PES University, Bangalore

(Established under Karnataka Act No. 16 of 2013)

B.TECH. VI SEMESTER

Session : Jan-May, 2020

UE17CS354 – COMPILER DESIGN LABORATORY
MINI-PROJECT REQUIREMENTS

Max Marks : 45

Parser is the master program. Hence, the first step in building a mini compiler is to frame the grammar for your programming language. The document provides the requirements that your grammar must compulsorily handle (mentioned in individual phases explanation). Therefore, construct your grammar in a way that it handles all the aspects mentioned below.

PHASE 1 : LEXICAL ANALYSIS

- You will use flex/lex to create a scanner for your programming language.
- Your scanner will transform the source file from a stream of bits and bytes into a series of meaningful tokens containing information that will be used by the later stages of the compiler.
- When writing your scanner, you will need to handle conversions from integer and real-valued literals into integer and real-valued numeric data. That is, if you encounter the sequence of characters 3.1415E+3, while scanning the input you should convert this into a double with this value. Similarly, when seeing 137 you should convert it to an int.
- Your scanner should consume any comments from the input stream and ignore them. If a file ends with an unterminated comment, the scanner should report an error.
- All token names must start with T_(token-name).
- The following operators and punctuation characters are used by your selected Programming language (minimum set):

+	-	*	/	%	<
<=	>	>=	=	==	!=
&&	||	!	;	,	.
[	]	( )	{ }	[]	

Note that [,], and [] are three different tokens and that for the [] operator (used to declare a variable of array type), as well as the other two character

operators, there must not be any space in between the two characters. Each single character operator has a token type equal to its ASCII value, while the multi-character operators have named token types associated with them. For example, the token type for [] is T_Dims, while || has token type T_Or.

- Scanner implementation: The yylval global variable is used to record the value for each lexeme scanned and the yylloc global records the lexeme position (line number and column). The action for each pattern will update the global variables and return the appropriate token code.
- Your goal at this stage:
    - skip over white space;
    - recognize all keywords and return the correct token
    - recognize punctuation and single char operators and return the ASCII value as the token;
    - recognize two character operators and return the correct token;
    - recognize int, double, bool, and string constants, return the correct token and set appropriate field of yylval;
    - recognize identifiers, return the correct token and set appropriate fields of yylval;
    - record the line number and first and last column in yylloc for all tokens;
    - report lexical errors for improper strings, lengthy identifiers, and invalid characters
- Recording the position of each lexeme requires you to track the current line and column numbers (you will need global variables) and update them as the scanner reads the file, mostly likely incrementing the line count on each newline and the column on each token.
- For each character that cannot be matched to any token pattern, report it and continue parsing with the next character.
- If a string erroneously contains a newline, report an error and continue at the beginning of the next line.
- If an identifier is longer than the maximum (31 characters), report the error, truncate the identifier to the first 31 characters (discarding the rest), and continue.

SYMBOL TABLE:

Scope :

- ❏ Match identifiers' declaration with uses
- ❏ Visibility of an entity
- ❏ Binding between declaration and uses
- ❏ The scope of an identifier is the portion of a program in which that identifier is accessible

- ❏ The same identifier may refer to different things in different parts of the program:
- ❏ Different scopes for same name don't overlap
- ❏ An identifier may have restricted scope

Scoping and Symbol Table :

- ❏ Given a declaration of a name, is there already a declaration of the same name in the current scope, i.e., is it multiply declared?
- ❏ Given a use of a name, to which declaration does it correspond (using the "most closely nested" rule), or is it undeclared?
- ❏ Symbol table: a data structure that tracks the current bindings of identifiers (for performing semantic checks and generating code efficiently)
- ❏ large part of semantic analysis consists of tracking variable/function/type declarations.
- ❏ As we enter each new identifier in our symbol table, we need to record the type information of the declaration.

Once all declarations have been processed to build the symbol table, and all uses have been processed, link each ID node in the abstract-syntax tree with the corresponding symbol-table entry.

It is used by various phases of compiler as follows :-

- ❏ Lexical Analysis: Creates new table entries in the table, example like entries about token.
- ❏ Syntax Analysis: Adds information regarding attribute type, scope, dimension, line of reference, use, etc in the table.
- ❏ Semantic Analysis: Uses available information in the table to check for semantics i.e. to verify that expressions and assignments are semantically correct(type checking) and update it accordingly.
- ❏ Intermediate Code generation: Refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.
- ❏ Code Optimization: Uses information present in symbol table for machine dependent optimization.

Items stored in Symbol table:

- ❑ Variable names, line number declared, Line number where it is used (required to get live ranges of a variable), type of a variable, storage required
- ❑ Procedure and function names
- ❑ Literal constants and strings
- ❑ Compiler generated temporaries (used during Intermediate code generation - as we store ICG in the form of quadruples)
- ❑ Labels in source languages

PHASE II : SYNTAX ANALYZER

- Syntax analysis is only responsible for verifying that the sequence of tokens forms a valid sentence given the definition of your Programming Language grammar.
- The parser will read your source programs and construct a Abstract Syntax Tree. If no syntax errors are encountered, your code will print the completed parse tree as flat text. At this stage, you aren't responsible for verifying meaning, just structure.
- The language must support :
    - global/local variables declarations and initializations,
    - functions,
    - classes(in case of C++, Java),
    - variables of various types (int, float, double, char) including arrays,
    - arithmetic and Boolean expressions,
    - Postfix and Prefix expressions
    - constructs chosen by you such as if, while, for, switch etc.
- Running bison/yacc on an incorrect or ambiguous grammar will report shift/reduce errors, useless rules, and reduce/reduce errors. To understand the conflicts being reported, scan the generated y.output file that identifies where the difficulties lie. Take care to investigate the underlying conflict and what is needed to resolve it rather than adding precedence rules like mad until the conflict goes away.
- You can rearrange the productions as needed to resolve conflicts.

- All conflicts and errors should be eliminated.
- Actions for each grammar rule can be added. But, Wait until all rules are debugged and then go back and add actions. The action for each rule will be to construct the section of the Abstract Syntax Tree corresponding to the rule reduced for use in later reductions. For example, when reducing a variable declaration, you will combine the Type and Identifier nodes into a VarDecl node, to be gathered in a list of declarations within a class or statement block at a later reduction.
- You can access locations using @n, get/set attributes using $ notation, set up the attributes union, know how attribute types are set, the attribute stack, yylval and yylloc, so on.
- If the action of a production forgets to assign $$ or inappropriately relies on the default result ($$ = $1), you don't get warnings or errors;
- At the end of parsing, if no syntax errors have been reported, the entire Abstract Syntax tree is printed via an inorder traversal.
- Error handling. If the input is not syntactically valid, the default behavior of a yacc/bison generated parser is to call yyerror to print a message and halt parsing.
- Provide a replacement version of yyerror that attempts to print the text of the current line and mark the first troubling token using the yylloc information recorded by the scanner.
- Pick out some simple errors that you think you can tackle and add some rudimentary error handling capabilities into your own parser. Describe which errors you attempt to catch and provide some sample instances that illustrate the errors from which your parser can recover.
- To receive full credit, your parser must do some error recovery. Simple recovery, say, at the statement and declaration levels, is enough.
- Update Symbol table with required information.

PHASE III : SEMANTIC ANALYSIS (WRITE APPROPRIATE ACTION RULES IN PARSER)

- Write appropriate rules to check for semantic validity (type checking, declare before use, etc.)
- Variables must be declared and can only be used in ways that are acceptable for the declared type.
- The test expression used in an if/while/for statement must evaluate to a Boolean value.
- In addition to type checking, there are other rules: new declarations don't conflict with earlier ones, access control on class fields aren't violated, break statements only appear in loops, and so on.

- One of the more interesting and worthwhile parts of the assignment is thinking through what is the best way to report various errors, so as to most help the programmer fix the mistake and move on.
- Your program will be considered correct if it verifies the semantic rules and reports appropriate errors.
- Your analyzer needs to show it can handle errors related to scoping and declarations, because these form the foundation for the later work.
- Update Symbol table with required information.

PHASE IV : INTERMEDIATE CODE GENERATION (3-address code)

- Write appropriate rules in Parser to generate Three address code.
- Code generated must be in Quadruple format.
- All temporaries must get a place in symbol table.

NOTE : Make 2 copies of your parser file. In one file you will add rules to output abstract syntax tree. In another copy you will add rules to output intermediate code.

PHASE V : INTERMEDIATE CODE OPTIMIZATION PHASE (3-address code):

- Eliminate Dead code/ unreachable code
- Implement Common subexpression elimination
- Implement Constant folding and Constant propagation
- Move loop invariant code outside the loop.
- Perform live variable analysis.

You must write a separate code in order to implement these, there are various algorithms available in order to implement the above mentioned optimizations.

The optimizations mentioned above are a must. You can optionally implement other optimizations for better grade.

PHASE VI : TARGET CODE GENERATION

- Assembly code for a particular ISA must be produced.
- Input : Optimized three-address code

INPUT/OUTPUT FOR YOUR PROJECT:

- Input : A complete program in the language chosen by you. (For example :
    - C program : May start with optional header file declaration, must have a main function.
    - C++ program : May start with optional header file declaration, class declaration , must have a main function.

    - Java program : May start with optional imports, must have a class and a main function.
    - Python program : May start with optional imports, must handle range() and lists.

Your input program if provided to a real compiler will execute.

- Output :
    - Symbol table with required Information
    - Abstract syntax tree printed in inorder way.
    - Three address code (in Quadruple format)
    - Optimized Three address code.
    - Assemble Code