

CS 415 Project - Spring 2000

Department of Computer Science
Rutgers University

1 Introduction

This semester, you will build a compiler for a language that is a small subset of Pascal. Here's an example program in this language:

```
program Demo;
  /* prints 4, 4, 1, 4, 8, 4 */
  var
    A: integer;
  function P(X: integer): integer;
  begin
    writeln(A);
    writeln(X - 3);
    P := X - 3;
    writeln(A)
  end;
begin
  A := 4;
  writeln(A);
  writeln(7+P(A));
  writeln(A)
end.
```

This language differs from Pascal in the following ways. Program headings contain no list of input and output files. Blocks have only variable and procedure/function declarations and compound statements. Types are limited to the scalars, integer, character, and boolean, and single dimensional arrays of scalars indexed by integers. Only the following statements are included: while, if, procedure call, assignment, and compound. Operators are restricted to arithmetic, logical, and relational.

Here's the grammar for our language:

```
start      ::= program ID ; block .
block      ::= variables procdcls cmpdstmt
procdcls   ::= procdcls procdcl | <empty string>
procdcl    ::= procedure ID parmlist ; block ;
           | function ID parmlist : stype ; block ;
parmlist   ::= ( parms ) | <empty string>
parms      ::= parms ; parm | parm
parm       ::= var vardcl | vardcl
variables  ::= var vardcls | <empty string>
vardcls    ::= vardcls vardcl ; | vardcl ;
vardcl     ::= idlist : type
idlist     ::= idlist , ID | ID
```

```

type      ::= array [ integer_constant .. integer_constant ] of stype | stype
stype     ::= integer | char | boolean
stmtlist  ::= stmtlist ; stmt | stmt
stmt      ::= ifstmt | wstmt | astmt | procstmt | cmpdstmt | writestmt
writestmt ::= writeln ( exp )
procstmt  ::= ID optexplist
ifstmt    ::= if exp then stmt else stmt | if exp then stmt
wstmt     ::= while exp do stmt
cmpdstmt  ::= begin stmtlist end
astmt     ::= lvalue := exp
optexplist ::= ( explist ) | <empty string>
explist   ::= explist , exp | exp
exp       ::= exp + exp | exp - exp | - exp | exp * exp | exp div exp
           | exp != exp | exp == exp | exp >= exp | exp > exp | exp < exp
           | exp <= exp | exp and exp | exp or exp | exp exor exp
           | not exp | ( exp ) | ID ( explist ) | lvalue | constant
lvalue    ::= ID | ID [ exp ]
constant  ::= integer_constant | char_constant | true | false

```

You will build the compiler in four phases according to the following schedule:

Phase	Due Date
1. Lexical analyzer (scanner)	7pm Wed 2/09/2000
2. Syntax analyzer (parser)	7pm Wed 3/01/2000
3. Semantic analyzer and symbol table	7pm Wed 3/22/2000
4. Code generator	7pm Wed 4/19/2000

Here's the first assignment, due by 2/9/2000, 7pm. BTW, my policy on late assignments is: **THERE IS NO LATE ASSIGNMENT POLICY.** That is, I will **NOT** accept any late assignments, whether they are 1 minute late or 10 days late. If you don't hand in the assignment by the time it's due, you will receive a 0 for that assignment. Please be clear on this; there will be no changes! Instructions for handing in the assignment will be posted on the web.

2 Building a Scanner

You will write a scanner for our language. Note that comments, that is, text that appear between the `/*` and `*/` symbols, are discarded by the scanner and so are not defined in our grammar.

You will use flex, an automatic scanner generator, to build your scanner. To learn how to use flex, read the man page (man flex), read the "lex & yacc" book, and ask Manju and I. I will not be going over flex in detail in class. For this assignment, you will write a file `scan.l` that will contain the rules for flex to build your scanner for you.

We will provide the following:

- `scan.l`: just a few lines to get you started.
- helper files: `attr.h`, `parse.tab.h`, and `scandriver.c`. `attr.h` contains the type definition for tokens. `parse.tab.h` mostly contains constants used to denote specific constants, and `scandriver.c` contains the driver to run your scanner.
- Makefile: in your working directory, type "make" to compile and link your scanner. This will generate a program called "scanner". If you don't know how make work, you should really learn a little bit about it.

- demo input and out files: example programs that you can run your scanner on and example output files for you to check the output of your scanner.

To get the above material, you should download the file `project1.tar.gz` from the class web.

With the given driver and make file, you will run the scanner as “`scanner < demo1`”, where `demo1` is a file containing code to be scanned. The code generated by flex (based on `scan.l`) produces a line-numbered version of the original program interspersed with diagnostics indicating any illegal characters, and returns a sequence of tokens to `scandriver.c`. The driver program writes the tokens to a file named `scanner.out`. For example, the input

```
PrOceDure AlongId;
    /* this should
       say what the proc does */
begin Q .. : 99 = 'hello' (+) [-] . end
```

produces the following two outputs:

```
From scan.l (to standard output)
%scanner < demo3
1      PrOceDure AlongId;
2          /* this should
3              say what the proc does */
4      begin Q .. : 99 = 'hello' (+) [-] . end
```

```
From scandriver.c (to the file scanner.out)
262
278      AlongId
59
260
278      Q
274
58
280      99
61
279      'hello'
40
43
41
91
45
93
271
261
```

You should detect the following illegal characters: `!`, `@`, `#`, `$`, `%`, `^`, `&`, `|`, `?`. However, these characters are legal in comments and character constants. You should detect EOFs in comments and EOLs in string constants. However, when such errors are encountered, the values in the `scanner.out` file are unpredictable.

```
%scanner < demo1
1      *and+/** /'hello world'
2      keep going
ERROR: EOF detected in comment
%scanner < demo2
```

```
1      *and+'/** /hello world
ERROR: character string not terminated
2      this = *this;
```

Finally, to make your projects easier to grade, your scanner should return the constants defined in `parse.tab.h` as values for the tokens indicated below. Your scanner should not be case-sensitive. This is accomplished by specifying the “-i” option for flex in the Makefile. You should return ascii codes for the literals `[`, `]`, `*`, `+`, `(`, `)`, `comma`, `:`, `;`, `=`, and `-`.

INT	integer
CHAR	char
BOOL	boolean
BEG	begin
END	end
PROC	procedure
FUNC	function
PROG	program
VAR	var
ASG	:=
ARRAY	array
IF	if
THEN	then
ELSE	else
PERIOD	.
WHILE	while
DO	do
RANGE	..
OF	of
WRITELN	writeln
DIV	div
ID	an identifier (begins with letter, then letters and digits)
CCONST	a character constant (in single quotes, no embedded quotes)
ICONST	an integer constant
EQ	==
NEQ	!=
LT	<
LEQ	<=
GT	>
GEQ	>=
AND	and
OR	or
EXOR	exor
NOT	not
TRUE	true
FALSE	false