Code Generation and Optimization

Due May 8, 2018 at 11:59 pm

Total 100 points.

Invocation:
> comp [*file*]

Wrong invocations will not be graded.

The program is to parse the input, generate a parse tree, perform static semantics, optimize code, and then generate a target file. Any error should display detailed message, including line number if available (depending on scanner).

The program has 3 parts to be properly generated:

1. Code generation
    - 40: input 5, output 5, assignment 5, expression 5, expression in condition 5, sequence 5, condition 5, loop 5
2. Optimization
    - 20: Remove unnecessary LOAD statements (see below)
3. Storage allocation
    - 40: Global - all variables allocated globally

Runtime 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 if statement in C, but without an else option
- 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 equality operator '=='
    - '#' is negation
- IO reads/prints a 2-byte signed integer
- All data is 2-byte signed integer

Data

- All data is 2-byte signed integers
- Assume no overflow in operations

Target Language

- VM ACCumulator assembly language
- Description provided in VM_Architecture.pdf and VM_Language.pdf on Canvas

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 to generate the code
- Some nodes are
    - code generating - most likely only those that have a token(s)
    - not code generating - most likely without tokens
        - if no children, return
        - if children, continue traversal then return
- When visiting code generating node
    - some actions can be preorder, some inorder, some postorder
    - temporary variables may be needed - generate global pool variables and allocate in storage allocation as global
    - if value is produced, always leave the result in the ACCumulator
    - each node-kind generates the same code
        - regardless of parents and children
        - may be one of multiple cases
        - the only difference may come from the token found in the node
- At the end of the traversal, print STOP to target (to be followed by global variables+temporaries in storage allocation)
- Variables will require
    - variable creation upon definition - see storage allocation
    - variable access upon use

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 global variables pool. 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 storage**
  - 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

Optimization

- The STORE command in the target language stores the value in the ACCumulator to the input argument's memory location, and the LOAD command moves the value located at the argument's memory location to the ACCumulator
- Thus it is not necessary to load the value if it was stored on the immediately previous instruction, as the value would still be in the ACCumulator
- Your optimization will remove these unnecessary LOAD commands from the output assembly program

Suggestions

Modify main:

- Call code generation function on the tree after calling static semantics function in main, and then call storage allocation function after that. Note that the ST container has to be created in static semantics but accessed in code generation.