(i) M + 0 = m;
(ii) M + suc(n) = suc(m+n)

Definition 1: N = suc(0);

Example with numbers:

	N is 0	
M	suc(n)	
Suc(suc(0))	suc(0)	= suc(suc(suc(0)) + 0)

// Basically what we did was use suc to allow the inductive definition to iterate through all cases

// this was to show we can do lots of definitions

The main point of this is to show how to formalize the math.

HW #3 is due Thursday, 2/17/2011
Test #1 is on Tuesday, 2/22/2011

Precis Natural_Number_Theory;
    uses ... ;

    Assumption N : *Set* and Is_Monogeneric_For (N, 0, suc);

    Inductive Definition on n : N of (m : N) + (n : N) is
      i. m + 0 = m;
      ii. m + suc (n) = suc (m + n);

    Theorem N1: Is_Associative (+);
    …

Proof for Natural_Number_Theory:
    Proof for N1:
        Goal: Is_Associative (+);
        Goal: ∀ k, m, n: N, (k + m) = n = k + (m + n);
        Definition: S_1 : ℙ (n) = { n: N | ∀ k, m : N, (k + m) + n = k + (m + n)};
                Suppose $S_1 \subseteq$ N and it satisfies the associative property of +.
        Goal: S_1 : N;
        Goal: 0 ∈ S_1 and ∀ n ∈ S_1, suc (n) ∈ S_1;

        Part 1:
        Goal: 0 ∈ S_1;
        Goal: ∀ k, m : N, (k + m) + 0 = k + (m + 0);
        Goal: If k, m : N, then (k + m) + 0 = k + (m + 0);   ∵ Universal Instantiation
        Supposition: k, m : N;
        Goal: (k + m) + 0 = k + (m + 0)
                = k + (m)   ∵ Property 1 of +
                = k + m
                = (k + m)
                = (k + m) + 0   ∵ Property 1 of +
        Deduction: If k, m : N, then (k + m) + 0 = k + (m + 0);
        ∴ ∀ k, m : N, (k + m) + 0 = k + (m + 0);   ∵ Universal Generalization
        ∴ 0 ∈ S_1;

        Part 2:
        Goal: ∀ n ∈ S_1, suc (n) ∈ S_1;
        Goal: ∀ n ∈ S_1, ∀ k, m : N, (k + m) + suc(n) = k + (m + suc(n));
        Goal: If n ∈ S_1 then   ∵ Universal Instantiation
                If k, m : N, (k + m) + suc(n) = k + (m + suc(n));
        Supposition: n ∈ S_1;

Goal: If k, m : N, (k + m) + suc(n) = k + (m + suc(n));

Supposition: k, m : N;

Goal: (k + m) + suc(n) = k + (m + suc(n));

= k + (suc(m + n)); ∵ Property 2 of +

= k + suc(m + n);

= suc(k + (m + n)); ∵ Property 2 of +

= suc((k + m) + n); ∵ Inductive Supposition

= (k + m) + suc(n); ∵ Property 2 of +

Deduction: If k, m : N, (k + m) + suc(n) = k + (m + suc(n));

…

(See Handout)

Precis Integer_Theory;

uses Monogenerator_Theory, ... ;

Assumption Z : *Set* and Is_Monogeneric_For (Z, 0, NB);

Inductive Definition on n : Z of Is_Neg (n) : **B** is

i. Is_Neg (0) = false;
ii. Is_Neg (NB(n)) = not Is_Neg (n);

Inductive Definition on n : Z of – (n) : Z is

i. – 0 = 0;
ii. – (NB(n)) = **?** (Question to think about)

2-24-2011

Formal Semantics and Program Correctness

Validity – formal denotational semantics can be used to define validity.

or put informally:

> Assertive code is valid if the program on starting on a state that satisfies the assumption ends in a state where the assertion at the end can be confirmed, and nothing else "goes wrong" in the middle.

Example 1
Context/
Assume $I > 0$;
skip;
Confirm $I = 17$;
✗

Example 2
Context/
Assume $I = 17$;
skip;
Confirm $I > 0$;
✓

Example 3
Context/
Assume $I > 0$;
$I :=: J$;
$J :=: K$;
Confirm $K > 0$;
✓

Example 4
Context/
Assume $I = 17$;
If $(I < 0)$ $I :=: J$;
Confirm $I > 0$;
✓

Example 5
Context/
Assume $I > 0 \wedge J > 0$;
$I :=: J$;
Confirm $I > 0 \vee J > 0$;
✓

Example 6
Context/
Assume true;
$I :=: J$;
skip;
Confirm $I \neq 0$;
✗

Provably correct – (informal) assertive code is provably correct iff there is a proof.

assertive code => (verification system) => provably correct or not?

Verification system is based on a proof system for a language.

The proof system consists of proof rules for each language construct.

Example:

Context/

    Assume I > 0;

        I :=: J;

        J :=: K;

    Confirm K > 0;

Proof rule for a swap statement:

$$\frac{\text{C/assertive_code; Confirm Q[X}\leadsto\text{Y, Y}\leadsto\text{X]}}{\text{C/assertive_code; X :=: Y; confirm Q;}}$$

CPSC 828 – note for Thursday, March 03, 2011

<u>Example:</u>

```
Context / K:=: I;
If GT(J,K) then
   K :=: J;
End;
Confirm K = Max (I,J);
```

Program

Precis Integer_theroy;

…

Definition

Max(I, J: **Z**):**Z**

$$\begin{cases} I \text{ if } I > j \\ J \text{ otherwise} \end{cases}$$

Mathematic

Concept Integer_template

Uses Integer_theory

...

Type Integer ⊆ **Z**;

Operation GT(X,Y: Integer):Boolean;

Ensures: Result of GT = (X > Y);

Bridge

<u>If – then proof rule</u>

1. c/ assertive code; **Assume** B; code1, **confirm** Q
2. c/ assertive code; **Assume** ~B; **confirm** Q

---

3. c/ assertive code
   If B then
   Code1;
   End;
   Confirm Q;

1. apply If – then rule:

1.1. context / K :=: I
Assume J>K;
K :=: J;
Confirm K= Max (I , J);

2.1. apply swap statement rule
c/ K:=: I;
Assume J > K;
Confirm J = Max (I , K);

3.1. Apply assume rule
c/ K :=: I;
Confirm (J > K) => J = Max (I , K);

4.1. Apply swap statement rule
c/ **Confirm** (J > I) => J= Max (K , I);

1.2. Context / K :=: I;
Assume ~(J>K);
Confirm K = Max (I , J);

2.2. Apply assume rule
c/ K :=: I;
Confirm ~(J > K) => K = Max (I , J);

3.2. Apply swap statement rule
c/ **Confirm** ~(J > I) => I = Max (K ,J);

The assertive code cannot be proven ➔ it is not correct and invalid

Notes for CpSc 828
March 15, 2011

Hw# 5 is due Thursday, March 17, 2011

Proof rule for Function Operation calls

Context/assertive_code; Confirm Q[v <~ f(u)];
Context/assertive_code; v := F(u); Confirm Q;

Where context includes:
Operation F(restores x:T_1): T_2;
ensures F = f(x);

Simple Operation call rule

Confirm
Context/assertive_code; ~~Assume~~ pre_P[x <~ u, y <~ v];
Confirm Q [v<~ post_P[x <~ u, #y <~ v];
Context/assertive_code; P(u, v); Confirm Q;

Where context includes:
Operation P(restores x:T_1; updates y: T_2);
requires pre_P <x, y>;
ensures y = post_P<x, #y>;

Operation Defensive_Pop (...)
ensures s = { #s if s = Λ
{ … otherwise

EXAMPLE
Context/Assume T = Λ;
I := Depth(T);
Confirm I = 0;

1. Apply the function call rule
   Context/Assume T = Λ;
   Confirm I = 0[I <~ |T|];

2. Simplify Confirm clause
   Context/ Assume T = Λ;
   Confirm |T| = 0;

   True;

Side note:
Whose responsibility is the pre-condition?
1. Caller ✓
2. ~~Implementer~~
3. ~~Both~~
4. ~~Neither~~

** **Specifications are contracts**

March 17, 2011

Simple operation call rule

```
c/assertive_code; confirm pre_p[x<~U, y<~V] and Q[V<~post_P[x<~U, #y<~V]];
--------------------------------------------------------------------------
c/assertive_code; P(U, V); confirm Q;
where context includes
        operation P(restores x: T1, updates y: T2);
                requires pre_P<x, y>;
                ensures y = post_p<x, #y>;
```

Procedure or Code Declaration Rule

```
c/Assume pre_p;  Remember;
        P_body;
Confirm y = post_p and x = #x;
--------------------------------------------------------------------------
c/Procedure P(restores x: T1, updates y: T2);
        P_body;
end P;
where context includes
        operation P(restores x: T1, updates y: T2);
                requires pre_P<x, y>;
                ensures y = post_p<x, #y>;
```

Example

```
context (includes stack_template)/
        operation Do_Nothing(restores S: Stack);
                requires |S| > 0;
        Procedure Do_Nothing(restores S: Stack);
                var Temp: Entry;
                Pop(Temp, S);
                Push(Temp, S);
        end Do_Nothing;
```

Alternative (better?) Specification of Pop

```
operation Pop(replaces R: Entry; updates S: Stack);
        requires|S| > 0 and consequently Prt_Btwn(0, 1, S): Prime_Str(Entry);
        ensures R = >Prt_Btwn(0, 1, #S)< and S = Prt_Btwn(1, |#S|, #S);
```

CpSc 828

8/29/11 (Tuesday)

Procedure Declaration Rule:

Context/ assume constraints

Assume pre_P <x,y>;

Remember;

    P_body;

    Confirm y = post_p<x, #y> and x=#x;

Context/procedure P(rest x:T1, upd y: T2);

    P_body;

    End;

Where context includes

    Operation P(rest x: T1; upd y: T2);

    Requires pre_p<x, y>;

    Ensures y = post_p(x, #y);

Example:

Operation Do_Nothing(rest s: stack);

    Requires |s| > 0;

Procedure

    Var Temp: Entry;

    Pop(Temp, s);

    Push(Temp, s);

End Do_Nothing;

For Reference

Operation Push (alters E:Entry, updates S:Stack);

    Requires |S| < max_depth;

    Ensures S = <#E> o #S

Operation Pop(replace R: Entry; updates S: stack);

    Requires |S| > 0;

    Ensures R = >Prt_Btwn(0,1,#S)< and

        S= Prt_Btwn(1,|#S|, #S);

After making the reference push and pop we went into a tangent on why to use alters in push. To demonstrate the value we made a stack of books from various students. The basic idea was if it was restores it would have had to make a copy of the book to put on the stack. So, alters gives the entry more flexibility than other uses.

We also discussed difficulties related to outside access that java grants when you put a pointer in a stack. The general conclusion was this makes verification difficult.

Back to example:

1) Apply procedure declaration rule:

c/ assume |S| <= Max_depth;

assume |S| > 0;

    Remember;

    Var Temp: entry;

    Pop(Temp, S);

    Push(Temp, S);

    Confirm S = #S;

2) Apply simple operation call rule
    a. c/… (is the same)
       confirm |S| < max_depth and S=#S[S~> (<#E> o #S) [#E ~> Temp, #S ~> S]]
    b. c/…
       pop(Temp, S);
       confirm |S| < Max_depth and S= #S[S~><Temp> o S];
    c. c/…
       pop(Temp, S);
       confirm |S| < max_depth and <Temp> o S = #S;

3) Apple simple op. call rule
   a. c/…
      var Temp: Entry;
      confirm |S| > 0 and (|S| < Max_depth and <temp> o S = #S) [Temp <~ >Prt_Between(0, 1, S) <   S <~ Prt_Btwn(1, |S|, S)];
   b. c/…
      Var Temp: Entry
      Confirm |S| > 0 and (|Prt_Btwn(1,|S|, S)| < max_detph and Prt_betwn(0, 1, S) o prt_btwn(1, |S|, S) = #S;

// note var decl rule says declared values are initialized

4) Apply var decl. rule
5) Apply remember rule
   c/ assume |S| <= max_depth;
   assume |S| > 0;
   confirm |S| > 0 and |prt_btwn(1, |S|, S)| < max_depth and prt_btwn(0,1,S) o prt_btwn(1, |S|, S) = S);

Class note for March 31, 2011

Some tips about homework :

① <u>context/assertive code; confirm Q[x <- y][y <- x];</u>

Context/assertive code; x : = : y ; confirm Q;

[If a rule is unsound, it is no sense to talk about completeness of the rule]

Unsoundness: Find an example that is invalid but provably correct

Example : Assume I = 0;

I : = : J;

Confirm I : = 0;

② context/assertive code; code; confirm Q;

<u>Context/assertive code; Assume NBn ; confirm Q;</u>

Context/assertive code; If B then code end; confirm Q;

[If give less assumptions, you can't make it unsound, just less code can be provably correct]

Example : context / Assume (I ≠ 0 => J = 0) ;

If (I = 0) then

I : = : J;

End;

Confirm J = 0 ; [valid but not provable]

③ simple var decl rule

<u>Context / assertive code; confirm Q [x <- T.init_val]</u>

Context / assertive code; var x : T ; confirm Q;

Example / Assume T = Ω ;

Var S : stack ;

Confirm S®T = Ω ;

Valid and provably correct

General operation call rule

Context / assertive code ; confirms pre_p [x <- U, y <- V] and

For all V' : T2 (post_p [x <- U, # y <- V, y <- v']) implies Q [V <- V'] ;

---

context / assertive code : P (U, V); confirm Q;

where C includes

operation P (rest x : T1 ; update y : T2);

requires pre_p < x, y >;

ensures post_p < x, # y, y >

examples of implicit ensures clauses :

operation pop (replace R : entry ; update S : stack);

requires |S| ≠ 0;

ensures #S = <R> ® S;

The above is a functional specification expressed implicitly. The general purpose of the rule is for application to relationally-specified operations where multiple outputs can result for a given input. Examples include specification of an operation to find an MST of a graph (because there may be many MST's for a given graph) and optimization problems.

Verification of Do_nothing with implicit specs for pop

① context / ….

Push (Temp , S);

Confirm S = # S;

② Apply simple call rule

Context / ….

Pop (Temp, S);

Confirm (|s| < max_depth) and (<Temp>®S = # S);

③ Apply general call rule

Context / Assume | S | > 0 and | S | ≤ max_depth

Remember;

Confirm (|S| > 0) and □ Temp’ : Entry , for all S’ : Stack

(S = <Temp’>®S’ => (|S’| < max_depth and <Temp’>®S = # S));

Notes for 4-5-2011

Test 2 on Tuesday

Everything from the 1st test

Included: everything from validity, provable correctness, etc (HW 4-6)

Not included: inductive principle

Review

An <u>unsound</u> rule lets you prove invalid code to be correct

An <u>incomplete</u> rule does not let you prove valid code to be correct

Mathematics vs code

Assertions are always mathematical

Mathematical: Ensures Depth = (|S|)

Code: If (Depth(S) > 0)

```
C/Assume I ≠ J;
If (I = J) then
    I :=: J;
End;
Confirm I ≠ J;
```

1.1

```
C/Assume I ≠ J;
Assume I = J;
I :=: J;
Confirm I ≠ J;

C/Assume I ≠ J;
Assume I = J;
Confirm I ≠ J;

C/Assume I ≠ J and I = J;
Confirm I ≠ J;

C Assume false;
Confirm I ≠ J;

true
```

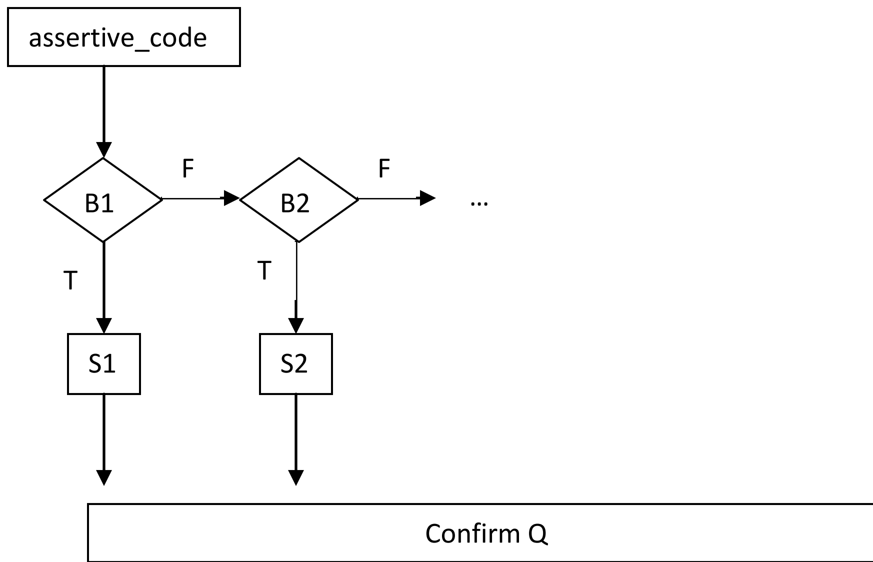
1.2

```
C/Assume I ≠ J;
Assume not(I=J);
Confirm I ≠ J;

C/Confirm (I≠J ⇒ I ≠ J);

true
```

Switch flow chart



HW #6

Context/assertive_code;

Confirm

Pre_P[X⇝U, Y⇝V] and

∀ U':T2 (

Q[

U⇝U',

V⇝T2.initial_val,

W⇝post_P(#X⇝U, #Y⇝V)

]

)

Context/assertive_code; P(U,V,W); Confirm Q;

where Context includes:

Operation P(alters x:T1; clears y:T2; replaces z:T3)

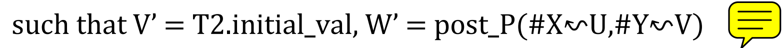
Requires pre_P(x,y)

Ensures z = post_P(#x,#y);

Another way to look at it:

∀ U’:T1, V’:T2, W’:T3

such that V’ = T2.initial_val, W’ = post_P(#X↝U,#Y↝V)



Q[U↝U’, V↝V’, W↝W’]

Procedure Declaration Rule

C/Assume constraints and pre_P;

Remember;

P_Body;

Confirm Z = Post_P and y = T2.initial_val;

C/Procedure P(x,y,z)

    P_body;

End;

Where Context includes …

4/19/2011

Verification of Iterative Implementation

Ex: Operation Flip(updates S: Stack):

```
        ensures S = #S^Rev
    Procedure
        Var Next_Entry: Entry;
        Var S_Flipped: Stack;

        While (Depth(S) ≠ 0)
            changing Next_Entry, S, S_Flipped;
            decreasing |S|;
            maintaining ...;
        do
            Pop(Next_Entry, S);
            Push(Next_Entry, S_Flipped);
        end;

        S_Flipped :=: S;
    end Flip;
```

Loop Invariant: It is an assertion that it is true at the beginning and at the end of each iteration, including the first and the last.

Simple partial correctness loop rule:

(1) C/ Assertive_Code; Confirm Inv;

(2) C/ Assume B_M and Inv; Body; Confirm Inv;

(3) C/ Assume $\sim B_M$ and Inv; Confirm Q;

– –

```
C/ Assertive_Code;
    While (B)
        maintaining Inv;
    do
        body
    end;
    Confirm Q;
```

Need an inductive proof of the invariant.

a) base case b) inductive case

So we need to show:

$|S| = 0 \wedge ??? \implies S_Flipped = \#S^{Rev}$

$??? = S^{Rev} \circ S_Flipped = \#S^{Rev}$

Proof of Example: (Partial Correctness)

(1) Apply procedure declaration rule

(2) Apply swap rule

```
Assume |S| ≤ Max_Depth;
Remember;
Var ... ;
While (Depth(S) ≠ 0)
      maintaining ... ;
do
      ...
end;
Confirm S_Flipped = #S^Rev;
```

(3) Apply while loop rule

3.1

```
Assume |S| ≤ Max_Depth;
Remember;
Var Next_Entry: Entry;
Var S_Flipped: Stack;
Confirm S^Rev o S_Flipped = #S^Rev;
```

3.2

```
Assume |S| ≠ 0 ∧ S^Rev o S_Flipped = #S^Rev;
      Pop(Next_Entry, S);
      Push(Next_Entry, S_Flipped);
Confirm S^Rev o S_Flipped = #S^Rev;
```

3.3

```
Assume |S| = 0 ∧ S^Rev o S_Flipped = #S^Rev;
Confirm S_Flipped = #S^Rev;
```