University of Toronto
CSC 488S Compilers and Interpreters **Winter 2007/2008**

Mid Term Test Solution

1. [20 marks] A regular expression definition for numeric constants is given below.

Digit	(0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)
OptionalSign	(’+’ | ’-’ | λ)
Number	Integer | RealNumber
Integer	Digits$^+$
RealNumber	Digit* (’.’ | λ) Digit* ((E | e) OptionalSign Digit$^+$ | λ)

Assume you are working with a very simple scanner that only returns single character lexical tokens.

Write a LL(1) grammar for parsing Number. Your grammar should enforce the requirement that the mantissa part of a real number must contain at least one digit. Prove that your grammar is LL(1) by showing the director sets for each rule. You may assume the set $follow(Number)$, but describe what it can not contain.

Solution

Digit:	’0’ , ’1’ , ’2’ , ’3’ ,	{ 0 } , { 1 } , { 2 } , { 3 }
	’4’ , ’5’ , ’6’ , ’7’ ,	{ 4 } , { 5 } , { 6 } , { 7 }
	’8’ , ’9’	{ 8 } , { 9 }
Number:	Integer realTail ,	{ 0 1 2 3 4 5 6 7 8 9 }
	’.’ realTail2	{ . }
Integer:	Digit integerTail	{ 0 1 2 3 4 5 6 7 8 9 }
integerTail:	Digit integerTail	{ 0 1 2 3 4 5 6 7 8 9 }
	λ	{ . E e } ∪ $follow(Number)$
realTail:	optDot integerTail optExponent	{ . 0 1 2 3 4 5 6 7 8 9 E e } ∪$follow(Number)$
realTail2:	Integer optExponent	{ 0 1 2 3 4 5 6 7 8 9 }
optExponent	’E’ optSign Integer	{ E }
	’e’ optSign Integer	{ e }
	λ	$follow(Number)$
optSign	’+’	{ + }
	’-’	{ - }
	λ	{ 0 1 2 3 4 5 6 7 8 9 }

For this grammar to be LL(1) the set $follow(Number)$ cannot contain { . E e 0 1 2 3 4 5 6 7 8 9 } .

2. [20 marks] A proposal has been made to slightly simplify the project language by removing the **result** reserved word and replacing it with **return**. The two return-type statements would then be:

return

return *expression*

a)[5 marks] Does this change have any significant effect on language?

b)[15 marks] How will this change affect parsing the language? Explain your answer.

a) The change means that *statement* now ends with an *optional expression*. Having something optional at the end of statement has major consequences for a language that does not use separators between statements (e.g. a ; in C)

Note that in the existing language, semantic analysis already has to check that the right kind of return/results statement is being used in a procedure/function.

b) The change creates a unresolvable syntactic ambiguity (shift/reduce conflict) in parsing the optional expression at the end of statement. The problem is made worse because assignment and equality test have the same form. Consider the two code fragments:

```
        1      if ( A = B ) then
        2           return   %  in a procedure
        3      B = C
and
        6      return  B = C % in a boolean function
```

There is no syntactic way to decide that the return on line 2 doesn't have the optional expression, lines 2/3 and 6 are identical syntactically.

There are several ways that this shift/reduce conflict could be solved, but lookahead (suggested by some students) is not one of them.
- don't change **result** to **return**
- change the form for functions to **return with** *expression*
  the **with** removes the conflict
- use *three* forms of statement list in the grammar
  - one containing all statements except **return**
  - one containing only **return** for use inside procedures
  - one containing only **return** *expression* for use in functions

This would result in a really ugly grammar, with a lot of duplicate, hard to maintain actions in the `cup` file.

3. **[10 marks]** In some languages new variable *names* can be *dynamically* created at run time. For example build up a string by concatenation and then use the value of the string as the name of a new variable. This feature is heavily used to create map-like associative tables. Newly created variables have a null/zero value until assigned to.

Describe a set of runtime mechanisms that will support the creation and use of dynamically created variables.

- All run time variable access should be handled by some form of run time table. A hash table or map would be a good choice.
- Code is generated for every *use* of a variable to look it up in the table. It it is found that the table entry is used. If it is not found it is a new variable and a new table entry is allocated and initialized.
- Note that in this style of language there is no concept of duplicate declaration or undeclared identifier. If I use a previously unused variable in an expression, it gets created, and I get its default null/zero value. If I use a previously unused variable on the left side of an assignment, the variable gets created and then the value is assigned.

4. [20 marks] One problem that a compiler has to deal with initialization of variables in their declaration. For example:

```
 #define BIGDATASIZE   <some large integer constant>
 typedef struct {
         int  count ;
         char *  name ;
         float   value ;
 }  dataStruct ;
 dataStruct bigData[ BIGDATASIZE ] =
     {{ 3 , "scorp" , 4.7 } , { 6 , "podger" , 9.2 } ,
      { 12, "framus" , 3.8 } , { 19 , "spanner", 7.3 } .
      /*  Assume many more values here */
}
```

a)[10 marks] Describe the semantic analysis checking required for this type of declaration.

– Check that anything named dataStruct hasn't already been declared in the current scope.
– Check that the field names in dataStruct are distinct i.e. no two fields with the same name
– In the declaration of bigData, check that dataStruct is a type
– Check that anything named bigData hasn't already been declared in the current scope.
– Check that BIGSIZE is positive and less than any implementation limits.
– Check that each initializing value contains a value for each of the three fields in dataStruct
– Check that all of the initializing values are constants. (some languages might allow expressions)
– Check the type of the constants in each initializing value for compatibility with int, char * and float respectively
– Check that the number of initializing values supplied is equal to BIGDATASIZE.
(Some languages may permit partial initialization)

b)[10 marks] Describe a *complete* scheme for the general case of handling storage and access to the initial value for arrays and structs with large initial values. Describe how the values are stored and how they are made available to later passes of the compiler. How hard would it be to implement this scheme in a strongly typed language like Java?

- The compiler would need a buffer somewhere to hold large initializing values. The buffer could be in memory or a disk file. If the compiler is one-pass if could copy the constant values to some place in its output stream.

- The variable being initializes would have to be laid out in memory (e.g. map the fields of a structure). In the best case, the compiler would build in its buffer a memory image of the initialized data structure. At some point this memory image would be emitted to the the compile output as a block of initialized data.

- Code would be generated to do a runtime copy of the initializing block to the variable at the point where the declaration occurs. Best case a block memory copy could be used. Some languages might require field by field assignment.

If the compiler was written in a strongly typed language like Java getting values into and out of the buffer would be a nuisance. Either make the buffer an array of data descriptors which can contain any type of constant but might waste space, or make the buffer an array of bytes and do a lot of type casting to move things into and out of the buffer. Some students suggested storing the values as a linked list of Objects in Java. This would work but it wouldn't be very space or time efficient.

5. [15 marks] Given the C structure declaration:

```
struct L1 {
       char key ;
       double value ;
       short  kind  ;
       union {
            struct L1 * self ;
            double value2 ;
            struct {
                  char key2 ;
                  double value3 ;
            } bigVal ;
            short sData[ 9 ] ;
       } bigU ;
       struct L1 * next ;
}
```

Show how this structure would be laid out in memory using the depth first structure mapping algorithm discussed in lecture. You should assume the size and alignment factors (in bits) in the table below.

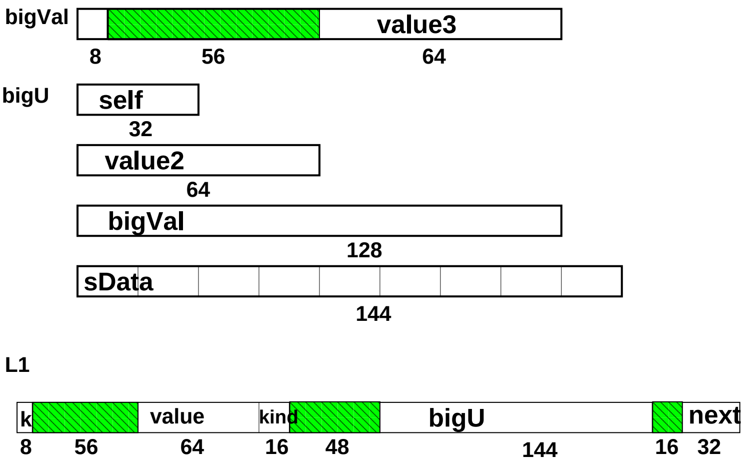
Type	Size	Alignment	Type	Size	Alignment	Type	Size	Alignment
char	8	8	int	32	32	float	32	32
short	16	16	pointer	32	32	double	64	64

Map from the inside out.

bigVal has total length 128, alignment 64. There are 56 bits of fill between key2 and value2.

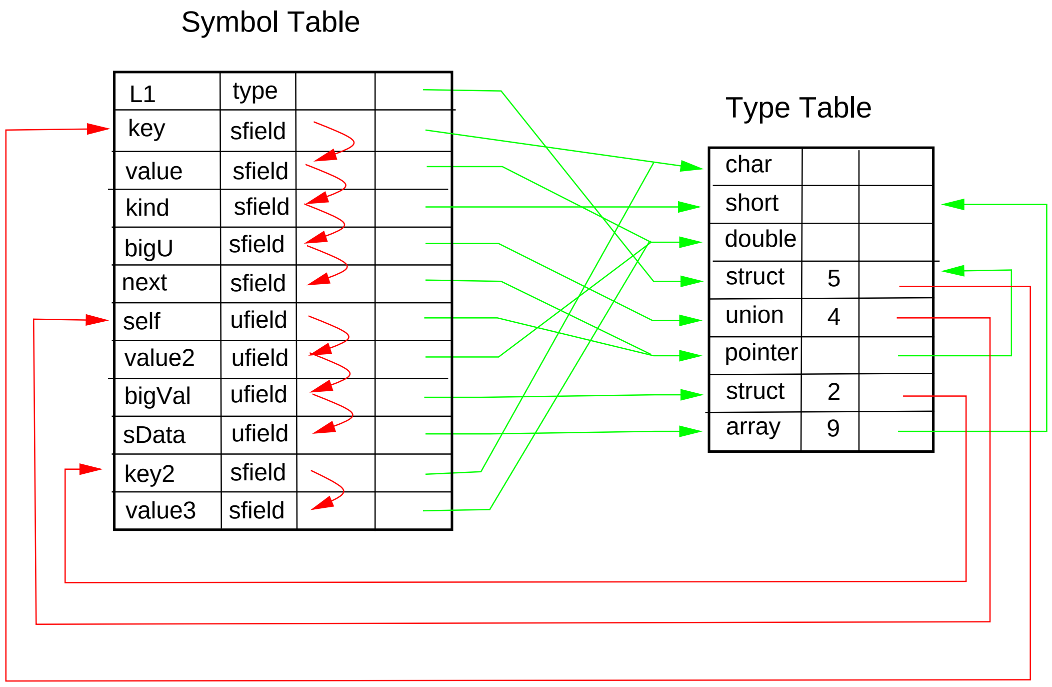
All of the fields in the bigU union overlay each other. *bigU* has total length 144 and alignment 64. The length is the length of the largest alternative sData and the alignment is max of the alignment for any field. There is no fill in bigU

L1 had total length 384 and alignment 64. The fill in this structure is: 56 bits in front of value, 48 bit in front of bigU, 16 bits in front of next.



6. [15 marks] Show the symbol and type table entries that a typical compiler would make for the structure declaration in Question 5.

One possible solution:



Symbol Table

L1	type		
key	sfield		
value	sfield		
kind	sfield		
bigU	sfield		
next	sfield		
self	ufield		
value2	ufield		
bigVal	ufield		
sData	ufield		
key2	sfield		
value3	sfield		

Type Table

char		
short		
double		
struct	5	
union	4	
pointer		
struct	2	
array	9	