Parsing and Semantic Analysis

DEADLINE: March 26, 23:59.

IMPORTANT: You should hand in your solution on the Minerva Dropbox. Create an archive using the `make_tarball.sh` script, and make sure you send it to the teaching assistants.
For any questions, contact us at `compilers@lists.ugent.be`.

1 Introduction

The goal of this lab is to complete the implementation of a parser for the same C-like language you have worked with in the first lab. You will need to create a BNF grammar that implements the rules of this language, and get familiar with the Bison parser-generator tool in order to implement this grammar.

The source files we give you provides the front-end of a very simple compiler: tokens are lexed using Flex in `lexer.l`, an abstract syntax tree representation is provided in the `ast` subfolder, and an initial Bison-based parser that constructs the AST is provided in `parser.y`. Semantic actions are implemented in the files of the `sema` subfolder, and are intended to be used during AST construction to implement logic that is not contained by the BNF grammar.

2 Set-up

The assignment for this lab builds on Bison 3.0[1], while also using LLVM 7.0 and Flex 2.6; correct versions of these packages are provided as part of the `tbesard/compilers:pract2` image on Docker Hub:

```
$ docker run --rm -it tbesard/compilers:pract2
root@0f01d2b521d6:/#
```

2.1 Configure and compile

Launch an instance of this container with the assignment's files mounted, and navigate to the folder (refer to the first lab's assignment for more details). Start by invoking CMake to configure the project:

```
# configure the code (do this once)
cmake .
```

To actually compile the files, use regular `make`:

```
# compile the code (do this after every modification)
make
```

[1]You can find the documentation for this specific version of Bison at `https://users.elis.ugent.be/~tbesard/compilers/bison/`

2.2 Execution and visualization

After compiling, a binary named `cheetah` will be created. This binary takes as input a Cheetah C (a subset of the C language) source file, and outputs an AST in the form of a GraphViz DOT graph. You can render these graphs to a PNG, or visualize them directly using the `xdot` tool:

```
./cheetah ../test/dummy.c > dummy.dot   # dummy.c can be parsed out-of-the-box
xdot dummy.dot                          # interactive viewer
dot -Tpng dummy.dot > dummy.png         # static PNG
```

Note that the `xdot` tool is not provided in the container, and needs to be installed on and executed from your host operation system.

In the resulting graph, declarations are colored green, expressions are blue, and statements are red. To keep the graph small, `DeclStmt` wrapper nodes are not visualized but directly render the contained declaration. Similarly, identifiers are displayed as part of the containing node.

3 Assignment

The goal of this lab is to complete the implementation of the parser, and add semantic actions where necessary. After completing all assignments, the AST for `fibonacci.c` should look like the graph as shown in `finbonacci.png`.

You will be mostly working in the `parser.y` file, which defines a BNF-like grammar to be processed by the Bison LALR(1) parser generator. The initial code as part of this assignment already contains quite a lot of boilerplate; do not pay too much attention to it, and focus on the TODO markers across the files.

To add functionality to the parser, you typically need to perform the following steps:

1. **define terminal symbols**: e.g. `%token EQUAL`, corresponding to an `EQUAL` token as produced by the lexer. After adding such a definition, you can use the terminal symbol in a rule.
2. **define semantic values types**: e.g. `%union { AST::Foo *bar_t }`. Bison puts values created by rules on a stack. The type of values on this stack is a union type, and you might need to add new types to this union.
3. **define nonterminals**: e.g. `%type <expr_t> bar`. This defines a nonterminal, `bar`, of type `expr_t` (which needs to be a valid semantic value type as described above). You can now use these nonterminals in a rule, and if you access them as part of the rule's action that value will be of type `expr_t`.
4. **define rules**: you can now use defined terminals (`%token`) and nonterminals (`%type`) to create productions, for example `foo: '(' bar ')';`. Multiple rules for the same result (here: `foo`) can be joined with the `|` character. To keep rules comprehensible, create additional nonterminals to group common grammar snippets.
5. **define actions**: finally, when rules match, certain actions such as constructing AST nodes need to be performed. You can encode these actions as snippets of C code next to the right-hand side of each rule, e.g. `foo: '(' bar ')' { foobar(); }`. Other rules need more complex semantic actions that cannot be expressed with grammar alone. You can implement this logic in the `Sema` class, and access an instance if this class through the `sema` object.

Within an action, you can access the individual components of the rule by using numbered variables, for example in the previous rule `$1` will match the output of `bar`. Sometimes, the type of these values will be too generic. For example, if you defined `bar` to be of `%type <expr_t>`, accessing `$1` will result in a value of type `AST::Expr*` (the `expr_t` field of the `%union`). You can work around this by manually casting the value to another type of the union using `$<bar_t>1`. If you need to perform this cast in the majority of cases, you better define the type of this terminal to be `bar_t`: `%type <bar_t> bar`.

To produce a value, assign it to the `$$` variable. Ensure that the type of the produced value matches the type of the `$$` variable, as defined by the aforementioned `%type` rules.

To get you started, we've provided the nonterminals and rules to parse basic declarations:

- `int foo`
- `float bar = something`

Look at the existing rules to understand how the parser works and processes `test/dummy.c`. Also look at the definitions of the corresponding AST nodes: they determine the types of the rule's components, which you need to match when implementing new rules.

Make sure you create additional tests covering the language as you implement new rules, and frequently verify the ASTs constructed by them. It is recommended you employ test-driven development: start by creating a source file that demonstrates the feature you want to implement, and extend the code until the file successfully parses.

3.1 Function call

Start with implementing the function call expression, `func(args)`. This rule needs to produce an expression, and as such needs to be part of the `expr` production. A function call is represented by a `AST::CallExpr` object, so you will need to create such an object in the action of your rule. The arguments to this AST node are the identifier of the called function, and an `AST::ExprList` representing the list of arguments.

An argument list of type `AST::ExprList` needs to be constructed incrementally. You should use the `ParseExprList` function as defined in the `Sema` class. You will need to define a recursive rule that creates such an expression list (also dealing with the case where no arguments are provided, e.g. `func()`, creating an empty expression list by just calling `sema.ParseExprList()`).

Also make sure you preserve location information of the newly created AST nodes.

3.2 Literals

Next, implement rules for literals (integer, floating-point and string). These are simple rules, returning an expression (i.e. part of the `expr` production) but require a semantic action to parse the token value to a semantic value. Do so by adding functions to the `Sema` class in `sema/semaexpr.cpp` and calling them through the `sema` object.

You can access the raw token value by calling `yytext(lexer)` from the parser (i.e. in `parser.y`). You should parse these strings by calling into the C standard library (look up `strtol` and `strtof`). Make sure you catch and report invalid conversion by throwing a `semantic_error` exception.

3.2.1 Operators

Now create terminals and rules for the following expressions:

- binary operators: `CEQ`, `CNE`, `CLT`, `CLE`, `CGT`, `CGE`, `PLUS`, `MINUS`, `MUL`, `DIV`, `MOD` and `EXP`.
- unary operators: `PLUS` and `MINUS`

These rules do not need semantic actions, and as such can directly construct the relevant AST nodes. However, you need to take care of the following precedence rules:

- `EXP` is the most tightly bound, right-associative operator
- next are unary operators
- `MUL`, `DIV`, `MOD`, with left associativity
- `PLUS` and `MINUS` with left associativity
- comparison operators, without associativity, and unable to appear multiple times in a row
- lastly, assignment (`EQUAL`), also without associativity

3.3 Control flow

Now implement the following statements:

- `if (cond) { stmts }` and `if (cond) { stmts } else { stmts }`, with `cond` an expression.
- `while (cond) { stmts }`
- `for (init, cond, inc) { stmts }` where `init` is a an expression or a variable declaration, and both `cond` and `inc` are regular expressions.
- `return` and `return expr`

The rules for these statements do not require semantic actions, and as such can directly construct the relevant AST nodes.

3.3.1 One-line conditionals

Finally, add support for one-line conditionals that do not use curly braces. For example:

```
if (something)
  foo();
else
  bar();
```

Make sure this does not introduce conflicts into your grammar!

4 Submission

You are only allowed to modify `parser.y`, and files related to semantic analysis. Make sure your solution successfully parses `fibonacci.c`, and compare the resulting AST to the our graph. Additional tests are handed in as well, and you will be graded on their coverage of the language as described in this document (even if those constructs are not present in `fibonacci.c`).