

CS 256 – Programming Languages and Translators

Assignment 4

- This assignment is due by 1 p.m. on Monday, March 10, 2014
- This assignment will be worth 10% of your grade
- You are to work on this assignment by yourself

Basic Instructions

For this assignment you are to modify your lexical and syntactical analyzer from HW3 to make it also do semantic analysis.

As before, your program must compile and execute on one of the campus Linux machines (such as *rcnnxcs213.managed.mst.edu* where *nn* is 01-32). If your flex file was named *mfpl.l* and your bison file was named *mfpl.y*, we should be able to compile and execute them using the following commands (where *inputFileName* is the name of some input file):

```
flex mfpl.l
bison mfpl.y
g++ mfpl.tab.c -o mfpl_parser
mfpl_parser < inputFileName
```

If you want to create an output file (named *outputFileName*) of the results of running your program on *inputFileName*, you can use:

```
mfpl_parser < inputFileName > outputFileName
```

As in HW3, no attempt should be made to recover from non-lexical errors; if your program encounters an error, it should simply output a meaningful message containing the line number where the error was found, and terminate execution. Listed below are the new errors that your program also will need to be able to detect for MFPL programs:

```
Arg n must be integer
Arg n must be string
Arg n must be integer or string
Arg n cannot be function
Too many parameters in function call
Too few parameters in function call
```

Since once again we will use a script to automate the grading of your programs, you must use these exact error messages.

Note that every parenthesized expression in MFPL (Mini Functional Programming Language) is of the form (function arg1 arg2 arg3 ...). For example, in the arithmetic expression (+ x y), *x* is the first argument for the + function, and *y* is the second argument. If *x* is not an integer, then we should output the message Arg 1 must be integer.

Your program should still output the tokens, lexemes, productions being processed, open/close scope messages, and symbol table insertion messages.

As before, your program should process a single expression from the input file (but remember that a single expression could be an expression list), terminating when it completes processing the expression or encounters an error.

Note that your program should NOT evaluate any statements in the input program; well do that in the next assignment! Consequently, you do not have to record an identifiers value in the symbol table storing an identifiers name, type, and number of parameters and return type (if it is a function) should be sufficient for now.

Programming Language

What follows is a brief description about the semantic rules that we want to enforce for the various expressions in MFPL. This should serve as a guide for your type-checking.

The following descriptions assume that you have defined integer constants to represent the following types: `BOOL`, `INT`, `STR`, `FUNCTION`, `INT_OR_STR`, `INT_OR_BOOL`, `STR_OR_BOOL`, `INT_OR_STR_OR_BOOL`

`N_EXPR` \rightarrow `N_CONST` | `T_IDENT` | `T_LPAREN N_PARENTHESED_EXPR T_RPAREN`

The resulting type of an `N_EXPR` is the resulting type of the `N_CONST` if that rule is applied, the actual type of the identifier if the `T_IDENT` rule is applied (you'll have to hit your symbol table to find out its type), or the resulting type of the `N_PARENTHESED_EXPR` if that rule is applied.

`N_CONST` \rightarrow `T_INTCONST` | `T_STRCONST` | `T_T` | `T_NIL`

The resulting type of an `N_CONST` is `INT` if the `T_INTCONST` rule is applied, `STR` if the `T_STRCONST` rule is applied, or `BOOL` if the `T_T` or `T_NIL` rules are applied.

`N_PARENTHESED_EXPR` \rightarrow `N_ARITHLOGIC_EXPR` | `N_IF_EXPR` | `N_LET_EXPR` | `N_LAMBDA_EXPR` | `N_PRINT_EXPR` | `N_INPUT_EXPR` | `N_EXPR_LIST`

The resulting type of an `N_PARENTHESED_EXPR` is the resulting type of whichever rule is applied.

`N_ARITHLOGIC_EXPR` \rightarrow `N_UN_OP N_EXPR` | `N_BIN_OP N_EXPR N_EXPR`
`N_BIN_OP` \rightarrow `N_ARITH_OP` | `N_LOG_OP` | `N_REL_OP`
`N_ARITH_OP` \rightarrow `T_MULT` | `T_SUB` | `T_DIV` | `T_ADD`
`N_LOG_OP` \rightarrow `T_AND` | `T_OR`
`N_REL_OP` \rightarrow `T_LT` | `T_GT` | `T_LE` | `T_GE` | `T_EQ` | `T_NE`
`N_UN_OP` \rightarrow `T_NOT`

The operand expressions of an `N_ARITHLOGIC_EXPR` (i.e., `arg1` and `arg2`) will need to be checked to see if they are appropriate for the operator being used. For `N_UN_OP`, the `N_EXPR` can be any type except `FUNCTION`. For `N_BIN_OP`, valid `N_EXPR` type combinations (regardless of order) are shown in the following table:

operator type:	relational	logical	arithmetic
INT, INT	legal	legal	legal
INT, STR	illegal	legal	illegal
INT, BOOL	illegal	legal	illegal
INT, FUNCTION	illegal	illegal	illegal
STR, STR	legal	legal	illegal
STR, BOOL	illegal	legal	illegal
STR, FUNCTION	illegal	illegal	illegal
BOOL, BOOL	illegal	legal	illegal
BOOL, FUNCTION	illegal	illegal	illegal
FUNCTION, FUNCTION	illegal	illegal	illegal

The resulting type of an `N_ARITHLOGIC_EXPR` will be `INT` if the operator is `*`, `+`, `-`, or `/`; otherwise, it will be `BOOL`.

`N_IF_EXPR` \rightarrow `T_IF` `N_EXPR` `N_EXPR` `N_EXPR`

All three operand expressions (i.e., `arg1`, `arg2`, and `arg2`, respectively) can be any type except `FUNCTION` (and they don't all have to be the same type). At this time we don't know whether the second or third expression actually will be evaluated. So the resulting type of an `_IF_EXPR` will be assigned from the type combinations of the second and third expressions (regardless of their order) as shown in the following table:

	INT	STR	BOOL
INT	INT	INT OR STR	INT OR BOOL
STR	INT OR STR	STR	STR OR BOOL
BOOL	INT OR BOOL	STR OR BOOL	BOOL
INT OR STR	INT OR STR	INT OR STR	INT OR STR OR BOOL
INT OR BOOL	INT OR BOOL	INT OR STR OR BOOL	INT OR BOOL
STR OR BOOL	INT OR STR OR BOOL	STR OR BOOL	STR OR BOOL
INT OR STR OR BOOL	INT OR STR OR BOOL	INT OR STR OR BOOL	INT OR STR OR BOOL
	INT OR STR	INT OR BOOL	STR OR BOOL
INT	INT OR STR	INT OR BOOL	INT OR STR OR BOOL
STR	INT OR STR	INT OR STR OR BOOL	STR OR BOOL
BOOL	INT OR STR OR BOOL	INT OR BOOL	STR OR BOOL
INT OR STR	INT OR STR	INT OR STR OR BOOL	INT OR STR OR BOOL
INT OR BOOL	INT OR STR OR BOOL	INT OR BOOL	INT OR STR OR BOOL
STR OR BOOL	INT OR STR OR BOOL	INT OR STR OR BOOL	STR OR BOOL
INT OR STR OR BOOL	INT OR STR OR BOOL	INT OR STR OR BOOL	INT OR STR OR BOOL
		INT OR STR OR BOOL	
	INT	INT OR STR OR BOOL	
	STR	INT OR STR OR BOOL	
	BOOL	INT OR STR OR BOOL	
	INT OR STR	INT OR STR OR BOOL	
	INT OR BOOL	INT OR STR OR BOOL	
	STR OR BOOL	INT OR STR OR BOOL	
	INT OR STR OR BOOL	INT OR STR OR BOOL	

`N_LAMBDA_EXPR` \rightarrow `T_LAMBDA` `T_LPAREN` `N_ID_LIST` `T_RPAREN` `N_EXPR`
`N_ID_LIST` \rightarrow ϵ | `N_ID_LIST` `T_IDENT`

You should simply assign the type of each `T_IDENT` in `N_ID_LIST` as `INT`. The `N_EXPR` in `N_LAMBDA_EXPR` (i.e., what should be considered the lambda function's `arg2`) can be any type except `FUNCTION`. But the overall type of an `N_LAMBDA_EXPR` is `FUNCTION`. The return type of the function is whatever type `N_EXPR` is, and the number of parameters for the function is the length of `N_ID_LIST`. Note: Recursive function calls are not supported in MFPL. The way we're managing the symbol table and doing the parsing should automatically catch such an attempt as an undeclared identifier.

`N_PRINT_EXPR` \rightarrow `T_PRINT` `N_EXPR`

The `N_EXPR` can be any type **except** `FUNCTION`. The resulting type of an `N_PRINT_EXPR` is then whatever the type of the `N_EXPR` is.

`N_INPUT_EXPR` \rightarrow `T_INPUT`

Input can either be a string or an integer (we won't know until runtime). So for now the resulting type of a `N_INPUT_EXPR` should be considered `INT_or_STR`. Note: An expression that is of type `INT_or_STR` should be considered type-compatible with both type `INT` and type `STR`.

$$N_EXPR_LIST \rightarrow N_EXPR N_EXPR_LIST \mid N_EXPR$$

The resulting type of an `N_EXPR_LIST` can be any type except `FUNCTION`. Make sure that if the very first `N_EXPR` is a function name or function definition that:

1. you check that the subsequent number of `N_EXPRs` in the `N_EXPR_LIST` matches the number of parameters expected by that function
2. you assign the resulting type of the `N_EXPR_LIST` to be the function's return type.

Symbol Table Management

You will need to make some changes to the symbol table classes to accommodate the type information that we need to assign and check in this project. For example, you might find it useful to define the following to store pertinent information:

```
#define UNDEFINED    -1    // Type codes
#define FUNCTION     0
#define INT          1
#define STR          2
#define INT_OR_STR   3
#define BOOL         4
#define INT_OR_BOOL  5
#define STR_OR_BOOL  6
#define INT_OR_STR_OR_BOOL 7

#define NOT_APPLICABLE -1

typedef struct {
    int type;           // one of the above type codes
    int numParams;      // numParams and returnType only applicable
    int returnType;     // if type == FUNCTION
} TYPE_INFO;
```

Additional Tips on Semantic Error Detection

In order to do type checking in this assignment, you will need to specify what kind of information will be associated with various symbols in the grammar. As in HW3, one way to do this is to define the following union data structure right after the `%}` in your `mfpl.y` file:

```
%union {
    char* text;
    TYPE_INFO typeInfo;
};
```

As in HW3, the `char*` type can be used to associate an identifier's name with an identifier token. The `TYPE_INFO` type can be used to associate a struct of type information with a nonterminal. You'll need to define what type is to be associated with what grammar symbol by using `%type` declarations in your `mfpl.y` file such as the following:

```
%type <text> T_IDENT
%type <typeInfo> N_EXPR N_PARENTHESIZED_EXPR N_IF_EXPR
```

Note that not every symbol in the grammar has to be associated with a %type; some symbols may not need any such information at this time (e.g., N_START, N_UN_OP, etc.). When you process a nonterminal during the parse, you can assign its type (encoded as typeInfo) as in the examples below:

```
N_CONST : T_INTCONST
{
    printRule("CONST", "INTCONST");
    $$ .type = INT;
    $$ .numParams = NOT_APPLICABLE;
    $$ .returnType = NOT_APPLICABLE;
}
```

```
N_EXPR : T_LPAREN N_PARENTHESIZED_EXPR T_RPAREN
{
    printRule("EXPR", "( PARENTHESIZED_EXPR )");
    $$ .type = $2.type;
    $$ .numParams = $2.numParams;
    $$ .returnType = $2.returnType;
}
```

You can then check the type of an expression within a particular context as in the following:

```
N_ARITHLOGIC_EXPR : N_UN_OP N_EXPR
{
    printRule("ARITHLOGIC_EXPR", "UN_OP EXPR");
    if ($2.type == FUNCTION) {
        yyerror("Arg 1 cannot be function");
        return(1);
    }
    $$ .type = BOOL;
    $$ .numParams = NOT_APPLICABLE;
    $$ .returnType = NOT_APPLICABLE;
}
```

Note that all output requirements specified in previous assignments are applicable for this assignment.

Submission

You will submit this assignment using `cssubmit`. Your *single* lexer file *must* have a `.l` extension, and your single yacc/bison file must have a `.y` extension. If it does not, you will receive a zero for this assignment. From the directory containing the `.l` file, `.y`, and symbol table files, you will run

```
cssubmit 256 a 3
```

on the cs213 Linux machines. This will collect your submission and submit it to me. You may submit as many times as you desire; only your last submission will be graded (previous submissions are overwritten). **READ** the output of `cssubmit`; it may have changed since you last used it.

When grading, I will run the following commands:

```
flex *.l
bison *.y
g++ *.tab.c
```

If you cannot generate an executable by following EXACTLY those steps from the directory you are submitting from, your submission will not receive full credit. If you find yourself having example .l and .y files, or some other file with a `.tab.c` extension, remove them from your submission directory.