CS 4031
Compiler Construction
Lecture 8

Mahzaib Younas

Lecturer, Department Computer Science

FAST NUCES CFD

Operator Precedence Parser

- This is a technique for constructing shift-reduce parsers by hand for a small class of grammars.
- To construct LL(1) parser, there is a restriction on the grammar; grammar should be free of left recursion and should be left factored.

Rules for Operator Precedence Parser

- The grammar that does not contain null production and two adjacent non-terminals on the right hand side of any production is called operator grammar
- In an operator grammar, no production rule can have:
  - Epsilon at the right side
  - Two adjacent non-terminals at the right side

Example:

Not-Suitable grammar Example:

- E → E B E | id
- B → + | − | =

Suitable grammar Example:

- E → E + E | E * E | E = E | id

Shift and Reduce

Let “a” be the top element on stack and “b” is the current element pointed by the input pointer, that is, the look ahead symbol b.

- if $a \lessdot b$ or $a \doteq b$, push b onto the stack and increment input pointer. (Shift action)
- if $a \gtrdot b$ then (Reduce action) Repeat Pop the stack until the top of the stack is $\lessdot$ to the terminal most recently popped
- If a = b = $, announces successful completionss

Precedence:

- Low to high Precedence

+, - , *, /, terminals

Using Operator Precedence Relations

- $E \rightarrow E + E \mid E - E \mid E * E \mid \frac{E}{E} \mid E \mid -E \mid id$

- Then the input string (id + id * id) with the precedence relations inserted will be :

$ <.id.> + <.id> * <.id> $

id + id * id

- Make the Table

	id	**+**	*****	**$**
id		⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	

Operator Precedence Grammar Example

- What is the equivalent operator grammar for the following grammar?
- S → AB
- A → c|d
- B → aAB |b

Example 2:

- Parse the input string a + b * c by using the above algorithm

Traversal

Stack	Input	Action taken
$	a + b * c$	
$a	+ b * c$	shift a because $ ⋖ a
$	+ b * c$	pop a because a ⋗ +
$+	b * c$	shift + because $ ⋖ +
$+b	* c$	shift b because + ⋖ b
$+	* c$	pop b because b ⋖ *
$+*	c$	shift * because + ⋗ *
$+*c	$	Shift c because * ⋖ c
$+*	$	pop c because c ⋖ $
$+	$	pop * because * ⋖ $
$	$	pop + because + ⋖ $
$	$	Accept

Comparison Between Top-Down and Bottom-Up Parser

Grammar		LL(1)	LR(0)	SLR(1)	LALR(1)	CLR(1)
1.	$S \rightarrow a$	√	√	√	√	√
2.	$E \rightarrow E + T \mid T$ $T \rightarrow a$	X	√	√	√	√
3.	$E \rightarrow E + T \mid T$ $T \rightarrow T * F \mid F$ $F \rightarrow id$	X	X	√	√	√
4.	$S \rightarrow Aa \mid bAc \mid dc \mid bda$ $A \rightarrow d$	X	X	X	√	√
5.	$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$ $A \rightarrow d, B \rightarrow d$	X	X	X	X	√
6.	$S \rightarrow a \mid A, A \rightarrow a$	X	X	X	X	X

Symbol Table

Symbol Table

- The symbol table is used to store essential information about every symbol contained within the program.
- This includes:
  - Keywords
  - Data Types
  - Operators
  - Functions
  - Variables
  - Procedures
  - Constants
  - Literal

Contents of the symbol table

- The symbol table will contain the following types of information for the input string in a source program:
  1. The lexem itself
  2. Corresponding tokens
  3. Its semantic component like variable operator constant functions
  4. Data type
  5. Pointers to other entries (when necessary)

Interaction between symbol table and the phases of compiler

- Virtually every phase of the compiler will use the symbol table:
  - Initialization
  - Scanner
  - Parser
  - Semantic action
  - Intermediate code generation
  - Object code generation

Phases of compiler

Phase Name	Usage
Initialization	It place keywords, operators, and standard identifiers.
Scanner/Lexical	Place User defined identifiers and Literals Return Token
Syntax Analysis	Place Token and return AST
Semantic Analysis	Place Data types and use for information checking
Intermediate code	Use pointers to entries in symbol table in creating the
Target Code	Use pointers to entries in symbol table as well as store the address of its procedures and function

Symbol Table

- A data structure used by a compiler to keep track of semantics of name.
  - Determine whether the variable is defined already or not.
  - Determine the scope
    - The effective context where a name is valid.
  - Where it is stored, storage address.
  - Type checking for semantic correctness determination.

Operations

- There common operations of symbol table are

1. Find / lookup / search

   Access the information associated with given name.

2. Insert

   Add a name into the table.

3. Delete

   Remove a name in to the table.

Basic Operation of the Symbol Table

Operation	Detail
Adding new symbol	The reserved words, standard identifiers and operators are placed in the Symbol Table during its initialization. New lexemes are added when the scanner encounters them, and they are assigned a token class. Similarly, the semantic analyzer adds the appropriate properties and attributes that belong to the lexeme
Deleting new symbol	When the compiler is finished with a given scope of the program, all the symbols belonging to that scope must be effectively removed before beginning to process a different scope of the program.
Searching symbol	Looking up a lexeme in the symbol table and retrieving any and all of the properties belonging to that symbol.

Symbol Table

- uses symbol table to keep track of scope (block) and binding information about names
- symbol table is changed (updated) every time
  - if a new name is discovered
  - if new information about an existing name is discovered
- Symbol table must have mechanism to:
  - add new entries
  - find existing information efficiently

Symbol Table

- Two common mechanism:
  - linear lists, simple to implement, poor performance
  - hash tables, greater programming, good performance
- Compiler should be able to grow symbol table dynamically
- If size is fixed, it must be large enough for the largest program

Symbol Table Information

- For each type name, its type definition (eg. for the C type declaration typedef int* mytype, it maps the name mytype to a data structure that represents the type int*)
- For each variable name, its type. If the variable is an array, it also stores dimension information. It may also store storage class, offset in activation record etc.
- For each constant name, its type and value.
- For each function and procedure, its formal parameter list and its output type. Each formal parameter must have name, type, type of passing (by- reference or by-value), etc.

Typical View of Symbol Table

	Variable Name	Address	Type	Dimension	Line Declared	Lines Referenced
1	COMPANY#	0	2	1	2	9, 14, 25
2	X3	4	1	0	3	12, 14
3	FORM1	8	3	2	4	36, 37, 38
4	B	48	1	0	5	10, 11, 13, 23
5	ANS	52	1	0	5	11, 23, 25
6	M	56	6	0	6	17, 21
7	FIRST	64	1	0	7	28, 29, 30, 38

Symbol Table Organization for Non-Block Structured Languages

- Each unit of code is a single module that has no sub module.
- The complexity of symbol table operation is determined by Average length of search which is average number of comparison required to retrieve a record.
- The collection of attributes stored in the table for a given variable is called symbol table record.
- The name of the variable for which an insertion of lookup has to be performed is called search argument.

Symbol Table Organization for Non-Block Structured Languages : Structures

- There are four structures

1. Unordered symbol Table
2. Ordered Symbol Table
3. Tree Structured Symbol Table
4. Hash Symbol Table

Symbol Table

- typedef struct {
- char *name;
- int value;
- }

- Symbol;
- Symbol symbolTable[MAX_SYMBOLS];
- int symbolCount = 0;
- int lookupSymbol(char *name)
- { for (int i = 0; i < symbolCount; i++) { if (strcmp(symbolTable[i].name, name) == 0) { return i; } } return -1;}
- void addSymbol(char *name, int value)
- { if (lookupSymbol(name) == -1) { symbolTable[symbolCount].name = strdup(name); symbolTable[symbolCount].value = value; symbolCount++; }}

- Symbol;
- Symbol symbolTable[MAX_SYMBOLS];
- int symbolCount = 0;
- int lookupSymbol(char *name)
- { for (int i = 0; i < symbolCount; i++) { if (strcmp(symbolTable[i].name, name) == 0) { return i; } } return -1;}
- void addSymbol(char *name, int value)
- { if (lookupSymbol(name) == -1) { symbolTable[symbolCount].name = strdup(name); symbolTable[symbolCount].value = value; symbolCount++; }}