Tom Gause
Project 3

junk

December 18, 2020
Programming Languages
Prof. Dufour

Grammar

I borrowed *extremely* heavily from the parser grammar for my new grammar. Little is changed aside from the addition of functions, for loops, a print statement, and the subsequent required adjustments. At least one space is required after all words and \t and \n characters may be used as desired.

<program> ::= program <progname> <compound stmt> endprogram
<compound stmt> ::= (<stmt> {; <stmt> })
<stmt> ::= <simple stmt> | <structured stmt>
<simple stmt> ::= <assignment stmt> | <read stmt> | <write stmt> | <function stmt> | <print stmt>
<print stmt> ::= print (<expression>)
<assignment stmt> ::= <variable> := <expression>
<read stmt> ::= read (<variable> { , <variable> })
<write stmt> ::= write (<expression> { , <expression> })
<function> ::= def <funcname> (<variable> {, <variable> }) <compound function stmt>
<compound function stmt> ::= (<stmt> {; <stmt>} <return> [<expression>])
<structured stmt> ::= <compound stmt> | <if stmt> | <while stmt> | <for stmt>
<if stmt> ::= if <expression> then <stmt> | if <expression> then <stmt> else <stmt>
<while stmt> ::= while <expression> do <stmt>
<for stmt> ::= for (<expression> , <expression>) do <stmt>
<expression> ::= <simple expr> | <simple expr> <relational_operator> <simple expr>
<simple expr> ::= [<sign>] <term> { <adding_operator> <term> }
<term> ::= <factor> { <multiplying_operator> <factor> }
<factor> ::= <variable> | <constant> | <string> | (<expression>) | <function call>
<function call> ::= <funcname> (<factor> {, <factor> })

<sign> ::= + | -
<adding_operator> ::= + | -
<multiplying_operator> ::= * | / | %
<relational_operator> ::= = | <> | < | <= | >= | >
<variable> ::= <letter> { <letter> | <digit> }
<constant> ::= <digit> { <digit> }
<string> ::= # <letter> { <letter> }
<progname> ::= <capital_letter> { <letter> | <digit> }
<funcname> ::= <letter> { <letter> }
<capital_letter> ::= A | B | C | ... | Z
<letter> ::= a | b | c | ... | z | <capital_letter>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The next page contains left and right derivations for these instructions:

program Loop (while x > 0 do x := x + 1) endprogram

(The font is extremely small to reduce each statement to a single line.)

Left Derivation:

<program>
program <progname> <compound stmt> endprogram
program Loop <compound stmt> endprogram
program Loop (<stmt>) endprogram
program Loop (<structured stmt>) endprogram
program Loop (<while stmt>) endprogram
program Loop (while <expression> do <stmt>) endprogram
program Loop (while <simple expr> <relational_operator> <simple expr> do <stmt>) endprogram
program Loop (while <term> <relational_operator> <simple expr> do <stmt>) endprogram
program Loop (while <factor> <relational_operator> <simple expr> do <stmt>) endprogram
program Loop (while <variable> <relational_operator> <simple expr> do <stmt>) endprogram
program Loop (while x <relational_operator> <simple expr> do <stmt>) endprogram
program Loop (while x > <simple expr> do <stmt>) endprogram
program Loop (while x > <term> do <stmt>) endprogram
program Loop (while x > <factor> do <stmt>) endprogram
program Loop (while x > <constant> do <stmt>) endprogram
program Loop (while x > 0 do <simple stmt>) endprogram
program Loop (while x > 0 do <assignment stmt>) endprogram
program Loop (while x > 0 do <variable> := <expression>) endprogram
program Loop (while x > 0 do x := <expression>) endprogram
program Loop (while x > 0 do x := <simple expr>) endprogram
program Loop (while x > 0 do x := <term> <adding_operator> <term>) endprogram
program Loop (while x > 0 do x := <factor> <adding_operator> <term>) endprogram
program Loop (while x > 0 do x := <variable> <adding_operator> <term>) endprogram
program Loop (while x > 0 do x := x <adding_operator> <term>) endprogram
program Loop (while x > 0 do x := x + <term>) endprogram
program Loop (while x > 0 do x := x + <factor>) endprogram
program Loop (while x > 0 do x := x + <constant>) endprogram
program Loop (while x > 0 do x := x + 1) endprogram

Right Derivation:

<program>
program <progname> <compound stmt> endprogram
program Loop <compound stmt> endprogram
program Loop (<stmt>) endprogram
program Loop (<structured stmt>) endprogram
program Loop (<while stmt>) endprogram
program Loop (while <expression> do <stmt>) endprogram
program <progname> (while <simple expr> <relational_operator> <simple expr> do <stmt>) endprogram
program <progname> (while <simple expr> <relational_operator> <simple expr> do <simple stmt>) endprogram
program <progname> (while <simple expr> <relational_operator> <simple expr> do <assignment stmt>) endprogram
program <progname> (while <simple expr> <relational_operator> <simple expr> do <variable> := <expression>) endprogram
program <progname> (while <simple expr> <relational_operator> <simple expr> do <variable> := <simple expr>) endprogram
program <progname> (while <simple expr> <relational_operator> <simple expr> do <variable> := <term> <adding operator> <term>) endprogram
program <progname> (while <simple expr> <relational_operator> <simple expr> do <variable> := <term> <adding operator> <factor>) endprogram
program <progname> (while <simple expr> <relational_operator> <simple expr> do <variable> := <term> <adding operator> <constant>) endprogram
program <progname> (while <simple expr> <relational_operator> <simple expr> do <variable> := <term> <adding operator> 1) endprogram
program <progname> (while <simple expr> <relational_operator> <simple expr> do <variable> := <term> + 1) endprogram
program <progname> (while <simple expr> <relational_operator> <simple expr> do <variable> := <factor> + 1) endprogram
program <progname> (while <simple expr> <relational_operator> <simple expr> do <variable> := <variable> + 1) endprogram
program <progname> (while <simple expr> <relational_operator> <simple expr> do <variable> := x + 1) endprogram
program <progname> (while <simple expr> <relational_operator> <simple expr> do x := x + 1) endprogram
program <progname> (while <simple expr> <relational_operator> <term> do x := x + 1) endprogram
program <progname> (while <simple expr> <relational_operator> <factor> do x := x + 1) endprogram
program <progname> (while <simple expr> <relational_operator> <constant> do x := x + 1) endprogram
program <progname> (while <simple expr> <relational_operator> 0 do x := x + 1) endprogram
program <progname> (while <simple expr> > 0 do x := x + 1) endprogram
program <progname> (while <term> > 0 do x := x + 1) endprogram
program <progname> (while <factor> > 0 do x := x + 1) endprogram
program <progname> (while <variable> > 0 do x := x + 1) endprogram
program <progname> (while x > 0 do x := x + 1) endprogram
program Loop (while x > 0 do x := x + 1) endprogram

Parser

See *parser.py*.

Run program with user input:
$ python3 parser.py

As Program.txt is moderately long, you will need a large terminal to view the visualized output.

Code

See *program.txt*.

Run program with *program.txt*:
$ python3 parser.py Program.txt

I included both functions and function calls in a single program.

Binding

Junk is a purely interpreted language.

Variables are stack-dynamic, i.e. bound to addresses on the stack during runtime. When any subroutine (function) is executed, memory cells for are allocated for variables, then deallocated when execution ends. At any point during runtime, the stack will contain memory cells for all currently executing subroutines. There is no heap memory space allocated for variables as there are no global variables in Junk.

To help avoid stack overflow, literals are stored on the heap with a hash table of pointers allocated on the stack. As speed is not of the essence and this language is purely educational, these pointers will be stored dynamically during runtime.

Junk has a memory manager that internally manages its private heap. During runtime, if a function is defined, Junk will build a function object, assign a name to it, and store it in the heap as well as storing a reference to this object on the stack. If the function is called, this reference then directs the interpreter to the definition on the heap. Note that due to this implementation, we must actually define functions before calling them. Furthermore, function parameters are stored on the stack, not the heap to help keep track of recursion. Since the references to these functions are stored on the stack, the definitions must be recreated each time the program is executed. Hypothetically, the interpreter would clean this memory up to avoid leakage.

Arithmetic operations are built into the interpreter (and the CPU itself) – as the interpreter translates each statement into a sequence of subroutines and then to machine code, it recognizes arithmetic operations and knows how to directly map these to instructions interpretable by the CPU. The instructions for this process are directly in the source code for the language and cannot be altered.