CISC 458
Tutorial 4

Phase II

Revised by Ahmed Harby from Previous Versions

Thursday, February 9, 2023

Tracing the Parser

New `ssltrace` Command

```
ssltrace "ptc -o2 -t2 -L lib/pt test.pt"
  lib/pt/parser.def -e
```

- `-t2` trace the output of the parser
- `-o2` execute only parser and scanner
- `-L` is the direct path to your library
- `lib/pt/parser.def` is where `ssltrace` should look for the parser tokens
- `-e` gives the trace of emitted tokens.

Compiling the Parser

- Compile with “`make`” in the `ptsrc/parser/` directory and not “`make scanner`” or “`make`” in the `ptsrc/` directory
  - `make scanner` in the `ptsrc/` directory **will disable** the parser!
  - `make` in the `ptsrc/` directory **will require** a complete `Quby` compiler!
- **Note**: Don’t run `make parser` in the `ptsrc/` directory; the `MakeFile` in that directory does not recognize it.

Updating the Tokens

Overview

- Most of the changes will be in `parser.ssl`
  - Contains rules to transform parser tokens (‘p’ prefix - e.g. `pExternal`) to semantic tokens (‘s’ prefix - e.g. `sBegin`)
- File Dependencies
  1. Update parser tokens (`pToken`) in `parser.ssl` with those in `scan.ssl` (or `stdIdentifiers`)
  2. Update semantic tokens (`sToken`) in `parser.pt` with those in `parser.def`
- **Note**: Order of tokens in the `.ssl` files must match! This is a common source of errors.

Updating Files with Dependencies

1. Updating parser tokens: update `parser.ssl` when `scan.ssl` (or `stdIdentifiers`) changes
    - For `parser.ssl`, parser tokens are in the `Input` section
    - For `scan.ssl`, parser tokens in the `Output` section
    - Be careful when copying and pasting!
2. Updating semantic tokens: update `parser.pt` when `parser.def` changes
    - Compiling `parser.ssl` is very similar to how `scan.ssl` was compiled in Phase 1
    - Refer to slides 3-4 in Tutorial 2 (and later slides which use `make scanner` instead of `ssl`)

Updating `parser.pt` with `parser.def`

- After compiling `parser.ssl`, definitions generated in `parser.def` need to be consistent with those in `parser.pt`
- Replace contents between the following paraphrased comments with the contents of your new `parser.def`

```
{ === Pasted contents of parser.def, ... }
... content ...
{ === End of contents of parser.def }
```

Final Reminders

- Remember to recompile (using `make`) after making changes to files mentioned earlier!
- Do not change `parser.def` or `parser.sst`! They are generated by the `ssl` command which is executed when recompiling!
- **Important**: Values of tokens in `parser.def`, `scan.def`, and `parser.pt` must match!
    - Look for sections with following comments in `parser.pt`

```
{ === Pasted contents ... }
... content ...
{ === End of contents of ... }
```

Updating the Rules - Programs, Modules, and Routines

Overview

- Most of the changes will be in `parser.ssl`.
- Change the rules in `parser.ssl` to handle the changes in language structure.
    - From `PT Pascal` to `Quby`
- The Assignment 2 handout discusses the changes in quite a lot of details and provides a step-by-step guide.
    - These slides are heavily influenced by that handout, so in the case of disagreement between slides and handout, handouts act as the standard.
- The changes are more numerous and more complicated than Phase 1. However, there is far less of a learning curve to work through this time.

General Strategy

- Observe output tokens from `PT Pascal`
- Preserve token streams as much as possible to avoid future changes in the semantic phase
  - Do more work now to do less work later
- Example:
  - In `PT Pascal`, there is a need for "`begin ... end`" keywords, which output `sBegin` and `sEnd` tokens
  - In `Quby`, there is no need for "`begin ... end`"
  - However, output should still have `sBegin` and `sEnd` tokens!

Token Definitions

- Parser Tokens (e.g. `pElse`)
    - Remove those taken out in Phase 1
    - Add the new ones from Phase 1
    - Order of tokens **MUST** match between dependent files
- Semantic Tokens (e.g. `sBegin`)
    - Add new semantic tokens (e.g. `sModule`)
    - Remove unused ones (e.g. `sRepeat`)

Programs

- Unlike PT Pascal, `Quby` makes no distinction between declaration and statements.
  - Modify The `Block` rule to accept a sequence of any number of declaration **or** statements **in any order**
- In order to make the differences in `Quby` less visible to the semantic phase (Phase 3), always output a sequence of statements and declarations as if it were a `PT Pascal begin` statement
  - `sBegin` before the statements and `sEnd` after them
- **Hint**: Merge the alternatives of the `Statement` and `Block` rule.

Program Declarations - Constants

- Modify the parsing of constant declarations to meet the `Quby` language spec. The output token streams for these declarations should be the same as for the equivalent PT declarations, in order to minimize the changes we'll have to make to the semantic phase.
  - For example, the Quby declarations:

```
val c = 27
```

  - Should yield the parser output token stream:

```
sConst
 sIdentifier <identifier index for 'c'>
 sInteger 27
```

Program Declarations - Variable

- Quby uses an variable to specify a variable's type. e.g,

```
var v : string
```

- Should yield the parser output token stream:

```
sVar
sIdentifier <identifier index for 'v'>
sIdentifier <identifier index for 'string'>
```

Program Declarations - Variable Cont'd

- To assist the semantic phase, if there is more than one identifier declared in a single Quby var declaration, then each one should be output with an sVar token,
  - For example: var a, b, c : integer
  - Should yield the parser output token stream:

```
sVar
sIdentifier <identifier index for 'a'>
sVar
sIdentifier <identifier index for 'b'>
sVar
sIdentifier <identifier index for 'c'>
sIdentifier <identifier index for 'integer'>
```

Program Declarations - Type

- Quby uses an initial value to specify a variable's type. e.g,

```
type t : integer
```

- Should yield the parser output token stream:

```
sType
sIdentifier <identifier index for 't'>
sIdentifier <identifier index for 'integer'>
```

- PT's `TypeDefinition`, `TypeBody`, and `SimpleType` are not needed anymore
- Create a new rule `ValueOrQuby` and call it instead of `TypeBody`
    - Accepts either a constant value or a `quby` clause

Routines (Procedures/Functions)

- **Main Goal**: The output token stream for the function should be the same as for the equivalent PT procedure, in order to minimize the changes we'll have to make to the semantic phase.
  - Statements are surrounded by `sBegin` and `sEnd` tokens
  - **Hint**: The `Block` rule comes in handy!

```
def P ()
   DeclarationsAndStatements
 end
```

  - Should yield the parser output token stream:

```
sProcedure
sIdentifier <identifier index for 'P'>
sParmEnd
sBegin
DeclarationsAndStatements
```

Modules

- Similar to an entire `Quby` program without the `using` clause
- **Main Goal**: Keep the output of the `Quby` parser similar to the output of the `PT Pascal` parser
  - Statements are surrounded by `sBegin` and `sEnd` tokens
  - **Hint**: The `Block` rule comes in handy!

```
module M is
  DeclarationsAndStatements
end;
```

```
sModule
 sIdentifier <identifier index for 'M'>
 sBegin
 DeclarationsAndStatements
 sEnd
```

Updating the Rules - Statements and Clauses

Statement Sequences

- In `Quby`, any sequence of declarations and statements can appear in eachalternative, not just one statement. So how do we keep the output token stream for case alternatives the same as it was in PT?
    - **Hint**: re-use your Block rule, which will once again emit an sBegin semantic token at the beginning of an alternative, and an sEnd at the end.

- The `Quby` else clause on case statements is new, and we must handle it specially. But what we will do is simple - just check for an else following the alternatives in the case statement, and output sElse followed by the statements of the else clause (enclosing the body of the else clause with sBegin .. sEnd as usual) before outputting the sCaseEnd semantic token

Statements

- **Main Goal**: Keep the output of the `Quby` parser similar to the output of the `PT Pascal` parser

What? Meet the `Quby` language specifications for the following `Quby` statements

- `if`
- `unless`
- `case`
- `do`

How? Modify the parsing (rule mechanisms) of the following statements

- `if`
- `case`
- `while`
- `repeat`
- `begin`

`elseif` **Clauses**

- Two ways to implement `elseif`:
  - Create a new `sElseIf` token to represent `elseif` and let the semantic phase deal with it;

    OR

  - Output `PT Pascal` nested `if` statements so that the semantic phase can understand it
- Either change now, or leave until the semantic phase, up to you (and your group). Just make sure you document your choice.
- You will be using your choice in this phase when implementing the semantic phase!

`elseif` **Options**

First option (Handle at semantic phase):

```
if Expression1 then{
    DeclarationsAndStatements1
} elseif Expression2 then {
    DeclarationsAndStatements2
} else {
    DeclarationsAndStatements3
}
```

Second option (Handle at parser phase):

```
if Expression1 then{
    DeclarationsAndStatements1
} else {
    if Expression2 then{
        DeclarationsAndStatements2
    } else {
        DeclarationsAndStatements3
    }
}
```

`Case` Statements

- Should produce the same output stream as `PT Pascal`'s case statements, along with the `Quby`'s default alternative if present
- `Quby` allows for any sequence of declarations and statements in each alternative, while `PT Pascal` allows only one.
  - **Hint**: The `Block` rule comes in handy (again)!
- `Quby` allows for an optional default alternative `else`
  - First process all the alternatives and emit the `sCaseEnd` token
  - Then Check for `else` and, if found, emit an `sCaseElse` token followed by its *DeclarationsAndStatements*

Case Statements Example

```
Input: case i
           when 42 then
             DeclarationsAndStatements1
           else
             DeclarationsAndStatements2
      end
```

```
Output: sCaseStmt
     sIdentifier <identifier index for i>
     sExpnEnd
     sInteger 42
     sBegin
     DeclarationsAndStatements1
     sEnd
     sElse
     sBegin
     DeclarationsAndStatements2
     sEnd
     sCaseEnd
```

`Do` Statements

- Modify handling of `PT Pascal`'s `repeat` statements to use the new `Quby` syntax
    - The `Quby` `Do` statement is similar to PT's `while` statement
    - The `Quby` `Do` statement is similar to PT's `repeat` statement
- **Note**:In `Quby`, we can't pull the same trick of making the Semantic phase think we still have PT for the do statement - there is no PT while or repeat statement equivalent to a Quby do loop, so we'll just have to leave it until the Semantic phase.
    - **Hint**: simply emit `sNot` following the conditional expression!

Syntactic Details

- Watch out for other minor syntactic differences between PT Pascal and `Quby` - for example, the assignment operator is := in PT but = in Quby, the equality operator is = in PT but == in `Quby`, the inequality operator is <> in PT but != in `Quby`, and the not operator is not in PT but ! in `Quby`.

String Type

- Remove handling for `PT Pascal`'s `char` data type and `char` literals
- Add handling for the `string` data type and `string` literals
- Add handling for the new string operators
  - The precedence of the operator `$` (including the .. portion) should be higher (more tightly binding) than `*`, div and mod and lower than not, the precedence of the ? operator is the same as that of `*`, div and mod, and the precedence of is the same as the precedence of not.
  - **Hint**: Adding ? to be the same precedence as `*`, div and mod, and to be the same precedence as not, is straight forward. Adding `$` involves introducing a new precedence level.
- All new operators should be postfix in their output stream
  - Semantic token for the operator should follow those of the operands

Substring Example

- Input Stream: "Hello" $ 1 : 2
- Output Token Strem:

```
sLiteral "Hello"
sInteger 1
sInteger 2
sSubstring
```

Final Notes

Suggestions and Reminders

- Start small. Changing the parser and semantic tokens are a good example.
- Build minimal programs around each requirement to guide and test your changes.
- Preserve the `PT Pascal` output as much as possible to save your group trouble in the next phase.
- Make sure all of the files have tokens in the correct order!

Suggestions and Reminders

- Do one feature at a time. Not all features needs to be implemneted in order to test.
    - If a test program doesn't have a loop, then loops don't have to be implemented yet to run tests.
- Work order is semilinear. Try not to split the phase up and have each member work on a separate section.
    - Highly recommend meeting up to at least work on common changes.
    - e.g. Accepting programs / updating tokens

Next Tutorial

- Phase 2 Submission Details