Compilers

CMPT 432

Semester Project *part one*

This part of your semester-long project is meant to be done over the course of our semester together, right up until about the final three or four weeks. Do not attempt it all at once. Rather, begin after our first class and make steady progress each week. This is the "traditional compiler" portion of the project… just highly condensed. You're going to have to be very productive to get it done, so lean in to your AI tools.

Use an ***Alan-approved*** programming language to write a four-phase compiler for the grammar found on our class web site at https://www.labouseur.com/courses/compilers/grammar.pdf and the scope and type rules discussed in class. The four phases are noted below. Read the source code from a file for command-line compilers or from an HTML text area element for browser-based compilers.

Lexer Notes and Requirements

- Begin by writing a lexer that validates the input source code against our grammar and creates tokens.
- You must lex multiple programs in sequence. Each program will be separated by the $ [EOP] marker.
- The lexer is not as simple as our examples in class, so be careful.
- Provide both errors and warnings. Warnings are non-fatal mistakes or omissions that your compiler should correct. Forgetting to end the final program with $ is one example. Detecting unterminated comment blocks is another.
- When you detect an error, report it in excruciating detail with helpful text and messages, including where it was found, what exactly went wrong, and how the programmer might fix it. Confusing, incomplete, or inaccurate error messages are serious (and intolerable) bugs.
- Include verbose output functionality that traces the stages of the lexer. Make this the default mode.
- When there are errors detected in Lex, do not continue to Parse.

Parser Notes and Requirements

- Add a **recursive descent** parser to your compiler that takes the tokens from your Lexer.
- While parsing, create a Concrete Syntax Tree (CST). If parsing is successful (i.e., no errors were found) then display the CST. Make it neat and pretty.
- Your parser must compile multiple programs in sequence, just like your Lexer.
- Provide errors, hints, and warnings. Errors are, well, errors. Hints are messages about code hygiene issues and opportunities for improvement. Warnings are non-fatal mistakes or omissions that probably should not have been done, but which don't actually violate the grammar, yet indicate potential issues, like uninitialized variables for example.
- When you detect an error, report it in excruciating detail with helpful text and messages, including where it was found, what exactly went wrong, and how the programmer might fix it. Confusing, incomplete, or inaccurate error messages are serious (and intolerable) bugs.
- Include verbose output functionality that traces the stages of your Parser. Keep this the default.
- When there are errors detected in Parse, do not display the CST and do not continue to Semantic Analysis. (Hints and warnings should not prevent you from going to the next phase.)

Compilers

CMPT 432

Semantic Analysis Notes and Requirements

- Build an Abstract Syntax Tree (AST) by either re-parsing the tokens or traversing the CST. Display it after a successful Lex and Parse.
- Scope-check the AST according to the scope rules we discussed in class.
- While you are scope-checking, build a symbol table of IDs that includes their name, data type, scope, position in the source code, and anything else you think might be important.
- Type-check the source code using the AST and the symbol table, based on our grammar and the type rules we discussed in class.
  - Issue errors for undeclared identifiers, redeclared identifiers in the same scope, type mismatches, and anything else that might go wrong.
  - Issue hints and warnings about declared but unused identifiers, use of uninitialized variables, and the presence of initialized but unused variables.
- Default to verbose output functionality that traces the Semantic Analysis stages, including scope checking, the construction of the symbol table, and type checking actions.
- When you detect an error, report it in helpful detail including where it was found.
- Create and display a symbol table with type and scope information, unless…
- … if there are errors detected in Semantic Analysis then do not display symbol table with type and scope information and do not continue to Code Generation.

Code Generator Notes and Requirements

- Write a code generator that takes your AST and generates 6502a machine code (found in 6502a-instruction-set.pdf) for our language grammar.
- As with the phases before this one, include verbose output functionality that traces the stages of the parser including the construction of the symbol table and type checking actions.
- When you detect an error report it in helpful detail including where it was found.
- The generated code must conform to the 6502a instructions set specified on our class web site and execute on SvegOS.
- If you're feeling up to it, consider adding one or more of the following for extra credit and extra coolness: various code optimizations (ask me about it), non-value-returning procedures (sub program call and return), value-returning functions (sub program call and return), integer arrays, or more.

General Requirements and Hints

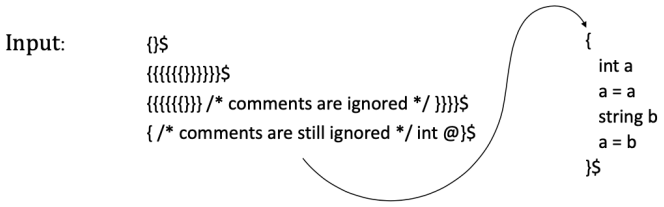
- At each phase, create a plethora (yes, a **plethora**!) of test programs that cause as many different kinds of errors as you can in order to thoroughly test your code. Think about code coverage and edge cases. Think about generating hints and warnings too. Include test cases that show it working as well. Write up your testing results (informally) in a document in your Git repository. Markdown is particularly good for this.

General Requirements and Hints continued

- Your code must …
  - separate structure from presentation.
  - be professionally formatted yet uniquely yours (show some personality).
  - use and demonstrate best practices.
  - make me proud to be your teacher.
- Remember the utility of comments and how much their presence and quality affect your professionalism and my opinion of your work.
- Details matter and neatness counts.
- See the examples on the next few pages and in the Hall of Fame on our web site for details and ideas.

Examples and Ideas

The following pages show a compiler in various phases of operation . . .



Output to screen:

```
INFO  Lexer - Lexing program 1...
DEBUG Lexer - OPEN_BLOCK [ { ] found at (1:1)
DEBUG Lexer - CLOSE_BLOCK [ } ] found at (1:2)
DEBUG Lexer - EOP [ $ ] found at (1:3)
INFO  Lexer - Lex completed with 0 errors

INFO  Lexer - Lexing program 2...
DEBUG Lexer - OPEN_BLOCK [ { ] found at (2:1)
DEBUG Lexer - OPEN_BLOCK [ { ] found at (2:2)
DEBUG Lexer - OPEN_BLOCK [ { ] found at (2:3)
DEBUG Lexer - OPEN_BLOCK [ { ] found at (2:4)
DEBUG Lexer - OPEN_BLOCK [ { ] found at (2:5)
DEBUG Lexer - OPEN_BLOCK [ { ] found at (2:6)
DEBUG Lexer - CLOSE_BLOCK [ } ] found at (2:7)
DEBUG Lexer - CLOSE_BLOCK [ } ] found at (2:8)
DEBUG Lexer - CLOSE_BLOCK [ } ] found at (2:9)
DEBUG Lexer - CLOSE_BLOCK [ } ] found at (2:10)
DEBUG Lexer - CLOSE_BLOCK [ } ] found at (2:11)
DEBUG Lexer - CLOSE_BLOCK [ } ] found at (2:12)
DEBUG Lexer - EOP [ $ ] found at (2:13)
INFO  Lexer - Lex completed with 0 errors

INFO  Lexer - Lexing program 3...
DEBUG Lexer - OPEN_BLOCK [ { ] found at (3:1)
DEBUG Lexer - OPEN_BLOCK [ { ] found at (3:2)
DEBUG Lexer - OPEN_BLOCK [ { ] found at (3:3)
DEBUG Lexer - OPEN_BLOCK [ { ] found at (3:4)
DEBUG Lexer - OPEN_BLOCK [ { ] found at (3:5)
DEBUG Lexer - OPEN_BLOCK [ { ] found at (3:6)
DEBUG Lexer - CLOSE_BLOCK [ } ] found at (3:7)
DEBUG Lexer - CLOSE_BLOCK [ } ] found at (3:8)
DEBUG Lexer - CLOSE_BLOCK [ } ] found at (3:9)
DEBUG Lexer - CLOSE_BLOCK [ } ] found at (3:38)
DEBUG Lexer - CLOSE_BLOCK [ } ] found at (3:39)
DEBUG Lexer - CLOSE_BLOCK [ } ] found at (3:40)
DEBUG Lexer - CLOSE_BLOCK [ } ] found at (3:41)
DEBUG Lexer - EOP [ $ ] found at (3:42)
INFO  Lexer - Lex completed with 0 errors

INFO  Lexer - Lexing program 4...
DEBUG Lexer - OPEN_BLOCK [ { ] found at (4:1)
DEBUG Lexer - I_TYPE [ int ] found at (4:36)
ERROR Lexer - Error:4:40 Unrecognized Token: @
DEBUG Lexer - CLOSE_BLOCK [ } ] found at (4:41)
DEBUG Lexer - EOP [ $ ] found at (4:42)
ERROR Lexer - Lex failed with 1 error(s)
```

```
INFO  Lexer - Lexing program 5…
DEBUG Lexer - OPEN_BLOCK [ { ] found at (5:1)
DEBUG Lexer - I_TYPE [ int ] found at (6:3)
DEBUG Lexer - ID [ a ] found at (6:7)
DEBUG Lexer - ID [ a ] found at (7:3)
DEBUG Lexer - ASSIGN_OP [ = ] found at (7:5)
DEBUG Lexer - ID [ a ] found at (7:7)
DEBUG Lexer - I_TYPE [ string ] found at (8:3)
DEBUG Lexer - ID [ b ] found at (8:10)
DEBUG Lexer - ID [ a ] found at (9:3)
DEBUG Lexer - ASSIGN_OP [ = ] found at (9:5)
DEBUG Lexer - ID [ b ] found at (9:7)
DEBUG Lexer - CLOSE_BLOCK [ } ] found at (10:1)
DEBUG Lexer - EOP [ $ ] found at (10:2)
INFO  Lexer - Lex completed with 0 errors
```

Input: {}$
{{{{{{{}}}}}}}$
{{{{{{}}} /* comments are ignored */ }}}}$
{ /* comments are still ignored */ int @}$

Output to screen:

```
DEBUG: Running in verbose mode

LEXER: Lexing program 1...
LEXER: "{" --> [LBRACE]
LEXER: "}" --> [RBRACE]
LEXER: "$" --> [EOP]
LEXER: Lex completed successfully

PARSER: Parsing program 1...
PARSER: parse()
PARSER: parseProgram()
PARSER: parseBlock()
PARSER: parseStatementList()
PARSER: Parse completed successfully

CST for program 1...
<Program>
-<Block>
--[{]
--<Statement List>
--[}]
-[$]


LEXER: Lexing program 2...
LEXER: "{" --> [LBRACE]
LEXER: "{" --> [LBRACE]
LEXER: "{" --> [LBRACE]
LEXER: "{" --> [LBRACE]
LEXER: "{" --> [LBRACE]
LEXER: "{" --> [LBRACE]
LEXER: "{" --> [LBRACE]
LEXER: "}" --> [RBRACE]
LEXER: "}" --> [RBRACE]
LEXER: "}" --> [RBRACE]
LEXER: "}" --> [RBRACE]
LEXER: "}" --> [RBRACE]
LEXER: "}" --> [RBRACE]
LEXER: "$" --> [EOP]
LEXER: Lex completed successfully

PARSER: Parsing program 2...
PARSER: parse()
PARSER: parseProgram()
PARSER: parseBlock()
PARSER: parseStatementList()
PARSER: parseStatement()
PARSER: parseBlock()
PARSER: parseStatementList()
PARSER: parseStatement()
PARSER: parseBlock()
PARSER: parseStatementList()
PARSER: parseStatement()
PARSER: parseBlock()
PARSER: parseStatementList()
PARSER: parseStatement()
PARSER: parseBlock()
PARSER: parseStatementList()
PARSER: parseStatement()
PARSER: parseBlock()
PARSER: parseStatementList()
PARSER: Parse completed successfully

CST for program 2...
<Program>
-<Block>
--[{]
--<Statement List>
---<Statement>
----<Block>
-----[{]
-----<Statement List>
------<Statement>
-------<Block>
--------[{]
--------<Statement List>
---------<Statement>
----------<Block>
----------[{]
----------<Statement List>
-----------<Statement>
------------<Block>
------------[{]
------------<Statement List>
-------------<Statement>
--------------<Block>
--------------[{]
--------------<Statement List>
--------------[}]
-------------[}]
----------[}]
--------[}]
-----[}]
--[}]
-[$]

LEXER: Lexing program 3...
LEXER: "{" --> [LBRACE]
LEXER: "{" --> [LBRACE]
LEXER: "{" --> [LBRACE]
LEXER: "{" --> [LBRACE]
LEXER: "{" --> [LBRACE]
LEXER: "{" --> [LBRACE]
LEXER: "}" --> [RBRACE]
LEXER: "}" --> [RBRACE]
LEXER: "}" --> [RBRACE]
LEXER: "}" --> [RBRACE]
LEXER: "}" --> [RBRACE]
LEXER: "}" --> [RBRACE]
LEXER: "}" --> [RBRACE]
LEXER: "$" --> [EOL]
LEXER: Lex completed successfully

PARSER: Parsing program 3...
PARSER: parse()
PARSER: parseProgram()
PARSER: parseBlock()
PARSER: parseStatementList()
PARSER: parseStatement()
PARSER: parseBlock()
PARSER: parseStatementList()
PARSER: parseStatement()
PARSER: parseBlock()
PARSER: parseStatementList()
PARSER: parseStatement()
PARSER: parseBlock()
PARSER: parseStatementList()
PARSER: parseStatement()
PARSER: parseBlock()
PARSER: parseStatementList()
PARSER: parseStatement()
PARSER: parseBlock()
PARSER: parseStatementList()
PARSER: parseStatementList()
PARSER: parseStatementList()
PARSER: parseStatementList()
PARSER: parseStatementList()
PARSER: parseStatementList()
PARSER: ERROR: Expected [EOL] got [RBRACE] with value '}' on line 0
PARSER: Parse failed with 1 error

CST for program 3: Skipped due to PARSER error(s).


LEXER: Lexing program 4...
LEXER: "{" --> [LBRACE]
LEXER: "int" --> [TYPE]
LEXER: ERROR: Unrecognized Token: @
LEXER: "}" --> [RBRACE]
LEXER: "$" --> [EOL]
LEXER: Lex completed with 1 error

PARSER: Skipped due to LEXER error(s)

CST for program 4: Skipped due to LEXER error(s).
```

Compilers

CMPT 432

Input:

```
{
   int a
   boolean b
   {
      string c
      a = 5
      b = true  /* no comment */
      c = "inta"
      print(c)
   }
   print(b)
   print(a)
}$
```

Output:

```
Program 1 Lexical Analysis
Program 1 Lexical analysis produced
0 error(s) and 0 warning(s)

Program 1 Parsing
Program 1 Parsing produced 0
error(s) and 0 warnings

Program 1 Semantic Analysis
Program 1 Semantic Analysis produced
0 error(s) and 0 warning(s)
```

```
Program 1 Concrete Syntax Tree
-------------------------------------
< Program >
-< Block >
--[ { ]
--< Statement List >
---< Statement >
----< Variable Declaration >
-----[ int ]
-----[ a ]
---< Statement List >
----< Statement >
-----< Variable Declaration >
------[ boolean ]
------[ b ]
----< Statement List >
-----< Statement >
------< Block >
-------[ { ]
-------< Statement List >
--------< Statement >
---------< Variable Declaration >
----------[ string ]
----------[ c ]
--------< Statement List >
---------< Statement >
----------< Assignment Statement >
-----------[ a ]
-----------[ = ]
-----------< Expression >
------------< Int Expression >
-------------[ 5 ]
---------< Statement List >
----------< Statement >
-----------< Assignment Statement >
------------[ b ]
------------[ = ]
------------< Expression >
-------------< Boolean Expression >
--------------[ true ]
----------< Statement List >
-----------< Statement >
------------< Assignment Statement >
-------------[ c ]
-------------[ = ]
-------------< Expression >
--------------< String Expression >
---------------[ " ]
---------------< Char List >
----------------[ i ]
----------------< Char List >
-----------------[ n ]
-----------------< Char List >
------------------[ t ]
------------------< Char List >
-------------------[ a ]
-------------------< Char List >
---------------[ " ]
-----------< Statement List >
------------< Statement >
-------------< Print Statement >
--------------[ print ]
--------------[ ( ]
--------------< Expression >
---------------[ c ]
---------------[ ) ]
------------< Statement List >
-------[ } ]
-----< Statement List >
------< Statement >
-------< Print Statement >
--------[ print ]
--------[ ( ]
--------< Expression >
---------[ b ]
--------[ ) ]
------< Statement List >
-------< Statement >
--------< Print Statement >
---------[ print ]
---------[ ( ]
---------< Expression >
----------[ a ]
---------[ ) ]
-------< Statement List >
--[ } ]
-[ $ ]
```

```
Program 1 Abstract Syntax Tree
------------------------------
< BLOCK >
-< Variable Declaration >
--[ int ]
--[ a ]
-< Variable Declaration >
--[ boolean ]
--[ b ]
-< BLOCK >
--< Variable Declaration >
---[ string ]
---[ c ]
--< Assignment Statement >
---[ a ]
---[ 5 ]
--< Assignment Statement >
---[ b ]
---[ true ]
--< Assignment Statement >
---[ c ]
---[ inta ]
--< Print Statement >
---[ c ]
-< Print Statement >
--[ b ]
-< Print Statement >
--[ a ]
```

```
Program 1 Symbol Table
--------------------------------------
Name  Type    Scope  Line
--------------------------------------
a     int     0      2
b     bool    0      3
c     string  1      5
```

Input:

```
{
   int a
   {
      boolean b
      a = 1
   }
   print(b)
}$
```

Output:

```
Program 2 Lexical Analysis
Program 2 Lexical analysis produced
0 error(s) and 0 warning(s)

Program 2 Parsing
Program 2 Parsing produced 0
error(s) and 0 warning(s)

Program 2 Semantic Analysis
Error: The id b on line 7 was used
before being declared.
Warning: The id a on line 2 was
declared and initialized but never
used.
Program 2 Semantic Analysis produced
1 error(s) and 1 warning(s).
```

```
Program 2 Concrete Syntax Tree
--------------------------------
< Program >
-< Block >
--[ { ]
--< Statement List >
---< Statement >
----< Variable Declaration >
-----[ int ]
-----[ a ]
---< Statement List >
----< Statement >
-----< Block >
------[ { ]
------< Statement List >
-------< Statement >
--------< Variable Declaration >
---------[ boolean ]
---------[ b ]
-------< Statement List >
--------< Statement >
---------< Assignment Statement >
----------[ a ]
----------[ = ]
----------< Expression >
-----------< Int Expression >
------------[ 1 ]
--------< Statement List >
------[ } ]
----< Statement List >
-----< Statement >
------< Print Statement >
-------[ print ]
-------[ ( ]
-------< Expression >
--------[ b ]
-------[ ) ]
-----< Statement List >
--[ } ]
-[ $ ]

Program 2 Abstract Syntax Tree
--------------------------------
< BLOCK >
-< Variable Declaration >
--[ int ]
--[ a ]
-< BLOCK >
--< Variable Declaration >
---[ boolean ]
---[ b ]
--< Assignment Statement >
---[ a ]
---[ 1 ]
-< Print Statement >
--[ b ]

Program 2 Symbol Table
not produced due to error(s) detected by
semantic analysis.
```

Input:

```
{
  int a
  {
    boolean b
    {
      string c
      {
        a = 5
        b = false
        c = "inta"
      }
      print(c)
    }
    print(b)
  }
  print(a)
}$
```

Output:

```
LEX: BEGINNING LEXING PROCESS ON PROGRAM 3...
LEX: SUCCESSFULLY COMPLETED WITH (0) errors.

PARSE: BEGINNING PARSING PROCESS ON PROGRAM 3...
PARSE: PARSE PROCESS SUCCESSFULLY COMPLETED.

CST: Printing Program 3:
 <Program>
-<Block>
--[{]
--<StatementList>
---<Statement>
----<VarDecl>
-----<Type>
------[int]
-----<ID>
------[a]
---<StatementList>
----<Statement>
-----<Block>
------[{]
------<StatementList>
-------<Statement>
--------<VarDecl>
---------<Type>
----------[boolean]
---------<ID>
----------[b]
-------<StatementList>
--------<Statement>
---------<Block>
----------[{]
----------<StatementList>
-----------<Statement>
------------<VarDecl>
-------------<Type>
--------------[string]
-------------<ID>
--------------[c]
-----------<StatementList>
------------<Statement>
-------------<Block>
--------------[{]
--------------<StatementList>
---------------<Statement>
----------------<Assign>
-----------------<ID>
-----------------[a]
-----------------[=]
-----------------<Expr>
-----------------<IntExpr>
-----------------<Digit>
--------------------[5]
---------------<StatementList>
----------------<Statement>
-----------------<Assign>
-----------------<ID>
-----------------[b]
-----------------[=]
-----------------<Expr>
-------------------<BooleanExpr>
--------------------<BoolVal>
---------------------[false]
----------------<StatementList>
-----------------<Statement>
-----------------<Assign>
-------------------<ID>
--------------------[c]
-------------------[=]
-------------------<Expr>
---------------------<StringExpr>
----------------------["]
----------------------<CharList>
-----------------------<Char>
------------------------[i]
----------------------<CharList>
-----------------------<Char>
------------------------[n]
-----------------------<CharList>
------------------------<Char>
-------------------------[t]
-----------------------<CharList>
------------------------<Char>
-------------------------[a]
------------------------[CharList]
----------------------["]
----------------[StatementList]
--------------[}]
------------<StatementList>
-------------<Statement>
--------------<Print>
---------------[print]
---------------[(]
--------------<Expr>
----------------<ID>
-----------------[c]
---------------[)]
-------------[StatementList]
----------[}]
--------<StatementList>
---------<Statement>
----------<Print>
-----------[print]
-----------[(]
-----------<Expr>
------------<ID>
-------------[b]
-----------[)]
---------[StatementList]
------[}]
----<StatementList>
-----<Statement>
------<Print>
-------[print]
-------[(]
-------<Expr>
--------<ID>
---------[a]
-------[)]
-----[StatementList]
--[}]
-[$]
```

```
SEMANTIC: STARTING SEMANTIC ANALYSIS ON PROGRAM 3.
SEMANTIC: SEMANTIC ANALYSIS SUCCESSFULLY COMPLETED ON PROGRAM 3.

AST: Printing AST for Program 3:
 <Block>
-<VarDecl>
--[int]
--[a]
-<Block>
--<VarDecl>
---[boolean]
---[b]
--<Block>
---<VarDecl>
----[string]
----[c]
---<Block>
----<Assign>
-----[a]
-----[5]
----<Assign>
-----[b]
-----[false]
----<Assign>
-----[c]
-----["inta"]
---<Print>
----[c]
--<Print>
---[b]
-<Print>
--[a]

SYMB: Printing Symbol Table for Program 3:
NAME TYPE       isINIT?   isUSED?   SCOPE
[a   int        true      true      0]
[b   boolean    true      true      1]
[c   string     true      true      2]
```