

CS 502 – Compiling and Programming Systems
Final and Qualifying Examination Part I, 12/11/2007

Instructions: Read carefully through the whole exam first and plan your time. Note the relative weight of each question and part (as a percentage of the score for the whole exam). The total points is 100 (*ie*, your grade will be the percentage of your answers that are correct).

This exam is **open book, open notes**. You are free to refer to any book or other study materials you bring to the exam room.

You have **2 hours** to complete all five (5) questions. Write your answers on this paper (use both sides if necessary).

Name:

Student Number:

Signature

1. (Parsing, context free grammars; 20%) Consider a simplified form of the expression syntax for the Smalltalk programming language. The “leaf” nodes of expressions are identifiers (terminal *Id*) and integer literals (*Int*). There are three kinds of expressions involving operators:

- Unary expressions: A unary operator is simply an identifier (*Id*), and is written *after* the sub-expression to which it applies. For example, “x sqrt” would compute the square root of x.
- Binary expressions: There are a large number of binary operators, so we will represent them using a terminal symbol *BinOp* (with the specific operator indicated by the semantic value associated with the symbol). They are written in infix form, and evaluate strictly from left to right (i.e., there is no precedence among them). Thus $x+y*z$ means the same thing as $(x+y)*z$.
- Keyword expressions: These take the form $e_0k_1e_1 \dots k_n e_n$, that is, sub-expressions e_i alternating with keywords k_i . Each keyword k_i associates with its paired expression e_i . There can be one or more keywords, with no language specified limit. Keywords are recognized by the scanner and presented as the terminal symbol *Key* (with the specific keyword available as the symbol’s semantic value). Textually, they look like an identifier immediately followed by a colon. An example keyword expression is:

a at: 3 put: (x+y)

Precedence: Unless parentheses are used, one always first applies unary operators, then binary operators, then evaluates keyword expressions. Thus, the expression:

a at: x+y put: (b+c / d sqrt) truncate

is equivalent to

a at: (x+y) put: (((b+c) / (d sqrt)) truncate)

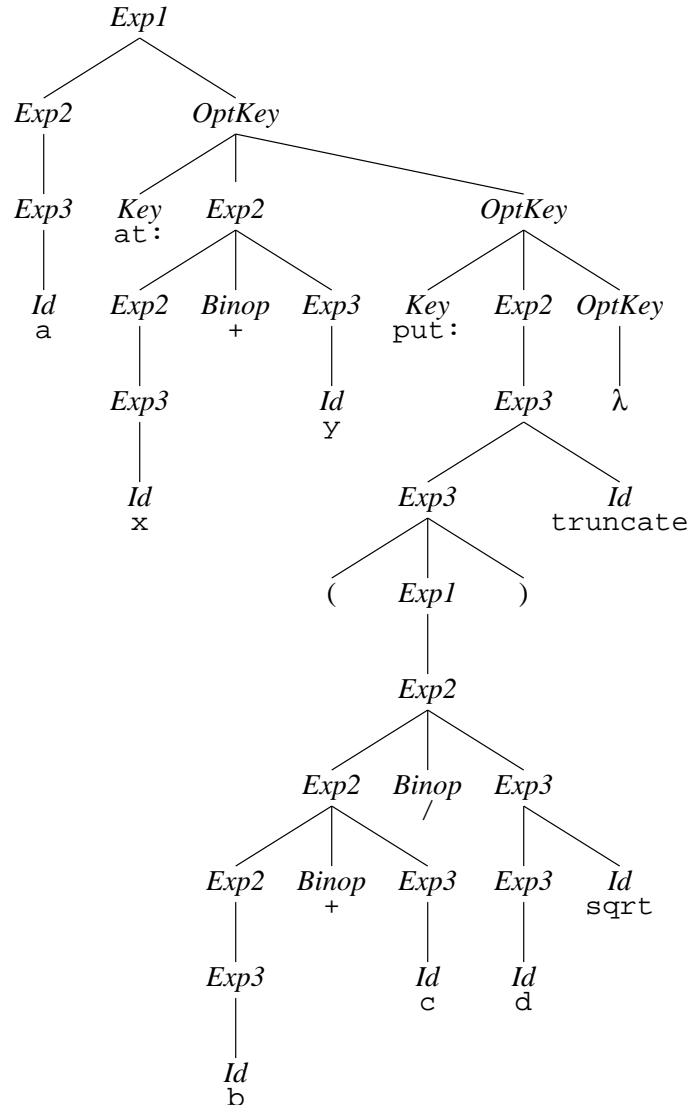
- (a) (10%) Give a grammar for Smalltalk expressions, capturing the precedence and associativity rules described above.

Answer:

$$\begin{aligned} \textit{Exp1} &\rightarrow \textit{Exp2} \textit{OptKey} \\ \textit{OptKey} &\rightarrow \epsilon \mid \textit{Key} \textit{Exp2} \textit{OptKey} \\ \textit{Exp2} &\rightarrow \textit{Exp2} \textit{BinOp} \textit{Exp3} \mid \textit{Exp3} \\ \textit{Exp3} &\rightarrow \textit{Exp3} \textit{Id} \mid \textit{Id} \mid \textit{Int} \mid (\textit{Exp1}) \end{aligned}$$

- (b) (5%) Give the (non-abstract) parse tree produced by your grammar for the expression:
a at: x+y put: (b+c / d sqrt) truncate

Answer:



- (c) (5%) How difficult would it be to devise an LL grammar for this (sub)language? If an LL grammar is possible, then how convenient would it be for use in a compiler? Answer the same questions with LL replaced by LR. (Be concise!)

Answer:

It would not be too hard to devise an LL grammar for this language, but it would not be a convenient form for parsing according to the precedence rules described above, in that the parse tree would not directly reflect the computational structure. To see this, consider `Id Id Id` and similar forms. We really want a left-recursive rule to build the most convenient tree, but we must use right-recursion in an LL grammar. An LR grammar would be easy to build and quite convenient. In fact, the grammar above is probably LR.

2. (Live Variable Analysis; 20%)

Consider the following WHILE program:

$[y:=x]^1; [z:=1]^2; \text{while } [y>1]^3 \text{ do } ([z:=z*y]^4; [y:=y-1]^5); [y:=0]^6$

Perform a live variable analysis for this program using the equations:

$$\begin{aligned} \text{kill}_{\text{LV}}([x := a]^\ell) &= \{x\} \\ \text{kill}_{\text{LV}}([\text{skip}]^\ell) &= \emptyset \\ \text{kill}_{\text{LV}}([b]^\ell) &= \emptyset \end{aligned}$$

$$\begin{aligned} \text{gen}_{\text{LV}}([x := a]^\ell) &= \text{FV}(a) \\ \text{gen}_{\text{LV}}([\text{skip}]^\ell) &= \emptyset \\ \text{gen}_{\text{LV}}([b]^\ell) &= \text{FV}(b) \end{aligned}$$

$$\text{LV}_{\text{exit}}(\ell) = \begin{cases} \emptyset & \text{if } \ell \in \text{final}(S_\star) \\ \bigcup \{\text{LV}_{\text{entry}}(\ell') \mid (\ell', \ell) \in \text{flow}^R(S_\star)\} & \text{otherwise} \end{cases}$$

$$\text{LV}_{\text{entry}}(\ell) = (\text{LV}_{\text{exit}}(\ell) \setminus \text{kill}_{\text{LV}}(B^\ell)) \cup \text{gen}_{\text{LV}}(B^\ell)$$

where $B^l \in \text{blocks}(S_\star)$

Answer:

ℓ	$\text{kill}_{\text{LV}}(\ell)$	$\text{gen}_{\text{LV}}(\ell)$
1	$\{y\}$	$\{x\}$
2	$\{z\}$	\emptyset
3	\emptyset	$\{y\}$
4	$\{z\}$	$\{y, z\}$
5	$\{y\}$	$\{y\}$
6	$\{y\}$	\emptyset

$$\begin{aligned} \text{LV}_{\text{entry}}(1) &= \text{LV}_{\text{exit}}(1) \setminus \{y\} \cup \{x\} & \text{LV}_{\text{exit}}(1) &= \text{LV}_{\text{entry}}(2) \\ \text{LV}_{\text{entry}}(2) &= \text{LV}_{\text{exit}}(2) \setminus \{z\} & \text{LV}_{\text{exit}}(2) &= \text{LV}_{\text{entry}}(3) \\ \text{LV}_{\text{entry}}(3) &= \text{LV}_{\text{exit}}(3) \cup \{y\} & \text{LV}_{\text{exit}}(3) &= \text{LV}_{\text{entry}}(4) \cup \text{LV}_{\text{entry}}(6) \\ \text{LV}_{\text{entry}}(4) &= \text{LV}_{\text{exit}}(4) \setminus \{z\} \cup \{y, z\} & \text{LV}_{\text{exit}}(4) &= \text{LV}_{\text{entry}}(5) \\ \text{LV}_{\text{entry}}(5) &= \text{LV}_{\text{exit}}(5) \setminus \{y\} \cup \{y\} & \text{LV}_{\text{exit}}(5) &= \text{LV}_{\text{entry}}(3) \\ \text{LV}_{\text{entry}}(6) &= \text{LV}_{\text{exit}}(6) \setminus \{y\} & \text{LV}_{\text{exit}}(6) &= \emptyset \end{aligned}$$

ℓ	$\text{LV}_{\text{entry}}(\ell)$	$\text{LV}_{\text{exit}}(\ell)$
1	$\{x\}$	$\{y\}$
2	$\{y\}$	$\{y, z\}$
3	$\{y, z\}$	$\{y, z\}$
4	$\{y, z\}$	$\{y, z\}$
5	$\{y, z\}$	$\{y, z\}$
6	\emptyset	\emptyset

3. (Dead Variable Analysis; 20%)

Recall the WHILE language:

$$\begin{aligned}
 a & ::= x \mid n \mid a_1 \text{ op}_a a_2 \\
 b & ::= \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \text{ op}_b b_2 \mid a_1 \text{ op}_r a_2 \\
 S & ::= [x := a]^\ell \mid [\text{skip}]^\ell \mid S_1; S_2 \mid \text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2 \mid \text{while } [b]^\ell \text{ do } S \mid
 \end{aligned}$$

Devise a *dead variable* analysis DV for the WHILE language. A variable is *dead* at the exit from a label if there is no path from the label to any use of the variable on which path the variable is not re-defined. The analysis DV will determine:

For each program point, which variables *must* be dead at the exit from the point.

Give both $kill_{DV}$ and gen_{DV} functions as well as data flow equations $DV^\#$.

Answer:

$$\begin{aligned}
 kill_{DV}([x := a]^\ell) &= FV(a) \\
 kill_{DV}([\text{skip}]^\ell) &= \emptyset \\
 kill_{DV}([b]^\ell) &= FV(b)
 \end{aligned}$$

$$\begin{aligned}
 gen_{DV}([x := a]^\ell) &= \{x\} \\
 gen_{DV}([\text{skip}]^\ell) &= \emptyset \\
 gen_{DV}([b]^\ell) &= \emptyset
 \end{aligned}$$

$$DV_{exit}(\ell) = \begin{cases} \text{Var}_\star & \text{if } \ell \in \text{final}(S_\star) \\ \bigcap \{DV_{entry}(\ell') \mid (\ell', \ell) \in \text{flow}^R(S_\star)\} & \text{otherwise} \end{cases}$$

$$DV_{entry}(\ell) = (DV_{exit}(\ell) \setminus kill_{DV}(B^\ell)) \cup gen_{DV}(B^\ell)$$

where $B^\ell \in \text{blocks}(S_\star)$

4. (Uninitialized Variable Analysis; 30%)

Consider an extension of the WHILE language that allows us to create (`malloc`) and destroy (`free`) cells in the heap. The data stored in a cell is accessed by dereferencing the pointer obtained when the cell is created. When a cell is destroyed its pointer becomes unusable. *Pointer dereference* is written with syntax `*x` similar to that of C. The extended syntax of the WHILE language is as follows:

$$\begin{aligned}
 a & ::= x \mid n \mid a_1 \text{ op}_a a_2 \mid \text{nil} \mid *x \\
 b & ::= \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \text{ op}_b b_2 \mid a_1 \text{ op}_r a_2 \\
 S & ::= [x := a]^\ell \mid [\text{skip}]^\ell \mid S_1; S_2 \mid \text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2 \mid \text{while } [b]^\ell \text{ do } S \mid \\
 & \quad [*x := a]^\ell \mid [\text{malloc } x]^\ell \mid [\text{free } x]^\ell
 \end{aligned}$$

Arithmetic expressions are extended to permit pointer dereference `*x` rather than just variables, and an arithmetic expression can also be the constant `nil`. The binary operators op_a are as before: that is, they are the standard arithmetic operations and in particular they do *not* allow pointer arithmetic. The boolean expressions are extended such that the relational operators op_r now allow testing for *equality of pointers*. Note that both arithmetic and boolean expressions can only *access* cells in the heap; they can neither create new cells nor update existing cells.

Assignment now also includes `*x := a` to perform a destructive update of the heap location pointed to by `x`. The extended language also contains a statement `malloc x` for creating a new cell and setting `x` to point to it, and a statement `free x` for destroying the cell pointed to by `x` and setting `x` to `nil`.

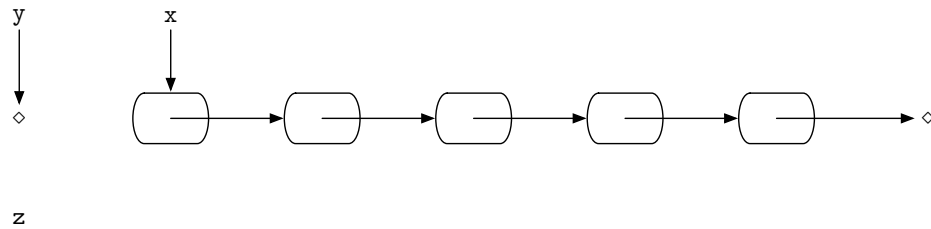
(a) Consider the following program:

```

[y:=nil]1;
while [x!=nil]2 do
  ([z:=y]3; [y:=x]4; [x:=*x]5; [*y:=z]6)
[z:=nil]7

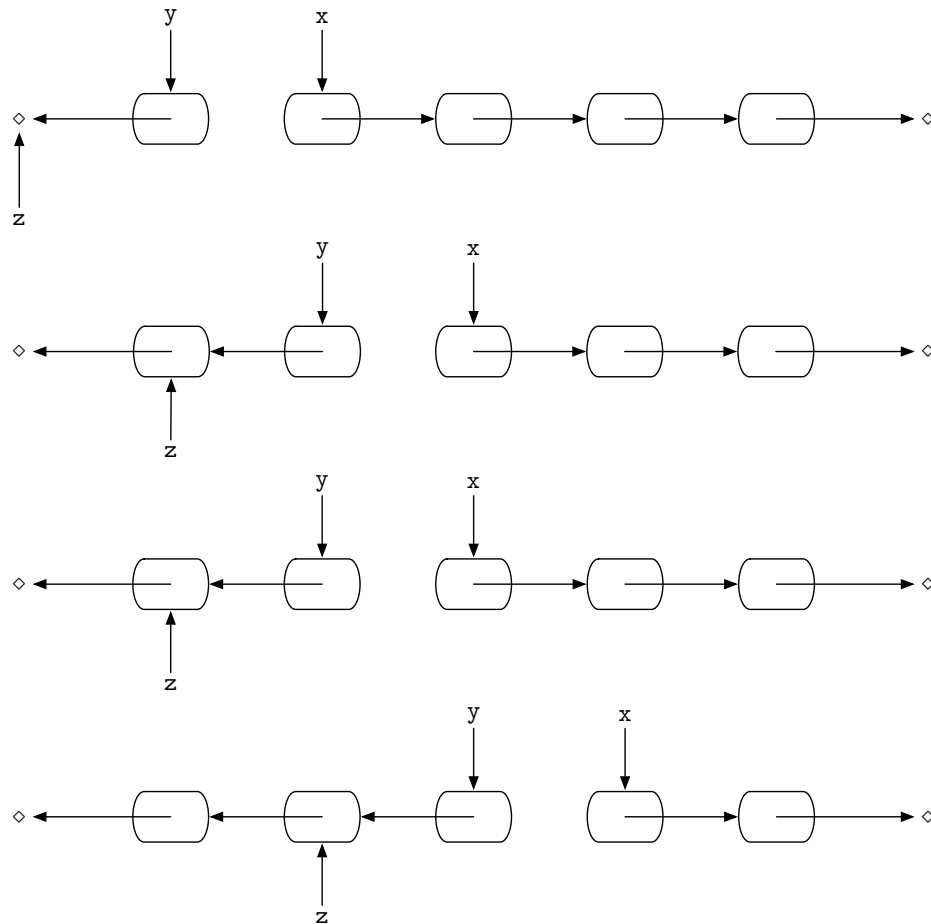
```

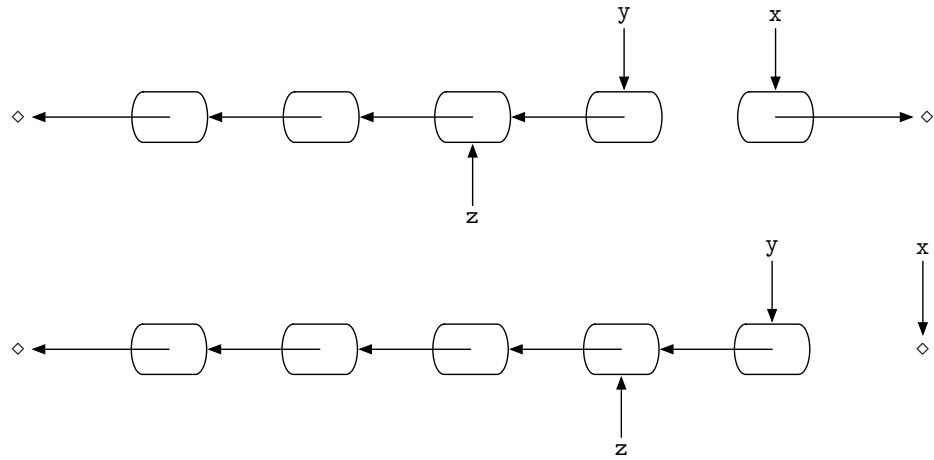
i. (10%) Suppose x initially points to a 5-element list and y and z are initially undefined. After statement 1 has executed, and right before entering the while-loop, x points to the list, y is nil (denoted by $\rightarrow \diamond$), and z is undefined, as illustrated by the following diagram of the state of the heap:



Show the state of the heap after each subsequent iteration of the body of the loop:

Answer:





ii. (5%) What does this program actually do?

Answer:

It reverses the list pointed to by x and leaves the result in y.

- (b) (15%) Devise a data flow analysis for the pointer-extended WHILE language to discover *Possibly Uninitialized Variables*. The analysis will determine:

For each program point, which variables *may* be uninitialized.

Variables are uninitialized at the start of the program, and revert to uninitialized when their value *may* be set to `nil`; assume that any assignment of a non-numeric or non-arithmetic expression to a variable may set it to `nil`. Moreover, assume that every dereference checks whether the pointer is `nil`, terminating the program immediately if it is. Thus, pointer dereference establishes that the dereferenced variable must not be `nil`.

- i. (10%) First, define *gen* and *kill* functions for *Possibly Uninitialized Variables*.

Answer:

$$\begin{aligned}
 \textit{kill}([x := a]^\ell) &= \{x\} \cup \{x' \mid *x' \in \mathbf{AExp}(a)\} \\
 \textit{kill}([\textit{skip}]^\ell) &= \emptyset \\
 \textit{kill}([b]^\ell) &= \{x' \mid *x' \in \mathbf{AExp}(b)\} \\
 \textit{kill}[*x := a]^\ell &= \{x\} \cup \{x' \mid *x' \in \mathbf{AExp}(a)\} \\
 \textit{kill}([\textit{malloc } x]^\ell) &= \{x\} \\
 \textit{kill}([\textit{free } x]^\ell) &= \emptyset
 \end{aligned}$$

$$\begin{aligned}
 \textit{gen}([x := a]^\ell) &= \begin{cases} \emptyset & \text{if } a \text{ is of the form } n \text{ or } a_1 \textit{ op}_a a_2 \\ \{x\} & \text{otherwise} \end{cases} \\
 \textit{gen}([\textit{skip}]^\ell) &= \emptyset \\
 \textit{gen}([b]^\ell) &= \emptyset \\
 \textit{gen}[*x := a]^\ell &= \emptyset \\
 \textit{gen}([\textit{malloc } x]^\ell) &= \emptyset \\
 \textit{gen}([\textit{free } x]^\ell) &= \{x\}
 \end{aligned}$$

- ii. (5%) Consider the data flow analyses discussed in class. Each of these is an instance of a *Monotone Framework*, having a property space L (a complete lattice in all cases), a partial ordering \sqsubseteq , a least element \perp , a least upper bound operation \sqcup , a flow F , extremal labels E , extremal (i.e., boundary) values ι , and a set of transfer functions \mathcal{F} defined as functions f_ℓ on the labels of the program S_\star being analyzed, with *kill* and *gen* as defined for each analysis.

	Available Expressions	Reaching Definitions	Very Busy Expressions	Live Variables
L	$\mathcal{P}(\mathbf{AExp}_\star)$	$\mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star^?)$	$\mathcal{P}(\mathbf{AExp}_\star)$	$\mathcal{P}(\mathbf{Var}_\star)$
\sqsubseteq	\supseteq	\subseteq	\supseteq	\subseteq
\sqcup	\cap	\cup	\cap	\cup
\perp	\mathbf{AExp}_\star	\emptyset	\mathbf{AExp}_\star	\emptyset
ι	\emptyset	$\{(x, ?) \mid x \in \mathbf{Var}_\star\}$	\emptyset	\emptyset
E	$\{init(S_\star)\}$	$\{init(S_\star)\}$	$final(S_\star)$	$final(S_\star)$
F	$flow(S_\star)$	$flow(S_\star)$	$flow^R(S_\star)$	$flow^R(S_\star)$
\mathcal{F}	$\{f : L \rightarrow L \mid \exists l_k, l_g : f(l) = (l \setminus l_k) \cup l_g\}$			
f_ℓ	$f_\ell(l) = (l \setminus kill([B]^\ell)) \cup gen([B]^\ell)$ where $[B]^\ell \in blocks(S_\star)$			

Identify your analysis as a Monotone Framework by defining each of the elements $L, \sqsubseteq, \sqcup, \perp, \iota, E, F$.

Answer:

L	$\mathcal{P}(\mathbf{Var}_\star)$
\sqsubseteq	\subseteq
\sqcup	\cup
\perp	\emptyset
ι	\mathbf{Var}_\star
E	$\{init(S_\star)\}$
F	$flow(S_\star)$

5. (Refinements; 10%) The formulation of Monotone Frameworks associates transfer functions with basic blocks. In a statement of the form

$$\text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2$$

this prevents us from using the result of the test to pass different information to S_1 and S_2 ; as an example suppose that b is $x \neq \text{nil}$ in an *Uninitialized Variable Analysis*. To remedy this deficiency consider writing $[b]^\ell$ as $[b]^{\ell_1, \ell_2}$ where ℓ_1 corresponds to b evaluating to true and ℓ_2 corresponds to b evaluating to false.

- (a) (5%) What benefit might we gain from capturing flow information that is sensitive to the success or failure of a branch when performing an analysis (e.g., consider *Uninitialized Variable Analysis*)?

Answer:

- (b) (5%) Sketch how to extend data flow analysis to make use of this extension.

Answer: