

COMP 442: Compiler Design
SLANG Compiler Project
Version 1.0

Name	Student ID
Nadia Chaouch	XXXXXXX

Concordia University	Version: 1.0
	Date: 22/04/04

Table of Contents

1.	Description of the Source Language SLANG	3
1.1	Syntax	3
1.1.1	SLANG Tokens	3
1.1.2	SLANG Statements	3
1.2	Lexical Conventions	4
1.3	Semantics	4
1.4	Changes in the Grammar	4
1.5	Limitations, Extensions	5
2.	User's Guide	6
2.1	How to Use	6
2.2	Features	6
3.	Structure of the Compiler	6
3.1	Design of the Lexical Analyzer	7
3.2	Design of the Parser	8
3.3	Design of the Symbol Table	8
3.4	Description of the Code Generation	9
3.4.1	Intermediate Code Generation	9
3.4.2	Moon Code Generation	10
3.5	Description of the Error Handling	12
3.6	List of the Errors Recognized by the Compiler	12
4.	Evaluation of the Design Decisions and Efficiency	12

SLANG Compiler Project

1. Description of the Source Language SLANG

1.1 Syntax

1.1.1 SLANG Tokens

The following tokens can be meaningfully used in SLANG:

Identifier specified by	(l+d)+ _(l+d)*			
Numerical constant specified by	dd*			
Character constant specified by	'c'			
()	;	+	-
*	/	:=	>	<
=	<=	>=	!=	[
]	,	:		
begin	end	program	variables	integer
array	char	module	if	then
else	loop	exit	read	write

1.1.2 SLANG Statements

Program declaration	<pre> program <i>programe</i>; variables <i>variables declarations</i> <i>module declarations</i> begin <i>main program code</i> end; </pre>
Variables declarations	<pre> <i>varname1</i> ,<i>varname2</i> ,<i>varnamen</i>: integer; <i>varname</i> : integer array[<i>size</i>]; <i>varname</i> : char; </pre>
Module declarations	<pre> module <i>modname</i> (<i>parameter declarations</i>) variables begin end; </pre>
if-then-else statements	<pre> if condition then <i>yes_statement</i>; else <i>no_statement</i>; </pre>
Loop statements	<pre> loop <i>loop statements (including one of multiple exit statements to exit loop)</i> end; </pre>
Assignment statements	<pre> <i>var_to_be_modified</i> := <i>value_to_be_copied</i> ; </pre>
Mathematical operations	<pre> <i>var</i> := <i>var1</i>+<i>var2</i>; <i>var</i> := <i>var1</i>-<i>var2</i>; <i>var</i> := <i>var1</i>*<i>var2</i>; <i>var</i> := <i>var1</i>/<i>var2</i>; </pre> <p>The parenthesis () can be used in these statements to create a long expression. Parenthesis can be nested</p>
begin-end statements	<pre> begin </pre>

Concordia University	Version: 1.0
	Date: 22/04/04

	end; This statements is to be used to wrap many statements together into a block. They can be used in association with the if-then-else statements, as well as program and module declarations
read/write statements	read <i>varname</i> ; write <i>varname</i> ;

1.2 Lexical Conventions

In this implementation of the SLANG compiler, whitespace is represented by either a space, tab or newline. Lines that are completely blank are ignored but are counted in the line numbers for the listing file. Two identifiers and/or reserved words must be separated by some type of non-alpha character. This character can be a whitespace.

Reserved words are stored as small letters. Otherwise, they will be treated as a regular identifier. For example, begin is not the same as BEGIN. "begin" will be identified as a keyword and "BEGIN" will be identified as an identifier.

Identifiers are held within a string and are kept as they are written. They are case sensitive as well. Therefore, identifiers "I" and "i" are different. This is due to the fact that string comparison is used.

Comments must only span one line. If they are not correctly terminated, an unknown token marker will appear in the code.txt file.

1.3 Semantics

Meaning	Tokens
Used to indicate the start of a program or module definition	program, module
Used for conditional statements	if, then, else
used to form a loop block	loop, exit, end
Used for input and output operations	read, write
Used in the declaration of variables	variables
Used to define the data type	integer, char, array
Used to form a block	begin, end
Used for comparison	<=, <, >=, >, !=, =
Used for mathematical operations	+, -, *, /
Used to enclose mathematical operations	()
Used for array indexing	[]
Used to end a statement	;
Used to separate tokens that will all follow the same instruction	,
Used in variable declaration to separate variable name from type	:
Used for assignment	:=
Used for character constants	'c'

1.4 Changes in the Grammar

The following is the modified grammar used by the parser:

- | | |
|-------------------------|----------------------------|
| #1 M → program i ; DL B | #5 MO → module i (VL) DV B |
| #2 DL → DV ML | #6 DV → variables VL |
| #3 ML → MO ML | #7 DV → e |
| #4 ML → e | #8 VL → V VM |

Concordia University	Version: 1.0
	Date: 22/04/04

#9 VM → V VM	#37 L → i AR
#10 VM → e	#38 LR → N
#11 V → IL : T ;	#39 LR → c
#12 T → integer AD	#40 C → E OR E
#13 T → char AD	#41 OR → =
#14 IL → i IM	#42 OR → <=
#15 IM → , IM	#43 OR → >=
#16 IM → e	#44 OR → !=
#17 AD → e	#45 OR → <
#18 AD → array [n]	#46 OR → >
#19 B → begin SL end ;	#47 AR → e
#20 SL → S SM	#48 AR → [E]
#21 SM → e	#49 / 50 / 51 E → F { OA F }
#22 SM → S SM	#52 OA → +
#23 S → i SO	#53 OA → -
#24 S → if C then S else S	#54 / 55 / 56 F → R { OM R }
#25 S → loop SL end ;	#57 OM → *
#26 S → exit ;	#58 OM → /
#27 SO → (LP)	#59 R → L
#28 S → B	#60 R → n
#29 S → read LN ;	#61 R → (E)
#30 S → write LO ;	#62 R → c
#31 S → e	#63 / 64 / 65 LO → LR { , LR }
#32 LP → e	#66 SO → AR := E ;
#33 LP → LN	#67 LR → i AR
#34 / 35 / 36 LN → L { , L }	

1.5 Limitations, Extensions

The parser has been implemented to use pass by value. Therefore, no pass by reference is possible, to counter this, global variables have been implemented. All variables declared as part of the main program are considered as globals and may be accessed by all scopes of the program.

Also, each function must take at least one parameter.

The current implementation does not support mutual recursion. To illustrate this, we cannot process a program that is composed of:

Mod1 calls Mod2

Mod2 calls Mod 3

Mod 3 calls Mod1

This is due to the fact that the source program is processed in one pass and that a module or variable must be declared before it is used. As the language does not provide a module declaration, which can be included at the beginning of the program (before the definition), there is no possible order to declare these methods.

Also the number of parameters that can be passed to a method is limited. This is due to the way the moon machine code generation has been implemented.

Due to the type of parameter passing used, moon machine code files may become very large when used with arrays.

To make the implementation more user-friendly it would be beneficial to have comments which may span more than one line. This would be useful for long comments as well as debugging purposes.

Concordia University	Version: 1.0
	Date: 22/04/04

Another feature to make the language cleared and easier to understand would be to replace the begin-end statements with a type of brackets such as is used in many common programming languages such as C++ eg. {}. This feature would be fairly easy to implement. However, this exceeds the scope of this project as well as adopted requirements therefore it will be left for future improvements.

2. User's Guide

2.1 How to Use

This project provides a compiler for the SLANG language. A source file is to be analyzed and then several files are returned as output. The purpose of this project is to provide conversion from the high-level SLANG language to moon machine code. The moon machine code may then be executed on the moon machine and can be verified for correctness. The present document include a user guide for SLANG programmers and contains all necessary information about the language and its' compiler.

To compile a SLANG source file:

- Name the source file "source.txt"
- Put the file in the same directory as the compiler executable
- Run the compiler program by either double-clicking on the icon or using the command line
- All associated files will then be created, including code for the moon machine

To execute a compiled program on the moon machine:

- Place the moon machine source file in the same folder as the moon machine executable
- Run the moon machine using the library file util.m and the source file moon.txt. This may be done using the Windows cmd command.

Filename	Contents
Source.txt	Contains the SLANG source code to be analyzed
Listing.txt	Contains the source code listing. Provides line numbers and removes comments
Productions.txt	Contains the production numbers used by the grammar
Symbol.txt	Contains a printout of the symbol table
Code.txt	Contains all the tokens and their associated types
Intermediatecode.txt	Contains the intermediate code generated by the parser
Moon.txt	Contains the code generated using the intermediate code and that is to be executed on the moon machine

2.2 Features

The SLANG compiler provided has many interesting features. First, it outputs meaningful errors along with the line number of the error. This is very useful in debugging code. When a syntax error occurs, compilation may continue and try to recover. When a variable is used without being declared, the compilation will stop because no meaningful code could be generated.

Furthermore, parameter passing is implemented as pass by value. Programmers also have access to recursion.

3. Structure of the Compiler

The following sections describe each step of the compiler and provide details of its implementation.

3.1 Design of the Lexical Analyzer

The scanner is used to analyze the source file and to separate the file into tokens. The following table illustrates the various token types that can be found.

<i>Token Type</i>	<i>Code</i>
EOF	ENDFILE
ERROR	ERROR
BEGIN	BEGIN
END	END
PROGRAM	PROGRAM
VARIABLES	VARIABLES
INTEGER	INTEGER
ARRAY	ARRAY
CHAR	CHAR
MODULE	MODULE
IF	IF
THEN	THEN
ELSE	ELSE
LOOP	LOOP
EXIT	EXIT
READ	READ
WRITE	WRITE
Identifier	ID
Numerical constant	NUM
Character constant	CH
(LPAREN
)	RPAREN
;	SEMI
+	PLUS
-	MINUS
*	TIMES
/	OVER
:=	ASSIGN
<	LT
>	GT
=	EQ
<=	LTE
>=	GTE
!=	NE
[LBRACK
]	RBRACK
,	COMMA
:	COLON

The scanner is controlled by a DFA. The DFA is used to control and keep track of the previous characters that were read and gradually narrows down the possible Token types. The longest substring principle is used in the implementation. The following states are found in the DFA.

Concordia University	Version: 1.0
	Date: 22/04/04

States	Function
START	Starting state for all tokens. Analyses first character of all tokens and sends to appropriate state. Also deals directly with single character tokens.
INID1	Identifier state that is a final state
INID2	Identifier state that is non-final due to an underscore character
INNUM	Numerical constant token being formed
INCHAR1	Has read the first apostrophe of the character token
INCHAR2	Has read the character of the character constant token
INSLASH	Has read a "/" symbol could be either beginning of a comment or divide sign
INCOMMENT1	The first "*" of the comment has been read
INCOMMENT2	Has read the second or more "*" token of a comment, waiting for a "/" symbol to be read
INCOLON	Has read a ":" could be either an assign statement or a single colon character
INLT	Has read a "<" character, could be a LT or LTE token
INGT	Has read a ">" character, could be a GT or GTE token
INNE	Has read a "!" character, could either be a NE token or ERROR
DONE	State that ends the current token
INERROR	Indicates that an error has occurred and should be handled. State goes to DONE so scanning can continue

3.2 Design of the Parser

The parser is without a doubt the most complicated part of the compiler. The parser drives all other subsequent compiler phases. The parser is a recursive descent parser and is modified to allow for LL(1) parsing.

In the first phase of the parser, the grammar rules were implemented to allow a verification of proper syntax and semantic meaning. Please see the grammar included in section 1.4

In the second phase of the parser, the symbol table was implemented. This required that additional statements be added where symbols were being used and/or declared.

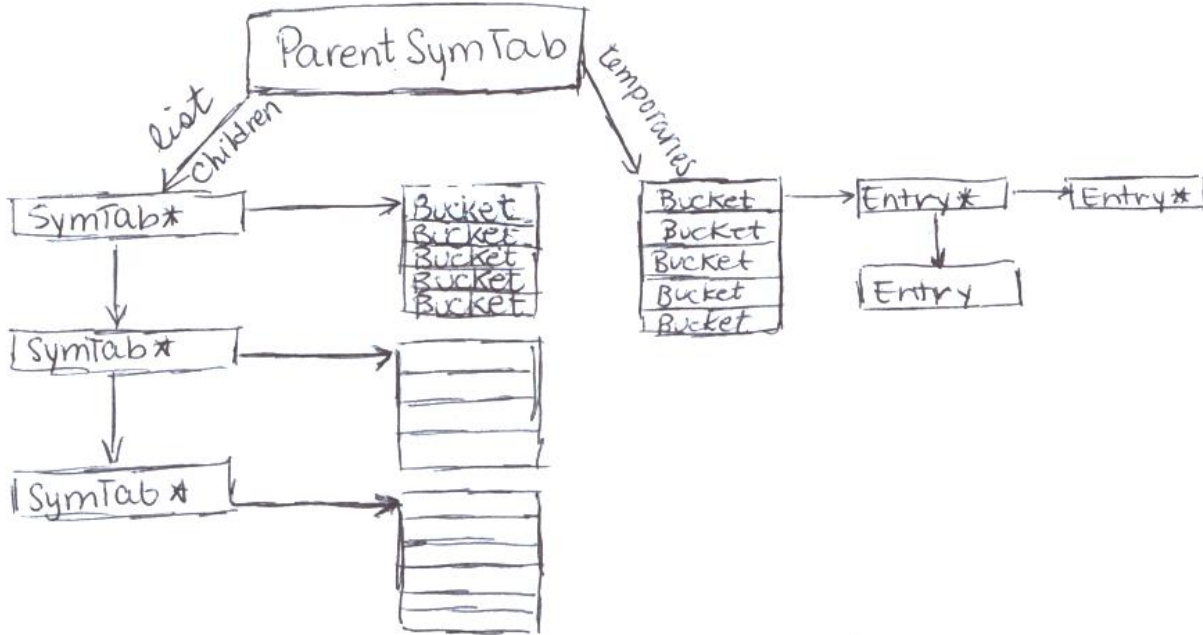
In the third phase of the parser, intermediate code generation functionality was implemented. Therefore, additional statements were needed in the parser to implement these new features.

3.3 Design of the Symbol Table

The Symbol table structure is actually a linked list of symbol tables. A class ParentSymTab controls and maintains the symbol tables (SymTab). ParentSymTab initially contains one child symbol table (SymTab). When a new scope is declared, a new symbol table is added to the head of the list. Each SymTab is implemented as a fixed size array of buckets that form a hash table. A bucket is a linked list of entries (Entry). An Entry is the unit in which identifier information is stored. Another symbol table is also kept in the ParentSymTab to store temporaries used in intermediate code generation.

Insertions and searches in the symbol table are controlled by the ParentSymTab which then passes control over to the first child SymTab. In the case of an insertion, a hash is performed and the identifier name and other relevant information such as structure and type is stored in that symbol table (of deepest depth). In the case of a search, the child SymTab's are hashed and searched one by one until a specific identifier is found. If it is not found, an error is passed to the cerr stream. The parsing will then stop because no meaningful will be produced.

To better visualize the Symbol table structure, the following diagram is provided.



3.4 Description of the Code Generation

3.4.1 Intermediate Code Generation

The intermediate code is based on a class InstrList that is supported by many subclasses. InstrList contains many important attributes as well as important functions. Most importantly it contains a vector of instructions (vector<Instr>) generated by the parser. The Instr class holds information about individual instructions. It contains one InstrType as well as 3 operands (class Operand). The class Operand contains information about each operand.

Instruction Types

Stored as an enum, it describes the various instructions that are used in the intermediate code.

InstrNone	Included for good programming practice
InstrAdd	Used for addition
InstrSub	Used for subtraction
InstrMul	Used for multiplication
InstrDiv	Used for division
InstrRead	Used to read
InstrWrite	Used to write
InstrMov	Used to move (copy) variables
InstrIfFalse	Used to jump in conditional statements
InstrEQ	Used for comparison
InstrLT	Used for comparison
InstrLTE	Used for comparison
InstrGT	Used for comparison
InstrGTE	Used for comparison
InstrNEQ	Used for comparison
InstrCall	Used for module calls
InstrParam	Used to declare parameters

Concordia University	Version: 1.0
	Date: 22/04/04

InstrJump	Used for statement jumps
InstrLabel	Used to add labels
InstrHalt	Used to indicate end of instruction list
InstrEntry	Used to indicate the beginning of the main module
InstrEndMod	Used to indicate end of module and that it must return to caller
InstrBeginMod	Used to indicate the beginning of a module
InstrPreCall	Used to indicate the start of a module call

Instruction Address

Stored as an enum, it describes the type of memory address used for the operands.

AddrEmpty	Default address
AddrTemp	Used for temporaries
AddrChar	Used for characters
AddrInt	Used for integers
AddrRef	Used for identifiers
AddrArray	Used for arrays

Instruction Offset

Stored as an enum, these are needed for arrays in order to determine if the index is a literal value or a pointer to an identifier found in the symbol table.

AddrNone	Included for good programming practice
AddrSym	Used when an identifier is used as the index
AddrLitt	Used when a literal is used as the index

3.4.2 Moon Code Generation

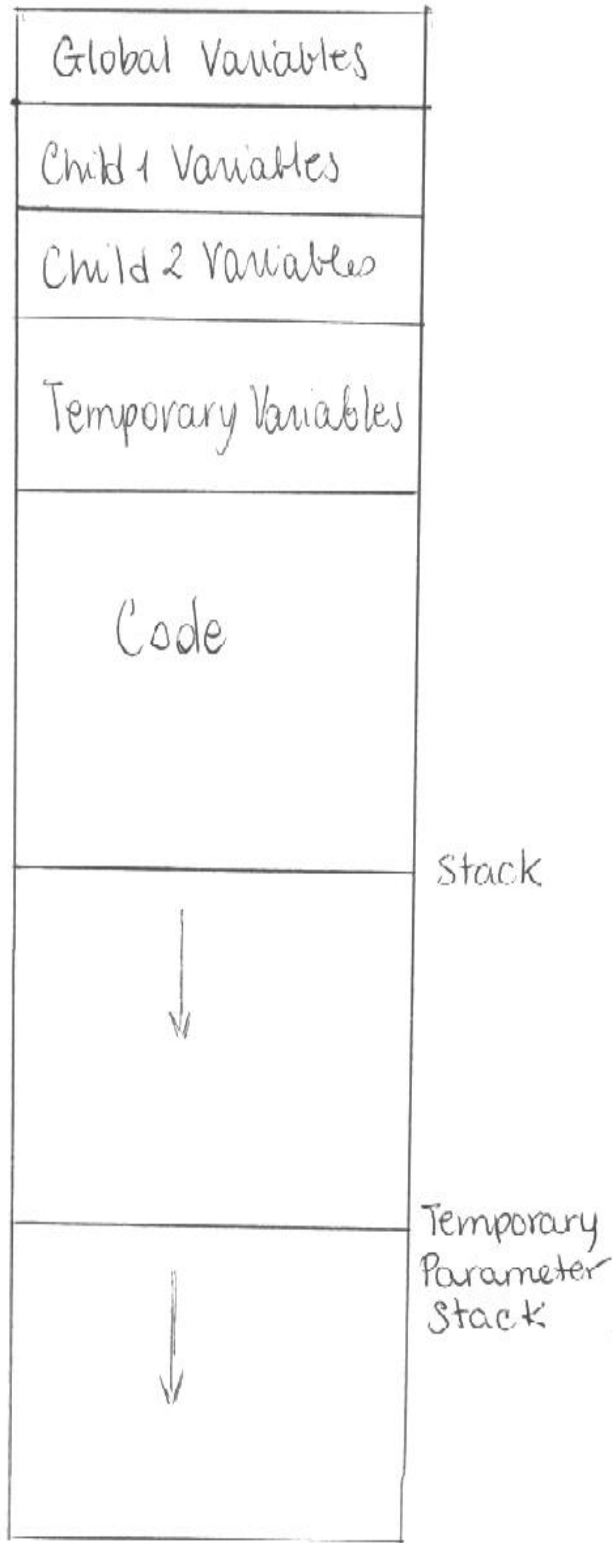
Moon code generation occurs once the intermediate code generation completes. The moon generator uses the Instruction list provided to it by the intermediate code generator. Moon Instructions (moonInstr) are from a vector in which all moon instructions are stored. Furthermore, a number of functions are found in the moon code generator that process all types of intermediate code instructions.

However, there are many aspects of code generation that are not handled by the intermediate code generator and that require more thought. These aspects include parameter passing, variable storage, correct number of passed parameters validation, limit of number of parameter passing, recursion vs. non-recursion. The following table explains what design decisions have been chosen for these features.

Feature	Implementation Details
parameter passing	Pass by value
variable storage	Variables are all stored together at the top of the memory area
correct number of passed parameters validation	The number of parameters are verified when the moon machine code generation is in progress
limit of number of parameter passing	There is a maximum space for variables to be backed up in the stack, the variable storage cannot exceed the stack size
recursion vs. non-recursion	To allow SLANG to become a more powerful language, recursion has been implemented.

To help illustrate the organization of the moon code, the following figure has been provided:

To allow for recursion, the following methods have been used. For a module to be called, its parameters are copied to a Temporary Parameter Stack, where they are left there for the called module to use. The called module is responsible for backing up its Data Content. To back up the Data Content, the Module copies all its variables to the Stack. Afterwards, it copies all the required parameters from the Temporary Parameter Stack to its Data Section, these operations are done in the BeginMod function.



Concordia University	Version: 1.0
	Date: 22/04/04

3.5 Description of the Error Handling

Throughout the parser, an error condition `errc` is used to return from functions when unexpected tokens are found. Furthermore, to follow good programming practices, there is an error handler that handles all errors generated by the compiler. The error handler is called using a single parameter: the error code. This function is then able to decide what error message should be displayed. Errors are output to the screen.

3.6 List of the Errors Recognized by the Compiler

Error No.	Text Output
1001	<code>cerr<<"Token is not of a known type ";</code>
2001	<code>cerr<<"Code ends before file ";</code>
2002	<code>cerr<<"An unexpected token: " << t.tokenStr << " of type: " << t.type <<" was found ";</code>
3001	<code>cerr<<"An identifier: " <<t.tokenStr<< " is being used without being declared. Identifier used ";</code>
3002	<code>cerr<<"Declaration of identifier has already been made. Redclaration occurs ";</code>

4. Evaluation of the Design Decisions and Efficiency

There are some observable areas that could be improved in the compiler. One of these areas is the scanner's `reservedLookup` function. In this function, string comparison is used for the reserved words. A more efficient implementation would use a data structure such as a hash table.

Also, the design of the moon machine could be redesigned in a way that all the parameters do not have to be copied multiple times. This would also for less data access and would improve the efficiency of the generated moon machine code.