University of Lethbridge

Technical Documentation of Compiler construction (Upto phase-III)

Lazima Ansari(001188187)
Ahamad Imtiaz Khan(001188188)

March 31, 2016

Scope and Type Check:

Type and scope checking is the third phase of compiler construction. In syntax analysis phase parser checks that if the tokens are arranged in a syntactically correct form or not. In this phase we look deeper to check that whether the tokens form a sensible set of instructions. It is also called semantic analysis. For example English language has structure. It consists of subject and predicate. Correct sequence of these make a semantically correct sentence. If we want our PL to be semantically valid all of our variables must be properly defined and types of the variables must be valid. Semantic analysis is the last phase to wipe out errors from source code. Before going to code generation phase we want to make it sure. We perform two types of checking in semantic analysis.

- Scope check
- Type check

Before describing scope and type check we would like to discuss about block table which is used in both scope and type checking.

Block Table

- Associated Files: blocktable.h blocktable.cc

Our Block table is actually a vector of vectors. Rows of the vector represent individual blocks of source code and row consists of several vectors which are actually TableEntry of the variables. blocktable.h is the interface of the class in which Block table is implemented and the implementation is done inside blocktable.cc. Brief description of the important methods of this class are given below.

bool search(int) : This method returns true if the id we are looking for is found in the current block and returns false otherwise.

bool define(int, PL_Kind, PL_Type, int, int): This method creates new entry in block table based on blockTable variable. It returns true if the current block doesn't contain an object with our current id after creating an entry. Otherwise it returns false indicating ambiguous name i.e there is already a definition for the specific id in the current block. When we create an entry we insert an structure in Block table. The structure is given below so that we can understand what type of information we store in Block Table.

```
typedef struct
{
int id;//position of name in symbol table
PL_Kind kind;// kind of name
PL_Type type;// data type of name
int size; // size of name
int value; // numerical value
}TableEntry;
```

TableEntry find(int, bool&): Using this method we search the entire table for an id in inside out fashion. Variable error is false if the id we are looking for is in the block table. error is true if not found in the Block table.

Now come to scope and type check:

Scope check: • Associated Files: parser.h parser.cc
No new file is needed for scope check. We just need to modify some methods inside parser class. We know that there are two parts inside a block of our PL. They are definitionPart and statementPart. definitionPart has some sub divisions like constantDefinition, variableDefinition, procedureDefinition and each of them has corresponding methods. Before returning from these methods we create entries for constantName, variableName and procedureName in block table. While creating an entry we set appropriate kind and type of these names. While creating an entry we also check that if the name is already inserted in the current block. If it's there then we show ambiguous name error.
In statementPart we simply check if the name is defined or not and we also check whether the name is of correct kind. For example inside procedureStament when we match procedureName we check its existence in block table, if it exists we check if it's kind is PROCEDURE. If one the the two condition is false we show error message.

Type check : • Associated Files: parser.h parser.cc
Our PL is quite simple. There are only two types in our PL. They are integer and Boolean. In definitionPart we set the types of the individual names. For example inside method variableDefinition(vector<Symbol>) when typeSymbol(vector<Symbol>) method is called we get the type of our variableName. Previously the return type of typeSymbol(vector<Symbol>) was void now we changed it to PL_Type so after calling this method we get the type for variableName. We did this and similar kind of modifications in some other methods of the parser that makes sure that the source code semantically correct or shows error if not. Other than definitionPart we modified numbers of methods of statementPart for type checking. For example

guarderCommand = expression '->' statementPart

Here expression must be of type BOOLEAN. Method that corresponds to expression has now return type PL_Type and when expression calls primaryExpression and inside primaryExpression if relationaOperatior is called after simpleExpression we know that the type of the primaryExpression is Boolean so it returns BOOLEAN (otherwise INTEGER and return type of primary expression is also changed to PL_Type). Finally expression method returns BOOLEAN.
For fulfilling this purpose we modified 17 methods of our parser so that it can perform type checking.

matchName(Symbol , vector<Symbol>): We have declared this new method only for matching names. The purpose is quite similar to our match(Symbol sym , vector<Symbol> stops) function. The difference after matching the name it returns the position of the name in symbol table which is stored as id in TableEntry in block table.

Syntax Analysis:

Syntax analysis is the second phase of compiler construction. Lexical analyzer processes stream of individual characters to token. Syntax analyzer or parser takes stream of tokens as input and with the help of Context Free Grammer (CFG) parse the stream of tokens we get from source file. Context free grammar is made up of four components:
• A set of non-terminals : Generate grammar.
• A set of terminals : Basic symbols or set of tokens.
• A set of productions : Grammars which consists of terminals and non-teminals. left side of a production is always non-terminal and right side is sequence of terminals and/or non-terminal that can be deduced form the left side.
• One non-terminal which is the start symbol from where the production starts.

Parser construction detail:

Based on major functionality, parser construction can be divided into three parts.

1. Recursive descent parsing
2. First and Follow set computation
3. Error Recovery

Before jumping into the parser we want to mention one thing that we changed in previous project file. There are three types of identifiers in the CFG of our PL but previously we had only one type of symbol ID for identifiers. Now, in parser an ID can be of variable name type or constatnt name type or procedure name type. If we don't resolve this problem then some productions become ambiguous for example, factor = constant | variableAccess. First of constant contains constantName and first of variableAccess contains variableName. So when we are at factor we must decide which production will we use next. So we must be able to differentiate. That's why we added an attribute for ID symbols which is idType.

• If idType = 1 then it is a variableName.
• If idType = 2 then it's a constantName.It occurs after CONST symbol.
• If idType = 3 then it's a procedureName.It occurs after PROC symbol.

When these cases occurs we set the idType of those id in symbol table.

Steps of syntax analysis:

a) Recursive descent parsing:

• Associated Files: parser.cc, parser.h

The main portion of Recursive Descent Parser is implemented with the help of these files. parser.h is the interface of the parse class and we implemented the methods inside parser.cc.

The implementation of grammar rules are quite self explanatory. Our parser is recursive descent parser. For every non-terminal we have a method. We implemented the PL grammar from Brinch hansen book's Appendix B section. The grammar is ready for LL(1) parsing because for any look ahead token there is at most one production. We didn't find any case in which the grammar is ambiguous (as we somehow resolved the case identifier issue). Here is an example how we implemented the recursive descent parser. We have a non-terminal block and the method for this non-terminal is block(). The production rule for this non-terminal is :

block = 'begin' definitionPart statementPart 'end'.

Here 'begin' and 'end' are terminal symbols and definitionPart, statementPart are non-terminals. When we expect a terminal we match the symbol. If it does not match then we show corresponding error message. For non-terminals, we call the corresponding method. The above mentioned production rule is implemented as follows.

```
void Parser::program(Symbol sym)
{
        outFile<<"program()"<<endl;

        //Building stop set using vector
        stopSet.push_back(sym);
        stopSet.push_back(DOT);

        //non terminal bloc
        block(stopSet);

        //Building stop set using vector that may appear after dot symbol
        vector<Symbol>().swap(stopSet);
        stopSet.push_back(sym); //adding symbols in stop set

        //this portion matches dot and if not matched the shows
           corresponding error message and finds the next stop symbol
        match(DOT,stopSet);

        //parsing done
        admin.done();
}
```

In this code block first we match 'begin' symbol, if not matched then show corresponding error message. Whe we match any terminal symbol we send an additional argument which is stop set. It helps parser to recover from error. Details about this will be discussed in Error recovery part. definitionPart can be empty so we need to grab a token in advance to see whether it is in first of definitionPart or in follow of definitionPart. If it is in first of definitionPart. Then we call definitionPart(vector<Symbol>) and statementPart(vector<Symbol>) and finally match 'end' symbol. The whole recursive descent parser is implemented in same design.

b) First and Follow set computation:

- Associated Files: firstfollow.cc, firstfollow.h

Without computing first and follow set it is not possible to implement an LL(1) parser. Some productions produces several grammar rule, some productions produces emty strings. In these case we need first and follow sets. We can go to at most one state. For example:

definition = constantDefinition | variableDefinition | procedureDefinition

In this case, when we are in definition we grab a look ahead token and for this look ahead token we must decide which method should we choose. We have firstOfConstDef(), firstOfVariDef() and firstOfProcDef() methods in which returns set of first symbol in a vector for the corresponding non-terminal. Union operation for sets of symbol is needed in construction of first and follow sets. We used + operator overloading of c++ for union operation of two vectors that are actually sets of symbols.

in(vector<Symbol>) method:
this functions is inside parser.cc file. After getting the vector for any first or follow set with the help of this function we check whether our look ahead token is in the set or not. If the look ahead symbol is in the set then it returns true otherwise false. In this scheme we choose which method to call using look ahead token.

In our PL grammar two productions produces empty in right hand side. We need to compute follow only for these two non-terminals. They are definitionPart and statementPart. As guardedCommand = expression '$->$' statementPart, so follow of guardedCommand is also included in follow set of statement part. We have three methods for computing follow sets: followOfDefPart(), followOfStatePart(),followOfGuardedCommand(). Different methods were used so that we could easily handle and reuse the first and follow sets.

c) Error Recovery:

• Associated Files: parser.cc, parser.h

Error recovery is an important part of syntax analysis. As our parser is recursive descent parser if there is no error recovery the parser will exit from the program after encountering any error. But our intention is to find as many syntax error as possible. In chapter 5 of Brinch Hansen's text there are several algorithms for recovering error. First one is not good at al,l the one I mentioned above. Second one is good but not good enough. It's quite crude. We will have some fixed no. of stop symbols and when any error will occur we will start to read tokens until we have any of the stop symbols then will start parsing from that sate. It's good but crude as well because we may miss many errors in between. The third one is the best among these. For every terminal there are different sets of stop symbols that may occur after the terminal symbol. So when we try to match any terminal but fail, then we call syntaxError(vector<Symbol>) method with corresponding stop symbols.

syntaxError(vector<Symbol>) method:

This method is used for error recovery. When we try to match any terminal symbol and fail then we call this method and pass set of stop symbols in vector form. We grab a look ahead token and check if the token is in stop symbol set using in(vector<Symbol>) method. If the look ahead token is in stop set we stop there but if not then we continue to grab tokens until we reach to any of the stop symbols for the symbol which occurred error. EOF is always in stop symbol set as we must stop at EOF otherwise the parser will fall in loop. Recursive descent parser itself helps us to construct stop sets. At first when we call program(Symbol) method we pass EOF to it because EOF will occur after the whole program. Then from this base we add or keep the stop symbol set and pass it to another non-terminal function calling. From a new method we see what may occur after this. If no non terminal or terminal will occur afterwards then we keep the stop symbol as it was otherwise we add the probable symbols with previous stop symbols.

syntaxCheck(vector<Symbol>) method:

We goal for error recovery was to ensure that the parser will always be in error free state. This method with syntaxError(vector<Symbol>) method make that sure. This method checks that if our current look ahead token is in stop set or not. If it isn't then after reporting syntax error it calls syntaxError(vector<Symbol>) which actually recovers parser from error.

match(Symbol,vector<Symbol>) method:

Though the main task of this method is to match terminals but if any terminal doesn't match it call syntaxError(vector<Symbol>) method and pass the stop set as parameter which recovers the parsers from error We tried to recover from as many errors as possible but sometimes the parser might not recover properly. It may not recover if the structure of program is very bad. It will still show some errors but will not be able to parse through last line. But in most of the cases the error recovery scheme we implemented is good enough to recover from error and parse through the last line of the source file.

Error reporting is not done inside parser.cc or scanner.cc. These files detected the errors and reported to administrations.cc. Error reporting is done in this file. New line counter was also implemented here.

Lexical Analysis:

Lexical analysis or scanning is the process where the stream of characters making up the source program is read from left-to-right and grouped into tokens. Lexical analyser, which is also known as scanner development is the first phase of compiler design. The purpose of scanner is to process stream of individual characters to tokens which will be used in the next phase of the compiler. In this project we have developed a scanner that is actually a finite state machine that continues to read input stream until it reaches to the end of file and produces a list of tokens from the source file.
For building a scanner we divided our project into 4 parts. They are as follows:

- Driver
- Administrator
- Scanner
- Hash table for storing word-tokens

Brief description about the parts mentioned above is given below:

- Driver:

Associated file: plc.cc
This is the part that drives the program. It allows the user to call the scanner from command line. When user runs the executable it actually calls the main function inside plc.cc. It also show the message whether scanning is done successfully or not afterwards.

- Administrator:

Associated files: administration.h, administration.cc
This part performs the tasks that are not directly related to compiler phases.

administration.h and administration.cc: administration.h is the header file for Administration class which consists of the declaration of the private and public variables and methods. Implementation of the class is done inside administration.cc.

scan() method: The most important method inside Administration class in scan() method. From this method we continue to call method nextToken() of Scanner class until we reach to the end of file or we reach maximum number of errors. In each iteration, we can get any one of the following: i) a valid token: In this case, the attributes of the token is stored in outFile. validTok(Symbol sym) method helps us to perform the task. ii) new line : If we get a token for newline we don't store it, we simply increase the line counter which is needed for the purpose of scanner. NewLine() method helps us to perform the task. iii) an invalid token : When we get any invalid token, we write the corresponding error along with the line number in outFile.The same validTok(Symbol sym) method helps us to perform this task.

- Scanner:

Associated files: symbol.h,token.h, token.cc, scanner.h, scanner.cc.

symbol.h: We made a list of the symbols of PL and defined the symbols as enum Symbol type in this file.The symbols defined inside this file are as follows:

ID=256, NUMERAL, BADNUMERAL, BADNAME, //256-259
BADSYMBOL, BADCHAR, NEWLINE, NONAME, //260-263
ENDOFFILE, DOT, COMMA, SEMICOLON, //264-267
LEFTBRACKET,RIGHTBRACKET, AND, OR, //268-271
NOT,LESST, EQUAL, GREATERT, //272-275
LTE, GTE, PLUS, MINUS, //276-279
TIMES, DIV, MOD,LEFTP, //280-283
RIGHTP, ASSIGN, GC1,GC2, //284-287
BEGIN, END, CONST, ARRAY, //288-291 //keywords form here
INT, BOOL, PROC, SKIP, //292-295
READ, WRITE, CALL, IF, //296-299
DO, FI, OD, FALSE, TRUE //300-304
We assigned an integer number for each symbol and store this number for each symbol type instead of it's name in string.

Now we will discuss about the Token class
token.h and token.cc: token.h is the interface of token class.The implementation of the functions are inside the file token.cc.When we create any object of Token class, by defaul it creates a token of NONAME symbol type which can be done by it's default constructor.But if we want to create a token of any specific symbol then the constructor is executed and Symbol, value and lexeme for that token are stored. For storing value and lexeme we have a structure named attVal in this class. The constructors are used to set attribute values for symbol.In order to get the attribute values we have three getter functions, getSymbol(), getValue() and getLexeme().

The most important part of the scanner is implemented inside scanner.h and scanner.cc.
scanner.h and scanner.cc: scanner.h is interface of Scanner class and we implemented the methods inside scanner.cc.

Token nextToken() method:The main operation of scanning is done inside this method which we call from Administration class.The finite state machine that we mentioned about scanner is actually implemented inside this function. At first we read a character from file and based on the character, we go to the next state of our finite state machine.The nested if else code blocks are used to implement this. Let us consider the following examples of how we implemented the finite machine to detect a valid or invalid token.

Example 1:Let us assume we have read a digit from source file. So it enters inside "else if(isdigit(ch))" code block. Inside this code block we continue to read digits until we reach to a valid or invalid ending. For this purpose we just look one character ahead rather than reading it from file. peek() method helps us in this purpose. When we reach to an end of a digit we check either the numeral is valid or not. If valid then we construct a token object of NUMERAL type with it's associated value (we set lexeme to be empty string as lexeme is not needed here) and return it. Example 2:This second example is about '<' and '<=' symbol. When our read character from file is '<' we enter inside "else if(isSpecial(ch))" which is in general for all special symbols of PL and inside this code block we enter inside "else if(ch == '<')". So we have read a character. Now our symbol can be only '<' or '<=', so we look at the next character using peek() method. If the next character is '=' then it's a '<=' symbol. We read the '=' as well from file and construct a token object of LTE (less than or equal to) type otherwise if the look ahead character is not '=' then we create a token object of LESST (less than) type. Values and lexemes are not important for special symbols so value is set to -1 and lexeme is set to empty string.

We used some other helper functions which are used by nextToken() method.

Keywords and Identifiers: The pattern for keyword and identifier is almost same. All keywords have same pattern as identifier by definition. So when an identifier/keyword like pattern is detected by the finite machine then we insert the lexeme in symbol table. If its an identifier then it returns 1 or if it's a keyword then it returns the Symbol value. Value attribute for these token are not important.-1 is set for those.

printSymTable() method: This method is optional. It is used to print Symbol Table in the terminal, if user wants to.

When we detect an error in reading a line the finite machine doesn't detect any new token rather it continues to read characters from file until the look ahead character is a newline character.

- Hash table for storing word-tokens:

Associated files: symboltable.h, symboltable.cc

We implemented a hashtable for storing the word-tokens. Word-tokens are either keyword or identifier. The pattern for keyword and identifier is same so we must seperate them from each other. This purpose is served by this part.
<u>symboltable.h and symboltable.cc</u>: symboltable.h is the interface for Symboltable class.The implementation is inside symboltable.cc.
We have 17 keywords in PL. At the beginning of the execution of program (before starting scanning process) we preload the symbol table "htable" with keywords. htable is a hash

table. We have methods for searching and inserting entries in this table.
hashfn(string): It returns an index for a lexeme.
search(string) method: This method is used to determine whether a lexeme is stored in the htable or not. insert() method: It is used to insert a keyword or identifier object into htable. At first it checks whether the lexeme is already stored in the htable or not. If it is already stored then it does not store it again rather returns 1 if it is an identifier or the Symbol if it's a keyword. We don't need to store lexemes for keywords in symbol table for comparing it with the incoming lexemes. We implemented this using a simple encoding scheme. We know that our htable is already preloaded with keywords, only Symbol enum values are stored for those in htable. So during the scanning process when we get a lexeme, how we can be sure whether it is an identifier or keyword? From hashfn(string) we get an index for the lexeme. If the position is empty then the lexeme is definitely an identifier. But if the position is occupied, the lexeme can be an identifier which we already declared before in the source file or it can be a keyword or it can be a new lexeme which returns the same index from hashfn(string). First of all we need to detect whether it is a keyword or not. If the occupied position has a token object of Symbol value >= 288 then the object in occupied position is a keyword for sure as we assigned enum symbol of keywords >=288. But how can we be sure that our lexeme is a keyword. Suppose the symbol value of occupied cell is 291. We substracted 288 from 291 which is 3. Now we compare our lexeme with the string at third position of keywords array. If they match then its a keyword. Cost of searching is O(1) here. If the symbol value is less than 288 then we compare the lexeme with the lexeme stored in that position. If it matches then it is an identifier we already stored. If none of the two cases occurs then we go to the next position and start doing the same thing.

The maximum capacity of our hash table is 307. So if it gets full program will show an error message and will exit.