Macquarie University, Department of Computing

COMP332 Programming Languages 2019

weBCPL, a retro programming language for the web

Introduction

BCPL is described by its inventor, Martin Richards, as being a *"simple systems programming language with a small fast compiler which is easily ported to new machines. The language was first implemented in 1967 and has been in continuous use since then. It is typeless and provides machine independent pointer arithmetic allowing a simple way to represent vectors and structures."*

The name BCPL is an acronym for **B**asic **C**ombined **P**rogramming **L**anguage and it was designed to be a simplified version of a more sophisticated, multi-paradigm language called **CPL**. Specifically, its original purpose was to provide a simple language for writing an operating environment and compiler for CPL.

BCPL is of particular importance in the history of programming languages, as it is the direct ancestor of the programming languages **B**, which is now pretty much forgotten, and **C**, which is still the most used systems programming language (and is beloved of COMP202 students). What always strikes us as remarkable is that so much of the programming culture that surrounds C was actually pioneered by Martin Richards, and his collaborators, in the development of BCPL.

Dom also has a particular fondness for BCPL. It is not a language that he would use to write serious code in nowadays, but it is the first language that he wrote a compiler for.

WebAssembly, on the other hand, is described, in its specification, as a *"safe, portable, low-level code format designed for efficient execution and compact representation. Its main goal is to enable high performance applications on the Web, but it does not make any Web-specific assumptions or provide Web-specific features, so it can be employed in other environments as well."*

Our plan is to build a simple compiler to translate BCPL into WebAssembly, and to run a few simple BCPL programs in our browsers. In assignment 2 we will construct a BCPL parser and in assignment 3 we will build a code generator for our compiler targeting WebAssembly.

```
    weBCPL := BCPL + WebAssembly
```

References

- Martin Richards, **The BCPL Cintsys and Cintpos User Guide**, *Computer Laboratory University of Cambridge*, 19 August 2019
- Andreas Rossberg (editor), **WebAssembly Specification**, *WebAssembly Community Group*, Version 1.0, 30 May 2019

The weBCPL language

In essence, weBCPL is largely compliant with the 1980 version of the BCPL language described in the classic text BCPL: The Language and its Compiler (1980). It adopts the modernised syntactic conventions of the post-2013 variant of BCPL described in The BCPL Cintsys and Cintpos User Guide. The latter document will be our primary reference for the BCPL language, and we shall refer to it as the BCPL Manual from hereon.

The BCPL language is described in detail in section 2 of the BCPL Manual, the primary differences between the language described there and weBCPL are:

1. weBCPL does not implement the floating point extension (Section 2.6 of the BCPL Manual). All variables in weBCPL contain 32-bit words which the arithmetic operations `+`, `-`, `*`, `/`, `MOD` and `ABS` treat as signed integers in 2s-complement form.
2. weBCPL does not implement the field selection operation `OF` or the associated literal constant form `SLCT` (Section 2.2.2, page 16 of the BCPL Manual).
3. weBCPL string constants only provide a UTF-8 (not a GB2312) unicode mode and they adopt a different convention for the specification of unicode character escapes (Section 2.2.2, page 17 of the BCPL Manual). These are given in terms of unicode codepoint numbers rather than UTF-8 byte sequences.
4. weBCPL maintains a strict distinction between the identifiers used to name variables and functions and those that are used to name the **labels** that may be attached to statements. In weBCPL labels are not classified as expressions, so they have no value and they cannot appear in positions where expressions are expected. `GOTO` statements may only make jumps to labels that are visible within the current scope.
5. weBCPL lacks a `GET` directive (Section 2.1.2 of the BCPL Manual) and conditional compilation directives (Section 2.1.3 of the BCPL Manual).
6. weBCPL also lacks the ability to compile BCPL modules separately and then to link the resulting object files together at a later point in time. While this facility is something to be desired, its implementation would just take us too far afield in these assignments. So for the moment all of the code for each complete weBCPL programs must be contained in a single source file.

We should also say that the BCPL Manual classifies all BCPL constructs into one of three classes: **declarations**, **commands**, and **expressions**. In weBCPL we do the same, except that we shall use the more standard term **statement** instead of command, largely in order to maintain consistency with the use of these terms in the COMP332 lecture notes.

Beyond these relatively minor differences / restrictions, our aim is that weBCPL should stick as closely as possible to the BCPL language described in Section 2 of the BCPL Manual. In particular, it should be able to correctly parse and translate a great proportion of legacy BCPL code.

Whether that translated code will then execute correctly on a WebAssembly platform (your browser) is another matter. That depends upon the extent to which we have time to map the BCPL standard library (Section 3 of the BCPL Manual) onto the facilities provided by your browser. In assignment 3 we will provide a minimal standard library, which will allow your weBCPL programs to read from and write to a simple terminal executing in your browser.

weBCPL syntax

Here is a complete context-free grammar for the weBCPL language, upon which the parser of a weBCPL compiler should be based:

```
program : (declaration ";")* declaration.

// Grammar of BCPL Declarations

declaration :
      "MANIFEST" "{" (manifestEntry ";")* manifestEntry "}"
    | "GLOBAL" "{" (globalEntry ";")* globalEntry "}"
    | "STATIC" "{" (staticEntry ";")* staticEntry "}"
    | "LET" (letDeclClause "AND")* letDeclClause

manifestEntry : idndef ("=" expression)?
globalEntry : idndef (":" integerConst)?
staticEntry : idndef ("=" expression)?

letDeclClause :
      (idndef ",")* idndef "=" (expression ",")* expression
    | idndef "=" "VEC" expression
    | idndef "(" ((idndef ",")* idndef)? ")" "=" expression
    | idndef "(" ((idndef ",")* idndef)? ")" "BE" statement

// Grammar of BCPL Statements

statement :
      labdef ":" statement
    | "CASE" expression ":" statement
    | "DEFAULT" ":" statement
    | unlabelledStmt
```

```
unlabelledStmt : repeatableStmt | iteratedStmt | testStmt

iteratedStmt :
      "UNTIL" expression "DO" statement
    | "WHILE" expression "DO" statement
    | "FOR" idndef "=" expression "TO" expression
          ("BY" expression)? "DO" statement

testStmt :
      "TEST" expression "THEN" statement "ELSE" statement
    | "IF" expression "DO" statement
    | "UNLESS" expression "DO" statement

repeatableStmt :
      repeatableStmt "REPEAT"
    | repeatableStmt "REPEATWHILE" expression
    | repeatableStmt "REPEATUNTIL" expression
    | simpleStmt

simpleStmt :
      (expression ",")* expression ":=" (expression ",")* expression
    | callExp
    | "BREAK" | "LOOP" | "ENDCASE" | "RETURN" | "FINISH"
    | "GOTO" labuse
    | "RESULTIS" expression
    | "SWITCHON" expression "INTO" blockStmt
    | blockStmt

blockStmt : "{" (declaration ";")* (statement ";")* statement "}"

// Grammar of BCPL expressions

expression :
      "VALOF" statement
    | "TABLE" (expression ",")* expression
    | expression "->" expression "," expression
    | expression "EQV" expression
    | expression "XOR" expression
    | expression "|" expression
    | expression "&" expression
    | "NOT" expression
    | expression "<<" expression
    | expression ">>" expression
    | expression "~=" expression
    | expression "<=" expression
    | expression ">=" expression
    | expression "=" expression
    | expression "<" expression
    | expression ">" expression
    | expression "+" expression
    | expression "-" expression
    | "-" expression
    | "+" expression
    | "ABS" expression
    | expression "*" expression
    | expression "/" expression
    | expression "MOD" expression
    | "!" expression
    | "%" expression
    | "@" expression
    | expression "!" expression
    | expression "%" expression
    | callExp
    | elemExp

callExp
    : (callExp | elemExp) "(" ((expression ",")* expression)? ")"

elemExp
    : "(" expression ")"
    | "TRUE" | "FALSE" | "?"
    | integerConst
    | charConst
    | stringConst
    | idnuse

idnuse : identifier
idndef : identifier
labuse : identifier
labdef : identifier
```

Precedence and associativity The grammar above is not immediately suitable for encoding as a parser. The `expression` non-terminal is ambiguous since it makes no allowance for precedence and associativity of the operators. You should rewrite that grammar production to implement the precedence and associativity rules described in the following table. This lists operators in order of precedence from highest (most binding) to lowest (least binding):

Operators	Description	Precedence	Associativity
function call, variable, parenthesised expression, `TRUE`, `FALSE`, and integer / character / string constants	primary expression	13	N/A
binary `!` and `%`	vector expression	12	left
unary `!`, `%` and `@`	address expression	11	N/A
binary `*`, `/` and `MOD`	multiplicative expression	10	left
unary `+`, `-` and `ABS`	unary additive expression	9	N/A
binary `+` and `-`	additive expression	8	left
binary `=`, `~=`, `<`, `>`, `<=` `>=`	relational expression	7	left (see note 1 below)
binary `<<` and `>>`	bit shifting expression	6	left
unary `NOT`	not expression	5	N/A
binary `&`	and expression	4	left
binary	or expression	3	left
binary `EQV` and `XOR`	xor/eqv expression	2	left
ternary `X->Y,Z`	if expression	1	right (see note 2 below)
`VALOF` and `TABLE` expressions	top level expression	0	N/A

Note 1: In BCPL we can write expressions consisting of sequences of relations, for example `a < b = c >= d`. The BCPL parser converts such sequences into a sequence of individual relations separated by **and** (`&`) operations. Here are some examples of this conversion:

- `a < b` converts to `a < b`
- `a < b = c` converts to `(a < b) & (b = c)`
- `a < b = c >= d` converts to `((a < b) & (b = c)) & (c >= d)`
- `a < b = c >= d ~= e` converts to `(((a < b) & (b = c)) & (c >= d)) & (d ~= e)`

Note 2: Right associativity of the ternary if expression implies, in particular, that expressions of the form `X -> Y, Z -> U, V` are parsed as `X -> Y, (Z -> U, V)`.

Lexical analysis The BCPL Manual specifies a few syntactic rules that are not presented in the grammar above:

- The following keywords / operation symbols may be used interchangeably:

  - `NOT` or `~` for the boolean / logical not operation,
  - `MOD` or `REM` for the arithmetic modulus / remainder operation,
  - `XOR` or `NEQV` for the boolean / logical exclusive or operation,
  - `{` or `$(` for the opening of a code block (these are called *section brackets* in BCPL jargon), and
  - `}` or `$)` for the closing of a code block.

- The keywords `DO` or `THEN` in test and iterated statements (see the productions for the non-terminals `testStmt` and `iteratedStmt`) may be omitted if they are followed by another keyword.
- Semicolons are used to separate successive declarations and statements (see the productions for the non-terminals `program` and `blockStmt`), but semicolons that occur at the end of a line may be omitted.
- The operators `+`, `-`, `!` and `%` may be used as unary **or** binary operations, but if they occur as the first non-whitespace character on a line they **must** be interpreted as unary operators.

The good news is that you don't need to worry about implementing these lexical rules, since they have already been implemented for you in the `LexicalAnalysis.scala` module, which provides:

1. Parsers for each of the weBCPL keywords (lines 344 - 395),
2. Parsers for each of the weBCPL operator symbols (lines 210 - 275),
3. Parsers for literal string (`stringConst`, line 162), character (`charConst`, line 166) and integer (`integerConst`, line 185) constants, and
4. A parser for identifiers (`identifier`, line 197) used for variable and label names.

When writing your weBCPL parser you should always use the parsers provided in the `LexicalAnalysis.scala` module to parse weBCPL keywords and operators. They will handle all of the fiddly extra lexical details discussed above, and allow you to concentrate solely on implementing the grammar given above.

Please note: the `LexicalAnalysis.scala` module provides two separate sets of parsers for parsing those operators that can be used unary or binary position:

- `unaryMinus`, `unaryPlus`, `unaryPling` and `unaryPercent` for the unary variants of `-`, `+`, `!` and `%` (respectively), and
- `minus`, `plus`, `pling` and `percent` for the binary variants of `-`, `+`, `!` and `%` (respectively).

When writing your parsers you should be careful to use the correct versions of these parsers when implementing your parsers for levels 8, 9, 11, and 12 in the precedence table above.

Dominic Verity
Last modified: 17 September 2019