Compiler Project Source Language

Kenneth Sundberg

October 17, 2013

Dedicated to Dr. Steven J. Allan

Abstract

CPSL (Compiler Project Source Language) is the programming language implemented in the Complier Construction course at Utah State University (CS 5300). CPSL is a strongly typed, static scoped, block structured, high level language in the spirit of Pascal.

Acknowledgement

With permission, this work is largely derived from the CPSL language created for CS 5300 by Dr. Allan. The mistakes are mine.

Contents

List of Tables

Chapter 1

Definitions

The character set of CPSL is ASCII. In the following *letter* denotes any upper- or lower-case letter, and *digit* denotes any of the ten decimal digits 0 through 9.

The syntax of CPSL is expressed in a EBNF-like format as follows:

⟨*Nonterminal*⟩ → ⟨*Nonterminal*⟩ `token` ⟨*token with state*⟩
| ⟨*Optional*⟩? ⟨*Repeated*⟩*
| ⟨*empty*⟩

Within this format non-terminal symbols will be formatted in mixed case and enclosed in angle brackets. Terminals without state, such as keywords, will be formatted in bold. Terminals with state, such as identifiers, will be formatted in lower case and enclosed in angle brackets. Symbol groups will be indicated with parenthesis. Optional symbols or symbol groups will be indicated with a question mark. Alternatives for a symbol or symbol group will be indicated with a vertical bar. The terminal *empty* in a production indicates an empty production.

Chapter 2

Lexical Structure

2.1 Introduction

The following is a detailed description of the lexical structure of CPSL. Any lexeme not described herein is an error and should result in an appropriate diagnostic message.

2.2 Keywords

Keywords are strings with predefined meaning. They can not be redefined by the user. These keywords are either all capitals or all lower-case, mixed-case variations are not keywords. For example, BEGIN and begin are keywords while Begin is not.

array	begin	chr	const	do	downto
else	elseif	end	for	forward	function
if	of	ord	pred	procedure	read
record	repeat	return	stop	succ	then
to	type	until	var	while	write

Table 2.1: Keywords of CPSL

2.3 Identifiers

Identifiers in CPSL consist of a letter, followed by a sequence of zero or more letters, digits, or underscores. Upper- and lower-case letters are considered **distinct**. There is **no limit** on the length of identifiers in CPSL.

2.3.1 Predefined Identifiers

CPSL has a small set of predefined identifiers. Unlike keywords, the meaning of these identifiers may be altered by the user.

Identifier		Meaning
integer	INTEGER	Basic Integer type
char	CHAR	Basic Character type
boolean	BOOLEAN	Basic Boolean type
string	STRING	Basic String type
true	TRUE	Boolean constant
false	FALSE	Boolean constant

Table 2.2: Predefined Identifiers in CPSL

2.4 Operators and Delimiters

CPSL has a set of symbols used as operators and delimiters. These characters are not a valid part of any keyword or identifier.

+	−	*	/	&	|	~	=
<>	<	<=	>	>=	.	,	:
;	(	)	[	]	:=	%	

Table 2.3: Operators and Delimiters of CPSL

2.5 Constants

2.5.1 Integer Constants

Integer Constants in CPSL are of three forms, each denoting an integer in a different base.

- A sequence of digits beginning with a 0 is interpreted as an octal number
- A 0x followed by a sequence of digits is interpreted as a hexadecimal number
- Any other sequence of digits is interpreted as a decimal value

2.5.2 Character Constants

A character constant represents a *single* character and is enclosed in a pair of single quotes. A character constant may not be blank, the lexeme consisting of a pair of single quotes is an error.

2.5.3 String Constants

A string constant represents a multi-character sequence and is enclosed in a pair of double quotes. String constants may not contain double quotes. A string constant may be empty.

2.5.4 Representing Characters in Character and String Constants

The newline character (ASCII 10) may not appear between the single or double quotes of a character or string constant. Any printable character (ASCII 32 to 126 inclusive) can be represented as itself with the exception of \. Also such a constant can contain a \followed by any printable character. Such a \- escaped sequence is interpreted as the character after the \with the following exceptions:

\n line feed

\r carriage return

\b backspace

\t tab

\f form feed

2.5.5 Comments

A comment in CPSL begins with a $ and continues to the end of the line.

2.5.6 Blanks, Tabs, Spaces, and New Lines

Blanks, tabs, spaces, and new lines (white space) delimit other tokens but are otherwise ignored. This does not hold inside character and string constants where such characters are interpreted as themselves.

Chapter 3

Syntactic Structure

The overall structure of a CPSL program is as follows:

⟨*Program*⟩ → ⟨*ConstantDecl*⟩? ⟨*TypeDecl*⟩? ⟨*VarDecl*⟩?
(⟨*ProcedureDecl*⟩ | ⟨*FunctionDecl*⟩)* ⟨*Block*⟩ .

3.1 Declarations

3.1.1 Constant Declarations

⟨*ConstantDecl*⟩ → `const` (⟨*ident*⟩ `=` ⟨*ConstExpression*⟩ `;`)+

3.1.2 Procedure and Function Declarations

Procedure and Function Declarations have the following structure:

⟨*ProcedureDecl*⟩ → `procedure` ⟨*ident*⟩ `(` ⟨*FormalParameters*⟩ `)` `;` forward `;`
| `procedure` ⟨*ident*⟩ `(` ⟨*FormalParameters*⟩ `)` `;` `<Body>` `;`

⟨*FunctionDecl*⟩ → `function` ⟨*ident*⟩ `(` ⟨*FormalParameters*⟩ `)` `:` ⟨*Type*⟩ `;` forward `;`
| `function` ⟨*ident*⟩ `(` ⟨*FormalParameters*⟩ `)` `:` ⟨*Type*⟩ `;` `<Body>` `;`

⟨*FormalParameters*⟩ → ⟨*empty*⟩
| `var`? ⟨*IdentList*⟩ `:` ⟨*Type*⟩ (`;` `var`? ⟨*IdentList*⟩ `:` ⟨*Type*⟩)*

⟨*Body*⟩ → ⟨*ConstantDecl*⟩? ⟨*TypeDecl*⟩? ⟨*VarDecl*⟩? Block

⟨*Block*⟩ → `begin` ⟨*StatementSequence*⟩ `end`

Notice that procedure and function definitions *cannot* be nested. In other words, we have only a global environment and a local environment. Procedures and functions can only be defined in the global environment. In the local environment we can only define constants, types, and variables. Notice also the

provision for accommodating **forward** references to procedures and functions. Notice also that both procedures and functions requires parentheses in the definition even if there are no parameters. This is also true for their invocations.

3.1.3 Type Declarations

In CPSL there are four predefined types (see figure 2.2), and two type constructors. The type constructors are array and record.

⟨*TypeDecl*⟩ → `type` (⟨*ident*⟩ `=` ⟨*Type*⟩ `;`)+

⟨*Type*⟩ → ⟨*SimpleType*⟩
| ⟨*RecordType*⟩
| ⟨*ArrayType*⟩

⟨*SimpleType*⟩ → ⟨*ident*⟩

⟨*RecordType*⟩ → `record` (⟨*IdentList*⟩ `:` ⟨*Type*⟩ `;`)* `end`

⟨*ArrayType*⟩ → `array [` ⟨*ConstExpression*⟩ `:` ⟨*ConstExpression*⟩ `] of` ⟨*Type*⟩

⟨*IdentList*⟩ → ⟨*ident*⟩ (`,` ⟨*ident*⟩)*

3.1.4 Variable Declarations

⟨*VarDecl*⟩ → `var` (⟨*IdentList*⟩ `:` ⟨*Type*⟩ `;`)+

3.2 CPSL Statements

Statements in CPSL have the following syntax:

⟨*StatementSequence*⟩ → ⟨*Statement*⟩ (`;` ⟨*Statement*⟩)*

⟨*Statement*⟩ → ⟨*Assignment*⟩
| ⟨*IfStatement*⟩
| ⟨*WhileStatement*⟩
| ⟨*RepeatStatement*⟩
| ⟨*ForStatement*⟩
| ⟨*StopStatement*⟩
| ⟨*ReturnStatement*⟩
| ⟨*ReadStatement*⟩
| ⟨*WriteStatement*⟩
| ⟨*ProcedureCall*⟩
| ⟨*NullStatement*⟩

⟨*Assignment*⟩ → ⟨*LValue*⟩ `:=` ⟨*Expression*⟩

⟨*IfStatement*⟩ → `if` ⟨*Expression*⟩ `then` ⟨*StatementSequence*⟩ (`elseif` ⟨*Expression*⟩ `then` ⟨*StatementSequence*⟩)* (`else` ⟨*StatementSequence*⟩)? `end`

⟨*WhileStatement*⟩ → `while` ⟨*Expression*⟩ `do` ⟨*StatementSequence*⟩ `end`

⟨*RepeatStatement*⟩ → `repeat` ⟨*StatementSequence*⟩ `until` ⟨*Expression*⟩

⟨*ForStatement*⟩ → `for` ⟨*ident*⟩ `:=` ⟨*Expression*⟩ (`to`|`downto`) ⟨*Expression*⟩ `do` ⟨*StatementSequence*⟩ `end`

⟨*StopStatement*⟩ → `stop`

⟨*ReturnStatement*⟩ → `return` ⟨*Expression*⟩?

⟨*ReadStatement*⟩ → `read (` ⟨*LValue*⟩ (`,` ⟨*LValue*⟩)* `)`

⟨*WriteStatement*⟩ → `write (` ⟨*Expression*⟩ (`,` ⟨*Expression*⟩)* `)`

⟨*ProcedureCall*⟩ → ⟨*ident*⟩ `(` (⟨*Expression*⟩ (`,` ⟨*Expression*⟩)*)? `)`

⟨*NullStatement*⟩ → ⟨*empty*⟩

3.3 Expressions

⟨*Expression*⟩ → ⟨*Expression*⟩ `|` ⟨*Expression*⟩
| ⟨*Expression*⟩ `&` ⟨*Expression*⟩
| ⟨*Expression*⟩ `=` ⟨*Expression*⟩
| ⟨*Expression*⟩ `<>` ⟨*Expression*⟩
| ⟨*Expression*⟩ `<=` ⟨*Expression*⟩
| ⟨*Expression*⟩ `>=` ⟨*Expression*⟩
| ⟨*Expression*⟩ `<` ⟨*Expression*⟩
| ⟨*Expression*⟩ `>` ⟨*Expression*⟩
| ⟨*Expression*⟩ `+` ⟨*Expression*⟩
| ⟨*Expression*⟩ `-` ⟨*Expression*⟩
| ⟨*Expression*⟩ `*` ⟨*Expression*⟩
| ⟨*Expression*⟩ `/` ⟨*Expression*⟩
| ⟨*Expression*⟩ `%` ⟨*Expression*⟩
| `~` ⟨*Expression*⟩
| `-` ⟨*Expression*⟩
| `(` ⟨*Expression*⟩ `)`
| ⟨*ident*⟩ `(` (⟨*Expression*⟩ (`,` ⟨*Expression*⟩)*)? `)`
| `chr (` ⟨*Expression*⟩ `)`
| `ord (` ⟨*Expression*⟩ `)`
| `pred (` ⟨*Expression*⟩ `)`
| `succ (` ⟨*Expression*⟩ `)`
| ⟨*LValue*⟩

⟨*LValue*⟩ → ⟨*ident*⟩ ((`.` ⟨*ident*⟩)|(`[` ⟨*Expression*⟩ `]`))*

⟨*ConstExpression*⟩ → ⟨*ConstExpression*⟩ `|` ⟨*ConstExpression*⟩
| ⟨*ConstExpression*⟩ `&` ⟨*ConstExpression*⟩
| ⟨*ConstExpression*⟩ `=` ⟨*ConstExpression*⟩
| ⟨*ConstExpression*⟩ `<>` ⟨*ConstExpression*⟩
| ⟨*ConstExpression*⟩ `<=` ⟨*ConstExpression*⟩
| ⟨*ConstExpression*⟩ `>=` ⟨*ConstExpression*⟩
| ⟨*ConstExpression*⟩ `<` ⟨*ConstExpression*⟩
| ⟨*ConstExpression*⟩ `>` ⟨*ConstExpression*⟩
| ⟨*ConstExpression*⟩ `+` ⟨*ConstExpression*⟩
| ⟨*ConstExpression*⟩ `-` ⟨*ConstExpression*⟩
| ⟨*ConstExpression*⟩ `*` ⟨*ConstExpression*⟩
| ⟨*ConstExpression*⟩ `/` ⟨*ConstExpression*⟩
| ⟨*ConstExpression*⟩ `%` ⟨*ConstExpression*⟩
| `~` ⟨*ConstExpression*⟩
| `-` ⟨*ConstExpression*⟩
| `(` ⟨*ConstExpression*⟩ `)`
| ⟨*intconst*⟩
| ⟨*charconst*⟩
| ⟨*strconst*⟩
| ⟨*ident*⟩

To resolve ambiguities in this grammar the following suffices:

- Arithmetic and Boolean binary operators are left-associative
- Relational operators are non-associative
- Unary minus and Boolean not are right-associative
- Operators have the following precedence (decreasing order)
  - Unary − (negation)
  - * / %
  - + −
  - = <> < <= > >=
  - ~
  - &
  - |