****These are notes for Prof. Janikow's section****

P3 Code Generation + Storage Allocation

Submission command:
`/accounts/classes/janikowc/submitProject/submit_cs4280_P3` *`SubmitFileOrDirectory`*

Invocation
`comp [`*`file`*`]`
where *`file`* is an optional argument. If the *`file`* argument is not given the program will read data from the keyboard as a file device. If the argument is given, the program reads data file *`file`*`.fs17`. (note that *`file`* is any name as given and the extension is implicit).
Programs improperly implementing file name or executable will not be grade.

Output
If the argument *`file`* is given (and the input is *`file`*`.fs17`) then the program will produce output target *`file`*`.asm`. When reading from the keyboard, the output file should be `out.asm`.

The project has 2 options which will affect all back end parts. You must include README.txt file with your submission stating on the first line Static Semantics: LOCAL or GLOBAL and on the second line Storage: LOCAL or GLOBAL.
Submissions without this information will be tested using the global option.

1. Local - variables inside of a block are scoped in that block
2. Global - all variables are global

The project has 3 related parts:

1. Static semantics (40 global or 60 local)
2. Code generation (120: input 10, output 10, assignment 10, expression 15, sequence 15, if 10, nested if 15, loop 10, nested loop 15, others 10)
3. Storage allocation (40 global or 60 (extra 20) local)

Target language is assembler, which comes with an interpreter (virtual machine).

Deadline is during scheduled final.

Everyone must demo their project in person in my office during the scheduled final except the following students are waived (you may still present in person but not required):

Extension must be approved in person.

Projects will be tested using the above virtual machine for execution of your targets.

Static Semantics

Stat Semantics Definition

• The only static semantics we impose that can be processed by the compiler (static) are proper use of variables.

- **Variables**
  - Variables have to be defined before used first time.
  - Variable name can only be defined once in a scope.

Two options for variable scope.

- **Global** option for variables
  - There is only one scope
- **Local** option for variables
  - Variables outside of a block are global
  - Variables in a block are scoped in this block
  - Rules as in C (smaller scope hides the outer/global scope variable)

Support and processing Global

Software support

- Use any container ST for names such as array, list, etc. with the following interface. It shows String as the parameter, which is the ID token instance, but it could include line number for more detailed error reporting.
  - `insert(String)` - insert the string if not already there or error if already there (you may return fail indication or issue detailed error here and exit)
  - `Bool verify(String)` - return true if the string is already in the container variable and false otherwise (I suggest you return false indicator rather than issue detailed error here with exit but either way could possibly work if you assume that no one checks verify() unless to process variable use)

Static semantics

- Instantiate STV for variables
- Traverse the tree and perform the following (looks like preorder traversal) based on the subtree you are visiting
  - If visiting <vars> or its subtree and you find ID token then call STV.insert(ID) // this is variable definition
  - Otherwise (you are under <stats> and not under another <vars> if you find token ID

        - call STV.verify(ID)

Support and Processing Local

You may process all variables using local scope rules, or process variables in the outside <vars> as global and all other variables as local. This describes the latter.

Software support

Implement a stack adapter according to the following

- Stack item type is String or whatever was your ID token instance. You may also store line number or the entire token for more detailed error messaging
- You can assume no more than 100 items in a program and generate stack overflow if more
- Interface
    - `void push(String);`
        - just push the argument on the stack
    - `void pop(void);`
        - pop, nothing returned
    - `int find(String);`
        -
        - the exact interface may change, see below
        - find the first occurrence of the argument on the stack, starting from the top and going down to the bottom of the stack
        - return the distance from the TOS (top of stack) where the item was found (0 if at TOS) or -1 if not found
-

Static semantics

- Perform left to right traversal, and perform different actions depending on subtree and node visited
    - When working in the outer <vars> subtree
        - process as in the global option (or process as local if desired)
    - When working in a <block>
        - set varCount=0 for this block
        - under <vars>
            - upon each `v` variable definition
            - when `varCount>0` call `find(v)` and error/exit if it returns non-negative `number< varCount` (means that multiple definition in this block)
            - `push(v)` and `varCount++`
        - otherwise (variable use, suppose variable instance is `v`)
            - `find(v)`, if -1 try STV.`verify(v)` (if STV used for the global variables) and error if still not found
        - call `pop() varCount` times when leaving a block (note that `varCount` much be specific to each block)

Test Files - Global Error

```
Var x , y , x .
Begin
  Output 1 ;
End
```

```
Var x , y .
Begin
  Var z, x
  Output 1 ;
End
```

```
Var x , y , z .
Begin
  Var a , b , b
  Output 1 ;
End
```

```
Var x , y .
Begin
   Output z ;
End
```

```
Var x , y .
Begin
   Output x + z ;
End
```

```
Var x , y .
Begin
   Output 1 ;
   If [ x > 1 + z ]
   Begin
      Output 1 ;
```

```
  End
End
```

```
Var x , y .
Begin
  Output 1 ;
  If [ x > 1 ]
  Begin
    Var z , y .
    Output 1 ;
  End
End
```

Test Files - Global Good

```
Var x , y .
Begin
 Output x + y ;
End
```

```
Var x , y .
Begin
 Var z .
 Output x + y + z ;
End
```

```
Var x , y .
Begin
 Begin
   Var z .
   Output z ;
 End
 Output z ;
End
```

Test Files - Local Bad

```
Var x , y , x .
Begin
```

```
  Output 1 ;
End
```

```
Var x , y .
Begin
  Var z, x , x .
  Output 1 ;
End
```

```
Var x , y , z .
Begin
  Var a , b .
  Output w ;
End
```

```
Var x , y .
Begin
   Begin
      Var w .
      Output x ;
   End
   Output w ;
End
```

Test Files - Local Good

```
Var x , y , z .
Begin
  Var a , b .
  Output x + a ;
  Begin
    Var x .
    Output x ;
  End
  Begin
    Var x .
    Output x ;
  End
  Output x ;
End
```

Code Generation + Storage

Language Semantics

- • Basic semantics as in C - program executes sequentially from the beginning to the end, one statement at a time
- Conditional statement is like the else-less if statement in C
- Loop statement is like the while loop in C
- Assignment evaluates the expression on the right and assigns to the ID on the left
- Relational and arithmetical operators have the standard meaning except: % is division and () is negation
- IO reads/prints a 2-byte signed integer
- All data is 2-byte signed integer

Target

Virtual machine based on simple accumulator-based assembler. See CS Students | CS Courses | 4280 |Instructor Corner | Janikow | Simple assembler virtual machine. The machine also supports stack operations needed to implement local scoping rules.

Suggested Methods

Storage allocation

- All storage is 2-byte signed
- Storage needed for
  - program variables
  - temporaries (e.g., if accumulator needs to be saved for later use)
    - temporaries can be added to to global variables pool or allocated locally if using local scoping. I would suggest global. We can assume not to use variables named T# or V# in the source, reserving such names for temporary variables.
    - there is no need to optimize reducing the number of temporaries
- **Global option**
  - storage allocation should follow static semantics and code generation
  - issue storage directive for every global variable and every temporary, using the global storage directive in the virtual machine, after the STOP instruction
- **Local option**
  - global variables and temporaries can be generated as in the global option, temporaries could also be local, or all could be local
  - local variable should be allocated on the virtual machine's system stack during the **single pass which performs static semantics and code generation**
  - modify the static semantics by adding code generation in the same pass
    - code generation discussed separately
    - storage allocation
      - every push() must be accompanied by PUSH in the target
      - every pop() must be accompanied by POP in the target

- every find() returning n>=0 (when used for data use, this means this is local variable) should be accompanied by
  - STACKR n if this is reading access
  - STACKW n if this is writing access

Code generation

- The parse tree is equivalent to the program in left to right traversal (skipping syntactic tokens if not stored in the tree). Therefore, perform left to right traversal
- When visiting a node, generate appropriate code at appropriate time if the node is code-generating
  - a node with no children and no token probably needs no code generated
  - a node with only one child and no tokens probably needs no code generated unless it is action node such as negation
  - a node always generates the same code except for possible different tokens and/or different storage used. Therefore, the code generator can be a set of functions, one function per each node kind, that is one per each parser function. Instead of a set of functions, could use a switch in a single function (in recursive traversal)
  - every code-generating node generates code and the same code regardless of its location in the tree
  - some nodes need to generate some code preorder, some in-order, some post-order, based on the semantics of the code corresponding to this node
  - at the end of the traversal, print STOP to target (to be followed by global variables+temporaries in storage allocation)
- Useful assumptions
  - assume and enforce that every subtree generating some value will leave the result in the accumulator
  - **global option**: separate traversal after static semantics is recommended
  - **local option**: a suggested approach is to perform static semantics, code generation, and storage allocation on the stack in a single pass - start with static semantics traversal and then modify the code
- Variables will require
  - variable creation upon definition - see storage allocation
  - variable access upon use
    - examples in class, such assignment node, were for **global option**
    - for **local option**, the access for local variables needs to be changed to stack access - see storage allocation. Global variables can be processed as in the global option.

CodeGen

Attached Files:

CodeGen.pdf

Gen_If.pdf

If and Loop examples
Attached Files:

ifLoop(1).pdf

Test IO Simple - echo input

```
Var x , y .
Begin
 Input  x ;
 Output  x ;
End
```

Test Assignment Simple - echo input

```
Var x , y .
Begin
 Input  x ;
 y : x ;
 Output  y ;
End
```

Test Expression Simple - print -2 3 5

```
Begin
 Output  2 + 3 - 7 ;
 Output  24 % 2 * 4 ;
 Output  2 - ( 3 ) ;
End
```

Test Expression Complex - print 3 (check it)

```
Begin
 Output  2 + 3 - 7 % 3 * ( [ 2 - 3 ] ) ;
End
```

If Simple - print 1 if negative input

```
Var x , y .
Begin
 Input  x ;
 Check [ x < 0 ]
   Output  1 ;
End
```

Test If Block - print 1 2 if non-negative input

```
Var x , y .
Begin
 Input  x ;
 Check [ x >= 0 ]
 Begin
   Output  1 ;
   Output  2 ;
 End
End
```

Test If Nested - print 1 if input in [1..10]

```
Var x , y .
Begin
 Input  x ;
 Check [ x >= 1 ]
   Check [ x <= 10 ]
     Output  1 ;
End
```

Test Loop - print input down to 0

```
Var x , y .
Begin
 Input  x ;
 Loop [ x >= 0 ]
 Begin
   Output  x ;
   x : x - 1 ;
 End
End
```

Test Loop w/If - print input down to 0 skippoing those less than 5 thus print input down to 5

```
Var x , y .
Begin
 Input  x ;
 Loop [ x >= 0 ]
 Begin
   Check [ x >= 5 ]
     Output  x ;
   x : x - 1 ;
 End
End
```

TestStorage Global - echo 3 inputs in reverse

#Global option on storage test. Works with either local or global static semantics#

```
Var x .
Begin
 Var y .
 Input x ;
 Input y;
 Begin
  Var z .
  Input z ;


  Output z ;
 End
 Output y ;
 Output x ;
End
```

TestStorage Local - echo 3 inputs in reverse

#Storage local option test. Works with local static semantics only

```
Var x .
Begin
 Input x ;
 Begin
   Var x .
   Input x ;
   Begin
     Var x .
     Input x ;
     Output x ;
   End
   Output x ;
 End
 Output x ;
End
```