From: Vandana Agarwal . (via Nalanda)
Sent: 08 March 2023 21:21
To: RAMAKANT PANDURANG TALANKAR .
Subject: 11551001318_CS F363_JAN_2023: Stage 2: Modules

V

Stage 2: Modules
by Vandana Agarwal . - Wednesday, 8 March 2023, 9:06 PM

Develop the following modules of the compiler for implementing the ERPLAG language.

Abstract Syntax Tree (AST): This module takes as input the parse tree generated in stage 1. The abstract syntax tree is generated by eliminating unnecessary details such as semicolon, colon, comma, parenthesis, square brackets, range operator, assignment operator etc. Any node in the abstract Syntax Tree retains the information about the non terminal symbol that would have derived the corresponding subtree (which is essential to keep the syntactic structure intact with you throughout, during front end). The AST retains only those children that are essential later for semantic analysis, while those appearing as a linear chain, are collapsed. Any child node corresponding to the operator (+, -, <= , AND etc) can be collapsed and the information regarding the terminal token such as PLUS, MINUS, LE, AND etc. is fetched up to the parent node. Implement an AST with the following general rule

(parent)(children list)

where parent is the name of the construct representing the sub-tree and can be either any new name or the same non-terminal symbol that exists in the parse tree. The children list contains the nodes which are meaningful. The leaf nodes of the AST still continue to contain the tokens and other relevant information extracted during stage 1. The AST later helps in traversing the tree faster than the parse tree, traversing the meaningful nodes only. You must

- Prepare the semantic rules to derive the abstract syntax tree structure
- Modify the structure of the parse tree node (at least a link to the ST can be added) to use for AST.
- Ensure a single pass of the parse tree.
- Produce as output the Abstract Syntax Tree, if the source code is syntactically correct.

Symbol Table (ST): This module takes as input the AST generated as above. Symbol table is a special data structure that maintains the information about the identifiers (variables that participate in computation as the source code is executed). The information gathered during semantic analysis phase is extremely valuable for generating the assembly language code for the input source code. Variables declared in different static scopes can be maintained in separate symbol tables. Since the scope of the variables is known only when the syntactic structure is established, i.e. after parsing, the symbol table links can be established to the function definitions after syntax analysis.

Implement Symbol Table to incorporate following information for all identifiers.

- Type
- Scope
- Offset etc.

Identifiers corresponding to the function names must be maintained separately. The information such as whether the token ID corresponds to a variable or it corresponds to the function name is obtained by the syntactic structure of the sub-tree that contains this identifier.

Type Extractor and Checker: Type of an identifier is extracted from the declaration statement that declares the identifier. The data types supported in the language you are implementing are: integer, real, boolean and array. The type of an element of an array is the type of the data the array refers to. For instance, consider the statement declare a: array[1..15] of integer;, then the type of an element a[5] is integer. The type checker verifies the type of an expression appearing at the right hand side of the assignment statement such as value := (a+b-c); and checks if it matches with that of the identifier on the left hand side. An arithmetic operator can have two operands of the similar type, where types can be integer and real data types. Example, if a, b, c and d are declared as integer then,

An expression in a statement a := (b*2-4*c)+5*d; its RHS has a type integer and since the identifier a is also of the same type, the type checker approves the expression assignment

An expression in a statement a:= (b*2 +c)/d <= c*10; has a boolean valued expression which is assigned to the integer valued identifier, hence the expression value assignment is wrong.

***Static type checking rules**:*

- The type of an identifier is the type appearing while declaring the variable.
- The type of NUM is integer.
- The type of RNUM is real.
- The type of true or false value is boolean.
- The type of an array variable A of type array [12..20] of real (say) is defined as an expression <real, 12, 20>. The type of an array element A[13] (say) is real if 13 is within the bounds [12, 20].
- The type of a simple expression (say E) of the form expression(say E1) <operator> Expression(say E2)
    -
    - is integer, if both expressions are of type integer and the operator is an arithmetic operator PLUS, MINUS or MUL.
    - is real, if both expressions are of type integer, or one is integer and the other is real, or both are real, and the operator is an arithmetic operator DIV.
    - is real, if both the expressions are of type real and the operator is arithmetic operator PLUS, MINUS or MUL.
    - is boolean, if both expressions are of type integer and the operator is a relational operator
    - is boolean, if both expressions are of type real and the operator is relational.
    - is boolean, if both expressions are of type boolean and the operator is logical.

- The type of the expression is ERROR, if the above rules do not derive the type of E appropriately.
- Types of expressions using unary - or + operators are same as those of their operands.
- Type checking rules for array construct are as follows:

The operations +, -, *, / and all relational and logic operators, cannot be applied on array variables of array type. For example, consider the declaration statement, declare A, B: array [12..20] of real; then the type of A and B are both <type, 12, 20>. The expression A+B, A-B, A*B and A/B are invalid and the type of these is assigned as ERROR.

The assignment operator applied to two array variables of the same type is valid. For example, if A and B are the array variables of type array[12..20] of real, then A:= B; is a valid statement. This applies to dynamic arrays of type array[a..b] of real as well.

Consider array elements with index represented by integer identifier say A[k]. Here type checking of variable k and A[k] are done at compile time, but the bound checking of A[k] is done at run time. If the type of k is integer, then it is valid, else it is reported as an error. Also, the type checking of A[13], for type of index (NUM), is done at compile time. The bound checking of A[13] where A is a static array is done at compile time. If A is a dynamic array, then bound checking of A[13] is done at run time [see below]

The type of an identifier or an expression is computed by traversing the abstract syntax tree. for declaration statement construct.

Dynamic type checking rules:

Following type checks are dynamic and your code generator takes care of dynamic type checking module.

- Address computation and type checking of variables of dynamic arrays such as in declare A: array[a..b] of integer; The offset computation is dependent on values of a and b and is done at run time.
- Bound checking of elements A[10] and A[n] for dynamic arrays, and of A[n] of static arrays is done at run time.
- Bound checking of array elements using arithmetic expression as index is done at run time. For example, an element A[p+q*r] is an array element whose index type is checked at compile time, but the bound checking is done at run time.
- The type checking of the arithmetic expression on the right hand side of the assignment operator in the statement m := n+A[p+q*r]; is done at compile time. Similar is done for the type checking of the operands of the assignment operator, which are m and n+A[p+q*r].

Semantic Analyzer: This module verifies the semantics of the code. Following are the rules that ERPLAG supports.

- An identifier cannot be declared multiple times in the same scope.
- An identifier must be declared before its use.

- The types and the number of parameters returned by a function must be the same as that of the parameters used in invoking the function.
- The parameters being returned by a function must be assigned a value. If a parameter does not get a value assigned within the function definition, it should be reported as an error.
- The function that does not return any value, must be invoked appropriately.
- Function input parameters passed while invoking it should be of the same type as those used in the function definition.
- A switch statement with an integer typed identifier associated with it, can have case statement with case keyword followed by an integer only and the case statements must be followed by a default statement.
- A switch statement with an identifier of type real is not valid and an error should be reported.
- A switch statement with a boolean type identifier can have the case statements with labels true and false only. The switch statement then should not have a default statement.
- Function overloading is not allowed.
- A function declaration for a function being used (say F1) by another (say F2) must precede the definition of the function using it(i.e. F2), only if the function definition of F1 does not precede the definition of F2.
- If the function definition of F1 precedes function definition of F2(the one which uses F1), then the function declaration of F1 is redundant and is not valid.
- A for statement must not redefine the variable that participates in the iterating over the range.
- The function cannot be invoked recursively.
- An identifier used beyond its scope must be viewed as undefined etc. (More semantics will be made available in the test cases)

Code Generator : This module takes as input the abstract syntax tree (AST) as intermediate representation. The function generates 8086 assembly code. Only trivial optimization such as avoiding redundant code, appropriate register usage etc. is needed while the detailed code optimization techniques are not expected to be implemented. The code generator generates the code for dynamic type checking as well. Your compiler must generate equivalent assembly code (in file code.asm) with instructions taken from instruction set of the NASM simulator (linux based, exact version will be provided later).