CPS2000 - Compiler Theory and Practice - Assignment Report

Daniel Cauchi

Contents

Chapter 1

Introduction

This is the report for the CPS2000 Compilers assignment, where a compiler and interpreter for the MiniLang programming language was created, with a Lexer, Parser, XML generator, Semantic Analyser and Interpreter. The C++ programming language was used in conjunction with the VSCode IDE. The program was tested on Ubuntu C++11, using the provided CMakeLists.txt. It works without error and any compiler warnings were dealt with. A video showcase is also attached to show the compiler in action, with several input files to show different aspects of it working in good order.

Chapter 2

Tasks

2.1 Task 1 - Lexer

2.1.1 Explanation and Implementation

The Lexer tokenizes the input stream by taking in character by character, grouping then into ordered tokens. A token is a tuple of (Lexeme, Attribute).

For the implementation, the entire file string is traversed and a vector of tokens is created. Errors are reported by line number. For the sake of debugging future errors (not lexer related), the tokens are also given a line number variable. The reason for this is so that the file can be closed and the string discarded as soon as the lexer finishes.

The following transition groups fo the characters were chosen (This is also the Classifier Table):

1. Singular punctuation (punctuation which exists on its own): + - ( ) {}; : ,
2. Exclamation point: !
3. Fullstop: .
4. Digit: 0-9
5. Alpha: A-Z a-z _
6. Forward slash: /
7. New-line: \n
8. Asterisk: *
9. Equals: =
10. Whitespace/tab or \r: ' ' \t \r
11. Greater/Smaller than: <>
12. Other: anything else

The following diagram is the state transition diagram DFA. Note: 'not X' means any other character which is not X.

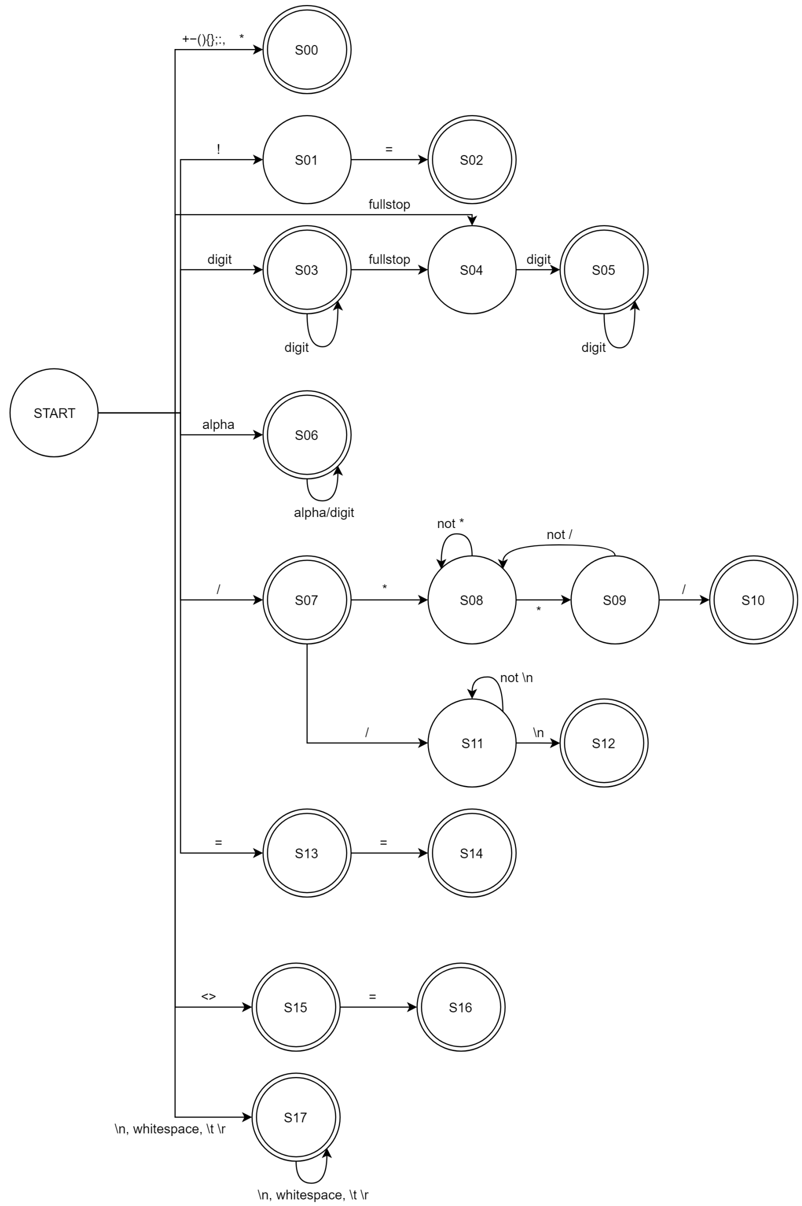

Figure 2.1: DFA for Lexer

As a table, it looks like this in the code. The comments help to understand it better [1]:

```
// List of states. STA is the start state and ERR is the error state
enum State{
  STA, S00, S01, S02, S03, S04, S05, S06, S07, S08, S09, S10, S11, S12, S13, S14, S15, S16, S17, ERR
};
//
State transitionTable[19][12] = {
  /*          0         1    2    3      4      5    6    7    8    9            10   11
  |||||||     +-(){};:, !    .    digit  alpha  /    \n   *    =    space/tab\r  <>   other*/
  /*00 STA*/ {S00,      S01, S04, S03,   S06,   S07, S17, S00, S13, S17,         S15, ERR},
  /*01 S00*/ {ERR,      ERR, ERR, ERR,   ERR,   ERR, ERR, ERR, ERR, ERR,         ERR, ERR},
  /*02 S01*/ {ERR,      ERR, ERR, ERR,   ERR,   ERR, ERR, ERR, S02, ERR,         ERR, ERR},
  /*03 S02*/ {ERR,      ERR, ERR, ERR,   ERR,   ERR, ERR, ERR, ERR, ERR,         ERR, ERR},
  /*04 S03*/ {ERR,      ERR, S04, S03,   ERR,   ERR, ERR, ERR, ERR, ERR,         ERR, ERR},
  /*05 S04*/ {ERR,      ERR, ERR, S05,   ERR,   ERR, ERR, ERR, ERR, ERR,         ERR, ERR},
  /*06 S05*/ {ERR,      ERR, ERR, S05,   ERR,   ERR, ERR, ERR, ERR, ERR,         ERR, ERR},
  /*07 S06*/ {ERR,      ERR, ERR, S06,   S06,   ERR, ERR, ERR, ERR, ERR,         ERR, ERR},
  /*08 S07*/ {ERR,      ERR, ERR, ERR,   ERR,   S11, ERR, S08, ERR, ERR,         ERR, ERR},
  /*09 S08*/ {S08,      S08, S08, S08,   S08,   S08, S08, S09, S08, S08,         S08, S08},
  /*10 S09*/ {S08,      S08, S08, S08,   S08,   S10, S08, S08, S08, S08,         S08, S08},
  /*11 S10*/ {ERR,      ERR, ERR, ERR,   ERR,   ERR, ERR, ERR, ERR, ERR,         ERR, ERR},
  /*12 S11*/ {S11,      S11, S11, S11,   S11,   S11, S12, S11, S11, S11,         S11, S11},
  /*13 S12*/ {ERR,      ERR, ERR, ERR,   ERR,   ERR, ERR, ERR, ERR, ERR,         ERR, ERR},
  /*14 S13*/ {ERR,      ERR, ERR, ERR,   ERR,   ERR, ERR, ERR, S14, ERR,         ERR, ERR},
  /*15 S14*/ {ERR,      ERR, ERR, ERR,   ERR,   ERR, ERR, ERR, ERR, ERR,         ERR, ERR},
  /*16 S15*/ {ERR,      ERR, ERR, ERR,   ERR,   ERR, ERR, ERR, S16, ERR,         ERR, ERR},
  /*17 S16*/ {ERR,      ERR, ERR, ERR,   ERR,   ERR, ERR, ERR, ERR, ERR,         ERR, ERR},
  /*18 S17*/ {ERR,      ERR, ERR, ERR,   ERR,   ERR, S17, ERR, ERR, S17,         ERR, ERR}
};
```

Figure 2.2: Table for Lexer

If the program goes to ERR (the error state), then the lexeme is over. If the current state is not a final state, then an error is reported and Lexical Analysis stops. If if the program goes to ERR and is in a final state, then the lexeme is kept and tokenized.

The token class is the following (Found in the Token.h file):

```
class Token{
  public:
    static string TokenString[];
    // the Token parameters
    TokenType type;
    string lexeme = "";
    float number;
    int lineNumber;

    // constructor
    Token (TokenType _type, string _lexeme, float _number, int
  _lineNumber){
      type = _type;
      lexeme = _lexeme;
      number = _number;
      lineNumber = _lineNumber;
    }

    // helper method to neatly print the current token
```

```
    void printToken();
};
```

The final states result in some token. The following is a list of which states can result to which tokens (based on the lexeme).

1. S0: PLUS, MINUS, OPEN_BRACKET, CLOSED_BRACKET, OPEN_BRACE, CLOSED_BRACE, COLON, SEMI_COLON, COMMA, TIMES
2. S2: NE
3. S3: INT
4. S5: FLOAT
5. S6: ID (or one of mykeywords[] below)
6. S7: DIVISION
7. S11: discard '/*' and '*/' and return COMMENT
8. S13: discard '//' and return COMMENT
9. S14: EQ
10. S15: EQQ
11. S16: GT, ST
12. S17: GE, SE
13. S18: discard, since it is white spaces or new lines or tabs or carriage returns only

The array of keywords in Token.cpp is as follows:

```
// Used to check if a given string is a keyword (or identifier), and what
    type of keyword it is
struct keyword_token{
  string text;
  Token::TokenType tok_type;
};

keyword_token my_keywords[] = {
  {"and", AND},
  {"or", OR},
  {"not", NOT},
  {"if", IF},
  {"else", ELSE},
  {"for", FOR},
  {"while", WHILE},
  {"fn", FN},
  {"return", RETURN},
  {"bool", TYPE_BOOL},
  {"float", TYPE_FLOAT},
  {"int", TYPE_INT},
  {"var", VAR},
  {"true", BOOL},
```

```
    {"false", BOOL}
    {"print", PRINT}
};
```

The Lexer's *Lex()* function does the following:

1. Initialise state to the start state
2. Initialise lexeme to the empty string
3. Until EOF:
4. Read next character from the file
5. See to which column in the table this character points to, and use the current state as the row value to go to the new state
6. If at an error state
7. Check if the the lexeme is valid with the previous state before the error state. If it is invalid, report the error, otherwise append the new token to the token vector, reset the lexeme and the state and go to step 3
8. else set the state as the one obtained from step 5 as the next state and append the character to the lexeme. Go back to step 3

2.1.2 Example Test

Using the *printToken()* in the *Token* class *printTokens()* method in the *Lexer* class, when the following input is given to the program:

```
someid true false 12.3 .4 56 89
< <= > >= == != and or not
= + - * /
if else for while fn return bool float int var
: ; ,
() {}
// hello world
/* hello
world
2 */
```

The following output is printed from *printTokens()*

```
<someid, ID, 0>
<true, BOOL, 0>
<false, BOOL, 0>
<12.3, FLOAT, 12.3>
<.4, FLOAT, 0.4>
<56, INT, 56>
<89, INT, 89>
<<, ST, 0>
<<=, SE, 0>
<>, GT, 0>
<>=, GE, 0>
<==, EQQ, 0>
<!=, NE, 0>
<and, AND, 0>
<or, OR, 0>
<not, NOT, 0>
<=, EQ, 0>
<+, PLUS, 0>
<-, MINUS, 0>
<*, TIMES, 0>
</, DIVISION, 0>
<if, IF, 0>
<else, ELSE, 0>
<for, FOR, 0>
<while, WHILE, 0>
<fn, FN, 0>
<return, RETURN, 0>
<bool, TYPE_BOOL, 0>
<float, TYPE_FLOAT, 0>
<int, TYPE_INT, 0>
<var, VAR, 0>
<:, COLON, 0>
<;, SEMI_COLON, 0>
<,, COMMA, 0>
<(, OPEN_BRACKET, 0>
<), CLOSED_BRACKET, 0>
<{, OPEN_BRACE, 0>
<}, CLOSED_BRACE, 0>
<hello world, COMMENT, 0>
<hello
world
2, COMMENT, 0>
```

Figure 2.3: Example output for the example input

2.2 Task 2 - Parser

2.2.1 Explanation and Implementation

The parser takes the tokens and puts them into a parse tree based on the grammar. The following is the grammar re-written to make terminals and non terminals more visible. Terminals are underlined while non-terminals are enclosed in brackets. Square brackets represent optional parts while curly braces represent parts which may be repeated. The peeking method is used whenever there is the use of options(sub-bullet points, which represent |), [] or {}, this is done in order to **avoid back-tracking**. The parser uses something close to the FIRST set in order to peek as much as it needs to, so as to determine the next production to choose. Note: An identifier non-terminal was created due to 'Factor', since it accepts a node, so ID was enclosed within a non-terminal.

- Type

  TYPE_FLOAT

  TYPE_INT

  TYPE_BOOL

- Literal

  FLOAT

  INT

  BOOL

- Identifier

  ID

- MultiplicativeOp

  TIMES

  DIVISION

  AND

- AdditiveOp

  PLUS

  MINUS

  OR

- ReltionalOp

  ST

  GT

  EQQ

  NE

  SE

  GE

- ActualParams

  (Expression) { COMMA (Expression) }

- FunctionCall

  (Identifier) OPEN_BRACKET [(ActualParams)] CLOSED_BRACKET

- SubExpression

  OPEN_BRACKET (Expression) CLOSED_BRACKET

- Unary

  MINUS (Expression)

  NOT (Expression)

- Factor

  (Literal)

  (Identifier)

  (FunctionCall)

  (SubExpression)

  (Unary)

- Term

  (Factor) { (MultiplicativeOp) (Factor) }

- SimpleExpression

  (Term) { (AdditiveOp) (Term) }

- Expression

  (SimpleExpression) { (RelationalOp) (SimpleExpression) }

- Assignment

  (Identifier) EQUALS (Expression)

- VariableDecl

  VAR (Identifier) COLON (Type) EQ (Expression)

- PrintStatement

  PRINT (Expression)

- ReturnStatement

  RETURN (Expression)

- IfStatement

  IF OPEN_BRACKET (Expression) CLOSED_BRACKET (Block) [ELSE (Block)]

- ForStatement

  FOR OPEN_BRACKET [(VariableDecl)] SEMI_COLON (Expression) SEMI_COLON [(Assignment)] CLOSED_BRACKET (Block)

- FormalParam

  (Identifier) COLON (Type)

- FormalParams

  FormalParam { COMMA (FormalParam) }

- FunctionDecl

  FN (Identifier) OPEN_BRACKET [(FormalParams)] CLOSED_BRACKET COLON (Type) (Block)

- Statement

  (VariableDecl) SEMI_COLON

  (Assignment) SEMI_COLON

  (PrintStatement) SEMI_COLON

  (IfStatement)

  (ForStatement)

  (ReturnStatement) SEMI_COLON

  (FunctionDecl)

  (Block)

- Block

  OPEN_BRACE { Statement } CLOSED_BRACE

- Program

  { (Statement) }

Since it is an Abstract Syntax tree, the following tokens are matched and immediately discarded by the parser: OPEN_BRACKET, CLOSED_BRACKET, OPEN_BRACE, CLOSED _BRACE, SEMI_COLON, COLON, COMMA, FN, VAR, RETURN, IF, ELSE, FOR, PRINT

From the above, it is shown that there are 25 possible types of nodes. This is because the tree is not fully abstract. The advantage of this is that operator precedence for expressions is handled automatically by the grammar, while the disadvantage is that it leads to more complexity, since each type of node needs to be catered for.

Regarding implementation, the *Parser* class receives the tokens and stores them in a *TokenManager* so that they are managed by the functions in this manager. The *Parser* class also holds the root node of the recursive AST which will be generated when the *parse()* method is called.

The 26 ASTNodes classes (27 counting the abstract class used for polymorphism) each have a parse method to parse their own form accordingly, and store what is parsed in their own way. For example, the *Type* ASTNode, will store a token whose type is TYPE_FLOAT, TYPE_INT or TYPE_BOOL. Meanwhile, a program will contain a list (vector) of statements.

Each parse method for the ASTNodes will return true or false, to tell the callback recursion whether it was successful or not. Since a predictive parser with follow sets is used, then all should return true, and if one is false, then parsing fails and the **error and the line number** is reported to the user. The function *match(TokenType)* is used to check if token is matched. It is used especially in cases where a bracket needs to be matched. If the required token type is not found, then the error that the required token was not found is reported to the user and the parser exits.

Taking the following parse method for ASTNodeIdentifier:

```
class ASTNodeIdentifier : virtual public ASTNode{
  public:
    // costructor is same as parent
    ASTNodeIdentifier(TokenManager *tokenManager) : ASTNode(tokenManager)
     {};
    virtual ~ASTNodeIdentifier(){};
    virtual bool parse(); // returns true if parse was successful
    virtual void accept(Visitor *v);

    Token* token;
};
```

```
bool ASTNodeIdentifier::parse(){
  token = match(ID);
  return true;
}
```

The process is that the token is simply stored within the variable *token*, and it becomes a leaf of the tree (all leaves of the tree are tokens, while the internal nodes are ASTNodes).

Taking a more complex example of the ASTNodeIfStatement:

```
class ASTNodeIfStatement : virtual public ASTNode{
  public:
    // costructor is same as parent
    ASTNodeIfStatement(TokenManager *tokenManager) : ASTNode(tokenManager) {};
    virtual ~ASTNodeIfStatement();
    virtual bool parse(); // returns true if parse was successful
    virtual void accept(Visitor *v);

    ASTNode* expression;
    ASTNode* block;
    ASTNode* elseBlock = NULL; // optional
};
```

```
bool ASTNodeIfStatement::parse(){
  match(IF);
  match(OPEN_BRACKET);

  ASTNode *n = new ASTNodeExpression(tokenManager);
  if (n->parse() == false) return false;
  expression = n;

  match(CLOSED_BRACKET);

  ASTNode *n2 = new ASTNodeBlock(tokenManager);
  if (n2->parse() == false) return false;
  block = n2;

  if(tokenManager->peekToken()->type == ELSE){
    match(ELSE);

    ASTNode *n3 = new ASTNodeBlock(tokenManager);
    if (n3->parse() == false) return false;
    elseBlock = n3;
  }

  return true;
}
ASTNodeIfStatement::~ASTNodeIfStatement(){
  delete expression;
  delete block;
  delete elseBlock;
}
```

The first thing to notice is that an IF statement has three children and none of them are leaves. The parse method starts by first matching an IF and OPEN_BRACKET and discards them (lines 2 and 3), afterwards it tries to parse an ASTNodeExpression(lines 5 and 6). If it fails, the entire process returns false (line 6). Otherwise, if it is successful, the expression node within the if node is set to the newly parsed expression node. This shows how the parse method is a recursive function, until leaves are found. The rest of the method follows suit. Note how the else block is optional, so first the parser checks if it should expect an else block by checking for an ELSE token (line 15), and if it does not exists, it is skipped over, otherwise it is set accordingly.

2.2.2 Example Test

The parse method returns true or false, and it is difficult to visualise the tree, so instead a visualization of this process can be shown in the next section XMLGeneration, for a proper visualization of the tree.

2.3 Task 3 - XML Generation

2.3.1 Explanation and Implementation

The visitor design pattern is used for the XML generator. Each ASTNode accepts the visitor class's *visit* function through the *accept* function.

An XMLVisitor class was created. This contains a string stream called *xml* which will contain the generated string after the tree is all visited. The *numberOfTabs* integer holds the number of indentation number which should be applied at each line. Note: The visit methods are of type *void** in order to cater for further visitors. For this visitor however, they all return NULL, or rather, 0.

```
class XMLVisitor : virtual public Visitor{
  public:
    XMLVisitor(){};
    virtual ~XMLVisitor(){};
    virtual void *visit(ASTNode*){};
    virtual void *visit(ASTNodeType *n);
    virtual void *visit(ASTNodeLiteral *n);
    virtual void *visit(ASTNodeIdentifier *n);
    virtual void *visit(ASTNodeMultiplicativeOp *n);
    virtual void *visit(ASTNodeAdditiveOp *n);
    virtual void *visit(ASTNodeRelationalOp *n);
    virtual void *visit(ASTNodeActualParams *n);
    virtual void *visit(ASTNodeFunctionCall *n);
    virtual void *visit(ASTNodeSubExpression *n);
    virtual void *visit(ASTNodeUnary *n);
    virtual void *visit(ASTNodeFactor *n);
    virtual void *visit(ASTNodeTerm *n);
    virtual void *visit(ASTNodeSimpleExpression *n);
    virtual void *visit(ASTNodeExpression *n);
    virtual void *visit(ASTNodeAssignment *n);
    virtual void *visit(ASTNodeVariableDecl *n);
    virtual void *visit(ASTNodeReturnStatement *n);
    virtual void *visit(ASTNodeIfStatement *n);
    virtual void *visit(ASTNodeForStatement *n);
    virtual void *visit(ASTNodeFormalParam *n);
    virtual void *visit(ASTNodeFormalParams *n);
    virtual void *visit(ASTNodeFunctionDecl *n);
    virtual void *visit(ASTNodeStatement *n);
    virtual void *visit(ASTNodeBlock *n);
    virtual void *visit(ASTNodeProgram *n);
    void trimXMLNewLines(); // remove empty lines from xml
    string getXML(){ return xml.str(); }
  private:
    stringstream xml;
    unsigned int numberOfTabs = 0;
    string tabsString();
};
```

An XML visit for a leaf node accept would look something like this:

```
void *XMLVisitor::visit(ASTNodeMultiplicativeOp *n){
  xml << "OP=\"" << n->token->lexeme << "\"";
}
```

The above is for a multiplicative operator. So ’*’ would be shown as ’OP=”*”’.

This is also a recursive method, so the program node (which contains a list of statements), iteratively goes trough the statements and calls the accept statement for the visitor on them as well, creating recursion.

```
void *XMLVisitor::visit(ASTNodeProgram *n){
  for(int i = 0; i < n->statements.size(); ++i){
    n->statements.at(i)->accept(this);
  }
}
```

After finishing, the xml string stream can be either outputted to the screen or stored in a separate file.

2.3.2 Example Test 1

Parsing the following as input(parsing as a program node):

```
x = 1+2*4;
```

The following output is produced by the xml generator:

```
<Assign>
  x</ID>
  BinExprNode OP="+">
    <IntConst>1</IntConst>
    <BinExpr OP="*">
      <IntConst>2</IntConst>
      <IntConst>4</IntConst>
    </BinExpr>
  </BinExprNode>
</Assign>
```

Note how operator precedence is kept.

In order to change precedence, enclose the addition in brackets:

```
x = (1+2)*4;
```

So the following output is now produced:

```
<Assign>
  x</ID>
  <BinExpr OP="*">
    BinExprNode OP="+">
      <IntConst>1</IntConst>
      <IntConst>2</IntConst>
    </BinExprNode>
    <IntConst>4</IntConst>
  </BinExpr>
</Assign>
```

2.3.3 Example Test 2

Putting an entire program as input this time:

```
var x : float = 0;

fn y(g:bool) : int{
  return z;
}

h = g();
```

The output by the XML generator is this:

```
<VarDecl>
  <Var Type = "float">x</ID>
  <IntConst>0</IntConst>
</VarDecl>
<FuncDecl>
  <FN Type = "int">y</ID>
  <F_Param> g</ID>:Type = "bool" </F_Param>
  <Return>
    z</ID>
  </Return>
</FuncDecl>
<Assign>
  h</ID>
  <FN_CALL FN=g</ID>"
  </FN_CALL FN>
</Assign>
```

2.4 Task 4 - Semantic Analysis

2.4.1 Explanation and Implementation

The task of the semantic analysis is to perform type checking and scope checking by traversing the AST and making the required checks at each node.

In order to achieve this, a new visitor inherited class was created called *SAVisitor*. It contains the following functionality and data in order to perform its job (as well as the visit methods).

```
 vector<map<string, TokenType>> scope;
void newScope(); // add scope as the 0 index of the vector
void insert(string, TokenType); // in current scope
void removeScope(); // remove scope at position 0
// set as pointer to TokenType due to the need to make it return null
TokenType* lookup(string); // lookup starting from vector 0 and going down
// a map, mapping function names to their parameter types
map <string, vector<TokenType> > functions;
TokenType *currentFunctionType = nullptr;
int lineNumber = 0;

// Methods to help determine if a function has a proper return statement in all paths
bool insideFor = false; // do nothing while inside for
bool insideFunction = false; // only applies if inside function
vector<bool> ifsReturn;
int ifsReturnIndex = -1;
bool goodReturn;
```

The scope vector is treated like a stack, but created as a vector to be iterated over easily. New entries are inserted at the front (position 0) of the vector and same thing with deletions. When iterating, iteration also starts from the front, so new entries cover shadow old ones (scope shadowing). The scope together with the functions map, together make the symbol table.

Next, the visit classes are discussed. Each one of these, although described as a *void**, has some form of concrete return value, and not all of them are the same. The functionality of each method and their return type is discussed below, for each type of ASTNode visit method:

- Type : Type

  returns FLOAT, INT or BOOL(the type) depending on its token

- Literal : Type

  returns the type of its literal value

- Identifier : string (the lexeme of the identifier token)

  returns name of the identifier. Lookup is handled by parent node

- MultiplicativeOp : MultOp Type (ex. TIMES, AND)

  returns the operator itself, so that the expression node it belongs to can check that the operator supports the types it is operating upon. (Ex. "7*8" is valid, but "7 and 8" is not)

- AdditiveOp : AddOp Type (ex. PLUS, OR)

  very similar to MultiplicativeOp but with different operators

- RelationalOp : RelOp Type (ex. EQQ, ST)

  very similar to MultiplicativeOp but with different operators

- ActualParams : Vector of Type

  iterates through all the parameters and returns a vector of all their types, so they can be matched with the formal parameters by the FunctionDecl node.

- FunctionCall : Type

  validates the function exists

  validates the parameters provided are of the required type

  If any of these are false, the required errors are reported and program exits

- SubExpression : Type

  returns the type of the expression it has

- Unary : Type

  validates that the unary operator is applied on the proper type and returns the type of the expression

- Factor : Type

  returns the type of its node

- Term : Type

  validates that any operators are being used properly and are of the correct type

- SimpleExpression : Type

  validates that any operators are being used properly and are of the correct type (ex. "4*6.8" returns float, while 9*true gives an error)

  very similar to Term but with additive operators instead

- Expression : Type

  very similar to Term and Simple but with relational operators instead

- Assignment : void

  validates the identifier exists

  validates the identifier's type and type of expression match

- VariableDecl : void

  makes sure that the types of the given type and expression are not conflicting

  adds the identifier to the symbol table, with the given type. Error if it already exists

- PrintStatement : void

  verify the given expression is correct

- ReturnStatement : void

  makes sure that the type of the expression also matches the type of the current function it is within by using the *currentFunctionType* variable.

- IfStatement : void

  validates that expression is of type bool

  start new scope

  validates block

  pop scope

  validates else block if it exists (while creating and popping scopes)

- ForStatement : void

  creates new scope

  Performs and validates the variable declaration

  validates that expression is of type bool

  validates assignment

  validates block

  pops scope

- FormalParam : Type

  put each paramater in the current symbol table

  returns type of parameter

- FormalParams : Vector of Type

  iterates trough each FormalParam declaration adding them to the vector

- FunctionDecl : void

  add function to the functionParams map after starting new scope and validating the FormalParams(Note: they are added in the scope within their own methods)

  set the currentFunctionType to the type of the function so ReturnStatement can know Add function to symbol table after validating size of symbol table is one (outer scope)

  validate the Block

  close scope

- Statement : void

  validate the statement

  if block, then start new scope

- Block : void

  validate the list of statements

- Program : void

  validate the list of statements. Starts with function declarations.

Note: scopes were not started and ended in the 'Block' node because of function declarations.

Semantic Analysis Features

This section is to discuss some features including additional extras in the semantic analyser implementation

The first feature is that error reporting includes precise errors as well as the included line number where the error occurred. This was made by having a global line number variable and updating it every time a node with a token is visited.

Second extra feature is that type conversion from integers to floats is made available. So float types can accept integers, however integers cannot accept floats. This was made trough type checking.

The program is allowed to exit abruptly, hence why return statements can be accepted from outside a function.

A function declaration is checked to make sure **all code paths return a proper value**. This means that if a return exists in an if, but no corresponding return in an else or outside the if statement exists, the function is declared semantically incorrect. Returns in for loops are also not considered to be enough for a function to be valid for the semantic analyser. This was made possible trough the use of a stack and a few global variables, in conjunction with checks when returns are made inside of functions.

Important to note is that function names cannot be overridden anywhere in the code.

Functions can only be defined in the outer scope

Functions can be used before they are declared, because their declaration is performed first. Hence no other variable anywhere in the program may use the function's name.

2.4.2 Example Tests

Given the following code, the semantic analyser returns that it is valid

```
print 5;
```

However, while the parser can successfully parse the following, the semantic analyser does not find x in the table:

```
return x;
```

So the semantic analyser complains by saying:

```
Identifier x at line 1 does not exist
```

The following is a function declaration, note how it is defined badly, because if the code goes to the scope where the comment resides, then it would return null, which is not a supported type

```
fn u():bool{
  if(4>6){
    return true;
  }else{
    if(5>6){
      return true;
    }else{
      //return true;
    }
  }
}
```

and the compiler complains:

```
Function u at line number 1: not all code paths return a value
```

This could be remedied by uncommenting the comment.

More items which are checked are correct type and amount of parameters to functions as well as that return statements are of the correct type when inside a function.

2.5 Task 5 - Interpretation

2.5.1 Explanation and Implementation

The interpreter is to execute the program line by line and report any run time errors if there are any. The symbol table is regenerated and this time it will contain identifiers and their contents besides just their types. This content will also be changed as the program executes.

A new visitor class was created called *IVIsitor*, which contains the following items:

```
vector<map<string, ValueType>> scope;
void newScope(); // add scope as the 0 index of the vector
void insert(string, ValueType); // in current scope
void removeScope(); // remove scope at position 0
ValueType lookup(string); // lookup starting from vector 0 and going down
// a map, mapping function names to wherever the function is
map <string, ASTNode*> functions;
int lineNumber = 0;

bool performFunction = true;
vector<ValueType>
bool returnFromFunction = false;
ValueType returnValue;
```

ValueType is a struct which was created for this visitor:

```
struct ValueType{
  void* value;
  TokenType type; //bool, int or float (needed for conversion)
};
```

performFunction is used to determine whether a function is currently being declared or performed. *parameters* is used to store the values of the parameters as they are sent to a function. *returnFromFunction* is set to true when a return is found, and is not set back to false until a function call ends (or the program terminates). While it is true, no statement or block is executed. *returnValue* is set to something by the return statement, nd set back to null after a function returns.

Next, the functionality of each node is discussed:

- Type : TokenType

  returns FLOAT, INT or BOOL(the type) depending on its token

- Literal : Value

  The value of the literal is returned

- Identifier : string (the lexeme of the identifier token)

  returns name of the identifier. Lookup for value is handled by parent node

- MultiplicativeOp : MultOp Type (ex. TIMES, AND)

  returns the operator itself, so that the expression node it belongs to can use it to operate on the values

- AdditiveOp : AddOp Token (ex. PLUS, OR)

  Very similar to MultiplicativeOp but with Additive operators

- RelationalOp : Type (ex. EQQ, ST)

  Very similar to MultiplicativeOp but with Relational operators

- ActualParams : Vector of Values

  gets value from each parameter and returns their vector

- FunctionCall : Value

  sets *performFunction* to true

  sets the *parameters* value to the values returned from the actual parameters

  gets function from *functions* map and traverses it until a return is found

  resets the parameters value

  resets the *returnValue* variable and returns it's previous value

- SubExpression : Value

  returns value of the containing expression

- Unary : Value

  applies the unary operator

  returns the value after application

- Factor : Value

  returns value of containing factor

  if it is an ID, looks it up in scope table

- Term : Value

  applies the operator

  returns the value after application

- SimpleExpression : Value

  applies the operator

  returns the value after application

- Expression : Value

  applies the operator

  returns the value after application

- Assignment : void

  change value of item in symbol table

- VariableDecl : void

  Add item to symbol table

- PrintStatement : void

  print given expression to screen (including a new line)

- ReturnStatement : void

  sets the *returnValue* variable to the expression

  sets the *returnFromFunction* variable to true, so that nothing is computed until the program exits the function

- IfStatement : void

  computes the given expression

  if true, opens new scope, visits the if-block, closes scope

  if false, and an else-block exists, then opens new scope, the else block is visited, closes scope

- ForStatement : void

  performs variable declaration

  start new scope, visit the block, end scope

  performs variable assignment

  checks expression and exits if it is false, otherwise repeats block

- FormalParam : string

  returns name of parameter

- FormalParams : Vector of strings

  returns all the names of the parameters

- FunctionDecl : void, Value

  If *performFunction* is false, simply add the function to the *functions* map

  If *performFunction* is true, open new scope, assign the variables to the given parameters, traverse trough function block, close block, set *returnFromFunction* to false, return the returnValue.

- Statement : void

  visits the statement

  if is is a block, start ne scope, visit block, end scope

- Block

  unless *returnFromFunction* is set to true, iterate trough the statements, until *returnFromFunction* is set to true

- Program

  iterates trough all statements until the end, or until *returnFromFunction* is set to true

Some thing to note about the interpreter:

- The only runtime error that can occur is division by zero, which is reported to the user if it is found.
- The program itself can return a value, if a return statement is found outside a function declaration.
- 3/2 is 1.5, not 1, because everything (including booleans) is treated as a float. However, if the statement:

  $$\text{var x : int} = 3/2$$

  is found, it is valid but x is set to 1. This approach was taken to allow as much functionality as possible, while avoiding certain ambiguities like how C/C++ says that 3/2 is 1, so you'd need to instead write 3.0/2 or 3/2.0 to get the proper 1.5.

2.5.2 Example Tests

For this section please take a look at the showcase video uploaded in the zip file. It includes examples with errors and their corresponding error reporting and fixes, as well as more concrete examples, with proper outputs.

Chapter 3

Conclusion

This concludes the report for the CPS2000-Compilers Assignment. For further information view the attached readme to get info on directory structure and how to use the compiler. The video showcase includes some examples which show the compiler's functionality and features in action.

Bibliography

[1] "Programming a state transition table," 9 1998. Available at: http://teaching.idallen.com/cst8152/98w/prog_trans_tbl.html.