CS406: Compilers

Programming Assignment 2: Parser, Due: 13/2/2020

1 Introduction

Your goal in this step is to generate the parser for your programming language's grammar. By the end of this step your compiler should be able to take a source file as the input and parse the content of that file returning "Accepted" if the file's content is correct according to the grammar or "Not Accepted" if it is not.

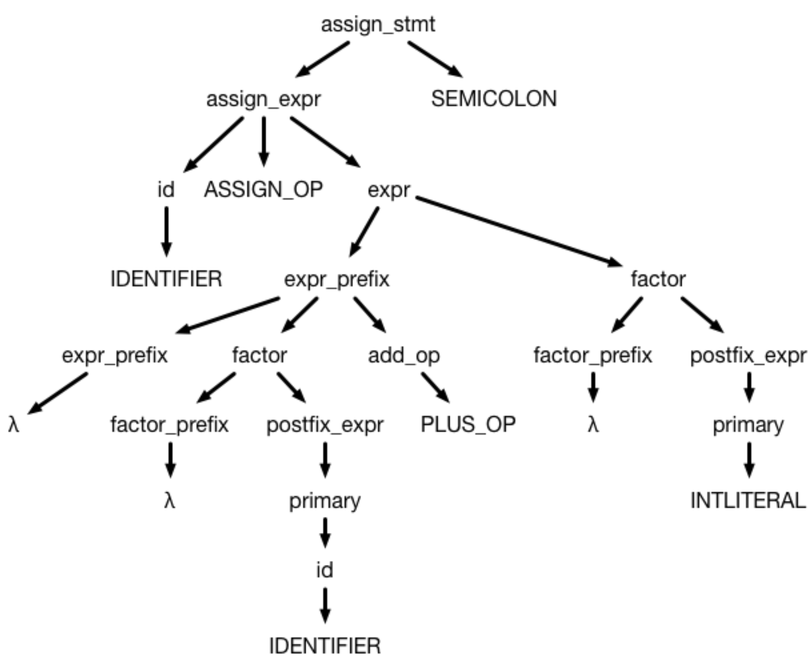
Now the scanner created in the first step will be modified to feed the parser. Instead of printing the tokens, the scanner has to return what token is recognized in each step.

2 Background

The job of a parser is to convert a stream of tokens (as identified by the scanner) into a *parse tree*: a representation of the structure of the program. So, for example, a parser will convert:

```
A := B + 4
```

into a tree that looks something like:



This tree may look confusing, but it fundamentally captures the structure of an *assignment statement*: an assignment statement is an *assignment expression* followed by a semicolon. An assignment expression is decomposed into an *identifier* followed by an assignment operation followed by an *expression*. That expression is decomposed into a bunch of *primary* terms that are combined with addition and subtraction, and those primary terms are decomposed into a bunch of *factors* that are combined with multiplication and division (this weird decomposition of expressions captures the necessary order of operations). Eventually, those *factors* become identifiers or constants.

One important thing to note is that the *leaves* of the tree are the tokens of of the program. If you read the leaves of the tree left to right (ignoring lambdas, since just represent the empty string), you get:

`IDENTIFIER ASSIGN_OP IDENTIFIER PLUS_OP INTLITERAL`

Which is exactly the tokenization of the input program!

2.1 Context-free grammars

To figure out how each construct in a program (an expression, an if statement, etc.) is decomposed into smaller pieces and, ultimately, tokens, we use a set of rules called a *context-free grammar*. These rules tell us how constructs (which we call "non-terminals") can be decomposed and written in terms of other constructs and tokens (which we call "terminals").

2.2 Micro

The context-free grammar that defines the structure of Micro (the name of your programming language) is:

```
/* Program */
program             -> PROGRAM id BEGIN pgm_body END
id                  -> IDENTIFIER
pgm_body            -> decl func_declarations
decl                  -> string_decl decl | var_decl decl | empty

/* Global String Declaration */
string_decl         -> STRING id := str ;
str                 -> STRINGLITERAL

/* Variable Declaration */
var_decl            -> var_type id_list ;
var_type              -> FLOAT | INT
any_type            -> var_type | VOID
id_list             -> id id_tail
id_tail             -> , id id_tail | empty

/* Function Paramater List */
param_decl_list     -> param_decl param_decl_tail | empty
param_decl          -> var_type id
param_decl_tail     -> , param_decl param_decl_tail | empty

/* Function Declarations */
func_declarations -> func_decl func_declarations | empty
func_decl           -> FUNCTION any_type id (param_decl_list) BEGIN func_body END
func_body           -> decl stmt_list

/* Statement List */
stmt_list           -> stmt stmt_list | empty
stmt                -> base_stmt | if_stmt | for_stmt
base_stmt           -> assign_stmt | read_stmt | write_stmt | return_stmt

/* Basic Statements */
assign_stmt         -> assign_expr ;
assign_expr         -> id := expr
read_stmt           -> READ ( id_list );
write_stmt          -> WRITE ( id_list );
return_stmt         -> RETURN expr ;

/* Expressions */
```

```
expr                -> expr_prefix factor
expr_prefix         -> expr_prefix factor addop | empty
factor              -> factor_prefix postfix_expr
factor_prefix       -> factor_prefix postfix_expr mulop | empty
postfix_expr        -> primary | call_expr
call_expr           -> id ( expr_list )
expr_list           -> expr expr_list_tail | empty
expr_list_tail      -> , expr expr_list_tail | empty
primary             -> ( expr ) | id | INTLITERAL | FLOATLITERAL
addop               -> + | -
mulop               -> * | /

/* Complex Statements and Condition */
if_stmt             -> IF ( cond ) decl stmt_list else_part FI
else_part           -> ELSE decl stmt_list | empty
cond                -> expr compop expr
compop              -> < | > | = | != | <= | >=

init_stmt           -> assign_expr | empty
incr_stmt           -> assign_expr | empty

for_stmt            -> FOR ( init_stmt ; cond ; incr_stmt ) decl stmt_list ROF
```

So this grammar tells us, for example, that an `if_stmt` looks like the keyword `IF` followed by an open parenthesis, followed by a `cond` expression followed by some `decl` (declarations) followed by a `stmt_list` followed by an `else_part` followed by the keyword `FI`.

An input program matches the grammar (we say “is accepted by” the grammar) if you can use the rules of the grammar (starting from `program`) to generate the set of tokens that are in the input file. If there is no way to use the rules to generate the input file, then the program does not match the grammar, and hence is not a syntactically valid program.

3 Building a Parser

There are many tools that make it relatively easy to build a parser for a context free grammar (in class, we will talk about how these tools work): all you need to do is provide the context-free grammar and some actions to take when various constructs are recognized. The tools we recommend using are:

- bison (this is a tool that is meant to work with scanners built using flex). Note that integrating a flex scanner with bison requires a little bit of work. The process works in several steps that seem interlocking:

  1. Define your token names as well as your grammar in your bison input file (called something like `microParser.y`)

  2. Run `bison -d -o microParser.cpp` on `microParser.y`, it will create two output files: `microParser.cpp` (which is your parser) and `microParser.tab.hpp` (which is a header file that defines the token names)

  3. In your scanner file (called something like `microLexer.l`), add actions to each of your token regexes to simply return the token name (from your `.y` file) you defined. (*Warning*: make sure that you don’t have a token named `BEGIN` even if the regex matches the string “BEGIN”, because that will cause weird, hard-to-find errors. Call that token something like `_BEGIN`.) To make sure that your scanner compiles, you will need to put `#include ‘‘microParser.tab.hpp’’` in the part of your `.l` file where you can include C code.

  4. Run flex on `microLexer.l` to produce `lex.yy.cpp`

5. In another file, write a `main` function (this file will also need to include `microParser.tab.hpp`). Your main function should open the input file and store the file handle in a variable called `yyin`. Calling `yyparse()` will then run your parser on the file associated with `yyin`.
6. Compile together all of `microParser.cpp`, `lex.yy.cpp`, and your `main` function to build your compiler.

- ANTLR (this is the same tool that can also build lexers). You should define your grammar in the same `.g4` file in which you defined your lexer.
  1. Running ANTLR on that `.g4` file will produce both a Lexer class and a Parser class.
  2. In your main file, rather than initializing a lexer and then grabbing tokens from it (as you may have done in step 1), you instead initialize a lexer, initialize a `CommonTokenStream` from that lexer, then initialize a parser with the `CommonTokenStream` you just created.
  3. You can then call a function with the same name as your top-level construct (probably program) on that parser to parse your input.

4 What you need to do

The grammar for Micro is given above. All you need to do is have your parser parse the given input file and print `Accepted` if the input file correctly matches the grammar, and `Not Accepted` if it doesn't (i.e., the input file cannot be produced using the grammar rules).

In bison, you can define a function called `yyerror` (look at the documentation for the appropriate signature) that is called if the parser encounters an error.

In antlr, this is a little more complicated. You will need to create a new "error strategy" class (extend `DefaultErrorStrategy`) that overrides the function `reportError`. You can then set this as the error handler for your parser by calling `setErrorHandler` on your parser before starting to parse.

Sample inputs and outputs: inputs and outputs.

5 What you need to submit

- All of the necessary code for your compiler that you wrote yourself. You do not need to include the ANTLR jar files if you are using ANTLR.
- A Makefile with the following targets:
  1. `compiler`: this target will build your compiler
  2. `clean`: this target will remove any intermediate files that were created to build the compiler
  3. `team`: this target will print the same team information that you printed in step 0.
- A shell script (this must be written in bash) called `runme` that runs your scanner. This script should take in two arguments: first, the input file to the scanner and second, the filename where you want to put the scanner's output. You can assume that we will have run `make compiler` before running this script.

While you may create as many other directories as you would like to organize your code or any intermediate products of the compilation process, both your `Makefile` and your `runme` script should be in the root directory of your repository.

Do not submit any binaries. Your git repo should only contain source files; no products of compilation.

You should tag your programming assignment submission as submission