SIMPL Project Specification

Computer Science 244

Stellenbosch University, 2021

Last updated on Thursday 23rd September, 2021 at 16:28

Contents

CHAPTER 1

Introduction

1.1 SCOPE

This document describes Stellenbosch Imperative Mini Programming Language 2021 (SIMPL), an LL(1) language[*] used during the 2021 session of Computer Science 244 at Stellenbosch University to introduce the C language, compiler design and the Java virtual machine (JVM) architecture. In this document, the keywords "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**MAY**", and "**OPTIONAL**" are to be interpreted as described in RFC 2119 [2]. When such keywords constitute a binding part of the specification, they will be set in maroon, boldface small capitals.

1.2 PLAGIARISM DECLARATION

When you sign you submission form, you stipulate to the following plagiarism declaration:

> By signing this document, I agree to the following.
>
> 1. I have read and followed the instructions given in the specification for this assignment.
> 2. I have read and understand the Stellenbosch University Policy on Plagiarism and the definitions of "plagiarism" and "self-plagiarism" contained in the Policy, and I agree that plagiarism is a punishable offence because it constitutes theft.
> 3. I understand that I may discuss the work with fellow students and/or staff (including demis, tutors, and lecturers), but that (a) I *may not* procure (ask for and/or receive, buy, or obtain by any other means) solutions to this assignment (in its entirety or in part) from anyone, and that (b) I *may*

[*]Refer to §A.3 for brief comments on different classes of grammars.

not copy-and-paste code from anywhere unless explicitly allowed in the specification for this assignment (and then, only with proper attribution).

4. Accordingly, my submission acknowledges all external libraries and sources used, and identifies any other students, staff (including demis, tutors, and lecturers), and/or any other party with whom I have discussed this assignment.
5. (a) I have not allowed, and will not in the future allow, anyone to copy any portion of my work, or give them access to it in any way. (b) I have not and will not make my work publicly available in any way, including posting my code in public source code repositories or forums. (c) I have not and will not facilitate plagiarism, such as by distributing any written work or source code created by myself or a fellow student.
6. I understand that any work I submit may be inspected for plagiarism (manually and/or by automated systems) and be retained for detecting plagiarism in other modules.
7. I declare that the work submitted for this assignment is my own original work, except for the inclusion of resources explicitly permitted for this assignment, and for assistance as noted in items 3 and 4 above.

CHAPTER 2

Rules and guidelines

2.1 PLATFORM

This project **MUST** be completed on a Unix system. I have made every effort to ensure that the project files compile on both Linux and macOS.[*] However, in the event of any discrepancies, the setup in the NARGA laboratories will be the final arbiter: Your submissions will be evaluated there. Therefore, if you work on your own computer instead of using SSH to connect to NARGA, you **MUST** ensure your project also works in NARGA. ***No excuses will be accepted if your work does not compile or compiles with warnings because your own setup differs from that in the lab, even if it is only in a subtle way.***

2.2 CODE FORMATTING

Your source code **MUST** be formatting according to Kernighan and Ritchie's book *The C Programming Language* [11]. For concrete examples, refer to the style of the skeleton files. In more detail,

1. with the exception of function bodies, an opening brace **MUST** appear on the same line as the statement for which it opens a block;
2. for function bodies, the opening brace **MUST** go on a new line, flush with the left margin on the line below the function signature;
3. if a closing brace is followed by the keyword `else`, both **MUST** appear on the same line, and if `else` is followed by an opening brace (including when an `if` guard appears between `else` and the opening brace), both **MUST** appear on the same line;
4. `case` statements **MUST** either be aligned to the same column as its `switch` keyword, or **MUST** have one extra level of indentation with respect to the `switch` keyword, but whichever your choose, you must do it *consistently*;

[*]WSL does not constitute a Unix system, and is not supported.

5. there **MUST NOT** be any spaces between a function name and the parentheses that delimit the list of **formal parameters** (also known as **parameters**) or **actual parameters** (also known as **arguments**); but

6. there **MUST** be one space between the keywords `if`, `for`, and `while`, and the parentheses that opens their guard conditions;

7. although `sizeof` is an operator, its use **MUST** be treated as that of a function call;

8. unless specifically overridden here, use the spacing rules for normal written work throughout, for example, there **MUST** be one space after each comma or semicolon if what follows is on the same line, including function definitions and calls;

9. there **MUST** be one space between the keyword `switch` and the parenthesised integral expression that follows it;

10. if an `if`, `for`, or `while` statement has a compound statement as body—that is, has statements between braces as body—the parenthesis that closes the guard condition and the brace that opens the block **MUST** be separated by one space;

11. the parenthesised integral expression that follows the `switch` statement **MUST** be separated by one space from the brace that opens the block of `case` statements;

12. there **MUST NOT** be any spaces between an opening parenthesis and the expression or parameter that follows it, and there **MUST NOT** be any spaces between a closing parenthesis and the expression or parameter that precedes it;

13. an additive operator **MUST** be separated from its operands by one space on each side (of the operator), except when the operator appears as part of an array index, in which case there **SHOULD** be no spaces;

14. a multiplicative operator **MUST** either be separated from its operands by one space on side (of the operator), or no spaces at all;

15. all preprocessor directives **MUST** be aligned flush with the left margin;

16. lines in your source code file **MUST NOT** end in one or more space characters, and in particular, a blank line **MUST NOT** consist of any characters except one EOL character;

17. all function definitions **MUST** be separated from other programming constructs by at least one blank line;

18. indentation **MUST** be made with tab characters, but aligning variables, members of `struct`s and `union`s, and comments after statements **SHOULD** be with space characters, but such alignment is **OPTIONAL**; and

19. the number of characters on any line **MUST NOT** exceed 80; for this purpose, the tab width is assumed to be four.

Also, I strongly **RECOMMEND** that you use the so-called "One True Brace" style. That is, even if there is only one statement in a block, do not leave out the braces—using braces makes it easier to insert statements for debugging.

Style will be marked aggressively. In particular, the insanity of writing loop and branching constructions as if they were functions must be stopped. ***If, for example, you write***

```
for( i = 0; i < n; i++ ) {
    /* do something */
}
```

instead of

```
for (i = 0; i < n; i++) {
    /* do something */
}
```

you will get a style mark of zero. (If you cannot see the difference, refer to items 5 and 6 above.)

2.2.1 Vim setup

As a minimum, you should have the following in your `~/.vimrc` file:

```
set textwidth=80  " maximum of 80 characters per line
set tabstop=4     " 1 tab = 4 spaces (visually)
set shiftwidth=4  " for when you use << and >> in Vim
set autoindent    " on newline, keep current indentation
```

The settings in the `vimrc` file apply to both command-line Vim and GVim, as well as NeoVim. GVim-specific settings, for example, the `guifont` setting, must go into the `~/.gvimrc` file. If you want to pimp your Vim, consider using a plugin manager such as vim-plug [21] or Dein.vim [5]; if you really want to delve into all things Vimscript, consider Steve Losh's book *Learn Vimscript the Hard Way* [13].

Note that the provided makefile in your repository contains advice on how to add syntax highlighting for SIMPL to Vim. The makefile also has a description of how to add command-line completion for SIMPL to the BASH shell.

2.2.2 Comments

Your code **MUST** be commented with Doxygen; use the skeleton code as guideline. As funny as this may sound, do not "overcomment" your code. Unlike in previous courses, I assume that you have at least a modicum of programming competency, and therefore, it is not necessary to belabour the obvious. You should not comment every single statement—your code should be clearly written in the first place.

Rather, provide comments for the main ideas in a function or the tasks it performs. If you find yourself explaining difficult or involved code, ask yourself whether you cannot rewrite the code in a simpler way. The comments **MUST NOT** under any circumstances contradict the code—it is not only confusing and can create havoc during debugging, but is also disrespectful towards those who read your code.*

*One day, they may include your managers and other damaged people, and they will likely have a say in your salary.

Consider the following code snippet:

```
/* loops over all command-line arguments */
for (i = 1; i < argc; i++) {
    /* displays the length of the command-line argument i */
    printf("strlen(argv[%d]) = %d\n", i, strlen(argv[i]));
}
```

These comments just clutter the source code. The intent of the programmer here is clear, without having to comment.

2.2.3 Instructions in Comments

To provide context, comments may start with a standard, uppercase tag. Out of the box, the C syntax highlighter for Vim recognises the tags `TODO`, `FIXME`, and `XXX`. The first two are self-explanatory, and `XXX` is a general "catch all" used before problematic code, for example, if something magically works, all appearances to the contrary, and you therefore expect the code to be buggy.

Although this document constitutes the complete specification for SIMPL, I use `TODO` notes in the skeleton to give you additional instructions for using the skeletons. Once you are done with a particular `TODO` context, you **MUST** remove the note; the only source file where `TODO` notes can remain until the final submission is `simplc.c`, because you will keep adding to this file for every submission except the first.

In a perfect world, you will not have any `FIXME` or `XXX` comments by the time of submission . . . but this is idle hope, I suppose. You **MAY** leave `FIXME` or `XXX` comments in your code: You will not be penalised for doing so, because it does show some awareness of bugs—in some cases, even forethought—and I would much rather have you know where to start looking if something goes belly up. However, `FIXME` is an instruction to yourself, and to yourself alone. Neither the demis nor I will fix anything to make it run or return the correct results. The bottom line is: If you know there is a bug, `UFIXITURSELF`.[*]

2.2.4 Identifiers

Identifiers **MUST** be all lowercase, and "words" in an identifier **SHOULD** be separated by underscores. *You **MUST NOT** use camelcase*, which you are used to seeing in Java. For example, write `number_of_elements` and `get_string` instead of, respectively, `numberOfElements` and `getString`. There are two exceptions: (1) When an identifier is an enumeration constant, or a symbolic constant or macro defined by `#define`, it **MUST** be written all in uppercase, with "words" separated by underscores. (2) The only exceptions to the camelcase rule are type definitions, but they are provided to you in the source code skeletons; once again, this follows the example of K&R [11].

[*] Yes, yes, professional programmers go about leaving the note "HERE BE DRAGONS" for especially scary code, even in code that is already in production—but you are not a professional programmer yet.

In general, you **SHOULD** use concise names for local variables and long, descriptive names for global variables; remember all global variables in C occupy the same namespace, so each had better be unique! There are exceptions to this dictum, and as always, common sense prevails in common practice. For example, in a compiler, the current token is an important global variable—we could call it `current_token`, but since it appears frequently, it seems sensible simply to call it `token`, and then to use other variable names for local token variables.

For local variables, keep the following mathematical metaphors in mind: (1) use `i`, `j`, and `k` for counters; (2) use `m` and `n` for the total numbers of objects; (3) use `c` or `ch` for a character on which you are operating; (4) use `s` and `t` for local strings or string pointers; (5) use `u`, `v`, and `w` for arbitrary pointers into linked structures; and (6) use `p` and `q` for general pointers. Be careful of using a single lowercase *l* as identifier, because depending on the typeface you use, it may easily be confused with the digit for one, especially in monospaced fonts; for example, compare `l` to `1`.[*]

Function and variable names **MUST** be written in English. This is not to enter into any language debate, and it is nothing against any other South African language (or French and German, for that matter). It is simply that English is the lingua franca of programming, and in particular, that C uses English keywords. The following, with its mixture of languages, simply does not look right—and worse, will be very confusing to international readers.

```
for (teller = 0; teller < aantal; teller++) {
    hayibo(teller);
}
```

Finally, you **SHOULD** use descriptive verb phrases as your function names, which is to say, do not use simple nouns or the third person declarative, but instead, use the second person imperative. For example, for a function that reads a character from somewhere, do not use `character` or `gets_character`; rather use `get_character`.

In C, as a rule of thumb, think of variables as nouns and functions as verbs.[†] In one sense, functions are the user-defined verbs of a programming language. So, naming them using verbs makes it clear what these functions do. You may abbreviate words, but the decision of whether to abbreviate or not is somewhat of an art that comes with experience.

[*]Indeed, many affordable typewriters from bygone years, especially portable ones, had only one key (and hence, glyph) for both one and ell. One should never frown at "ancient" technology: In these latter, postmodern days, it often helps us to think in terms of essentials, weighing every so-called "advance" critically . . . rather like the C compiler, where you have to consider your use of dynamic memory in its full implications. Knuth, for example, is famous for writing out *The Art of Computer Programming* in longhand before typesetting it in LaTeX. Writing in longhand is a practice I heartily recommend for study, research, and just getting your thoughts in order.

[†]I cannot stress enough that programming is an art, in both the sense of "creative and expressive endeavour" and the sense of "craft", the latter word referring to the "skill at doing a specified thing". Good programmers know their math or programming metaphors; great programmers find grace in mastering both the artisanal and the artistic, achieving the fluency of thought and the elegance of expression for which great authors strive. To mend an aphorism from Einstein: Math without programming is lame; programming without math is blind. For me, both programming and math are narrative arts.

2.3 REPOSITORIES

Your personal project repository is available at

`rw244-2021@gitolite.cs.sun.ac.za:`⟨*SU number*⟩`/simpl`

where you substitute ⟨*SU number*⟩ by your own student number. Only you have read–write access; the demis have read-only access, so they can neither damage nor fix your repository (without your being present). Of course, I have root, but after creating your repositories, I back off. Our Git server is open to the Internet[*] at large; therefore you do not need VPN or jump host access to reach your repository from outside the university network.

2.3.1 Branching

Since your repository already contains some files, clone it before starting to work. Cloning also sets up our Git server to be the remote repository and tracks the master branch automatically. If this sounds like Greek, did you read Chapter 2 of *Pro Git* [4], available on SUNLearn?

You should consider using Git branches to organise your workflow, but it is **OPTIONAL**. Driessen's GitFlow [6] is probably the most popular and well-known branching strategy out there, but I find it overly complex—especially for smaller projects—as indeed do others, to the point of being harmful [16]. GitHub Flow [3] and GitLab Flow [18] are alternatives to consider. Just remember that, whether you use branching or not, we check out tags when marking, and we will not go searching for your work in the branches.

2.3.2 Public repositories

In the past, I have had requests from students who wanted to publish their work in a publicly accessible repository as part of a programming portfolio. Unfortunately, this is not possible. Doing so is in contravention of the Stellenbosch Computer Science plagiarism policy, and since I provide you with a lot of code, it is also subject to copyright by the university.[†]

2.3.3 Keeping your repository in order

Keeping your repository in working order is your responsibility: If you break it, you fix it. Use your test repository if you need a live setup on which to test the effects of Git commands; if this repository breaks, there will be no repercussions.

Your repository is not a general dumping place for miscellaneous files. In particular, *you **MUST NOT** track large files* like PDFs and audiovisual containers. I have set up a `.gitignore` file in your project repository's root directory, so if certain file types will not track for commits, that is where you look. You **MAY** create and track directories other than those initially

[*] Yes, I still write "Internet" with a capital letter. What we know as the Internet is just an internet, which means that, technically speaking, if all the three- and four-letter organisations hack it, or if some x (insert government name here) royally destroys it, we can just build another one.

[†] If you feel any better, all the marvellous things I do also belong to the university by default.

included, in particular if you want to include your test suite. You **MUST NOT**, however, add subdirectories under your `src` directory.

You **MUST** set up a a global Git ignore file, that is, one that affects all Git repositories in your account. GitHub has a nice collection of ignore files [7]; refer to the `Global` directory, and also read the article *Ignoring files* [8]. For the lab setup (and also on your own computer, if you use Linux), you **MUST** add the ignore files for `Linux`, `Tags`, and `Vim`, and you **SHOULD** add the ignore files for `Archives`, `Backup`, `Diff`, `GPG`, `Images`, and `Patch`. If you work on macOS, replace the `Linux` ignore file by the `macOS` ignore file.

Your global Git ignore file is not something we can check: It remains and is in effect only on a particular machine. However, Vim saves swap files by default, and the NFS mounts used in the lab have the nasty habit of creating "silly renames" [15], with names like `.nfs0000000004aa492d00000287`. Since these file names start with a dot, they are hidden in normal listings, but we can see them quite clearly when we pull your work. (And, of course, when you commit and also if you do a directory listing with `ls -a`, so can you.) The global ignore files for Linux and Vim as given on GitHub protect against these kinds of files being committed by accident. ***If we find such "temporary" files in your repository, you will be penalised heavily in terms of your marks for style.*** If you are unsure about whether your repository state is acceptable, ask a demi for help.

2.3.4 Keeping executable files out of your repository

Remember, a software repository is for tracking source, not binaries or other artefacts of compilation—programmers who clone your repositories must be able recreate executable code from the files you committed. Therefore, your repository **MUST NOT** contain any binary executable files, except that you **MAY** include shell or Python scripts, for example, for running tests. If you commit Python modules, you **MUST NOT** track Python bytecode (`.pyc`) files or `__pycache__` directories; in this case, you **SHOULD** update your repository `.gitignore` to include the relevant files for Python in GitHub's ignore file collection.

The makefile in your repository—see §2.6—places executables in the repository's `bin` directory, which is set up to be ignored. But be particularly careful not to commit executables you may put elsewhere, for example, during testing.

You **MUST NOT** set execute permission for any files in your repository, except shell scripts or command-line executable Python scripts with an appropriate shebang [24]. *Be very careful when copying files from flash drives.* Most flash drives are formatted to some kind of FAT file system, and when Linux mounts FAT, it may add execute permission to files by default. Your source files **SHOULD** have either 644 or 600 as permissions; for details, especially on the octal permission specification, refer to the manual page for `chmod`.

The bottom line: Make sure you do not commit executable object files, and make sure that only the allowed script files have execute permission set. ***If you do not follow the rules on execute permission, you will be penalised heavily in terms of your marks for style.*** Again, if you are unsure about whether your repository state is acceptable, ask a demi for help.

TABLE 2.1: The submission details of the project; for an explanation of the scanner deadline and weight, see §2.4.1

DESCRIPTION	DUE DATE & TIME	TAG	WEIGHT*
Scanner	3 September 2021, 14:00	`scanner`	—
Scanner & parser	10 September 2021, 14:00	`parser`	10%
Symbol table and type checking	1 October 2021, 14:00	`symtable`	10%
Code generation	8 October 2021, 14:00	`codegen`	10%

*Given with respect to the final mark of the module.

2.4 SUBMISSION

The project's submission details are given in TABLE 2.1. Note that you (1) **MUST** complete a separate plagiarism declaration for *each* submission, which (2) **MUST** be accompanied by a submission report in the prescribed format. If either or both of these are incomplete or missing, your marks will not be posted until you complete both.

The plagiarism declaration, skeletons for the submission reports, and a script to help you attach your signature to the plagiarism declaration are included in the `sub` directory of your repository; please follow the instructions in the `sub/README.txt` file. The script requires LaTeX to be installed, as is the case in NARGA, and the file `sub/personal.txt` has been populated with your personal details from the class spreadsheet. ***Any attempt to tamper with the text of the plagiarism declaration will result in an immediate referral to the Central Disciplinary Committee.***

2.4.1 The scanner submission

The first submission, for the scanner, is a formative assessment, which is to say, you will get feedback, but no marks—informally, it is a "dry run" for the first true submission (scanner *and* parser) the following week. Therefore, you may (and should) use the tutorial of 3 September 2021 to check with a demi whether or not your submission conforms to the specification; marking will not be done until after the tutorial is finished. If you do *not* submit anything for the scanner, ***it has no effect on your marks***, but you do lose an important opportunity for getting feedback.

The feedback you will receive will be the same as for the remaining three submissions, which is to say, the number of test cases passed will be reported, and the report from the style checker will be produced. Note that the test cases for the scanner will only be released when those for the parser are released. So, should you submit the scanner, and your submission does not pass all test cases, you will only know that something is wrong, but you have to determine on your own exactly what is wrong.

2.4.2 Git tagging

To indicate a particular commit is to be your submission for a project part, it has to be tagged in Git; refer to TABLE 2.1 for the tags. ***Remember, tags are not uploaded automatically to a remote repository.*** The easiest way to upload tags is to push with the `--tags` flag.

If you have pushed prematurely, you must delete both the local and remote tag. To delete a local tag, use the `git tag --delete` command. To delete a remote tag, you have to push to the remote repository by executing `git push --delete origin` ⟨*tag*⟩, where ⟨*tag*⟩ is the name of your tag.

This means:

1. ***If you do not tag correctly, your work will not be marked.*** In particular, note that *tags are case-sensitive*.

2. If you do not commit regularly during the week, you may be penalised. ***When we evaluate your work, and you committed nothing during the week,*** but then five minutes before the deadline, a fully functional implementation appears in your repository, ***your submission will be subject to particular scrutiny.*** This does not suggest programming acumen or complete and utter intellectual brilliance, but rather, that you do not know how to use the prescribed tools, or worse, that the implementation is not yours at all.

3. If you "break" your submission repository, even right before a deadline, you have to fix it yourself—it is neither my nor the demis' responsibility to fix anything. Any such breakage will be evaluated on merit; in general, being scientists, any interesting corner cases will be deserving of our attention.

2.5 MARKING SCHEME

For each submission, the marking scheme consists of the following:

- 75% for test cases and functionality, and

- 25% for style, which includes code formatting and repository state.

Any caps will applied to the total mark.

2.6 COMPILING YOUR PROJECT

2.6.1 The makefile

Your repository contains a makefile, which you **MUST** use to build your project. It contains a rule for each project part, and sets up the appropriate compiler flags. We use the makefile for testing, and ***if your work does not compile successfully via this exact makefile, you get zero.*** You should read Chapter 2 of the GNU Make manual [9]—it is a quick and easy read.

2.6.2 Compiler errors and warnings, and programs crashing

Your code **MUST NOT** produce any warnings during compilation. If any warning remains, you forfeit the style marks, so in effect, your mark is capped at 75%. Of course, ***if your code contains errors, and does not compile to an executable at all, you get zero***: Neither the demis nor I will edit or correct your code to make it compile, even if only a semicolon is missing.

If your program crashes during testing, your mark is capped at 50% for that part. You can reasonably expect crashes to occur wherever arrays and pointers are involved. So, you should check those again and again and again. . . .

The bottom line is: Program defensively. Rather add explicit checks than just assume certain conditions hold. You will never be penalised if your code is somewhat paranoid about guarding against memory corruption or boundary errors. Also, learn to use C's assertion facility, available via the `assert.h` header. It works similarly to Java's `assert` statement, and can also be compiled away, without changing source code, after you have finished debugging.

2.7 DEADLINES AND EXTENSIONS

The submission dates are set in stone. The schedule simply does not allow for any changes or extensions. Crosscheck your "busy times" with other subjects as soon as possible, so that you can plan appropriately. You have to plan as if there will be a power failure[*] or network failure at any time. This is also why I strongly urge you to use to use tmux or GNU Screen—if you lose your SSH connection to the lab for any reason, you will be able to reconnect to your session once power or connectivity is restored.

Since test cases are released immediately following a submission, extensions in general are not possible, even if you organise it in advance. (If you know in advance, you can plan accordingly.) ***If you miss a submission, you forfeit the marks for that submission, and it is treated as a "missed assessment", which can lead to your receiving an INCOMPLETE.***

I acknowledge that there can be emergencies, but the same contraints that apply to, say, a test that must be written in a very specific time interval do not apply to an assignment with a deadline. In any event, any emergency will handled strictly according to merit. If you are granted an extension due to an emergency, but there is the reasonable belief that you would have had the advantage of access to the test cases, a penalty may be applied to your marks.

[*] Sure, they can be called "controlled power cuts", but "load shedding" is an insiduous term to smooth over the serious consequences of Eskom's incompetence: It sounds like there is too much of something, which is the exact opposite of what is really happening. In my mind, "load shedding" is right up there with the politically-driven euphemism of saying "climate change" instead of "global warming". Bottom line: Let's call a spade a spade (or, at the very least, a "graaf").

CHAPTER 3

EBNF and scanner

The Extended Backus–Naur Form (EBNF) of the 2021 SIMPL grammar is given in FIGURE 3.1. Some language properties must be handled in the lexical analyser (scanner), but cannot be specified in the grammar; they follow below in §3.1. This section concludes with the errors that may be emitted by the scanner, that is, which are handled by the scanner locally and terminates the compiler.

3.1 LEXICAL LANGUAGE PROPERTIES

3.1.1 The regular part of the grammar

The scanner is actually a Deterministic Finite State Automaton (DFA), which means it can recognise a regular language. These are just the languages we can write without recursion in the grammar productions. Therefore, your scanner **MUST** recognise the productions for ⟨*id*⟩, ⟨*num*⟩, and ⟨*string*⟩; the productions for ⟨*letter*⟩ and ⟨*digit*⟩ (and for parsing, also ⟨*relop*⟩, ⟨*addop*⟩, and ⟨*mulop*⟩) are just there so that, for example, we do not have to write:

⟨*id*⟩ = (“a”|...|“z”|“A”|...|“Z”|“_”) {“a”|...|“z”|“A”|...|“Z”|“_”|“0”|...|“9”}.

This is confusing to read, which is why we rather create extra EBNF productions. Note that ⟨*id*⟩ or ⟨*num*⟩ lexemes **MUST NOT** contain any spaces.

3.1.2 Comments

A SIMPL source code file **MAY** contain comments, which are skipped by the scanner. A comment block **MUST** be started by the character pair “(*” and **MUST** be ended by the character pair “*)”. Comments **MAY** be nested. Comments **SHALL NOT** be allowed inside literals. In particular, when the character pairs “(*” and “*)” appear inside a string literal, they are to be treated as normal characters in the string, and **MUST NOT** be taken the signify the start and end, respectively, of a comment. Although counting arguments are possible, the function for skipping comments **MUST** be written recursively.

⟨*program*⟩ = “program” ⟨*id*⟩ {⟨*funcdef*⟩} ⟨*body*⟩ .

⟨*funcdef*⟩ = “define” ⟨*id*⟩ “(” [⟨*type*⟩ ⟨*id*⟩ {“,” ⟨*type*⟩ ⟨*id*⟩}] “)” [“->” ⟨*type*⟩] ⟨*body*⟩ .

⟨*body*⟩ = “begin” {⟨*vardef*⟩} ⟨*statements*⟩ “end” .

⟨*statements*⟩ = “chill” | ⟨*statement*⟩ {“;” ⟨*statement*⟩} .

⟨*type*⟩ = (“boolean” | “integer”) [“array”] .

⟨*vardef*⟩ = ⟨*type*⟩ ⟨*id*⟩ {“,” ⟨*id*⟩} “;” .

⟨*statement*⟩ = ⟨*exit*⟩ | ⟨*if*⟩ | ⟨*name*⟩ | ⟨*read*⟩ | ⟨*while*⟩ | ⟨*write*⟩ .

⟨*exit*⟩ = “exit” [⟨*expr*⟩] .

⟨*if*⟩ = “if” ⟨*expr*⟩ “then” ⟨*statements*⟩ {“elsif” ⟨*expr*⟩ “then” ⟨*statements*⟩}
[“else” ⟨*statements*⟩] “end” .

⟨*name*⟩ = ⟨*id*⟩ (⟨*arglist*⟩ | [⟨*index*⟩] “<-” (⟨*expr*⟩ | “array” ⟨*simple*⟩)) .

⟨*read*⟩ = “read” ⟨*id*⟩ [⟨*index*⟩] .

⟨*while*⟩ = “while” ⟨*expr*⟩ “do” ⟨*statements*⟩ “end” .

⟨*write*⟩ = “write” (⟨*string*⟩ | ⟨*expr*⟩) {“&” (⟨*string*⟩ | ⟨*expr*⟩)} .

⟨*arglist*⟩ = “(” [⟨*expr*⟩ {“,” ⟨*expr*⟩}] “)” .

⟨*index*⟩ = “[” ⟨*simple*⟩ “]” .

⟨*expr*⟩ = ⟨*simple*⟩ [⟨*relop*⟩ ⟨*simple*⟩] .

⟨*relop*⟩ = “=” | “>=” | “>” | “<=” | “<” | “#” .

⟨*simple*⟩ = [“-”] ⟨*term*⟩ {⟨*addop*⟩ ⟨*term*⟩} .

⟨*addop*⟩ = “-” | “or” | “+” .

⟨*term*⟩ = ⟨*factor*⟩ {⟨*mulop*⟩ ⟨*factor*⟩} .

⟨*mulop*⟩ = “and” | “/” | “*” | “mod” .

⟨*factor*⟩ = ⟨*id*⟩ [⟨*index*⟩ | ⟨*arglist*⟩] | ⟨*num*⟩ | “not” ⟨*factor*⟩ | “true” | “false” | “(”
⟨*expr*⟩ “)” .

⟨*id*⟩ = ⟨*letter*⟩ {⟨*letter*⟩ | ⟨*digit*⟩} .

⟨*num*⟩ = ⟨*digit*⟩ {⟨*digit*⟩} .

⟨*string*⟩ = “"” {⟨*printable ASCII character*⟩} “"” .

⟨*letter*⟩ = “a” | . . . | “z” | “A” | . . . | “Z” | “_” .

⟨*digit*⟩ = “0” | . . . | “9” .

FIGURE 3.1: The EBNF of SIMPL-2021.

TABLE 3.1: The allowed escape sequences in SIMPL string literals.

ESCAPE SEQUENCE	MEANING
`\n`	Linefeed (end-of-line)
`\t`	Tab
`\"`	Double quote (so we can use it in a string)
`\\`	Backslash (so we can use it in a string)

3.1.3 Valid characters

Except inside comments and string literals, any character not explicitly allowed by the EBNF **MUST** be an error anywhere in SIMPL source code. Comments may contain any character from the UTF-8 character set,* which, for all practical intents and purposes, means you skip everything that is not "`(*`" or "`*)`".

String literals **MUST** be handled according to what the Java assembler Jasmin allows. To be safe, we restrict ourselves to the 7-bit ASCII characters, as can be determined by the functions given in the `ctype.h` header, and we only allow the escape sequences given in TABLE 3.1. If a backslash is followed by anything other than specified in TABLE 3.1, you **MUST** display an error and exit. *Note, however, that you **MUST NOT** convert an escape sequence into its equivalent ASCII code.* We must not do the conversion because the Java assembler expects strings to contain raw escape sequences, that is, with the escape sequences *not* converted to ASCII codes. Also, the two delimiting double quote characters are not part of the string and **MUST NOT** be stored by scanner.

3.1.4 Identifiers

The maximum length for identifiers **MUST** be 32 characters, not including the terminating null character. As always on a Unix system, identifiers **MUST** be case-sensitive.

3.1.5 Integer domain

Since SIMPL has the JVM as target architecture, integral numbers **MUST** be signed 32-bit integers. Note that the EBNF does not provide for literals of negative numbers, which is to say, the scanner itself cannot recognise negative numbers. They may, however, be written by negating a positive number; see the optional minus in the $\langle$*simple*$\rangle$ rule. Therefore, the magnitude of our integers are symmetric about zero (whereas two's complement signed

*A note to the curious: The UTF-8 character set, the default character set on modern Unices, is backwards compatible with 7-bit ASCII in its lower seven bits. However, to encode the full complement of Unicode characters, it uses a variable length encoding. Letters for most European and Middle Eastern languages fit into one- and two-byte sequences, whereas ideograms for Eastern languages often take up three bytes each. So, technically speaking, if we allow any UTF-8 character, we have to make sure that whatever character we react to for delimiting comments are not part of a multibyte sequence. However, since few, if any, of you will know Unicode character codes and be as anal as I am about setting up a Compose key to produce diacritics, we leave the matter here.

integers are actually asymmetric about zero since zero is treated as a nonnegative number)—refer to Appendix A in your textbook for more detail; we won't pay the matter any further attention here.

While scanning an integer, you **MUST** ensure that

$$10v + d \leq \texttt{INT\_MAX}, \tag{3.1}$$

where v is the current value of the integer scanned thus far, $d \in \{0, 1, \ldots, 9\}$ is the numeric face value of the next digit, and `INT_MAX` is the maximum positive magnitude of the system's `int` type, as specified in `limits.h`. You **MUST** test for overflow *before* it occurs, and you **MUST NOT** do the conversion by converting to a type other than `int`. Remember: C does not guarantee a `long` value is at least 64 bits. Therefore, we cannot simply use the `long` and flag overflow after it has already occurred. Rewrite Eq. (3.1) into a suitable form, so that you can test for overflow before it occurs.

3.1.6 Reserved words

The scanner **MUST** recognised all reserved words in SIMPL, which **MUST** be accomplished by a binary search of the sorted sequence of reserved words specified in the EBNF. A **reserved word** is a sequence of symbols that cannot be used as an identifier, typically, because it appears as a syntactic feature of the language, which, in the case of SIMPL, means it appears as a terminal in the EBNF. Examples of SIMPL reserved words are "`if`" and "`true`".

In SIMPL, all keywords are reserved words, but not all reserved words are keywords, strictly speaking. A **keyword** is a semantic feature of the language, which is to say, it has meaning in the language itself. Strictly speaking, although they are reserved words, "`read`" and "`write`"—the identifiers of the input and output subroutines which constitute SIMPL's standard library—are not keywords.*

3.2 SCANNER ERRORS

The scanner **MUST** recognise the errors given in this subsection. Pay particular attention to the format of the error message as your work will be tested programmatically. In the following, ␣ indicates a single space, and `'` is a normal single quote character. Also, note that messages generally do not end on a full stop, and that no message has initial or trailing whitespace characters.

*Once we add the symbol table, you should understand this better: Although we will not do it this way, "`read`" and "`write`" can be inserted into the symbol table when the compiler starts, and then treated like any other identifier in the source code during lookup, even though they refer to subroutines added as boilerplate code by the code generation unit. Instead, we are going to treat them like keywords, which is to say, we will have logic *inside* the compiler to handle them if and when they appear, and we do not go via the symbol table. As an aside, to make things even crazier and curiouser, the language Fortran allows the user to redefine keywords—yes, even keywords like "`if`" and "`then`"—which means its keywords are not reserved words!

Very important: You **MUST** use the `leprintf` to display your messages. This function will prefix the line number to your error message. *In the following, the line number part of the message is not shown.*

3.2.1 Illegal characters

If any illegal character appears in the input, the following error message **MUST** be displayed:

```
illegal␣character␣'⟨character⟩'␣(ASCII␣#⟨ASCII␣code⟩)
```

Here ⟨*character*⟩ **MUST** be the actual character that is illegal, and ⟨*ASCII code*⟩ **MUST** be its ASCII code. The token position **MUST** be set to the exact position of the offending character.

3.2.2 Numbers that are too large

If any number is too large to handle, the following error message **MUST** be displayed:

```
number␣too␣large
```

The token position **MUST** be set to the first digit of the offending number token.

3.2.3 Identifiers that are too long

If any identifier is longer than 32 characters, the following error message **MUST** be displayed:

```
identifier␣too␣long
```

The token position **MUST** be set to the first digit of the offending number token.

3.2.4 An illegal escape code found in a string

If an escape not listed in TABLE 3.1 appears in a string, the following error message **MUST** be displayed:

```
illegal␣escape␣code␣'⟨escape␣code⟩'␣in␣string
```

Here, ⟨*escape code*⟩ **MUST** be the illegal escape code, *including initial backslash*. The token position **MUST** be set to the backslash of the offending escape code.

3.2.5 Non-printable characters appear in a string

If a string contains a non-printable character, the following error message **MUST** be displayed:

```
non-printable␣character␣(ASCII␣#⟨ASCII␣code⟩)␣in␣string
```

Here, ⟨*ASCII code*⟩ is the ASCII code of the offending character. Note that you **MUST NOT** print the character itself. The token position **MUST** be set to the exact position of the offending character. This error takes precedence over ¶3.2.6.

3.2.6 A string was not closed

If a string has no double quote that properly closes it, the following error message **MUST** be displayed:

```
string␣not␣closed
```

Remember, it is possible to escape a double quote character inside a string. The token position **MUST** be set to the double quote character that starts the offending string.

3.2.7 A comment was not closed

If a comment was not closed or properly nested inside another, the following error message **MUST** be produced:

```
comment␣not␣closed
```

The token position **MUST** be set to the left (opening) parenthesis that start the offending comment. When comments are nested, the innermost comment that is not closed **MUST** be indicated as the offending comment.

3.3 EXERCISES

EXERCISE 3.1 According to the specification, what is the correct error message when a string literal occupies multiple lines? ▮

EXERCISE 3.2 What is the difference between a **reserved word** and a **keyword**? ▮

EXERCISE 3.3 How would you go about adding floating-point numbers to the EBNF? Is it possible to check, in the scanner, the magnitude of floating-point literals? Even if it is, do you think it would be worth the bother? ▮

EXERCISE 3.4 Although SIMPL supports **block comments**, many other programming and input languages also support **line comments**, where everything from a designated line comment character or sequence to the end of the current line is ignored. For example, Python and BASH have "#", LaTeX has "%", and NASM has "`;`", whereas C (some standards), C++, and Java have the sequence "`//`". What would be involved in adding a line comment character to SIMPL, especially considering that (1) multiple, consecutive lines could contain only line comments, and that (2) line comments could be interleaved with block comments? ▮

EXERCISE 3.5 For the really curious and studious: Have a look on the Internet for how industrial-strength compilers handle the difference in specification between regular tokens and context-free productions. In particular, search for "compiler compilers" (sic.), "scanner generators", and "parser generators". ▮

EXERCISE 3.6 Regular languages can be specified concisely by **regular expressions**.[*] They are everywhere: Sure, we use them to specify languages and build compilers ... but chances are if you type your cell, credit card, or ID number in on some app or website, a regular expression ("regex") engine will test whether or not what you have typed in is valid. Also, if you use the "/" command in Vim's command mode, you can specify what you are looking for by a regular expression.[†]

Regular expressions are defined over an **alphabet** (informally, a character set), and support three basic operations, namely (in order of highest to lowest precedence), (1) the **(Kleene) star**, written as a postfix "`*`", indicating zero or more occurrences of a character or character sequence, (2) **concatenation**, written by juxtaposing characters or character sequences, which means one character or character sequence is followed by another, and (3) **alternation**, written as an infix "`|`", which means the character or character sequence either before or after the operator can be matched. For example, the regular expression `ab*|c` means that strings starting with an *a* and followed by any number (including zero) of *b*'s, or just the string "c" can be matched. In set notation, the language defined thus is {`a`, `ab`, `abb`, ..., `c`}.

Most regex engines understand other syntax as well:[‡] We can override precedence with parentheses in the usual way—for example, `a(b*|c)` will match {`a`, `ab`, `abb`, ..., `ac`}, and yes, we can distribute concatenation over alternation, so that this regex is equivalent to `ab*|ac`—the postfix operator "`+`" means one or more occurrences, the postfix operator "`?`" means zero or one occurrence, and a sequence of characters between brackets means that any one of those characters must match. For example, to match a single digit, we can use the regex `[0123456789]`. Fortunately, we can specify a range inside a **bracket expression**, so that we can abbreviate `[0123456789]` as simply `[0-9]`. Now, we can specify an SIMPL ⟨*num*⟩ as `[0-9][0-9]*`, or more succinctly, as `[0-9]+`. Similarly, we can use multiple ranges inside a bracket expression to specify an SIMPL ⟨*id*⟩ as `[a-zA-Z_][a-zA-Z0-9_]*`.

Now, consider the following definition from K&R [11, p. 194]:

> A floating[-point] constant consists of an integer part, a decimal point, a fraction part, an `e` or `E`, an optionally signed integer exponent and an optional type suffix, one of `f`, `F`, `l`, or `L`. The integer and fraction parts both consist of a sequence of digits. Either the integer part of the fraction part (not both) may be missing; either the decimal point or the `e` and the exponent (not both) may be missing.

Using the regex notation defined above, give a regex that will match a floating-point constant in C. And, yes, there is more than one correct way of doing it. ▮

---

[*] And if you're wondering about this long exercise: It is what my PhD thesis is about. I have built a virtual machine for matching regexes (in C, what else? and looking at the V8 JavaScript regex engine for inspiration), and currently, I am trying to optimise how fast I can match. This involves software engineering techniques such as **memoisation**, reasoning over a particular kind of finite automaton called an **alternating finite automaton**, and from a graph-theoretic and algebraic point of view, trying to drag **semirings** into the picture. If I appear unempowered and of more dense verbiage than usual some mornings, you now know why. But I love to discuss these things; if you're interested, let me know.

[†] If you want immediate motivation for learning about regexes, see `http://vimregex.com/`.

[‡] Do you see the similarities between the regex notation and that for EBNFs? [Where and why do they differ?]

CHAPTER 4

Parsing SIMPL

Appendix A contains a lengthy (in second-year terms) description of writing a recursive-descent parser for an LL(1) language. Therefore, if you did not follow everything during the lectures, this should be your first stop. Also, the slides contain an "animation" of the syntax recognition process. ***Before starting on the parser, first make sure that your scanner is working perfectly.***

4.1 OUR APPROACH TO PARSING

Our approach to writing the parser will be simple: Turn the EBNF into a program by the (almost mechanical) application of the "rules" in TABLE A.1. In the last two parts of the project, we shall add additional functionality to the parser, so as to accomplish semantic analysis and code generation.

4.1.1 Parsing functions corresponding to EBNF productions

The idea is to perform an implicit walk of the parse tree for the SIMPL source code being compiled—in other words, we do not build an explicit parse tree. You **MUST** start by writing a `void` function with a `void` parameter list for each production in the EBNF that (1) is not handled by the scanner, unless (2) its right-hand side consists of terminals only, except (3) in the case where a terminals-only production appears in the right-hand side of more than one (other) productions. According to these rules, the productions for ⟨*type*⟩ and ⟨*simple*⟩, for example, have functions, but those of ⟨*addop*⟩ and ⟨*mulop*⟩ do not. There is a global token variable, called `token`, and it is the address of this variable that **MUST** be passed for each call to `get_token`.

You **MUST** name each function `parse_`⟨*production*⟩, where ⟨*production*⟩ is the name of the production parsed by that function. For example, ⟨*program*⟩ will be parsed by a function called `parse_program`. Since a parsing function is not part of the API exported to some other program, it is a good idea to limit your function documentation comment to a text-only version of the production being parsed. This should serve as a quick sanity check when actually writing the function.

The golden rule for recursive-descent parsing with one lookahead symbol is this:

> *Ensure that each parsing function leaves the next, not-yet-handled token ready for the function that follows, that is, the function that will be called next.*

This is why, in the parser skeleton, there is a call to `get_token` before `parse_program` is called. If you follow the rules mechanically—just imagine you are an engineering student—then there will be very small chance of error. Act correctly locally (inside a parsing function), and the global results will simply follow as if by magic.

Do not try to "optimise" your code by skipping testing for tokens, especially if you believe that you are testing for the same token twice. For example, inside the function for ⟨*statement*⟩, you have to write a `switch` to determine which nonterminal to follow, and accordingly, which function to call. If the token is, say, `TOK_WHILE`, do not call `get_token` before the call to `parse_while`, but leave the consumption of `TOK_WHILE` to the latter. (This is not an ad hoc arrangement, but exactly in accordance with TABLE A.1.) ***Remember, there is a big difference between* testing *for a token and* actually consuming *that token.***

4.1.2 Expecting terminals

The parser skeleton has a helper function called `expect`, which tests whether the current token matches an expected token, and if there is match, then calls `get_token`. You **SHOULD** use this function every time you need to consume a terminal according to the EBNF.

There is also a function called `expect_id` that matches against `TOK_ID` and duplicates the token lexeme after a successful match. However, this function only becomes necessary once we do type checking with the symbol table, when the compiler must actually start "remembering" and doing something with identifiers. If you do decide to use `expect_id` while writing your parser, and you free (deallocate) the memory allocated by this function, remember to reconsider your deallocation strategy once you add the symbol table; otherwise, you might very well free a pointer in the parser that must still be valid in the symbol table.

4.2 DEBUGGING

The parser skeleton contains three macros, conditionally wrapping a function each, for reporting debugging status: `DBG_start`, `DBG_info`, and `DBG_end` **SHOULD** be used to display information messages at the start, during, and at the end of a parsing function. These functions take the same parameters as `printf`, but write unbuffered output to the standard error stream. For an example of use, look at the parser skeleton.

The functions underlying these macros are only compiled when the `DEBUG_PARSER` constant is defined, say, via the makefile. This means that you may leave calls to these functions in your code after you are done debugging—but then make sure that `DEBUG_PARSER` is undefined in the makefile, and these functions will not produce any output. Yet, they are still available if some other bugs crop up.

It may seem like a chore to add these functions when you are first writing your code. However, you will appreciate them when you have to debug, as then they can be "turned on and off" by a flick of the `DEBUG_PARSER` flag.

4.3 PARSER ERRORS

All parser errors **MUST** be reported via the `abort_c` function. It takes an error code from the `Error` enumerated type, defined in `errmsg.h`, and thereafter a variable argument list may follow. Use the skeleton code in `abort_c` to finish the function. Note that the case for `ERR_EXPECT` has unique formatting; other cases will be different from `ERR_EXPECT`, but similar to one another.

There is also an `abort_cp` function, which takes an additional parameter, a pointer to a `SourcePos` structure. This parameter overrides the current file position for error reporting. It is useful when you must somehow read past an error position to trigger the error. In many cases, especially while still just parsing (and not checking types yet), it is not necessary to override the file position; if your scanner sets positions correctly, it will not be necessary to use `abort_cp` during parsing.

ERROR POSITION For all of the following, the position reported for an error **MUST** be the first character of the token that was actually found instead of the token that was expected. If the source file ends prematurely, which is to say that end-of-file was reached, the position of the token found **MUST** be the last character in the source file, typically an EOL character.

EXAMPLE 4.1 Suppose you run the `simplc` executable on the source file `test.simpl`, which produces the following error report:

```
simplc: test.simpl:2:14: error: expected ')', but found identifier
```

If you open the source file in Vim, and you enter the command `2G14|`—the last character is this sequence is the vertical bar or pipe character—the cursor should move to this file position, and then must be on top of the first character of the identifier that was found instead of the expected right parenthesis. ▮

Note that tab characters in the source file **MUST** be counted as one column each by the scanner. Therefore, it is **RECOMMENDED** that you use spaces instead of tabs for indentation of SIMPL source code. To keep things easy, this is how our test cases are set up.

4.3.1 Missing type

If a type token, `TOK_BOOLEAN` or `TOK_INTEGER`, is missing, the following error message **MUST** be displayed:

```
expected␣type,␣but␣found␣⟨token⟩
```

Here, ⟨*token*⟩ **MUST** be the string representation, as returned by `get_token_string`, of the token found instead of the type token. This case is handled separately because there is no suitable “`TOK_TYPE`” token available.

4.3.2 Missing statement

If a statement is missing, the following error message **MUST** be displayed:

`expected␣statement,␣but␣found␣`⟨*token*⟩

Again, ⟨*token*⟩ is the string representation of the token actually found. This case is handled separately because different kinds of statements are given in the EBNF.

4.3.3 Missing factor

If a factor is missing, the following error message **MUST** be displayed:

`expected␣factor,␣but␣found␣`⟨*token*⟩

Yet again, ⟨*token*⟩ is the string representation of the token actually found. This case is handled separately because different kinds of factors are given in the EBNF.

4.3.4 Missing argument list or variable assignment expression

The ⟨*name*⟩ production handles both variable assignment and procedure calls. As such, an identifier matched by this production must either be followed by an argument list or a variable assignment expression; if it is not, the following error message **MUST** be displayed:

`expected␣argument␣list␣or␣variable␣assignment,␣but␣found␣`⟨*token*⟩

The ⟨*token*⟩ is the string representation of the token actually found.

4.3.5 Missing array allocation sequence or expression during assignment

If there is no expression or array allocation sequence after `TOK_GETS` in the assignment part of a ⟨*name*⟩ statement, the following error message **MUST** be displayed:

`expected␣array␣allocation␣or␣expression,␣but␣found␣`⟨*token*⟩

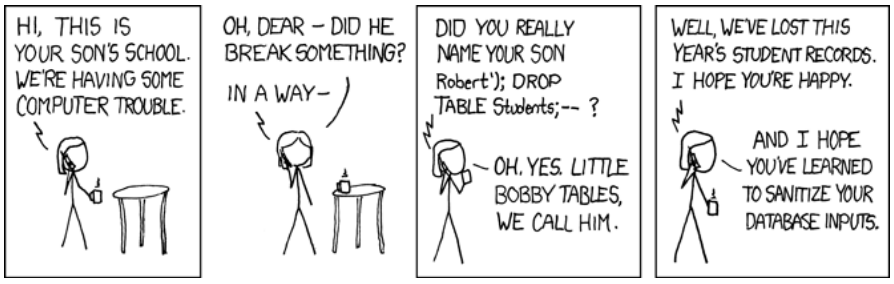
Again, ⟨*token*⟩ is the string representation of the token actually found.

4.3.6 Missing string or expression in output statement

If a string or expression is missing in an output statement, the following error **MUST** be displayed:

`expected␣expression␣or␣string,␣but␣found␣`⟨*token*⟩

Again, ⟨*token*⟩ is the string representation of the token actually found.


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

SOURCE: `https://xkcd.com/327/`

FIGURE 4.1: Exploits of a mom.

4.3.7 Unexpected terminal found

This kind of error is handled by the function `expect`, and besides the correct use of this function, there is nothing you have to do.

4.3.8 Unreachable code

There is an extra error constant for unreachable code. It **SHOULD** be used in the `default` cases of switch statements. If your parser works correctly, the control flow of your compiler will never reach the point at which that error is triggered. However, if you make a mistake, and some tokens "disappear" or are repeated, having compilation abort on this error is useful for debugging.

Especially in a program like a compiler, in which you are parsing input that could be incorrectly formatted, or even malicious by nature (think of all the wonderful things the demis and I will dream up), it is a good idea in most cases to have `else` clauses for all `if` statements, and `default` cases for all switches. As a rule of thumb, all grouped parts of EBNF productions that contain one or more choices **SHOULD** have a default case or else clause.

All this is to say: Test for everything explicitly. For example, if there are only five possible cases in a switch, do not test only for the first four and handle the fifth as the default case. Rather, test for all five cases independently, and keep the default case for the seemingly impossible. Depending on your skill as programmer, the seemingly impossible might actually be quite probable. Attackers get up to all manner of evil; it is just good practice to be slightly paranoid about security exploits; refer to FIGURE 4.1.

4.4 EXERCISES

EXERCISE 4.1 Why do you think I did not write the production for ⟨*expr*⟩ as follows?

$$\langle expr \rangle = \langle simple \rangle \left\{ \langle relop \rangle \langle simple \rangle \right\}.$$

(Note the braces instead of brackets.) ▮

EXERCISE 4.2 After studying the EBNF for SIMPL, would you consider the semicolon to be a **statement terminator** or a **statement separator**? Is this the same as for languages like C and Java? ▮

EXERCISE 4.3 If we decide to employ the semicolon strictly as a **statement terminator**, how can we simplify the grammar—that is, remove unnecessary terminals—and still keep it LL(1)? ▮

EXERCISE 4.4 One could argue that the ⟨*name*⟩ production of the SIMPL grammar in FIGURE 3.1 tries to handle too much. What do you think? How can you refactor this production so that procedure calls and assignment expressions are specified in different productions, but the grammar is still LL(1)? ▮

CHAPTER 5

Semantic analysis

Now that the parser is able to determine whether or not a source program has valid syntax, we can move on to consider **semantics**, that is, the meaning of the program. The final meaning of the program will become apparent once we have the equivalent Java bytecode instructions; this is done during the final part of the project, code generation. But first, we have to check whether expressions in the source program make sense. This is done by evaluating data type information and comparing it to information gathered into a symbol table.

5.1 THE SYMBOL TABLE

A **symbol table** is a data structure that holds information about identifiers in source programs [1]. Typically, such information includes its lexeme, its type, and its position in storage. For an identifier that denotes a function or a procedure, this also includes information about its parameter list and its return type.

Symbol tables can be implemented in a number of ways, but almost intuitively, the most suitable data structure seems to be a hash table, which associates values with keys. Both key and value are chosen from predetermined, but arbitrary sets. As such, a hash table is a logical extension of the concept of an array. But where an array can only index over a dense set of integral values, hash tables can "index" over an arbitrary set of values—provided that these values can be turned into integral values by some deterministic (and preferably, rapid) procedure.

For more information on symbol tables in general, and hash tables in particular, refer to Sedgewick and Wayne [17]. You should note, however, that we employ the term *symbol table* here specifically with respect to use in a compiler, which is to say, for the component that stores and retrieves data associated with identifiers and some other values and symbols. In contrast, Sedgewick and Wayne use "symbol table" simply to refer to a data structure that associates values with keys, like the map data types in Java's libraries, or the dictionary data structure built into Python. To distinguish between our idea of a symbol table and that of Sedgewick and Wayne, in the sequel, we use the term **map** for the latter.

In this project, we implement the symbol table in two separate program units:

1. a general hash table, able to store arbitrary key–value pairs, made possible by the use of `void` pointers; and
2. a SIMPL-specific symbol table that wraps around the hash table to implement the association of identifiers with their attributes.

5.1.1 Maps and hashing

MAPS A **Map** allows us to store elements so that they can be retrieved rapidly using **keys**. One efficient way to implement a map is to use a **hash table**, a key-indexed, table-based search mechanism that essentially generalises arrays. Key-indexed search uses keys as array indices rather than comparing them and depends on the keys being distinct integers falling in the same range as the table indices. If the keys are distinct small integers, for example, we may store an item with key k at index k in an array. Given the key k, inserting, deleting from, and searching the array takes $O(1)$ time.

If we do not happen to have keys with such fortuitous properties, we use **hashing**. We still use an array, but one with length proportional to the number of entries actually stored, not to the total number of keys available in the key space. From any given key, we then *compute* the array index at which an entry will be stored. (When we do hashing, we usually refer to the underlying array as a "table".) Under some fairly general and generous conditions, we still have $O(1)$ time performance on average for insertion, search, and deletion (if the latter is implemented).

Hashing algorithms consist of two separate parts. The first step is to compute a **hash function** that transforms the key into a table address. Ideally, different keys would map to different table entries, but in practice, different keys often map to the same table entry, especially if the table size is small. Thus, the second step in a hashing algorithm is to choose a **collision-resolution** process that deals with such colliding keys. **Open addressing** schemes, for example, attempt to find a different open slot in the table itself. For this project, however, we will use **separate chaining** for collision resolution, where each entry in the table is a linked list, and after hashing a key, a linear search of the list at the appropriate index may locate the associated value.

HASH FUNCTIONS What remains is to consider the hash function, which transforms keys into table addresses. If we have a table that stores N items, we need a function that transforms a key into an integer in the range $[0, N-1]$. Our ideal hash function is one that is easy to compute and approximates a uniformly random function: For each input, every output in $[0, N-1]$ should be, in some sense, equally likely.

The hash function depends on the key type. Integer keys, for example, might be mapped to the required range simply by applying the remainder (also known as the modulo) operator. So, for an integer k, we compute $k \bmod N$ to get an index into the table. However, we have to bear in mind that the C modulo operator "%" is defined slightly differently from what we might expect from pure mathematics: You have to be careful with negative operands.

Also, it would seem that this modular mapping is sensitive to the table size N. It is not unreasonable to assume that there might be some pattern to the keys. If N is a composite integer, and the keys are multiples of a factor of N, we get many collisions, which we call **clustering**. For example, for $N = 24$, any multiple of 2, 3, 4, 6, 8, 12, and 24 creates clusters—even integers larger than 23 would "wrap around". So, it would seem that a prime N is a good value for the table size. Unfortunately, convenience rarely is gratis. Primes are intimately linked to the problem of factorisation, which is hard[*] and therefore the basis of cryptographic systems in common use.[†] If we select the table size once, and then stick to it, no problem. However, as we shall see later, it is convenient if we could let the table size grow with the number of items we store in the table. But then we would have to find a prime number for the new table size when we are resizing, which implies that we either have to store a sequence of prime tables sizes, or that we have a way of rapidly computing a new prime table size.

5.1.2 Implementing a general hash table

Although C provides neither generics nor templates, we are are able to write a general hash table implementation—one that may have keys and values of arbitrary types—because C allows `void` pointers. Since the C compiler does not guarantee anything about the types to which these pointers point—no run-time type checks are performed—and since all pointer values (that is, addresses) on a particular architecture is of the same size, we can cast to and from `void` pointers with impunity. In a sense, they are assignment compatible with other pointers in the same way that any Java object reference can be stored in a variable of type `Object`.

EXAMPLE 5.1 Suppose a function `produce` creates a new string, and therefore, has `char *` as return type. Suppose also that a function `consume` uses a string in some way. If we create a data structure which allows storage and retrieval of arbitrary pointers, its insertion function, say, `put`, must have a `void *` parameter, and the retrieval function, say, `get` must have a `void *` return type. The function `produce` can now create an array of characters in some way, and pass this to `put`, because we may pass a `char *` value to a `void *` parameter.

[*]Describing a computational problem as "hard" is not the same as saying, for example, "Mathematics is hard"—here, "hard" means "difficult", or for a student, "difficult to understand." A second meaning could be that it "requires a lot of effort", and it is this meaning we usually use when describing computational problems as hard. Calling a computational problem hard means that the resources required by the known algorithmic solutions do not scale well over the input size. That is, for large inputs, these solutions would either run very slowly (as in longer than the age of the known universe in extreme cases) or have very large memory space requirements (as in more atoms than in the known universe in extreme cases), or both. True, many of these problems are difficult to understand too, but others, like factorisation, we understand, and solve for small examples, already in primary school.

[†]This is a marvellous example of something that is easy to compute in one direction, but hard in the reverse. Computationally speaking, it is relatively easy to multiply two large primes p and q, but hard to factorise their product pq.

Similary, `consume` may retrieve a pointer via `get`, and simply cast the retrieved `void *` to `char *` before use. ▮

We also employ function pointers.[*] When a hash table is initialised, two functions are passed as parameters to the initialisation function: (1) one that can hash keys from our designated keyspace, and (2) another that can compare two particular keys and that returns −1, 0, or 1, depending on whether the first key is less than, equal to, or greater than, respectively, the second key. These functions must be provided by whatever unit creates and manages the hash table, because the hash table itself is completely blind to the type of information that it stores.

5.1.3 The load factor

To measure the occupancy of a map, we use a metric called the **load factor**,

$$\lambda = \frac{n}{N}\,, \tag{5.1}$$

where n is the number of entries in the map and N is the table size. When we start out, because $n = 0$, also $\lambda = 0$, but as we start adding entries, λ starts growing to 1. Once we hit $\lambda > 1$, we are 100% sure that there are collisions in the hash table. (Actually, owing to how we expect keys to be distributed in the keyspace, we typically start getting collisions much sooner!)

If there are too many collisions, the performance of the hash table starts to degrade: Too many lookups result in us having to walk down the **chains** (linked lists) attached to each **bucket** (position) in the table. Therefore, during initialisation, we take a floating-point parameter that gives the maximum load factor λ_{max} allowed for that particular hash table. Once λ exceeds λ_{max}, we rehash to a larger underlying table, and the colliding elements spread out again, statistically speaking.

To **rehash** a table, we simply (1) choose a size for the new, larger table, (2) allocate space for the new table, (3) traverse the old table, and for each element, we (4) compute a new hash value (with respect to the new table), and (5) insert it into the new table. Then, we release the memory occupied by the old table, which works out just fine, because the existing entries have been copied to the new table. Although steps 2 and 5 seem to suggest creating an entirely new structure to replace the old structure, things are actually simpler. When, as in our implementation, we have access to the underlying nodes in the linked list, in step 5, we can simply unlink the node we are currently rehashing, and then relink it into the new table.

The code skeleton for the hash table contains an integer array called `delta`, which gives the difference between 2^k, where k is an index into `delta`, and the largest prime less than 2^k. For example, for $k = 4$, the entry in `delta` is 3, and we get the largest prime less than 2^4 as $2^4 - 3 = 13$. When rehashing, use this array to compute a new underlying table size that is both prime and almost double the previous size.

[*] On some architectures, function pointers may be an exception to the rule of all pointers being the same size. As a matter of fact, the C standard does not explicitly specify all pointers to be of the same size. But it does specify that we must be able to assign to and from `void` pointers without loss of information, and making all pointers the same size is an easy way to accomplish this.

5.1.4 Wrapping a symbol table around a hash table

With the heaving lifting done in the hash table implementation, it is now relatively easy to create a symbol table by rephrasing, as it were, the symbol table functions as calls to an underlying hash table. As long as we provide suitable hash and comparison function implementations, all will be well.

Our symbol table associates alphanumeric identifiers with their type properties. So, the identifiers in the SIMPL source file being compiled are the keys of the hash table. This means we have to create a hash function capable of hashing alphanumeric strings, and similarly, a comparison function that can compare alphanumeric strings (in lexicographic order). Fortunately, the latter already exists in the form of `strcmp`, so we only have to worry about the hash function.

5.1.5 The hash function

Now, how do we hash keys that are not integers? A naive approach would be simply to sum the components—in the case of strings, the numeric values of the constituent characters. However, consider the five strings "`post`", "`pots`", "`spot`", "`stop`", and "`tops`". All five consist of the same characters and, under a simple summation scheme, would yield the same result. A better solution needs to take ordering into account.

An alternative to the simple summation scheme that takes position into account is to view any structured value type x, sensitive to ordering, as a tuple $(x_0, x_1, \ldots, x_{j-1})$ of primitive, atomic types, and then to choose a nonzero constant $a \neq 1$ so that we can use as hash function the expression

$$x_0 a^{j-1} + x_1 a^{j-2} + \cdots x_{j-2} a + x_{j-1}, \tag{5.2}$$

which is then used, modulo the table size, as table index. Mathematically speaking, this is simply a polynomial in a that takes the components $(x_0, x_1, \ldots, x_{j-1})$ of x as its coefficients. Such a hash function is, therefore, called a **polynomial hash function**. By Horner's Rule, this polynomial can be written

$$x_{j-1} + a\Bigg(x_{j-2} + a\bigg(x_{j-3} + \cdots + a\big(x_2 + a(x_1 + a x_0)\big)\cdots\bigg)\Bigg), \tag{5.3}$$

and then computed efficiently. [Comparing Eq. (5.2) to Eq. (5.3), can you see why computing the latter is more efficient than computing the former? Count the number of operations.] Intuitively, a polynomial hash function uses multiplication by the constant a as a way of "making room" for each component in a tuple of values while also preserving a characterisation of the previous components.

I **RECOMMEND** you use a variant of the polynomial hash function, where multiplication by a is replaced by the **cyclic shift** of partial sums by a fixed number of bits. In C, we can specify an integral variable to be unsigned, so we do not have to worry about overflow affecting the sign. Compared to other instructions, bit shifts are extremely fast, but with all the good properties of the polynomial hash functions, which is why I recommend them. Just remember, whatever hash function $h(x)$ you choose, you still have to use $k = h(x) \bmod N$ as the table index.

5.2 ADDING TYPE CHECKING TO THE PARSER

With an operational symbol table, we can now *extend the parser* to do type checking. That is, we add some statements that check type to those functions already checking syntax. A compiler does type checking by (1) assigning a type expression to each component of the source program, and then (2) checking whether these type expressions conform to logical rules.

5.2.1 Type checking of expressions

SIMPL is **strongly typed** in that our compiler guarantees the compiled code will run without type errors.[*] To specify typing for expressions, we rewrite some productions in the EBNF as an attributed BNF grammar (see §B). Then we update the parse functions for $\langle$*expr*$\rangle$, $\langle$*simple*$\rangle$, $\langle$*term*$\rangle$, and $\langle$*factor*$\rangle$ to be able to propagate typing information to where it can be handled appropriately. This means we have to add reference parameters for these functions.

The attributed grammar for SIMPL is given in TABLE 5.1, which is to be interpreted thus: The SYNTAX column gives the BNF of a particular production, and the notation "expr(T_0)" is read as "$\langle$*expr*$\rangle$ has the type T_0"; the ATTRIBUTE column specifies how types are propagated; and the CONTEXT provides additional constraints on the types in question. Further notations are array, bool, function, and int for arrays, boolean, functions, and integer, respectively, and the use of italics for a grammar element retrieved as a single token from the scanner. The functions base(T), find(w), nparams(w), and ptype(w, i) are described below the table.

The quick (and easy) way to explain how it works is to consider some examples.

EXAMPLE 5.2 The first row in the table says that if an expression $\langle$*expr*$\rangle$ is just a simple expression $\langle$*simple*$\rangle$—that is, it is not a relational expression—then the type of the $\langle$*simple*$\rangle$ becomes the type of the $\langle$*expr*$\rangle$. For example, for `b` a boolean variable, the expression `not b` must still have a type of bool; and for `i` an integer, the expression `i + 1` must still be of type int. ▮

EXAMPLE 5.3 The second row in the table says that if an expression $\langle$*expr*$\rangle$ is a test for equality on two simple expressions, then the type of $\langle$*expr*$\rangle$ is bool. But there is also the context that the type of the simple expressions must be the same: We cannot test whether a boolean value is equal to an integer. For example, for boolean variables `a` and `b`, and integer variables `i` and `k`, both the expressions `a = b` and `i = k` are valid, and are of type integer. However, `a = k` is invalid, because the type of `a` is not the type of `k`. ▮

[*] Waving hands, typing in C is **static**—determined at compile time—but **weak**, because, for example, `void` pointers exist, and integral types can be freely mixed. Python is strongly typed in that there are few implicit type conversions; but its typing is **dynamic** in that typing is enforced only at run time.

TABLE 5.1: The attributed grammar for SIMPL expressions.

SYNTAX		ATTRIBUTE	CONTEXT
expr(T_0) =	simple(T_1)	$T_0 \leftarrow T_1$	
	$\mid$ simple(T_1) “=” simple(T_2)	$T_0 \leftarrow$ bool	$T_1 = T_2$
	$\mid$ simple(T_1) “#” simple(T_2)	$T_0 \leftarrow$ bool	$T_1 = T_2$
	$\mid$ simple(T_1) “>=” simple(T_2)	$T_0 \leftarrow$ bool	$T_1 = T_2 =$ int
	$\mid$ simple(T_1) “>” simple(T_2)	$T_0 \leftarrow$ bool	$T_1 = T_2 =$ int
	$\mid$ simple(T_1) “<=” simple(T_2)	$T_0 \leftarrow$ bool	$T_1 = T_2 =$ int
	$\mid$ simple(T_1) “<” simple(T_2)	$T_0 \leftarrow$ bool	$T_1 = T_2 =$ int
simple(T_0) =	term(T_1)	$T_0 \leftarrow T_1$	
	$\mid$ “-” term(T_1)	$T_0 \leftarrow T_1$	$T_1 =$ int
	$\mid$ term(T_1) “+” simple′(T_2)	$T_0 \leftarrow T_1$	$T_1 = T_2 =$ int
	$\mid$ term(T_1) “-” simple′(T_2)	$T_0 \leftarrow T_1$	$T_1 = T_2 =$ int
	$\mid$ term(T_1) “or” simple′(T_2)	$T_0 \leftarrow T_1$	$T_1 = T_2 =$ bool
term(T_0) =	factor(T_1)	$T_0 \leftarrow T_1$	
	$\mid$ factor(T_1) “*” term′(T_2)	$T_0 \leftarrow T_1$	$T_1 = T_2 =$ int
	$\mid$ factor(T_1) “/” term′(T_2)	$T_0 \leftarrow T_1$	$T_1 = T_2 =$ int
	$\mid$ factor(T_1) “rem” term′(T_2)	$T_0 \leftarrow T_1$	$T_1 = T_2 =$ int
	$\mid$ factor(T_1) “and” term′(T_2)	$T_0 \leftarrow T_1$	$T_1 = T_2 =$ bool
factor(T_0) =	*num*	$T_0 \leftarrow$ int	
	$\mid$ *id* idf(T_1, *id*)	$T_0 \leftarrow T_1$	
	$\mid$ “(” expr(T_1) “)”	$T_0 \leftarrow T_1$	
	$\mid$ “not” factor(T_1)	$T_0 \leftarrow T_1$	$T_1 =$ bool
	$\mid$ “false”	$T_0 \leftarrow$ bool	
	$\mid$ “true”	$T_0 \leftarrow$ bool	
idf(T_0, *id*) =	“[” simple(T_1) “]”	$T_0 \leftarrow$ base(T_2)	$T_1 =$ int
			$\wedge\ T_2 \leftarrow$ find(*id*)
			$\wedge\ T_2 \subseteq$ array
	$\mid$ “(” expr(T_1) param(k_1) “)”	$T_0 \leftarrow T_2$	$T_1 =$ ptype(*id*, 0)
			$\wedge\ T_2 \leftarrow$ find(*id*)
			$\wedge\ T_2 \subseteq$ function
			$\wedge\ k_0 \leftarrow k_1 + 1$
			$\wedge\ k_0 =$ nparams(*id*)
	$\mid$ ε	$T_0 \leftarrow T_1$	$T_1 \leftarrow$ find(*id*) $\wedge\ T_1 \nsubseteq$ function
param(k_0, *id*) =	param′(k_1, *id*) “,” expr(T_1)	$k_0 \leftarrow k_2$	$k_2 \leftarrow k_1 + 1$
			$\wedge\ n \leftarrow$ nparams(*id*)
			$\wedge\ T_1 =$ ptype(*id*, $n - k_2$)
	$\mid$ ε	$k_0 \leftarrow 0$	

base(T)	returns the base type of array type T
find(w)	returns the type of w (if w is in the symbol table)
nparams(w)	returns the number of parameters of the function named w
ptype(w, i)	returns the type of parameter i of the function named w, where $i \in [0, \text{nparams}(w) - 1]$

EXAMPLE 5.4 The fifth row in the table says that if an expression ⟨*expr*⟩ is a test to see whether the first simple expression is greater than the second, then the type of the ⟨*expr*⟩ is bool, and the context is that both simple expressions must be of type int. This is to say it does not make sense to test whether one boolean value is greater than another. ▮

Note that TABLE 5.1 essentially specifies typing to be propagated bottom-up, that is, from the leaves of the parse tree. If you think about it some, it does make intuitive sense: A ⟨*num*⟩ is definitely an integer, and "`true`" and "`false`" are definitely boolean values; also, the type of an identifier can simply be looked up in the symbol table.

Handling identifiers is a bit more involved. For example, we have to ensure that we do not try to "call" variables or "assign to" functions. The new ⟨*idf*⟩ production handles everything that may *follow* an ⟨*id*⟩ in the ⟨*factor*⟩ production.

EXAMPLE 5.5 The ε-option[*] of ⟨*idf*⟩ says that if an ⟨*id*⟩ is not followed by an index or an argument list, it must be a non-array variable. We verify this by checking in the symbol table that the ⟨*id*⟩ is neither an array nor a function. Its type is simply what is saved in the symbol table. ▮

EXAMPLE 5.6 The first option of ⟨*idf*⟩ says that if ⟨*id*⟩ is followed by an index, then it must be an array name. Also, the index itself must of type integer. However, the retrieved array element itself cannot be of an array type. Instead, it must have the base type of the array. For example, if `a` is an integer array, then `a[0]` must be an integer. ▮

EXAMPLE 5.7 The productions for a function call work similarly, except that we check not only the return type, but also the types and number of parameters. It looks complicated when stated formally, but it is just the obvious: The types of the arguments to a function must correspond to the types of the parameters of that function, and also, we cannot pass too few or too many arguments when we call a function.

Do not be troubled by the $n - k_2$ in the one call to ptype. It is just because, in a BNF, we are essentially handling a production from right to left. (Because EBNFs are left-to-right, they is easier to implement as a parser by hand.) If anything else bothers you, look at Exercise 5.2. ▮

5.2.2 Type checking of statements

Each SIMPL statement, except the singleton "`chill`", must also perform type checking. For example, variable assignment must happen on an expression of the appropriate type, and the loop condition expression for a "`while`" statement must be of type bool. An attributed grammar can be written for these, but as is evident from our machinations with the ⟨*id*⟩s, it will not be as succinct as TABLE 5.1, and we stick with the following list of rules:

1. Only procedures—subroutines without any return type—may be "called" by the ⟨*name*⟩ production. Similarly, a procedure may be not be called from inside an expression.

[*]Remember, the epsilon ε refers to the empty string.

2. The type of the ⟨*simple*⟩ expression following the keyword "`array`" in ⟨*name*⟩ **MUST** be of type int. That is, an array must be created with an integer size. Do not worry about negative array sizes. Once the JVM class file is generated, the JVM will pick that up as a run-time error.

3. The type of the ⟨*expr*⟩ that may follow the "`<-`" operator in ⟨*name*⟩ **MUST** match the type of the ⟨*id*⟩. Also, the "`array`" keyword is only allowed when (a) the ⟨*id*⟩ is an array, and (b) the ⟨*id*⟩ is not followed by an index. The reverse is also true: You cannot assign a scalar boolean or integer value to an "unindexed" array. Of course, you cannot assign to a function or a procedure identifier either.

4. The ⟨*expr*⟩ following the "`if`" and "`elsif`" keywords **MUST** be of type bool.

5. The ⟨*expr*⟩ following the "`while`" keyword **MUST** be of type bool.

6. The ⟨*simple*⟩ expression between brackets, denoting indexing into an array, **MUST** be of type int.

7. When calling a function or a procedure, the types of the arguments **MUST** match the types of the parameters exactly.

If you are afraid of missing an error, use the error constants in `errmsg.h` as guidelines. Use the WWWD principle: What Would Willem Do (to try and break your code)?

5.2.3 Changing the parser functions

Some parser functions' prototypes must be changed so that type information can be propagated properly. In particular, the functions parsing the productions ⟨*expr*⟩, ⟨*simple*⟩, ⟨*term*⟩, and ⟨*factor*⟩ must each have a single parameter, a pointer to a value type enumeration, instead of a `void` parameter list.

EXAMPLE 5.8 The new signature for the function that parses the ⟨*expr*⟩ production is as follows.

```
void parse_expr(ValType *type)
```

Note that the return type of `void` is unchanged. *We can use the pointer parameter as both input and output to the function.* ∎

Since the `ValType` parameter is a pointer, space for the data must be reserved in some way. In typical C, there are two places to allocate space: (1) on the heap, via a call to `malloc`, and (2) on the stack, via a normal variable definition. In the latter case, a pointer to the variable can passed by using `&`, the address operator, on an existing local variable, and *this is what we do*.

The question is now: Where do we define such variables? A little thought shows that each of the functions in TABLE 5.1 must pass its type to its caller. So, the caller is the logical place to define the variables. A quick look at the EBNF should convince that you that, in many cases, the callers are those functions discussed in §5.2.2.

EXAMPLE 5.9 The function parsing the ⟨*while*⟩ production must have a local variable of type `ValType`. When the parse function for the loop guard, which is an ⟨*expr*⟩ production, is called, the address of this `ValType` variable is passed by prefacing its identifier with the address operator. ▮

Another glance at the EBNF should also convince you that the functions discussed in §5.2.2 are not the only ones to have such definitions. For example, the function parsing ⟨*expr*⟩ itself gets at least one type from a call to the function parsing ⟨*simple*⟩. So, it too has at least one definition of a `ValType` variable.

A parse function that originates a type, either by recognising a token of a fixed type (like the token for "`true`") or by a call to the symbol table, can pass this information back to its called by assignment on the *dereferenced* pointer. This is why we do not change the return types of the functions involved.

5.2.4 Representing the SIMPL types

A look at `symboltable.h` and `valtypes.h` might convince you that something funny is afoot, because `ValType` is actually just an enumeration. However, consider the following definition from `valtypes.h`, and pay particular attention to the binary forms (only the least significant nibble) of the integers, given in comments:

```
typedef enum valtype {
    TYPE_NONE     = 0,    /* 0000 */
    TYPE_ARRAY    = 1,    /* 0001 */
    TYPE_BOOLEAN  = 2,    /* 0010 */
    TYPE_INTEGER  = 4,    /* 0100 */
    TYPE_CALLABLE = 8     /* 1000 */
} ValType;
```

Do you see that none of the 1-bits clash with another 1-bit in the same column, that is, vertically? Mathematically, the array type is 2^0, boolean is 2^1, integer is 2^2, and callable is 2^3. We make the definitions thus so that we can use a `ValType` value as a **bit field** [22]. Essentially, we use a bit position in a `ValType` value as a little on/off switch: We are not interested in the value of a `ValType` variable as a number, but rather in the individual bits. Having a 1 at a particular bit position means the value has the type property corresponding to the bit position; a 0 means it does not. Using **masks** [23], we can turn a bit on or off, flip it, or get its value.

EXAMPLE 5.10 To set the type for a SIMPL boolean variable, we simply assign the enumerated value `TYPE_BOOLEAN` to the `ValType` field in the variable's identifier properties structure; also see LISTING 5.1. ▮

EXAMPLE 5.11 To set the type for a SIMPL integer array, we need to use the value $5_{10} = 101_2$. In the binary form, the least significant 1 corresponds to being an array, and the most significant 1 to being integer-valued. How to set up the `ValType` value in this way is discussed in the paragraphs below. ▮

EXAMPLE 5.12 To set the type for a SIMPL function that returns a boolean value, we need to use the value $10_{10} = 1010_2$. ▮

EXAMPLE 5.13 A procedure is the only construct in SIMPL without data type; but it still needs `TYPE_CALLABLE` so that we know we can call it. ▮

In memory, we cannot address bits directly. The best we can do is address a byte. C does provide portable **bit fields**, but we are going to use a more old-school approach, which will also be useful once we start programming in assembly language.*

BIT MASKING To isolate a single bit, we use a mask. The mask for bit k—indexing from right to left in the usual way for numbers—is simply 2^k. In binary, a power of two has exactly one bit set to 1, and the rest are 0.

EXAMPLE 5.14 Consider an 8-bit byte. A bit mask for the fourth bit, counting from the right, is $2^3 = 8_{10} = \texttt{00001000}_2$; we use an exponent of 3, because we start indexing at 0. Also note our notation here. In C, a integer literal that starts with `0` is treated as octal, and an integer literal that starts with `0x` is treated as hexadecimal. Our notation uses a subscript to give the base. When we write $\texttt{00001000}_2$, it is base-2 (binary), and we use leading zeros to emphasise the 8-bit width of the byte, not to indicate that it is in any way octal. In the sequel, such bit patterns are also written in a monospaced (typewriter) typeface. ▮

Finally, note that bit masks can also mask more than one bit. For bits k and ℓ, where $k \neq \ell$, the bit mask is simply $2^k + 2^\ell$. [Why? Just pause a moment, and write out some examples.] However, when we build and use bit masks, we usually do not use arithmetic expressions. Instead, we use the bitwise operators, because that is how we implement them, for doing so is faster to calculate for a computer, and therefore, the low-level thing to do.

BITWISE OPERATORS C defines six operators for bit manipulation: Four bitwise logic operators, and two additional operators for shifting; they are given in TABLE 5.2. When we say an operator is "bitwise", it means the operation defined by the operator is applied to each bit in the operand(s) in parallel. The truth tables for the bitwise operators are given as TABLE 5.3.

EXAMPLE 5.15 Consider two `char` variables `c1` and `c2` containing, respectively, the values $28_{10} = \texttt{00011100}_2$ and $56_{10} = \texttt{00111000}_2$. Then `c1 & c2` $= \texttt{00011000}_2 = 24_{10}$, `c1 | c2` $= \texttt{00111100}_2 = 60_{10}$, `c1 ^ c2` $= \texttt{00100100}_2 = 36_{10}$, and `~c1` $= \texttt{11100011}_2 = -21_{10}$. To understand the last result, remember that `char` is equivalent to `signed char`, and also that

*And always remember, the main focus of this module's practicums is to give you rapid immersion, in just 13 weeks, from Java into C and assembly language.

TABLE 5.2: C operators for bit manipulation.

OPERATOR	DESCRIPTION	ARITY
`&`	bitwise AND	binary
`|`	bitwise (inclusive) OR	binary
`^`	bitwise exclusive OR (XOR)	binary
`~`	one's complement	unary

TABLE 5.3: Truth tables for the C bitwise operators.

a	b	a & b	$a \mid b$	a ^ b	~a
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

C uses **two's complement representation** for signed integer representation [20, Appendix A]. ▮

EXAMPLE 5.16 The bitwise operators `&` and `|`, and the logic operators `&&` and `||` are closely related, but not equivalent. Consider two `char` variables `b1` and `b2` containing the decimal values 12 and 3, respectively. Then `b1` & `b2` = 0, but `b1` && `b2` = 1. However, as we will soon see, when compiling to the JVM, we need only the bitwise versions to implement the logic operators. In addition, short-circuiting behaviour can be added by using explicit jump instructions. ▮

The two shift operators are `<<` for left shift, and `>>` for right shift. Both require two operands: The first is the variable to be shifted, and the second is the integral number of positions by which the first operand is shifted. Left shift always fills the blanks on the right with zeros. Right shift of an unsigned quantity is a logical shift, where the blanks on the left are filled with zeros. Right shift of a signed quantity is an arithmetic shift, where the blanks on the left are filled with sign bit (the original most significant bit) of the first operand. An important consequence of this is that left shifts are equivalent to multiplication by a power of two, and right shifts are equivalent to integer division by a power of two, even when the first operand is negative.

EXAMPLE 5.17 Consider two `char` variables `c1` and `c2` containing, respectively, the values $13_{10} = 00001101_2$ and $-16_{10} = 11110000_2$. Then `c1` $\ll 2 = 00110100_2 = 52_{10}$, and `c2` $\gg 2 = 11111100_2 = -4_{10}$. Remember, we use two's complement arithmetic. Therefore, to negate a number, calculate one's complement and add 1. To confirm the bit pattern `11111100` is really -4, calculate $\sim 11111100_2 + 1$, which yields $00000011_2 + 1 = 00000100_2 = 4_{10} = -(-4)_{10}$. ▮

LISTING 5.1: Definition of identifier properties type structure

```
typedef struct {
    ValType       type;    /* variable type or function *
                            * return type              */
    unsigned int  offset;  /* local variable offset for *
                            * code generation          */
    unsigned int  nparams; /* number of parameters;     *
                            * 0 for variables          */
    ValType      *params;  /* array of parameter types; *
                            * NULL for vars            */
} IDprop;
```

USING BIT FIELDS Assume we are working with a bit field **b** and a bit mask 2^k. Since, for a bit b, b & 1 = b and b & 0 = 0, computing **b** & 2^k results in “masking off” all bits in **b**, except the kth bit, which is left unchanged. Since b | 1 = 1 and b | 0 = b, computing **b** | 2^k results in “masking on” the kth bit, regardless of whether it it was 1 before, and leaving the other bits unchanged. Since b ^ 1 = ~b and b ^ 0 = b, the effect of **b** ^ 2^k results “flipping” the kth bit, and leaving the other bits unchanged.

Setting up the bit mask appropriately, we can isolate more than one bit at the same time. Also, bitwise and logical operators can be freely mixed—*but we have to keep the operator precedence table in mind.*

EXAMPLE 5.18 To check whether a `ValType` variable `t` is boolean array in SIMPL, the following C expression must be true:

```
t & (TYPE_BOOLEAN | TYPE_ARRAY) == (TYPE_BOOLEAN | TYPE_ARRAY)
```

This is actually tantamount to testing a whether boolean variable, say, `is_valid`, is true by writing `is_valid == TRUE` instead of just writing `is_valid`. Therefore, the type expression above should just be tested as `t & (TYPE_BOOLEAN | TYPE_ARRAY)`. ▮

The file `valtypes.h` contains a number of type querying and setting macros, which you must complete and use when adding type checking to the parser.

5.2.5 Handling identifiers

We handle identifiers by looking up their properties in the symbol table. The lookup returns the property structure given in LISTING 5.1.

1. The `type` field stores the type of the identifier.

2. The `offset` field is used during code generation, when it is used as an index into the array of local variables for a method in the JVM. For now, suffice it so say that when an identifier is inserted, a global counter in the symbol table is incremented, and this counter is used as offset value.

3. The `nparams` field stores the number of parameters if the associated identifier is a function or a procedure.

4. If the associated identifier is a function or procedure, the `param` field must store an array of the parameter types. If the associated identifier is a variable, or if it is a function or procedure with no parameters, this field must be set to `NULL`.

ARRAYS AND INDICES We must distinguish between using an array in a scalar context, or referring to the array itself. For a reference to an entire array, the only operations allowed are assigning an existing array to an array variable and the array creation option in the ⟨*assign*⟩ production. All other statements must operate on an array element, which means that the array name must be indexed. In particular, it is impossible in SIMPL to read into or write out an array without treating each element separately by index. Note that assignment of an unindexed array (reference) to a variable will eventually create an alias in the JVM.

EXAMPLE 5.19 To declare and initialise an integer array of size 10, the following SIMPL code should be used:

```
integer array nums;
nums <- array 10;
```

(Since the last statement ends on a semicolon, at least one other statement should follow.) Just like Java, SIMPL allows the dynamic creation of arrays, which is to say, the ⟨*simple*⟩ expression after the "`array`" keyword can be an integer expression, that value of which is computed at runtime. Before an array can be used—accessed by index for value insertion or retrieval—it must first be initialised, but this is a kind of error our compiler cannot pick up without resorting to additional static analysis, which we do not do. ▮

FUNCTION AND PROCEDURE CALLS Procedure calls **MUST** only originate from the ⟨*name*⟩ production, whereas functions **MUST** only be called from the ⟨*factor*⟩ production. A procedure **MUST** have its return type set to `TYPE_CALLABLE`. A function called from factor **MUST** match the type of the calling context. For example, if a function call appears inside an integer expression, it must return a value of type int.

As is the case with other strongly typed languages, the types of the actual parameter list (the arguments passed to subroutine) **MUST** match the formal parameter types entered into the symbol table when a function is defined. You **MUST** also check that neither too many nor too few arguments are specified when a function or procedure is called.

EXAMPLE 5.20 As always, common sense applies. If, say, the function `inc` has been declared to take one integer parameter and to return this value incremented by one, for example, the SIMPL compiler *must not allow* the factor "`not inc(1)`". ▮

To handle argument lists, the signature for the ⟨*arglist*⟩ parse function must be changed to the following:

LISTING 5.2: Definition of the parameter type structure

```
typedef struct variable_s Variable;
struct variable_s {
    char       *id;   /* variable identifier            */
    ValType     type; /* variable type                  */
    SourcePos   pos;  /* variable position in the source */
    Variable   *next; /* pointer to the next variable    *
                       * in the list                     */
};
```

```
void parse_arglist(char *id, IDprop *prop)
```

The identifier and associated properties of the function or procedure being called must be passed to parse_arglist. The properties are used to check the type of each argument, and the identifier is merely used for reporting errors.

FUNCTION AND PROCEDURE DEFINITIONS Function and procedure definitions deserve a bit more attention and, indeed, necessitates the last change to the signatures of parse functions. The function parsing the ⟨*type*⟩ production must now become:

```
void parse_type(ValType *type)
```

Additionally, this function must be updated to use the bit manipulations of §5.2.4 to build up the appropriate type. The pointer parameter type is used for returning the calculated type to the function caller.

Type information, however, is only one part of the picture. The parse function for ⟨*funcdef*⟩, which handles both function and procedure definitions, and which needs to call parse_type, must associate an identifier with its type, its position in the source file, *and* the types of its parameters. The Variable structure of LISTING 5.2 must be used for this purpose.[*]

The presence of the next field suggests that Variable can be used as a node in a linked list. We use this property to build the parameter sequence in parse_funcdef. However, these sequences require more care: To be able to create a dynamically-allocated array of ValTypes for the params field of the IDprop structure in LISTING 5.1, we must know the number of parameters. This implies first parsing all parameters, and while doing so, *linking all the variable structures for the parameters into one linked list*. The linked list is necessary, because (1) we need to read the entire parameter list from the source file *before* we can allocate space for the params array, and (2) since our symbol table does not support removal or updates, we cannot add something provisionally. Once all parameters have been parsed, and once we have determined whether this definition is for a function or a procedure, we can

[*] Since parameters are essentially local variables with values initialised by the function call mechanism, we use the same structure for handling variable sequences in function definitions and variable declaration sections.

allocate space to the `param` field of the `IDprop` structure and turn the list of parameters into an array of parameter types, associated with the function name.

USE OF THE SYMBOL TABLE Whereas a procedure has no return type—it only has `TYPE_CALLABLE` to show that it can be called—a function does have a return type. The last bit of detail to consider then is how a function's local context is handled. That is, how are the function's parameters and local variables kept separate from the main program? (We do not allow nested functions or global variables, because that would just complicate things even more.)

The idea is then to have two different symbol tables:

1. The global one is initialised when parsing starts, and stores information about the main program variables, as well as about the function signatures and return types.

2. When a function is being parsed, we need another local symbol table that stores not only the function signature, but parameter, return, and local variable information as well.

The `open_function` symbol table function must (1) install the function name (and associated properties) in the top-level symbol table, after which (2) it saves the top-level symbol table, and (3) creates a new local table wherein the function parameters and local variables are stored. A bit of thought will show that function parameter identifiers and local variables have no meaning outside of the function. Indeed, parameters are just local variables that are initialised to the values passed when the function is called. (When parsing a function call, we only want the parameter *types* to match.)

Therefore, whereas the function name itself goes into the top-level symbol table, the parameter and local variables names go into the local function symbol table. This also implies that the `find_name` function, should a function symbol table currently be active, must check for function (but not variable) identifiers in the top-level table as well. If this were not done, recursion, for example, will not be possible. There are no global variables (akin to fields in a Java class), and we have to make certain that a local variable name cannot hide a function name. However, function names are global, and are visible in the second lookup step.

Once we are done parsing the definition for a particular function or procedure, we no longer need the local symbol table: The function or procedure identifier lives in the top-level table anyway, and should we keep the local table around, it would in effect add to the global namespace, and we would not be able to have variables of the same name in different functions. Therefore, once we are done with the local table, we simply remove it and its information, and then we "restore" the saved top-level symbol table.

A VERY IMPORTANT NOTE The hash table implementation **MUST NOT** under any circumstances terminate the program. The hash table **MAY** let the symbol table know that something bad happened, but it is up to the symbol table and the parser to terminate on an error condition. If this requirement seems harsh, think of it this way: How would you like it if something bad happened in Java's `HashMap`, but instead of letting you know via an exception, it just decided to terminate the JVM with a call to `system.exit`?

5.3 TYPE CHECKING ERRORS

The function `check_types`, defined away in the original parser skeleton, **MUST** be used for all simple type checks. In addition to the type found and the expected type, `check_types` also accepts a variable argument list for error reporting. This argument list is simply passed along to a call on `vsnprintf`, so detailed and contextual error messages can be originated by whichever parser function calls `check_types`.

5.3.1 Illegal array operation

If there is an illegal array operation, for example, using an unindexed array variable in an expression, the following error message **MUST** be produced:

```
⟨op⟩␣is␣an␣illegal␣array␣operation
```

Here, ⟨*op*⟩ **MUST** be the string representation, as returned by `get_token_string`, of the offending operator, and the error position **MUST** be the first character of its lexeme. NOTE: An array identifier may appear unindexed on the right-hand side of an assignment operator on two conditions: (1) There is nothing but this identifier on the right-hand side, and (2) the identifier on the left-hand side must refer to a variable of the appropriate array type.

5.3.2 Scalar variable expected

If an attempt is to read directly into an array variable, the following error message **MUST** be produced:

```
expected␣scalar␣variable␣instead␣of␣'⟨id⟩'
```

Here, ⟨*id*⟩ is the offending identifier, and the error position **MUST** be the first character of its lexeme.

5.3.3 Not a function

If a function operation is attempted on a different type of identifier, the following error **MUST** be produced:

```
'⟨id⟩'␣is␣not␣a␣function
```

Here, ⟨*id*⟩ refers to the offending identifier, and the error position **MUST** be the first character of its lexeme at that place in the source file where the function operation is attempted. An example is when an attempt is made to call, from in the ⟨*factor*⟩ production, an identifier that is not a function.

5.3.4 Not a procedure

When an identifier that does not refer to a procedure is followed by an argument list when parsed by the ⟨*name*⟩ production, the following error **MUST** be reproduced:

```
'⟨id⟩'␣is␣not␣a␣procedure
```

Here, ⟨*id*⟩ refers to the offending identifier, and the error position **MUST** be the first character of its lexeme at that place in the source file where the procedure call is attempted.

5.3.5 Not a variable

When an assignment or input operation is attempted on an identifier that is not a (scalar) variable, the following error must be produced:

```
'⟨id⟩'␣is␣not␣a␣variable
```

Here, ⟨*id*⟩ refers to the offending identifier, and the error position **MUST** be the first character of its lexeme at that place in the source file where the operation is attempted.

5.3.6 Not an array

When an array operation, like indexing, is attempted on an identifier that is not an array, the following error **MUST** be produced:

```
'⟨id⟩'␣is␣not␣an␣array
```

Here, ⟨*id*⟩ refers to the offending identifier, and the error position **MUST** be the first character of its lexeme at that place in the source file where the array operation is attempted.

5.3.7 Missing argument list for a function

If a (possibly empty) argument list does not follow the identifier of a function when this function is called, the following error **MUST** be produced:

```
missing␣argument␣list␣for␣function␣'⟨id⟩'
```

Here, ⟨*id*⟩ refers to the identifier of the function that is missing an argument list, and the error position **MUST** be the first character of its lexeme at that place in the source file where the call is attempted.

5.3.8 A function or procedure takes no arguments

If a function or procedure that takes no arguments is called with a non-empty argument list, the following error **MUST** be produced:

```
⟨subkind⟩␣'⟨id⟩'␣takes␣no␣arguments
```

Here, ⟨*subkind*⟩ is either "`function`" or "`procedure`" as is appropriate for ⟨*id*⟩, the identifier of the function or procedure being called. The error position **MUST** be the first character of this identifier at that place in the source file where the call is attempted.

5.3.9 Too few arguments for a function or procedure call

When a function or procedure is called with too few arguments, the following error message **MUST** be produced:

```
too␣few␣arguments␣for␣call␣to␣'⟨fname⟩'
```

Here, ⟨*name*⟩ is the identifier that is the name of the function or procedure for which too few arguments were provided. The error position **MUST** be the first character of the first token that is read from the scanner *after* the available argument tokens have been exhausted.

5.3.10 Too many arguments for a function or procedure call

When a function or procedure is called with too many arguments, the following error message **MUST** be produced:

```
too␣many␣arguments␣for␣call␣to␣'⟨fname⟩'
```

Here, ⟨*name*⟩ is the identifier that is the name of the function or procedure for which too many arguments were provided. The error position **MUST** be the comma token that appears after the last allowed parameter was read.

5.3.11 Procedures may not have exit expressions

If an "`exit`" statement inside a procedure is followed by an expression to be evaluated for return to the caller, the following error message **MUST** be displayed:

```
an␣exit␣expression␣is␣not␣allowed␣for␣a␣procedure
```

The error position **MUST** be the first character of the offending expression.

5.3.12 Functions must have exit expressions

If an "`exit`" statement inside a function is *not* followed by an expression to be evaluated for return to the caller, the following error message **MUST** be displayed:

```
missing␣exit␣expression␣for␣a␣function
```

The error position **MUST** be the first character of "`exit`" lexeme that is not followed by an exit expression.

5.3.13 Unknown identifier

If any identifier has not been defined, the following error **MUST** be produced:

```
unknown␣identifier␣'⟨id⟩'
```

Here, ⟨*id*⟩ refers to the identifier that was not defined, and the error position **MUST** be the first character of the offending identifier. *Note that this error has precedence above other the other "not a(n)" errors.* That is to say, if the identifier in question is written as part of a function call, but the identifier is not in the symbol table, it **MUST** be reported as an unknown identifier, and not as "not a function".

5.3.14 Multiple definitions for an identifier

If an attempt is made to declare the same identifier in the same context again, the following error **MUST** be produced:

```
multiple␣definition␣of␣'⟨id⟩'
```

Here, ⟨*id*⟩ is the offending identifier, and the error position **MUST** be the first character of this identifier where it appears for the *second* time. Note that parameter or variable name **MUST NOT** "hide" the name of any previously defined function or procedure, *including the name of the function or procedure currently being parsed.*

5.3.15 Type mismatch

The function `check_types` **SHOULD** be used to flag a mismatch between what was expected and what actually resulted from the parse. It **SHOULD** also be used to enfore type rules for a particular operator, for example, the operator "`and`" must have two type bool operands. If any context can result in more than one error, the more specific error from the previous subsections **MUST** be preferred. In general, using the following order when checking type should deliver a suitable error message:

1. check for any errors on the identifier, including undefined identifiers, using unindexed arrays in a scalar context, and trying to assign to function or procedure names;
2. check whether an operator has the correct type of operands, for example, addition must have operands of type int; and then
3. check that both operands have the same type.

The last two items have different effects where an operator is allowed for both bool and int values.

If `check_types` flags a mismatch, the following error message **MUST** be produced:

```
incompatible␣types␣(expected␣⟨expected⟩,␣found␣⟨found⟩)␣⟨detail⟩
```

Here, ⟨*found*⟩ is the type that resulted from the parse, and ⟨*expected*⟩ is the type that is expected by the context. The ⟨*detail*⟩ message and error position for each context are given in the paragraphs that follow.

ARRAY INDICES An array index **MUST** be of type int. If it is not, the ⟨*detail*⟩ message **MUST** be:

```
for␣array␣index␣of␣'⟨array␣id⟩'
```

The error position **MUST** be the first character of the ⟨*simple*⟩ expression in the bracket pair that follows ⟨*array id*⟩.

ARRAY CREATION SIZE The size given when an array is created **MUST** be of type int. If it is not, the ⟨*detail*⟩ message **MUST** be:

```
for␣array␣size␣of␣'⟨array␣id⟩'
```

The error position **MUST** be the first character of the ⟨*simple*⟩ expression that follows the "`array`" keyword for creating ⟨*array id*⟩. Note that we cannot conclusively determine whether the size is nonnegative. Negative array size is a run-time error that is handled by the Java virtual machine, and not by the SIMPL compiler.

ASSIGNMENT For an assignment that is not an array creation statement, the type of the ⟨*id*⟩ to the left of the "`<-`" operator **MUST** match that of the expression to the right of "`<-`". If it does not, the ⟨*detail*⟩ message **MUST** be:

```
for␣assignment␣to␣'⟨id⟩'
```

The error position **MUST** be the first non-space character that follows the "`<-`" operator.

If an attempt is made to "assign" a new array to an indexed array variable (on the left-hand side of "`<-`"), the ⟨*detail*⟩ message **MUST** be:

```
for␣allocation␣to␣indexed␣array␣'⟨array␣id⟩'
```

Here, ⟨*array id*⟩ **MUST** be the array identifier, and the error position **MUST** be the first character of the "`array`" lexeme.

EXIT STATEMENTS In a function, the type of the expression after an "`exit`" statement **MUST** match the return type of the function. If it does not, the ⟨*detail*⟩ message **MUST** be:

```
for␣'exit'␣statement
```

PARAMETERS Any argument (alternatively, actual parameter) to the function or procedure ⟨*name*⟩ **MUST** match the type of the parameter (alternatively, formal parameter) at the corresponding position in the function or procedure definition. If it does not, the ⟨*detail*⟩ message **MUST** be:

for␣parameter␣⟨*parameter␣position*⟩␣of␣call␣to␣'⟨*id*⟩'

Here, ⟨*parameter position*⟩ **MUST** be the integral index of the offending parameter in the function or procedure parameter list. In addition, the ⟨*parameter position*⟩ **MUST** be reported counting from offset 1, not offset 0. The error position **MUST** be the first character of the offending argument.

GUARD CONDITION EXPRESSIONS The guard condition expression for any `if` clause, `elsif` clause, or `while` statement **MUST** be of type bool. It if it is not, the ⟨*detail*⟩ message **MUST** be:

for␣'⟨*context*⟩'␣guard

Here, ⟨*context*⟩ **MUST** be either "`if`", "`elsif`", or "`while`" (without quotes) as determined by the context. The error position **MUST** be the first character of the guard expression.

BINARY OPERATORS A relational operator that tests order **MUST** have operands of type int; "and" and "or" take operands of type bool; and all arithmetic operators **MUST** have operands of type int. If a particular operator does not, the ⟨*detail*⟩ message **MUST** be:

for␣operator␣⟨*op*⟩

Here, ⟨*op*⟩ **MUST** be the string representation, as returned by `get_token_string`, of the operator. The error position **MUST** be the first character of the operator.

Whatever the binary operator, its operand types **MUST** match. If they do not, use the ⟨*detail*⟩ message of this subsection.

An important difference between error reporting for binary operators and for assignment is that for a binary operator, the error position is at the binary operator; for assignment, it is on the right-hand side of the "`<-`".

UNARY MINUS The unary minus must be followed by an expression of type int. If it does not, the ⟨*detail*⟩ message **MUST** be:

for␣unary␣minus

The error position **MUST** be the first character of the offending expression. Note that if the expression is an array, it **MUST**, rather, be handled as an illegal array operation with ⟨*token*⟩ set to "unary minus" (without quotes); see §5.3.1.

NEGATION The factor that follows the "`not`" keyword **MUST** be of type bool. If it is not, the ⟨*detail*⟩ message **MUST** be:

```
for␣'not'
```

The error position **MUST** be the first character of the offending factor.

5.4 EXERCISES

EXERCISE 5.1 Now that we have type checking enabled, you should figure out if SIMPL allows "normal" rules of precedence for the boolean operators "`and`" and "`or`". In Python, for example we can write:

```
if i > 0 and j > 0:
    return True
```

One attempt to rewrite this in SIMPL is:

```
if i > 0 and j > 0 then
    leave true
end
```

Will this, however, pass the type checking tests? ▮

EXERCISE 5.2 Are the rules for idf and param entirely correct? For example, what about empty parameter lists? Do we allow them? ▮

EXERCISE 5.3 Often, when encountering multiple definition error, compilers will show the file position of the original definition and the attempted redefinition. Our SIMPL compiler does not. How can we update our data structures to accommodate showing both positions when there is an error? ▮

CHAPTER 6

Code generation for the Java virtual machine

With the symbol table and type checking complete, the addition of code generation should be relatively painless.

6.1 RULES

1. You **MUST** use Jasmin [14] to assemble your bytecode stream to valid JVM class files. To understand how the mechanism behind fork-and-exec works (to spawn a child process), refer to `https://ece.uwaterloo.ca/~dwharder/icsrts/Tutorials/fork_exec/`.

2. You **MUST** place the appropriate `gen*` commands in your parser functions to generate code.

3. Debugging output **MUST NOT** be turned on in your final submission.

4. Your compiler **MUST** be compiled with the provided makefile.

5. The name of the generated class file **MUST** be the identifier specified after the "`program`" keyword in the SIMPL source code file.

6. Any temporary files used during compilation and assembly—for example, the Jasmin assembly language file—**MUST** be unlinked from the file system before the compiler terminates, provided it terminates successfully.

7. Your compiler **MUST** read the location of the Jasmin JAR from an environment variable called `JASMIN_JAR`. If it does not, your submission will not be marked.

TABLE 6.1: Marks allocation for the final submission.

DESCRIPTION	WEIGHT
Test cases: variables, expressions, control flow, input, output	50%
Test cases: functions	25%
Style	25%

6.2 MARKING GUIDELINES

The final compiler will only be tested on valid SIMPL source code files. Also, as part of your last submission form, you have to mark which SIMPL constructs your compiler can handle code generation, and only these will be marked. Therefore, caps will only apply if you claim something is complete, but it breaks. The allocation of marks is given in TABLE 6.1.

Strictly speaking, checking for syntax or type errors will not be evaluated again. This does not, however, mean that you can simply remove the symbol table from the equation. Information associated with identifiers must still be retrieved from the symbol table.

Finally, if your designated SIMPL constructs compile successfully, and in addition, there are no memory leaks, your final marks (including style) for the final submission will be multiplied by 1.1. If you have memory leaks, your marks remain unchanged.

6.3 PLAN OF ACTION

Taking TABLE 6.1 into account, you should complete code generation in the following order:

1. Ensure that you set the target class file name with the appropriate function, and see if you can compile the trivial program; this will be the first test case.
2. Handle output statements. A “Hello, world!” example will be the second test case. Also, if output does not work, no other test cases can be run.
3. Handle mathematical expressions.
4. Handle Boolean expressions.
5. Handle variable creation and assignment, which includes handling arrays.
6. Handle the “`if`”(–“`elsif`”–“`else`”) statement.
7. Handle the “`while`” statement.
8. Handle input.
9. Handle functions and procedures.

6.4 THE JASMIN ASSEMBLER

You **MUST** only use the following bytecodes, as defined in `jvm.h`:

> (1) `aload`, (2) `areturn`, (3) `astore`, (4) `getstatic`, (5) `goto`, (6) `iadd`, (7) `iaload`, (8) `iand`, (9) `iastore`, (10) `idiv`, (11) `ifeq`, (12) `if_icmpeq`, (13) `if_icmpge`, (14) `if_icmpgt`, (15) `if_icmple`, (16) `if_icmplt`, (17) `if_icmpne`, (18) `iload`, (19) `imul`, (20) `ineg`, (21) `invokestatic`, (22) `invokevirtual`, (23) `ior`, (24) `istore`, (25) `isub`, (26) `irem`, (27) `ireturn`, (28) `ixor`, (29) `ldc`, (30) `newarray`, (31) `return`, and (32) `swap`.

Consult the JVM specification [12] for details, and ensure that you know which instructions take their operands only from the stack, and which instructions specify operands as part of the instruction stream. Also look at the Jasmin website [14] for assembler usage notes.

Remember, if you want to have any particular examples of code generation, just write a small Java class, compile it with `javac`, and decompile them with `javap`. You should find the `-c` and `-verbose` flags helpful for decompilation. When comparing your own generated code to that produces for an equivalent Java class, do not worry about optimisation.

6.4.1 Methods

SIMPL has functions and a main program, whereas the JVM supports static and instance methods. Each SIMPL function **MUST** be compiled to a static method in the target class. The main SIMPL program—that is, the SIMPL code that appears between the "`main`" and "`end`" keywords—**MUST** be compiled to the `main` method of the class file.

6.4.2 Variables

All SIMPL variables **MUST** be stored as in a particular Jasmin method's local variable array. Besides, the code provided, you **MUST NOT** use static fields, which is to say, "global variables".

For functions, the offset into the local variable array starts at 0, make sure that your symbol table sets this correctly. If your function has m parameters and n local variables, indices $0, 1, \ldots, m-1$ are used for the parameters—and the JVM will populate them appropriately when the function is called—whereas indices $m, m+1, \ldots, m+n-1$ are used for local variables.

EXAMPLE 6.1 Suppose your are generating code for an SIMPL function that starts as follows:

```
define foo(integer p1, integer p2, boolean p3) -> integer
begin
    integer array v1;
    boolean v2;
```

In the stack frame's local variable array, the parameters and local variables will be packed as:

OFFSET:	0	1	2	3	4
CONTENT:	p1	p2	p3	v1	v2

Note that each parameter or variable occupies exactly one cell.[*] ▮

The "main" SIMPL function must be handled slightly differently: The Java interpreter expects a main method, which must have a Java string array for command-line arguments. Therefore, the "main" SIMPL function, the offset for local variables **MUST** start at 1. You may do this either by modifying the symbol table, or by adding logic to the code generation unit.

6.4.3 Input and output

The SIMPL "output" statement outputs its arguments via Java's `System.out.print` method. This to say, just like in C, you **MUST NOT** have an automatic newline at the end of any "output" output. If a SIMPL programmer wants a newline, it must be written by appending "\n" to the "output" arguments.

For input, SIMPL needs to distinguish between numeric and Boolean literals. It does so by borrowing Sedgewick and Wayne's `readBoolean` and `readInt` methods [17]. They are provided as boiler-plate Jasmin code in the code generation unit.

6.4.4 Jasmin assembler path

To read the path to the Jasmin assembler from an environment veriable, use the `getenv` function from `stdlib.h`. An environment variable is set (and exported to child processes) by doing the following on the command line:

```
$ export JASMIN_JAR=/path/to/jasmin.jar
```

Remember, ***if you do not support this environment variable, your work cannot be marked.***

[*]The JVM allows "wide" data types like `double` and `long` in Java, but SIMPL does not use them.

APPENDIX A

Parsing and Extended Backus–Naur Form

The idea of defining languages with mathematical precision goes back to Noam Chomsky. The language specification formalisms stemming from Chomsky's work were found to be insufficient in representing the complexity of spoken languages; they are, however, quite useful in the specification of computer input and programming languages. Although we do not study compiler design in detail in this course, we do need a way to specify input precisely for the compiler project.

In 1960, Algol60 became the first programming language to be defined formally and precisely. Its syntax was defined with a formalism now known as Backus–Naur Form (BNF). Niklaus Wirth in 1977 postulated some extensions to BNF to express repetitions in grammar structure better. These extensions are known as Extended BNF (EBNF), and we will use this standard to specify input formats. Although EBNF is now an ISO standard, we use the simplified version originally proposed by Wirth [26], and is also known as Wirth Syntax Notation (WSN).

A.1 DEFINING A LANGUAGE WITH EBNF

In a self-referential twist, EBNF is employed to define itself in FIGURE A.1.* Note that EBNF gives only syntax, that is, how the input is formatted, and not semantics, that is, what the input means. An EBNF for a set of grammar rules consists of a number of productions. Each production looks like a mathematical equation: The idea is that the identifiers in EBNF function like variables, and a production specifies how a right-hand side (of the equals sign)

*To be clear: For the sake of simplicity, FIGURE A.1 is not entirely accurate. In the EBNFs we use, for example, many additional characters are allowed in the ⟨*punctuation*⟩ production. Also, the nonterminals have not been delimited by angular brackets.

syntax = {*production*}.

production = *variable* “`=`” *expression* “`.`”.

expression = *term* {“`|`” *term*}.

term = *factor* {*factor*}.

factor = *variable* | *terminal* | “`(`” *expression* “`)`” | “`[`” *expression* “`]`”
| “`{`” *expression* “`}`”.

variable = *letter* {*letter* | *digit*}.

terminal = “`"`” {*letter* | *digit* | *punctuation*} “`"`”.

letter = “`a`” | … | “`z`” | “`A`” | … | “`Z`”.

digit = “`0`” | … | “`9`”.

punctuation = “`=`” | “`.`” | “`|`” | “`(`” | “`)`” | “`[`” | “`]`” | “`{`” | “`}`” | “`"`”.

FIGURE A.1: EBNF for Extended Backus–Naur Form; for simplicity, nonterminals have not been delimited by angular brackets.

may be substituted for the variable on the left. A sequence of such substitutions until no variables remain derives a sentence in the specified language.

In the next list, let α, β, γ, and δ be any grammar symbol, which is to say, a terminal or variable. Then, to interpret EBNF, note the following:

1. An **variable** (identifier), also called a **nonterminal**, is a sequence of alphanumeric characters that starts with a letter. They function as variables that will be substituted by the right-hand side of a production.

2. A **terminal** (string) is a sequence of alphanumeric characters that are delimited by double quotation marks. They are terminal values in that they cannot be substituted by another symbol—as such, these terminals appear as literal strings in the input.

3. A choice is expressed by a vertical bar; it is interpreted as a logical (exclusive) OR. For example, $\alpha = \beta \mid \gamma$ means that α may be replaced by either β or γ.

4. Optionality is expressed in two ways:

   (a) Square brackets mean zero or one occurrences. Thus, a sequence of symbols delimited by brackets may either be present in the input once, or absent entirely. For example, $\alpha = \beta[\gamma]\delta$ means that α may be replaced by either $\beta\gamma\delta$ or $\beta\delta$.

   (b) Braces (curly brackets) mean zero or more occurrences. Thus, a sequence of symbols delimited by braces may be present any number of times, or not at all. For example, $\alpha = \beta\{\gamma\}\delta$ may be replaced by either $\beta\delta$, or $\beta\gamma\delta$, $\beta\gamma\gamma\delta$, $\beta\gamma\gamma\gamma\delta$, and so on.

5. Note that terms may be grouped by parentheses. This works like a normal mathematical equation. Here the bar functions like and has the precedence of a plus sign. There is no operator akin to the times sign—according to the ⟨*term*⟩ rule, other ⟨*factors*⟩ may follow the first factor arbitrarily, and this concatenation is like an implies times sign or logical AND. (If it helps, the algebraically inclined may imagine a concatenation operator between factors.) For example, $\alpha = (\beta \mid \gamma)\delta$ may be replaced by either $\beta\delta$ or $\gamma\delta$.

A.2 DERIVING A PARSER FROM AN EBNF

The EBNF specification in this course may be parsed with a simple one-symbol lookahead predictive parser. This is to say that we may use the method of recursive-descent, as is described in the course slides. Following Wirth's suggestions [25], we may derive our parsers directly from the EBNF. We typically split the parsing action into two components:

1. the **scanner** or lexical analyser, which reads the input and builds tokens, i.e. the "words" in the language; and
2. the **parser** or syntactic analyser, which takes the tokens requested from the scanner and ensures that they form valid sentences for our input language.

Note that the tokens returned by the scanner correspond to the terminals in the language specification. This makes intuitive sense, since for input languages, we expect to have well-formed sentences, built from scanned literals, as input. It is also interesting to note in passing that your garden variety scanner works over a regular language, and therefore, can be modelled by a finite automaton. However, parsers analysing Context-Free Grammars (CFGs), which allow recursion in the productions, cannot be modelled by finite automata; although we could use a push-down automaton, the method presented here works directly over the parse tree obtained from the input.

To construct a parser by hand, we associate with each grammar construct K a program fragment $\mathrm{Fr}(K)$ as in TABLE A.1. Here, `token` is a global variable holding the token last read from the input, while NEXT() and ERROR() are routines to read the next token into `token` and report an error, respectively. The set $\mathrm{First}(K)$ contains all the tokens with which a sentence derived from construct K may start. Note that $\mathrm{First}(K)$ may be a singleton set.

If we study the last K in TABLE A.1, an interesting question appears: What should happen if both `token` $\in \mathrm{First}(t_i)$ and `token` $\in \mathrm{First}(t_j)$ for some $i \neq j$, where $0 \leq i, j \leq n$? If more than one such choice derive a valid sentence, the grammar is **ambiguous**. In TABLE A.1, the program fragment associated with whichever of i or j is first in the **if** statement will be executed; but is this necessarily what we want?

Since computers are notoriously silly about intuiting what we as humans want from them, ambiguity in which production to use can only complicate things—unless we give a way of resolving the ambiguity, or better yet, disallow ambiguities entirely. For the sake of simplicity, we opt for the latter. We do so by giving constraints to which certain constructs K must

TABLE A.1: Grammar constructs and their associated program fragments.

K	**Fr**(K)
"x"	**if** `token` = "x" **then** NEXT() **else** ERROR() **end if**
(x)	Fr(x)
$[x]$	**if** `token` $\in$ First(x) **then** Fr(x) **end if**
$\{x\}$	**while** `token` $\in$ First(x) **do** Fr(x) **end while**
$f_0 f_1 \ldots f_n$	Fr(f_0) Fr(f_1) ... Fr(f_n)

$t_0 \mid t_1 \mid \ldots \mid t_n$	**if** `token` $\in$ First(t_0) **then** Fr(t_0)
	else if `token` $\in$ First(t_1) **then** Fr(t_1)
	...
	else if `token` $\in$ First(t_n) **then** Fr(t_n)
	else ERROR()
	end if

adhere; these constraints are given in TABLE A.2. The set Follow(K) contains all the symbols that may follow K, where ε denotes the empty string. The EBNF specification for the compiler project adheres to the constraints of TABLE A.2.

The final idea we need to construct a recursive-descent parser concerns how we construct the program fragments for nonterminals according to the rules in TABLE A.1. Wirth gives the following rule [25]:

> A parsing algorithm is derived for each nonterminal symbol, and it is formulated as a procedure* carrying the name of the symbol. The occurrence of the symbol in the syntax is translated into a call of the corresponding procedure.

Note that this rule holds regardless of whether the procedure is recursive or not, and that we exclude those productions that are handled by the scanner. Put another way, such a procedure mimics the right side of a production according to the constructs in TABLE A.1.

Essentially, what we are doing then, is traversing the parse tree of our input. A **parse tree** gives the structure of a particular program in a language. Every node is labelled by a grammar symbol: Internal nodes are labelled by nonterminals, while leaves are labelled by terminals. Then any internal node and its children correspond to a production, where the internal node itself corresponds to the left-hand side of the production, and its children to the right-hand side.

*Here, a **procedure** is a specific kind of subroutine—one that does not have a return value, and typically operate by causing side effects. The languages for which Wirth is famous, such as Pascal and Oberon, all distinguish between with functions (with return values) and procedures (without return values, like `void` functions in C). A **side effect** is when a subroutine causes some observable change in program state outside the subroutine or when an operator modifies its operand(s). For example, in C the expression `x + 1` has no side effects (the value of `x` remains unchanged), but `x++` has the side effect of adding one to the value stored in `x`.

TABLE A.2: Ambiguity constraints over some grammar constructs.

K	**Constr**(K)
$t_0 \mid t_1$	$\text{First}(t_0) \cap \text{First}(t_1) = \varnothing$
$f_0 f_1$	If $\varepsilon \in f_0$, then $\text{First}(f_0) \cap \text{First}(f_1) = \varnothing$
$[x]$ or $\{x\}$	$\text{First}(x) \cap \text{Follow}(K) = \varnothing$

$$
\begin{aligned}
\langle expr\rangle &= \langle term\rangle\ \{(\text{“+”} \mid \text{“-”})\ \langle term\rangle\}.\\
\langle term\rangle &= \langle factor\rangle\ \{(\text{“*”} \mid \text{“/”})\ \langle factor\rangle\}.\\
\langle factor\rangle &= \langle number\rangle \mid \langle id\rangle \mid \text{“(”}\ \langle expr\rangle\ \text{“)”}.\\
\langle number\rangle &= \langle digit\rangle\ \{\langle digit\rangle\}.\\
\langle id\rangle &= \langle letter\rangle\ \{\langle letter\rangle \mid \langle digit\rangle\}.\\
\langle digit\rangle &= \text{“0”} \mid \ldots \mid \text{“9”}.\\
\langle letter\rangle &= \text{“a”} \mid \ldots \mid \text{“z”}.
\end{aligned}
$$

FIGURE A.2: EBNF for arithmetic expressions.

EXAMPLE A.1 Suppose we wish to write a parser for arithmetic expressions over numbers and identifiers, specified as an EBNF in FIGURE A.2. The question is now which productions are handled by the scanner, and which by the parser. Does the scanner, for example, return a unique token for every letter and digit, or does it treat ⟨*number*⟩ and ⟨*id*⟩ as token types?

The answer follows from noting that (1) the scanner constructs tokens from their respective **lexemes**—the actual character sequences that constitute a token and are read from the input—and (2) the productions for ⟨*number*⟩ and ⟨*id*⟩ are **regular**, that is, they do not derive "sentences" containing themselves (which would have required a recursive parsing approach). Since "`(`", "`)`", "`+`", "`-`", "`*`", and "`/`" appear as terminals in the first three productions, it is clear that they are tokens, and that each of these tokens corresponds to one lexeme only. Next note that digits and letters on their own have no syntactic meaning—we are, rather, interested in the syntactic classes ⟨*number*⟩ and ⟨*id*⟩, and wish *them* to be returned as tokens. Here, then, we have tokens where each token corresponds to more than one lexeme. For example, "`var1`", "`x`", and "`length`" are different lexemes for an ⟨*id*⟩ token.

The pseudocode for the NEXT() routine is given in FIGURE A.3. Assume that GETCHAR() reads a single character from the input stream, and that INTVAL(`ch`) returns the numeric value of the character stored in `ch`. Also assume that `token`, `num`, and `id` are global variables. If the token is a ⟨*number*⟩, then the numeric value of the token is given in `num`; if the token is an ⟨*id*⟩, then the token lexeme is given in `id`. The variable `token` will typically be implemented as an enumerated type.

Finally, the parsing routines in FIGURE A.4 are derived according to TABLE A.1 and

Wirth's rule for a parsing algorithm. Consider the parse tree for `13 * (v1 + v2)`, given in FIGURE A.5. Convince yourself that this tree illustrates both how the parsing routines of FIGURE A.4 are called, as well as that every internal node and its children correspond to a production in the EBNF of FIGURE A.2. Also make sure that you understand why and where the parsing of an invalid sentence such as `13 * -(v1 + v2)` will fail. ❚

A.3 FINAL REMARKS

A final word on parsers are now in order. When writing a large-scale, complex compiler, we do not construct either the scanner or parser by hand. For scanners, we use tools such as `lex` or its open-source incarnation `flex`. They use regular expressions to build scanning routines—in essence, glorified finite automata—automatically. Similarly, we use tools such as `yacc` ("Yet Another Compiler Compiler"), the open-source `bison` (you had to see this name coming), or `javacc` for Java. The first two may interface with `lex` or `flex`, so that to handle parsing, we write a token specification as regular expressions (like those employed by Vim) and a CFG (passed to `yacc` or `bison`).

Note that the class of grammars we can parse with a typical compiler compiler is larger than that which we have treated here. The class of LR(1) grammars handled by these programs is a proper superset of the LL(1) grammar we use in this course.[*] As a matter of fact, LR(1) parsers can be constructed for virtually all programming language constructs for which CGFs can be written. Additionally, with an LR(1) parser it is possible to detect syntactic errors as soon as possible. Coupled with error-recovery markers so that we can find as many errors as possible in a single pass, LR(1) parsers are much more efficient than a simple recursive-descent parser. It is, however, possible to structure the latter so that we do not stop after the first error is encountered.

[*]The first "L" is for left-to-right scanning of the input, the second "L" or "R" is for using, respectively, leftmost or rightmost derivations, while a number in parentheses indicates the number of lookahead input symbols that are used.

```
procedure NEXT()
▷ Before this procedure is called the first time, ch ← GETCHAR() must be executed once
during initialisation.
    num ← 0
    clear string buffer id
    while ch is a space character do
        ch ← GETCHAR()
    end while
    if ch = "(" then
        token ← lparen; ch ← GETCHAR()
    else if ch = ")" then
        token ← rparen; ch ← GETCHAR()
    else if ch = "+" then
        token ← plus; ch ← GETCHAR()
    else if ch = "-" then
        token ← minus; ch ← GETCHAR()
    else if ch = "*" then
        token ← times; ch ← GETCHAR()
    else if ch = "/" then
        token ← div; ch ← GETCHAR()
    else if ch is a digit then
        repeat
            num ← num × 10 + INTVAL(ch)
            ch ← GETCHAR()
        until ch is not a digit
        token ← number
    else if ch is a letter then
        repeat
            append ch to end of string buffer id
            ch ← GETCHAR()
        until ch is not an alphanumeric character
        token ← id
    else
        report error: "invalid character"
    end if
end procedure
```

FIGURE A.3: A scanner for the EBNF in FIGURE A.2.

```
procedure EXPR()
▷ Before this procedure is called the first time, NEXT() must be called once to ensure
that the lookahead symbol is available.
    TERM()
    while token = $plus$ ∨ token = $minus$ do
        NEXT()
        TERM()
    end while
end procedure

procedure TERM()
    FACTOR()
    while token = $times$ ∨ token = $div$ do
        NEXT()
        FACTOR()
    end while
end procedure

procedure FACTOR()
    if token = $number$ then
        NEXT()
    else if token = $id$ then
        NEXT()
    else if token = $lparen$ then
        NEXT()
        EXPR()
        if token = $rparen$ then
            NEXT()
        else
            report error: “right parenthesis expected”
        end if
    else
        report error: “number, id, or left parenthesis expected”
    end if
end procedure
```

FIGURE A.4: A parser for the EBNF in FIGURE A.2.

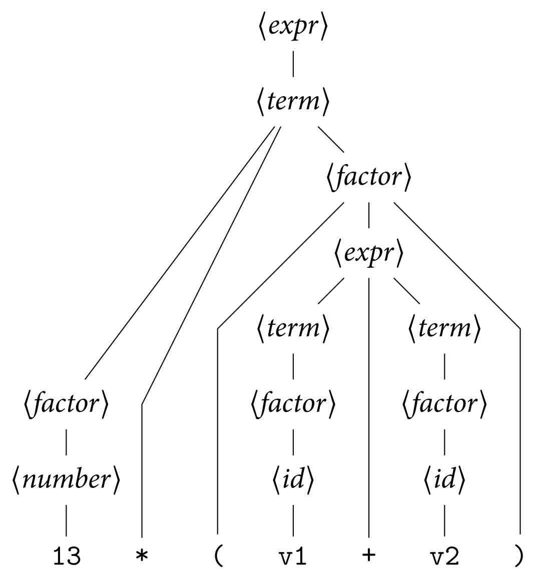


FIGURE A.5: Parse tree for `13 * (v1 + v2)`.

APPENDIX B

Converting EBNF to BNF

For certain activities, such as writing a bottom-up parser, the older BNF grammar format is better, or at least, better known. It is also easier to write attributed grammars, where we add information on semantics, in BNF. In particular, we are interested in adding type and evaluation rules to our grammar.

BNF only has the "operator" for choice, the vertical bar; there are no constructs for expressing optionality, repetition, or grouping. BNF grammars, therefore, must show these by the use of recursion in the productions. However, it is very easy to convert an EBNF grammar to a BNF grammar. Simply do the following:

1. Replace each option $[\alpha]$ by a new nonterminal A, and add a new production

   $$A = \alpha \mid \varepsilon,$$

   where ε denotes the empty string. Essentially, the choice between having α and not having α is moved to the new production for A. This is equivalent to saying we convert $B = \beta[\gamma]\delta$ to

   $$B = \beta\gamma\delta \mid \beta\gamma.$$

2. Replace each repetition $\{\alpha\}$ by a new nonterminal A, and add a new production

   $$A = A\alpha \mid \varepsilon.$$

   Here, recurring on A allows α to be repeated, and the ε allows the recursion to stop (and α to be absent).

3. Replace each group (α) by a new nonterminal A, and add a new production

   $$A = \alpha.$$

Bibliography

[1] Alfred V. Aho, Monica A. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Second. Reading, Mass.: Addison-Wesley, 2007.

[2] S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. RFC 2119. RFC Editor, 1997-03. 3 pp. ISRN: 2070-1721. URL: `https://www.ietf.org/rfc/rfc2119.txt`.

[3] Scott Chacon. *GitHub flow*. 2011-08-31. URL: `http://scottchacon.com/2011/08/31/github-flow.html`.

[4] Scott Chacon and Ben Straub. *Pro Git*. Second. New York, N.Y.: Apress, 2014. URL: `https://git-scm.com/book/en/v2`.

[5] *Dein.vim*. URL: `https://github.com/Shougo/dein.vim` (visited on 2019-08-05).

[6] Vincent Driessen. *A successful Git branching model*. 2010-01-05. URL: `http://nvie.com/posts/a-successful-git-branching-model/`.

[7] GitHub. *A collection of gitignore templates*. GitHub repository. URL: `https://github.com/github/gitignore` (visited on 2020-08-10).

[8] GitHub. *Ignoring files*. GitHub repository. URL: `https://docs.github.com/en/github/using-git/ignoring-files` (visited on 2020-08-10).

[9] *GNU Make: A Program for Directing Recompilation*. Free Software Foundation, 2016. ISBN: 1-882114-83-3. URL: `https://www.gnu.org/software/make/manual/` (visited on 2020-08-10).

[10] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification: Java SE 8 Edition*. Redwood City, Calif.: Oracle, 2014. URL: `https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf`.

[11] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Englewood Cliffs, New Jersey, USA. Prentice Hall, 1988.

[12] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification: Java SE 8 Edition*. Redwood City, Calif.: Oracle, 2014. URL: `https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf`.

[13] Steve Losh. *Learn Vimscript the Hard Way*. First edition. Leanpub, 2013-04-04. URL: `http://learnvimscriptthehardway.stevelosh.com/` (visited on 2020-08-10).

[14] Jonathan Meyer, Daniel Reynaud, and Iouri Kharon. *Jasmin. An assembler for the Java Virtual Machine*. URL: `http://jasmin.sourceforge.net/` (visited on 2020-08-10).

[15] *NFS Documentation. D2: What is a "silly rename"?* URL: `http://nfs.sourceforge.net/#faq_d2` (visited on 2020-08-10).

[16] Adam Ruka. *GitFlow considered harmful.* 2015-05-03. URL: `http://endoflineblog.com/gitflow-considered-harmful`.

[17] Robert Sedgewick and Kevin Wayne. *Algorithms.* Fourth. Upper Saddle River, N.J.: Pearson Education, 2011. ISBN: 9780132762564.

[18] Sytse Sijbrandij. *GitLab Flow.* 2014-09-29. URL: `https://about.gitlab.com/2014/09/29/gitlab-flow/`.

[19] Michael Sipser. *Introduction to the Theory of Computation.* Third edition. Boston, Mass.: Cengage Learning, 2012.

[20] Andrew S. Tanenbaum and Todd Austin. *Structured Computer Organization.* Sixth (international) edition. Boston, MA: Pearson, 2013.

[21] *vim-plug.* URL: `https://github.com/junegunn/vim-plug` (visited on 2020-08-10).

[22] Wikipedia. *Bit field.* Wikipedia, The Free Encyclopedia. 2004. URL: `https://en.wikipedia.org/wiki/Bit_field` (visited on 2020-08-10).

[23] Wikipedia. *Mask (computing).* Wikipedia, The Free Encyclopedia. 2004. URL: `https://en.wikipedia.org/wiki/Mask_(computing)` (visited on 2020-08-10).

[24] Wikipedia. *Shebang (Unix).* 2004. URL: `https://en.wikipedia.org/wiki/Shebang_(Unix)` (visited on 2020-08-10).

[25] Niklaus Wirth. *Compiler Construction.* Harlow, U.K.: Addison Wesley Longman, 1996.

[26] Niklaus Wirth. "What can we do about the unnecessary diversity of notation for syntactic definitions?" In: *Communications of the ACM* 20.11 (1977-11), pp. 822–823. ISSN: 0001-0782. DOI: `10.1145/359863.359883`. URL: `http://doi.acm.org/10.1145/359863.359883`.