

1 Objectives

- Generating Intermediate Code.
- Finding Basic Blocks.

2 Problem Statement

You are to write a program that:

- reads a filename from the command line
- using the grammar that follows, determines whether or not the filename specified contains text that matches the grammar.
- outputs a new file that contains equivalent intermediate code to the input source code program.

3 Grammar

The following grammar specifies the language you are to use; the start rule is *prog* and all other rules are alphabetized for ease in finding them. Note that it is a slightly amended version of the grammar from project 1.

<i>prog</i>	→	<i>decl prog</i> <i>decl</i>
<i>base_type</i>	→	INT VOID
<i>compound</i>	→	LBRACE <i>stmtlist</i> RBRACE
<i>constr</i>	→	LSQR NUM_INT RSQR ϵ
<i>decl</i>	→	<i>vardecl</i> SEMICOLON <i>funcdecl</i>
<i>expr</i>	→	<i>simplexpr</i> <i>simplexpr</i> GREATER <i>simplexpr</i> <i>simplexpr</i> LESS <i>simplexpr</i> <i>simplexpr</i> EQUALTO <i>simplexpr</i>
<i>exprlist</i>	→	<i>expr</i> COMMA <i>exprlist</i> <i>expr</i>
<i>factor</i>	→	ID ID LPAREN <i>exprlist</i> RPAREN ID LSQR <i>expr</i> RSQR NUM_INT LPAREN <i>expr</i> RPAREN
<i>funcdecl</i>	→	<i>type</i> ID LPAREN <i>params</i> RPAREN <i>vardecl_list</i> <i>compound</i>
<i>idlist</i>	→	ID COMMA <i>idlist</i> ID
<i>paramlist</i>	→	<i>type</i> ID COMMA <i>paramlist</i> <i>type</i> ID
<i>params</i>	→	<i>paramlist</i> ϵ
<i>sign</i>	→	PLUS MINUS
<i>simplexpr</i>	→	<i>term</i> <i>sign</i> <i>term</i> <i>simplexpr</i> PLUS <i>term</i> <i>simplexpr</i> MINUS <i>term</i>
<i>stmt</i>	→	<i>compound</i> <i>variable</i> ASSIGN <i>expr</i> SEMICOLON IF <i>expr</i> <i>stmt</i> IF <i>expr</i> <i>stmt</i> ELSE <i>stmt</i> WHILE <i>expr</i> <i>stmt</i>
<i>stmtlist</i>	→	<i>stmt</i> <i>stmtlist</i> <i>stmt</i>
<i>term</i>	→	<i>factor</i> <i>term</i> MUL <i>factor</i> <i>term</i> DIV <i>factor</i>
<i>type</i>	→	<i>base_type</i> <i>constr</i>
<i>vardecl</i>	→	<i>type</i> <i>idlist</i>
<i>vardecl_list</i>	→	<i>vardecl</i> SEMICOLON <i>vardecl_list</i> ϵ
<i>variable</i>	→	ID ID LSQR <i>expr</i> RQSR POINTER ID

4 Lexical Analysis

The following table summarizes the tokens used in the grammar:

Token Type	Explanation
ASSIGN	= (assignment operator)
COMMA	, (used as a separator)
DIV	/ (division operand)
ELSE	(keyword)
EQUALTO	== (equality comparison)
GREATER	> (greater than operator)
ID	<i>letter followed by zero or more letters/digits</i>
IF	(keyword)
INT	integer (keyword for int type)
LESS	< (less than operator)
LBRACE	{ (used for opening a compound statement)
LPAREN	((various uses)
LSQR	[(start of array dimension)
MINUS	- (subtraction operator)
MUL	* (multiplication operator)
NUM_INT	<i>integer constant value</i>
PLUS	+ (addition operator)
RBRACE	} (end of compound statement)
REAL	real (keyword for floating point type)
RPAREN) (various uses)
RSQR] (end of array dimension)
SEMICOLON	; (terminates a statement)
VOID	void (keyword for void type)
WHILE	(keyword)

5 Summary of Grammar Changes From Project #1

- New tokens: **LESS**, **GREATER**, **EQUALTO**.
- No pointer or real types, arrays can only be one-dimensional.
- *expr* rule now handles relational expressions as well.
- *constr* rule no longer allows pointers or recursive definitions (i.e. multidimensional arrays).
- *base_type* no longer allows real numbers.
- *variable* and *factor* rules no longer supports pointer dereference or multidimensional arrays.
- *stmt* and *stmtlist* rules fix “extra” semicolon issue from project 1

6 Intermediate code

You are to generate “quadruples” - i.e. statements of the form:

```
label: operation op1, op2, op3, op4
```

where **operation** is the *only* required portion for any given statement. An example of such code is ILOC, as found in the text book. Feel free to use your own (sensible) extensions (keeping in mind that your next project will be converting each of your intermediate code statements into corresponding assembly language). For example, I added:

- `PARAMETER param_value` , which denotes that `param_value` is a parameter that is being passed to the upcoming function call.
- `CALL label`, which denotes an actual function call.
- `RETURN`, which denotes a return from a function.

7 Arrays

You are only required to implement array functionality if you are a Graduate Student! It is up to you to come up with a reasonable scheme for handling arrays. Do not worry about assigning entire arrays to entire arrays in an assignment statement (although you should be able to copy an individual element from an array to another individual element in an array). You should, however, worry about passing entire arrays as parameters.

8 Extra Credit

Don't forget – get this to me 48 hours or more early and you get a free 5 points. So, submit this project by 4/3 at 11:59 PM and you'll get a free 5 points!

9 Grading Breakdown

Correct Submission	10%
Successful Compilation & Makefile	35%
Correct Execution	40%
Comments & read.me File	15%
Extra Credit	5%

10 What to Submit

You will be submitting a tar file containing all source code, a makefile, and a `read.me` file for the project. If necessary, we will discuss the creation of tar files as well as how to build (and use) an appropriate `Makefile` in class. The `read.me` file should be a plain ASCII text file that overviews your project, including:

- your name.
- how to compile your project (if you have submitted a Makefile as required, this should be something like “type `make` to build the project”).
- how to run your project
- any known bugs your project has, and possible fixes to those bugs (partial credit abounds here).
- an overview of how you solved the project.
- what you named your output files(the file that contains intermediate code, broken down by basic block.