Programming Languages
(CS 550)
Lecture 2 Summary
Mini Language Interpreter

Jeremy R. Johnson
Theme

- This lecture builds an interpreter for the mini language from Louden Chapter 13 of the text. A parser is written that translates the input program into a data structure that can easily be interpreted. The language is extended to support procedures.
Outline

- Go over last week’s practice exercise
- Introduce Mini Language syntax and semantics
  - Environments (Symbol Table)
- Abstract syntax tree (more yacc and attribute grammars)
- Mini Language Interpreter
- Exercise 1: Modify the Mini Language and interpreter to support “repeat … until” statement
Outline

- Adding user defined functions to the mini language
  - parameter passing
  - local variables (local environment)
  - function application
    - Execute procedure body in local environment with formal parameters bound to actual argument values
  - return value
  - recursion

- Exercise 2: Modify the extended Mini Language and interpreter to use an explicit return statement
Outline

- Discuss assignment 2
  - Add lists to the mini language
  - Values are now lists or ints (modify Environment)
  - Support list constants and built-in list processing functions
    - cons( e, L ) - appends element e to the front of list
    - car( L ) - returns the first element in the list
    - cdr( L ) - returns the rest of the list (minus the first element)
    - nullp( L ) - returns 1 if L is null, 0 otherwise
    - intp( e ) - returns 1 if e is an integer, 0 otherwise
    - listp( e ) - returns 1 if e is a list, 0 otherwise to allow construction and access to lists.
Mini Language Syntax

1. < program > → < stmt-list >
2. < stmt-list > → < stmt > ; < stmt-list > | < stmt >
3. < stmt > → < assign-stmt > | < if-stmt > | < while-stmt >
4. < assign-stmt > → < identifier > := < expr >
5. < if-stmt > → if < expr > then < stmt-list > else < stmt-list > fi
6. < while-stmt > → while < expr > do < stmt-list > od
7. < expr > → < expr > + < term > | < expr > - < term > | < term >
8. < term > → < term > * < factor > | < factor >
9. < factor > → ( < expr > ) | < number > | < identifier >
10. < number > → < number > < digit > | < digit >
11. < digit > → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
12. < identifier > → < identifier > < letter > | < letter >
13. < letter > → a | b | c | ... | z
Operational Semantics

- The meaning of a program is obtained by interpreting the result of each statement in some model of computation.
- We use C++ to interpret execution of each statement.
- Louden [chapter 13] uses a reduction machine which we will discuss later in the term.
Environments

- Let an Environment be a map from identifiers to values = integers $\cup$ undefined

- Mini language programs can be thought of as a map from an initial Environment to a final Environment (assuming it terminates)

- The initial environment maps all identifiers to an undefined

- Each statement is defined in terms of what it does to the current environment (another mapping)
Semantics of Mini Language Statements

1. Env: Identifier → Integer Union {undef}

2. \((\text{Env and } \{I = n\})(J) = n \text{ if } J = I, \text{ Env}(J) \text{ otherwise}\)

3. Env\_0 = undef for all I

4. for if-stmt, if expr evaluates to value greater than 0, then evaluate stmt-list after then, else evaluate stmt-list after else

5. for while-stmt, as long as expr evaluates to a value greater than 0, stmt-list is repeatedly executed and expr evaluated.
Example Mini Language Program

1. n := 0 - 5;
2. if n then i := n else i := 0 - n fi;
3. fact := 1;
4. while i do fact := fact * i; i := i - 1 od

What is the final environment?
Implementing the Interpreter

- Syntax directed semantics
- The interpreter is implemented by creating a class, with an evaluate method, for each syntactic category. Use inheritance to derive specialized statements from more general categories of statements. When the parser detects a syntactic category the corresponding constructor is called. A map is used to store the environment and the program is executed by calling all of the evaluate methods of the statements in the program.
Adding Functions

- In this implementation we will insist that all functions are closed. I.E. they only communicate with the calling environment through parameter passing and their meaning is determined solely from the statements in their definition and the parameter values.
  - parameter passing
  - local variables (local environment)
  - separate function table
  - function application
    - Execute procedure body in local environment with formal parameters bound to actual argument values
  - return value
  - recursion
Example Mini Language Procedure

define add
proc(n)
    i := n;
    s := 0;
    while i do s := s + i; i := i-1 od;
    return := s
end;
n := 5;
s := add(n)

What is the final environment?
Example Recursive Mini Language Procedure

define addr
proc(n)
  if n then return := n + addr(n-1) else return := 0 fi
end;
n := 5;
s := addr(n)

What is the final environment?
What Next?

- **Dynamic memory management and garbage collection**
  - Assignment: modify mini language and interpreter to handle dynamic memory management and garbage collection

- **Functional programming**
  - Functions as first class objects
  - Introduce proc() … end as a value
  - Assignment: modify mini language and interpreter to support this functional programming