A Guide to the Rochester PL/0 Compiler [*]

Michael L. Scott

Computer Science Department
University of Rochester

Revised August 2004

Rochester's PL/0 compiler attempts to illustrate the characteristics of a "real" compiler on a modest scale. It runs on Unix systems and generates MIPS I assembler for the `spim` simulator. (An alternative code generator, for the x86, should be ready in time for this fall's code generation assignment.) Unlike many pedagogical compilers, it has multiple passes, employs a table-driven scanner and parser, performs high-quality syntax error repair, and issues well-formatted diagnostic messages. It does not perform code improvement, though this may be added at some point in the future. For now, it generates very poor code.

1 The PL/0 Language

The PL/0 compiler is small (about 5400 lines of C++ and about 400 lines of grammar source) mainly because it compiles a very small input language. PL/0 was defined by Niklaus Wirth (the inventor of Pascal and Modula) as part of an extended example in his 1976 book, *Algorithms + Data Structures = Languages* (pp. 307–347). It is essentially a subset of Pascal, with nested procedures, but without parameters; functions; `case`, `for`, or `repeat` statements; `else` clauses; or any types other than `integer`. The reference version of the PL/0 compiler implements the language exactly as defined by Wirth, with four extensions: comments, primitive numeric I/O, output of constant strings, and value/result parameters. Students may be given a version of the compiler that implements less than this.

1.1 Tokens

Case is insignificant in PL/0. The PL/0 tokens are:

- keywords
  `begin`, `call`, `const`, `do`, `end`, `if`, `in`, `odd`, `out`, `procedure`, `then`, `var`, `while`.
- identifiers
  Any other string of letters and digits beginning with a letter.
- strings
  As in C: delimited by double quotes. May contain quotes, backslashes, or newlines only

[*]Copyright © 1994–2004 Michael L. Scott

if escaped by a backslash. Backslashes also introduce escape sequences for nonprinting characters.

- numbers
  A string of (decimal) digits.

- operators, relations, and punctuation marks
  `+`, `-`, `*`, `/`, `=`, `#`, `<`, `>`, `<=`, `>=`, `(`, `)`, `,`, `;`, `.`, `:=`.

Comments are also as in C: bracketed by `/*` and `*/`, or by `//` and end-of-line. Comments may not nest, but either kind of comment can be used to "comment out" the other.

1.2 Syntax

In the EBNF below, `typewriter font` is used for tokens and *italicized font* is used for non-terminals. Bold parentheses are used for grouping. The vertical bar is used for alternation ("or"). Bold curly brackets indicate an optional item. A Kleene star (*) indicates that the preceding item can be repeated zero or more times. Epsilon (ϵ) denotes the empty string. White space (spaces, tabs, and newlines) is required in PL/0 programs to separate characters that could otherwise be considered part of the same token. Otherwise, input is "free format": indentation and additional white space are irrelevant.

```
program    ⟶  block .
block  ⟶  { CONST identifier = number ( , identifier = number )* ; }
          { VAR identifier ( , identifier )* ; }
          ( PROCEDURE identifier { ( param_list ) } ; block ; )*
          statement
param_list  ⟶  mode identifier ( , mode identifier )*
mode   ⟶  in | out | in out | ϵ
statement  ⟶  identifier := expression
             | identifier := string
             | CALL identifier { ( expr_list ) }
             | BEGIN statement ( ; statement )* END
             | IF condition THEN statement
             | WHILE condition DO statement
             | ϵ
expression  ⟶  fragment ( ( + | - | * | / ) fragment )*
fragment  ⟶  identifier | number | ( + | - ) fragment | ( expression )
condition  ⟶  ODD expression
             | expression ( = | # | < | > | <= | >= ) expression
expr_list  ⟶  expression ( , expression )*
```

1.3 Semantics

PL/0 is simple enough that semantic issues generally amount to common sense. The checks performed by the PL/0 compiler are listed in section 4.5 below. Constants, variables, and procedures declared within a block are local to that block, and cannot be referenced outside it. Procedure paremeters, likewise, are local to the block that comprises the procedure's body.

Parameters are passed by value/result. `In` parameters are copied into the procedure at call time. `Out` parameters are copied back to the caller when the procedure returns. `In out` parameters are copied at call time *and* when the procedure returns. Arguments passed as `out` or `in out` parameters must be l-values. A parameter with no specified mode is implicitly passed by value (`in`).

The arithmetic operators associate left-to-right. Precedence is as follows: unary `+` and `-` group most tightly, followed by binary `*` and `/`, and then binary `+` and `-`. Conditions (comparisons) are above the level of expressions in the grammar, effectively giving them the lowest precedence of all. The symbol `#` means "not equal".

Two special "variables" are pre-defined: The variable `input`, when read, yields the next integer from the standard input stream. The variable `output`, when assigned an integer expression, prints the value of that expression (with trailing linefeed) on the standard output stream. One can also assign a string to `output`; it is printed *without* appending a linefeed. (NB: when running under `spim`, output is not flushed until a linefeed appears.)

2 Tools

Construction and operation of the PL/0 compiler are assisted by a number of tools, as illustrated in figure 1.

2.1 Scanner and Parser Generators

`Scangen` and `fmq` are scanner and parser generators, respectively. They were written by students of Charles Fischer at the University of Wisconsin around about 1980, and are described in detail in appendices B, C, and E of Fischer and LeBlanc's book, *Crafting a Compiler*.[1] `Scangen` takes as input a set of regular expressions, and produces DFA tables (file `tables`) to drive a scanner for the tokens described by the regular expressions. `Fmq` takes as input a context free grammar, and produces tables (file `ptablebin`) to drive an LL(1) parser for that grammar. `Fmq` takes its name from the Fischer/Milton/Quiring syntax error repair algorithm, for which it also generates tables (file `etablebin`).

Both `scangen` and `fmq` produce numeric tables, in contrast to the C code produced by Unix's standard `lex` and `yacc`, or the similar Gnu `flex` and `bison`. (`Scangen` produces tables in ASCII. `Fmq` can produce tables in either ASCII or binary; the PL/0 compiler uses the binary option.) These numeric tables are turned into initialized data structures by the tools `makescan` and `makeparse`. The C++ output files, `scantab.cc` and `parsetab.cc` are then compiled by the GNU C++ compiler, `g++`, along with the other C++ source files. The files `scantab.h` and `parsetab.h` contain definitions for variables exported by `scantab.cc` and `parsetab.cc` respectively.

2.2 Mungegrammar

Several source files for the PL/0 compiler contain redundant information about the PL/0 syntax. These include the input to `scangen` and `fmq` (`scangen.in` and `fmq.in`, respec-

[1]Note that `fmq` is referred to as `LLgen` in the book. In the standard tool distribution, `llgen` is a version of `fmq` without error repair.

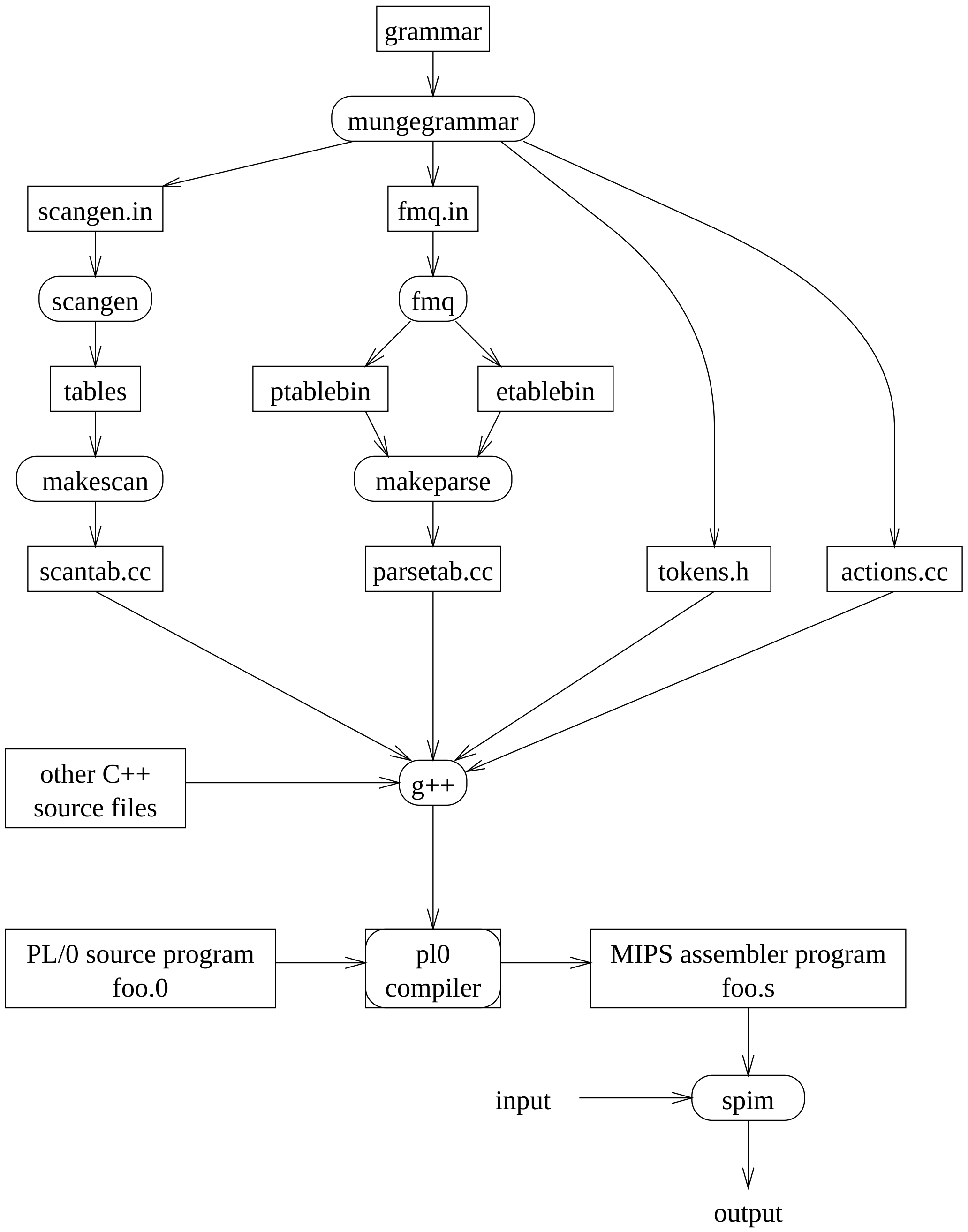

Figure 1: Construction and operation of the PL/O compiler.

tively), semantic action routine numbers (switch statement labels in `actions.cc`), and declarations of token numbers used in semantic analysis (`tokens.h`). To free the programmer from the need to keep these files mutually consistent, they are all generated automatically by a preprocessor named `mungegrammar`. The input to `mungegrammar` is roughly a merger of the input expected by `scangen` and `fmq`, with some syntactic changes necessary to make the merger work and to simplify the specification of long productions and action routines. The `grammar` file distributed with the PL/0 compiler contains extensive self-descriptive comments. Read them carefully before making any changes. Principal features include the following:

- Tokens are declared in a single list that includes alphabetic token names, optional character string images (for error messages), insertion and deletion costs (for syntax error repair), and lexical definitions (regular expressions). A token may have *variants*, which will all appear the same to the parser, but may be distinguished by semantic action routines. Variants are output as tokens with matching major token numbers in `scangen.in`. Note that the semicolon that terminates a token definition must be the last thing on its input line (other than white space or a comment).

- Two special token names *must* appear: `IDENT` and `SPACE`. Reserve words must match the lexical definition of `IDENT`, but appear in the token list with a double-quoted character string definition, rather than a regular expression. `SPACE` is the only token recognized by the scanner but not passed to the parser.

- Semantic action routines can be embedded directly in productions, delimited by double square brackets (`[[...]]`). The code in `actions.cc` will look better if you leave the square brackets on lines by themselves, but this is not required.

- Second and subsequent lines of multi-line productions do not begin with ellipsis (`...`), as they do in `fmq.in`. First lines of productions, however, must begin in column 1, and second and subsequent lines must not do so.

- Ada-style (double dash through end-of-line) comments are allowed anywhere in the `grammar` file, except within semantic action routines.

In addition to generating `scangen.in` and `fmq.in`, `mungegrammar` collects action routines into the arms of a large switch statement, which it then embeds in a C++ function named `do_action` in file `actions.cc`. The parser calls `do_action` when it reaches the appropriate point in the right-hand side of a production. Finally, `mungegrammar` generates file `tokens.h`, containing a definition (`#define`) for every major and minor token number. Major token numbers distinguish non-terminals in the context-free grammar understood by the parser. Minor token numbers distinguish token variants. For example, in PL/0 the multiplication and division operators play identical roles in the grammar, and are therefore defined as variants of a single token named MULOP. `Tokens.h` defines the symbol `MAJ_MULOP` to contain the major token number for MULOP. The two variants of MULOP are named TIMES and SLASH, with minor token numbers `MIN_TIMES` and `MIN_SLASH`, respectively in `tokens.h`. The character string image for MULOP is "`*`", which the syntax error repair routines will use in error messages when they insert a MULOP. If an image is not provided, the repair routines will use the token name.

With the exceptions noted above, the token definitions (regular expressions) and productions in the `grammar` file conform to the syntax expected by `scangen` and `fmq`. Note that concatenation within tokens is indicated with a dot (`.`) rather than juxtaposition; alternation within tokens is indicated with a comma (`,`) rather than a vertical bar (the vertical bar is used to separate to token variants).

Two potentially confusing features are the `NOT()` operator and the `{TOSS}` annotation in regular expressions. `NOT()` generates the complement (with respect to the entire character alphabet) of the (comma-separated) list of characters inside the parentheses. `{TOSS}` indicates that the preceding character or `NOT()` expression will not be important to the rest of the compiler, and may be discarded by the scanner. In general, characters of comments, whitespace, and tokens that represent only a single possible string (e.g. `:=`) can safely be tossed; characters of tokens for which there are many possible strings (e.g. `IDENT` or `NUMBER`) should not be tossed. If you want you can keep some characters and toss the rest.

Please note that `mungegrammar` is currently implemented as a simple-minded script. Syntactic errors in the `grammar` file will usually pass through `mungegrammar` undetected, leading to error messages from `scangen` or `fmq`. If this happens you may need to read the `scangen.in` or `fmq.in` files to figure out what's going on. Remember, though, to make any necessary changes in the `grammar` file, not in the files created from it.

2.3 RCS

The PL/0 distribution uses RCS (the Revision Control System, developed by Walter Tichy of Purdue University) for version control. As you make changes over time you should "check them in" to RCS, which will keep track of all the old versions of all your files, in an organized and space-efficient way. Read the man pages for `ci`, `co`, `rcs`, `rlog`, and `rcsdiff`, in that order.

2.4 The `Makefile`

Creation of the PL/0 compiler is automated by a `Makefile`. You should read the `Makefile` and figure out how it works. You should also learn to use the extra rules it provides, in addition to `make pl0`. In particular, you will want to use the following:

`make depend` The `Makefile` "includes" an auxiliary file named `makefile.dep`, which incorporates knowledge of which files `#include` which others. If you add or alter inclusions anywhere in the compiler, you will want to re-run `make depend`, which rebuilds `makefile.dep`.

`make tags` This rule creates cross-reference indices (files `tags` and `TAGS`), which are used by editors like `vi` and `emacs` to assist in source perusal. Typing the appropriate command to the editor will cause the cursor to move to the file and line at which the identifier is declared. In `vi`, the command is `:ta tag_name`, or control-`]` when the cursor is positioned on the identifier. In `emacs`, the command is meta-`x` `find-tag` *tag-name*, or meta-`.` *tag-name*. If the cursor is on or near an identifier, `emacs` will supply it as a default when prompting for *tag-name*. To pop back to the previous tag in `vi`, type control-`t`. To pop back in `emacs`, type meta-`-` meta-`x` `find-tag`, or meta-`-` meta-`.`. Note that the algorithm used to build the tag cross-reference database is heuristic; tags work most of the time but not always. `Vi` keeps track of a single location for each tag. `Emacs` keeps track of several possibilities, and jumps to the most "likely" one first. If you don't like that one and want to try the next, type control-`u` meta-`.`.

In addition to creating cross-references for identifiers, `make tags` creates cross-references for action routines, grammar symbols, and production numbers. Tag search for a grammar symbol moves the cursor to a production in the grammar in which that symbol appears on the left hand side. Tag search for `R37` (and similarly `R`*n* for any appropriate *n*) moves to the 37th (*n*th) action routine. This is useful for finding syntax errors in action routines, since the C++ compiler produces error messages for the file `actions.cc`, not for `grammar`. Tag search for `P37` (in general P*n*) moves to the 37th (*n*th) production in the grammar. You may want to try this after looking at a parse trace, generated by `pl0 -Dparse` or `pl0 -Pparse`.

`make sources` This rule simply prints the names of all the source files required to build the compiler. It is useful when embedded in backward quotes in the argument lists of other programs. For example,

```
grep foobar ‘make sources‘
```

will find all occurrences of `foobar` in the source of the compiler. Similarly,

```
ls -l ‘make sources‘ | grep "^-rw"
```

will list all source files that are currently writable. If you use RCS with strict locking, these will be the files that you have checked out to make modifications. If you don't want the long version of the listing,

```
ls -l ‘make sources‘ | grep "^-rw" | sed -e "s/.*:.. //"
```

will print just the names of the checked-out files. You may want to create aliases for these, e.g. `llrw` and `lsrw`. I frequently type `rcsdiff ‘lsrw‘` to find out what changes I've made recently, and `ci -u ‘lsrw‘` to check them all in.

3 Creating and Running Your Copy of the Compiler

The root directory of the PL/0 distribution contains a copy of the source for the PL/0 compiler, together with a `Makefile`, a `README` file, and a `NEWUSER` script. The `README` file explains how to use the `NEWUSER` script. This script will copy the source, create an RCS archive of it, run `make` to build the compiler, and create a test directory.

The `Makefile` places the final PL/0 compiler in a file named `pl0` . The compiler reads a PL/0 source program from standard input or from a file named in its command-line arguments. In the latter case, the name must end with "`.0`". The compiler prints error messages and diagnostics to standard output (fatal errors to standard error), and writes MIPS I assembler target code to a file whose name is created by replacing the "`.0`" in the source file name with "`.s`" or, if compiling standard input, to a file named `plzero.s` .

The assembler output is intended for execution by James Larus's MIPS interpreter, `spim`, from the University of Wisconsin. To compile and run a PL/0 program under `spim`, type:

```
pl0 foo.0
spim -file foo.s
```

You might want to create the following shell alias:

```
alias run 'spim -file \!^.s | tail +6'
```

Then you can type

```
pl0 foo.0
run foo
```

Note of course, that `pl0` must be on your `PATH`. If it's in the current directory (or if you're in the `test` directory with a symlink), you can type `./pl0`.

The pl0 compiler accepts several additional arguments:

`-C` Include PL/0 source lines in the target code as comments. This makes it easier for you to read and understand the output.

`-v` Print summary statistics at end of compilation (verbose).

`-Dscan` Dump all tokens to standard output, as they are recognized by the scanner.

`-Dparse` Dump a trace of parser activity to standard output: predictions, matches, and calls to action routines. `-Dscan` and `-Dparse` are mutually exclusive.

`-Dast` Dump the abstract syntax tree to standard output immediately after parsing.

`-Dsymtab` Dump the contents of the symbol table to standard output at the end of compilation.

`-P`*phase* Stop compilation after a given intermediate phase, where *phase* is one of `scan` (scanning), `parse` (parsing), `ast` (abstract syntax tree construction), or `seman` (semantic analysis). `-Pscan` implies `-Dscan`. `-Pparse` implies `-Dparse`. `-Past` implies `-Dast`. `-Pseman` implies `-Dsymtab`.

4 PL/0 Compilation Phases

The PL/0 compiler consists of about 5,400 lines of heavily-commented C++, plus automatically-generated scanner and parser tables, and 400 lines of grammar source. Constants in the code are printed entirely in capital letters, with sub-words separated by underscores, e.g. `SCRATCH_REG`. Most other identifiers are printed entirely in lower case, with sub-words separated by underscores. User-defined type and class names end with `_t`, e.g. `token_attrib_t`.

4.1 Source Buffering and Diagnostic Messages

Files `inpbuf.[h,cc]`

The compiler reads its source on demand, and keeps it all buffered in a linked list within an object of class `input_buffer_t`. Together with each line, the input buffer keeps a list of diagnostic messages associated with that line. Buffering allows messages to be generated out of order, but printed in order. The later phases of the compiler keep track, for each syntactic construct, of the location (line and column) at which that construct began. Messages pertaining to that construct can then be issued at the appropriate location.

Each diagnostic has a key letter in column 2 that indicates what kind of a message it is.

- `S` means syntax repair.
- `W` means warning.
- `E` means semantic error.

Semantic errors inhibit code generation; syntax repairs and warnings do not.

4.2 Scanner

Files `scantab.[h,cc]`, `scanner.[h,cc]`

The scanner is a straightforward finite automaton. It keeps a copy of tokens whose characters are significant (were not `{TOSS}`ed in the `grammar` file). Its main entry point is the function `token_get`, called by the parser. The only tricky part of the scanner is the mechanism for backtracking, employed when one token in the language is a prefix of another, but more than one character of lookahead is required to distinguish between the two. (For example, consider `3.1459` and `3..14` in Pascal, or `DO5I=1,10` and `DO5I=1.10` in Fortran.) No such tokens occur in PL/0, but they could easily be added, and the scanner will support them.

If it encounters a mal-formed token, the scanner issues a warning message and skips ahead to the next point at which it can recognize a token.

4.3 Parser and Syntax Error Repair Routines

Files `parsetab.[h,cc]`, `parser.[h,cc]`

The parser is a straightforward LL(1) pushdown automaton, augmented with error repair routines and with markers to support action routines and automatic management of space for attributes (see below). The error repair routines employ a locally least cost technique due to Fischer, Milton, and Quiring. Basically, they calculate the "cheapest" set of insertions and/or deletions that will allow the parser to consume one more token of real input. Insertion and deletion costs are specified in the `grammar` file. The choice among alternative repairs can be tuned by changing the relative costs of various insertions and deletions. Generally speaking, it should be easy to insert or delete tokens that a programmer is likely to forget or to type by mistake, and hard to insert or delete tokens that require matching constructs later in the program, since incorrect repairs of this type are guaranteed to lead to future errors. When defining new tokens, use the existing insertion and deletion costs as a guide. The compiler will work correctly whatever costs you give it, but it will do a better job of avoiding spurious, cascading errors if the costs are reasonable.

4.4 Syntax Tree Construction

Files `attributes.[h,cc]`

The semantic action routines in the `grammar` file do one thing only: construct an abstract syntax tree. They perform no symbol table manipulations, nor do they check for any error conditions. Space for attributes is maintained in a so-called "attribute stack", which is pushed and popped automatically at appropriate times in the parse. At any particular point in time, the attribute stack contains a vertical slice of the parse tree: one record for each symbol in each production between the current node and the root. The full parse tree is never in memory at once. When the parser predicts a production, it calls into the attribute manager to push records onto the attribute stack for all the symbols on the right-hand side (RHS). It also pushes a marker onto the parse stack. The marker allows it to tell when it has popped all the RHS symbols off the parse stack, at which point it calls into the attribute manager to pop the RHS symbols off the attribute stack.

Each element of the attribute stack may contain the synthetic attributes of a token from the scanner, a pointer to a syntax tree node, a list of syntax tree nodes, and/or a source code location (or, often, nothing at all). The purpose of the action routines in the grammar is to pull the token information out of the attribute stack, construct appropriate pieces of the syntax tree, and hang them off of records in the attribute stack that will be available to routines later in the parse (higher in the parse tree). When parsing is completed, the root of the complete syntax tree is found in the attribute record of the goal symbol, the last remaining record on the attribute stack.

4.5 Semantic Checks

File `semantics.cc`

The semantic checker enters identifiers into the (scoped) symbol table, and performs all static semantic checks. For PL/0, the only checks are the following:

- is every identifier declared before use?
- is no identifier declared more than once in the same scope?
- is every left-hand side of an assignment a variable?
- is every identifier in an expression a variable or constant?
- is every called identifier a subroutine?
- are strings assigned only into `output`?
- does every subroutine call have the right number of arguments?
- is every argument passed as an `out` or `in out` argument an l-value (i.e. a variable or a parameter)?

In addition, the PL/0 compiler performs the following “sanity checks”, and issues warnings (not error messages) if they fail:

- is every variable both written and read?
- is every constant read?
- is every subroutine called?
- is the program non-empty?

There is no separate `semantics.h` file. File `semantics.cc` contains nothing but the `check_semantics` methods of descendants of `syntax_tree_t`, as declared in `attributes.h`.

4.6 Code Generation

Files codegen.[h,cc], mips.[h,cc]

There are two sub-phases to code generation. The first assigns frame pointer offsets for local variables, and allocates registers; the second generates code. Register allocation is extremely naive: the registers of the target machine not reserved for any other purpose are used as a LIFO evaluation stack. Most programs use nowhere near the full register set. Any program requiring more than the available number of registers produces a fatal compiler error. A good compiler (with a code improver and a smart register allocator) would keep many variables in registers much of the time, spilling them to memory only when absolutely necessary.

For the purposes of code generation, additional fields are required in both the nodes of the syntax tree and the records of the symbol table. These fields are maintained in separate records so that they need not be visible to the early phases. They are declared in codegen.h.

File codegen.cc, at just over 1000 lines, is the largest in the PL/0 compiler. It contains definitions for

- the allocate_registers and generate_code methods of descendants of syntax_tree_t; and
- the allocate_space and generate_data methods of descendants of symbol_entry_t, as declared in symtab.h.

The routines in mips.[h,cc] encapsulate the MIPS assembler syntax by overloading the function mips_op.

4.7 Symbol Table

Files symtab.[h,cc]

The symbol table has a wide interface (many methods in scope_t and symbol_entry_t), but a simple internal structure. There is a single main hash table, keyed by {name, scope} pairs, and a stack of open scopes. To look up an element, the symbol table routines traverse the open scope stack from top to bottom, probing the hash table at each level to see if the desired name was declared in that scope. Faster schemes are possible, but this one is simple, works well, and is not outlandishly slow.

5 Compiler Classes

5.1 Syntax Tree Nodes

Every node in the attributed syntax tree is an object belonging to one of the concrete subtypes of the abstract type syntax_tree_t. Figure 2 shows the hierarchy of PL/0 syntax tree classes.

Each subclass of syntax_tree_t must implement the following methods:

- constructor (in attributes.cc)
- check_semantics (in semantics.cc)
  Checks the semantics of the node and of its children, if any.

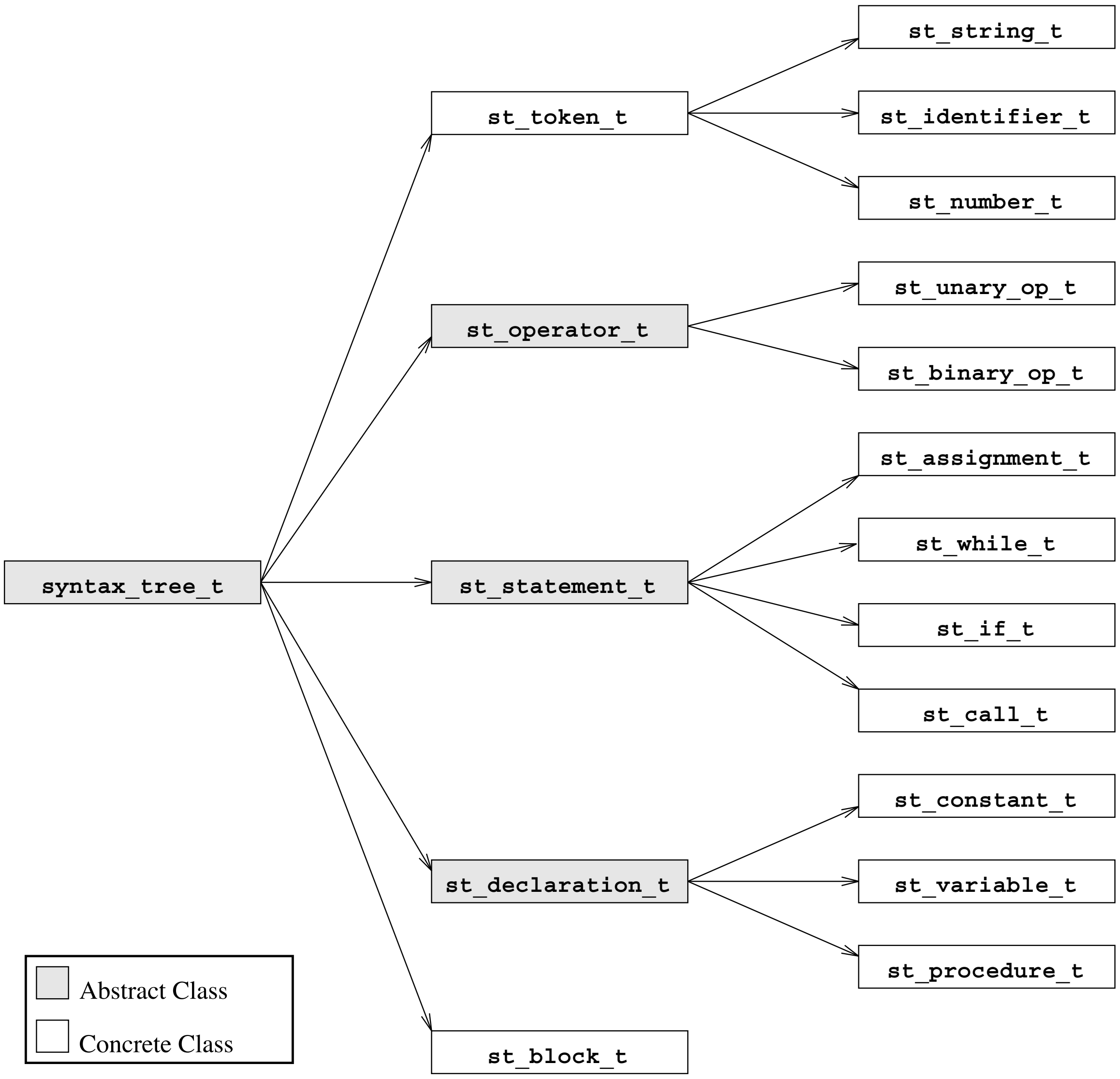

Figure 2: Syntax tree classes.

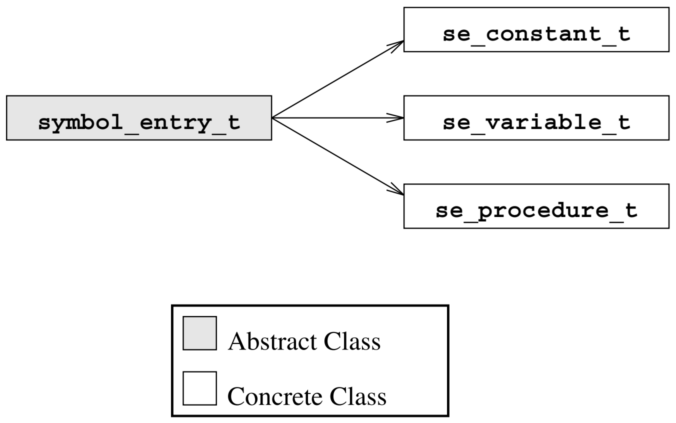

Figure 3: Symbol table classes.

- `allocate_registers` (in `codegen.cc`)
  Allocates any registers used by the node and lets its children also allocate their needed registers. Returns a pointer to its `syntax_code_t` structure, which details register usage and indicates where the node's parent can locate any results generated by the node.

- `generate_code` (in `codegen.cc`)
  Generates any assembly code needed to implement the node and its children, and dumps it to the output file using the `mips_op` functions.

- `debug_dump` (in `attributes.cc`)
  Dumps all useful debugging information about a syntax tree node to `stdout`.

Several subclasses implement additional functions, mostly accessors (`get_`*field* and `put_`*field*). Perhaps the trickiest routine is `st_identifier_t::generate_lcode`, which generates code to store a value to a variable, load a variable's address into a register, or write a value to `OUTPUT`.

No destructor is needed for `syntax_tree_t`; once created, syntax tree nodes remain allocated through the end of the compiler run.

5.2 Symbol Table Entries

All entries in the symbol table must be members of a concrete type derived from the abstract type `symbol_entry_t`. Figure 3 shows the PL/0 symbol table entry classes.

Each subclass of `symbol_entry_t` must implement the following methods:

- constructor (in `symtab.cc`)

- `allocate_space` (in `codegen.cc`)
  Allocates any space either in the stack frame or globally which will be needed to represent the symbol in the compiled program.

- `generate_data` (in `codegen.cc`)
  Outputs any pseudo-symbols or other code for creating the symbol in the output assembly file. For example, global variables output the pseudo-opcode "`.dword 0`".

- `debug_dump` (in `symtab.cc`)
  Dumps all useful debugging information about a symbol table entry to `stdout`.

In addition, the `symbol_entry_t` base class has a `generate_code` method (in `codegen.cc`) that is called from various places to generates the code to access a symbol.

No destructor is needed for `symbol_entry_t`; once created, symbol table entries remain allocated through the end of the compiler run.

Acknowledgments

Michael Scott wrote the original version of the PL/0 compiler (and the `makeparse` and `makescan` tools) in Pascal in the summer of 1992. He borrowed some of the code from his compiler for the Lynx programming language, which in turn contained pieces of his undergraduate compiler course project from the University of Wisconsin. He based the general structure of the parser and error repair routines on code written by Jon Mauney to accompany the distribution of `fmq`. In the fall of 1993 Michał Cierniak replaced the original code generator (which produced a subset of C) with code to generate MIPS assembly language. Galen Hunt re-wrote the compiler in C++ in the fall of 1994, introducing the `symbol_entry_t` and `syntax_tree_t` class hierarchies, substantially simplifying code generation via overloaded functions, and introducing a variety of other improvements. Michael Scott created the `mungegrammar` tool in the summer of 1995, replacing an earlier pre-processor that did not produce `tokens.h` or `scangen.in`. Additional changes in the summers of 2000, 2002, and 2004 introduced namespaces for the various compiler modules, improved debugging output, and converted most of the code (pretty much everything other than the parser) to good object-oriented form. Thanos Papathanasiou added value/result parameters in the fall of 2004, and extracted the pedagogical subset. Bijun He built the x86 code generator.