Project 3a: Recursive descent

Due: 29 February 2012

As promised, the next project in the ongoing sequence is to write a parser. In fact, you'll be writing two—one by hand, and one using `bison`. This document primarily describes the hand-built parser you'll be building.

1 The grammar

The grammar presented here is a relatively small subset of C: large enough to be able to write interesting things, but missing many of the standard features. The grammar will be extended later when you have tools to help you.

c-file:
  top-level EOF
  top-level c-file

top-level:
  func-def
  var-decl

func-def:
  type ident `(` *opt-params* `)` `{` *decls stmts* `}`

decls:
  var-decl decls
  ϵ

stmts:
  stmt stmts
  ϵ

var-decl:
  type var `;`

var:
  ident

type:
  `char`
  `int`

opt-params:
  params
  ϵ

params:
  param-decl
  params `,` *param-decl*

param-decl:
  type ident

stmt:
  expr `;`
  compound-stmt
  if-stmt
  while-stmt
  `return` *expr* `;`

compound-stmt:
  decls stmts

if-stmt:
  `if (` *expr* `)` *stmt* `else` *stmt*

while-stmt:
  `while (` *expr* `)` *stmt*

expr:
  expr bin-op expr
  `(` *expr* `)`
  ident
  ident `(` *opt-exprs* `)`
  char-literal
  int-literal

opt-exprs:
  exprs
  ϵ

exprs:
  expr
  exprs `,` *expr*

bin-op:
  `=`
  `+`
  `-`
  `*`
  `==`
  `!=`

The terminals *ident*, *char-literal*, and *int-literal* as defined in the previous project.

But even with the heavily-restricted subset of C, the above grammar is still not sufficient for what you need to do: it is not in LL(1) form. There are some cases of left recursion and several rules that require left-factoring. The precedence and associativity rules for the binary operators are not accounted for, either. One of your first tasks in approaching this project is to convert the grammar into one that recognises the same language but is LL(1).

2 The calling convention, and lexer tweaks

You'll define a class for your parser, and this class will contain all of your recursive descent functions as methods, most of which can be private. The two important things that this class will need to have are a constructor that accepts, as its only argument, a reference to a lexer object (the one you wrote for the previous project); and a method `parse`, with no arguments, that returns an abstract syntax tree for the entire file.

Inside that `parse` function, you'll dive right into your predictive parsing, but when you reach the leaves of the parse tree, you'll have to interact with the lexer. To make that interface clean, you'll want to tweak the interface of the lexer slightly: specifically, introduce public methods `peek()` (which returns the token most recently returned by `yylex`) and `consume()` (which

actually calls `yylex` again). Your parser will not call `yylex` directly.

You'll also need to add a semicolon to your available symbols, and to your lexer rules (oops).

3 The return type

Mindful of the fact that the next phase after parsing is analysis and code generation, the return type of your `parse` function should be a pointer to a (`new`-allocated) object that gives the abstract syntax tree (AST) for the entire file. Indeed, the return type for nearly every parsing function in your parser will be a pointer to an AST object, except for a few that might return a *collection* of pointers to AST objects, and maybe some leaf-level parsing functions that return an int or something.

To make those later analysis and generation phases easier, you want to build your AST from specific subtypes of an abstract AST node class; that way their instance variables can be tailored to what they actually need to do, and they can have specific methods that are aware of those instance variables.

As an example, somewhere in your class hierarchy, you might end up with a class defined as follows:

```
class simple_stmt_node : public stmt_node
{
  public:
    simple_stmt_node (expr_node *e);
    virtual ~simple_stmt_node();

    virtual void print_with_prefix (const string &prefix) const;

  private:
    expr_node *expr;
};
```

You don't necessarily need this class, and if you have one it needn't look exactly like this; but this might give you an idea of what I'm looking for (and how to structure it).

One of your first tasks in approaching this project is to design a class hierarchy of abstract syntax tree node classes that will be able to support all

the ASTs that will arise from the given grammar.

4 The output format

In future project, you'll manipulate the trees directly, but for now I just want you to print them out. The easiest way to print out an AST (or any tree) is if each node type knows how to print itself at a given indentation level—and takes responsibility for asking its child nodes to print themselves at a deeper indentation level. Your complete `main` function could thus be:

```
int main ()
{
  CLexer lexer;
  cparser parser(lexer);
  parser.parse()->print_with_prefix("");

  return 0;
}
```

As an example of how this will work, one of your `print_with_prefix` methods might be defined as follows:

```
void if_stmt_node::print_with_prefix(const string &prefix) const
{
  cout << prefix << "IF:" << endl;
  cond_expr->print_with_prefix(prefix + "  ");

  cout << prefix << "THEN:" << endl;
  then_body->print_with_prefix(prefix + "  ");

  cout << prefix << "ELSE:" << endl;
  else_body->print_with_prefix(prefix + "  ");
}
```

Getting all of these print methods written is admittedly a bit tedious, but judicious use of helper methods should ease the burden somewhat.

5 For Monday

For Monday, you should have draft versions of each of the following:

1. An LL(1) version of the grammar, including FIRST and FOLLOW sets for each grammar symbol
2. The class hierarchy of the classes used to represent ASTs, including instance variables for each
3. A memory diagram showing a complete (but short) concrete example of an AST

The difference between the two diagrams is that one shows the subclass relationships among classes (the "is-a" relationships) and the other shows the tree structure among objects (the "has-a" relationships).

Bring your draft versions to class and be prepared to talk about them. I'll want you to turn in your final versions of all three on Wednesday.

6 Final version

The actual program is due on 29 February. Hand in using the handin script on torvalds, with assignment name `proj3a`. Include a Makefile and/or any instructions I'll need to get the program running, and also include appropriate test cases that you've run, that demonstrate that your program works.