Code Generation + Storage

Lang sem

- 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
  - access upon use
    - examples in class, such as R node or 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 teh global option.

Test File ...