Relatório CP1 — Grupo DOPE

O desenvolvimento do parser seguiu correndo bem, até o fatídico momento em que tentamos definir e resolver os conflitos das regras referentes a declarações, que incluem uma regra do tipo:

```
declaration :
      declaration-specifiers
    | declaration-specifiers init-declarator-list
```

isto é, uma lista de especificadores de tipo, qualificadores, etc, seguida de uma lista de - simplificando - identificadores sendo declarados, que é opcional.

Ocorre que identificadores em C podem ser associados a um tipo (via `typedef`) e assim podem ser utilizados como um especificador de tipo. Isso gera o problema de determinar em qual token de identificador deixamos de tratar um especificador de tipo e passamos a tratar um nome a ser declarado (de variável etc). Pesquisando sobre o problema chegamos à decepcionante descoberta de que o problema é ainda pior e a gramática de C contem uma ambiguidade um tanto inconveniente (link) (link).

```
T(a);
```

O código acima pode significar a chamada da função de nome `T` com argumento `a`, mas também pode ser uma declaração válida da variável `a` com tipo `T`.

Para resolver essa ambiguidade precisamos alimentar o lexer com informação da tabela de símbolos para podermos distinguir os identificadores que foram anteriormente declarados como tipos. Para isso escrevemos uma implementação mínima de tabela de símbolos com apenas as operações de abrir e fechar escopo e de adicionar e verificar se uma string é um nome de tipo.

Versões do Bisão

Tivemos um problema que surgiu devido a versões diferentes do Bison sendo usadas no desenvolvimento. Como é possível ver no changelog do Bison, recentemente (aparentemente na versão 3.6) foi adicionado um *alias* para o tipo que deve ser retornado pelo scanner:

```
typedef enum yytokentype yytoken_kind_t;
```

Para que esse nome (`yytoken_kind_t`) possa ser utilizado normalmente com uma versão anterior do Bison, adicionamos essa declaração no cabeçalho `parsing.h`. Também explicitamos no arquivo `README.md` quais versões das ferramentas foram utilizadas para desenvolvimento e testes.

Exceções

Devemos pontuar algumas opções que fizemos na implementação: escolhemos tratar os comandos `break` e `continue` na próxima etapa da implementação

(analisador semântico), bem como o tratamento de literais de string no tocante a caracteres especiais como o `\n`.

Testes

Decidimos compilar um executável separado que apenas imprime os tokens identificados pelo scanner para possibilitar a inspeção desse componente. Isso é feito passando um parâmetro para o compilador para definir um nome de macro (`-D DUMP_TOKENS`) que faz com que sejam condicionalmente definidas macros alternativas para o tratamento dos tokens, i.e. que apenas imprimem os tokens.

Os testes são feitos executando-se esse binário, assim como binário do parser de fato, com os arquivos de entrada de exemplo contidos em `tests/input/` e armazenando-se as saídas em `tests/tokens/` e `tests/output/`, respectivamente. Esses arquivos são então comparados usando `diff` contra os arquivos nas pastas correspondentes com o sufixo `-expected`, que contém as saídas experadas para cada caso de teste.

Suporte CRLF

Os caracteres de controle que representam uma quebra de linha em arquivos de texto podem ser diferente dependendo do SO. Esses caracteres são CR (*carriage return*) e LF (*line feed*), respectivamente `0x0D` (representado por `\n`) e `0x0A` (representado por `\r`). Sistemas Unix utilizam a terminação `LF`, enquanto o Windows utiliza dois caracteres: `CR LF`. O antigo MacOS (pré-OSX Macintosh) utilizava apenas `CR`, porém escolhemos não dar suporte a essa formatação em nosso scanner por ser extremamente legado.

Para a maioria dos casos, não é necessário explicitar o caso `\r\n` como quebra de linha, já que o caractere `\r` está representado no *pattern* `.`, que corresponde a qualquer byte exceto `\n` (`0x0A`), desta forma, só precisamos explicitar o caso CRLF nos comentários de linha (marcados por `//`) que possuem quebra de linha.