Lecture 9

COP3402 FALL 2015 – DR. MATTHEW GERBER – 10/5/2015

FROM EURIPIDES MONTAGNE, FALL 2014

Tonight

The Scanner, Part 1

The Scanner's Job

Decompose the program's source code into its elementary tokens.

- Read the code, one character at a time
- Group characters into tokens
- Remove white space, comments and control characters
- Encode token types
- Generate the symbol table
- Detect errors and generate error messages

Lexical Analysis

ID	Name	Type
1	fahrenheit	real
2	celsius	real

Recall the statement:

```
fahrenheit := 32 + celsius * 1.8;
```

In the presence of the symbol table at the top right, the scanner reads this and translates it into a stream of tokens:

[SYMBOL ID 1] [:=] [INT 32] [+] [SYMBOL ID 2] [*] [REAL 1.8] [;]

The scanner gets rid of white space and comments.

Note: We've cheated a bit with the symbol table here. Can you figure out how?

Scanner Design: First Steps

Consider what you're going to need to scan, and what it's made up of.

That means, first of all, figuring out what kind of tokens we're going to process.

- In the case of PL/0, we've already seen a lot of these
  - **Identifiers** and **Literal Values**
  - **Reserved Words:** begin, end, if, then, while, do, call, const, var, procedure, write, read, else
  - **Arithmetic Operators:** + - * /
  - **Comparisons:** odd = <> < <= > >=
  - **Syntax** and **Assignment**: ( ) , ; . :=

Example definitions

(these are incomplete, you may or may not use them depending on your approach)

```
/* list of reserved word names */
char *word [ ] = { "null“, "begin“, "call", “const”, “do’, “else”, “end”, “if”,
                   “odd”, “procedure”, “read”, “then”, “var”, “while”, “write”};

/* internal representation  of reserved words */
int  wsym [ ] =  { nul, beginsym, callsym, constsym, dosym, elsesym, endsym, ifsym,
                   oddsym, procsym, readsym, thensym, varsym, whilesym, writesym };

/* list of special symbols */
int ssym[256]

ssym['+']=plus;      ssym['-']=minus;      ssym['*']=mult;
ssym['/']=slash;     ssym['(']=lparen;     ssym[')']=rparen;
ssym['=']=eql;       ssym[',']=comma;      ssym['.']=period;
ssym['#']=neq;       ssym['<']=lss;        ssym['>']=gtr;
ssym['$']=leq;       ssym['%']=geq;        ssym[';']=semicolon;
```

Whether you use those or not…

…you will need some way to deal with everything you're going to detect.

We'll discuss a general approach in a bit, but first let's nail down at least one thing for you. PL/0 has a nicely small set of tokens, so we can actually give you a list of them…

A Partial Token Table

Category	Lexeme	Token Name	Numerical Value
		nulsym	1
Literals and Identifiers	letter (letter | digit)*	identsym	2
	(digit)+	numbersym	3
Arithmetic Operators	+	plussym	4
	-	minussym	5
	*	multsym	6
	/	slashsym	7
Comparisons	odd	oddsym	8
	=	equalsym	9
	<>	neqsym	10
	<	lessym	11
	<=	leqsym	12
	>	gtrsym	13
	>=	geqsym	14
Syntax and Assignment	(	lparentsym	15
	)	rparentsym	16
	,	commasym	17
	;	semicolonsym	18
	.	periodsym	19
	:=	becomesym	20

Here's a partial token table for PL/0.

- The **PL/0 reference document** you have been provided contains the complete version of this token table. Use it for your actual work, not this one.

Each lexeme becomes a token; each token has a numerical value. This will be important.

A **Provided** Definition

(you *should* use this one)

```
typedef enum {
   nulsym = 1, identsym, numbersym, plussym, minussym,
   multsym,  slashsym, oddsym, eqsym, neqsym, lessym, leqsym,
   gtrsym, geqsym, lparentsym, rparentsym, commasym, semicolonsym,
   periodsym, becomessym, beginsym, endsym, ifsym, thensym,
   whilesym, dosym, callsym, constsym, varsym, procsym, writesym,
   readsym, elsesym
} token_type;
```

Other **Example** Definitions

(you definitely won't use these)

```
#define  norw      15     /* number of reserved words */
#define  imax   32767     /* maximum integer value */
#define  cmax      11     /* maximum number of chars for idents */
#define  nestmax    5     /* maximum depth of block nesting */
#define  strmax   256     /* maximum length of strings */
```

The Symbol Table

The *symbol table or name table* records information about every defined symbol in the program.

- Each piece of information associated with a name is called an *attribute*
- Attributes might include:
  - Variable type
  - Number of parameters in a procedure
  - Number of dimensions for an array

There are a number of ways to represent a symbol table.

- Most real-world compilers use hash tables
- Linked lists, trees, or tries in various forms can also be reasonable choices
- *You* will have few enough symbols that you can just iterate through an array
- You'll talk about all those methods in your labs (eventually)

Using the Symbol Table

Any data structure supports some subset of four operations:

- Insertion, modification, retrieval and deletion
- For a symbol table, we are most concerned with insertion and retrieval
  - We will also need modification eventually, so make sure your retrieval retrieves a reference of some kind

When a *declaration* is processed, the name is inserted into the symbol table

- If the language doesn't require declarations, we insert the name the first time we run into it

Each time the name is subsequently used, we *look up* the symbol

Symbol Table Structure

```
#define MAX_SYMBOL_TABLE_SIZE 100

/* For constants, store kind, name and val
   For variables, store kind, name, L and M
   For procedures, store kind, name, L and M */

typedef struct symbol {
       int kind;              // const = 1, var = 2, proc = 3
       char name[12]; // name up to 11 chars
       int val;               // value
       int level;     // L level
       int addr;              // M address
} symbol;

symbol symbol_table[MAX_SYMBOL_TABLE_SIZE];
```

Symbol Table Caveat

We're talking about the symbol table now because under *most* circumstances, the symbol table is generated by a combination of the scanner and parser.

However, in your PL/0 compiler, the symbol table will be generated exclusively by the parser.

- (The way PL/0's scoping rules work just makes it too messy to do it any other way)

So that said, let's talk about a completely different caveat.

Lookahead

Lookahead plays an important role in a scanner.

- Let's say we're beginning to read a token, and we read the letter "i".
- Is this:
  - Identifier "i"?
  - Identifier "iList"?
  - Reserved word "if"?
- We have no way of telling without looking forward.

Regular Expressions

…and fortunately, you've already started looking at how to do that.

Next Time: The Scanner, Part 2