CS235 Languages and Automata **Handout # 10**
Prof. Lyn Turbak **October 31, 2007**
Wellesley College

Kitty Reference Manual

1 Overview

Kitty is a simple subset of the Tiger language described in Appendix A of Appel's textbook.[1] Like Tiger, Kitty provides various operations for manipulating integers, simple input/output operations, scoped mutable variables, conditionals, and loops. But Kitty does not support other Tiger features like functions, compound data structures (arrays and records), and explicit types.

In CS235 this semester, we will use Kitty as a language for exploring lexical analysis and parsing.

2 Syntax

2.1 Lexical Conventions

Lexical conventions specify the structure of identifiers, literals and comments. The lexical conventions for are similar to those for Tiger, except for the conventions for character literals (which are not present in Tiger). For completeness, here we augment the lexical conventions in Section A.1 of Appel's book with the changes and clarifications needed for Kitty.

1. An identifier is a maximal sequence of letters, digits, and underscores, starting with a letter. Uppercase and lowercase letters are distinguished. An identifier must be distinct from the reserved keywords words of the language.

2. An integer literal is a maximal sequence of digits. All integer literals are positive integers; the unary minus sign before a negative literal is a distinct token.

3. A character literal is a pair of single quotes delimiting a single character or escape sequence. `'A'` and `'a'` are examples of single-character character literals. The escape sequences are as follows:

   - `\n` (newline)
   - `\t` (tab)
   - `\'` (single quote)
   - `\\` (backslash)
   - `\^`c, where c ranges over the 26 capital letters and the seven punctuation marks `@`, `[`, `]`, `\`, `^`, `_`, and `?`. These are so-called **control characters**.
   - `\`ddd where d ranges over the decimal digits. This denotes the character whose ASCII value is the specified three digit decimal number, which must be in the range [0 .. 255].

4. A string literal is a pair of double quotes delimiting any sequence of characters or escape sequences. The escape sequences within a string literal are the same as those for character literals except:

   - `\'` is not an escape character within a string literal.

[1] Actually, Kitty is not technically a subset of Tiger because it has a few features that Tiger does not have. In particular: (1) Kitty supports character literals that are distinct from strings; (2) Kitty has constants `true`, `false`, `minint`, and `maxint` that are not defined in Tiger; and (3) Kitty provides several operators not provided by Tiger: `readc`, `readi`, `writec`, `writei`, and `writes`. Nevertheless, in all other respects but these, Kitty is a subset of Tiger.

- \¨ is an escape character within a string literal (but not within a character literal).

Unlike Tiger, Kitty allows string literals that extend over multiple lines without requiring the use of backslashes at the beginning and end of every line. (Compare to Section A.3, p. 517 of Appel's book.)

5. Whitespace (i.e., space, tabs, and newlines) appearing between any two tokens is ignored. Whitespace is optional as long as there is no ambiguity in token structure without it.

6. A comment may appear between any two tokens. Comments start with `/*` and end with `*/` and may be nested.

2.2 Grammar

A syntactically well-formed Kitty program is an expression derived from the variable *Exp* in the grammar in Fig. 1. Italicized annotations beginning with # are comments and not part of the grammar.

The grammar in Fig. 1 is ambiguous. The ambiguities are resolved by the following rules:

Precedence: The precedence of operators from highest to lowest is as follows (operators on the same line have the same precedence):

unary minus (negation)
`*`, `/`
`+`, `-`
`<`, `<=`, `=`, `<>`, `>=`, `>`
`&`
`|`

Associativity: The operators `*`, `/`, `+`, `-`, `&`, and `|` are all left-associative. E.g., `1 - 2 + 3` is parsed as if it were written `(1 - 2) + 3`. The relational `<`, `<=`, `=`, `<>`, `>=`, and `>` are all non-associative. E.g., `1 < 2 = 3` is not a legal expression, even though the explicitly grouped versions `(1 < 2) = 3` and `1 < (2 = 3)` are legal expressions.

Dangling Else: The presence of both `if-then` and `if-then-else` expressions in a language introduces an ambiguity as to which `if` expression an `else` clause belongs. The Kitty convention (as in many other languages) is that an `else` clause belongs to the innermost if expression enclosing it. Thus, the expression

`if` E_1 `then if` E_2 `then` E_3 `else` E_4

is parsed as if it were written

`if` E_1 `then (if` E_2 `then` E_3 `else` E_4`)`

Weak Precedence of Assignments and Keyword Expressions: It is assumed that assignments and expressions beginning with the keywords `if`, `while`, and `for` bind less tightly than binary operator expressions. In other words, the final expression in an assignment or a `if`/`while`/`for` expression should be interpreted as extending as far right as possible. For example:

Expression	**parses as**	**and not as**
`x :=` E_1 `+` E_2	`x :=` (E_1 `+` E_2)	(`x :=` E_1) `+` E_2
`if` E_1 `then` E_2 `+` E_3	`if` E_1 `then` (E_2 `+` E_3)	(`if` E_1 `then` E_2) `+` E_3
`if` E_1 `then` E_3 `else` E_3 `+` E_4	`if` E_1 `then` E_3 `else` (E_3 `+` E_4)	(`if` E_1 `then` E_3 `else` E_3) `+` E_4
`while` E_1 `do x :=` E_2 `+` E_3	`while` E_1 `do x :=` (E_2 `+` E_3)	(`while` E_1 `do x :=` E_2) `+` E_3
`for i :=` E_1 `to` E_2 `do` `x :=` E_3 `+` E_4	`for i :=` E_1 `to` E_2 `do` `x :=` (E_3 `+` E_4)	(`for i :=` E_1 `to` E_2 `do` `x :=` E_3) `+` E_4

```
Exp ::= # Kitty expressions
      IntLit | CharLit | Ident | Const
    | ()                    # The literal for "no value"
    | Nullop()              # Parentheses are required
    | Unop (Exp)            # Parentheses are required
    | -Exp                  # Unary negation
    | writes(StringLit)     # Parentheses are required
    | Exp Binop Exp
    | Ident := Exp          # Assignment
    | if Exp then Exp else Exp
    | if Exp then Exp
    | let DecSeq in ExpSeq0 end
    | while Exp do Exp
    | for Ident := Exp to Exp do Exp
    | (ExpSeq2)             # Sequence expression; parenthses required
    | (Exp)                 # Grouping via optional parentheses

ExpSeq0 ::= # Expression sequences with 0 or more expressions
              # Empty expression sequence
    | ExpSeq1  # Nonempty expression sequence

ExpSeq1 ::= # Expression sequences with 1 or more expressions
    Exp | Exp, ExpSeq1

ExpSeq2 ::= # Expression sequences with 2 or more expressions
    Exp, ExpSeq1

Dec ::= var Ident := Exp  # Variable declarations

DecSeq ::= # Declaration sequences with 1 or more declarations
    Dec | Dec; DecSeq

Const ::= true | false | minint | maxint | true # Constants

Nullop ::= readc # Nullary (zero-argument) operators

Unop ::= not | readi | writec | writei # Unary (one-argument) operators

Binop ::= # Binary (two-argument) operators
      + | - | *
    | / # Integer division
    | % # Integer remainder
    | < | <= | = | <> | >= | > # Relational binops
    | & # Short-circuit and
    | | # Short-circuit or

IntLit ::= # as specified by the lexical conventions for integer literals

CharLit ::= # as specified by the lexical conventions for character literals

Ident ::= # as specified by the lexical conventions for identifiers

StringLit ::= # as specified by the lexical conventions for string literals
```

Figure 1: A context-free grammar for Kitty.

2.3 Desugaring

Several of the Kitty expressions in Fig. 1 can be viewed as desugaring into other expressions, as illustrated in the following table:

Source language expression	**Expression after desugaring**
`-`E_1	`(0 -` E_1`)`
`if` E_1 `then` E_2	`if` E_1 `then` E_2 `else ()`
E_1 `&` E_2	`if` E_1 `= 0 then 0 else` E_2

E_1 `|` E_2	`let var` I_1 `:=` E_1 `/* Assume` I_1 `is fresh */`
	` in if` I_1 `= 0 then` E_2 `else` I_1
	`end`

`writes("abc...")`	`(writec('a'); writec('b'); writec('c'); ...)`

`for` I `:=` E_1 `to` E_2 `in` E_3	`let var` I_1 `:=` E_1`;`
	`  var` I_2 `:=` E_2 `/* Assume` I_2 `is fresh */`
	` in while (`I_1 `<=` I_2`) do`
	`  (`E_3`;` I_1 `:=` I_1 `+ 1)`
	`end`

The desugarings for Kitty's logical operators (`&` and `|`) give them short-circuit semantics. I.e.:

- E_1 `&` E_2 evaluates to 0 if E_1 evaluates to 0; in this case E_2 is not evaluated.
- E_1 `&` E_2 evaluates to E_2 if E_1 evaluates to a non-zero integer.
- E_1 `|` E_2 evaluates to E_1 if E_1 evaluates to a non-zero integer; in this case E_2 is not evaluated.
- E_1 `|` E_2 evaluates to E_2 if E_1 evaluates to 0.

3 Semantics

The meaning of Kitty expressions is specified informally by English descriptions. All the constructs taken from Tiger have the same meaning as specified in Section A.3 of Appel's book. We do not repeat those descriptions here. Instead, we (1) describe the meanings of the few Kitty constructs that are not taken from Tiger and (2) highlight a few subtleties of the constructs taken from Tiger.

3.1 Valued and Valueless Expressions

Every Kitty expression either denotes an integer value or it denotes no value. We call the former **valued** expressions and the latter **valueless**. The valueless expressions are:

- `()`, the "no value" literal
- assignment expressions
- applications of `writec`, `writei`, and `writes`
- `if-then` expressions
- `if-then-else` expressions both of whose branches are valueless
- `while` expressions
- `for` expressions

- `let` expressions whose bodies are valueless
- sequence expressions whose last expressions are valueless

All other valid expressions are valued. Note that an expression is only valid if all contexts that expect an integer value are supplied with one. It is an error to have a valueless expression in a context where a valued expression is required. It is also an error for the two branches of an `if-then-else` expression to differ in their "valueness". Such errors can be either be reported statically (before the evaluation process) or dynamically (when they are encountered during the evaluation process). In general, static checking will catch some errors that cannot be detected dynamically.

3.2 Character Literals

A character literal denotes an integer in the range [0 .. 255] that is its associated ASCII value. Here is a table showing the relationship between integers and character literals for those characters with ASCII values in the range [0 .. 127]:

```
0:'\^@'  1:'\^A'  2:'\^B'  3:'\^C'  4:'\^D'  5:'\^E'  6:'\^F'  7:'\^G'
8:'\^H'  9:'\t'   10:'\n'  11:'\^K' 12:'\^L' 13:'\^M' 14:'\^N' 15:'\^O'
16:'\^P' 17:'\^Q' 18:'\^R' 19:'\^S' 20:'\^T' 21:'\^U' 22:'\^V' 23:'\^W'
24:'\^X' 25:'\^Y' 26:'\^Z' 27:'\^[' 28:'\^\' 29:'\^]' 30:'\^^' 31:'\^_'
32:' '   33:'!'   34:'"'   35:'#'   36:'\$'  37:'%'   38:'&'   39:'\''
40:'('   41:')'   42:'*'   43:'+'   44:','   45:'-'   46:'.'   47:'/'
48:'0'   49:'1'   50:'2'   51:'3'   52:'4'   53:'5'   54:'6'   55:'7'
56:'8'   57:'9'   58:':'   59:';'   60:'<'   61:'='   62:'>'   63:'?'
64:'@'   65:'A'   66:'B'   67:'C'   68:'D'   69:'E'   70:'F'   71:'G'
72:'H'   73:'I'   74:'J'   75:'K'   76:'L'   77:'M'   78:'N'   79:'O'
80:'P'   81:'Q'   82:'R'   83:'S'   84:'T'   85:'U'   86:'V'   87:'W'
88:'X'   89:'Y'   90:'Z'   91:'['   92:'\\'  93:']'   94:'^'   95:'_'
96:'`'   97:'a'   98:'b'   99:'c'   100:'d'  101:'e'  102:'f'  103:'g'
104:'h'  105:'i'  106:'j'  107:'k'  108:'l'  109:'m'  110:'n'  111:'o'
112:'p'  113:'q'  114:'r'  115:'s'  116:'t'  117:'u'  118:'v'  119:'w'
120:'x'  121:'y'  122:'z'  123:'{'  124:'|'  125:'}'  126:'~'  127:'\^?'
```

In general, there may be several character literals corresponding to the same value. For instance, `'\t'`, `'\^I'`, and `'\009'` are all character literals denoting the integer 9.

Because Kitty character literals stand for integers, they can be used in any context where an integer is expected. For instance, the Kitty expression `'A' - 'a'` denotes the integer 32 (the result of $97 - 65$).

3.3 Constants

Like character literals, the constants `true`, `false`, `minint`, and `maxint` denote particular integers:

`true`	1
`false`	0
`minint`	$-2147483648 = -2^{31}$
`maxint`	$2147483647 = 2^{31} - 1$

In Kitty, the integer 0 is treated as `false` while any non-zero integer is treated as `true`. The constant `true` is arbitrarily defined to be 1, but it could be defined as any non-zero integer. The

rational behind the values of `minint` and `maxint` is that these are the smallest and largest integer values than can be expressed in 32 bits (a standard machine word size) using a twos-complement representation.

3.4 I/O operators

Kitty supports a somewhat different input/output (I/O) model than Tiger. In Kitty, the input operators `readc` and `readi` operate on the current input stream and the output operators `writec`, `writei`, and `writes` operate on the current output stream. It is assumed that the current input stream and current output stream can be specified by the environment for executing Kitty programs, which is outside the scope of this language description. Nevertheless, it is imagined that most execution environments will provide a way to choose between standard input and a file for the input stream and to choose between standard output and a file for the output stream.

The meaning of the individual I/O operators is as follows:

`writec(`E`)` first evaluates the expression E, which should denote an integer i. The lower 8 bits of i are interpreted as the ASCII value of the character to be written to the current output stream. For instance, `writec(65)`, `writec(321)`, and `writec(577)` all write the character `A` to the current output stream (because $321 = 65 + 256$ and $577 = 65 + 2 \cdot 256$).

`writei(`E`)` first evaluates the expression E, which should denote an integer i. The digits of i, preceded by a minus sign if i is negative, are written to the current ouput stream. For instance, `writei(321)` writes the three digits `3` and `2` and `1`, while `writei(-273)` writes the `-` character and the three digits `2` and `7` and `3`.

`writes(`S`)` writes the individual characters of the string literal S to the current output stream. It is simply a more convenient way of specifying a long sequence of `writec` applications.

`readc()` consumes the next character from the current input stream, and returns its ASCII integer value, which is in the range [0 .. 255]. If there are no more characters in the input stream (i.e., the end of the stream has been reached), `readc` returns -1.

`readi(`E`)` first evaluates the expression E, which should denote an integer i. Then any whitespace characters (spaces, tabs, and newlines) in the current input stream are consumed. If the first non-whitespace character is a digit, or a minus sign followed by a digit, the maximal sequence of characters interpretable as an integer is consumed, and the integer represented by the returned characters is returned. However, if the first non-whitespace character is not the first character of an integer representation, it is not consumed, and the value i is returned. Thus, i serves as an indication of the failure of `readi` to read an integer. It is helpful to parameterize over this value because different failure values are suitable in different contexts.[2] For instance, 0, -1, `minint`, and `maxint` are all typical values of the argument to `readi`.

4 Examples

This section presents a few examples of simple Kitty programs.

[2] Of course, any attempt to encode a failure value as an integer necessarily means that there can be an ambiguity between actually reading that integer and reading no integer. In more advanced languages, this problem can be dealt with by either returning a compound data structure (such as SML's `int option`) that can represent failure as a value distinct from the integers, or by raising an exception when no integer is read.

4.1 Count

Here is a Kitty program that counts the number of characters in the input stream:

```
let var count := 0;
    var c := readc()
 in while c >= 0 do
      (count := count + 1;
       sc := readc());
    writei(count)
end
```

4.2 Uppercase

This Kitty program copies all characters from the input stream to the output stream, capitalizing all lowercase letters that it encounters:

```
let var c := readc()
 in while c >= 0 do
      (writec(if c >= 'a' & c <= 'z' /* Is c a lower case letter? */
              then c + 'a' - 'A'     /* Yes -- capitalize it */
              else c);               /* No -- just copy it */
       c := readc())
end
```

Capitalization is performed by arithmetic involving character literals (which are just convenient ways to write integers). Note that the `if-then-else` expression in this case returns a value.

4.3 WriteInts

Here is a Kitty program that prints the integers from 1 to 100 to the output stream:

```
for i := 1 to 100 do
  (writei (i);
   writec (if i % 10 = 0 then '\n' else ' ')
   )
```

The output is formatted as s ten lines with ten integers on each line:

```
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100
```

4.4 Sum

Assume that the input stream is a sequence of whitespace-separated integers that does not include 0. The following Kitty program sums all the integers in the input stream and writes the sum to the output stream:

```
let var sum := 0;
    var num := readi(0)
 in while num <> 0 do
      (sum := sum + num;
       num := readi(0));
       writei(sum)
end
```

If the input stream does contain the integer 0, then the above program will only sum all the integers that precede 0 in the stream. Depending on the expected range of integers, it might be better to choose a different failure value other than 0.