Scil Version 1.0.2 Documentation

Kim Skak Larsen

October 11, 2019

Abstract

This documents the Scil compiler implementation meant for teaching purposes, and the name is an acronym for "A simple compiler in a learning environment". The goal of the implementation is to illustrate important compiler techniques in a simple setting and to enable the students to make minor adjustments and extensions. The source language is a simple imperative language, with integers being the only type, but including expressions, assignments, control structures, and function definitions and calls, including recursion and static nested scope. The target language is 64 bit X86 Assembly/GAS Syntax. The discussions here detail the language and the use of this software in a linux environment. Note that focus is on clarity in the compiler code as well as in the generated assembler code and not on efficiency or optimizations.

Introduction

This documentation is very brief and will likely be extended. It can only be understood fully with some knowledge of standard programming languages and the workings of a compiler, corresponding to an introductory undergraduate course on the topic.

Development

The Scil compiler is written in Python, developed using Python 3.6.7, and the ply package version 3.11.

Attempts were made to follow the PEP 8 style guide for python code and verify this via flake8, using the command

```
python3 -m flake8 --exclude parsetab.py *.py
```

parsetab.py is generated by the ply package and does not conform to PEP 8. Due to conditions in the same package, a few lines in the file **lexer_parser.py** are longer than recommended.

Use

On command line, the compiler can be run as

```
./compiler.py < test_file.src
```

Or prefixed with python (python3) if the file `compiler.py` does not have execution permissions.

Output is sent to **stdout**. Thus to run the compiled programs, one possible command sequence is:

```
./compiler < test_file.src > assembler_file.s
gcc -no-pie assembler_file.s
./a.out
```

If the input program is bugged, the first detected error is reported and compilation is terminated.

A small collection of test programs are available in the test directory.

Language

Scil is a very simple imperative programming language designed for teaching. It has only one native type, integer, so all variables are of that type. Comparison operators are included, so Boolean expressions in a limited form are available in if-then-else-statements (else cannot be omitted) and while-statements. Additionally, the language includes basic arithmetic, assignment, a print statement (no input), and, most importantly, function definitions and calls. It supports static nested scope. Compound statements are surrounded by curly brackets (C-style), but these do not introduce a new scope.

The language is partially defined by the grammar of Fig. 1.

⟨program⟩ : ⟨body⟩
⟨body⟩ : ⟨optional_variables_declaration_list⟩
⟨optional_functions_declaration_list⟩
⟨statement_list⟩
⟨optional_variables_declaration_list⟩ : ε
| ⟨variables_declaration_list⟩
⟨variables_declaration_list⟩ : **var** ⟨variables_list⟩
| **var** ⟨variables_list⟩ ⟨variables_declaration_list⟩
⟨variables_list⟩ : **ident**
| **ident ,** ⟨variables_list⟩
⟨optional_functions_declaration_list⟩ : ε
| ⟨functions_declaration_list⟩
⟨functions_declaration_list⟩ : ⟨function⟩
| ⟨function⟩ ⟨functions_declaration_list⟩
⟨function⟩ : **function ident (** ⟨optional_parameter_list⟩ **) {** ⟨body⟩ **}**
⟨optional_parameter_list⟩ : ε
| ⟨parameter_list⟩
⟨parameter_list⟩ : **ident**
| **ident ,** ⟨parameter_list⟩
⟨statement⟩ : **return** ⟨expression⟩ **;**
| **print** ⟨expression⟩ **;**
| **ident =** ⟨expression⟩ **;**
| **if** ⟨expression⟩ **then** ⟨statement⟩ **else** ⟨statement⟩
| **while** ⟨expression⟩ **do** ⟨statement⟩
| **{** ⟨statement_list⟩ **}**
⟨statement_list⟩ : ⟨statement⟩
| ⟨statement⟩ ⟨statement_list⟩
⟨expression⟩ : **integer**
| **ident**
| **ident (** ⟨optional_expression_list⟩ **)**
| ⟨expression⟩ **bin_op** ⟨expression⟩
| **(** ⟨expression⟩ **)**
⟨optional_expression_list⟩ : ε
| ⟨expression_list⟩
⟨expression_list⟩ : ⟨expression⟩
| ⟨expression⟩ **,** ⟨expression_list⟩

Figure 1: The grammar defining Scil.

In the grammar, **ident** and **integer** are written as terminal symbols, but they represent usual identifiers and non-negative integers. **bin_op** can be any of the standard four arithmetic operations (division is integer division) or any of the six comparison operators (C-style). There is no unary minus.

A type checking phase ensures that programs are statically type correct before target code is generated.

It is a requirement that a **return** statement is the last statement executed in any scope (⟨body⟩); this includes the main scope, which should normally return zero.

Phases

The phases of the compiler are listed in Fig. 2.

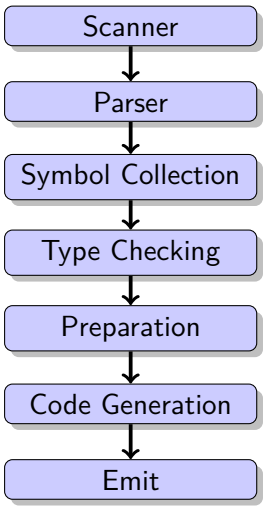

Figure 2: The phases of Scil.

The scanner and parser phases implement lexical and syntax analysis and are tightly coupled due to the use of the ply package. The phase returns an abstract syntax tree (AST).

The symbol collection phase collects all identifiers from the AST, adding them to a symbol table, organized in units corresponding to the scopes of the user program. The phase registers the placement of variables and formal

parameters in sequences for later use in connection with offsets in the code generation phase.

Using the AST and the symbol table, the type checking phase checks that the program is statically correct.

The preparation phase traverses the AST and equips functions with labels for the subsequent code generation. This is necessary if functions are defined in an order different from bottom-up with regards to their use. Thus, without this, mutual recursion would not be possible (or some patching-up would be required in the code generation phase).

The code generation phase generates the assembler code from the AST, using a few meta-instructions that indicate caller/callee code blocks, etc.

The emit phase outputs the finished assembler.

Contact

Reports of errors or suggestions for improvements are received with gratitude. Please contact the author in person or by sending an email to `kslarsen@imada.sdu.dk`.