CS75 Project 2: Parser for C- -
Spring 2007, Meeden
Part 1: Due by midnight, Thursday, February 22
Part 2-3: Due by midnight, Thursday, March 8

Introduction

For this project you may work in teams of two. Do an `update75` before you begin. You will design and implement a parser for C- - as follows:

1. Derive an LL(1) grammar for C- -. Place your solution in the file: `cs75/projects/2a/LL1grammar`.
2. Using this grammar, implement a recursive-descent parser for C- - that returns an abstract syntax tree representing the complete program. Place your solution in the file: `cs75/projects/2b/parser.py`.
3. Add error recovery to your parser.
4. Use `handin75` to turn in your solutions.

Specifications

- The precedence of C- - operators from lowest to highest is:
  - assignment (=)
  - or (||)
  - and (&&)
  - equal, not equal (==, !=)
  - less than, less than or equal, greater than, greater than or equal (<, <=, >, >=)
  - addition, subtraction (+, -)
  - multiplication, division (*, /)
  - not, unary minus (!, -)
  - parentheses ()
- There is one portion of the C- - grammar that we cannot convert into an LL(1) format and it occurs in some expressions involving the assignment operator. The difficulty is in determining if the expression on the left side of an assignment is a valid *lvalue* (or storage location) at parse time. For example, these are all valid C- - expressions:

  1. `a[i];`
  2. `a[i] = x;`
  3. `x = 8 + (y = 6);`

  In the first example, `a[i]` is not an lvalue. In the second example, `a[i]` is an lvalue. In the third expression, both `x` and `y` are lvalues, and the expression `(y = 6)` returns the value `6`.

  These are **invalid** C- - expressions:

  1. `a[i] + b[i] = x;`
  2. `x = 8 + y = 6;`

  In the first example, `a[i] + b[i]` is not a valid lvalue. In the second example, `8 + y` is not a valid lvalue, and it will be interpreted this way because addition has a higher precedence than assignment.

  Correctly handling these cases requires more than one token of lookahead. Therefore your LL(1) grammar will incorrectly accept strings like `a[i] + b[i] = 8;` Later, when you implement the code generator, you can ensure that any expression appearing on the left side of an assignment expression is a valid lvalue.

Verifying your LL(1) Grammar

You should test your LL(1) grammar by hand to verify that its rules can correctly parse some very small C- - programs. For example, try some programs with zero, one, or two global variable declarations, and one or two small functions:

```
int x;
char y;

int main(char a, int b) {
   x = b;
   y = a;
}
```

Be sure that you have not altered the productions in such a way as to change the language that is recognized. In addition, you should compute the FIRST and FOLLOW sets of your modified grammar to prove that it is actually LL(1).

Verifying your Parser

You should make sure that your parser can correctly parse all valid C- - language constructs and combinations of language constructs before you start adding error recovery.

Your parser should return an abstract syntax tree represented as a Python list.

- At the top level, this list should have two sublists: a list of global declarations (which is potentially empty) and a list of function declarations (there must be at least one called `main`).
- Each global declaration should be a sublist containing the variable type and name.
- Each function declaration should be a sublist containing the function's return type, name, a list of parameters (which is potentially empty), and a block containing the function's statements.
- Each block should be a list containing a sublist of local variable declarations (which is potentially empty) followed by its statements.
- Each statement and expression should be represented as lists in prefix format (where the operator or keyword is first).

Recall that in a recursive descent parser each non-terminal in the LL(1) grammar is implemented as a method. In order to construct the abstract syntax tree, you will need to create and pass lists amongst some of these methods. In Python, lists are objects and therefore maintain state. Any change made to a list that is passed into a function is a permanent change which will be recognized by the calling function.

Error Recovery

Once your parser works, add some simple error recovery. You should implement basic panic and phrase-level recovery schemes for a single missing or incorrect token. In addition, you should add (or integrate existing) error recovery for your scanner such as cases where there are missing or incorrect characters in the input that can be easily handled (e.g. a missing & in the AND operator). When your compiler detects an error it will print out an error message, then either recover from the error and continue parsing or exit if it cannot recover. You are welcome to try more complicated error recovery, but first try implementing some relatively easy cases.

Sample C- - programs and their ASTs

```
int x;

int double(int y) {
  int z;

  z = 2;
  return z*y;
}

int main() {
  read x;
  write double(x);
  return 1;
}
```

```
[[['int', 'x']],
 [['int',
   'double',
   [['int', 'y']],
   ['block',
     [['int', 'z']],
     ['assign', ['id', 'z'], ['num', 2]],
     ['return', ['mul', ['id','z'], ['id', 'y']]]]],
  ['int',
   'main',
   [],
   ['block',
     [],
     ['read', 'x'],
     ['write', ['funcall', 'double', [['id', 'x']]]],
     ['return', ['num', 1]]]]]]
```

```
int initArray(int a[], int size, int value) {
  int i;

  i = 0;
  while (i < size) {
    a[i] = value;
    i = i + 1;
  }
}

int main() {
  int b[10];

  initArray(b, 10, 1);
}
```

```
[[],
 [['int',
   'initArray',
   [['array', 'int', 'a'], ['int', 'size'], ['int', 'value']],
   ['block',
     [['int', 'i']],
     ['assign', ['id', 'i'], ['num', 0]],
     ['while',
       ['ls', ['id', 'i'], ['id', 'size']],
       ['block',
         [],
         ['assign', ['arrayref', 'a', ['id', 'i']], ['id', 'value']],
         ['assign', ['id', 'i'], ['add', ['id', 'i'], ['num', 1]]]]]]],
  ['int',
   'main',
   [],
   ['block',
     [['array', 10, 'int', 'b']],
     ['funcall', 'initArray', [['id', 'b'], ['num', 10], ['num', 1]]]]]]]
```