IIT CS443: Compiler Construction

Project 1: MiniIITRAN Parser

Prof. Stefan Muller

Out: Tuesday, Sep. 10
Due: Tuesday, Sep. 17, 11:59pm CDT

Logistics

Submission Instructions

Please read and follow these instructions carefully.

- Download the starter code by cloning the Github repo for the assignment. When you first go to the link, you'll be asked to create/join a team (if you're working by yourself, just create a team with just you). Github will create one repo per team.
- Submit your homework by pushing your changes to Github by the deadline (or the extended deadline if taking late days). You can, of course, push non-finished code to your repo (e.g., while collaborating). I'll grade the last commit before the deadline. If you push to the repo one or two days after the deadline, I'll consider that a submission using late days and grade the last submission from the last day.
- If you accidentally push to your repo after the deadline and didn't intend to take late days, email me ASAP. Otherwise, you do not need to let me know if you're using late days; I'll count them based on the date of your last submission.
- Compile (by running `make`) before submitting. **Submissions that don't compile will not get credit.**

Collaboration and Academic Honesty

You may work in groups of at most 2 on this project. Read the policy on the website and be sure you understand it.

1 MiniIITRAN Language Specification

In this project, you'll continue to work with the MiniIITRAN language (and AST definition in `iit_ast.ml`) from Project 0. The language specification is repeated here.

1.1 Syntax

	digit	::=	$0-9$
	alpha	::=	$a-z, A-Z$
	alphanum	::=	*alpha* | *digit* | _
Identifiers	*id*	::=	*alpha* *alphanum**
Ident.Lists	*idlist*	::=	*id* | *id*, *idlist*
Numbers	*num*	::=	$-?$ *digit*+
Types	*typ*	::=	INTEGER | CHARACTER | LOGICAL
Constants	*c*	::=	*alpha* | *num*
Binary Operators	*bop*	::=	$+ \mid - \mid * \mid /$ | AND | OR $\mid < \mid \leq \mid > \mid \geq \mid \# \mid =$
Unary Operators	*unop*	::=	$\sim$| NOT | CHAR | LG | INT
Expressions	*e*	::=	*c* | *id* | *e bop e* | *e* $<-$ *e* | *unop e*
Statements	*s*	::=	*e* | STOP | DO s^* END | IF *e s* | IF *e s* ELSE *s* | WHILE *e s*
Declaration	*d*	::=	*typ* *idlist*
Program	*p*	::=	d^* s^*

Operator Precedence and Associativity.

Operator(s)	Precedence	Associativity
All unary operators	6	—
*, /	5	Left
+, -	4	Left
Comparison operators	3	Left
AND	2	Left
OR	1	Left
$<-$	0	Right

(Higher numbers indicate that operators have higher precedence, i.e., "bind tighter"). All operators are left-associative except assignment. So, $x <- y <- 2+5$ should parse as $x <- (y <- (2+5))$.

Comments. Comments start with `$` and go to the end of the line.

1.2 Semantics

Constants Numeric constants are specified as integers, possibly preceded by a $-$ sign. Character constants are specified as `'c'`. There is no direct way to specify logical constants, but logical constants can be introduced using LG n, which becomes logical true if $n \geq 0$ and logical false if $n < 0$.

Variables Variables are declared with declarations, which declare one or more variables with a given type. A variable may not be used without a declaration (this is a change from real IITRAN, but makes compilation easier). Later declarations of variables take precedence over older declarations (the older declarations become useless, as declarations must precede all statements.) Variable names are case-insensitive. Variables are initialized to 0 (for logical variables, this means false, and for character variables, it means ASCII 0).

Binary Operations e_1 *bop* e_2
Operations are evaluated left-to-right (with short-circuiting as described below under "logical operators"): e_1 is evaluated before e_2. This mostly matters when expressions contain assignments. For example, if `B` is initially 0, then `(B <- 1) + (B <- B + 1)` should evaluate to 5.

Individual categories of binary operators are described below.

Arithmetic Operators (+, -, *, /) e_1 *bop* e_2
Types: e_1 : INTEGER, e_2 : INTEGER, result: INTEGER
Note: Integer overflows and division by zero result in runtime errors.

Comparison Operators $(<, \leq, >, \geq, \#, =)$ $e_1\ \mathit{bop}\ e_2$
Types: e_1 : INTEGER, e_2 : INTEGER, result: LOGICAL
Note: # is "not equal to."

Logical Operators (AND, OR) $e_1\ \mathit{bop}\ e_2$
Types: e_1 : LOGICAL, e_2 : LOGICAL, result: LOGICAL
Important Note: Both AND and OR should *short circuit*, that is: if e_1 evaluates to false, e_2 should not be evaluated at all in e_1 AND e_2 (and similar for e_1 OR e_2 if e_1 evaluates to true). In MiniIITRAN, this is mainly relevant if e_2 might divide by zero. For example, `1 > 0 OR 1 / 0 > 0` should never raise a divide-by-zero exception.

Assignment $x <- e$
Types: x and e must have the same type. The result is of that same type.
Result: Compute e, assign its value to x and return the value.
Note: $e_1 <- e_2$ where e_1 is some expression other than a variable is a runtime error. For the purposes of this class, it is unspecified whether or not such assignments are syntactically valid (so your parser may accept them or not).

Integer Negation $\sim\ e$
Types: e : INTEGER, result: INTEGER
Result: $0 - e$

Logical Negation NOT e
Types: e : LOGICAL, result: LOGICAL

Type Conversions INT, LG, CHAR.
The result type is as specified (INTEGER, LOGICAL, CHARACTER, respectively). The argument can have any type.
LG n becomes logical true if $n > 0$ and logical false if $n \leq 0$.
LG c, where c is a character is always logical true unless c is ASCII 0.
INT c returns the ASCII code of c.
INT l of a logical constant l returns 0 for false and 1 for true.
CHAR n returns the character with ASCII code n.
CHAR l returns the character with ASCII code 0 or 1.

Expression statements
Types: The expression must be well-typed with any type.
Result: The expression is computed. Any assignments are performed, but otherwise the value is ignored.

If statements IF e s_1 ELSE s_2
Types: e : LOGICAL.
Result: If e evaluates to true, performs s_1, otherwise s_2. If s_2 is absent, control continues to the next statement.

While statements WHILE e s
Types: e : LOGICAL.
Result: If e evaluates to true, performs s and then evaluates e again and loops. When e evaluates to false, control continues to the next statement.

Stop *Result:* Ends execution of the program.

Do DO $s_1 s_2 \ldots s_n$ END
Types: All substatements must be well-typed. *Result:* Substatements are executed in order.

Program results. The program returns the value in the designated variable RESULT when execution ends, either by control reaching the end of the programming or encountering a STOP. By convention, RESULT should be declared to be an integer (programs should return integer values, and your compiler may assume this is the case).

2 Task 1: MiniIITRAN Parser

In the first part of the project, you'll write a Menhir parser for MiniIITRAN. I've given you the lexer, in `lexer.mll`. You may want to take a look at it to get a sense of how the various tokens correspond to the MiniIITRAN syntax. The main thing to note is that the tokens `TINT`, `TCHARACTER`, and `TLOGICAL` correspond to the types INTEGER, CHARACTER, and LOGICAL, while the tokens `CCHAR`, `CINT`, and `CLG` correspond to the unary operators for type conversion. Otherwise, things should be self-explanatory.

Note: The handout code will not quite compile as is, but you should still be able to get at least syntax highlighting without it fully compiling. Other language server stuff isn't as useful for writing parsers anyway.
Task 1 (Programming, 25 points).

Implement the parser for MiniIITRAN using Menhir, in `iit_parser.mly`.

- The file so far has some headers, and `%start` and `%type` annotations. The definitions of the tokens are in `iit_tokens.mly`. Don't modify any of this.
- The parser file also contains dummy definitions for many, **but not all**, of the nonterminals you'll need. You shouldn't need to modify the definitions of the following nonterminals: `expr`, `identlist`, `decl`, `stmt`, `stmtlist`, `stmtlist`, `decllist`, `prog`. You'll need to modify the other definitions and add others as you need them.
- As you might expect, `expr_` produces a `p_expr_` and `expr` converts this into a `p_expr` by supplying the location information. The same for `stmt_` and `stmt`. Remember that AST nodes for `expr_`'s contain `expr`'s, and your rules for `expr_` can refer to the nonterminal `expr` (same for statements). If, for some reason, you need to define other nonterminals that produce `p_stmt_`'s, you can add a nonterminal that produces the corresponding `p_stmt` by mimicking our definition of the `stmt` nonterminal.
- You should also add precedence and associativity annotations in `iit_parser.mly`, right above the `%start` annotation.

You can run Menhir on your parser by running `make`. Conflicts will be reported in `parser.conflicts`. **Your completed parser should have *no* shift/reduce or reduce/reduce conflicts.**

3 Testing

Note that, because it's a solution to another assignment, I **did not** include an IITRAN interpreter in the handout code. But you should have one by now! Overwrite `src/iit_interp.ml` with your solution from Project 0 (or, if you're not confident in the correctness of yours, you can use the posted solution code.) After you do this, run `make` (in the top level directory of the repo, not in `src`) to compile your interpreter. This produces a binary `iit`, just like in Project 0, which you can run on the command line like so:

```
./iit file.iit
```

where `file.iit` is an IITRAN source code file. This parses the source code, type-checks it and runs the interpreter on it.

We've given you a number of test files in `tests`. Each program begins with a comment indicating the expected result (recall that this is the value of variable RESULT), if the program is syntactically correct. Files of the form `tests/synerrN.iit` should not parse and should result in a syntax error. You can run all of the tests by running (on the command line, still in the top level of the repo) `make test`.

4 Task 2: Writing Test Cases

Task 2 (Test Cases, 5 points).

Each group should write (at least) one IITRAN program that is not substantially the same as the test cases I've given you or others written by other students/groups (this is good incentive to write your tests early and post them before other groups!) You're encouraged to try and find corner cases and explore code paths other tests might have missed. Don't be too adversarial though; reasonable student solutions should pass your test (e.g., please don't try to cause a stack overflow in someone's parser by submitting a 500MB source file). Your test case *can* be designed to trigger a syntax error (but make it interesting, don't just submit a file of gibberish).

Post your test case as a new thread on the "Project 1 Test Cases" discussion board on Canvas. Make sure to include the expected (integer) result somewhere, in a comment at the top of the test case and/or in your post on the discussion board. You can (and should!) test your parser on other students' test cases. I may do so as well during grading. Feel free to ask clarification questions, note issues, etc., as replies in the threads created by other students.